Improving Browser Performance using NVRAM

Aastha Tripathi
University of Texas at Austin
Austin, TX 78712
aastha_tripathi@utexas.edu

Mit Shah
University of Texas at Austin
Austin, TX 78712
mkshah@cs.utexas.edu

1. Introduction

The major objective of the project was to recreate the results of the research paper - *Kim, Kyusik, et al. "How to improve the performance of browsers with NVRAM." Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2017 IEEE 6th. IEEE, 2017.*

This research paper tried to explore the usage of Non volatile persistent memory(NVRAM) storage in the aspect of application performance. They presented in the paper, schemes for improving the user-perceived performance of browsers by using NVRAM. They defined the user-perceived performance of browsers using two metrics: browser launch time and page loading time. The browser launch time is minimized by caching the displayed frame buffer data in the fast NVRAM and thus reduced the I/O and processing time required for browser launch. The page loading time is be also reduced by preloading the web pages that can be revisited soon in the NVRAM. Through implementation on open source browser, we show that their schemes significantly improve both the browser launch time and the page loading time.

2. Background

In this section we will look at basics of NVRAM, how browsers work and later in the next section, we will see how they can be integrated.

2.1. NVRAM

In past few years, research and development of NVRAM has received tremendous interest from academia and industries. Fast advancement of semiconductor technologies has been a major factor for this. Currently available technologies for NVRAM are STT-RAM, PCRAM, and Memristor based RAM. Even though the physical characteristics of them are not identical, they are commonly fast, non-volatile, and byte addressable. Due to these characteristics, NVRAM is anticipated as the next generation storage media.

2.2. Browser

Let's go through the browser launch process. The first step is to load the browser executable, libraries, and configuration files from the storage. Generally NAND flash memory is used as the storage device. Second step of launch is to make widgets ready and load the default web page configured by the user.

To load the web page, browser first fetches the web resources that belong to the page from remote web server or local browse cache. Afterwards, HTML is parsed into DOM tree, CSS is processed for style formatting and javascripts are executed. An internal representation is generated in the form of render tree, which is later painted by the browser. Same procedure happens when the user requests another web page afterwards.

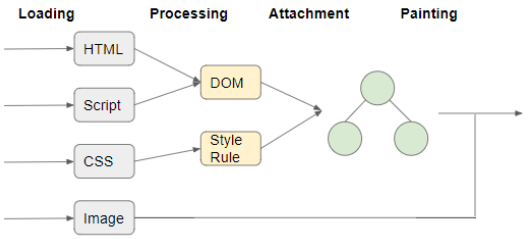

Figure 1: Processing of Webpage

3. Enhancing Browser Performance through NVM

Here we discuss the architecture required for reducing browser launch time and page loading time. As shown in Figure 2, they use NVRAM between main memory and storage devices. They use it as a space to store resources for browser cache, bitmap data and preloading graph.

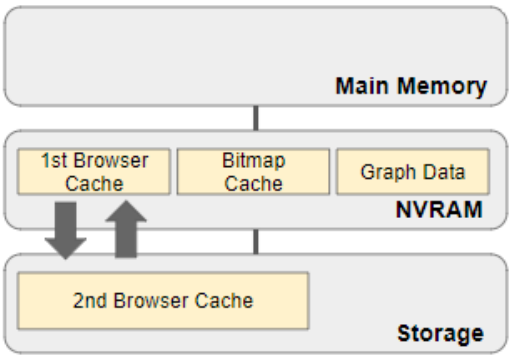

Figure 2: Memory Architecture using NVRAM for browser

3.1. Reducing Launch Time

3.1.1 Loading Executable Libraries and Configurations

Launch time of any application consists of a sequence of the I/O access times and the computation times. The first block required for an application launch is read from the storage and then the CPU proceeds the launch process with the block. This process is repeated until the launch of an application is finished, which is a cold start scenario. As discussed in the background section, similarly for browser also it is required to load executable libraries and configurations from the storage. These files can be extracted using *strace* tool, which helps in tracing the system calls. We can monitor system calls that have file name as the argument, like open(), create(), stat() and execve(). If we can place these files in NVRAM, which can be as fast as 2500 times than NAND flash memory, launch time of browser can be significantly reduced. Also, as we will know the full sequence of blocks, we can actually prefetch all the blocks before browser requests them. This will further reduce the launch time.

3.1.2 Making Widgets and Loading Home page

Afterwards it makes some widgets such as bookmarks, toolbar, menu etc. and configures window attributes. Last step is to load the default web page in order to finish the launch process, which is usually done as described in the previous section. One way is to keep the default page in NVM. However, for the fast "user perceived" launch of browser, we can cache the bitmap data into frame buffer, which removes the need for these steps. User may feel that browser is instantly launched, but as it is just bitmap data he can not interact with it. So actual processing is still performed in the background. Once it finishes, the bitmap data is transparently replaced by the actual web page.

3.2. Reducing Page Loading Time

They propose using user pattern aware pre-loading of web pages. Pre-loading and caching are the most used techniques for doing so. In preloading, web resources that are yet not requested are fetched and processed. It requires accurate prediction because it ends up consuming additional systems resources like network bandwidth, CPU cycles and memory space.

Proposed scheme preloads several web pages that has high probability given the current web page that user is browsing. In case of hit, page can be instantly displayed to the user. For doing so, user's visit page patterns are analyzed from his history and a graph is made that represents relationship among web pages as shown in Figure 3. This graph is also stored in NVRAM.

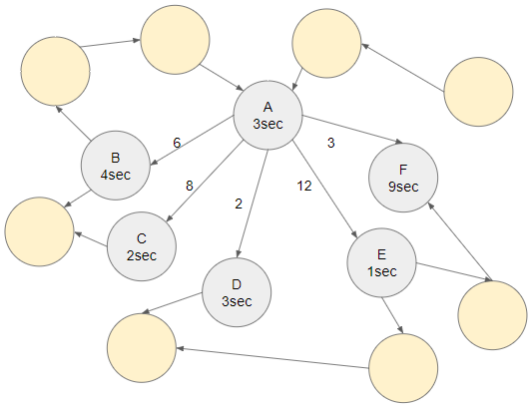

Figure 3: Visit pattern graph

Each node here denotes the web page visited and has an average page loading time. The value of edge from node A to node B denotes the count that visits page B via page A. In this example, page A has 3 seconds of average page loading time, and page B, C, D, E, and F are visited 6 times, 8 times, 2 times, 12 times, and 3 times via page A, respectively. In terms of visit count, it should be preloaded in E, C, B, F, and D order while user watches the page A. However, authors consider the average page loading time of each web page. They preload from the pages that have both higher visit count and longer page loading time. It is evaluated for each page by multiplying the visit count and page load time. One with highest values are then loaded. Since too excessive preloading can result in much overhead, they the number of pages preloaded in the background.

4. Implementation

4.1. NVRAM Setup

Major Design Decision

To recreate the results of the paper we had to first emulate NVRAM. There are several ways to emulate NVRAM for ex- using pmem library, RAMDisk, or using ext4+DAX (Direct Access). We go ahead with the third one that is ext4+DAX, in this case there was no code level modification, it used the same file system calls and load and store instructions. Pmem.io library itself emulate using the same technique but built a high level library over that to provide various other functionalities. Since we only need to read and write from NVRAM, these additional functionalities were not as useful and thus we emulated directly using ext4+DAX.

We emulated the NVRAM as described by the pmem.io. We first created a reserved region and made it persistent which was seen as a device named `/dev/pmem`. As linux give the functionality to have direct access we provided this device the direct access and mounted ext4 file system on the device.

The page cache is usually used to buffer reads and writes to files. It is also used to provide the pages which are mapped into userspace by a call to mmap.For block devices that are memory-like, the page cache pages would be unnecessary copies of the original storage. The DAX code removes the extra copy by performing reads and writes directly to the storage device. For file mappings, the storage device is mapped directly into userspace.

4.2. Performance model for NVRAM

As there is no NVRAM, all the functionalities of the code would be the same. However, we have to put in manual delays to measure the performance of NVRAM. In our use case, we are using NVRAM to read the saved default web page and the preloaded pages and to write the preloaded pages to NVRAM. Thus, the basic usecase of NVRAM is to read and write files in it.

To get the read performance, while reading most of the reads are satisfied by the cache which would be the same in case we are using NVRAM also (in place of DRAM). But the reads that are missed by last level cache are served by the DRAM, which in our case will now be served by the NVRAM and this is the place where we the delay needs to be accounted. The reads for NVRAM are generally speculated to be 2 times slower than DRAM, but for sensitivity study we took a range of values starting from 2 to 50. This helped us in showing that for even such a high latency of read in NVRAM, our browser performs better.

Major Design Decision

To get the write performance, first we analyzed whether for our performance measurement writes performance model is needed. As the writes are there when we actually visits a page as get the next probable page to preload, we preload those web pages as in we download those html files of the preload pages and writes to the NVRAM. These writes are done in background which does not affect the user perceive time. This brought us to the conclusion that writes performance does not affect our performance measurement of browser and does not needs to be encountered.

4.3. Browser Implementation

We implemented the browser using open source QTWebkit. We made a simple browser with refresh, edit toolbar, tabs etc. The browser code first initializes the The browser on start opens the default by loading it from the NVRAM.

4.4. Read Performance measurement

To measure the read performance we first measured the time taken by a page to be loaded in milliseconds. Then in the part of the code where it actually reads the file (either the default web page or preloaded page) from NVRAM, what are the number of cycle events which missed from the last level cache using `perf_events`, these are the reads which are served from the DRAM and on which we need to add NVRAM read delay. Lets assume there x number of events which get served from the DRAM and the DRAM takes t time in nanoseconds to serve that event. If the total time to perform a read request is T milliseconds then the new total time T' would be:

$$T' = T - x * t/1000 + factor * (x * t/1000)$$

where factor corresponds to the delay imposed by NVRAM which we took as different increasing values namely `2,5,10,20,30,50`.

5. Evaluation

We mainly used two metric to measure the performance of our browser: 1) Browser Launch Time for different user configured home pages and 2) Page Loading Time. In the following subsections, we will discuss them in detailed.

5.1. Browser Launch Time

The browser launch time is the duration from when a user touches or clicks the browser icon until a default web page is displayed. Here are the results by the optimization

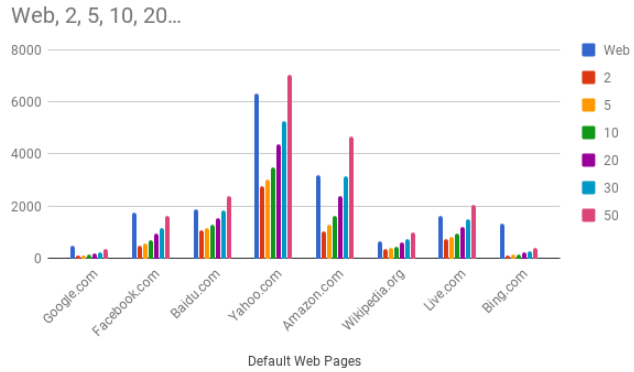

Figure 4: Launch Time for Different Home pages

technique discussed.

In Figure 4 we can see the browser launch time for different default web pages. Comparison is given for the cases when pages are loaded from network and when pages are loaded from NVRAM, with considering different factors (2,5,10,20,30,50) for read slowdown compared to DRAM. We can easily that for all the cases, time for loading from NVRAM is higher only when read slowdown of 50 is considered.

Another important factor here is to note the differences between static and dynamic web pages. Static web page means that it does not have much multi-media data like images, videos, etc, while dynamic web page means it has lots of dynamic content like that. It is easy to see that for static pages like Google, Wikipedia difference is not that much. On the other hand, if we look at Amazon and Bing, significant difference is notable, as both the web pages have lots of dynamic content.

5.2. Page Loading Time

It is the time required from requesting a new web page to point when browser has retrieved the web resources belong to the web page from web server fully. It is one of the most important metric that determines the performance of browsers.

As discussed previously, we employed user pattern aware pre-loading techniques to optimize page loading time. We modeled the patterns from 5 different users by using their browser history files. And afterwards, asked them to browse for a certain period of time through our implemented browser, which performed prefetching according to that particular user's visit pattern graph. We mainly noted two things: 1) Hit ratio - Percentage of web pages already prefetched from total web pages browsed by user and 2) Comparison of Average Time taken to load a web page with respect the case when all the web pages were fetched from the net. These two graphs are shown in Figures 5 and 6 respectively.

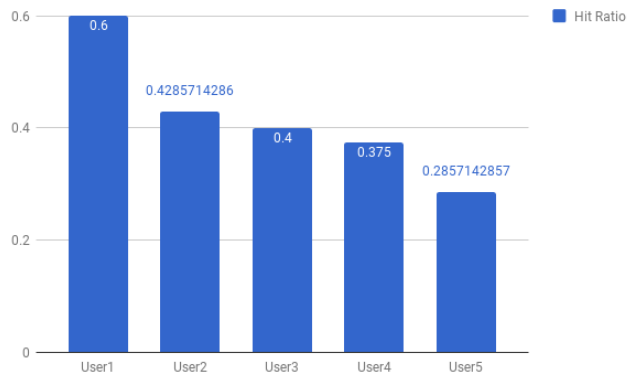

Figure 5: Hit Ratio for 5 users

We can see that hit ratio varied from around 0.6 to 0.3. And the most notable improvement was that average page loading time was almost decreased by 30%.

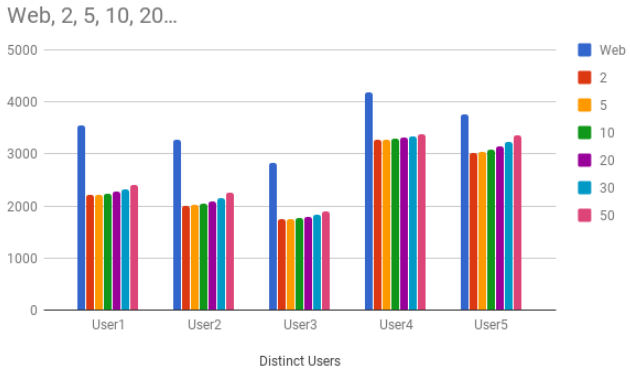

Figure 6: Average Page Loading time for 5 users

6. Conclusion

NVRAM was used as space to store resources for browser cache, the visit pattern graph for preloading, and default web page data for instant launch. The performance results (as shown) were reproduced as the paper and the results were justifying the paper. The Browser launch time wass decreased by around 61% and the Page load time decreased by around 30% as compared to the case in normal browsing schemes. We did sensitivity study for a range of values for emulating reads on NVRAM which showed that even the reads are as large as 50 times slower than DRAM the performance is still better or compared to the normal browser.

7. Limitations and Future Work

Limitations

For to evaluate the page load improvement performance. The paper gathered real workloads from users. The durations of collection are about a month and the numbers of pages visited by each user are from 1265 to 28039. They divided users into groups and evaluated an average performance. However, due to unavailability of the corresponding data, instead of dividing users into groups, we experimented the methodology on particular users. We built visit pattern graph corresponding to each user and thus each user have preloaded pages according their visit pattern and thus will improve the performance over the method used by paper(to have similar graph for all the users in the group).

Our performance measurement did not account the various numbers of webpages that we can preload. For now we by default preload 3 most probable pages but this can vary according to the size of NVRAM as if the size is large we can store more webpages without affecting the overall performance. This was because of the limitation of time as well as data to measure the same. Also the performance will also be depended on the nodes in the graph, which we could not measure because of the same limitation of data as the explained earlier, the paper collected the data over 2000 users which is not currently available.

Though the paper provided those results. Where they showed that as the preloading degree is higher, the hit ratio usually increase. However, in the results with T4 and T5 (as shown in figure 7), the preloading degree did not give any impact on the hit ratio. It may be because locality is very high as well as many part of requests are serviced by various schemes that browsers provide such as page caching or memory caching. Too excessive preloading can give a bad influence on user latency as well as be a burden to devices.

And in the figure 8 shown below they showed that as increase in size of the graph there is not much overhead for the same.

Future Work

In the given implementation we parse the graph as soon as the browser starts which is static, while this graph could be dynamic and can be machine learn over the sessions to dynamically update the probabilities of the node and the page load time corresponding to each node which might result in better performance.

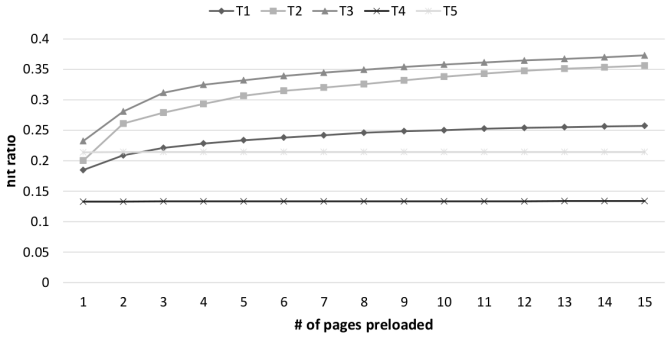

Figure 7: The effect of preloading degree

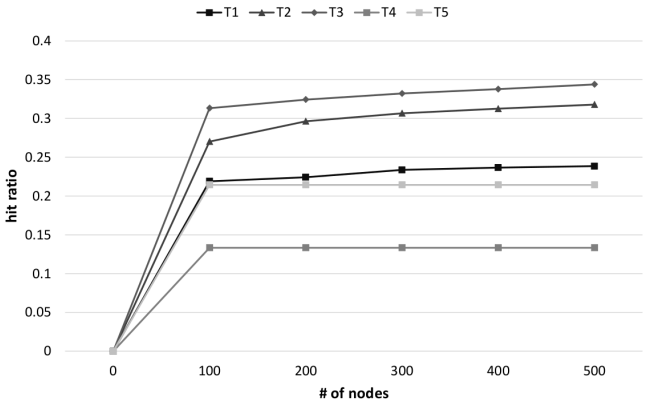

Figure 8: The effect of graph size for preloading