# Understanding Memory Access Behavior for Heterogeneous Memory Systems

*Saikat Sengupta*

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science.

**Thesis Committee:**

Dr. Prasad Kulkarni: Chairperson

Dr. Perry Alexander

Dr. Jerzy W. Grzymala-Busse

Date Defended

The Thesis Committee for Saikat Sengupta certifies
That this is the approved version of the following thesis:

**Understanding Memory Access Behavior for Heterogeneous Memory Systems**

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

# Acknowledgements

I want to thank my advisor Dr. Prasad Kulkarni for advising me and mentoring me throughout this whole process and supporting me at times when the results were not always positive.
I also want to thank my parents for being my constant support throughout my whole life and my academic career.

# Abstract

Present day manufacturers have invented different memory technologies with distinct bandwidth, energy and cost tradeoffs. Systems with such heterogeneous memory technologies can only achieve the best performance and power characteristics by appropriately partitioning process data on OS pages and placing OS pages in the right memory areas. To achieve effective data partitioning and placement we need to first understand how programs access memory and how those patterns change at various stages (phases) of program execution. The goal of this work is to build a framework, design experiments and conduct analysis to understand overall memory usage patterns across many programs.

We use Intel's Pin dynamic binary translation and instrumentation system for this work. Our Pin based framework instruments programs at run-time to collect data regarding memory allocations, de-allocations, reads and writes, which we then analyze using our specialized scripts. We collect and analyze information including page access counts, hot page ratio, memory read and write access patterns and how that varies in different program phases. We also analyze the similarities regarding memory behavior between distinct phases during program execution. We also study memory behavior both with cache and without cache to understand how caches affect the memory access behavior.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Many new memory technologies and packaging strategies are now available that offer distinct advantages and tradeoffs to the conventional DRAM memory technology and placement. Newer storage-class non-volatile memory technologies such as spin-torque transfer (STT) RAM, phase change memory (PCM), and resistive RAM (ReRAM) provide byte-addressable storage and lower energy consumption than DRAM since they do not need periodic refresh. However, they may suffer from poorer read/write latencies, lower bandwidth, and durability as compared with DRAM, especially at launch. Manufacturers have also invented memory technologies that offer orders of magnitude higher bandwidth compared to currently conventional memory systems. High-bandwidth memory (HBM) technologies include AMD's HBM interface for 3D-stacked DRAM, Intel and Micron Technology's Hybrid Memory Cube RAM interface and MCDRAM [11,15]. While these technologies can provide fast memory access, they currently support restricted storage capacity.

Several related trends are driving the push towards heterogeneous or hybrid memory (HM) systems that incorporate multiple distinct memory technologies.

First, physical limitations are throttling the scaling in conventional DRAM density and cost [11, 18] even as modern applications like data analytics [5, 22] and key-value stores [2, 14] with large and growing datasets are demanding exponentially expanding memory capacity to realize their performance goals [4]. Second, many data-center and desktop applications are now demanding increased memory bandwidth with lower latency [6] than can be provided by conventional DRAM and may require high-bandwidth memory that is located closer to the processor, which along with cost limits its size [20]. Third, several researchers posit that the best power-performance-cost factor with modern applications and computer systems can only be achieved by combining multiple memory technologies in a heterogeneous configuration [16].

Achieving the best power and performance characteristics on next-generation desktop and server class computers will require a combination of: (a) hardware that integrates multiple classes of memories in the same system, and (b) system software that can effectively exploit the underlying hardware for each application. For instance, a system build with Intel's Knights Landing processor can combine fast MCDRAM, conventional DRAM and non-volatile PCM memories. It is equally important for system software, including the compiler, the user-level runtime system and operating system, to have the ability to appropriately locate the application's data objects on OS pages and then partition and place the OS pages on the right memory systems to achieve the best performance and power benefits.

To realize the best performance and cost tradeoffs, hardware systems should be designed with the right mix of memory technologies and capacities. Likewise, software systems need to understand the memory behavior of typical applications

(for a certain domain) to develop a general model for memory allocation policies and data movement across memories. However, application memory usage are still not understood well enough to build such models and policies for program data partitioning and hardware memory placement. The goals of this work are to achieve such understanding by: (a) developing a framework to explore memory allocation and usage behavior of software, and (b) employing this framework to study and make observations regarding the memory behavior of computer intensive benchmarks in SPEC cpu2006 [7].

We build our framework to explore memory usage properties of programs on top of Intel's *Pin* dynamic binary translation and instrumentation system [12]. Pin allows us instrument programs at run-time to collect data regarding memory allocation and deallocations and all memory reads and writes. This data is then analyzed to make observations regarding the general characteristics of memory usage and behavior across one important class of programs. We study the memory behavior both with and without *caches* to better understand how their presence affect these memory usage properties.

The following is the outline for the rest of this thesis. The next chapter describes the tools used in this work along with our experimental configurations. Chapters 3 and 4 presents our results and observations from experiments that use cache and no-cache configurations respectively. We discuss related works in Chapter 5. Finally, we present directions for future work and our conclusions in Chapter 6.

# Chapter 2

# Tools and Experimental Configuration

Pin is a free tool provided by intel which can be used by the programmers as a program analysis tool. We can write our own pin tool using this software which can be used to examine a certain program's behavior. Pin also provides a record or replay toolkit called Pinplay which is used to capture the execution of a program, record it as pinballs. These pinballs can be replayed which should provide the same program behavior that has been recorded while running the program and can be used further to determine the program behavior, memory access, cache profiling and so on.

In our experiment we have used Pinplay to record the program behavior of different programs including the SPEC benchmarks to generate the pinballs. Then we have used these pinballs to generate the memory traces for each of the programs while its execution for different phases. To divide the program execution for different phases we have used the concepts of Simpoint. The integration of

this simpoint with pinplay tool has been named as pinpoints and we have used these pinpoints to analyze our program behaviors for large applications.

Phase classification while the program executes is one of the primary step that we followed while performing our experiment. We divided each of the benchmark execution in five phases consisting of 100000000 instruction for each phases. The use of phase classification is to detect the program behavior changes for different stages of the phases. A program execution is not random and it shows repeating behavior in different stages if examined perfectly. That was the main use of using Simpoint which is able to identify these similar behavior of two different phases and our experiment included the analysis of the data that we collected over time.

Our experiment follows the below mentioned steps in order.

- Generating the Pinpoints for different phases (in our case we have found the statistics for 5 phases).
- For these pinpoints we have found the memory traces while execution of the SPEC benchmarks. We have written a pin tool which is able to generate these memory traces.
- The pinpoints are created by the Pinplay record/replay toolkit provided by Intel. Once the traces has been recorded in the pinpoints, it can be replayed as many times as the user wants and they can generate the memory trace files each time for different phases.
- These memory trace files are then analyzed using a script to find the following statistics.

    * The distinct number of page accesses for different phases.
    * The number of access for each of the individual page which accessed the memory.
    * The number of distinct memory address access for each of the pages.
    * The number of hot pages for each phases. Hot pages are those which constitutes 90 percent of total memory access.
    * Amongst these hot pages we also found the access pattern for each of them. Access pattern includes the identification whether the pages the read dominated or write dominated for each phases.

- We also integrated a 3 level cache simulator to our program and found the memory traces after it access the cache and indicated the different behavior of phases after the same list of analyses.

- Based on these statistics we have provided the data for the similarity metric, weighted similarity metric between the phases in a charted list.

- We have also provided some graph for visualization of the statistics in a better way. The graphs includes the pages access count, distinct memory access count and a chart graph to generate the hot page vs total page ratio throughout different phases for all benchmarks.

**Generating Pinpoints:** Pinpoints is using a combination of Pin and Simpoint. Each phase is a pinpoint that can be replayed by the user to generate the statistical data. We need to use a configuration file with the command to run the program to generate the files.

**Generating Trace files:** For each pinpoints we have generated the memory traces for each of the phases. For this we have used the replay framework of pinplay. We have written a pin tool using C++ which is able to find the memory traces both for read and write access.

**Analysis:** Now, we have written a script in C++ which can analyze the memory traces generated by pinplay. Our analysis file contains the following details.

- Number of distinct pages for each phase. Each page size is considered to be 4k.
- We have found count of access for each page.
- Number of distinct memory access by each page.
- Number of pages that constitutes 90 percent of memory access.

We have used this analysis file to get the details of hot pages. We have considered pages as hot which consists of 90 percent of total number of access. From our analysis we have found out that if we are considering full memory trace without a cache the number of total access is a lot higher. Though the number of hot pages which consists of 90 percent of access is a varying number. For some benchmarks we have found that the number of hot pages is really less which indicates the program execution needed a lesser number of distinct page access. But the access count for individual pages is very high in such cases. We have plotted a graph for the page access count for each page for individual phases of program execution.

# Chapter 3

# Analysis with Cache Integration

**Total Memory Access Statistics :**

In our pintool we have used a 3 level cache which reduced the access to the main memory to a large extent. This consists of L1, L2 and L3 cache along with a TLB and then we are tracing the memory access that are being bypassed after they miss the last level cache. The trace files generated after running this tool are then analyzed to provide the details of the memory pages and access patters mentioned in the description.

As discussed previously, we have considered the slice size to be 1000000000 in the configuration file while running each benchmarks through pinplay. Each of the phases will have this many instructions and based on that we will have the memory traces in our output files which will later be used for analysis purpose.

When we are running our programs with a 3 level of cache integrated with it, the count of direct memory access will drop to a significant amount and that is what we can see in the plotted chart 3.1.

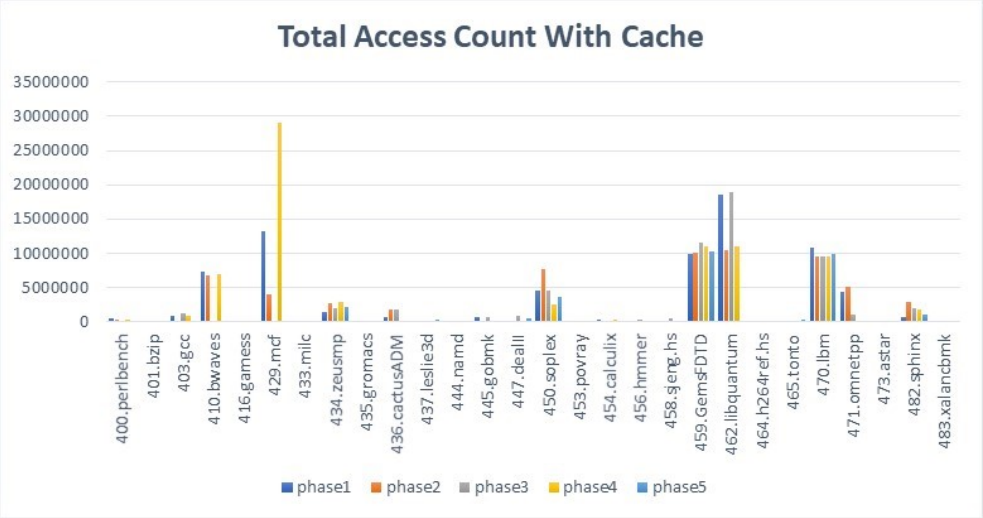


**Figure 3.1.** Memory Access Statistics

For most of the benchmarks we can examine that the access count is very low, even lower than 100000 instructions. Which tells us most of the instruction are processed by cache memory itself. We have 3 level of cache integrated with our tool which L1, L2, L3. In the cache, hit ratio is higher for most of the benchmarks which results in a lower cache miss rate and actual access to main memory percentage becomes low for these benchmarks.

For a few benchmarks namely, 429.mcf, 459.GemsFDTD, 462.libquantum and 470.lbm, we can notice the access count is much higher compared to the all other benchmarks. Among these 429.mcf and 462.libquantum are Integer benchmarks whereas 459.GemsFDTD and 470.lbm are floating point benchmarks. So, we can conclude there is no particular pattern for the execution of Integer or Floating point benchmarks in terms of cache access. Both of them varies significantly for various benchmarks and for different phases.

**Hot Page counts ratio :**

We have decided to term the pages hot which are comprising 90 percent of the total page access. We have run it for all the benchmarks and there has been some similarities among all of them. When we are using cache the page access count is moderately very low compared to the scenario when there is no cache. The hot page count access compared to the total number pages count access has a comparatively larger value in case of testing with the benchmarks. Here are some of the graphs to provide the idea of how the hot page count is varying for different benchmarks in each of the phases of program execution.

In this section we have provided the graphs that provides the general idea regarding how the Address count is changing for each pages number. In the provided graphs, the X-axis represents the Page number of memory access and Y-axis represents the access count corresponding to each pages. We have provided 5 different kind of graphs that were generated during the program execution phases in the next part.

4.2 kind of graphs are not very relevant during our program execution. Few of the benchmarks including 401.bzip2, 462.libquantum produces this kind of program behavior for phases where most of the pages have a very near count until the count reaches very low value for a few of the pages. That is why the curve has a sudden drop at certain point where the count becomes very low. This also provides us with the information that is many pages will be considered as hot pages in such scenario.

4.3 type of graphs are relevant for program execution behavior. As seen from the figure, it is seen there are very few pages which has a high access count. There

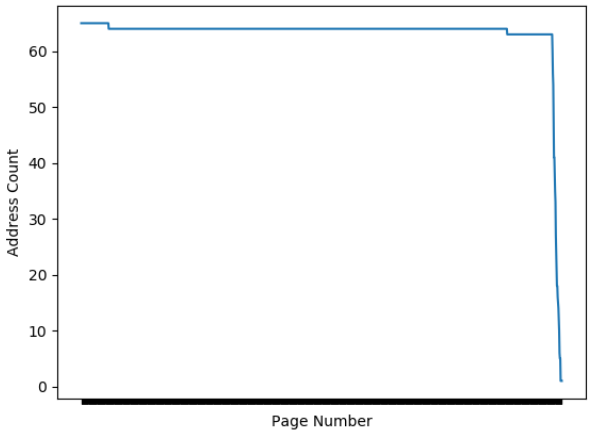


**Figure 3.2.** Type A : Hot Page count

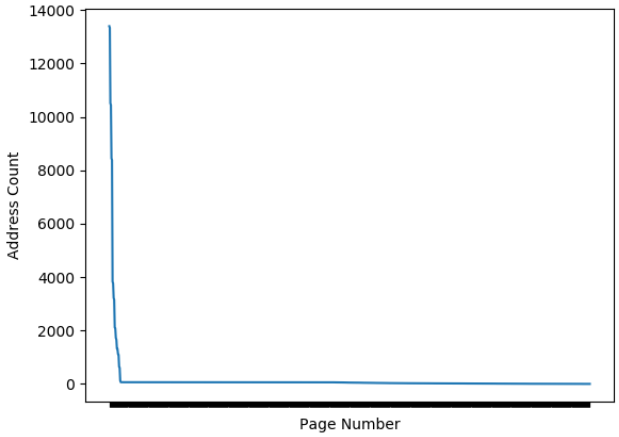


**Figure 3.3.** Type B : Hot Page count

might be about 2-3 pages which actually shows such behavior where the access count is very high and rest of them will have an access count that dropped rapidly.

As said earlier, this kind of graphs occurrence is very relevant and it might be seen for a few phases of most of the benchmarks. Here the hot page count will be comparatively lower.

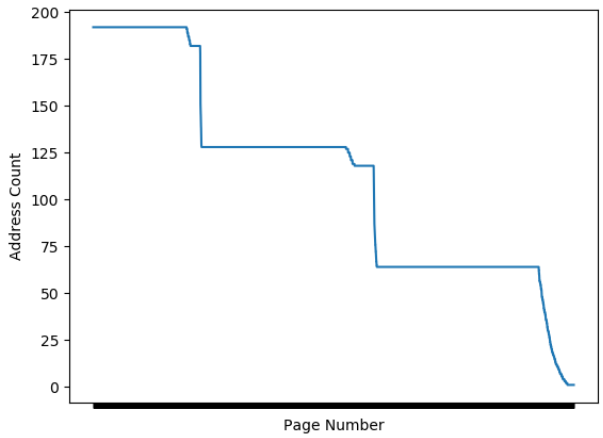


**Figure 3.4.** Type C : Hot Page count

4.4 kind of graphs are not very common though few benchmarks like 436.CactusADM, 410.bwaves exhibit this kind of program execution behavior. Here we can see the count has been varies continuously for different pages throughout the phase of the program execution. The figure is like a staircase which means for each of the level we can think the count are nearly constant and then the count drops significantly for the next step which are some new sets of memory pages, until it reaches lower value at the end.

Here the number of hot pages will mostly consist of the pages which are at mostly higher level. For such memory pages the count is relatively higher.

3.5 type of graphs shows a curve like behavior. This means the count for each of the memory pages are varying continuously. Many benchmarks including 403.gcc, 434.zeusmp, 450.soplex shows this type of page access behavior where the access count is changing continuously for the whole program execution.

3.6 type of graph behavior is very rare. Here the page count is similar for a few

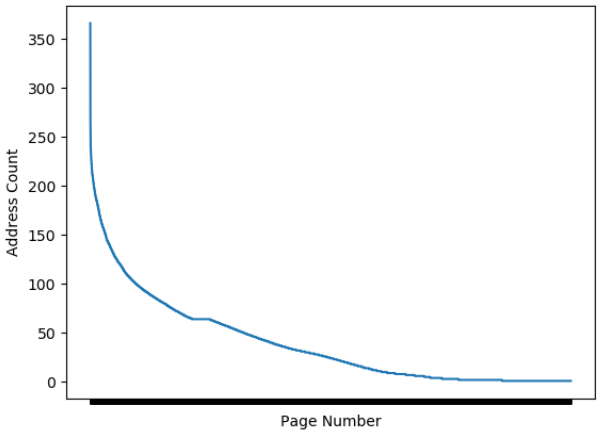


**Figure 3.5.** Type D : Hot Page count

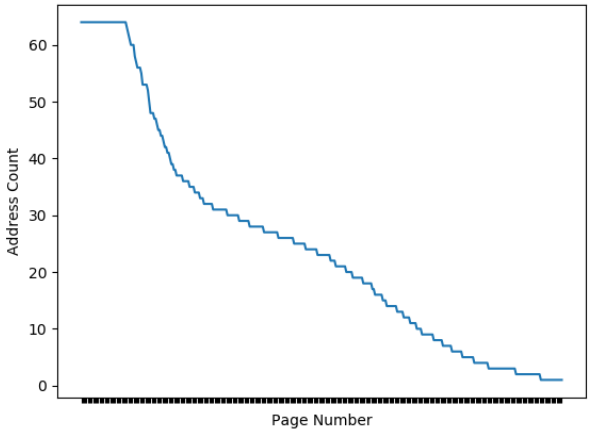


**Figure 3.6.** Type E : Hot Page count

pages and then it changes for the next few pages. It also resembles the curve like graph structure but it is slightly different.

Benchmarks like 453.povray, 473.astar shows this kind of program execution behavior.

The graphs are plotted based on descending order of page access counts for each of the hot pages. From the graph it is evident that the page access is not excessive for any of the phase which can take up most of the percentage of total access count in a particular phase. Rather, the page count is more moderate and almost similar for a majority of page number. From this observation we can say that when we are using a cache, we are encountering a majority of pages which can be considered as hot pages which has a count more or less same for the plotted graph.

For most of the benchmarks program execution behavior with cache memory consideration has a lower hot page count/ total page count ration as shown in the graphs.

Now, if we want to put the hot pages in the high bandwidth memory which can be used for faster access while a program is being executed, we will need to put most of the pages in such category and we will need a higher memory size to reach certain goal for that.

**Distinct Memory Address Accessed :** Along with the statistics of page access count, we have provided the details of distinct memory access count for each pages throughout different phases of program execution.

Let us take a look at some of the graphs that are plotted to depict the distinct memory count traces along with page number.

Now, from the graphs we can see that the number of distinct memory address

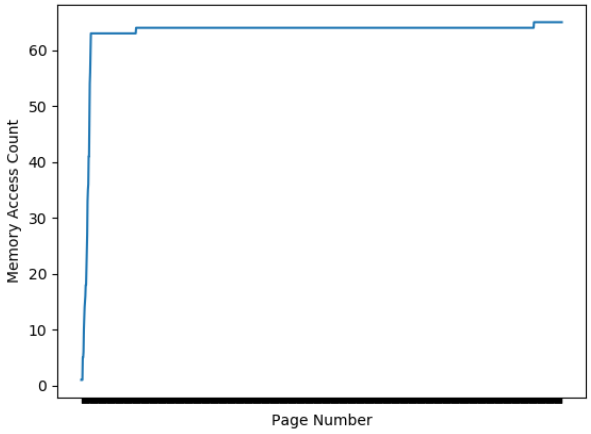


**Figure 3.7.** Type A : Distinct Memory Address Accessed

referenced by each of the memory page is varying for each of the curves. Our memory page has 4kb memory which means it can have as much as 4096 distinct memory address reference for each memory page. In our graphs we can see, in most of the cases it reaches around the border of 60-100 counts.

For Type A, majority of the memory pages reaches a count of 64. For type B we can see that there are very few pages consisting of a count as high as 1000 but then it drops significantly. For Type C, we get a curvy structure where the graphs is gradually increasing in behavior. For Type D, the curve shows a step

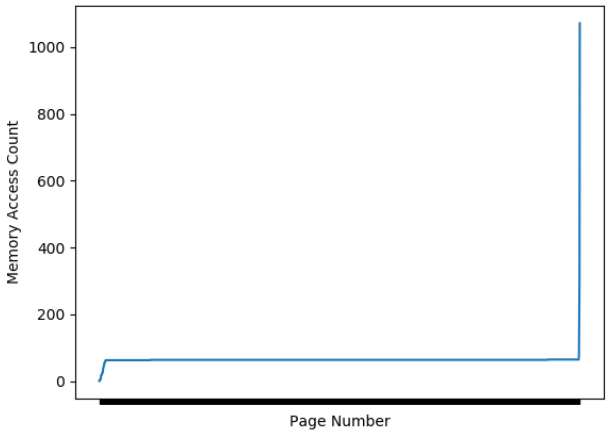


**Figure 3.8.** Type B : Distinct Memory Address Accessed

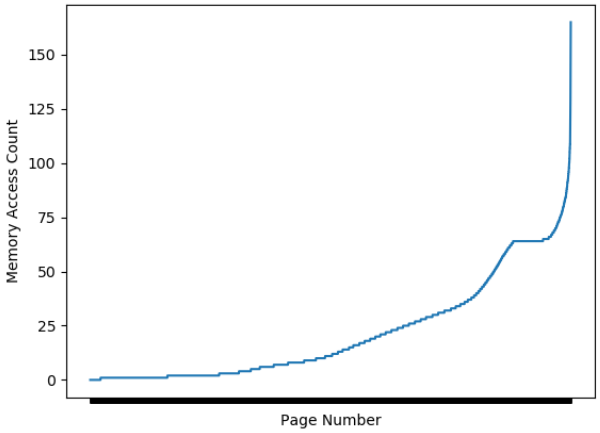


**Figure 3.9.** Type C : Distinct Memory Address Accessed

like behavior where few of the counts are same for a certain number of pages and then it again increasing.

This just shows that there is a pattern for distinct memory access count for the memory pages for each of the phases we have run for our benchmarks.

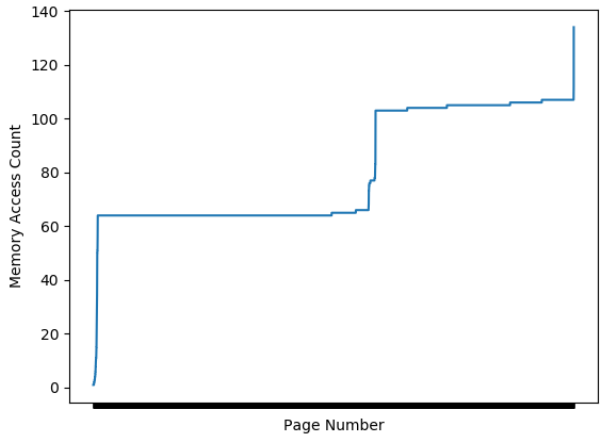


**Figure 3.10.** Type D : Distinct Memory Address Accessed

**Hot page / Total page ratio :**

Below is a chart that is providing the details of how the hot page/ total page ratio varies for different benchmarks in our experiment. We have done the plotting for each of the phases.

The x-axis is the name of the benchmark with the phases and y-axis is the plot of the ratio which must be a number lesser than 1. Each of the phases has been identified by different color. From the chart one thing we can notice that almost for all of the benchmarks the ratio is a bit higher. The ratio is reaching more than 0.5 for almost all phases of most of the benchmarks which means that half of the pages are being considered as hot pages in such cases. Hot pages are those which consists 90 percent of total page access count. That means if we want to details of 90 percent of the access count throughout each phases we have to consider 50 percent of the pages for each of the benchmarks which is a large quantity. As we have found in the previous part that the page access count is not very high when we are using a cache before the memory access. That is the reason behind such

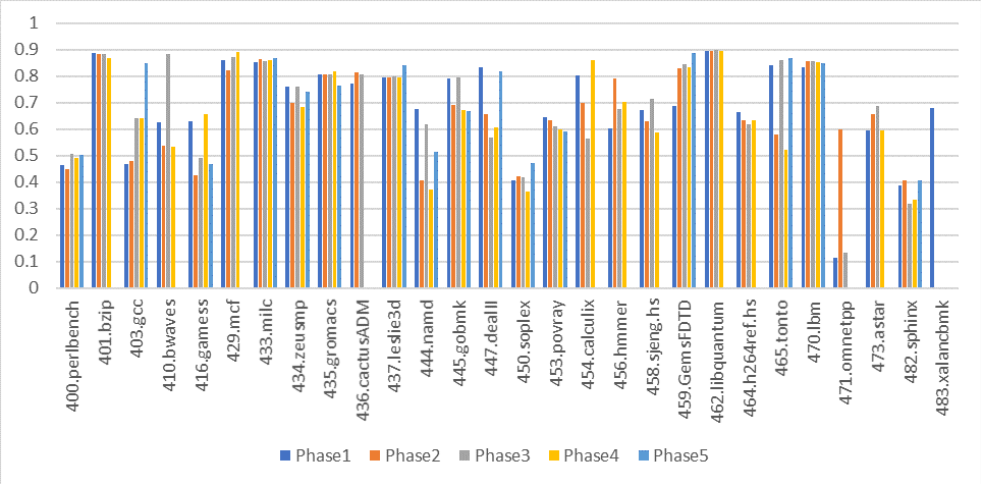


**Figure 3.11.** Hot Page count ratio for cache

a low ratio for such cases. When we take note of the page access count, most of the pages exhibit a similar behavior and the count is nearly same. So, to reach a percentage near 90 we would have to consider most of the pages as hot pages based on the statistics.

**Access Patterns : (Read-Write Access)**

Based on the hot pages statistics, we have tried to find the read and write access patterns for the memory pages. In this part of the analysis we tried to find the percentage of read or write dominated pages for each of the phases while program execution. For this, we are finding the ratio of read only pages count and total page count in every phases.

$$\text{Read-Access ratio} = \frac{\text{Read only page count}}{\text{Total page count}}$$

$$\text{Write-Access ratio} = \frac{\text{Write only page count}}{\text{Total page count}}$$

If any of the ratio is exceeding 90 percent, which means out of total count, 90 percent of the counts are read only, then we will be term those page as read only. Same way, if 90 percent of the counts are write access then those page are termed as write only.

After finding this statistics, the count of total number of pages which can be termed as read-dominated or write-dominated are found. For each phases we have found the ratio of read-dominated page count/total number of pages and write-dominated page count/total number of pages. This ratio for each phases has been plotted in the two separate charts for read-dominated and write-dominated access respectively. From the graphs shown previously, we can see that the ratio for read-dominated pages is larger compared to write dominated pages for most of the benchmarks. Majority of benchmarks follow an execution pattern where most of the memory access are read. There are few benchmarks like 444.namd, 458.sjeng.hs, 459.GemsFDTD, 462.libquantum, 471.omnetpp, 482.sphinx for which the write only memory access ratio is very low or almost 0. So, when we will be

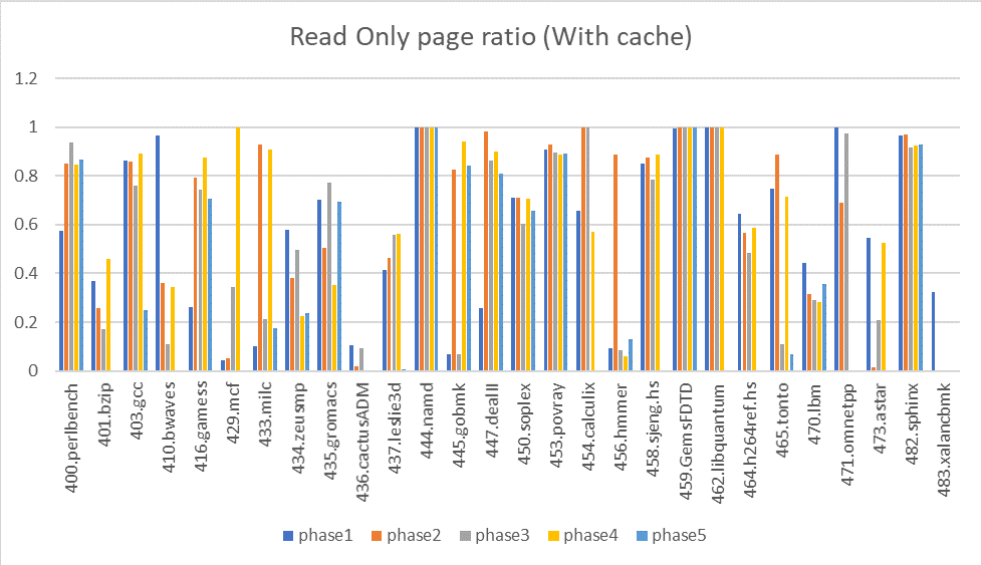


**Figure 3.12.** Read-Only Access

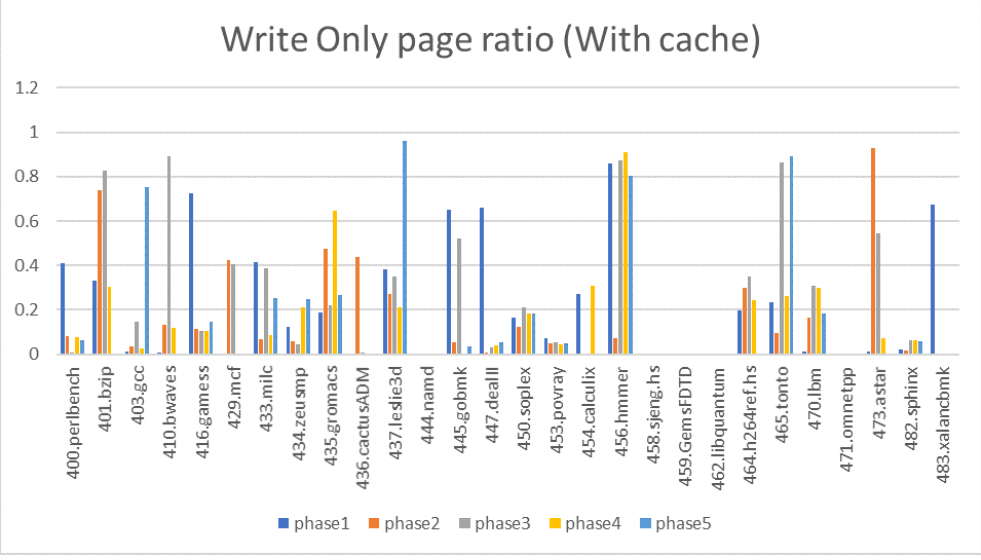


**Figure 3.13.** Write-Only Access

working with those benchmarks we can surely work with read-only memory patterns and it should not create a problem.

We can also notice from the graphs is there are few phases for a program execution which are read-dominated and the next phase is write-dominated. This is also an interesting criteria to work with. In 456.hmmer, we can notice all the phases except phase 2 is showing a write dominated access behavior which is quite interesting. A program behavior can change a lot based on access patterns during a program execution and that is evident from our findings. As we know in the DRAM technologies, both the read and write access takes place simultaneously but in technologies like STTRAM and PCRAM where read access is generally dominant we can use the read-dominated pages to construct the memories and it will make the access a lot faster for read dominated program execution.

**Similarities between the phases :**

As discussed earlier, the program execution does not change randomly over a period of whole execution time. It follows a particular program behavior that falls under same categories which are called program phases. We have used pinpoints which is similar to Simpoint but uses pin to generate the memory trace files for program execution. The phases are being considered as the set of intervals of a certain slice size where program behavior is considered to be similar. A phase can occur multiple number of times when a program is being executed.

In this section we have analyzed the similarities between different phases over the program execution time. As per our configuration file, we have recorded the program execution for 5 phases for each of the benchmarks.

We have used 3 ways to find this similarities between two phase which are mentioned below.

- **Similarity Metric :** This similarity metric is calculated for the hot pages for each of the phases over the program execution time. If we consider two phases named phase1 and phase 2 then similarity metric will be calculated like the equation below.

$$\text{Similarity Metric } = \frac{\text{Intersection of Number of hot pages in Phase1 and Phase 2}}{\text{Union of number of hot pages in Phase1 and Phase2}} \tag{3.1}$$

- **Weighted Similarity Metric for Hot pages :** Weighted Similarity Metric is calculated for the Hot pages by using the page count as the weight to calculate the similarity. If we consider two phases named Phase1 and Phase 2 then Weighted similarity metric for hot pages will be calculated like

the equation below.

$$\text{Weighted Similarity Metric (Hot Pages)} = \frac{\sum \text{Count of Intersection of Hot pages in Phase1 and Phase2}}{\sum \text{Count of Union of Hot Pages in Phase1 and Phase2}} \tag{3.2}$$

- **Weighted Similarity Metric for All pages :** Weighted Similarity Metric is calculated for all the pages accessed by using the page count as the weight to calculate the similarity. If we consider two phases named Phase1 and Phase 2 then Weighted similarity metric for all pages will be calculated like the equation below.

$$\text{Weighted Similarity Metric (All Pages)} = \frac{\sum \text{ Count of Intersection of all pages in Phase1 and Phase2}}{\sum \text{Count of Union of all Pages in Phase1 and Phase2}} \tag{3.3}$$

In the next part we have provided the tabular structure for different benchmarks to show the values calculated by the 3 equations mentioned above between all 5 phases.

We have also provided a scattered graph to get the idea of how the phase behavior changes for different phases for the benchmarks. In the scattered graph, each dot signifies the similarity ratio value between two phases of program execution.

The graphs are plotted below the table for each scenario. From the graph we can see that the similarity metric provides the values that varies significantly for different benchmarks. Even for different phases for the same benchmarks the similarity metric is changed rapidly for a few of the benchmarks.

| Benchmarks | Statistics between the Phases | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **1-2** | **1-3** | **1-4** | **1-5** | **2-3** | **2-4** | **2-5** | **3-4** | **3-5** | **4-5** |
| 400.perlbench | 0.013 | 0.008 | 0.014 | 0.012 | 0.616 | 0.712 | 0.682 | 0.634 | 0.71 | 0.694 |
| 401.bzip | 0.008 | 0.696 | 0.707 | NA | 0.036 | 0.001 | NA | 0.766 | NA | NA |
| 403.gcc | 0.209 | 0.26 | 0.187 | 0.051 | 0.081 | 0.093 | 0.012 | 0.558 | 0.011 | 0.007 |
| 410.bwaves | 0.563 | 0 | 0.567 | NA | 0.019 | 0.753 | NA | 0.037 | NA | NA |
| 416.gamess | 0.106 | 0.067 | 0.476 | 0.077 | 0.495 | 0.138 | 0.565 | 0.09 | 0.447 | 0.085 |
| 429.mcf | 0.596 | 0.009 | 0.764 | NA | 0.057 | 0.626 | NA | 0.008 | NA | NA |
| 433.milc | 0.596 | 0.588 | 0.597 | 0.761 | 0.33 | 0.998 | 0.439 | 0.329 | 0.753 | 0.44 |
| 434.zeusmp | 0.379 | 0.562 | 0.562 | 0.573 | 0.744 | 0.775 | 0.305 | 0.623 | 0.408 | 0.317 |
| 435.gromacs | 0.762 | 0.603 | 0.056 | 0.633 | 0.565 | 0.057 | 0.541 | 0.054 | 0.767 | 0.053 |
| 436.cactusADM | 0.451 | 0.48 | NA | NA | 0.867 | NA | NA | NA | NA | NA |
| 437.leslie3d | 1 | 0.999 | 0.998 | 0.401 | 0.999 | 0.998 | 0.401 | 0.999 | 0.401 | 0.401 |
| 444.namd | 0.034 | 0.108 | 0.015 | 0.032 | 0.127 | 0.52 | 0.419 | 0.1 | 0.151 | 0.366 |
| 445.gobmk | 0.275 | 0.915 | 0.257 | 0.336 | 0.293 | 0.496 | 0.539 | 0.248 | 0.346 | 0.408 |
| 447.dealII | 0.015 | 0.006 | 0.239 | 0.215 | 0.001 | 0.017 | 0.001 | 0 | 0.002 | 0.065 |
| 450.soplex | 0.433 | 0.371 | 0.437 | 0.433 | 0.439 | 0.272 | 0.309 | 0.329 | 0.325 | 0.397 |
| 453.povray | 0.87 | 0.717 | 0.601 | 0.581 | 0.7 | 0.604 | 0.597 | 0.84 | 0.783 | 0.83 |
| 454.calculix | 0 | 0 | 0.027 | NA | 0.046 | 0 | NA | 0 | NA | NA |
| 456.hmmer | 0.371 | 0.39 | 0.329 | NA | 0.297 | 0.304 | NA | 0.241 | NA | NA |
| 458.sjeng.hs | 0.536 | 0.572 | 0.531 | NA | 0.547 | 0.573 | NA | 0.526 | NA | NA |
| 459.GemsFDTD | 0.412 | 0.533 | 0.458 | 0.44 | 0.633 | 0.671 | 0.611 | 0.877 | 0.74 | 0.739 |
| 462.libquantum | 0.818 | 0.798 | 0.795 | NA | 0.796 | 0.792 | NA | 0.809 | NA | NA |
| 464.h264ref.hs | 0.23 | 0.356 | 0.329 | NA | 0.257 | 0.242 | NA | 0.502 | NA | NA |
| 465.tonto | 0.058 | 0.091 | 0.107 | 0.06 | 0.023 | 0.406 | 0.014 | 0.059 | 0.421 | 0.033 |
| 470.lbm | 0.781 | 0.868 | 0.791 | 0.841 | 0.717 | 0.732 | 0.709 | 0.795 | 0.822 | 0.763 |
| 471.omnetpp | 0.039 | 0.832 | NA | NA | 0.047 | NA | NA | NA | NA | NA |
| 473.astar | 0.063 | 0.177 | 0.088 | NA | 0.154 | 0.13 | NA | 0.074 | NA | NA |
| 482.sphinx | 0.475 | 0.569 | 0.593 | 0.638 | 0.403 | 0.53 | 0.531 | 0.643 | 0.561 | 0.647 |
| 483.xalancbmk | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |

**Table 3.1.** Similarity Metric with Cache traces

The similarities between a few of the phases has a high values as near as 1 which is the highest value possible. This means these two phases are totally similar and the number of intersected and union pages are absolutely same which is a rare phenomenon for our experiment. Weighted Similarity Metric for hot pages provides a very much similar graph as the previous similarity metric. The slight differences that are found due to the addition of count to calculate the weighted

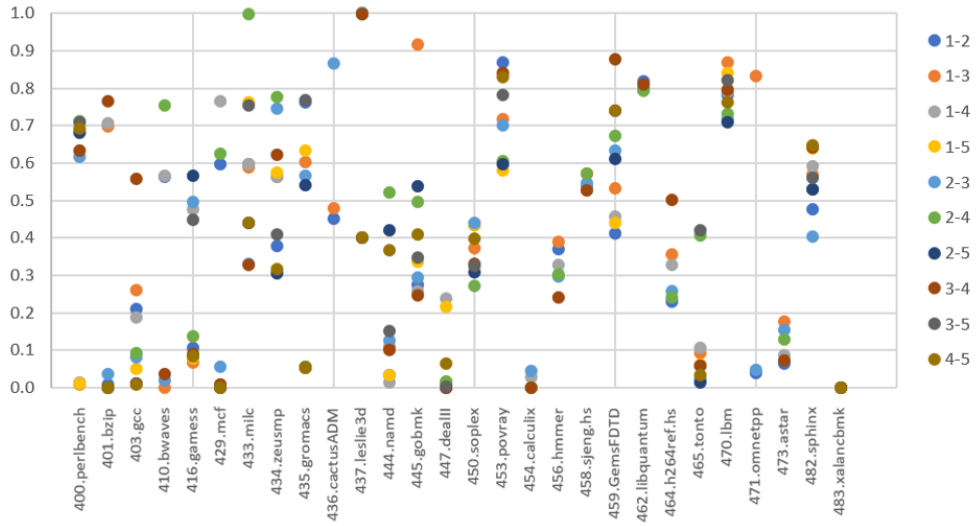


**Figure 3.14.** Similarity Metric Scattered Graph

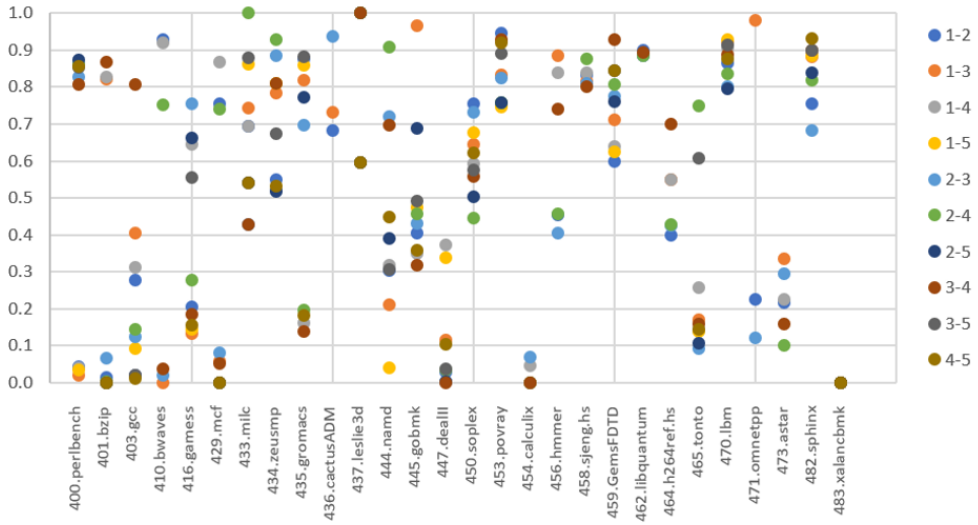


**Figure 3.15.** Weighted Similarity metric for hot Pages Scattered Graph

| Benchmarks | Statistics between the Phases | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **1-2** | **1-3** | **1-4** | **1-5** | **2-3** | **2-4** | **2-5** | **3-4** | **3-5** | **4-5** |
| 400.perlbench | 0.042 | 0.019 | 0.041 | 0.034 | 0.828 | 0.863 | 0.873 | 0.807 | 0.854 | 0.855 |
| 401.bzip | 0.015 | 0.82 | 0.828 | NA | 0.067 | 0.002 | NA | 0.867 | NA | NA |
| 403.gcc | 0.278 | 0.405 | 0.312 | 0.094 | 0.125 | 0.145 | 0.021 | 0.806 | 0.021 | 0.012 |
| 410.bwaves | 0.929 | 0 | 0.92 | | 0.019 | 0.753 | | 0.037 | | |
| 416.gamess | 0.206 | 0.132 | 0.644 | 0.143 | 0.754 | 0.277 | 0.661 | 0.185 | 0.555 | 0.157 |
| 429.mcf | 0.754 | 0.058 | 0.868 | NA | 0.082 | 0.741 | NA | 0.053 | NA | NA |
| 433.milc | 0.694 | 0.743 | 0.695 | 0.861 | 0.428 | 0.999 | 0.54 | 0.428 | 0.879 | 0.542 |
| 434.zeusmp | 0.55 | 0.783 | 0.521 | 0.809 | 0.886 | 0.929 | 0.518 | 0.81 | 0.673 | 0.531 |
| 435.gromacs | 0.876 | 0.818 | 0.162 | 0.86 | 0.698 | 0.196 | 0.771 | 0.14 | 0.883 | 0.181 |
| 436.cactusADM | 0.682 | 0.73 | | | 0.937 | | | | | |
| 437.leslie3d | 1 | 1 | 0.999 | 0.595 | 1 | 0.999 | 0.594 | 0.999 | 0.595 | 0.595 |
| 444.namd | 0.304 | 0.212 | 0.319 | 0.039 | 0.719 | 0.908 | 0.389 | 0.697 | 0.307 | 0.447 |
| 445.gobmk | 0.405 | 0.967 | 0.349 | 0.473 | 0.43 | 0.458 | 0.687 | 0.319 | 0.491 | 0.358 |
| 447.dealII | 0.028 | 0.117 | 0.374 | 0.34 | 0.004 | 0.035 | 0.002 | 0.001 | 0.039 | 0.104 |
| 450.soplex | 0.754 | 0.646 | 0.593 | 0.677 | 0.733 | 0.445 | 0.504 | 0.559 | 0.575 | 0.623 |
| 453.povray | 0.944 | 0.833 | 0.754 | 0.746 | 0.824 | 0.757 | 0.758 | 0.929 | 0.891 | 0.919 |
| 454.calculix | 0 | 0 | 0.047 | | 0.07 | 0 | | 0.001 | | |
| 456.hmmer | 0.453 | 0.884 | 0.84 | | 0.406 | 0.458 | | 0.739 | | |
| 458.sjeng.hs | 0.832 | 0.816 | 0.839 | | 0.808 | 0.877 | | 0.802 | | |
| 459.GemsFDTD | 0.598 | 0.712 | 0.638 | 0.624 | 0.776 | 0.808 | 0.76 | 0.929 | 0.845 | 0.845 |
| 462.libquantum | 0.9 | 0.888 | 0.885 | | 0.886 | 0.884 | | 0.894 | | |
| 464.h264ref.hs | 0.4 | 0.55 | 0.551 | | 0.426 | 0.429 | | 0.701 | | |
| 465.tonto | 0.142 | 0.169 | 0.259 | 0.138 | 0.093 | 0.749 | 0.106 | 0.16 | 0.606 | 0.145 |
| 470.lbm | 0.865 | 0.922 | 0.905 | 0.927 | 0.8 | 0.836 | 0.795 | 0.888 | 0.914 | 0.878 |
| 471.omnetpp | 0.227 | 0.981 | | | 0.122 | | | | | |
| 473.astar | 0.217 | 0.336 | 0.226 | | 0.294 | 0.101 | | 0.158 | | |
| 482.sphinx | 0.756 | 0.881 | 0.889 | 0.886 | 0.683 | 0.817 | 0.84 | 0.9 | 0.899 | 0.932 |
| 483.xalancbmk | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |

**Table 3.2.** Weighted Similarity Metric for Hot pages with Cache traces

similarity is very negligible. The graph for weighted similarity metric for all pages has changed a lot compared to the previous two graphs. The main reason of such changes is the consideration of all the pages that are accessed during the program execution. There are many pages which are not part of hot pages can be a part of the intersection set between two phases which will generally make the ration higher. Also, the opposite can happen as well where the intersection set is

| Benchmarks | Statistics between the Phases | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **1-2** | **1-3** | **1-4** | **1-5** | **2-3** | **2-4** | **2-5** | **3-4** | **3-5** | **4-5** |
| 400.perlbench | 0.234 | 0.166 | 0.227 | 0.206 | 0.902 | 0.889 | 0.913 | 0.873 | 0.903 | 0.888 |
| 401.bzip | 0.037 | 0.784 | 0.839 | NA | 0.103 | 0.019 | NA | 0.912 | NA | NA |
| 403.gcc | 0.407 | 0.711 | 0.614 | 0.121 | 0.125 | 0.145 | 0.021 | 0.806 | 0.021 | 0.051 |
| 410.bwaves | 0.943 | 0 | 0.943 | | 0.052 | 1 | | 0.051 | | |
| 416.gamess | 0.255 | 0.186 | 0.653 | 0.178 | 0.804 | 0.36 | 0.813 | 0.277 | 0.634 | 0.266 |
| 429.mcf | 0.904 | 0.053 | 1 | NA | 0.081 | 0.889 | NA | 0.048 | NA | NA |
| 433.milc | 0.732 | 0.887 | 0.733 | 0.894 | 0.577 | 1 | 0.604 | 0.578 | 1 | 0.605 |
| 434.zeusmp | 0.72 | 0.843 | 0.57 | 0.827 | 0.991 | 0.908 | 0.689 | 0.88 | 0.751 | 0.688 |
| 435.gromacs | 0.963 | 0.883 | 0.152 | 0.912 | 0.774 | 0.185 | 0.867 | 0.133 | 0.934 | 0.171 |
| 436.cactusADM | 0.852 | 0.857 | | | 1 | | | | | |
| 437.leslie3d | 1 | 1 | 1 | 0.66 | 1 | 1 | 0.659 | 1 | 0.659 | 0.66 |
| 444.namd | 0.357 | 0.315 | 0.329 | 0.118 | 0.736 | 0.928 | 0.5 | 0.709 | 0.392 | 0.574 |
| 445.gobmk | 0.524 | 0.993 | 0.431 | 0.622 | 0.544 | 0.567 | 0.783 | 0.401 | 0.648 | 0.461 |
| 447.dealII | 0.056 | 0.133 | 0.512 | 0.364 | 0.005 | 0.06 | 0.012 | 0.018 | 0.084 | 0.152 |
| 450.soplex | 0.915 | 0.876 | 0.804 | 0.833 | 0.845 | 0.766 | 0.809 | 0.873 | 0.86 | 0.81 |
| 453.povray | 0.991 | 0.874 | 0.83 | 0.827 | 0.869 | 0.829 | 0.825 | 0.968 | 0.949 | 0.973 |
| 454.calculix | 0 | 0 | 0.059 | | 0.123 | 0.001 | | 0.001 | | |
| 456.hmmer | 0.544 | 0.927 | 0.881 | | 0.465 | 0.536 | | 0.771 | | |
| 458.sjeng.hs | 0.944 | 0.962 | 0.939 | | 0.942 | 0.942 | | 0.926 | | |
| 459.GemsFDTD | 0.873 | 0.976 | 0.857 | 0.864 | 0.858 | 0.868 | 0.87 | 0.891 | 0.897 | 0.948 |
| 462.libquantum | 1 | 1 | 1 | | 1 | 1 | | 1 | | |
| 464.h264ref.hs | 0.416 | 0.646 | 0.587 | | 0.439 | 0.437 | | 0.799 | | |
| 465.tonto | 0.221 | 0.175 | 0.308 | 0.146 | 0.141 | 0.833 | 0.122 | 0.196 | 0.743 | 0.151 |
| 470.lbm | 0.998 | 1 | 1 | 0.999 | 0.997 | 0.997 | 0.996 | 1 | 0.999 | 0.999 |
| 471.omnetpp | 0.582 | 0.989 | | | 0.505 | | | | | |
| 473.astar | 0.238 | 0.38 | 0.365 | | 0.428 | 0.209 | | 0.236 | | |
| 482.sphinx | 0.89 | 0.954 | 0.955 | 0.948 | 0.853 | 0.933 | 0.954 | 0.959 | 0.962 | 0.987 |
| 483.xalancbmk | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |

**Table 3.3.** Weighted Similarity Metric for All pages with Cache traces

having a lower value and union set having a higher value making the whole ration to be a lot lower compared to the previous scenario.

We have also tried to find the similarity trends for the integer benchmark and floating point benchmarks separately to get any particular format, but that was not the case or finding. The similarity metric does not follow any particular

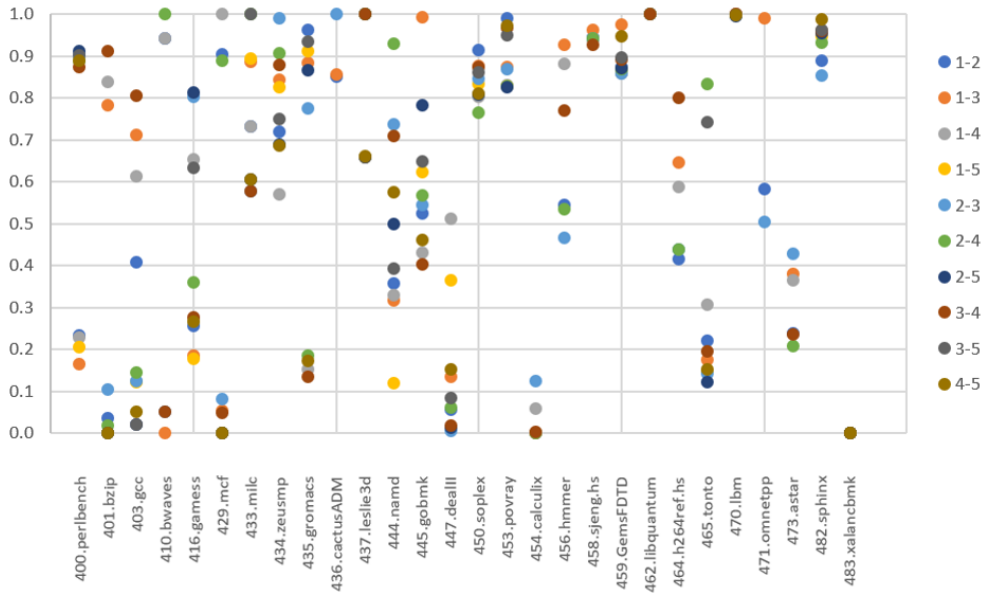


**Figure 3.16.** Weighted Similarity metric for all Pages Scattered Graph

pattern in our test case benchmarks. Rather the values that we received were quite random.

# Chapter 4

# Analysis without Integration of a Cache

**Total Memory Access Statistics :**

This is the statistics of the memory traces to the files without the consideration of a cache. As expected, the count of access in this scenario is a lot higher compared to the previous discussion with a cache. Our slice size was taken as 100000000 for each phases and we did the execution for 5 phases. For different phases we have plotted the access count for the instructions which is the main memory access in this case.

We can see the count is a lot higher in this case which is what we expected as there is no consideration of cache access. Another thing to note here is, the program execution through pin in this scenario is more compared to the scenario where we used cache. Also, for some benchmarks where the execution time is lesser and instruction count for execution is less, we have found lesser phases which were executed. For example, 401.bzip2 has 4 phases and 403.xalancbmk has 1 phase

compared to 403.gcc which has total 5 phases.

The chart provides us with a proper depiction which provides us an ideal about our data with which we have started our analysis. Among these we worked with the hot pages which are most significant for the analysis part.

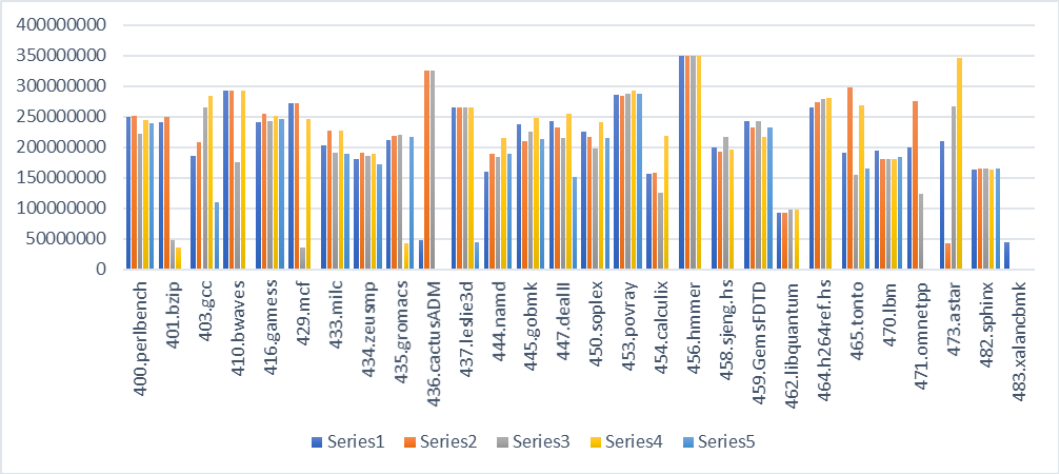


**Figure 4.1.** Memory Access Statistics

**Hot page count ratio :**

Below are a few graphs shown to provide an idea of how the page access count changes for each benchmarks while program execution without an integration of cache. Most of the curves generated shows a very similar behavior in this case compared to the previous section where we discussed about the program behavior without cache integration. The graphs has Page number in the X-axis and access count in Y-axis. From the graph we can interpret that there are only a few pages with a large access count which is nearly at 107 or 106 order. Rest of the pages have a much lower access count which is almost identical. Now, these pages

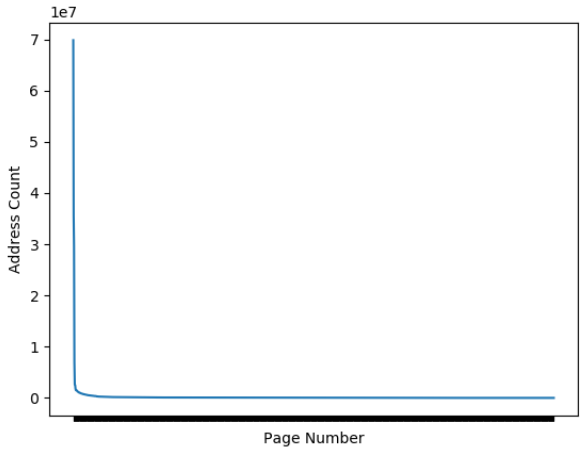


**Figure 4.2.** Type A : Hot Page count (No Cache)

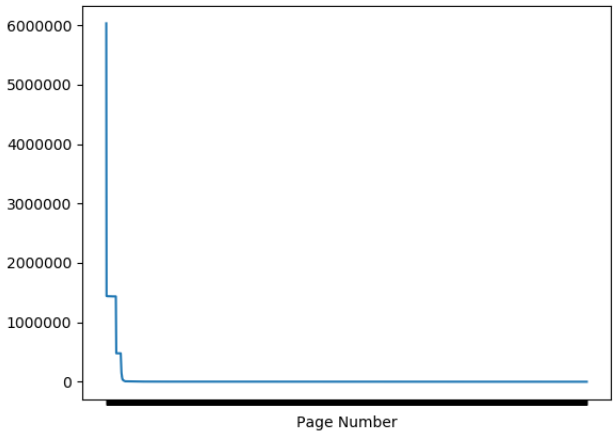


**Figure 4.3.** Type B : Hot Page count (No Cache)

with count of 107 or 106 page access constitutes almost all the hot pages which is comprising of 90 percent of total page access for a few benchmarks. But for 401.bzip2, the number is not quite very low compared to other benchmarks like 403.gcc. For 401.bzip2, the ratio between hot pages and total number of pages for different phases are 0.220668, 0.241071, 0.257426, 0.178112 whereas for 403.gcc, the ratio is 0.00564722, 0.0253585, 0.0339744, 0.0107288, 0.00366972 respectively.

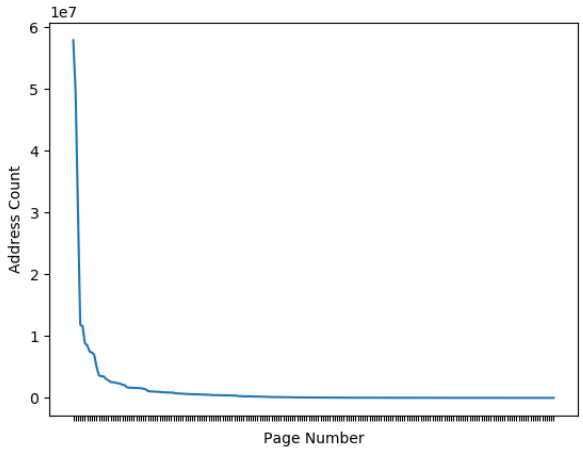


**Figure 4.4.** Type C : Hot Page count (No Cache)

It is evident that the ratio is much lesser in case of gcc compared to bzip2 which means, the hot page count for gcc is much lesser compared to bzip2. If we can identify the pages which are hot and we can find a pattern which is able to identify the similarity of hot pages for different phases, it can be made useful while accessing the memory. For example, if we can identify a certain number of pages are always hot throughout the different phases of program execution, that means the program is accessing the same pages for the entire program and the rest of the pages are basically being ignored while program execution. This can lead to a better usefulness of those hot pages. These hot pages can be used with the higher bandwidth memory which are being accessed a lot of times by the program and the rest of the pages can be put into lower bandwidth memory where the access probability is much lesser. This can lead to a better resourcefulness for using a memory and can allow the user to use lesser memory for a particular program execution.

**Distinct Memory Address Accessed :**

Our next graph depicts the distinct memory access count for the hot pages. Each of our page is of size 4K and it can have maximum of 4096 number of memory access. That means one particular page can have reference of 4096 memory access. From the figure 4.5 we can see that a good portion of pages are having a distinct

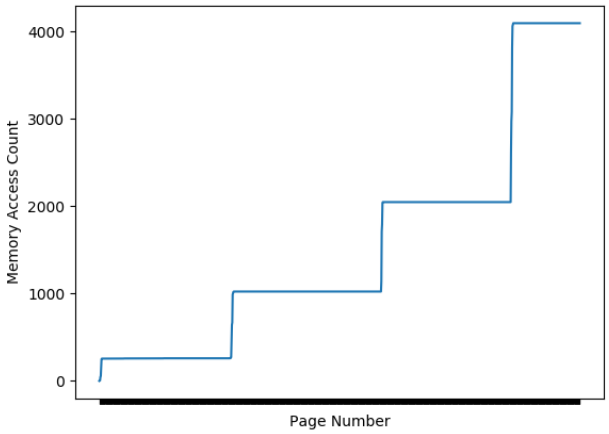


**Figure 4.5.** Type A : Distinct Memory Address Accessed (No Cache)

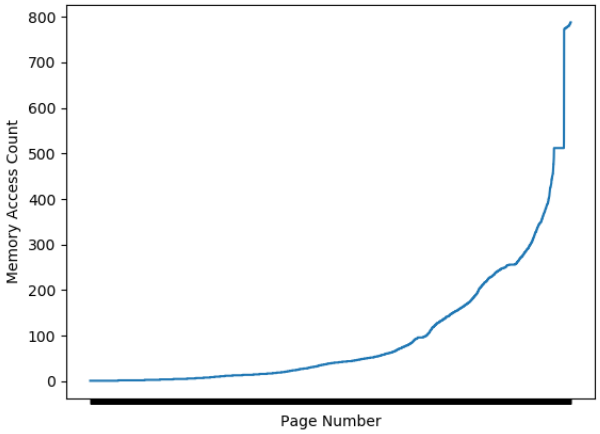


**Figure 4.6.** Type B : Distinct Memory Address Accessed (No Cache)

memory access count close to 4096. These are the pages where all of the page table entries are accessed fully. There are also some pages with memory access counts of near to 64, 1024, 2048 as seen in the graphs. For such pages the page table entries are mostly empty and access count is lower.

For the graph 4.6, we can see there is a steady increase of distinct memory access count for individual pages. None of the pages for any phases has a memory access more than 1000 in any of the phases. That indicates most of the page entries will remain empty for the pages.

**Hot page / Total page ratio :**

We have previously discussed the ratio of hot pages/ total pages when there is a consideration of a level 3 cache for a program execution. Here we are providing the findings of the same thing but without a consideration of a cache.

From previous discussion it is already clear that the access count will be much larger compared to the scenario with a cache. Here is the chart that shows the hot page/total page ratio for all the phases for each benchmark. If we compare this chart with the previous chart, there are some huge differences of the values. For this chart many benchmark phases exhibit a ratio much smaller compares to the previous scenario. That means for such cases count of hot pages a relatively low compared to the total number of pages. There are very few benchmarks which have a ratio more than 0.5. Also, if we look closely, there are a few benchmarks which has some phases that has a ratio much higher compared to other phases.

To find the similarity of phases we have done our next experiment.

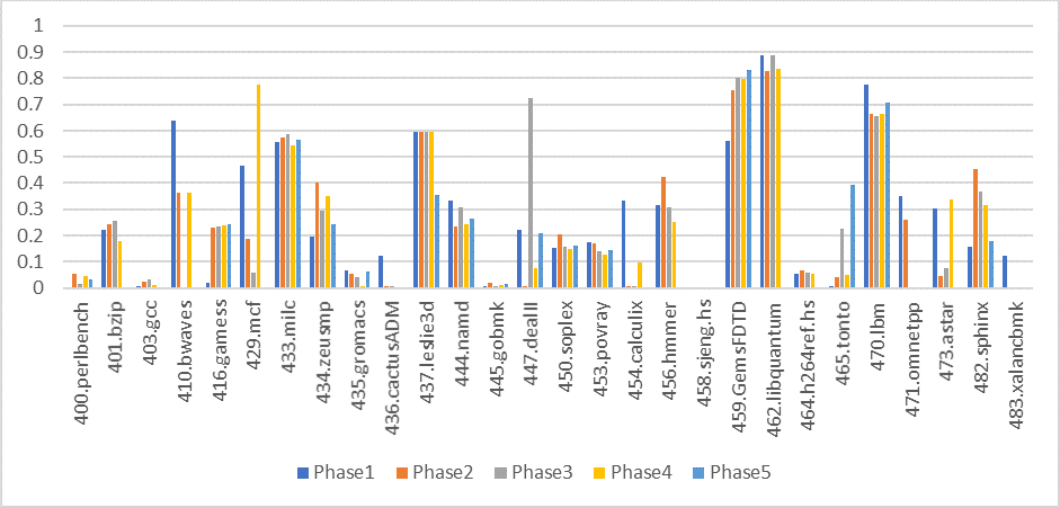


**Figure 4.7.** Hot Page count ratio without cache

**Access Pattern :**

We have provided a detailed overview of access patterns for the hot pages in cases of program execution without a cache. Here we will be discussing about the same topic but without the integration of a cache memory. As there is no consideration of cache the access count is definitely a lot higher in this case. We are finding the read-access and write-access ratio of the phases and provided the chart based on read and write dominated page ratio count to total number of page access count. From the graphs we can very easily conclude that the ratio of read-dominated pages is a lot higher compared to the write-dominated pages just like the previous scenario. Also, one observation that is different from the case with cache integrated memory access is neither read-only nor write-only pages are dominating for most of the benchmarks under consideration. If we look closely, majority of benchmarks have a ratio with lower value for most of the phases which tells us the program execution for these benchmarks are balanced between read-only and write-only pages.

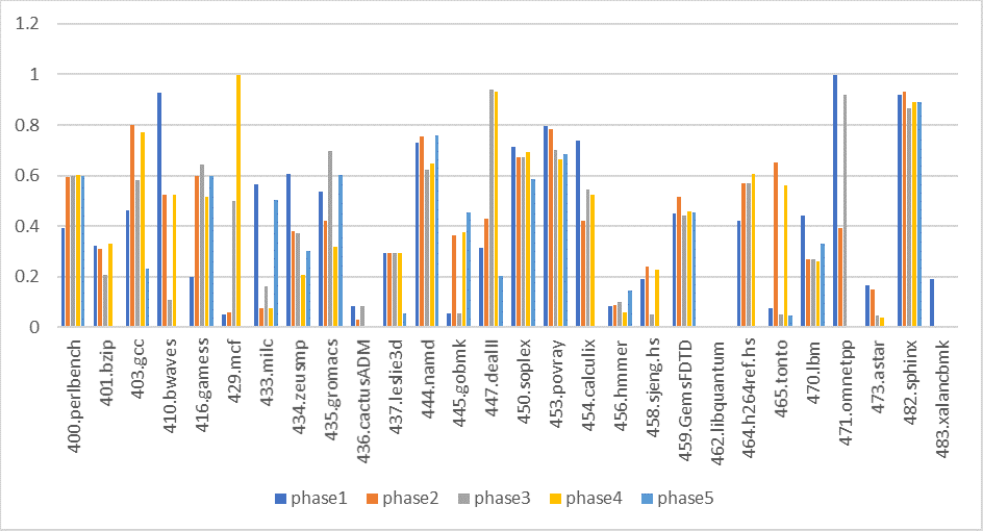


**Figure 4.8.** Read-Only Access (No Cache)

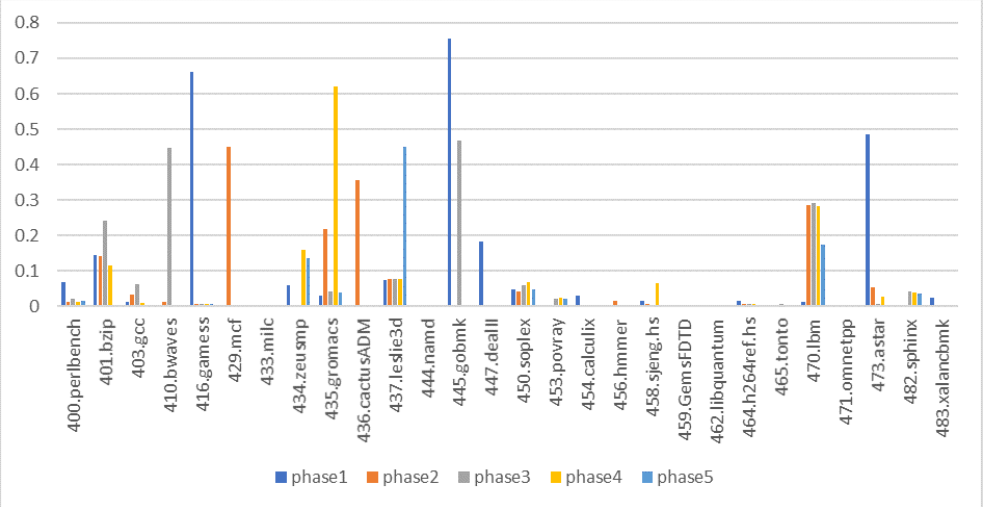


**Figure 4.9.** Write-Only Access (No cache)

**Similarities Between the phases :**

Previously we discussed the similarity between the two phases of program execution with integrated cache. Here, in this section we will discuss the same for program execution without integration of a cache. We have used the same equations that were discussed previously to generate our data for different benchmarks.

| Benchmarks | Statistics between the Phases | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **1-2** | **1-3** | **1-4** | **1-5** | **2-3** | **2-4** | **2-5** | **3-4** | **3-5** | **4-5** |
| 400.perlbench | 0.034 | 0.05 | 0.039 | 0.05 | 0.253 | 0.464 | 0.438 | 0.298 | 0.392 | 0.492 |
| 401.bzip | 0.017 | 0.397 | 0.347 | | 0.02 | 0.012 | | 0.295 | | |
| 403.gcc | 0.012 | 0.02 | 0.039 | 0.017 | 0.003 | 0.003 | 0.013 | 0.053 | 0.002 | 0.006 |
| 410.bwaves | 0.43 | 0 | 0.526 | | 0 | 0.746 | | 0 | | |
| 416.gamess | 0.045 | 0.051 | 0.015 | 0.049 | 0.408 | 0.279 | 0.651 | 0.176 | 0.5 | 0.246 |
| 429.mcf | 0.204 | 0.006 | 0.441 | | 0.017 | 0.125 | | 0.004 | | |
| 433.milc | 0.596 | 0.792 | 0.51 | 0.751 | 0.491 | 0.539 | 0.449 | 0.436 | 0.876 | 0.395 |
| 434.zeusmp | 0.112 | 0.362 | 0.044 | 0.204 | 0.434 | 0.698 | 0.303 | 0.362 | 0.304 | 0.335 |
| 435.gromacs | 0.206 | 0.447 | 0.013 | 0.705 | 0.039 | 0.017 | 0.105 | 0.014 | 0.596 | 0.008 |
| 436.cactusADM | 0.037 | 0.037 | | | 1 | | | | | |
| 437.leslie3d | 0.988 | 0.986 | 0.993 | 0.783 | 0.992 | 0.984 | 0.784 | 0.98 | 0.784 | 0.782 |
| 444.namd | 0.036 | 0.08 | 0.032 | 0.017 | 0.147 | 0.48 | 0.279 | 0.052 | 0.152 | 0.357 |
| 445.gobmk | 0.377 | 0.659 | 0.656 | 0.347 | 0.514 | 0.303 | 0.711 | 0.455 | 0.479 | 0.239 |
| 447.dealII | 0.007 | 0.01 | 0.042 | 0.033 | 0 | 0.02 | 0.002 | 0.001 | 0.001 | 0.003 |
| 450.soplex | 0.466 | 0.531 | 0.431 | 0.448 | 0.43 | 0.281 | 0.327 | 0.376 | 0.416 | 0.558 |
| 453.povray | 0.868 | 0.8 | 0.659 | 0.8 | 0.821 | 0.634 | 0.868 | 0.789 | 0.895 | 0.744 |
| 454.calculix | 0 | 0 | 0.032 | | 0.5 | 0 | | 0 | | |
| 456.hmmer | 0.462 | 0.67 | 0.457 | | 0.392 | 0.263 | | 0.306 | | |
| 458.sjeng.hs | 0.806 | 0.818 | 0.588 | | 0.657 | 0.733 | | 0.556 | | |
| 459.GemsFDTD | 0.398 | 0.576 | 0.417 | 0.438 | 0.609 | 0.581 | 0.589 | 0.642 | 0.724 | 0.643 |
| 462.libquantum | 0.733 | 0.785 | 0.769 | | 0.723 | 0.753 | | 0.781 | | |
| 464.h264ref.hs | 0.193 | 0.354 | 0.266 | | 0.192 | 0.183 | | 0.377 | | |
| 465.tonto | 0.034 | 0.016 | 0.069 | 0.005 | 0.023 | 0.5 | 0.007 | 0.027 | 0.242 | 0.009 |
| 470.lbm | 0.534 | 0.523 | 0.533 | 0.687 | 0.451 | 0.58 | 0.509 | 0.767 | 0.763 | 0.757 |
| 471.omnetpp | 0.006 | 0.007 | | | 0.005 | | | | | |
| 473.astar | 0.075 | 0.063 | 0.002 | | 0.015 | 0.028 | | 0.005 | | |
| 482.sphinx | 0.313 | 0.407 | 0.439 | 0.71 | 0.545 | 0.572 | 0.396 | 0.798 | 0.508 | 0.544 |
| 483.xalancbmk | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |

**Table 4.1.** Similarity Metric without Cache traces

Along with the table, we have provided the scattered graph for better understanding of the analyzed data. From the graph we can see that the similarity metric provides the values that varies significantly for different benchmarks. It is also different from the scenario where we found the graph for program execution with cache.

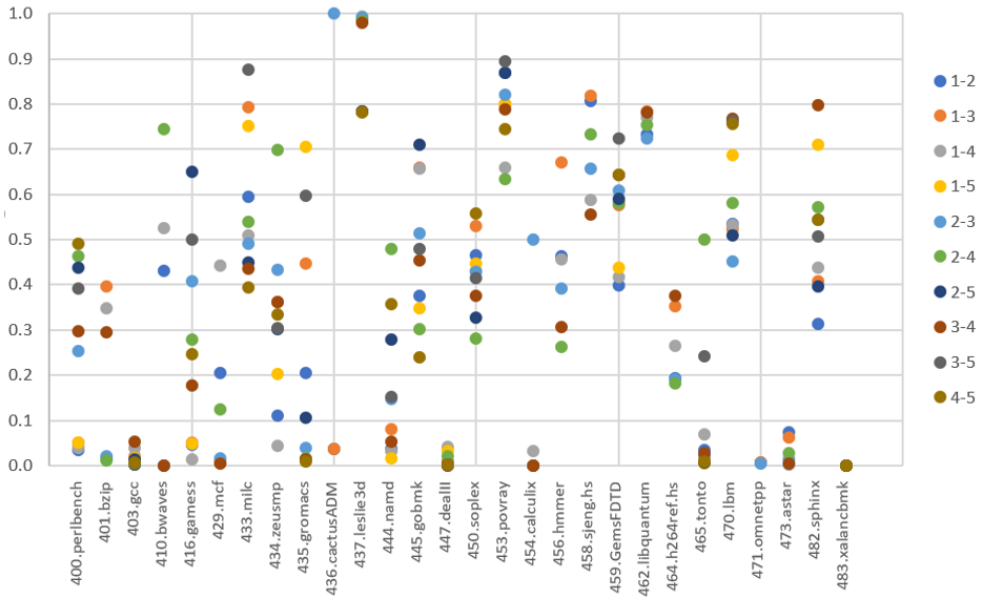


**Figure 4.10.** Similarity Metric Scattered Graph (No Cache)

Very few of the phases has a similarity value as near as 1 which is the highest value possible. Weighted Similarity Metric for hot pages provides a comparatively different graph from the previous similarity metric graph. The main change to notice here is many values are reaching a high value as high as 1 in this case.

This signifies, though most of the pages are not common between such two phases, when we are using count to find the weighted metric, the numerator value is made a lot higher and hence the ratio value increases accordingly. It means the hot pages which are the intersect between two phases have the count very high compared to other hot pages which are not under intersection set. The graph for weighted similarity metric for all pages has changed a lot as well compared to the previous two graphs. We can see that most of the numbers are reaching a ratio as high as 1 in this case. This means the total number of access of pages between two phases is almost similar and the number of intersected pages is really high

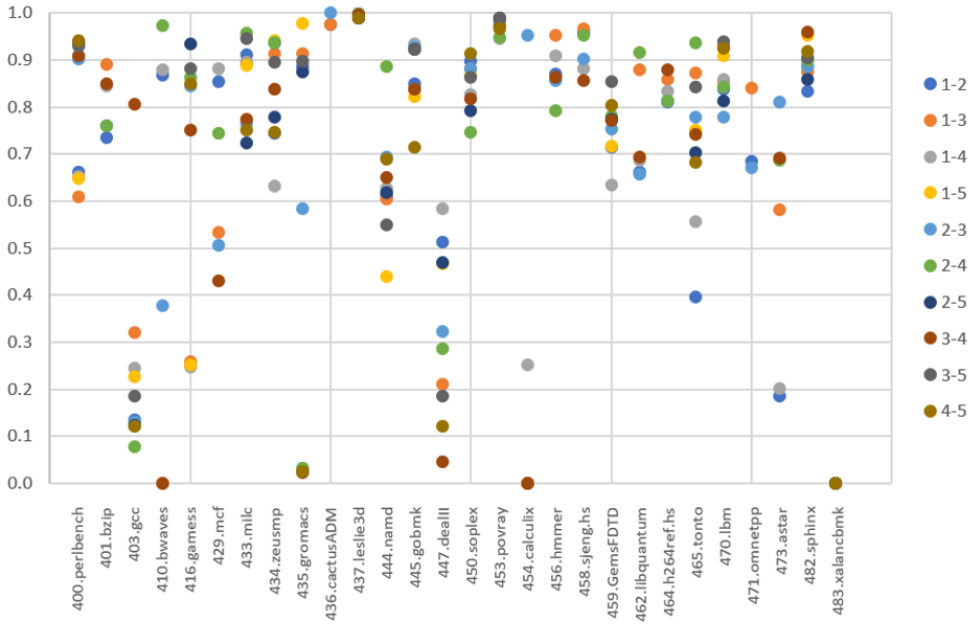


**Figure 4.11.** Weighted Similarity metric for hot Pages Scattered Graph (No Cache)

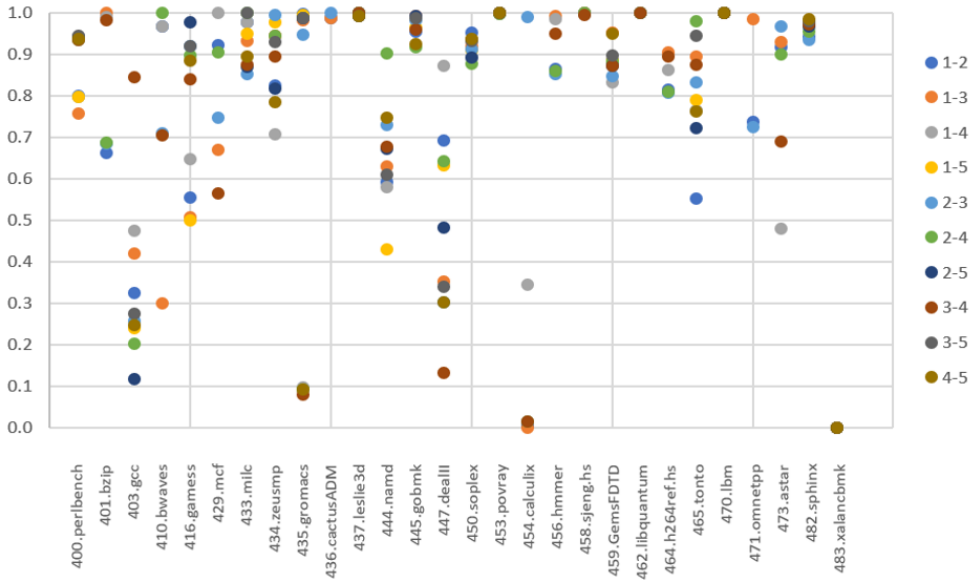


**Figure 4.12.** Weighted Similarity metric for all Pages Scattered Graph (No Cache)

| Benchmarks | Statistics between the Phases | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **1-2** | **1-3** | **1-4** | **1-5** | **2-3** | **2-4** | **2-5** | **3-4** | **3-5** | **4-5** |
| 400.perlbench | 0.661 | 0.61 | 0.653 | 0.648 | 0.902 | 0.937 | 0.936 | 0.91 | 0.929 | 0.94 |
| 401.bzip | 0.735 | 0.891 | 0.845 | | 0.761 | 0.76 | | 0.85 | | |
| 403.gcc | 0.135 | 0.321 | 0.245 | 0.226 | 0.131 | 0.078 | 0.124 | 0.806 | 0.185 | 0.121 |
| 410.bwaves | 0.867 | 0 | 0.879 | | 0.378 | 0.972 | | 0 | | |
| 416.gamess | 0.251 | 0.259 | 0.248 | 0.251 | 0.844 | 0.863 | 0.935 | 0.75 | 0.881 | 0.85 |
| 429.mcf | 0.854 | 0.534 | 0.881 | | 0.505 | 0.744 | | 0.43 | | |
| 433.milc | 0.912 | 0.894 | 0.896 | 0.888 | 0.764 | 0.957 | 0.723 | 0.773 | 0.946 | 0.75 |
| 434.zeusmp | 0.744 | 0.913 | 0.631 | 0.942 | 0.936 | 0.937 | 0.778 | 0.838 | 0.894 | 0.745 |
| 435.gromacs | 0.887 | 0.913 | 0.033 | 0.977 | 0.584 | 0.031 | 0.875 | 0.023 | 0.898 | 0.024 |
| 436.cactusADM | 0.976 | 0.976 | | | 1 | | | | | |
| 437.leslie3d | 0.998 | 0.998 | 0.999 | 0.989 | 0.999 | 0.997 | 0.989 | 0.997 | 0.989 | 0.989 |
| 444.namd | 0.612 | 0.605 | 0.628 | 0.439 | 0.694 | 0.887 | 0.619 | 0.651 | 0.55 | 0.689 |
| 445.gobmk | 0.849 | 0.926 | 0.934 | 0.822 | 0.927 | 0.837 | 0.923 | 0.837 | 0.924 | 0.715 |
| 447.dealII | 0.512 | 0.211 | 0.583 | 0.466 | 0.323 | 0.286 | 0.469 | 0.047 | 0.185 | 0.122 |
| 450.soplex | 0.898 | 0.881 | 0.827 | 0.868 | 0.881 | 0.747 | 0.791 | 0.818 | 0.863 | 0.915 |
| 453.povray | 0.987 | 0.982 | 0.945 | 0.98 | 0.983 | 0.947 | 0.989 | 0.97 | 0.99 | 0.965 |
| 454.calculix | 0 | 0 | 0.251 | | 0.951 | 0 | | 0 | | |
| 456.hmmer | 0.869 | 0.951 | 0.909 | | 0.856 | 0.792 | | 0.864 | | |
| 458.sjeng.hs | 0.958 | 0.966 | 0.881 | | 0.901 | 0.953 | | 0.855 | | |
| 459.GemsFDTD | 0.715 | 0.775 | 0.634 | 0.715 | 0.753 | 0.781 | 0.773 | 0.772 | 0.853 | 0.803 |
| 462.libquantum | 0.661 | 0.879 | 0.686 | | 0.658 | 0.916 | | 0.695 | | |
| 464.h264ref.hs | 0.81 | 0.859 | 0.834 | | 0.813 | 0.812 | | 0.879 | | |
| 465.tonto | 0.397 | 0.872 | 0.556 | 0.751 | 0.778 | 0.937 | 0.702 | 0.741 | 0.842 | 0.682 |
| 470.lbm | 0.839 | 0.854 | 0.86 | 0.908 | 0.779 | 0.843 | 0.813 | 0.927 | 0.938 | 0.924 |
| 471.omnetpp | 0.685 | 0.841 | | | 0.672 | | | | | |
| 473.astar | 0.186 | 0.581 | 0.201 | | 0.811 | 0.687 | | 0.691 | | |
| 482.sphinx | 0.834 | 0.875 | 0.889 | 0.953 | 0.887 | 0.899 | 0.858 | 0.959 | 0.904 | 0.918 |
| 483.xalancbmk | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |

**Table 4.2.** Weighted Similarity Metric for Hot pages without Cache traces

in this case making the numerator a lot higher. The pages which are common between two phases mostly has a high access count value, making the ratio higher in this case. So the phases are considered to be more symmetric here.

We have also tried to find the similarity trends for the integer benchmark and floating point benchmarks separately to find any particular pattern for program

| Benchmarks | Statistics between the Phases | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **1-2** | **1-3** | **1-4** | **1-5** | **2-3** | **2-4** | **2-5** | **3-4** | **3-5** | **4-5** |
| 400.perlbench | 0.799 | 0.759 | 0.799 | 0.798 | 0.94 | 0.935 | 0.944 | 0.935 | 0.945 | 0.938 |
| 401.bzip | 0.663 | 1 | 0.99 | | 0.686 | 0.688 | | 0.983 | | |
| 403.gcc | 0.324 | 0.42 | 0.475 | 0.24 | 0.259 | 0.203 | 0.117 | 0.846 | 0.274 | 0.247 |
| 410.bwaves | 0.968 | 0.301 | 0.968 | | 0.709 | 1 | | 0.705 | | |
| 416.gamess | 0.555 | 0.507 | 0.647 | 0.5 | 0.92 | 0.899 | 0.978 | 0.841 | 0.922 | 0.886 |
| 429.mcf | 0.923 | 0.67 | 1 | | 0.748 | 0.904 | | 0.564 | | |
| 433.milc | 0.978 | 0.933 | 0.978 | 0.951 | 0.852 | 1 | 0.871 | 0.877 | 1 | 0.895 |
| 434.zeusmp | 0.826 | 0.946 | 0.707 | 0.978 | 0.996 | 0.946 | 0.818 | 0.896 | 0.93 | 0.786 |
| 435.gromacs | 0.998 | 0.982 | 0.097 | 0.996 | 0.947 | 0.094 | 0.987 | 0.081 | 0.988 | 0.092 |
| 436.cactusADM | 0.988 | 0.988 | | | 1 | | | | | |
| 437.leslie3d | 1 | 1 | 1 | 0.993 | 1 | 1 | 0.993 | 1 | 0.993 | 0.993 |
| 444.namd | 0.592 | 0.631 | 0.581 | 0.431 | 0.73 | 0.904 | 0.672 | 0.678 | 0.61 | 0.747 |
| 445.gobmk | 0.955 | 0.994 | 0.985 | 0.965 | 0.983 | 0.918 | 0.994 | 0.961 | 0.987 | 0.925 |
| 447.dealII | 0.694 | 0.354 | 0.873 | 0.633 | 0.302 | 0.643 | 0.483 | 0.133 | 0.341 | 0.304 |
| 450.soplex | 0.952 | 0.921 | 0.893 | 0.911 | 0.914 | 0.878 | 0.892 | 0.936 | 0.933 | 0.938 |
| 453.povray | 1 | 0.999 | 0.998 | 0.999 | 0.999 | 0.999 | 0.999 | 1 | 1 | 0.999 |
| 454.calculix | 0.005 | 0.001 | 0.344 | | 0.991 | 0.016 | | 0.014 | | |
| 456.hmmer | 0.867 | 0.992 | 0.985 | | 0.854 | 0.861 | | 0.951 | | |
| 458.sjeng.hs | 1 | 0.999 | 0.999 | | 0.999 | 1 | | 0.997 | | |
| 459.GemsFDTD | 0.885 | 0.952 | 0.832 | 0.885 | 0.848 | 0.883 | 0.872 | 0.873 | 0.898 | 0.951 |
| 462.libquantum | 1 | 1 | 1 | | 1 | 1 | | 1 | | |
| 464.h264ref.hs | 0.815 | 0.904 | 0.863 | | 0.809 | 0.811 | | 0.896 | | |
| 465.tonto | 0.553 | 0.896 | 0.766 | 0.791 | 0.834 | 0.979 | 0.724 | 0.876 | 0.946 | 0.763 |
| 470.lbm | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 471.omnetpp | 0.738 | 0.986 | | | 0.726 | | | | | |
| 473.astar | 0.918 | 0.93 | 0.48 | | 0.969 | 0.901 | | 0.691 | | |
| 482.sphinx | 0.942 | 0.971 | 0.973 | 0.973 | 0.934 | 0.955 | 0.969 | 0.973 | 0.98 | 0.985 |
| 483.xalancbmk | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |

**Table 4.3.** Weighted Similarity Metric for all pages without Cache traces

execution without cache as well. But, as we analyzed we found that, the similarity metric does not follow any particular trend in our test case benchmarks.

# Chapter 5

# Related Works

Several researchers have build tools and explored memory access patterns for typical programs and application domains. In this chapter we compare our work to some such existing works.

Understanding and optimizing memory accesses have a long history. Seminal works in understanding memory access patterns revealed the presence of spatial and temporal locality in variable accesses and led to the design of caches in modern machines [21]. Other important works revealed the pattern of object lifetimes in typical object-oriented programs that show that most objects are short-lived and led to the design of generational garbage collectors that are common in modern runtime systems [10]. Such earlier works targeted different problems as compared to our goal, which is to optimize data placement for hybrid memory systems.

A few recent works have studied software memory access patterns for hybrid memory systems. One study compared a hybrid hierarchical memory model (fast DRAM and slow flash/PCM) with the existing *flat* main memory implementation and concluded that a hierarchical model has the potential to reduce both cost and power consumption with low performance overhead [23]. Shen et al. built a new

benchmark suite for studying the performance and energy impacts of H [19]. They also developed a new profiling tool to guide manual data placement in heterogeneous memory systems. This tool uses Pin-based instrumentation to only detect frequently accessed or hot arrays and passes that information to the programmers. Ji et al. developed a profiling tool to identify access patterns of data structure objects and automate the task co-locating all objects of different data structures into distinct memory regions and then map them to either DRAM or NVM [4]. Ji et al. developed a fast whole-program profiling approach that combines a fast online and slow offline profiling pass to understand several aspects of application memory behavior [8]. Our work differs for these existing works in the memory characteristics that we study and quantify.

Several other researchers have developed tools and algorithms for effective data placement on HM systems [1, 3, 9, 13, 17, 24]. We do not yet suggest an approach that will use our observations to guide automated or manual data placement on heterogeneous memory systems.

# Chapter 6

# Conclusion

As discussed, the recent memory technologies has been advancing a lot and utilizing the heterogeneous memory systems has been a primary research area over past few years. To get best performance and power characteristics we need to place OS pages in proper memory areas and need to appropriately partition data areas. For this reason, it is necessary to understand the program behavior and how the memory access is changing over the program phases. In our experiment we have tried to study this program behavior by using Pin dynamic binary translation and instrumentation system at run-time. We have understood how pin works and how the program execution time increases a lot while being executed through pin. The file traces generated are large in size as it has accessed a huge number of memory pages. Using our analysis script we have tried to generalize the statistics collected throughout. From our studies we can conclude that the program behavior for different benchmarks has changed significantly and the count for hot pages has varied as well. Also, the read write access pattern studies has found few benchmarks are having read-dominated access whereas other few had write-dominated access.

In our experiment we have collected humongous amount of data over the execution of separate benchmarks over different phases. Our data collected can be viewed using the graphs that we have provided with the document. But, we need some proper techniques for visualizing the data and for that purpose we might use some data mining or machine learning techniques. These techniques can provide us with a better understanding about how the execution behavior changes for programs over a period of time for different phases. We can also have a better ideas about how the access patterns changes and we can use those statistics to generalize our data and use to describe program behavior. Apart from that, we can use the concepts of Clustering algorithms to get the details of hot pages. In our experiment we have used the hot page as those pages with 90 percent of overall access. But, in some scenario it can be too many hot pages for a single phase execution and for those cases the actual hot pages count could be a lot lesser. For those analysis we might use clustering algorithms which can find us a more accurate number of hot pages for our analysis purpose and we can work with those memory pages.

# Chapter 7

# Appendix I

**Hot page count graph (With Cache):**

This is the Part-I of all the graphs plotted for each benchmarks to find the Hot pages count.

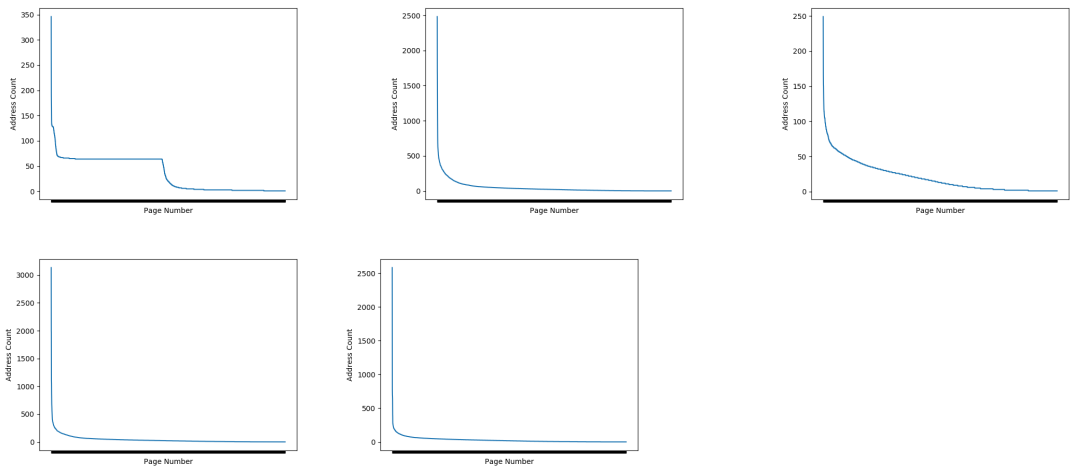


**Figure 7.1.** 400.perlbench

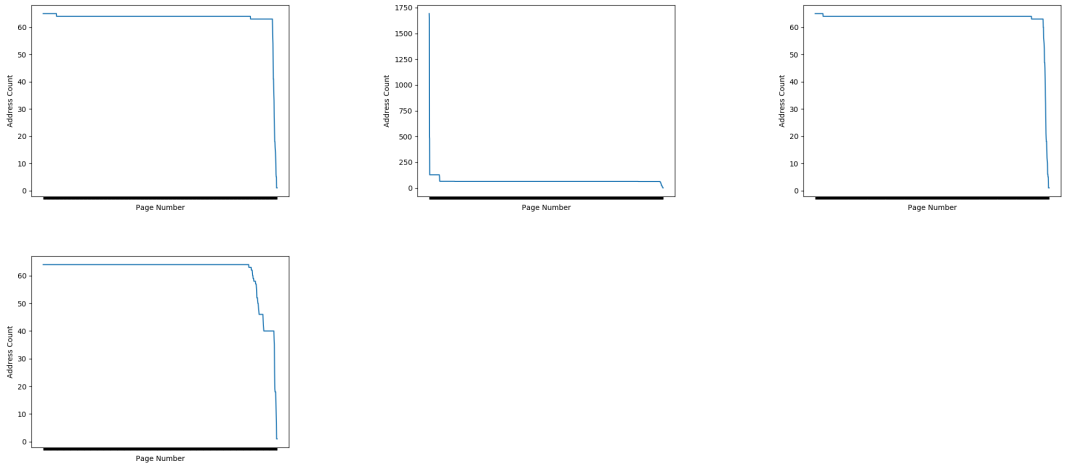


**Figure 7.2.** 401.bzip2

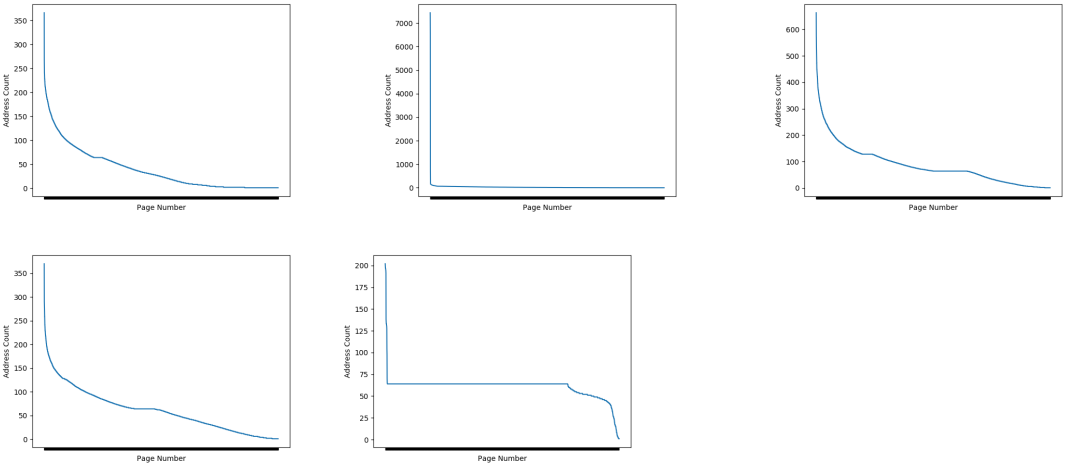


**Figure 7.3.** 403.gcc

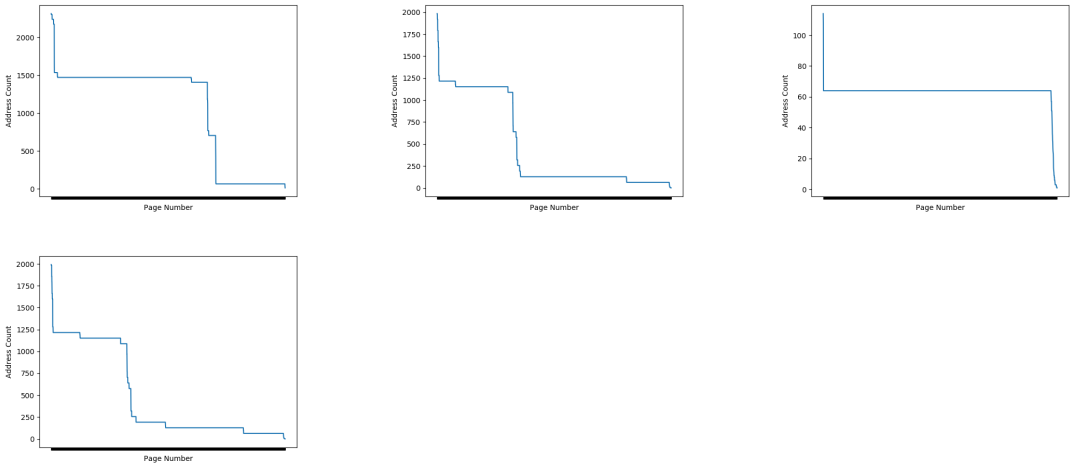


**Figure 7.4.** 410.bwaves

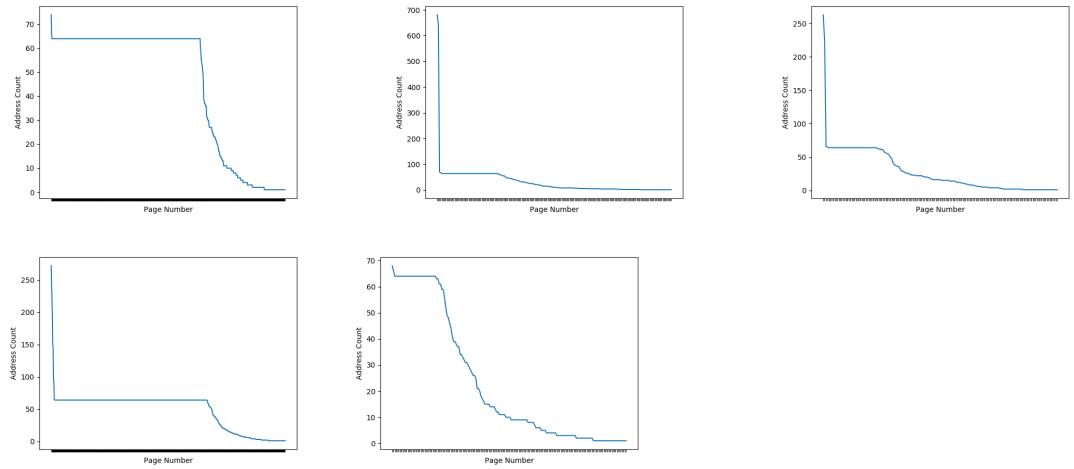


**Figure 7.5.** 416.gamess

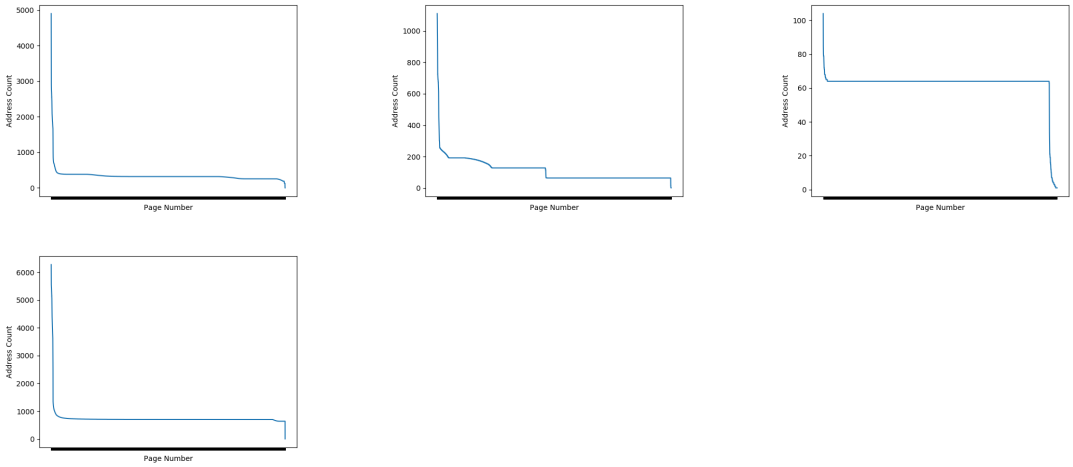


**Figure 7.6.** 429.mcf

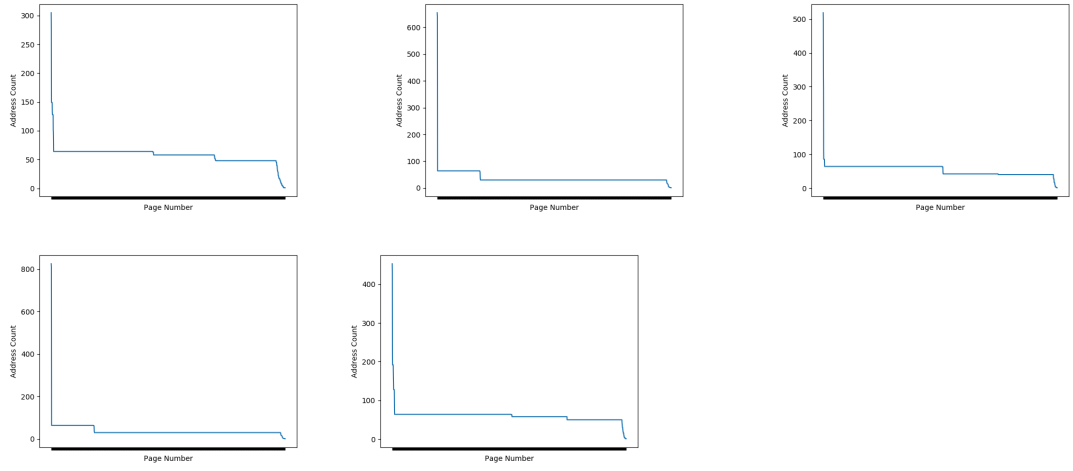


**Figure 7.7.** 433.milc

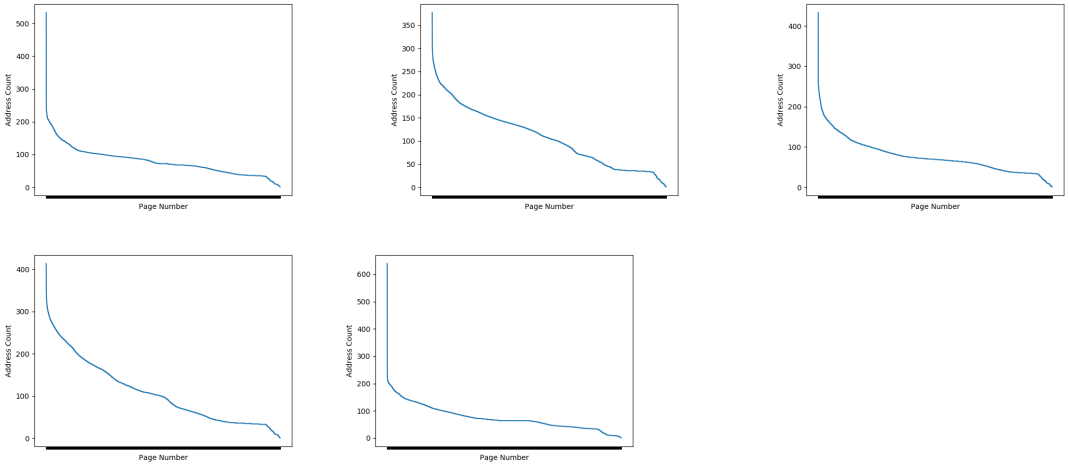


**Figure 7.8.** 434.zeusmp

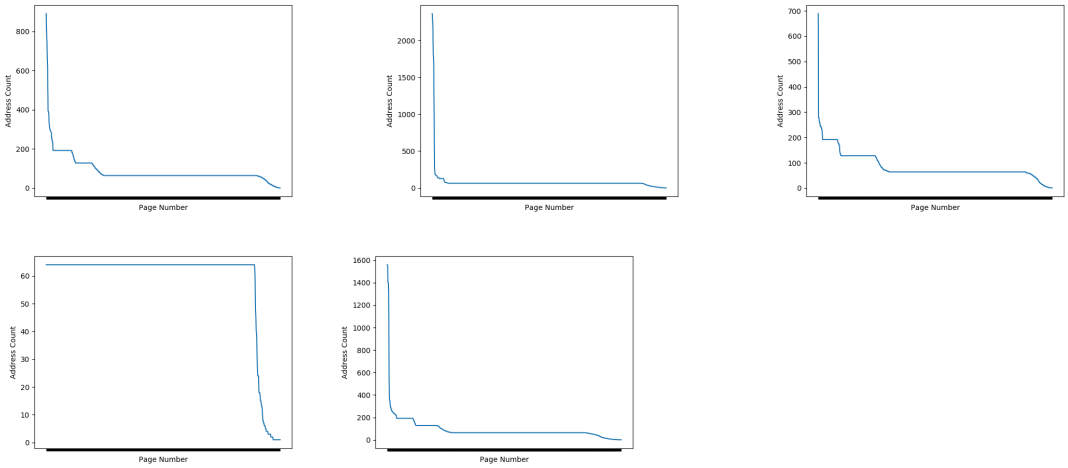


**Figure 7.9.** 435.gromacs

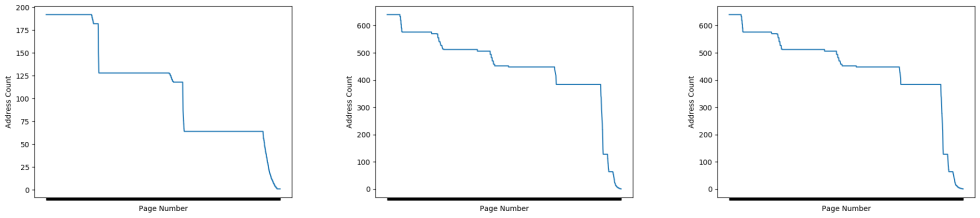


**Figure 7.10.** 436.cactusADM

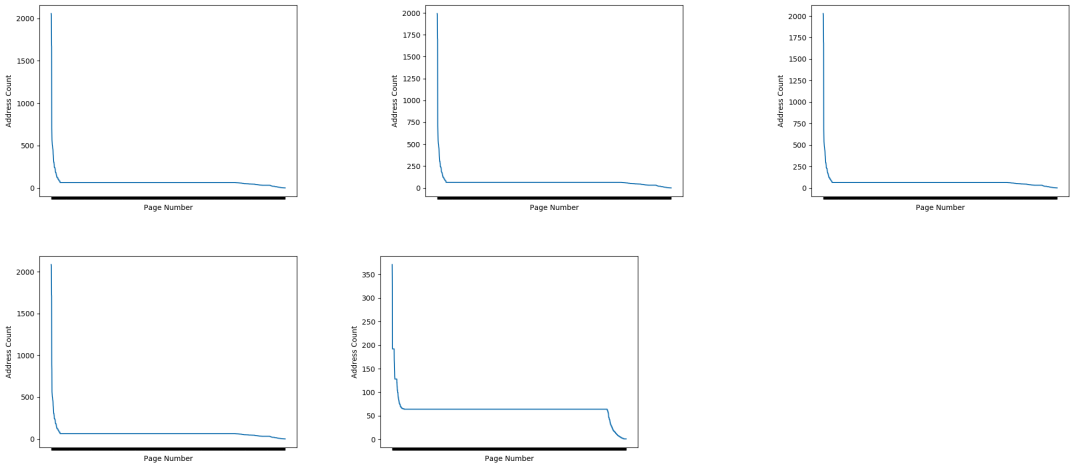


**Figure 7.11.** 437.leslie3d

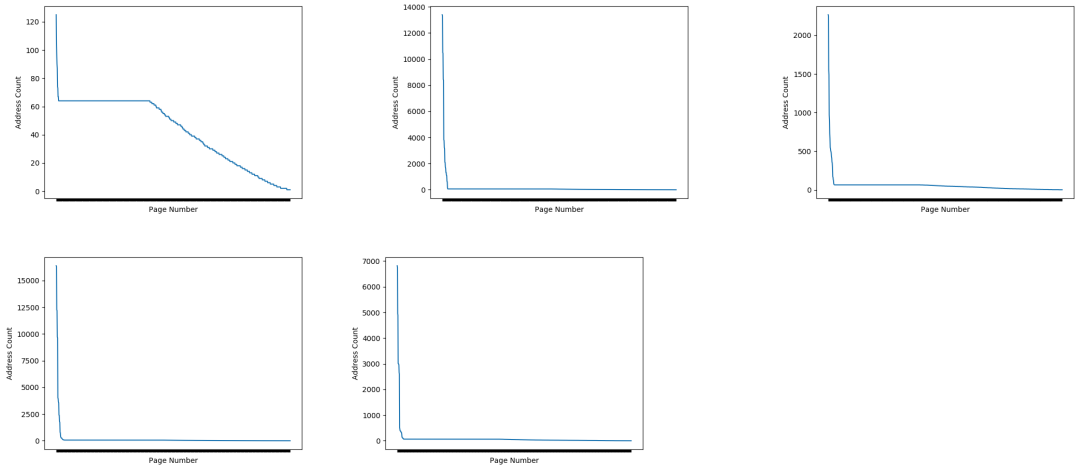


**Figure 7.12.** 444.namd

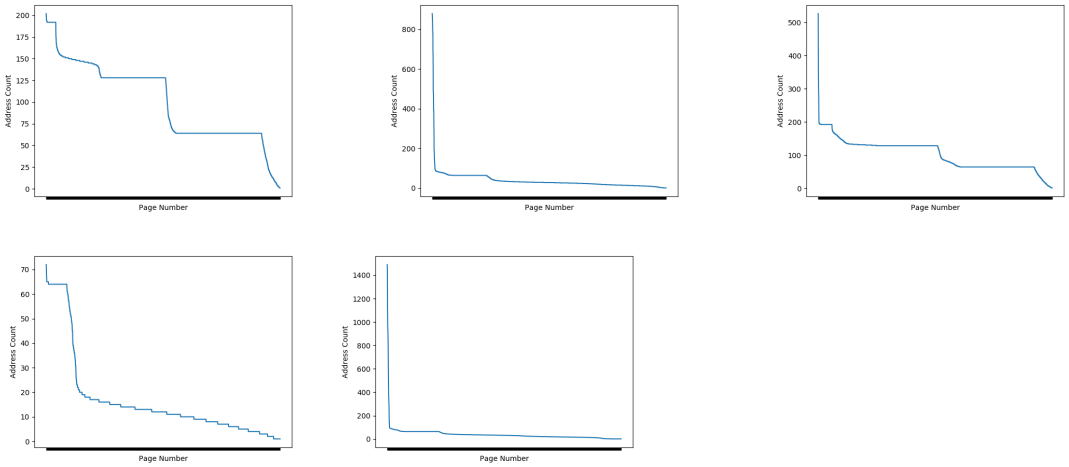


**Figure 7.13.** 445.gobmk

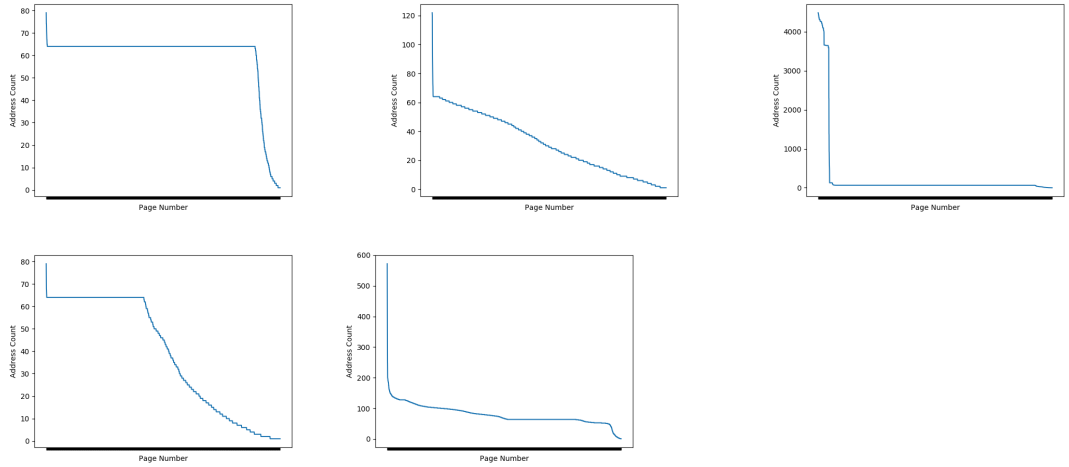


**Figure 7.14.** 447.dealII

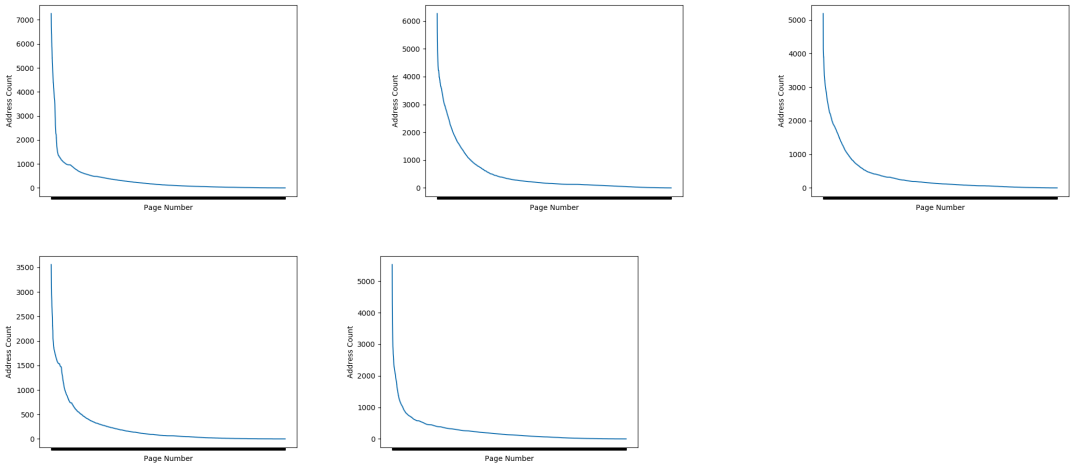


**Figure 7.15.** 450.soplex

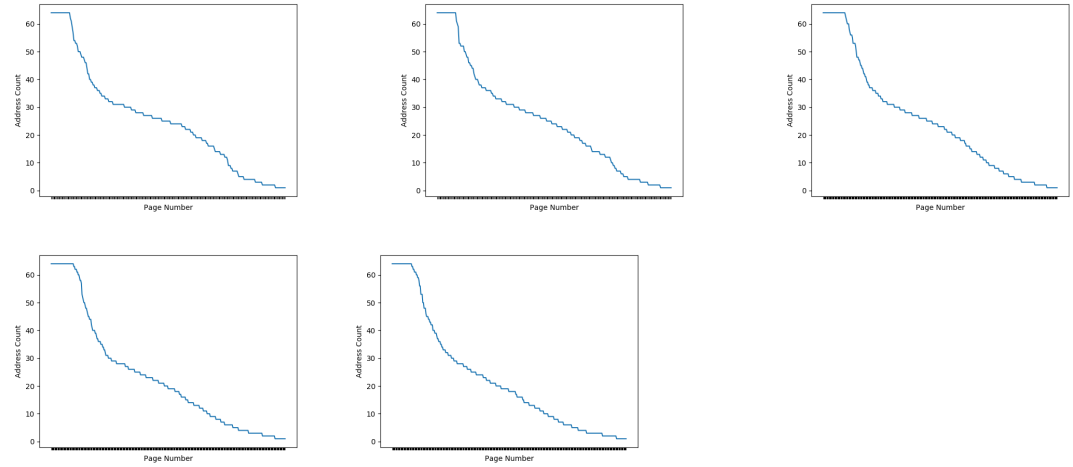


**Figure 7.16.** 453.povray

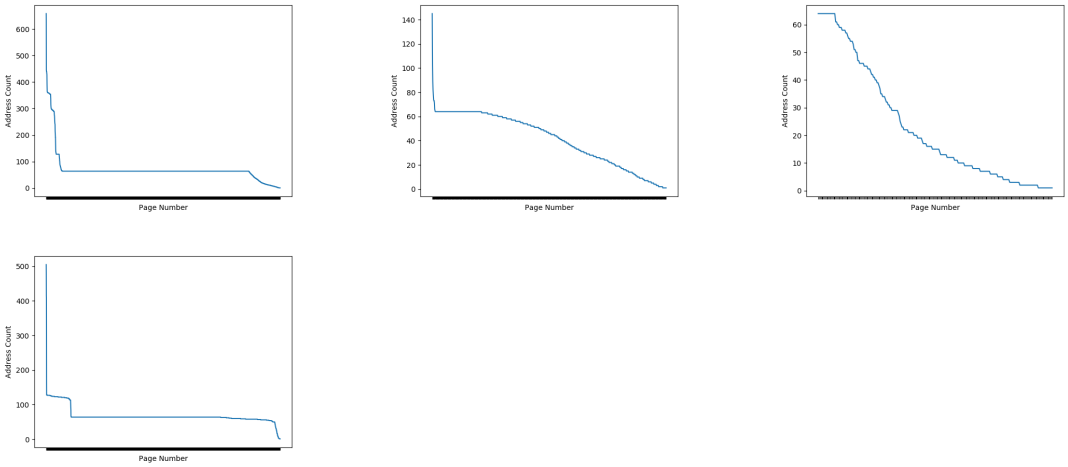


**Figure 7.17.** 454.calculix

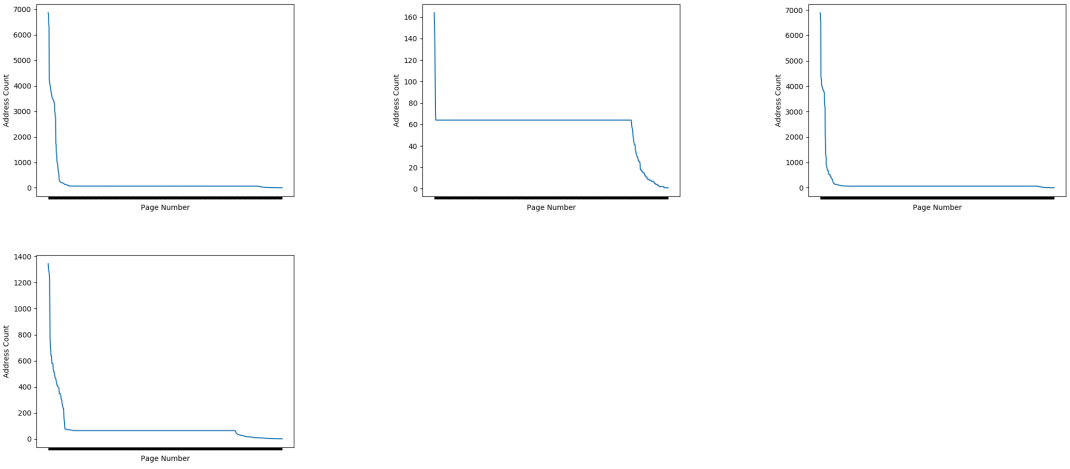


**Figure 7.18.** 456.hmmer

# Chapter 8

# Appendix I-B

**Hot page count graph (With Cache):**

This is the Part-II of all the graphs plotted for each benchmarks to find the Hot pages count.

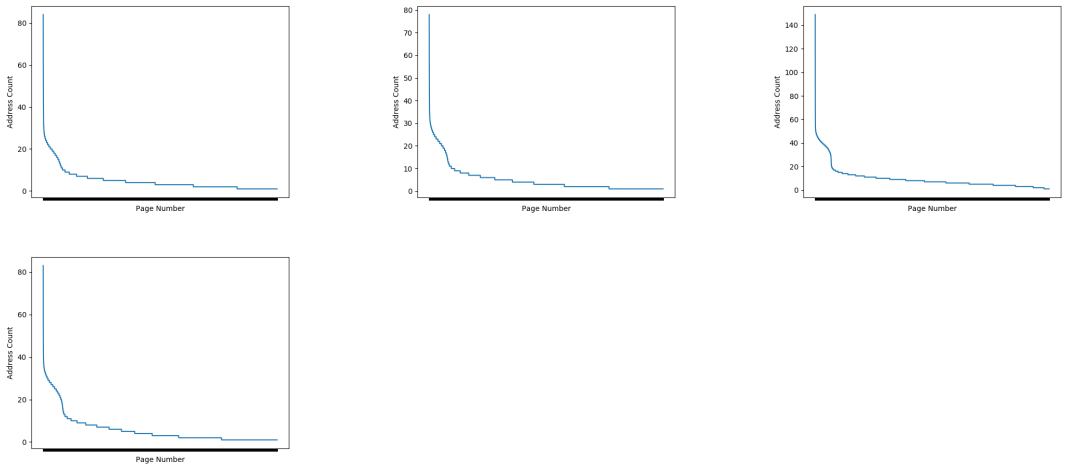


**Figure 8.1.** 458.sjeng.hs

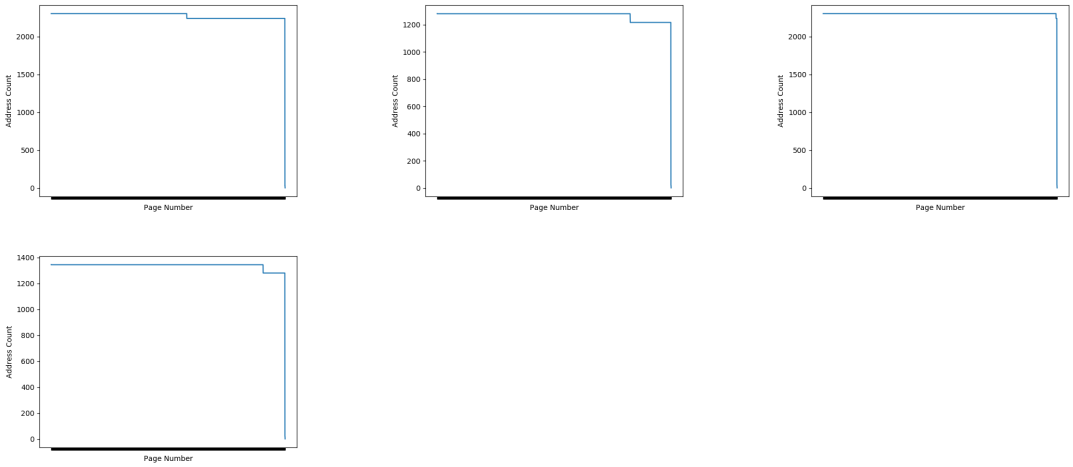


**Figure 8.2.** 462.libquantum

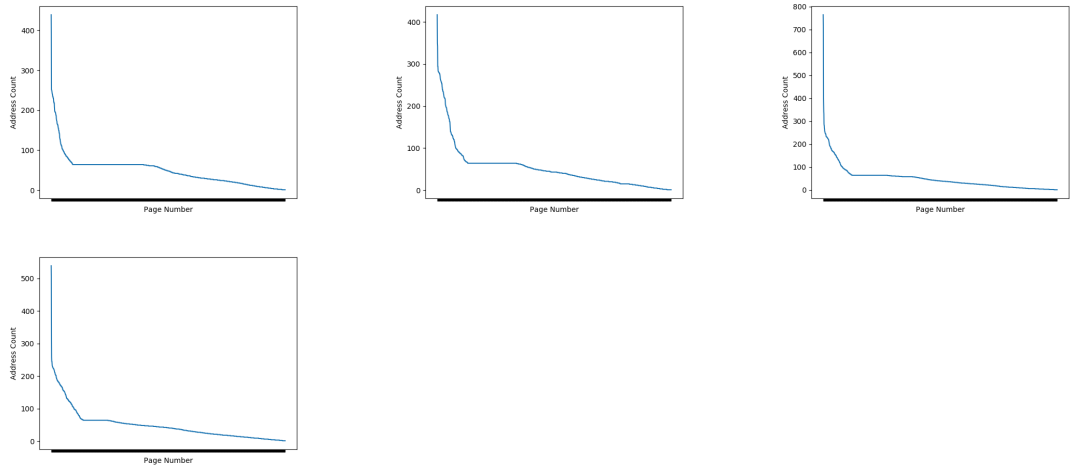


**Figure 8.3.** 464.h264ref.hs

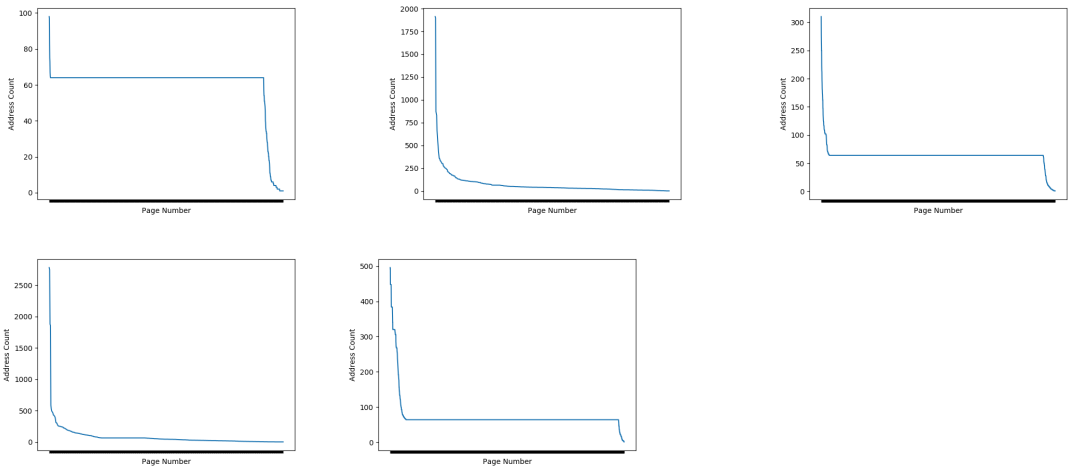


**Figure 8.4.** 465.tonto

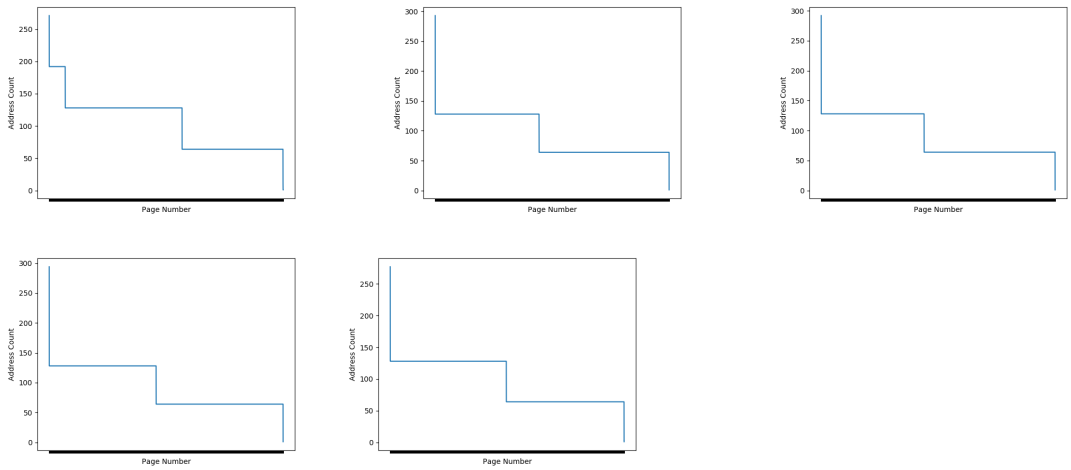


**Figure 8.5.** 470.lbm

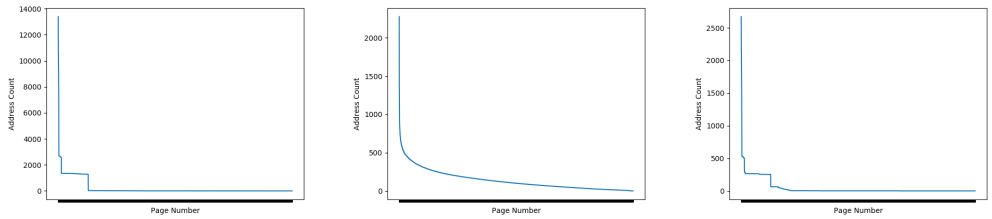


**Figure 8.6.** 471.omnetpp

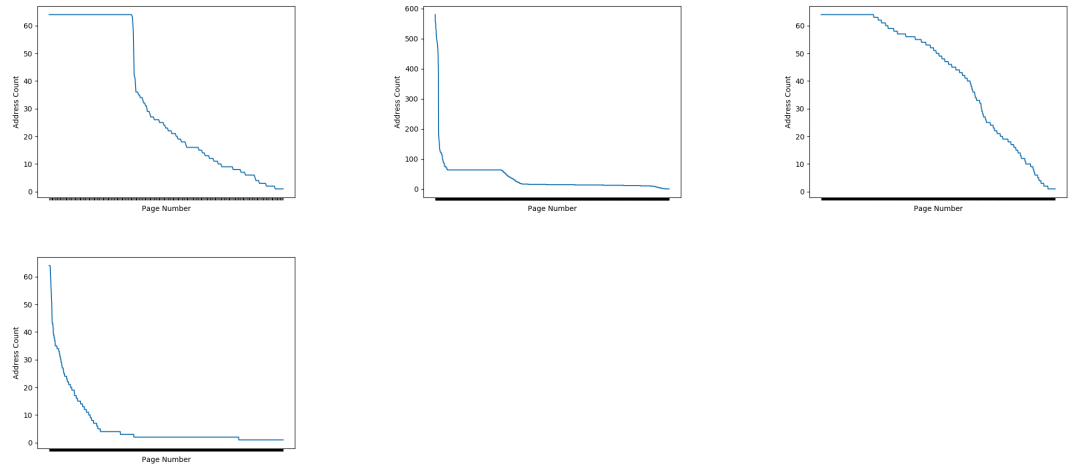


**Figure 8.7.** 473.astar

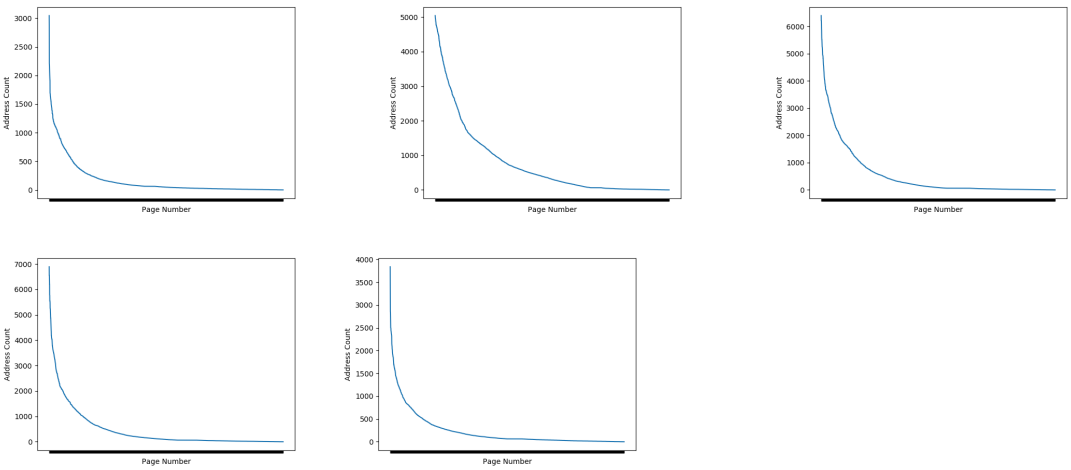


**Figure 8.8.** 482.sphinx

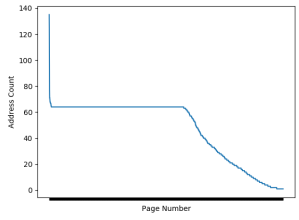


**Figure 8.9.** 483.xalancbmk

# Chapter 9

# Appendix II

**Hot page count graph (Without Cache):**

This is Part-I of all the graphs plotted for each benchmarks to find the Hot pages count without cache integration.

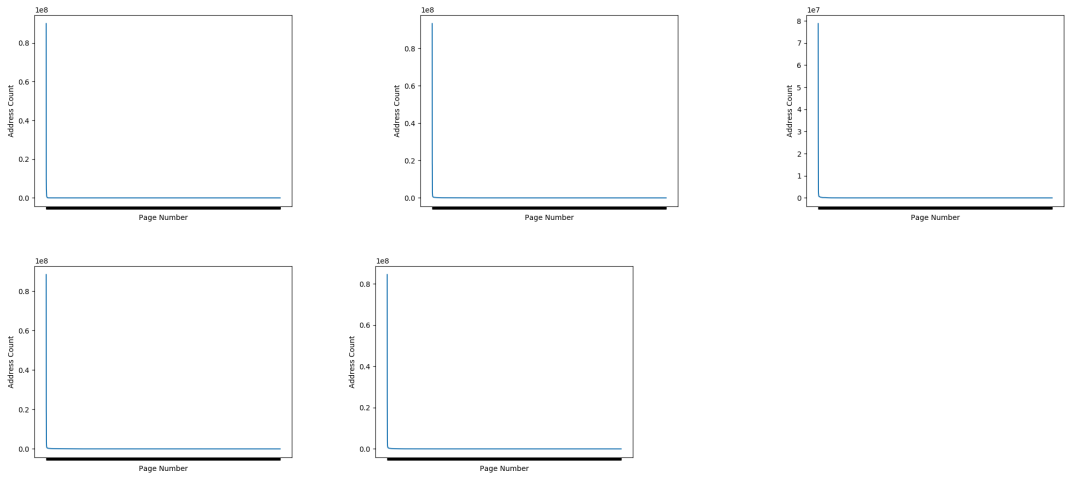


**Figure 9.1.** 400.perlbench

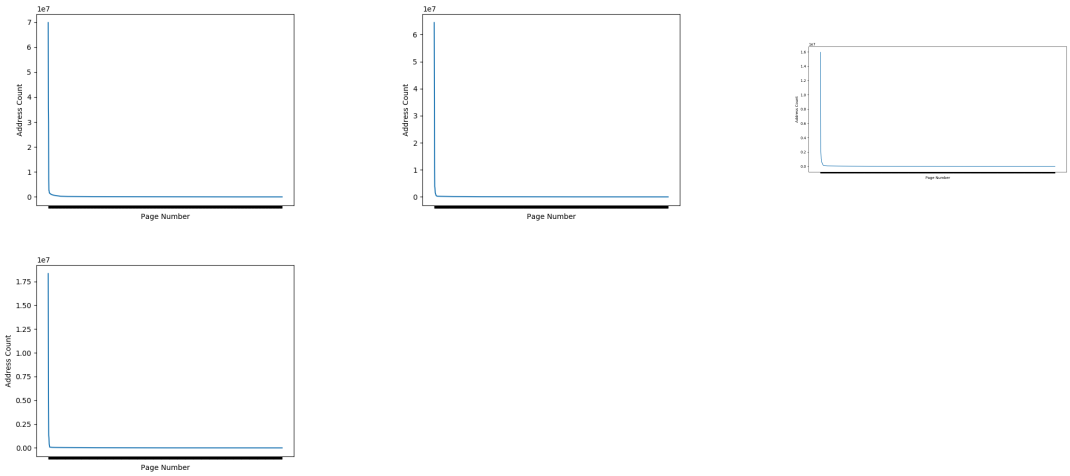


**Figure 9.2.** 401.bzip2

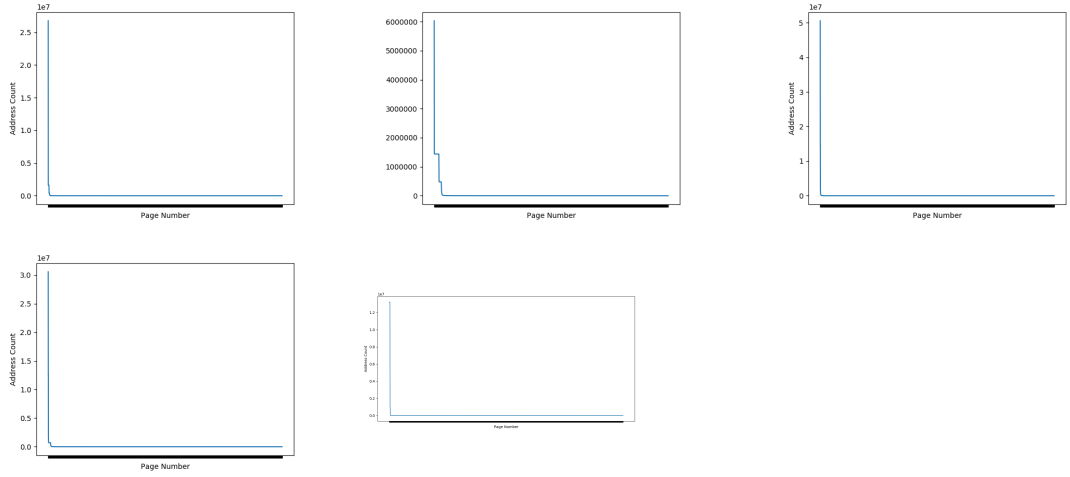


**Figure 9.3.** 403.gcc

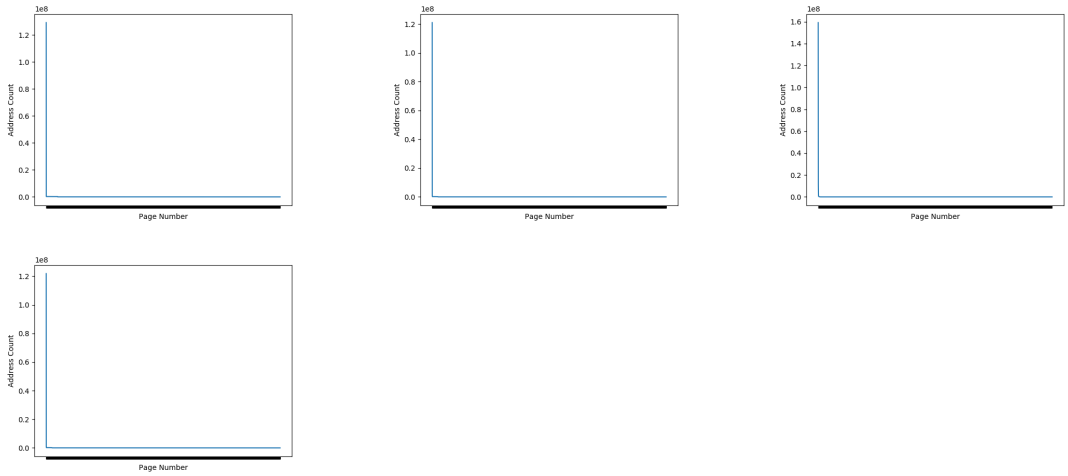


**Figure 9.4.** 410.bwaves

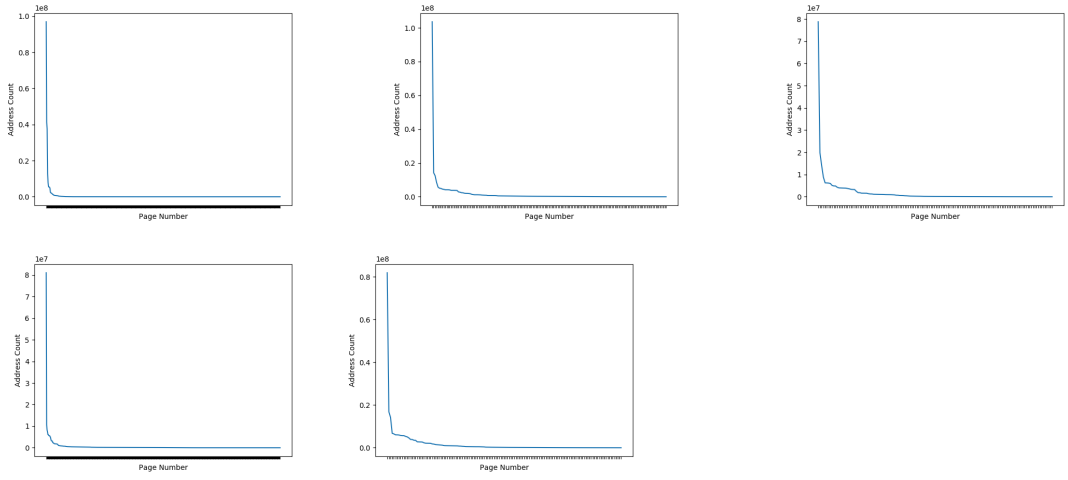


**Figure 9.5.** 416.gamess

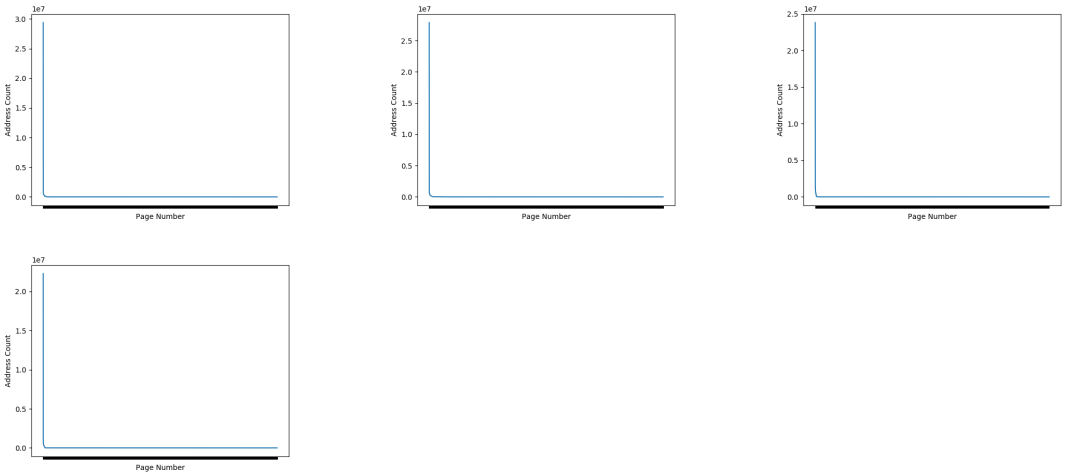


**Figure 9.6.** 429.mcf

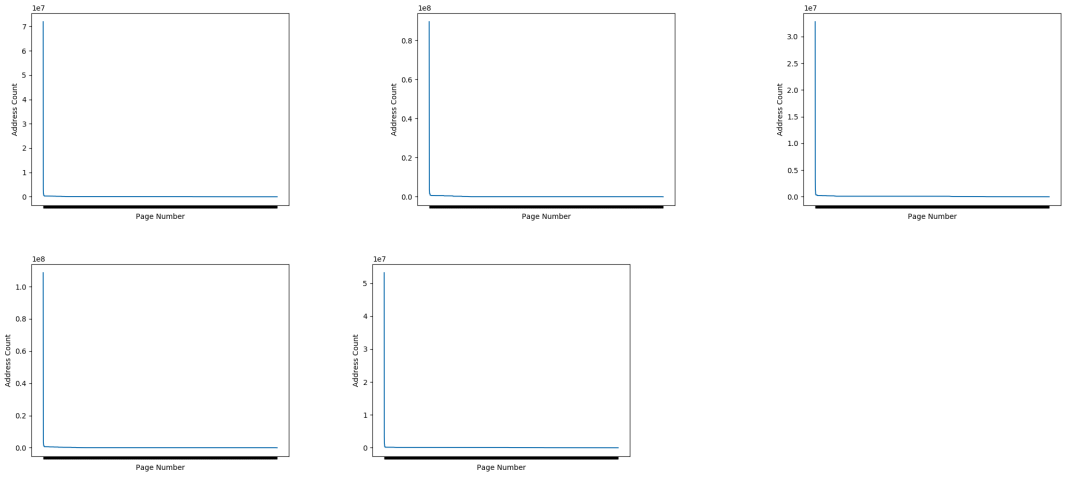


**Figure 9.7.** 433.milc

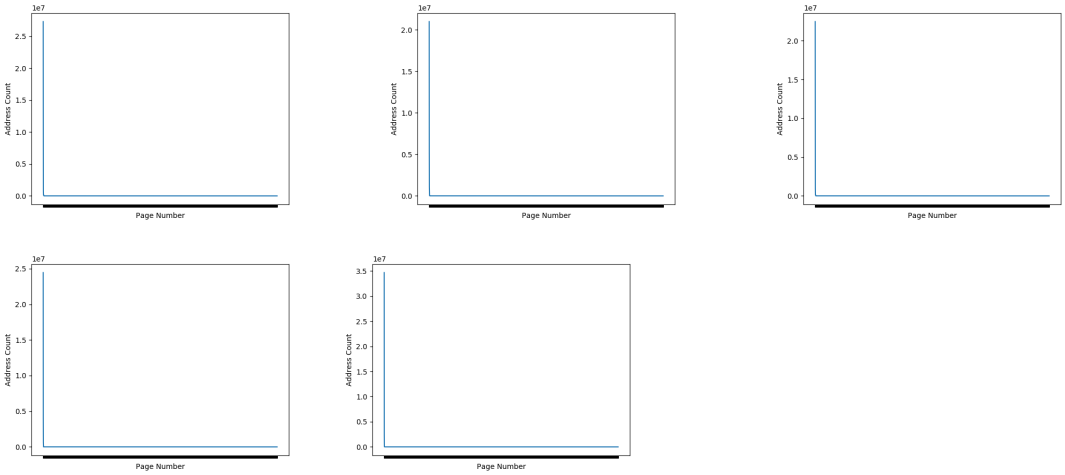


**Figure 9.8.** 434.zeusmp

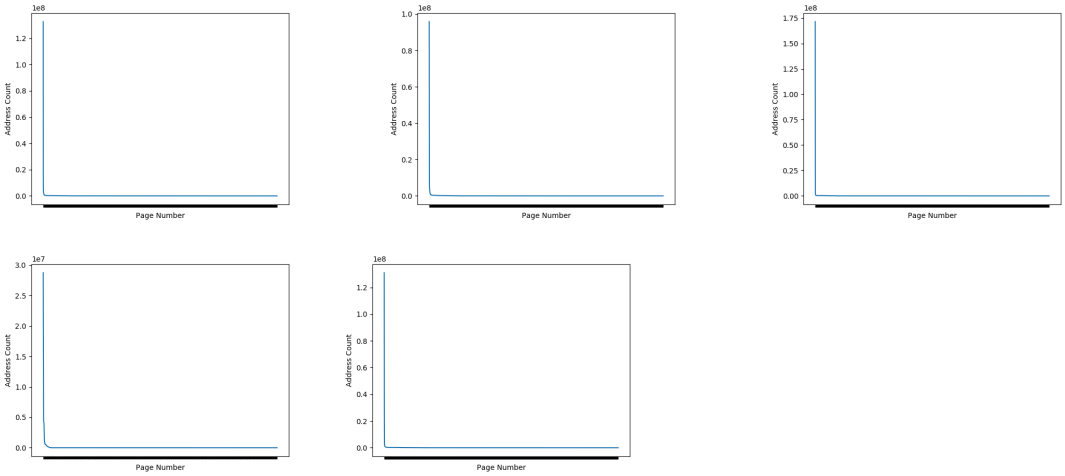


**Figure 9.9.** 435.gromacs

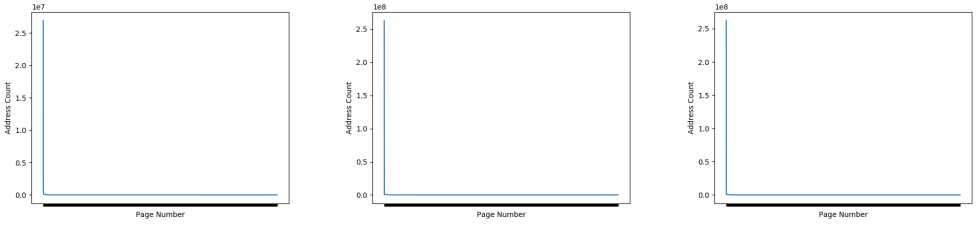


**Figure 9.10.** 436.cactusADM

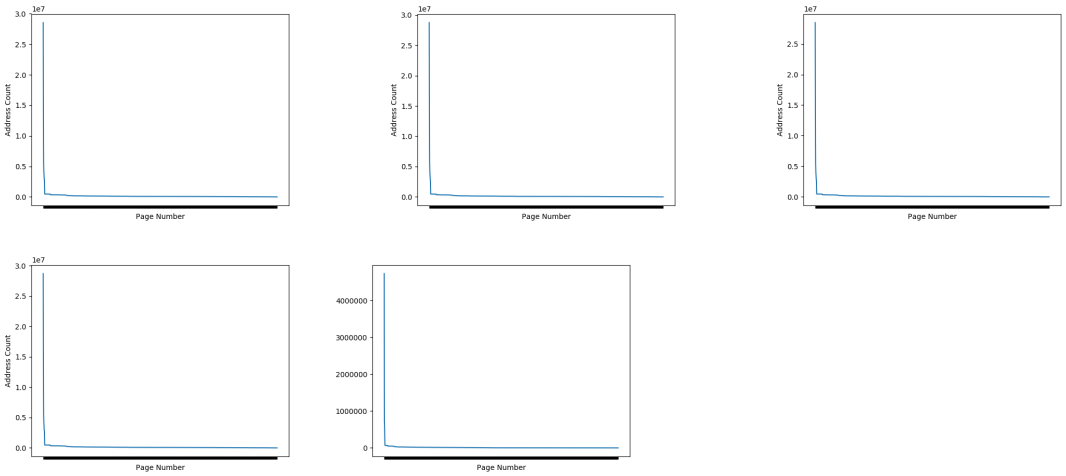


**Figure 9.11.** 437.leslie3d

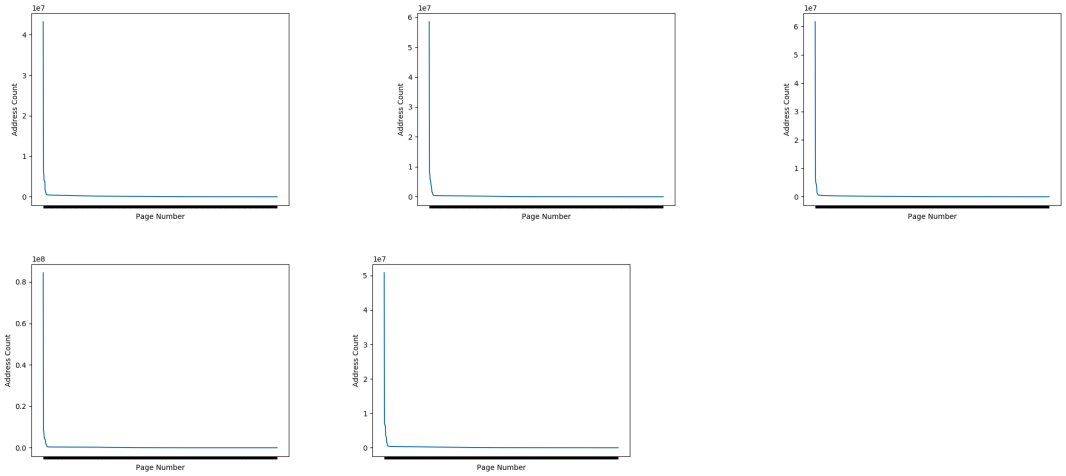


**Figure 9.12.** 444.namd

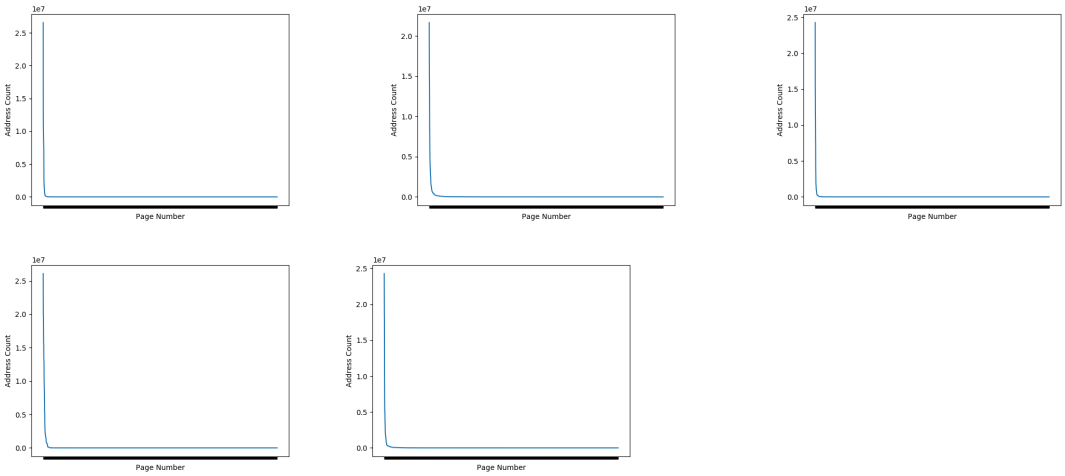


**Figure 9.13.** 445.gobmk

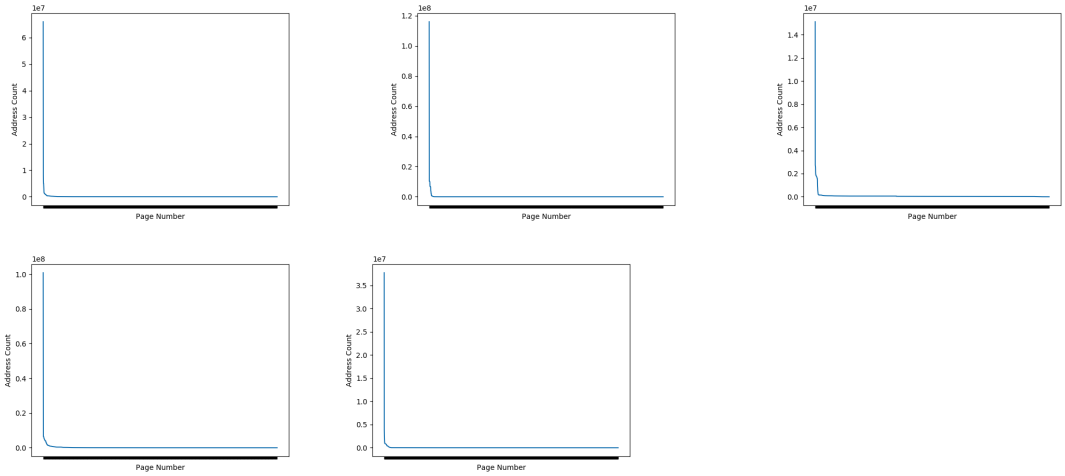


**Figure 9.14.** 447.dealII

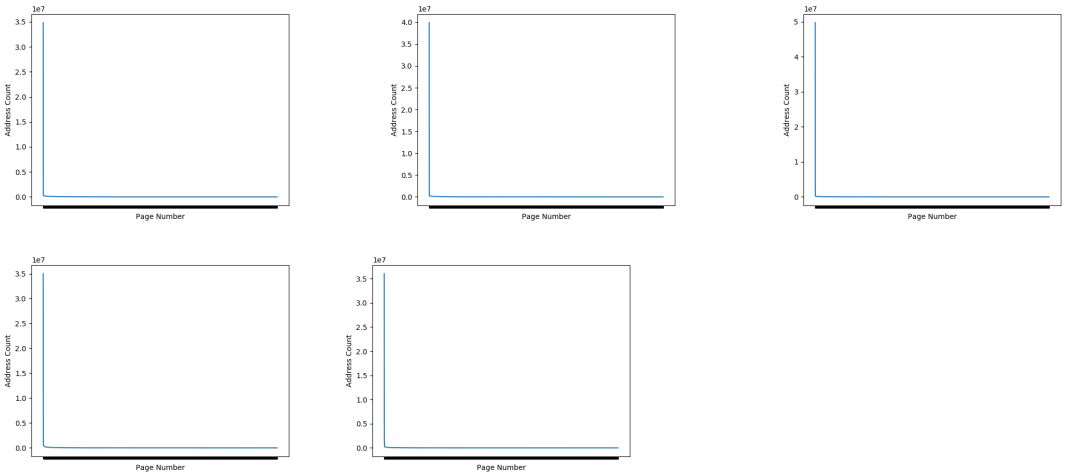


**Figure 9.15.** 450.soplex

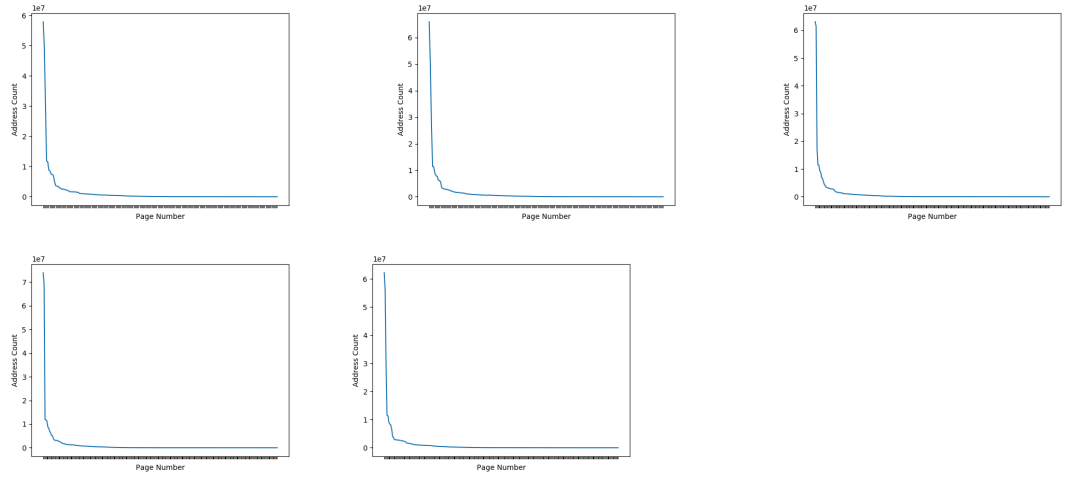


**Figure 9.16.** 453.povray

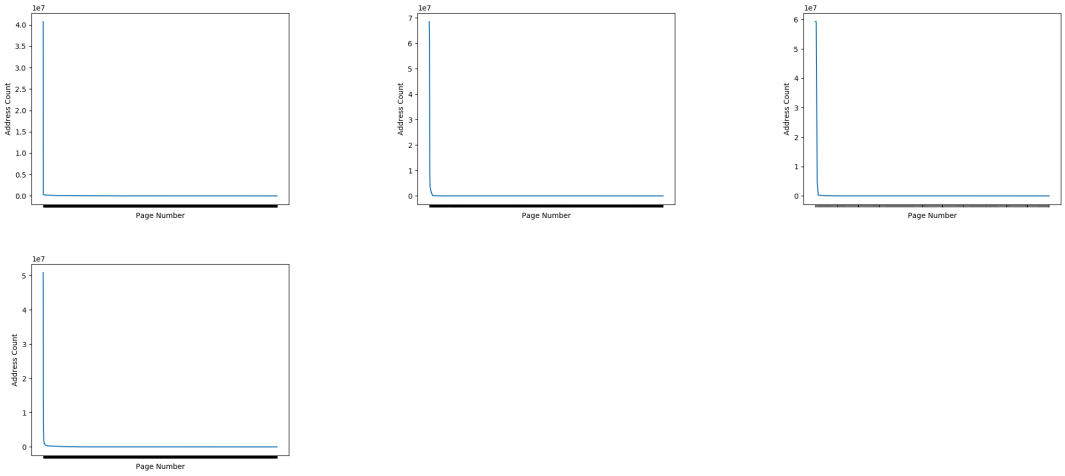


**Figure 9.17.** 454.calculix

# Chapter 10

# Appendix II-B

**Hot page count graph (Without Cache):**

This is Part-II of all the graphs plotted for each benchmarks to find the Hot pages count without cache integration.

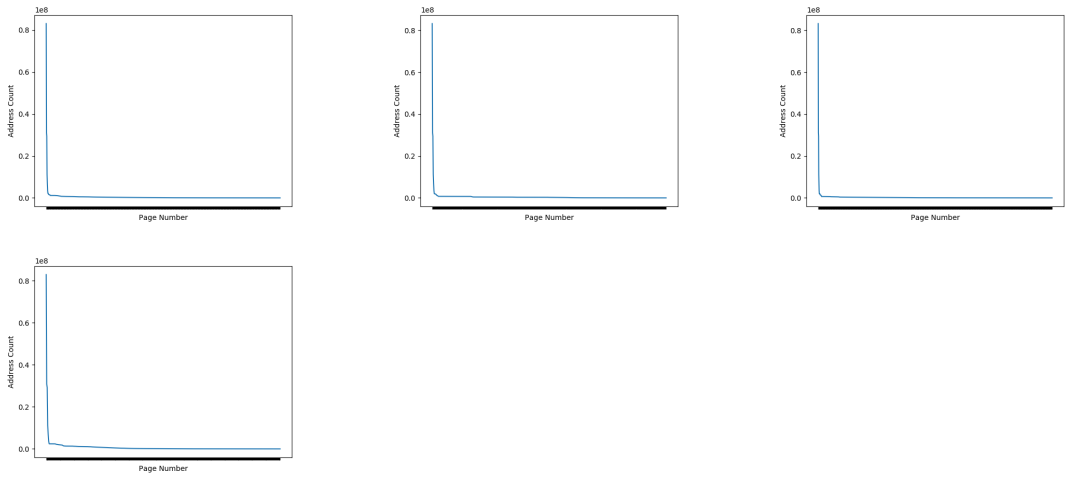


**Figure 10.1.** 456.hmmer

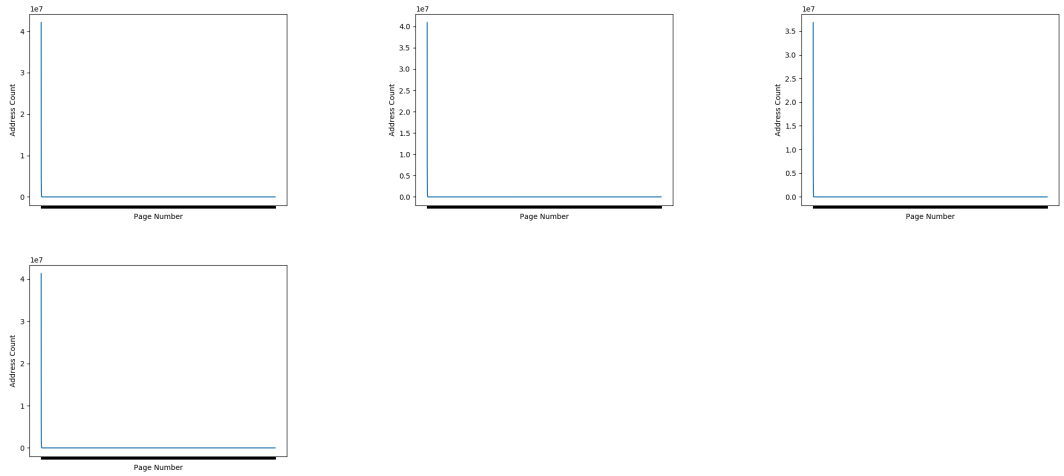


**Figure 10.2.** 458.sjeng.hs

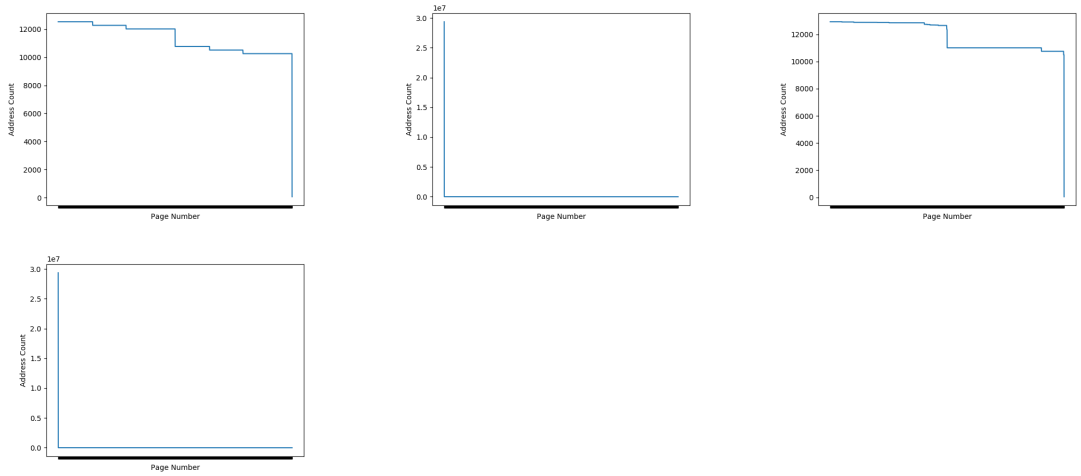


**Figure 10.3.** 462.libquantum

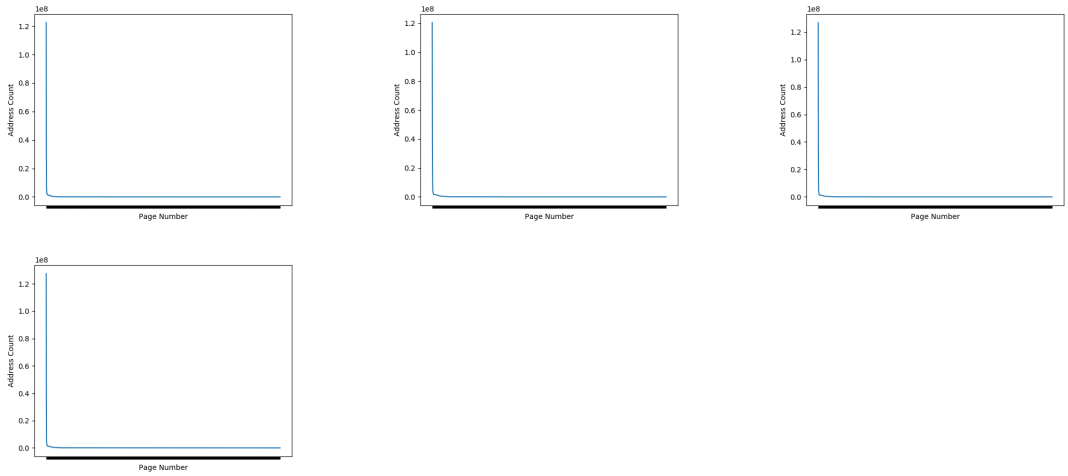


**Figure 10.4.** 464.h264ref.hs

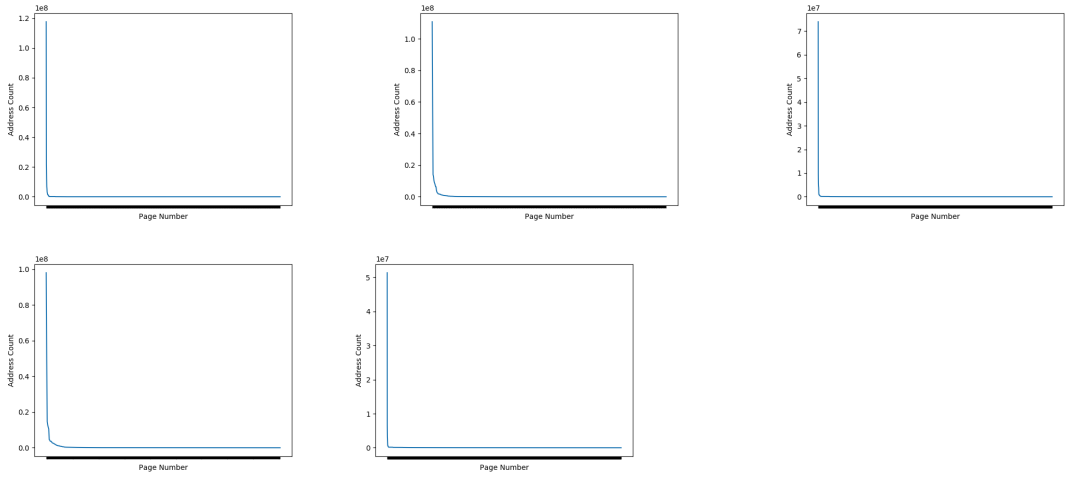


**Figure 10.5.** 465.tonto

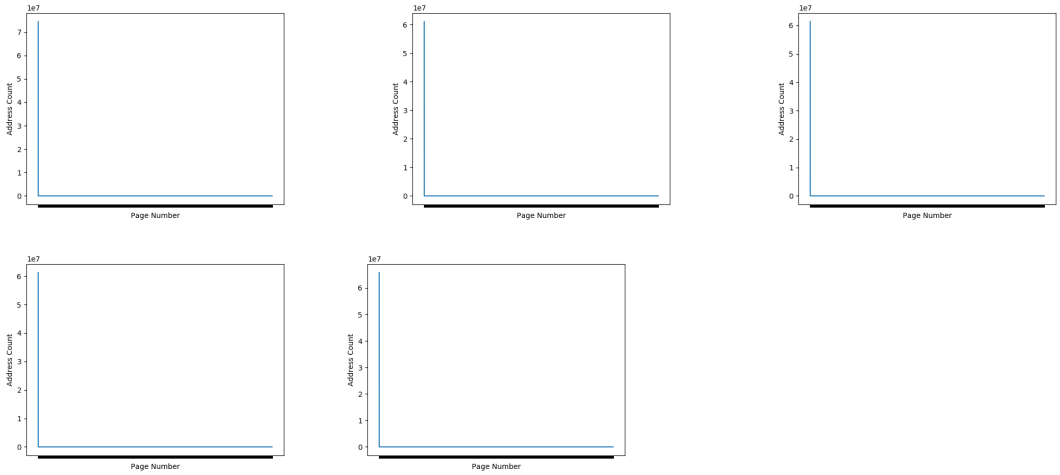


**Figure 10.6.** 470.lbm

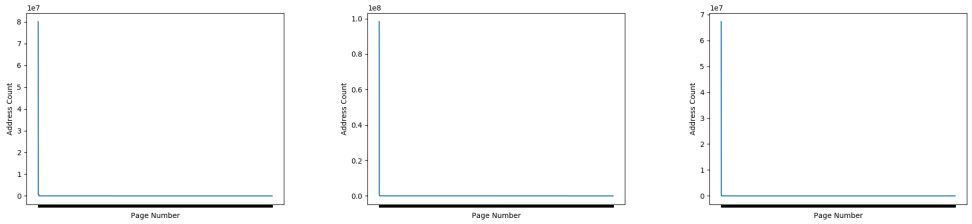


**Figure 10.7.** 471.omnetpp

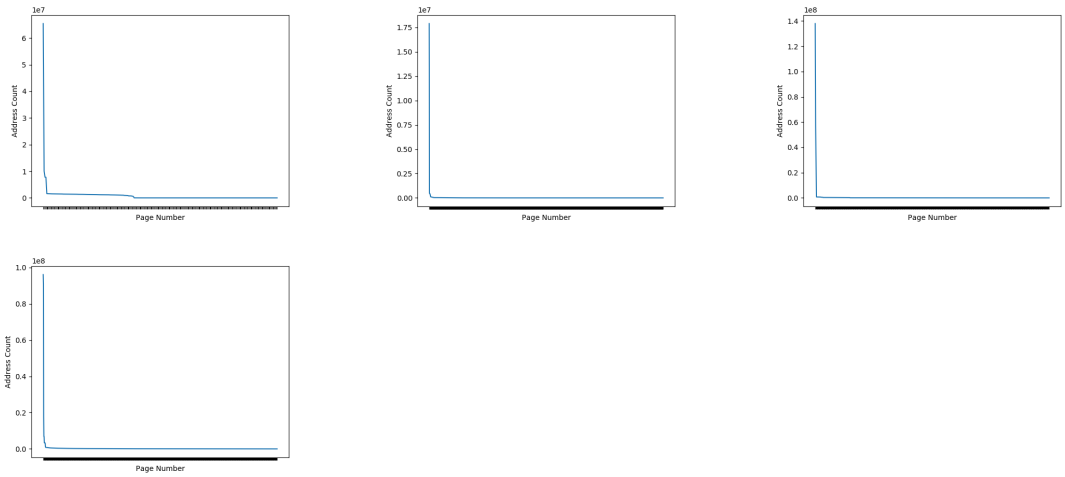


**Figure 10.8.** 473.astar

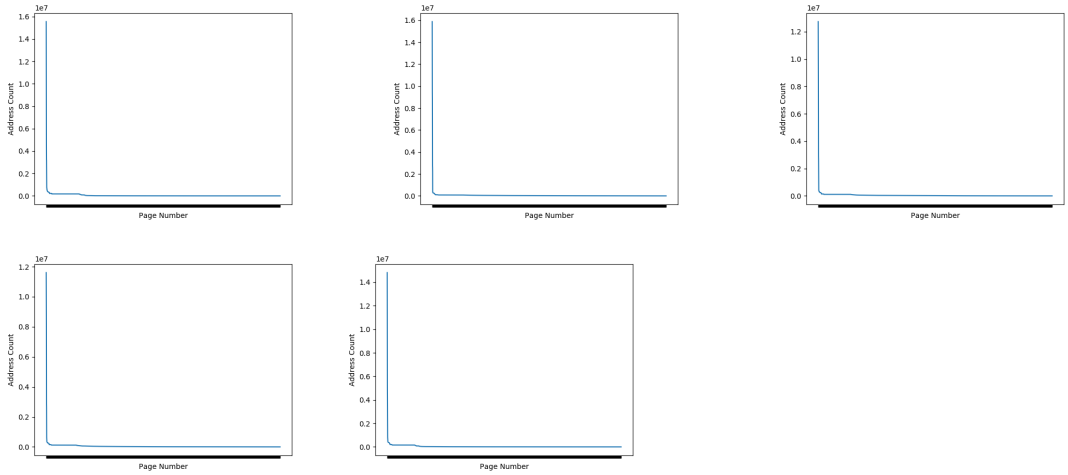


**Figure 10.9.** 482.sphinx

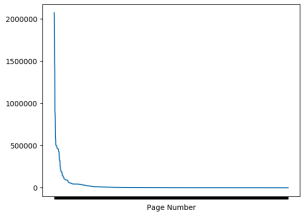


**Figure 10.10.** 483.xalancbmk

# Bibliography

[1] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 631–644, New York, NY, USA, 2017. ACM.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[3] Thaleia Dimitra Doudali and Ada Gavrilovska. Comerge: Toward efficient data placement in shared heterogeneous memory systems. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, pages 251–261, New York, NY, USA, 2017. ACM.

[4] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of*

*the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 15:1–15:16, New York, NY, USA, 2016. ACM.

[5] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[6] Mohammad Hossein Hajkazemi, Mohammad Khavari Tavana, Tinoosh Mohsenin, and Houman Homayoun. Heterogeneous hmc+ddrx memory management for performance-temperature tradeoffs. *J. Emerg. Technol. Comput. Syst.*, 14(1):4:1–4:21, September 2017.

[7] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[8] Xu Ji, Chao Wang, Nosayba El-Sayed, Xiaosong Ma, Youngjae Kim, Sudharshan S. Vazhkudai, Wei Xue, and Daniel Sanchez. Understanding object-level memory access patterns across the spectrum. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 25:1–25:12, New York, NY, USA, 2017. ACM.

[9] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. *SIGARCH Comput. Archit. News*, 45(2):521–534, June 2017.

[10] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[11] Gabriel H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 453–464, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[13] Matthew Benjamin Olson, Joseph T. Teague, Divyani Rao, Michael R. JANTZ, Kshitij A. Doshi, and Prasad A. Kulkarni. Cross-layer memory management to improve dram energy efficiency. *ACM Trans. Archit. Code Optim.*, 15(2):20:1–20:27, May 2018.

[14] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.

[15] J. T. Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24, Aug 2011.

[16] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis. Exploring the performance benefit of hybrid memory system on hpc environments. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 683–692, May 2017.

[17] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. Rthms: A tool for data placement on hybrid memory system. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pages 82–91, New York, NY, USA, 2017. ACM.

[18] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[19] Du Shen, Xu Liu, and Felix Xiaozhu Lin. Characterizing emerging heterogeneous memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 13–23, New York, NY, USA, 2016. ACM.

[20] Avinash Sidani. Knights landing (KNL): 2nd generation intel xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, August 2015.

[21] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[22] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *CoRR*, abs/1503.07241, 2015.

[23] Wei Wang, Qigang Wang, Wei Wei, and Dong Liu. Modeling and evaluating heterogeneous memory architectures by trace-driven simulation. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?*, MAW '08, pages 369–376, New York, NY, USA, 2008. ACM.

[24] Kai Wu, Yingchao Huang, and Dong Li. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 58:1–58:14, New York, NY, USA, 2017. ACM.