Visualizing large page mappings

A tool to understand memory mappings of a program

Robin Olsson
Abstract

Visualizing large page mappings

Robin Olsson

Larger page file sizes were introduced to increase the TLB’s reach. Although using large pages might improve performance, it might also introduce performance degradation. With no available tools able to visualize how much large page allocation a program obtains and how a program handles large page mappings, the aim of this thesis is to develop such a program. The final result is a monitoring program, in an alpha-state, built in Python. It starts a program to monitor and uses several Linux kernel commands to gather page file data for the memory mappings of that program. In this thesis I discuss the inner workings of this monitoring program, its additional graph-building module, as well as how it can be used to monitor if and how a program benefits in using large page mappings and how much the frequency of large page promotion affects performance. I found that by using this monitoring program, in conjunction with the Linux command ‘time’, and the graphs produced by its graph-building module, it was possible to see why some programs benefited by different promotion frequencies. In short, the monitoring program can help users decide if large page support should be enabled, as well as optimized for the use of large pages.
Acknowledgements

I would like to thank my supervisor, Chang Hyun Park, for the encouragement and guidance throughout the entire thesis. His help has been invaluable.

I would also like to thank my subject reviewer, David Black-Schaffer, for his support and help in making sure this thesis is on the correct level.

I would like to thank my parents for their endless support and encouragement with me deciding to quit my job, and start studying again. Without their support, I would not have been able to start, or study for this long.

Finally, I would like to thank my lovely girlfriend who have continuously motivated and encouraged me throughout my studies. She has given me the energy to keep going, even when she herself have had it rough.
1 Introduction

Large page mappings were introduced to help mitigate the disparity between the scaling of the TLB and the growth of system memory[1]. However, since large pages must be mapped in contiguous memory, through the lifetime of a process fragmented memory may prevent large page mappings, resulting in degradation of performance. The kernel will try to address this and generate large page mappings by defragmenting in the background.

Although large pages have the possibility to improve performance, users have reported high CPU utilization and latency spikes while using large pages, resulting in recommendations for large page support being disabled[2][3][4].

With performance benefits of using large pages being application specific, being able to visualize it is highly beneficial, helping users to get a deeper understanding of how much large page allocation a program obtains and how a program handles the use of large pages. Currently there are tools available to visualize 4kB page mappings, but not large page mappings. Since there are no tools available, the aim of this thesis is to build such a program with the goal of using it to both monitor the changes in large page mappings and gather data showing how pages are promoted to large pages, visualized through the use of graphs.

The monitoring program that was developed can visualize both 4kB pages and huge pages, and does so by reading the per-process page tables, gathering the necessary data needed to differentiate between different types of pages, explained in section 2.2 and 3.2. It then visualizes the data, making it easier for the user to follow the changes of the program during execution, further detailed in section 3.4.

The monitoring program’s performance and utilities are evaluated in section 4. Furthermore, in section 4.3 and 4.4, I evaluate the performance a program can experience if using large pages, how the monitoring program can help understand how the performance is impacted with large pages enabled, and also how the results were not fully in line with expectations and the reason as to why.
2 Background

2.1 Huge Pages

The capacity of system memory has been continuously increasing whereas the Translation Lookaside Buffer (TLB), a part of the memory management unit that handles the translation of virtual memory addresses to physical memory addresses, has not followed the same growth. With programs’ memory usage growing increasingly in size, this can induce a considerable amount of TLB misses. This is caused by a higher risk of a page frame number not being present in the TLB. This in turn results in performance loss since the processor now has to search the page table instead. A way to alleviate the amount of TLB misses is the use of larger page sizes, called Huge pages, instead of regular 4kB pages.

Huge pages are continuously mapped in a larger memory region where the size is configurable to ranges from 2MB up to 1GB. Basically, a Huge page with the size of 2MB can hold a mapping equalling 512 4kB pages while only taking up one entry in the TLB, resulting in the TLB reach becoming larger. However, for a program to make use of Huge pages it must be explicitly instructed to do so. In order to reduce the amount of work to make use of Huge pages, Transparent Huge Pages (THP) was introduced. While Huge pages are allocated at system boot, THP are promoted, compacting a number of 4kB pages into one THP depending on THP size, dynamically during run time through the use of the khugepaged thread in the kernel. When THP are enabled, khugepaged scans an amount of pages, at default 4096 pages, to check whether regions of memory can be promoted to THP. However, THP can be difficult to promote due to reasons such as memory fragmentation and immovable pages.

With applications containing everything from a few thousand pages to upwards of millions, trying to analyze how the promotion of THP occur with raw data not only becomes difficult, but also incredibly cumbersome. Thus abstracting data through visualization helps the user to analyze the data, as it is more easily understood, as well as it saves time. However, if the visualization is not properly explained, it can become too abstract resulting in a user not being able to fully utilize it.

2.2 Understanding Pagemap

The Operating System (OS) manages memory mappings by using page tables and other auxiliary data structures. Thus information about virtual to physical memory mappings are available in the per-process page tables. However, it is non-trivial for a user to get access to and decipher the OS’s page table. Therefore, to aid application developers, the Linux kernel provides an interface for users to query for such mapping information.

Introduced to UNIX systems in version 2.6.25, this kernel interface, henceforth called per-page meta data interface, can be found in the pseudo-file system /proc/, which provides an interface to kernel data structures for every Process ID (PID). The per-page meta data interface, which provides information such as whether a page is present in RAM or swap space and whether it is a THP or not, consists of four components:

• **pagemap**, which converts a Virtual Page Number (VPN) of a page into a 64-bit number, containing the Physical Frame Number (PFN) and flags such as if the page is swapped (bit 62) or if it is present in RAM (bit 63).

• **kpagecount**, containing a 64-bit value corresponding to the number of times the page has been mapped;

• **kpageflags**, which contains 64-bits of flag information for a page. Currently, as of kernel version 5.8.0-55-generic, there are only 25 flags in use (bit 0 to 25), with flags such as Dirty (bit 4), Buddy (bit 10), Swapbacked (bit 14), where the flag THP (bit 22), Transparent Huge Page, is of particular interest.

• **kpagecgroup**, containing a 64-bit inode number of the memory control group the page belongs to.

In order to use this per-page meta data interface, we must first gather the VPNs of the process we want to monitor. maps, a pseudo file located in /proc/PID/, contains the currently mapped memory regions, start and end of VPN mappings, of the PID. By reading pagemap at an offset calculated by dividing
the VPN by the size of a page file, usually 4kB, times 8, as a VPN contains 8 bytes of meta-data, we can gather the PFN for that VPN by reading the value of bit 0-54 through the bitwise operation \( \text{Pagemap-value} \& ((1 << 54) - 1) \). To check whether a page is present in RAM (bit 63) or swap space (bit 62), we do a bitwise shift-left and then a bitwise AND on the pagemap value (Pagemap-value \( << \) bitposition & 1). So to check whether the page is present in RAM, we do \( \text{Pagemap-value} << 63 \& 1 \).

In order to know whether we have encountered a THP or not, we need to read \( kpageflags \) with the PFN times 8 as an offset. Using the 64-bits number from \( kpageflags \), we do the same bitwise operation as with pagemap but use bit 22 to check if it is a THP or not. If we have encountered a THP we can jump to the next VPN that is 2MB away from the one we just checked. In conclusion, through the use of the per-page meta data interface, we can find out which physical page each virtual page is mapped to, whether it is present in RAM or in swap space, whether it is a THP, and more.

### 2.3 Pagemon

The design of the developed monitoring program is heavily inspired by Pagemon[13]. Pagemon is an interactive memory monitoring program that not only visualizes the mappings of a process but also gives the user the option to get a detailed view of a page if that is desired. However, Pagemon does not show large pages, which is a reason as to why this monitoring program is needed.

![Figure 1: Pagemon[13]](image)

In the leftmost side of figure 1 the VPN of the first page of each row is shown. At the top, the address which the cursor is pointing to can be found, as well as the current zoom of the middle window. In the middle part of the figure the pages are visualized. In the bottom part of the screen, instructions as to what each rectangle represents can be found. The rectangles, shown at the bottom, represent the following:

- **A**: This rectangle, shown as an 'M' in the middle part of the window, is either an anonymous page that is not backed by a file, or a page backed by a file.
- **P**: Represents pages that are present in RAM.
- **D**: Represents pages in RAM that are Dirty. A page becomes "dirtied" if it has changed from what is currently stored on disk.
- **S**: Represents pages that are in swap space.
- **•**: Represents pages that are not present in RAM.
3 Design & Implementation

To build a program that can not only visualize pages and store the page file data, but also keep track of both physical and virtual memory allocation, a few points need to be taken into consideration:

1. The program should be able to monitor a program by either being given a PID as an argument or start the program itself and fetch its PID. It should gather the number of 4kB pages, THP, total virtual memory allocated, total physical memory allocated, and total THP memory allocated.

2. If a program is large, there will be a large amount of page file information to handle. This requires that a data structure is chosen so that updates on specific pages can be efficient and easy to do.

3. With a large amount of pages that needs to be visualized, an efficient Graphical User Interface (GUI) should be considered.

4. During the run of the monitoring program, the data of every update should be stored into a data file so it can be used after the run for data analysis, as visualized data is more easily understood than raw data.

This program now exists under the name PageViz, developed by us, and will be further described in detail in section 3.1, 3.2, 3.4.

3.1 Overview of Top-level System Architecture

PageViz does not only need to be able to gather the data efficiently but also draw every page. It should also be able to start up a program to monitor and gather its PID so that it can track the changes of the program. Once the program being monitored has terminated, the data gathered should be stored into a json file, so that it can be used for analysis and to create graphs if so wanted. Figure 2 gives us the top-level overview of the system architecture, where the two modules 'Update pages', for gathering the page file data, and 'Update GUI', for drawing the pages, are the most important.

![Figure 2: Top-level overview](image)

PageViz starts off by starting the program it should monitor and fetching its PID. It then calls the 'Update pages' module and starts the 'GUI main loop', opening the main window. When the 'Update pages' module has finished gathering the data of the page mappings, which is explained in more detail in section 3.2, it sends the data to the 'Draw/Delete/Update pages' module. In this module, pages are either drawn as cells, deleted, or updated. This is explained in more detail in the section 3.4. When this module is done, it stores the data in a json-file under its current update number, updates the main window, and then calls the 'Update pages' module again. This continues until either the program being monitored is terminated or PageViz is closed.
3.2 Organization of the Data Structure

The 'Update pages' module uses the per-page meta data interface to gather information about individual pages, maps to gather the virtual addresses of each memory region the monitored program uses, and the command `ps` to gather the amount of virtual and physical memory allocated by the program.

The implementation of the data gathering module went through a lot of changes. In the beginning the thought was to create a class to hold all the information. However, as the class itself was written using lists, the time complexity of inserting and deleting a page was considerably high. This led to a lot of overhead as there are times when several hundreds of thousands of pages need to be deleted and/or inserted. The way the VPN ranges are iterated results in the data not needing to be sorted. The algorithm uses a while loop, starting from the start VPN and iterating with a variable page size until the final VPN of the range is reached. With this in mind, a hash-map, using the VPN as a key, could be used to mitigate the high time complexity of searches, inserts, and deletions. This resulted in the data structure being changed to a nested dictionary.

Dictionaries, consisting of key:value-pairs where the keys are unique identifiers and the value holds what is to be stored, is comparable to a hash-map\[14\]. Comparing lists and dictionaries, the time complexity of inserting and/or deleting an item is significantly lower, with the dictionary having a time complexity of $O(1)$, whereas lists have the time complexity of $O(n)$\[15\]. Using dictionaries instead of classes also meant that new key:value-pairs could be dynamically added.

In the outermost dictionary, the start and end VPN gathered from maps, for example '55a0c01d0000 -55a0c1d9000', was used as the key. This decreased the time to check whether a mapping needed to be added, deleted or updated. For example, if the end address would be set earlier (0x55a0c1d7000), the pages after this address would need to be deleted. The values of the outermost dictionary was a dictionary for every page in that memory region where the VPN of each page was set as its dictionary’s key, and the values was set to be a dictionary as well containing the following:

- **Frame**: The PFN the VPN is mapped to.
- **Present**: True if the page is present in RAM, else False.
- **Swap**: True if the page is in swap space, else False.
- **THP**: True if the page is a THP else False.
- **Changed**: True if the page was changed during this update else False.
- **Checked**: True if the page was checked at all during this update, else False.
- **XY**: The X and Y coordinates of the cell that is to be drawn.
- **Cell**: The drawn square if it had been drawn in a previous update.

Using this setup made it very simple to know whether a page needed to be updated or created, or whether the page was to be deleted which is shown in section 3.4.
3.3 Data Gathering

Following figure 3, the first thing the 'Update pages' module does is gathering the virtual address range from maps. The address range for each memory region is then checked whether it already exists in the outermost dictionary, shown in the red area in figure 3. If it exists, we just update the pages. If either the start or end address have changed, we need to either remove some pages, update pages that already exist, or add new pages into the dictionary. If the address range does not exists, the entire region needs to be created. When a new page is created, every key except 'Cell' is added. This works as a sign for the 'GUI module' to know whether it is a new page, and thus a new object needs to be created, or not. If the page have been updated, the key 'Changed' is set to True, which tells the 'GUI module' that the page has been updated. If a page was not present during the update and thus is to be removed, the key 'Checked' will be unchanged (False), which tells the 'GUI module' that the page and corresponding 'Cell' are to be deleted.

The updating or creation of a page follows the same path, shown in the green area in figure 3. For each VPN in the address range, which is iterated through a while loop, pagemap is read to gather the PFN. If the PFN is 0 the page is not mapped to memory, and thus the keys 'Present', 'Swap', and 'THP' are set to False, and if its PFN was something other than 0 before this update, its key 'Changed' is set to True. If the PFN is not 0, then we first check whether it is present in RAM or swap and then check if it is a THP. If it is a THP we update the step size to be 2MB instead of the default 4kB step size. Every page that have been checked will at the end have its key 'Checked' set to True. Once every address range has been checked, we call the 'GUI module' to update the window and possibly delete some pages.
3.4 Visualization of the Pages

Using Pagemon as inspiration, figuring out how PageViz would look like was quite straight forward. However, having rarely created a GUI, this was still a challenge. Finding an API that would be easy to learn was a big issue. Both PyQt5 and wxPython was considered in the beginning, but later discarded in favor of Tkinter, the standard GUI interface of Python, since Tkinter had more information and tutorials on how to set up compared to the other two.

The main window of the GUI is initiated when PageViz starts. It is then populated with cells, the colored rectangles, and VPNs, shown in figure 5 where each cell represents a page. This is done by the ‘GUI module’, Figure 4, receiving the dictionary containing all page mappings from the ‘Update pages’ module.

Looking at the red area of figure 4 for each page, if the key ‘Cell’ doesn’t exists, a new cell object needs to be created and stored into that page’s dictionary. If the key ‘Cell’ does exist, it checks whether the key ‘Checked’ is set to False. If it is, the cell and page are deleted from both the GUI and the dictionary since it was not present during the last update. If set to True, then it checks whether the key ‘Changed’ is False, the page is only moved to its X and Y position and the border colour of the cell is set to grey. If it is True, then we need to update the cell.

From here, both the cells that are to be updated and created follow a similar path. First it is checked if it has a PFN that is 0, shown in the green area. If it is, the colour of the cell is set to grey. If not, it is then checked whether it is present in RAM and whether it is a THP or not, shown in the blue area. If it is a THP, the colour of the page is set to yellow, and if not a THP but present in RAM, the colour is set to green. If it is not present in RAM or a THP, it is checked whether it is in swap space or not, shown in the pink area of the figure. If it is, the colour is set to blue, otherwise it is set to grey.

During the drawing and deletion of each cell, the number of 4kB pages or THP would be either incremented or decremented depending on the page type. At the end of the GUI update, the total amount of 4kB pages, THP, total memory allocation, V-memory allocation, THP memory allocation, and the current update number would be added at the bottom of the the GUI, as to help the user more easily follow the changes, as well being stored into a dictionary which at the end of the run would be saved into a json file.
As with the data gathering module, the GUI module went through a lot of changes. In the first version of the GUI, the window was cleared and each page was drawn every update. This lead to a lot of overhead and an unresponsive UI was experienced even at a few thousand pages. With dictionaries added, the three keys 'Cell', 'Checked', and 'Changed' helped mitigate some of the performance loss.
3.5 Finished Product

Figure 5: PageViz

Figure 5 shows a snapshot of how the finished product ended up. In the window underneath 'Cells (Pages/THP)', every green cell is a 4kB page that is present in RAM, grey cells are 4kB pages not present in RAM, blue cells are 4kB pages in swap space, and yellow cells are 2MB THP. Cells that have a red border around it have just been updated, and cells with grey borders did not change since last update. When a memory region has been drawn, the next memory region starts at a new row. This helps differentiate between the different memory mappings of a process.

In the leftmost window, underneath 'V-Addresses', the address shown is the first VPN of the virtual memory region that the OS have mapped for the process. A new VPN is added at the first row of the new virtual memory region.

The window at the bottom consist of the information regarding the memory allocations, the number of pages/THP, as well as explanations for the different coloured boxes.
3.6 Prerequisites & How to Run

3.6.1 Prerequisites

In order to run PageViz, the user must have root access. Without root access, pagemap will only return 0 when reading it for security purposes.

The user also needs to have the GUI package tkinter to be able to use the GUI. Tkinter, if not already installed, can be installed with the command:

```
pip3 install --user tk
```

Further more, if graphs are wanted, the module pageViz_plotter.py can be used. It grabs the stored data in `[benchmark]_data.json` and plots it. In order to use it, matplotlib must be installed. The graphs made in this report are made using pageViz_plotter.py.

Matplotlib, if not already installed, can be installed with the command:

```
sudo pip3 install matplotlib
```

3.6.2 How to Run

Before running PageViz, the kernel must be instructed to enable THP. This is done by changing `transparent_hugepage/enabled` to `always` if it is not already set to it:

```
sudo bash -c "echo always >/sys/kernel/mm/transparent_hugepage/enabled"
```

With THP now enabled, khugepaged will, at default, check every 10 seconds if THP should be promoted.

In order to speed this up, at the cost of CPU usage, the time between scan can be lowered with the command:

```
sudo bash -c "echo 100 >/sys/kernel/mm/transparent_hugepage/khugepaged/scan_sleep_millisecs"
```

This command instructs khugepaged to wake up every 100ms, increasing the amount of times allocation for THP are possible. It is possible to set the frequency of scans to 0ms, resulting in 100% utilization of one core.

If PageViz is run through a remote desktop, there is the possibility of the error: *Couldn't connect to display*. Running

```
xhost +
```

might solve this problem.

The way PageViz is currently coded, the programs to be monitored needs to be in a folder called 'benchmarks', which should be placed in the same folder as pageViz.py.

To run PageViz, use following command:

```
sudo python3 pageViz.py programToCheck.py args
```

Example:

```
sudo python3 pageViz.py graph500-graph500-2.1.4/omp-csr/omp-csr -s 21
```

where 'graph500-graph500-2.1.4/omp-csr/omp-csr' is the benchmark to be run, and '-s' and '21' are the arguments passed to the benchmark.
4 Evaluation

In this section I show the results that are produced by PageViz, the analysis, and further look into some performance anomalies I found along with solutions to remove the anomalies.

4.1 Methodology

The goal was to gather data from a large set of benchmarks to show how PageViz works and performs and as well as how the use of THP can affect different sets of workloads. However, numerous benchmarks from the Parsec Suite encountered an error while compiling or during execution, and could thus not be used. The following benchmarks were used to gather the data presented in the following sections:

- **Parsec Suite**[16]
  The Parsec Suite consists of several benchmarks to encompass a variety of workloads. All Parsec benchmarks able to run used the data set native.
  - **Canneal**
    Parsec Canneal has the most demanding memory behavior, with a huge working set and intensive communication[17]. Only the sequential version was used as the parallel version encountered errors during execution. It has a maximum memory allocation of roughly 700MB
  - **Blackscholes**
    Parsec Blackscholes is the simplest of all Parsec workloads with small working sets and negligible communication[17]. It was run both sequentially and parallelized, using all cores, with a maximum memory allocation of roughly 600MB.
  - **Freqmine**
    Parsec Freqmine identifies patterns in a transaction database and has huge working sets and some sharing[17]. It was run both sequentially and parallelized, using all cores, with a maximum memory allocation of roughly 650MB.

- **GUPS**[18]
  GUPS tests the speed at which a machine can update the elements of a table spread across global system memory. GUPS was run sequentially with the input '28 4000 500024', with a maximum memory allocation of roughly 2GB.

- **Liblinear**[19]
  Liblinear is a simple package for solving large-scale regularized linear classification and regression. Liblinear was run sequentially with the data set 'url_combined'[20], with a maximum memory allocation of roughly 4.5GB.

- **Graph 500**[21]
  Graph 500 consists of two kernels. The first kernel is a scalable data generator producing edge tuples containing the start vertex and end vertex for each edge for an undirected graph. The second kernel performs a breadth-first search of the graph. Graph 500 was run sequentially and parallel, using OpenMP, with the input '-s 21', with a maximum memory allocation of roughly 1.2GB.

- **simpleBench.py**
  simpleBench.py was created to have an easy to access benchmark that continuously allocates memory until a set amount was reached. It can be configured with '-d' for a "dynamic" run, which is switching between allocating and freeing, and '-x', where 'x' should be changed for either 1, 2, 3, or 4, for allocating 1-4GB of data continuously in a for-loop.

All data were gathered on Ubuntu 20.04.2 LTS, kernel 5.8.0-55-generic, with an HP Pavilion 15-bc007no laptop running an Intel Core i5 6300HQ and 8GB DDR4-2133.
4.2 Evaluation of PageViz

4.2.1 Data Gathering & Plotting

For each update, PageViz stores the total number of 4kB pages and THP present as well as the total amount of physical memory, virtual memory, and THP memory allocated to the program being monitored. In order to show how PageViz’s performance is hit by the amount of pages present, it stores the total time for each update.

Figure 6 and 7 were created by PageViz_plotter.py, a module for PageViz. This module was created to aid in the understanding of how memory is allocated as well as how many 4kB pages and THP are present at any update tick. By visualizing the data through graphs a user can more easily understand how a program changes throughout its execution, instead of trying to parse through several hundreds of raw data points.

The module produces an .svg vector figure composed of two graphs stacked on top of each other. Looking at Figure 6 as a reference, the top graph shows the number of THP as the blue line, with its corresponding values to the left, and the number of 4kB pages as the red line, with its values on the right hand side. The bottom graph consists of three solid lines; Physical, Virtual, and THP, corresponding to total physical memory allocation, total virtual memory allocation, and total THP allocation, with the values on the left side. The dotted line represents the recorded time of an update, which is the timed execution from the start of the ‘Update pages’ module until the end of the ‘Update GUI’ module, with the values on the right side. At the bottom of the graph, the update number, representing the update at which the values was recorded, is located. This was implemented to help find the correct snapshot of the PageViz window, which has yet to be implemented.
4.2.2 Visualizing the Promotion to THP

The following three figures are snapshots of the benchmark `simpleBench.py -2`, which allocates and fills a 2GB region through a for-loop and then sleeps for 30 seconds, and are used to explain both how PageViz visualizes the updates and what is happening.

In this snapshot, the data has been allocated and the benchmark is currently sleeping for 30 seconds. khugepaged is now working on promoting pages to THP. All the cells with red borders have just been updated, and a few yellow cells can be seen dotted around the grey cells. These are THP that have just been promoted, while the gray cells are either not determined or have not been allocated yet.

khugepaged has, in this snapshot, now both defragmented and promoted even more THP. This can be observed both through the visual representation of cells, where the yellow cells that have a red outline are the memory regions that have been promoted, and reading the Nr of THP(2MB) in the lower middle of the screen, showing an increased number of THP compared to previous snapshot.

Finally, nearing the end of the benchmark, khugepaged have found a last region of memory to promote to a THP. With this promotion, all THP are now in contiguous memory.
4.2.3 Performance of PageViz

One downside of PageViz is its performance when a lot of pages need to be drawn. Figure 8 shows the time between updates are roughly 15 seconds, which in turn reduces how often changes in THP promotion can be observed. The graphs in figures 8-11 show how PageViz gathers data throughout the benchmark Liblinear, running train with the data set url_combined.

![Figure 8: Liblinear - 10000ms (default)]
![Figure 9: Liblinear - 1000ms]
![Figure 10: Liblinear - 10ms]
![Figure 11: Liblinear - 0ms]

When a large amount of pages are present, update time increases. This correlation can be seen in the graphs, as the 'Update time' line closely follows the changes of the 'Nr Pages' line. Promotions of THP occur continuously during the execution in figure 10 whereas figure 9 shows THP promotion only during a few updates after 'Update Nr' 15. For all runs, almost the entire virtual memory allocation happen at around 'Update Nr' 15. Shortly after, the total physical memory starts to be allocated, with its maximum being reached after a few updates. When khugepaged scan frequency is increased, instructing how often khugepaged should wake up and check if pages should be promoted\[8\], the number of 4kB pages becomes fewer over time as more are promoted to THP. This helps PageViz run quicker, as there are fewer pages to draw, and Liblinear to complete the benchmark faster. However, having khugepaged constantly scanning is not suitable for all programs, which is demonstrated in the next section, 4.3.
4.3 Benchmark Performance Using THP & Different Scan Frequencies

The following graphs show the performance difference of different benchmarks with THP disabled and enabled with `khugepaged` at different scan frequencies. The Y-axes of the graphs correspond to the amount of time, in seconds, for each benchmark to complete, and the X-axes corresponds to the different values of `khugepaged` scan frequency and when THP support was disabled. Each of the benchmarks were timed with the Linux built in function `time`, recording the total time from start to the termination of the benchmark, with the first run having THP disabled, the second run with THP enabled and `khugepaged` scan frequency set to its default (10000ms) value, and the following runs having the frequency of `khugepaged` scan increased (1000ms, 100ms, 10ms, 0ms). The first graph, figure 12, contains the benchmarks that was only able to run sequentially, and figure 13, 14 and 15 contains the benchmarks that was able to run both sequentially and parallelized.

The benefits from using THP are very application specific, that is not all applications benefit from using them. Figure 12 and 13 shows a performance increase. In figure 12 going from left to right, both Parsec Canneal and Liblinear show a continuous increase in performance as the frequency of `khugepaged` scan is increased. Gaps_vanilla had the largest increase in performance when THP was enabled, where a performance increase of >3x was observed. However, increasing the `khugepaged` scan frequency did not show any prominent benefit unless set to 0ms. The graphs in figure 14 and 15 show no tangible benefits of using THP and instead show a hit to performance when `khugepaged` scan frequency is set to 0ms.

The data gathered did not entirely follow our expectations. I had two hypothesis that were proven incorrect by these results. The first hypothesis was that if a program were not to benefit from using THP, having `khugepaged` enabled could induce a hit to performance if the program utilized all cores as `khugepaged` would take up resources that would otherwise be available to the program. The second hypothesis was that if the program ran sequentially, then no matter the frequency of `khugepaged` scan the performance should be similar since it would be run on a different core, thus no there would be no loss in available resources for the program. This was however not the case, as shown in figures 13, 14 and 15 where a hit to performance, was observed when `khugepaged` scan was set to 0ms during its sequential execution, which is discussed in section 4.4.

<table>
<thead>
<tr>
<th>Figure 12</th>
<th>Figure 13</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Figure 12" /></td>
<td><img src="image2.png" alt="Figure 13" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Figure 14</th>
<th>Figure 15</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Figure 14" /></td>
<td><img src="image4.png" alt="Figure 15" /></td>
</tr>
</tbody>
</table>
4.4 Impact of Using Intel Turbo Boost

As mentioned in section 13 if a program is running sequentially, then having *khugepaged* enabled should not impact the performance as it is running on a different core. This was not the case in figure 13 and 15. The reason was that Intel Turbo Boost 2.0 was enabled. Intel Turbo Boost 2.0 boosts the maximum frequency of a processor’s cores dynamically depending on the workload[22] and the amount of cores currently working. The amount is decided by how much headroom the processor have left before hitting either the power or temperature limit. When the frequency of *khugepaged* scan was increased, the processor would boost the core working on that thread more often, effectively reducing the maximum boosted frequency of the cores. This is especially noticeable when *khugepaged* scan is set to 0ms, resulting in the cores in use having a lower boosted frequency throughout the benchmark.

With Intel Turbo Boost 2.0 disabled, the benchmarks run in figure 12,15 were rerun. The new data gathered showed no performance hit to sequential runs of the benchmarks when *khugepaged* scan was set to 0ms, and instead showed a small performance increase for some benchmarks compared to earlier runs, shown in Gups *vanilla* in figure 16 and Graph500 in figure 17.

Comparing the benchmark results of *Liblinear* in figure 16 with the graphs in figures 8,11, plotted with data gathered through PageViz, I observe that as the frequency of *khugepaged* scan is increased, more 4kB pages are promoted to THP. With *khugepaged* scan at 0ms, the majority of pages are being promoted to THP at around update 28. Furthermore, comparing the results of Parsec Blackscholes in figure 14 with the graphs 20,23 I observe that the majority of the pages are promoted to THP at the beginning of the benchmark, between 'Update Nr' 10-15, when the data is loaded for all scan frequencies. This suggest that the majority of pages able to be promoted to THP are done so in the beginning, which can be observed through the 'Nr Pages' line getting close to zero.

In conclusion, by using PageViz, in addition to monitoring the promotion and demotion of THP, it can be possible to observe if and how a program benefits from an increased *khugepaged* scan frequency. But in order to make sure consistent results are gathered, turning of any boosting technology is recommended.
Figure 20: Blackscholes - 10000ms (default)

Figure 21: Blackscholes - 1000ms

Figure 22: Blackscholes - 10ms

Figure 23: Blackscholes - 0ms
5 Conclusion & Future work

5.1 Conclusion

PageViz was created to address the lack of visualization tools for THP, and in that regard it was a success. By using PageViz, developers can get a better insight into how the kernel handles an application and as a result have an opportunity to optimize it for THP use if beneficial.

It is, however, not optimized and thus runs quite slow when a lot of pages are to be drawn. This could be addressed through better algorithms or having a better API for drawing the cells, with the possibility to use a GPU to alleviate the processor thread. There are a few bugs that needs to be addressed, with the three most noticeable being: PageViz sometimes stops entirely with it seemingly stuck in an infinite loop; empty spaces can show up in the middle of a memory region; and V-Addresses being placed in the wrong row or not at all. Two possible reasons as to why this is happening is the address regions are changed and thus are deleted, or the demotion of a THP.

In conclusion, as shown in the evaluations in section 4.4, using THP can be highly beneficial since it tends to lower the amount of TLB misses. Instructing khugepaged to run at a lower period tends to increase performance of an application if it runs on only one core and benefits from using THP. If the application is utilizing all cores, then having the frequency of khugepaged scan set to a minimum of 10ms tends to be beneficial. Any lower and there is a risk of performance loss.

One downside of the data gathered is that it was only gathered on my laptop, an HP Pavilion 15-bc007no, with an Intel Core i5 6300HQ and 8GB DDR4-2133, thus limiting the benchmarks to only run at around 6GB of memory allocation. And on the topic of benchmarks, most failed to either build or run. The benchmark Parsec Canneal was only able to run sequentially, shown in figure 12 and 16. Since this benchmark is the most TLB intensive out of the Parsec suite, not being able to run it on multicore, to especially observe whether it would benefit from having khugepaged scan frequency set to 0ms, is unfortunate. The majority of the benchmarks from the Parsec suite were not able to be built. A possibility as to why is that the latest release of Parsec, Parsec 3.0, was released as far back as 2011 but dependencies, such as gcc, has been continuously updated. Regarding Intel Turbo Boost 2.0 impacting performance, there is a high possibility of the boost technologies employed by other manufacturers, such as AMD and ARM, having a similar result. With the data only gathered on an Intel Core i5 6300HQ, this remains unanswered. However, in order to get consistent results, turbo boosting technology may need to be turned off to ensure that data gathered are not influenced by boosting technologies.

There is the potential risk that data got into the page cache after the first run, and thus sped up the benchmark. This was tested, by rebooting the laptop after each run, for the benchmarks Parsec Canneal and Liblinear. No difference in results could be found for these two, but there is still a risk of it affecting the other benchmarks.
5.2 Future Work

*PageViz* is by no means fully completed, and should be seen as being in an 'alpha' state. There are a lot of repetitive code that can be factored out, resulting in a cleaner and more readable code. With optimizations and through the use of better algorithms and an API that utilize the GPU, the speed of *PageViz* can be significantly improved.

Another optimization would be to make *PageViz* only draw the pages visible in the window, as this would lower the time spent on drawing a considerable amount. *PageViz* currently shows 4kB pages and THP as equally sized rectangles. Having THP cells at least two times the size of the cells representing 4kB pages would make it them more easily distinguishable.

On the leftmost side of figure 5 in section 3.5, only the start VPN of each address range is shown. This could be changed to show the entire VPN range of a row instead, which would further help distinguish between the mappings.

Implementing an option to run *PageViz* without the graphical interface is of high interest. This would result in data being able to be gathered more frequently, enabling graphs of a higher resolution. With this implemented, it would be easier to see small changes in THP promotions.

The way *PageViz* is coded right now is not very user friendly. The user needs to place the programs to be monitored into a folder *Benchmarks* which has to be located at the same directory as *PageViz.py*. However, having the user input the entire path as an argument can be cumbersome, and thus a suggestion would be to have the option of starting the application to monitor through the window of *PageViz*. Also, if a user wants to monitor a process that is already running, the user would have to terminate the process and then restart it through *PageViz*. Instead, giving the user the option to monitor a process through its PID would increase the user friendliness of *PageViz*.

With THP currently only being available on Linux systems, *PageViz* was developed with only that in mind. However, if THP are implemented into other OS’s, that is providing a similar feature as *pagemap*, updating *PageViz* to be OS-agnostic should be fairly straight forward.

*PageViz* is released to github as an open-source project. It can be accessed through the link: [https://github.com/zapfire88/PageViz](https://github.com/zapfire88/PageViz)
References


[17] Martí Torrents Lapuerta
   The Parsec Benchmark Suite
   Jan 27, 2012

[18] GUPS
   https://github.com/alexandermerritt/gups

[19] Liblinear
   https://github.com/cjlin1/liblinear/tree/v230
   Data set: https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/

[20] Liblinear data sets
   https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/

[21] Graph 500
   https://github.com/graph500/graph500/tree/graph500-2.1.4

[22] Intel Turbo Boost 2.0