2014 International Conference on Future Internet of Things and Cloud

Improving I/O Performance in Smart TVs

Cheolhee Lee
Department of Computer and Software
Hanyang University
Seoul 133-791, Korea
Email: lch6719@hanyang.ac.kr

Taeho Hwang
Department of Computer and Software
Hanyang University
Seoul 133-791, Korea
Email: htaeh@hanyang.ac.kr

Youjip Won
Department of Computer and Software
Hanyang University
Seoul 133-791, Korea
Email : youjip.won@gmail.com

***Abstract*—To use the XML file, it must be converted into a tree structure form in smart TV application. However, when the application is terminated, tree are eliminated in process address space. When application is restarted, XML file need to be converted to tree again in order to execute application. This study presents a Fast I/O technique that enables restarting a application without data conversion process by adding persistency in tree in a smart TV environment. Fast I/O technique provides an object, in which the tree are saved adding persistency in process address space. The data structure gains persistency by saving the tree in an object and reusing it without data conversion when restarting the application. Fast I/O technique was applied in the web browser to parse HTML and skip the process of composing a tree. Running time was reduced up to 61% in the test environment, consisting of CPU, memory, and an SSD.**

***Keywords*-Smart Tv; Web Browser; XML File; DOM Tree; Persistency; mmap() System Call**

I. Introduction

Modern TV is a computing device. It is loaded with open source middleware, e.g. Android [1], and Tizen [2], or proprietary middleware. It can execute variety of applications including web browser, video player and etc. Most TV application maintains its content in XML format which is stored in the NAND-flash based storage devices. To use XML file the storage device, the application need to converts the XML format contents into in-memory representation. This process is called serialization [3]. This in-memory representation include DOM tree and render tree [4]. For web browser, the nodes of the tree contain information on the web pages location, size, and colors of each elements of the web page [4]. This process is very time-consuming and cpu-intensive task which constitute significant fraction of application start-up latency.

We design Fast I/O technique that adds persistency to memory data that have been dynamically allocated in smart TV. Persistency means that data in space can be reused even after the corresponding application is terminated. By applying this technique, persistency is added to tree in applications using XML form such as web browsers in a smart TV. Because saved tree can be reused even after a process terminates, the process of converting XML file into tree form can be omitted (Figure 1) which can improve the performance of applications.

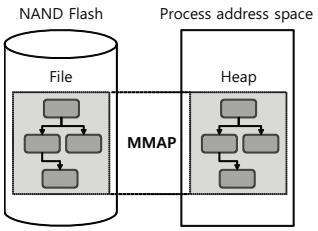

Figure 1. Web Browsing in Prsistent Object

In this work, we develop a set of interfaces to maintain the in-memory representation of XML document in persistent manner and the new type of heap segment, *persistent heap*. We mmap the heap region to NAND storage area so that the in-memory representation of XML object which resides at heap segment becomes persistent. Subsequently, the lifetime of the in-memory representation of the XML object becomes orthogonal to the lifetime of the process which creates it. The new set of interfaces allow the application programmer to manipulate the object in the persistent heap, e.g. open, close, read, write and etc. To support persistency for in-memory data, this study presents objects that dynamically allocate memory region. The object exists in process address space and is mapped to files through mmap() system call. The mapped file is called an object file. The object is not volatile because it is saved in object file through page cache. The object can be allocated a heap region in byte granularity.

Due to the virtual memory mechanism of the modern Operating System, the virtual memory location of a file is determined by the Operating System. The file can be mapped into different locations in the virtual address space in each execution of mmap. The pointers in the tree become invalid if the file which harbors the tree is mapped into different virtual address space.

II. Related Work

Persistency [5] is a concept proposed by Atkinson in 1981. It means 'data requiring system must be maintained until its necessity disappears'. Grasshopper [6] is a single-level store [7] operating system. Fully partitioned address

978-1-4799-4357-9/14 $31.00 © 2014 IEEE
DOI 10.1109/FiCloud.2014.71

space, which is the intermediate step between single address space [8] and private address space [9], is provided. Grasshopper must go through pointer swizzling [10] process to solve pointer validness problems. Therefore, it requires complicated code modification to be applied in applications. SoftPM [11] secures memory data persistency through a container. By saving the root node, all nodes that do not belong in the container are moved to the next container and persistency is added. Therefore, SoftPM automatically adds persistency to memory data connected with the root node. In SoftPM, since page mapping address is not fixed when restoring stored container, swizzling process is required. On the other hand, an object is mapped to the fixed address space in Fast I/O technique and does not incur pointer swizzling overhead. Mnemosyne [12] and NV-Heaps [13] are studies on adding persistency in memory. These provide address space that has persistency and selectively add persistency to data to be reused without conversion process.

This study presents a technique that enable reusing tree without process of conversion and to add persistency in memory of systems that does not incur pointer swizzling overhead.

III. Design

The purpose of this technique is to add persistency to XML file that have been changed to tree in a smart TV and to reuse tree structure without data conversion process. To achieve this, object having persistency is presented and interface that dynamically allocates byte unit memory is provided. The mechanism of this technique is as follows.

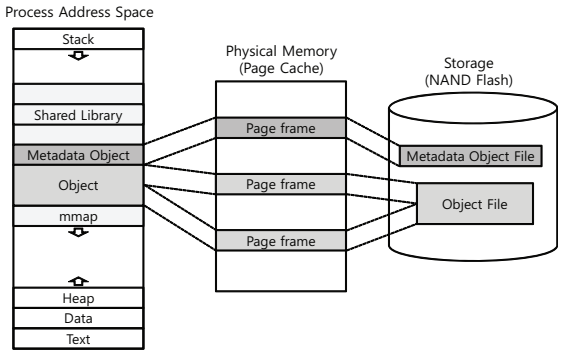

Figure 2. Persistency of Object

A file is created to save memory data. This file is then mapped to process address space through mmap() system call (Figure 2). The address space is called an object and the file that is mapped to the object is called an object file. Byte unit data, called a node, is allocated and removed in the object. By mapping object file in process address space, the object and the data in object can be reused after rerunning the process. Metadata used to manage objects are saved in a special object called metadata object.

A. Design Considerations

1) Pointer validness problems followed by object mapping location: To reuse the data structure saved in an object, object file must be mapped to the same process address space. If the saved object file is mapped to process address space when the location of data in the process address is not fixed, the pointer value showing the connection between nodes in the object becomes invalid (Figure 3). This problem must be considered when reusing data structure in an object.

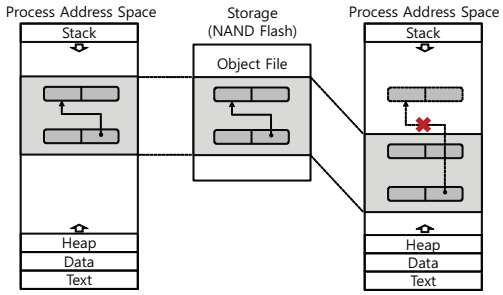

Figure 3. Pointer Validness Problem

2) Occurrence of collision between shared library and object followed by location change of mmap_base: In Linux, shared library is loaded in mmap area. mmap_base, which is the starting point of mmap area is determined by adding a random value to process address space address 0xB7701000 and mmap is allocated from this point. mmap_base in the beginning of the mmap area changes every time. A process is rerun as address space layout randomization(ASLR) [14] is applied and the address of shared library is changed. When an object is created in process and process is rerun, the location of loaded shared library changes which can cause a collision with the object. For instance, after process and creating an object in 0X3000, the object file cannot be mapped to 0X3000 when shared library is loaded in 0X3000.

3) Object sharing problem between programs: The size of loaded shared library is different for by each program run. Thus, the area where shared library can be located also varies in different programs. Because of this feature, a collision between shared libraries can occur when an object used in one program is used in another program (Figure 4).

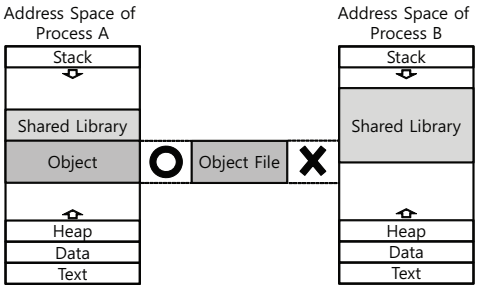

Figure 4. Object sharing problem between programs

B. Design Considerations on Pointer Validness and Shared Library Relocation

1) Fixed address mapping method: If an object file is mapped to process address space when the location of data in the process adress is not fixed, the pointer value showing the connection between nodes in the object becomes invalid. There are two ways to solve this problem. The first is the offset based method which does not fix mapping address of the object and uses relative address (offset value at file setting) instead of object pointer in the object. The second is using fixed address space to be mapped to objects in order to secure validness of object pointer. In the first method, because an offset value is used instead of an absolute address of object pointer, pointer swizzling process is required to gain the real address. The design in this study uses the second method to always map the objects in the same address space. In order to map objects in the same address space, each object must be distinguished. To distinguish each object, an object must have its own name and be managed through namespace. Also, metadata must be maintained to save mapping information of each object. Metadata showing mapping information of namespace and object are managed in a special object called metadata object.

2) Shared library area reservation: A collision between objects can occur as loaded location of shared library changes. To solve this problem, the area where shared library can locate is calculated and objects are always allocated to that fixed address. For this, anonymous mmap is done under the loaded address of current shared library to reserve areas around it so that the areas are not used for object creation (Figure 5).

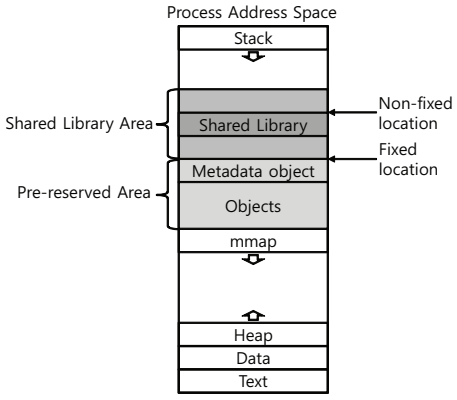

Figure 5. Shared Library Area Reservation

C. Object Protection and Node Allocation

Linux manages used address space by data structure called vm_area_struct. mmap() system call searches vm_area_struct to map unused addresses. In fixed address mapping method, objects must always be mapped to the same address. However, address space cannot be used for mapping if it is being used for other purposes. To prevent an address space from being used for other purposes, a reservation of address space for object mapping is done. When starting a program, namespace is circulated to create vm_area_struct for all object address spaces.

Node allocation in object is based on memory allocation algorithm of glibc 2.11.1. Metadata exists in the first region of each object to manage free chunks in the object. When the system receives byte unit memory allocation request by factor of object name, it searches namespace, finds the object's metadata, confirms free chunks in the object, and allocates memory chunk that fits the requested size. When available empty space is insufficient and the request cannot be satisfied, the object is increased and memory chunk is allocated to that spot. Increases in objects are continuous for heap. However, continuous address space may already be in use for some objects which would result in discontinuous increase.

IV. Implementation

Fast I/O technique was implemented in x86-32bit environment with Linux version 2.6.35. To maintain high stability and compatibility with Linux system, implementation took place at library level without kernel modification.

A. Object Mapping Information Management

Mapping location of each object must be remembered for fixed address mapping of objects. To distinguish each object, all objects have a name. This is managed through namespace which is composed of hash table. Metadata with mapping information is composed as follows. Metadata p_superblock manages metadata of all objects, p_ns_entry and metadata p_vm_area exists in each region (Figure 6).

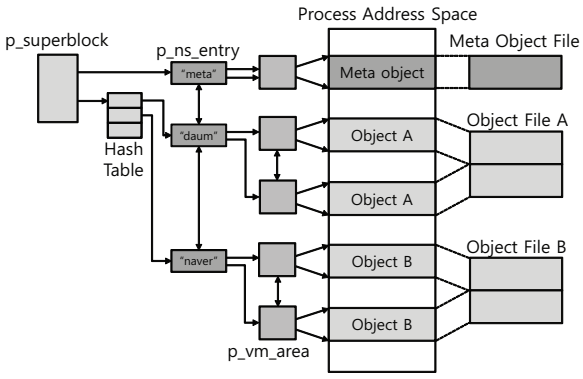

Figure 6. Overall Structure of Fast I/O Mechanism

p_superblock is implemented to manage metadata of objects, p_ns_entry, in a hash table which allows fast insertion, deletion, and searching of objects. Also, by managing p_ns_entry by a list, p_superblock does not need to search all entries of hash table which makes sequential search easier. p_ns_entry contains name fields of corresponding objects. It

manages p_vm_area, which is metadata of regions that make up an object, in a list and maintains pointers of the first and the last region. Since insertions and deletions of p_vm_area only take place at the end of the objects, p_vm_area can be managed by a list, without complicated data structure such as a tree. p_vm_area saves information on offset of object file and the location and size of process address space to which the corresponding region is mapped. Also, prev, next fields are maintained to approach other regions of the corresponding object.

These metadata must also have persistency which means that metadata must be managed in library. When managing metadata in kernel area, system calls are required for namespace approach which slows down the process. Also, since kernel modification damages compatibility, this study manages namespace related metadata at library level without kernel space. There is a unique object to save namespace related metadata. This object is called metadata object. These metadata managed node's allocation/removal

B. Shared Library Reservation

To prevent collision between objects and relocated shared library, the system checks relocation range of shared library and allocates objects to lower address. Since shared library is mapped to the starting address in mmap area, mmap_base must be known. mmap_base is located random value away from address 0xB7701000. The range of this random value is between (-1M byte) and (+1M byte). Therefore, the lowest address where shared library can be loaded is 0xB7701000 - 1M - (size of shared library). This address is called p_base. Collision with shared library can be prevented if an object is allocated to address lower than p_base. An object is allocated through mmap() system call which searches mmap_base for lower address to allocate empty address space. Therefore, for an object to be allocated to address lower than p_base, empty address space between mmap_base and p_base must be reserved. When starting a program, proc/maps file of the corresponding process is called and parsed. The size of shared library is calculated through parsed data. Then, p_base is found and empty area between mmap_base and p_base is reserved by anonymous mmap. Therefore, objects allocated through mmap are always mapped to address lower than shared library.

C. Object Reservation

Objects are always mapped to the same address. Therefore, collision occurs when the address space, to which an object is to be mapped, is used for other purposes. To prevent this, namespace is searched when starting a program and address space is reserved for objects to be mapped. Reservation means blocking usage of corresponding address space through anonymous mmap.

Metadata p_ns_entry of object in namespace is managed by hash table. Because only part of hash table is used, searching all entries is a waste. Therefore, p_ns_entry are managed by a list and all objects can be approached by circulating this list. Also, the list of p_vm_area on each object is circulated to reserve address space of all regions of the corresponding object (Figure 7).

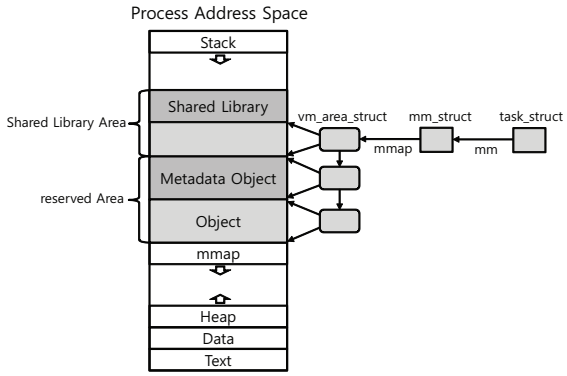

Figure 7. Shared library reservation and object reservation

D. Object Interface

In this section, object interface is described. Address space of library area and all objects must be reserved by calling p_reserve() function before calling main function when starting a program. Reserving shared library area prevents a collision between shared library and an object and reserving all objects prevents usage of object address space for other purposes. p_create() function is called for object creation. 4KB sized object file is created when p_create() is called as factor of object name, and this file is mapped to process address space for 4KB sized object to be created. The function that allocates nodes in an object is p_malloc(). p_malloc() function allocates object name, receives memory size as factor, and searches and returns free node in corresponding object. To reuse objects created previously, p_map() function is called and object name is received as a factor. p_map() searches metadata of corresponding object(p_ns_entry) through hash function, circulates metadata to all regions of an object, and maps the metadata with object file. Objects can be approached through this process and all nodes of tree data structure can be approached by approaching the root node in object through get_prime_node() function.

V. EVALUATION

This section explains the performance evaluation of Fast I/O technique in smart TVs. The following two experiments were conducted and the performance was measured. The first experiment was to compare the performance of File I/O to that of Fast I/O technique interface on object creation, deletion, mapping, expansion, and reduction. The second experiment was to measure and compare the performance of smart TVs on executing simple web browser with and

without Fast I/O technique. The experiments were conducted in AMD Phenom X4 925 Processor, 4GB DDR3 DRAM environment. An SSD (60GB OCZ VERTEX2 SATA 2) was used for storage and additional experiments were conducted on web browser with ramdisk, 500GB, 7200RPM hard disk. Dillo 2.2 [15] was used for web browser.

A. Interface

Performance of Fast I/O technique interface was measured and compared to that of File I/O on object creation, mapping, expansion, and reduction. With p_create() command, Fast I/O technique creates object file, maps the file to address space after increasing its size, and creates metadata that manage the object. To achieve the same tasks with file system, open(), ftruncate(), mmap(), and fsync() functions were used. Figure 8 shows measured performance on object creation, deletion, expansion, and reduction. Fast I/O technique interface and File I/O functions showed similar performance overall.

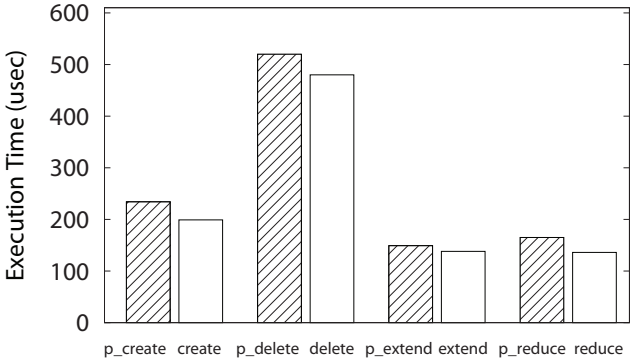

Figure 8. Fast I/O vs File I/O (create, delete, extend, reduce)

Figure 9 shows performance measurement on object mapping and removal. In common with other interfaces, interface that file is mapped to process address space have similar latency

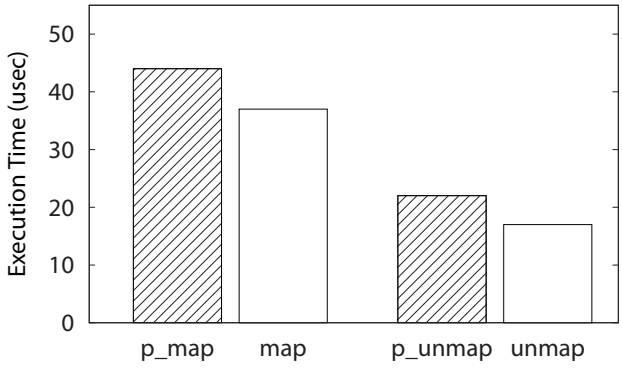

Figure 9. Fast I/O vs File I/O (map, unmap)

B. Web Browsing

In this study, cache mechanism of web browser was changed by applying Fast I/O technique. When opening a web page in the original web browser(Legacy Dillo), HTML is received by web, cached in disk, HTML is parsed, and tree is composed. The web page is displayed by a tree and when the browser is closed, the tree are eliminated. When opening a cached web page, HTML is read from disk and generate a tree after parsing process. And the web page is then displayed on the screen. When smart TV Fast I/O technique(F-Dillo) is applied, the web browser creates the tree structure through parsing process and caches the structure. This tree is reused to display a cached web page when a user re-visits the page after closing the web browser, bypassing parsing process and tree building process.

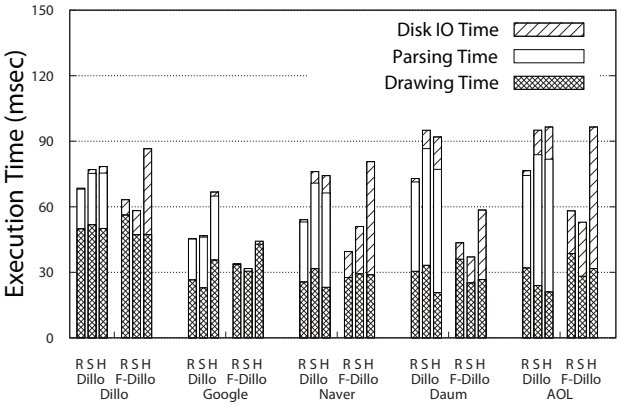

Figure 10. Execution Time by Web Page

For performance evaluation, the time from starting a web browser to displaying of web page is measured. Running times for disk IO, parsing, and displaying steps were measured. Figure 10 shows the time measurements for disk IO, parsing, and displaying steps. The original web browser and the web browser with Fast I/O technique are marked as Dillo and F-Dillo, respectively. For storage, ramdisk, an SSD, and a hard disk were used which are shown as “R”, “S”, and “H”, respectively, in Figure 10. With Fast I/O technique, parsing time is removed but disk IO time is substantially increased. This is due to the overhead caused by large tree which increases to 10 times larger than the size of HTML. When using a hard disk as storage, the overhead of disk IO offsets the effect of omitting parsing process, and it can be confirmed that Fast I/O technique is not appropriate in an environment using a hard disk as storage. When using an SSD as storage, parsing time was eliminated and disk IO time increased as in an HDD environment. However, since SSDs have Fast I/O speed, the performance enhancement by parsing time omission is larger than overhead increase by disk IO. With ramdisk, although parsing time was also eliminated, increase in disk IO overhead was more severe

than with an SSD.

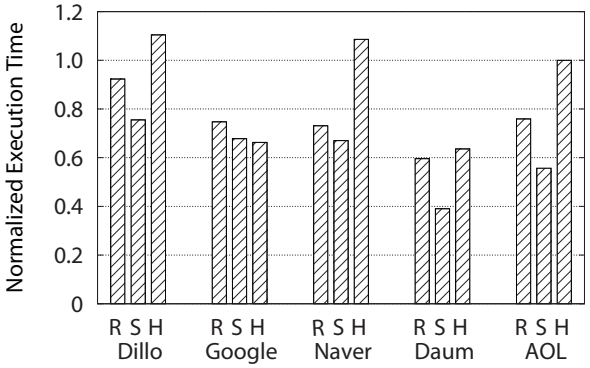

Figure 11. Normalized Execution Time by Web Page

Figure 11 shows running time of F-Dillo in an HDD and an SSD environments, normalized against Legacy Dillo running time. In an HDD environment, there were web pages that showed reduced performance with Fast I/O technique due to high disk IO overhead. However, with an SSD, all web pages showed performance enhancement with Fast I/O technique with maximum of 61% improvement.

VI. Conclusion

This study presents Fast I/O technique in a smart TV environment. Through Fast I/O method, persistency can be selectively added to memory data that have been dynamically allocated. Applying this method, applications can reuse XML's tree without deserialization process even when a process is rerun after it is terminated. However, as seen in the experiment results, using Fast I/O technique increases disk IO overhead for saving converted data structure. For storage with slow input/output performance, such as a hard disk, performance reduction from increased disk IO overhead offsets the performance improvement from eliminating deserialization step. Using Fast I/O technique is not appropriate in such case. On the other hand, for SSDs with Fast I/O characteristic, increase in disk IO overhead is relatively small which brings overall performance improvement when applying Fast I/O technique. Since smart TVs use NAND based storage with relatively Fast I/O speed, performance improvement can be expected when Fast I/O technique is applied.

Acknowledgment

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

References

[1] Android open source project. [Online]. Available: http://source.android.com

[2] Tizen open source project. [Online]. Available: http://www.tizen.org/

[3] T. C. Lam, J. J. Ding, and J.-C. Liu, "Xml document parsing: Operational and performance characteristics." *IEEE Computer*, vol. 41, no. 9, pp. 30–37, 2008.

[4] K. Zhang, L. Wang, A. Pan, and B. B. Zhu, "Smart caching for web browsers," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 491–500.

[5] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris, "Operating system support for persistent and recoverable computations," *Communications of the ACM*, vol. 39, no. 9, pp. 62–69, 1996.

[6] A. Dearle, R. Di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan, "Grasshopper: An orthogonally persistent operating system," *Computing Systems*, vol. 7, no. 3, pp. 289–312, 1994.

[7] E. Shekita and M. Zwilling, *Cricket: A mapped, persistent object store*. Center for Parallel Optimization, Computer Sciences Department, University of Wisconsin, 1990.

[8] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke, "The mungi single-address-space operating system," *Software: Practice and Experience*, vol. 28, no. 9, pp. 901–928, 1998.

[9] J. Mossière and X. R. de Pina, "Single address space or private address spaces?" in *Proceedings of the 6th workshop on ACM SIGOPS European workshop: Matching operating systems to application needs*. ACM, 1994, pp. 72–77.

[10] S. J. W. D. J. Dewitt, "A performance study of alternative object faulting and pointer swizzling strategies," in *Proc. 18th Int. Conf. Very Large Data Bases, Vancouver, BC, Canada*, 1992.

[11] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, "Software persistent memory," in *Proc. of the USENIX Annual Technical Conf., Boston, MA*, 2012.

[12] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2011, 2011, pp. 91–104.

[13] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories."

[14] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 298–307.

[15] J. Arellano-Cid and H. H. von Brand, "Network programming internals of the dillo web browser." in *sccc*, 2000, pp. 178–182.