NVM Heaps for Accelerating Browser-based Applications

Sudarsun Kannan
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
sudarsun@gatech.edu

Ada Gavrilovska
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
ada@cc.gatech.edu

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
schwan@cc.gatech.edu

Sanjay Kumar
Intel Labs
Hillsboro, Oregon, USA
sanjay.kumar@intel.com

ABSTRACT
The growth in browser-based computations is raising the need for efficient local storage for browser-based applications. A standard approach to control how such applications access and manipulate the underlying platform resources, is to run in-browser applications in a sandbox environment. Sandboxing works by static code analysis and system call interception, and as a result, the performance of browser applications making frequent I/O calls can be severely impacted. To address this, we explore the utility of next generation non-volatile memories (NVM) in client platforms. By using NVM as virtual memory, and integrating NVM support for browser applications with byte-addressable I/O interfaces, our approach shows up to 3.5x reduction in sandboxing cost and around 3x reduction in serialization overheads for browser-based applications, and improved application performance.

Categories and Subject Descriptors
D.4.2 [Operating Systems]: [Storage Management]; D.4.8 [Operating Systems]: [Performance]

General Terms
Design, Performance

Keywords
NVM, Browsers, Sandboxing, Native Client, Virtual Memory

1. INTRODUCTION
Browsers have become an indispensable computing platform for client devices, ranging from cell phones, laptops, tablets, and desktops, not just for web browsing, but rather a complete computing framework. Browser capabilities in support of rich in-browser services are increasing at a rapid pace, including by providing direct access to the underlying hardware and accelerators. For instance, the HTML5 Web Workers standard allows JavaScripts to exploit thread level parallelism in multi cores, and the WebGL APIs allow applications to utilize GPUs and access to other multimedia devices. With growing computing needs, large data access and storage needs have also increased, and fetching data from network is time consuming.

For performance reasons, however, recent popular runtimes like HTML5, JavaScripts, and Google Native Client (NaCl), have started supporting direct I/O access for web applications. The local storage interface for browsers exists in multiple forms like (1) simple key value store, (2) JavaScript (JS) based SQLite interface, (3) synchronous and asynchronous POSIX I/O interfaces for storage of large blobs of data. All of the above methods are compatible across different browsers but are limited by JS and dynamic compilation bottlenecks. Other state of the art methods like Google’s NaCl, support richer applications written in native languages (C, C++) that are 4-5x [24] faster than JS. Similar progress in reducing JS overheads has been made in the Firefox community via the asm.js [13] project.

While such features provide web applications with greater flexibility and performance in how they access and store data, a key challenge with all of the above methods is that access to underlying storage resource can leave the system in vulnerable state due to security threats. A commonly used solution to overcome security vulnerabilities involves isolation between web applications such that each web page instance has an exclusive access to its state, and does not share persistent data. Further, the untrusted web applications are completely isolated from the trusted browser framework and underlying OS by ‘sandboxing’ [22] [20] [24]. Sandboxing enables secured access to system resources like memory, network, and storage, by intercepting applications’ access (system calls) to these resources (i.e., system call interception) apart from static analysis. While sandboxing is important, as a side effect, they increase system resource access (system call) cost. Specifically for resources with software controlled access, the impact of sandboxing is higher. For instance, with storage devices, frequent I/O calls can substantially increase I/O latency and reduce throughput, such
that sandboxing becomes a dominant cost, irrespective of
the underlying storage device used. In addition, due to cur-
rent block-based storage interface, the impact of necessary
data serialization (before writing to storage device), and de-
serialization (retrieving data from storage) in a sandboxed
environment further degrades application runtime.

Hence a key principle in reducing sandboxing cost is to re-
duce the software intercepted resource access without com-
promising the protection features of sandboxing. We use this
principle, by proposing to use next generation storage class
nonvolatile memory (NVM) like PCM, as a virtual mem-
ory (VM) \cite{21, 10} as opposed to a block-based device, and
exploit the hardware controlled virtual memory-based iso-
lation between applications. By using a VM-based interface,
coupled with features like memory page protection, each web
application is restricted/isolated to access its own state in a
restricted boundary. Our approach requires browser appli-
cations to use NVM-specific heap interfaces to overcome the
limitations of both block-based and memory mapped I/O
system calls. While this requires application level changes,
it avoids a substantial number of sandboxing interceptions
(for example, intercepting every read/write call), and hence,
reduces the overall resource access latency critical for end
user devices.

To realize the benefit of our proposal, we use the well
known Google Chrome-based Native Client (NaCl) frame-
work. With NaCl, applications run as a browser extension
across client devices supporting major OSs, like (Windows,
Linux,Mac, ChromeOS). We refer to the NaCl framework
as state of the art, because NaCl applications are approx-
imately 4x faster than their HTML5 JS counterparts, and
compute intensive NaCl applications developed in C, C++
experience less than 5\% overhead relative their native alter-
 natives. We provide NVM support for browsers by emulat-
ing DRAM as PCM which are not yet commercially avail-
able.

The technical contributions of this work include:

- **Analysis of browser I/O performance:** We analyze the
  impact of sandboxing and serialization on browser I/O
  performance for the state of the art NaCl and the com-
  monly used HTML5 I/O interface.

- **NVM heap interface to improve I/O:** We propose a
  browser framework that uses NVM as a persistent heap
to reduce the sandboxing and serialization impact on
  I/O performance.

- **Evaluation of applications and benchmarks:** We im-
  plement and evaluate our approach in the well known
NaCl framework using representative benchmarks, demon-
strate reduction of sandboxing and serialization cost,
and improved performance. Our approach applies to
JS-based sandboxing too with per webpage sandbox.

2. **MOTIVATION AND BACKGROUND**

**I/O in browsers.** I/O capabilities for in-browser web ap-
applications have existed for a while, with every browser sup-
porting a customized application interface. Recent HTML5
standardization efforts, however, provide several I/O and
data storage interfaces. First, there are the traditional appli-
cation transparent and explicit browser caching which have
been extensively studied in the past. \cite{23, 15}. Other impor-
tant forms include a simple key value store used for stor-
ing user personalization information. More structured data,
that includes metadata storage of browser cache, browser

game states and others, use a JavaScript (JS) based SQLite
interface. Applications that require storage in blobs, for
instance, downloading a compressed video file and decom-
pressing it before playing, use synchronous and asynchronous
file system interfaces. Other motivating examples include
persistent uploader, where a file is to be uploaded is copied to
a sandboxed region of a web application that allows persist-
tent upload across browser crashes, offline video viewer with
efficient seek capability, ability to prefetch multimedia as-
sets in the background, audio-video editor that is supported
by offline access of cache, offline web email clients. Yee et
al. \cite{24} discuss an interesting photo storage application us-
ing the NaCl framework to store and process images. While
the need for I/O in browsers has been increasing, poor I/O
performance has continued to pose significant limitations to
web application developers \cite{4}.

**Storage access overheads.** Recent studies on end-client
devices \cite{15} analyzed different end user applications, smart-
phones and flash storage devices and concluded that (1) flash
storage performance variation across devices as the main
reason for poor application I/O including browsers, and that
(2) high random writes dominate the I/O cost. Replacing
the flash devices with future byte-addressable nonvolatil-
memory devices with 100x higher bandwidth and similar
random vs. sequential access cost can considerably reduce
the impact of (1) and (2). to achieve complete latency ben-
efit from these new types of devices. We need to revisit the
device interface, so as to reduce the software overheads for
data accesses, i.e., reducing the number of indirection levels
before accessing the device, which can dominate the access
latency cost. For instance, in case of Android, due to the
inherent design of the OS and multiple layers of sandbox-
ing, I/O performance can be substantially smaller (simple
database row insert test using the Android SQLite interface
showed around 300\% slowdown). In other environments,
like state of the art NaCl, the indirection happens from un-
trusted to trusted region and finally to OS as a system call.
Reducing software interactions, and exploiting the hardware-
supported (virtual memory-based) data access, is therefore
key to reducing data access cost.

**Sandboxing.** The key goal of application sandboxing is
to isolate applications from code and data of other appli-
cations by restricting the access to system resources and to
comply with user granted permissions. Sandbox mechanism
variants across systems, ranging from rule-based execu-
tions to virtual machine emulation to static code profiling.
In the case of higher-level languages like Java, language con-
structs and runtimes provide the sandbox (Dalvik VM in the
case of Android-based systems), for systems that support
native C, C++ languages (NaCl), a separate sandboxing layer
enforces restrictions on instructions and system calls that
applications can use. But on the whole, while sandboxing is
required to improve security when running untrusted and
untrusted code, frequent access to system resources can af-
flect performance. Specifically, access to storage device comes
with increased access latency.

**Sandboxing in Google Native Client.** We next provide
a brief background on sandboxing, and discuss the impor-
tance of choice of interface for improving the I/O perfor-
mance. While we use the Google-based NaCl, other run-
times also have similar sandboxing cost. The NaCl based
browser application consists of a trusted and an untrusted component. The browser user interface and application libraries are untrusted components whereas the browser execution environment and the NaCl framework are trusted components. The trusted and untrusted components have their own private address space similar to user and kernel layers of a traditional OS. A transition from untrusted to trusted region (or the reverse) requires stack switching. More details about trusted and untrusted components can be found in [24]. The NaCl framework adapts two levels of sandboxing – inner sandbox and outer sandbox [24]. The inner sandbox provides binary validation by using static analysis and restricts unsafe instructions. As most analysis is done statically, the inner sandbox has less impact on application runtime whereas for outer sandboxing, untrusted applications’ use of system call wrappers are intercepted by a trusted region. Similar to context switches between user and kernel-level, control transfers happen between untrusted (browser application) and trusted regions (trusted browser framework) using springboard and trampoline techniques, making a system call highly expensive compared to general applications. For browser-based I/O, NaCl uses the HTML5-compatible Pepper library and memory access by untrusted applications is restricted to a specific address range using page protection mechanism and any region can be expanded/shrunk by registering it with the NaCl runtime. The runtime maintains a per-process (an untrusted application) address table mapping containing the address range and access permissions, and registered address regions do not incur sandboxing costs, but rather leverage the hardware support for illegal access protection. This is in contrast to file system operations, where every I/O syscall needs to trap. Frequent I/O calls by applications cause severe I/O and bandwidth impact irrespective of the underlying physical device (e.g., NVM, RAMDisk, or SSD), which makes such file system calls highly unsuitable in browser-based environments.

To understand the importance of choice of interface, we did a simple test using NaCl. Figure 2 shows a simple benchmark demonstrating the I/O performance issue in browsers, comparing the browser-based case with the native execution of a simple benchmark that opens a file, writes data to it in chunks of 512 bytes, and then closes it. There is a large increase in I/O time for the browser case, attributed to the fact that with increasing numbers of I/O calls, sandboxing overheads also increase. Use of the existing memory map (mmap) system call interface when dealing with few large files can reduce sandboxing impact, but when the number of files that need to be mapped and unmapped is high (for instance compressing all image files in a directory by mmaping them), the overall user-kernel context switching can negate the memory map gains as shown in our evaluation section (see Sec. 4).

### Serialization in Sandboxed Environments

Serialization is a well known method of converting in-memory data structures to a sequential persistent data format where deserialization is the inverse operation. Serialization results in additional I/O system calls primarily in the form of seeks and writes/reads to write data to persistent storage. In the case of sandboxed environments, the cost of serialization impact is substantially higher due to additional system call interception. Providing applications with a non-volatile heap-based interface can avoid serialization/deserialization across I/O data access, by storing and loading data structures exactly in the way they are stored in memory.

### 3. DESIGN

```c
// storing to NV Heap
Image* imgdb = nvmalloc("img_root", size);  
for each new image:  
  Image*imgdb[cnt++]=nvmalloc(size, NULL);  
  cnt++;  
// reading from NV Heap
img = nvmread("img_root", &size);  
// implicit load of all child ptrs
```

Figure 4: Programming Model
I/O calls in a sandboxed environment transition from untrusted to trusted to privileged (kernel) layers. Our design reduces the multiple levels of software redirections for I/O calls, by relying on fast hardware-based page access for persistent storage. We achieve this by exploiting the byte addressability and hardware-supported page-based memory management and protection techniques for NVMs. Applications allocate persistent regions of NVM similar to a heap, and instead of file system reads and writes, perform load and store operations to the persistent regions. Key differences with prior work [24] include NVM support for and several optimizations specific to persistent browser applications in sandboxed environment and a virtual memory-based NVM kernel manager compared to the file buffer cache extensions in prior work.

3.1 OS Support

To integrate NVM at the OS level, it is represented as a special node in a heterogeneous memory system. We leverage OS-level NUMA support, by configuring a NUMA memory node as NVM during system boot. To manage the NVM node, we have developed a custom NVM manager by extending the Linux memory management which controls allocation, deallocation and persistent metadata structure maintenance in the kernel layer. Each process has a persistent page tree which is loaded from persistent storage (SSD currently) during process launch. Each process uses a unique identifier to load its persistent pages in its address space. The NVM manager provides persistent NVM allocations using the ‘nvmmap’ and ‘nvunmmap’ system calls, generally used by the allocator. The mmap call results in creation of a virtual memory area (VMA) structure containing several pages. The kernel internally maintains persistent per-process kernel data structures, which contain a tree of VMA structures where each VMA contains a tree of pages for supporting applications across restarts.

To emulate application session-level persistence, we prevent the OS from swapping persistent pages allocated by the NVM manager and use an SSD for storing kernel structures and data pages for persistence across reboots. A key feature specific to browsers is the support for compartments, where each VMA contains a tree of pages for supporting applications across restarts. To prevent the OS from swapping persistent pages allocated by the NVM manager and use an SSD for storing kernel structures and data pages for persistence across reboots. A key feature specific to browsers is the support for compartments, where each VMA contains a tree of pages for supporting applications across restarts.

3.2 NVM Support for Sandboxed Browsers

Our design consists of a user-level library to allow NaCl browser applications to explicitly allocate and access persistent data in NVM. The NaCl framework categorizes the runtime into trusted and untrusted components (see Figure 3). The trusted region implements protection; it is responsible for providing all system resource references and handles, along with system call interception. The untrusted region provides the user-level interfaces to the NaCl applications which are intercepted by the trusted runtime. Since the two regions maintain separate stacks, a call from the untrusted to trusted region results in expensive stack switching. To avoid such costs, application-level resource management can be done in the untrusted region after getting a resource reference. For instance, user-level memory allocators like ‘dlmalloc’ can be implemented in the untrusted region, and the reference of memory addresses using the sbrk()/mmap() call can be obtained from the trusted region. To match this division of state and functionality across NaCl components, we divide our user-level NVM component across the trusted and untrusted layers of the NaCl library. We first describe the untrusted NVM component followed by the trusted component.

Untrusted NVM allocator. The untrusted NVM component provides byte-addressable interfaces to applications and implements user-level management of NVM state. Applications allocate persistent chunks using the interfaces offered by the untrusted NaCl NVM allocator component. The persistent allocator is an extension of the glibc ‘dlmalloc’ library similar to other works [10], and is implemented in the untrusted layer. Figure 4 shows a simple programming model of how applications allocate (nvmalloc) and access persistent heap (nvread). Persistent pointer handling across restarts is done by well-known pointer swizzling technique. Placing the allocator in the untrusted region, instead of the trusted component avoids substantial stack switching overhead between the two regions for every application memory allocation call, but also requires sandboxing specific allocator optimizations. The implementation is secure because (1) an mmap-based reservation by the allocator and other memory references are still obtained via a call to the trusted region, and (2) any illegal memory address access outside the registered range of the untrusted application would result in an application exception. The allocator maintains a log of process-level persistent allocations, and the log is used on restarts to locate allocations from previous sessions. An allocated chunk can be located at any offset of a compartment (mmap’ed region), where every chunk consists of metadata maintaining detail about its parent compartment as well as an offset within the compartment. All metadata is maintained in persistent memory.

Sandboxing specific allocator optimizations. Most optimizations revolve around reducing frequent use of the system calls in the allocator (e.g., sbrk, mmap, unmmap, etc.), as this can negate the benefits over a POSIX I/O interface. Two key optimizations include (1) allocator memory reservation size, and (2) dividing the memory reservations among multiple threads. Regarding (1), allocators generally use mmap/sbrk to reserve a few pages (‘dlmalloc’ uses OS pagesize) of memory, try to fit in application allocations in the reserved regions, and when the reserved region is insufficient, invoke a mmap/sbrk call again. For applications requiring large persistent storage needs, this can result in a substantial number of system calls (and hence sandboxing overheads). To avoid this, applications provides a hint to the allocator for making larger reservations, (maximum of 16 MB), with default reservations of 4MB. (2) making large reservations in multithreaded applications can be dangerous. To avoid this, we divide the application reservations into thread-level compartments discussed earlier [24].
Trust NVM component. The trusted NVM component is a thin layer responsible for providing (1) an indirect access to the system level NVM interfaces like ‘nvmmap’, for allocating and accessing persistent regions, (2) maintaining a per-application persistent NVM access region table with different protection levels, and (3) handling out of bounds access protection faults. Every untrusted application registers a unique key with the trusted region for the first time, and the same key is used across sessions. The unique key registration also creates a persistent access table for the application (see Figure 3). After registration, applications use untrusted allocators for persistent memory allocation, which invokes an NVM specific mmap call to the trusted component. The trusted component checks if the requested memory reservations are private, and adds the memory address range to the access table. Since the trusted and untrusted components have separate logical segments and stacks, after memory allocation, the trusted component converts the memory reference to the untrusted application address range.

Once the NVM address ranges are mapped into the process address space, the applications are free to access any memory address in the range and do not encounter sandboxing costs. This provides substantial performance benefits by reducing the outer sandboxing overheads. Across application (browser application) restarts, unique keys are used as unique naming entities for reloading the application access table. Our current design relies on the browser application to provide a unique key which is similar to sandboxing in the Android framework[7], where each application has a unique key across sessions. Future work would focus on more application-transparent key generation.

4. EXPERIMENTS

In order to investigate the benefits of leveraging the byte-addressability of future nonvolatile memories in improving browser application’s I/O performance, we seek to understand the following. (1) Is the current storage device performance mainly responsible for the I/O slowdown in sandboxed environments like browsers? (2) What is the impact of the choice of storage interfaces on a sandboxed environment? (3) What are the benefits of treating NVMs as a nonvolatile heap as opposed to a block storage device? To answer these questions, we use the browser-based WebShootbench[6] benchmark, and two applications: Snappy data compression, and an offline content-based email classifier. We next provide details on the experimental methodology.

Evaluation Methodology. For representing an end-user device like a smartphone, we use a dual-core 1.66 GHz D510 Atom-based development kit with 2GB DDR2 DRAM, Intel 520 120GB SSD. For NVM, the I/O interface is replaced with a heap interface (nvmalloc). In all workloads, to account for slow NVM writes that miss the 1MB L2 cache, we use Pin-based binary instrumentation to measure the ratio of load and stores in the NVM-allocated address range, and then use hardware counter values to estimate the total load/store misses due to NVM accesses and use access latency values from [21]. We observed that for most applications except for a hashtable benchmark, the cache misses were less than 1-1.5% as discussed in other work [12].

4.1 Benchmark Analysis

WebShootbench[6] is an open source NaCl benchmark originally derived from the Computer Language Benchmarks Game[1] to compare the speedup of NaCl with JavaScript; we focus just on the workloads that depend on I/O. Table 1 shows the I/O vs. compute time on a vanilla Linux Atom platform. To understand the impact of storage device on performance, we evaluate our experiments in SSD, RamDisk and emulated NVM.

- Fasta (FS) is a write intensive benchmark that generates random DNA sequences by weighted random selection from a list of predefined sequences, and writes 3 sequences line-by-line. The number of ‘writes’ system calls are substantial
- Revcomp (RC) reads DNA sequences line-by-line from the output generated by Fasta, and for each sequence, writes the ID, description, and the reverse-complement sequence to output. Blocking read calls dominate the I/O time of the application.
- kNucleotide (KN) reads the DNA sequence from Fasta’s output line-by-line, generates k-nucleotide sequences and each k-nucleotide is updated to a hashtable, with values representing counts of occurrences. The I/O time is less than 13% compared to the total compute time (hashing).
- Spell Check (SC) loads popular ‘Wordnet’ dictionary files[16] into a hashtable, generates words from an input file and identifies words that are not in the dictionary. The dictionary set contains 16 files each containing its own hashtable. We use four 16 MB input text files.

Table 1: Time spent on I/O by benchmark apps.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>I/O time(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fasta</td>
<td>41.207</td>
</tr>
<tr>
<td>Revcomp</td>
<td>49.33</td>
</tr>
<tr>
<td>kNucleotide</td>
<td>12.32</td>
</tr>
<tr>
<td>SpellCheck</td>
<td>19.89</td>
</tr>
</tbody>
</table>

Table 2: NVM gains: server (Sandybridge) vs. client (Atom)
Observations Figure 5 compares the use of NVM as a heap with RAMdisk and SSD performance. The applications generate/access around 64MB of I/O data. (1.) As expected, NVM as a heap shows significant performance gains (Y-axis shows runtime) in all the benchmarks with maximum gains for read-intensive ‘Revcomp’ (3.5x) and least gains for compute intensive kNucleotide (20%) and short running spell check application. (2.) An interesting result is that, both RAMdisk and SSD perform poorly with very little difference between them. This shows that, irrespective of the storage device, frequent I/O read/write calls hurts both SSD and RAMdisk which shows the impact of sandboxing in browser I/O slowdowns, and (3.) due to sandboxing, even writes that generally benefits from a buffer cache, suffer substantially using the POSIX I/O interface in sandboxed machines. As expected, increasing the I/O size, resulted in a widening gap between the NVM heap and RAMdisk approaches (not shown here for brevity). We also observed that, the speedup achieved from our NVM-based design (compared to RAMdisk) on the client platform (Atom) to be higher than the server platforms (Sandybridge) as shown in Table 2. The benchmark runtime was maintained constant across platforms. This is mainly because, software-based sandboxing increases total instructions executed, and reducing such actions in slower cores with our NVM design shows higher benefits. These observations show that the choice of interface is critical to I/O performance in browsers apart from the storage device and using NVM as heaps can avoid substantial sandboxing cost.

4.2 Application Evaluation

We next evaluate the effectiveness of NVM as a heap for browser I/O using two other applications: (i) a NaCl-based disk cache compression using Snappy [5], (ii) Bayesian-based offline email classifier. Using Snappy, we compare the implications of using a memory interface for NVM vs. a POSIX-based block interface or a memory backed ‘mmap’ interface. With the email classifier, we analyze the serialization/deserialization benefits of NVM. For the POSIX I/O and mmap interface, we use a RAMDisk-based file system (tmpfs) to avoid the storage device noise.

Snappy Compression. Snappy is a high performance compression/decompression library (ships with Chrome sources) with preference for speed than compression size. We ported it to NaCl approx. in 2 hours and use it to compress approximately 500MB of default browser cache data (3001 files), as shown useful in [24, 8]. The cache consists of binary, text, images and video files, and compression achieves 28% reduction in cache size. The compression/decompression time is well within the limits of average web page load time (currently around 4-7 seconds). For the POSIX I/O and mmap-based interface, all the cache is in RAMDisk, whereas for NVM heap, we manually load the cache to the emulated NVM region with a helper process. We are currently working on changing the entire cache interface to support NVM, and future work would not require such a helper process. The Snappy algorithm loads the entire file to memory, performs compression and finally writes to an output file (RAMDisk or NVM). In the case of ‘mmap’, each file is mapped into memory, compressed and unmapped, whereas for the POSIX I/O interface we use fread/fwrite. For NVM, we use the nvread() method.

Figure 7 shows the runtime comparison and Figure 8 compares the total context switch, system calls and stack switching for all three interfaces. The NVM-based interfaces outperforms mmap-based interface by nearly 2.5x and the POSIX I/O interface by over 3x. Next, analyzing the reason for the performance difference, the left of Figure 6 shows the average user-kernel context switch counts per second. The total number of context switch when using a mmap() interface is substantially higher compared to block-based fread(), fwrite() calls. When using the ‘mmap/unmmap’ system calls, every invocation results in a context switch as confirmed by the figure in the center which captures the overall system call invocation of the application. In the POSIX I/O interface, while fopen/fclose results in a system call, fread/fwrite are library calls, which explains substantially lower context switch. But the amount of stack switching between untrusted and trusted regiona due to fread/fwrite calls are substantially higher due to sandboxing, which explains why the POSIX interface suffers compared to the mmap-based interface even though user-kernel transition is less when compared to mmap interface. In the case of our proposed nvread() interface (see Figure 4), files are stored as memory chunks (objects) with a name identifier maintained by the allocator. A nvread() call results in mapping a 2MB region (see allocator optimizations), containing one or more cache objects, resulting in fewer system calls, and hence 2x fewer system calls compared to the POSIX I/O, reducing the sandboxing cost. Further, the application has a sequential access pattern with runtime less than a second resulting in comparatively lower cache misses as observed by others on client platforms [12].

User Personalization: Email Classifier.

We next analyze an offline mode Bayesian-based email classifier ported as a NaCl application. It classifies new emails using learning data generated from prior classifications with learning data stored in a persistent storage. We model the application such that, all classification is done before a webpage load. We use the CMU text learning group dataset for user personalization [17], which contains 10 newsgroup email categories like sports, economics, movies, etc. and randomly choose 100 emails as input. The learning
data is approx. 253 MB (24 MB per category). Using this initial categorization, the application (1) extracts feature points from new emails, (2) loads training data and (3) compares the input feature points and training data set. The library reads the learning data file line by line, generating special classification structures and tokens, with token generation consuming the maximum time. While token generation varies based on the input data that needs to be classified, the header structure is constant and results in a substantial number of fseek/fread calls, and hence the deserialization cost in a sandboxed environment. When using NVM, all the structures are stored in persistent memory as-is avoiding the need for deserialization during load. The x-axis in Figure 6 varies the number of learning categories to compare the NVM and POSIX I/O approach. Clearly with increasing number of email categories, the benefits from reducing the serialization and sandboxing provides up to 2x improvements, which shows the effectiveness of using NVM as nonvolatile heaps in reducing I/O serialization cost in a sandboxed environment.

5. RELATED WORK

Sandboxing and Browsers: The impact of software-based sandboxing has been extensively studied, from the seminal work of Wahbe et al. [22] and most recently by [24]. A recent work on browsers focused on complete browser and OS redesign for security [20], but lacks support for current browser frameworks. To the best of our knowledge, however, our work is first in exploring the opportunities of using future NVMs as a heap for reducing sandboxing impact on storage. Other efforts are focused on moving a major portion of sandboxing mechanisms to the OS, similar to Android [2], but for OS agnostic applications like browsers, completely relying on OS-based isolation seems unlikely. Related work like [25] uses NVM to optimize storage in virtualized environments. Finally, a recent work on exceptionless system calls [19] studies the impact of reducing system call blocking costs and we believe such kernel techniques can be very useful in our future work.

NVM as a Heap: Prior work [9, 11] has proposed the use of NVM with the POSIX I/O interface. Our observations for browsers clearly show the disadvantages of such proposal. NVM as a heap using PCM was first proposed by Volos et al. [21] while other research like [11] discussed other issues like language/interface support, orphan pointers and pointer swizzling. Our work of using NVM as a heap is complimentary to the above, but the key focus is to understand issues like sandboxing in end-user devices and designing and addressing such issues using a specialized heap-based solution. Additional contributions of this work include the design of an NVM kernel memory manager that provides an end-to-end system support with flexibility such as compartments, whereas most prior work uses an extension of the RAMDisk-based file system to emulate NVM as a virtual memory.

6. CONCLUSION AND FUTURE WORK

By using NVM as a heap and exploiting the byte-addressability of NVM devices like PCM, aided by hardware paging and page protection techniques, we showed that close to 3x improvements in storage performance can be achieved in sandboxed environments like browsers. Considering the rapid growth of end user devices with a rich pool of applications, almost all frameworks are moving towards some form of sandboxing model. Hence, we believe our proposal can provide substantial performance, specifically storage access gains in such sandboxed environments. Our next steps will include studying more complex applications, such as games which require persistence for accessing graphical as well as user data, and the impact of using NVM on other browser components, such as database and cache. We also plan to explore optimization in the kernel that can further enhance our design.

7. ACKNOWLEDGMENTS

This research is supported in part by the Intel URO program on software for persistent memories and by NSF award CCF-1161969.

8. REFERENCES


http://bit.ly/qOTFid


*Commun. ACM*, 38.


