Dynamic Page Migration For Disaggregated Memory Systems

A M. Tech. Report Submitted
in Partial Fulfillment of the Requirements
for the Degree
of

Master of Technology

by

Ankit Kumar
(214101008)

under the guidance of

Dr. John Jose

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Dynamic Page Migration For Disaggregated Memory Systems**" A M. Tech. Report Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Technology by Ankit Kumar (214101008) under the guidance of Dr. John Jose to the Department Of Computer Science AND Engineering, IIT Guwahati - 781039, ASSAM is a bonafide work of **Ankit Kumar** (**Roll No. 214101008**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Dr. John Jose**

Associate Professor,

June , 2025

Department of Computer Science & Engineering,

Guwahati.

Indian Institute of Technology Guwahati, Assam.

Acknowledgements

First, I would like to sincerely thank **Dr. John Jose** for the sheer patience he showed with me, the guidance he provided, and the continuous support through my placement process as well as throughout my project, especially at the end. Without his help, I would've been completely stuck.

I also want to express my gratitude to Mr. Amit Puri for helping me immensely throughout my project and for being so kind to me. Without his help, I would not have been able to do anything at all.

I am also indebted to Kailash and Kartheek for meeting with me whenever I asked them to, clearing my doubts, and helping me with everything. Without them, it would've been extremely tough to figure anything out.

Lastly, I wish to thank everyone I mentioned above once again and my family for being so kind to me, perhaps more than I even deserve.

Abstract

Since the latest workloads keep on growing in their memory requirements, and these requirements are dynamic so, we have to assign memory to them based on their peak memory requirement. This leads to the under-utilization of onboard memory in traditional server systems. Hence Memory disaggregation has surfaced as an excellent alternative to traditional server systems for data centers. In our case, rack-scale memory disaggregation is where each node has a small amount of local onboard memory, and rest of the memory that was once associated with nodes is split into remote memory pools within a rack which themselves are connected via a high-speed interconnection network and are managed by an in-network centralized memory manager which allocates the remote memory to computing nodes. Maintaining the memory access latency low in such systems is a big concern as remote memory is accessed through an interconnection network. In order to combat this memory latency issue, hot-page migration is an approach that's used in hybrid DRAM-NVM memory systems. It can also be used with disaggregated memory systems as well since the main problem of the latency of far memory being higher than the near memory is similar. So page migration approach utilizes the locality of memory accesses by getting the most frequently accessed pages from remote memory to local memory. Existing solutions, however, often cause non-profitable migrations due to the use of different metrics used while identifying hot and cold pages in remote and local memory, respectively. Also, They do not change their page migration behaviour depending on the memory access patterns during the execution of a workload or for different workloads having different access patterns.

Based on these observations, we propose DPM, an dynamically adjusting page migration approach for disaggregated memory systems. The main idea is to consider the temperature of data across the whole local-remote memory space when figuring out whether a page migration should take place or not between local and remote memory. In addition, DPM readjusts to workload memory access pattern change by dynamically adjusting page migration parameters for that compute node. We perform experiments on a custom-built cycle-approximate system simulator that models a rack-scale memory disaggregation environment. The results show some improvement in the system's performance for HPC workloads when compared to the system without page migration. It also brings down the average memory access latency in multiple workloads by some extent by making sure a higher percentage of accesses are served from local memory rather than remote memory, which in turn also reduces the time it takes to serve from remote memory due to less contention at remote memory pools.

Contents

List of Figures

Chapter 1

Introduction

As the latest workloads keep growing in their memory demands and the memory capacity wall has limited the performance of server systems used in data centers and high-performance facilities.[23]. This large computing ability allows multiple memory-intensive applications to run simultaneously, but since these memory-intensive applications themselves are dynamic as in they change during the workload's execution having different requirements of memory at different stages of its execution. This change in memory requirement often leads to under-utilization of memory on traditional server systems, as shown in Fig.1.1 having a fixed onboard memory cause we have to over-provision memory resources considering the peak memory requirements. This under-utilization is cause all the allocated memory to a workload is not used at all times, and there is a huge imbalance between utilization and allocation of the memory that's been allocated to a workload.[31]

Hence rack-scale memory disaggregation has emerged as a solid alternative to traditional servers for the future generations of data centers[22],[28], which decouples the memory resources from the rest of the onboard server resources. This separation allows capacity addition and memory up-gradation independently from the other server hardware. In disag-

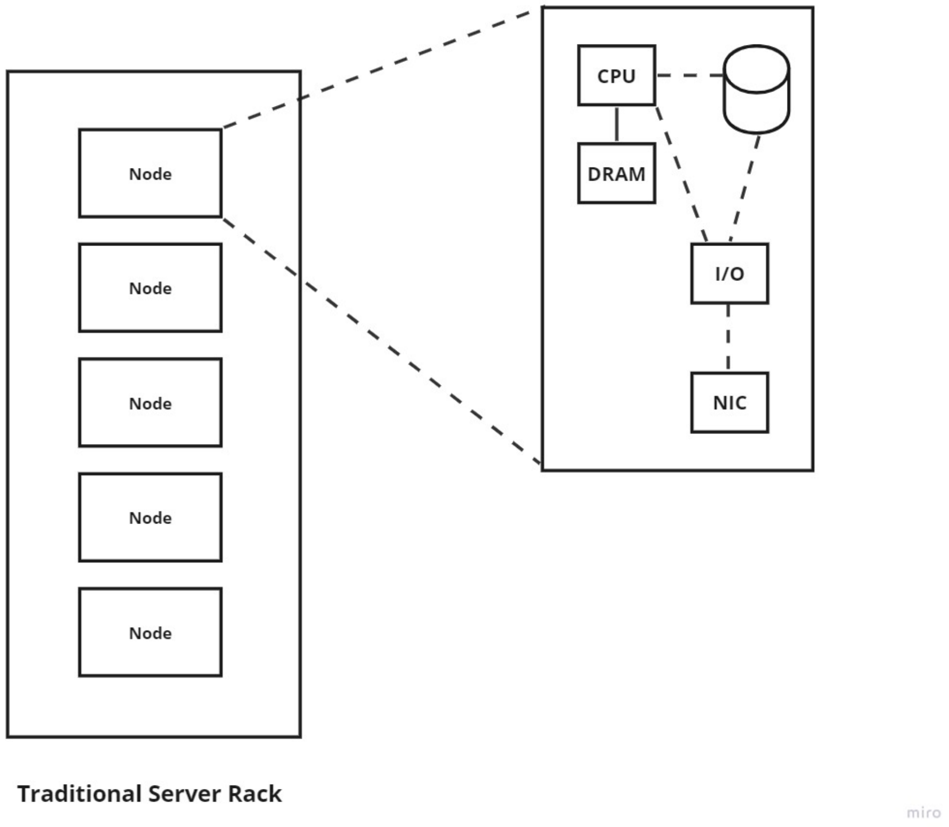

Fig. 1.1 Traditional System

gregated memory systems, compute nodes with small local memory can access a large pool of remote memories in the same rack through high-speed interconnects. Shown in Fig.1.2. Disaggregated memory design gets the hardware support from fabrics such as Gen-Z [14],[20], to attach on-network resources. It uses an in-network centralized memory manager unit hosted on a programmable switch to manage remote memory. This gives us the flexibility to just allocate memory to workloads as and when it's needed, and it also allows nodes to be able to share the memory resources which were held by single workloads in traditional server systems. Gen-Z is an architecture that implements a memory pool system with a large memory capacity. Here local and remote memories are organized in a linear address space.

But it's also fair to see the caveats as well, which are high remote memory access latency which can't match the onboard local memory speed, contention at remote memory pools, etc. To overcome high latency, disaggregated memory system requires some optimizations to be able to bring down the average memory access latency.

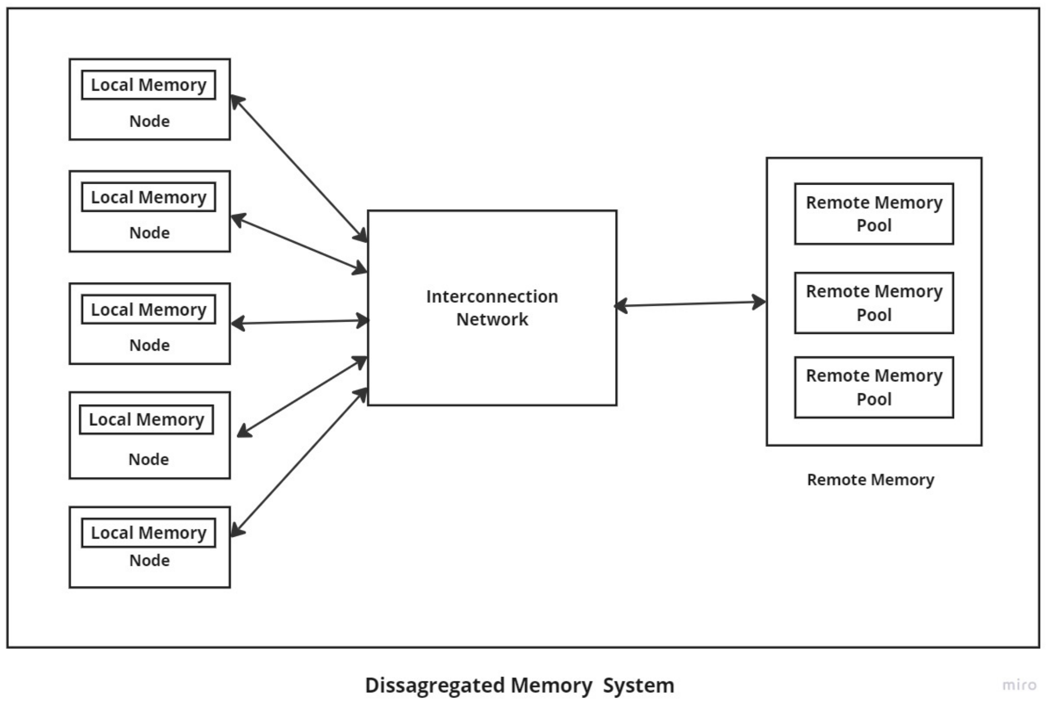

Fig. 1.2 Server Rack in Disaggregated Memory System

Page migration is one such optimization that has been used extensively in hybrid memory systems e.g., DRAM-NVM systems; this approach suggests that we bring the most frequently used pages from the slower far NVM to the faster near DRAM [27],[24]. In this report, we exploit an on-the-fly(OTF) hot page migration in a disaggregated memory rack system, where multiple nodes run different workloads, and each node might access memory in a different pattern than other nodes. Hot page migration implementation in such a system that can understand the need of each compute node requires hardware as well as software support from the in-network CMMU [22] and Operating system at the compute node.

Epoch-based page migration requires specific page migration parameters such as page migration threshold(PMT), epoch length and number of pages to migrate(NPM) for hot page selection and carrying out the page migration. If the parameter selection was random and doesn't take into account a workload's memory access behavior, the page migration could easily downgrade the system's performance. In addition, compute nodes that execute different workloads cannot benefit from the same migration parameters as their corresponding memory access patterns differ. It's possible that a few nodes might get better performance, and others end up getting worse performance when compared to no page migration at all. On top of that, this different response to a set of parameters might also be dependent on how friendly your workload's memory access pattern actually is to those parameters meaning if the workload has a uniform memory access pattern or shows no locality, it will show worse performance than a workload that shows more locality while accessing memory. Even within a single workload, different parts of a workload might show different extents of migration-friendliness[4]. Due to such constraints and the observation made by Adavally et al. [16] that migrating recently accessed hot pages results in better performance than migrating the "hottest" pages with accesses accrued over a longer period. We propose an OTF page migration approach that immediately gets rid of two parameters NPM and epoch length, but just keeping a static PMT is not suited cause of the fact that memory access patterns don't stay the same among workloads and even during the execution of a single workload so there is a requirement of an adaptive page migration approach that depending on how friendly or non-friendly a workload is it can change how much or how little page migrations should take place respectively. In our case, depending on how beneficial or degrading each migration is to the system performance, our approach will increase or decrease the page migrations respectively. This approach, in case of some unwanted behavior, can be set back to some initial state. In our case, if PMT goes below a certain point, it is reset back to the initial PMT. In OTF page migration, the basic concept is instead of waiting for a certain epoch to finish and migrate all the pages that met the PMT criteria during that epoch and stalling the CPU and then migrating the pages and the end of the epoch but instead of migrating the page as soon as it gets hot. However, the issue being since a page can be migrated at any time the CPU stall can occur at the moment, which can hugely degrade the system performance. Hence, we find alternative approaches to hide some of these extra delays in page-swap latency that causes long CPU stalls. We also implement a TLB/cache-aware victim page selector that efficiently works in the background to find local cold pages that can be swapped with hot remote pages when a certain hotness criterion is met.

We also noticed that existing approaches, when comparing the hotness of pages in local and remote memory, use separate measures like in PACT [19]. They use counters for finding the hot pages in the remote memory but use reference bit to find the victim page in the local memory due to extra 32 or 64 bit they might have to store for each page in local memory, but the problem with this approach is it's possible that the victim page that you might be evicting might itself be hotter than the hot page you're bringing from the remote memory. This not only leads to the further page accesses of the page you just transferred to remote memory to be slower, but it may happen that the page you just sent to remote memory was so hot in fact that it meets the migrating criteria again and gets back into the local memory again anyways this is what we call a non-profitable migration. We also took into account that just because two pages have been accessed a similar number of times doesn't mean that they are equally hot. This is entirely based on access time locality; the page that's been accessed more recently should be more hot/popular. Thus, while updating the page hotness of a page, we shouldn't just add one. Rather, we should also take into account how many cycles it has been since this page was last accessed. In order to achieve this, we use an attenuation coefficient that lowers the page's popularity if this page has not been accessed for many cycles. We study our page migration system through a simulation where reducing average memory access latency is our main goal. We use a disaggregated memory simulator that can model such behavior by deploying multiple compute nodes with the remote memory pools, and a central memory manager, which is also the model for rack-scale memory disaggregation.

Chapter 2

Background and Motivation

A large amount of research work has been carried out in the past [12],[32],[35] for hot page migration in hybrid memory systems such as NVM-DRAM systems, where DRAM supports low latency access but has more cost and energy concerns, and NVM that comes with large capacity at a fraction of the cost but has more access latency which causes an issue cause it has to allow loads and stores to directly access the slower NVM of last-level cache (LLC) misses. To overcome this high memory access latency, page migration is used to reduce the number of memory accesses happening from NVM by migrating the most frequently accessed pages to DRAM so most of the future accesses can be done from DRAM, reducing the average memory access latency. B. Yang et al.[27] utilize the frequency of writes and reads to gauge the hotness of a page. Y. Wang et al.[11] proposed migrating huge pages in tiered memory systems, which suffer from excessive misses in TLB. B. Want et al.[38] applied hot-page migration to cloud computing platforms and devised for a virtual machine a hot page capturer to reduce the remote page faults when a remote node is restarted.

Disaggregated memory systems can be thought of in a similar fashion where we have similar latency constraints as hybrid memory systems, thus giving us an opportunity to exploit page migration. Although both local and remote memory uses DRAM in disaggregated systems, the interconnect latency is what causes the remote memory accesses to be comparatively slower. Hammond et al. [19] explored the epoch-based page migration support when it comes to disaggregated memory systems and proposed a global memory controller that holds hot pages for each node. This was a static page migration approach where the controller performed page migration to all compute nodes at a fixed small interval/epoch using fixed threshold values for a fixed number of pages. Although this work implemented page migration in disaggregated memory for the first time, but it has a major issue with regards to the fact that the controller lacked any intelligence that was needed in order to be able to serve multiple nodes, each executing a different workload each having a distinct memory access pattern of its own. Hammond et al. performed an exhaustive search where they used different parameters to find the best parameters for each workload, which is not possible when it comes to a real-time scenario. Having fixed parameters will not solve our problem, as the same parameters for different workloads have different effects on performance, both significantly positive and significantly negative[19]. In epoch-based page migration during an epoch, whichever pages were accessed the most, also termed as hot pages, are moved to local memory right before the beginning of the next epoch if they pass a certain threshold also known as page migration threshold. But the problem with the epoch-based approach, as noticed in hybrid memory systems, is perfectly summed up by the statement made by Adavally et al.[16] that migrating recently accessed hot pages results in better performance than migrating the "hottest" pages with accesses accrued over a longer period. Hence they proposed an On-The-Fly (OTF) page migration in hybrid memory systems that relies on OTF migration of heavily accessed pages from slower memories to faster memories. Unlike epoch-based approaches, which migrate several pages together at regular epochs, we migrate a page immediately when it becomes hot. This helps us in achieving more hits from the fast memory. In the OTF approach, migration can take place anytime. Hence it's important that the program does not stop as soon as a page migration begins dealing with this problem of migration without stopping by devising a migration technique that takes place in the background. Even though the OTF migration was implemented on hybrid memory systems, the same concepts can also be used when it comes to disaggregated memory systems.

Before diving deep into the implementation of any page migration approach in disaggregated memory systems, we should also be aware of the overheads that are associated with page migration, one of which is page migration introduces multiple delays at the compute nodes. Firstly, swapping of physical memory pages demands updating page table entries (PTEs) with new virtual to physical address mappings. Updates to the PTEs are always performed in pairs, as one of the hot remote pages is swapped with a local cold/victim page, except for the case the victim page is at at a memory location that is still unallocated. Once the PTEs are infact, updated, TLB entries have to be invalidated so our address translation for future memory references to swapped pages can occur correctly. The process of TLB invalidation is known as TLB shootdown, which involves an inter-processor interrupt (IPI) [26] to other cores for performing the invalidation at their local TLB. Also similarly, cache invalidation is also needed on all the cores for whatever cache blocks that are using the old physical tag. Cache invalidation and TLB shootdown are expensive processes that introduce long CPU stalls until these in-validations are performed and acknowledged back from all the cores. As pointed out by [33],[37] TLB-shootdown time increases with the number of CPU cores, takes around 4, 5, 8 and 13 µs for 4, 8, 16, 32 cores, respectively on an AMD 32-core processor. Similarly, re-accessing a cache block from memory would take about 40-60 ns also, there is a possibility of having multiple cached blocks belonging to a single remote page.

So, as discussed earlier, keeping migration parameters the same during the entire execution of a program when it keeps on changing its memory access pattern is not ideal and will degrade the system's performance at one point or another during the workload's execution and might take away any benefit you got from using page migration. Although the aggregate average access to pages varies in among different workloads, as can be seen in Fig. 2.1 , it remains similar in every small interval, as we can see in Fig. 2.2 we make use of this fact for selecting the hot pages and make use of certain hot page thresholds in smaller intervals rather than throughout the workload execution. Even though there is a change in memory access pattern when you move across these small intervals, our algorithm successfully deals by resetting the page migration parameters.

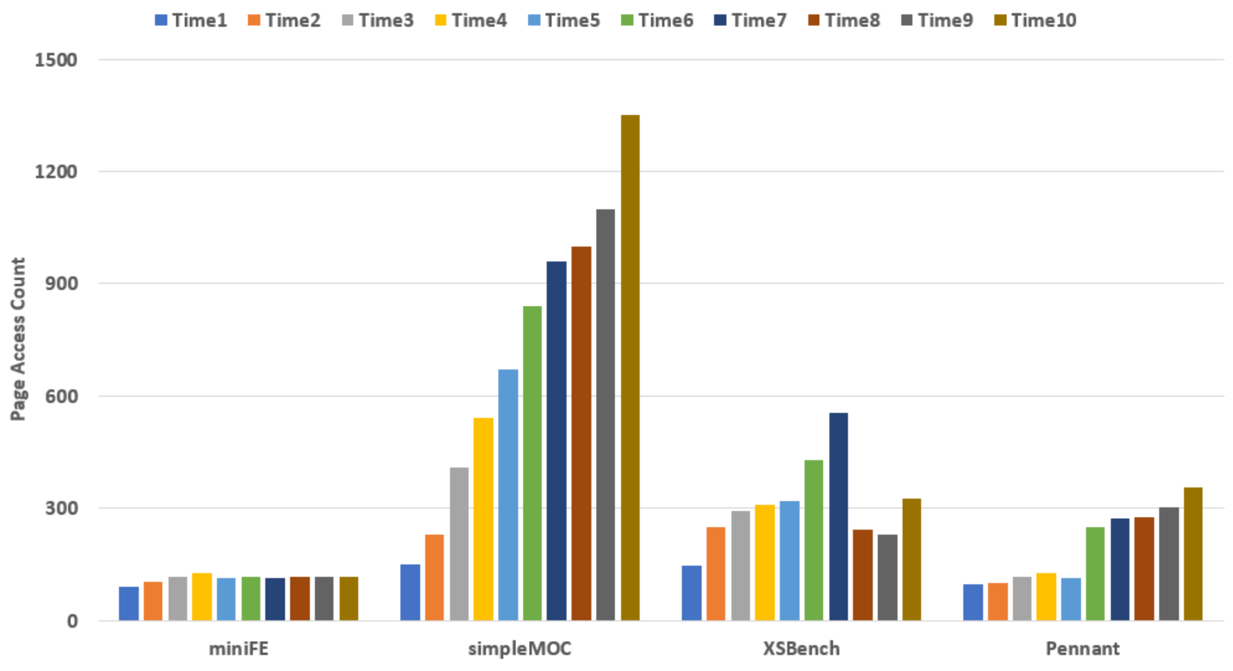

Fig. 2.1 Average Page Access Count to Top 10% Pages (Aggregate)

Most of the past work is limited to DRAM-NVM systems, where we provide page migration to such systems. Disaggregated systems have similar constraints, And the need for an intelligent adaptive migration policy that changes its parameters based on a node's memory access pattern is evident. Further, hardware and software support introduced in the past work can be hidden with system support and extra migration logic. We also look at migrations that are non-profitable that degrade system performance due to the fact that the metric used for measuring the hot and cold pages is different, and sometimes it just so happens that the cold page you are trying to evict from the local memory might itself be hotter than the

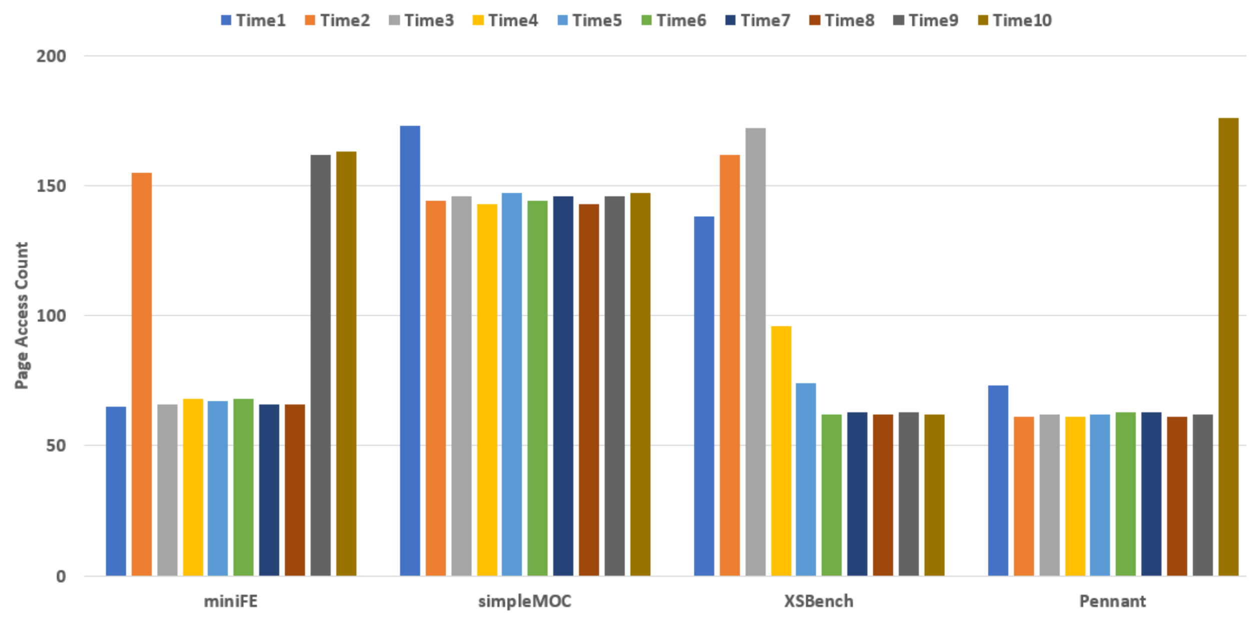

Fig. 2.2 Average Access Count to Top 10% Pages (Per-small-interval)

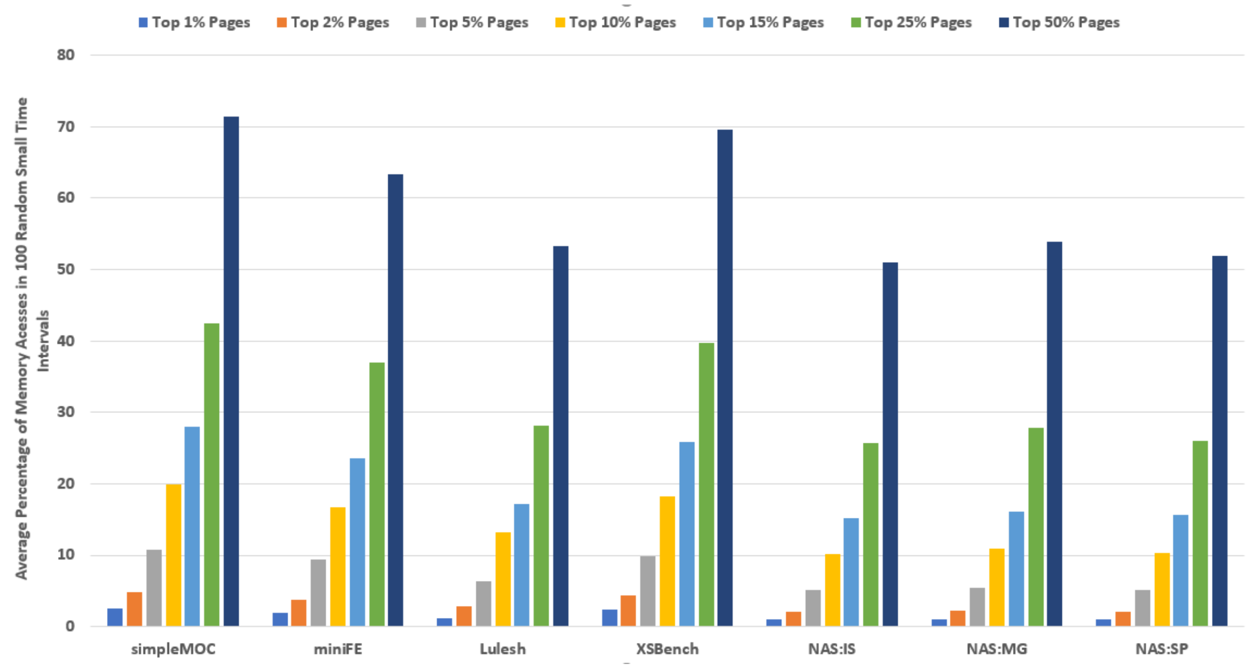

Fig. 2.3 Average Percentage of Memory Acesses in 100 Random Small Time Intervals

page hot page you are trying to bring from the remote memory. We focus on an On-The-Fly page migration system that uses a dynamically adaptive policy that changes its parameters depending on the workload's memory access pattern and migrates a page as soon as it gets hot. One more reason why page migration is necessary is to exploit the memory accesses happening from a small amount of pages as shown in Fig. 2.3. As it can be seen in the image some workloads are migration friendly and some just aren't our goal is to improve the system performance in the cases where most of the memory accesses occur from a few pages and not degrade the system performance too much where the conditions aren't as favorable.

Chapter 3

Literature Survey

Previously memory in server systems had to be kept on board as a way of overcoming the limitations that were plaguing these traditional servers, Ivy and Pearce et al. [29] where they analyzed over 2 million jobs on HPC clusters and showed their results and also questioned can disaggregated memory benefit HPC systems as well as what kind of applications will benefit from this memory disaggregation. Disaggregated memory systems such as network-attached memory is quite a popular disaggregated architecture as discussed by William and Bernardoni at el. [3] and Youngmoon and Zhang et. al [9] Howeever Ivy and Pearce et al. did a large scale study to understand how exiting workloads on production HPC systems utilize the memory resources and what they realized after looking at over 2 million jobs on hpc clusters that most of the jobs only utilize a small fraction of memory resources ,and for more than 90 percent time, a node utilizes less than 35 percent memory capacity.

Need for memory disaggregation is seen while looking at the memory utilization of jobs running on the system due to the unflexible nature of traditional servers with regards to memory being fixed and on top of the modern workloads having increasing memory demands that grow and shrink over time has led to under-utilization of memory in traditional server systems having fixed onboard memory. Memory disaggregation works best when jobs can be

can be prefetched from remote memory to local memory complete memory disaggregation is seen to be not as good for most applications hence having a small local memory with every node is a better idea for most of the applications and Memory disaggregation shows huge improvements in applications which are highly localized in their memory accesses.
When the different memories in a tiered-memory system have same access latency and different like 3-D DRAM [1] and DRAM [6] so Chiachen Chou et. al [5] proposed Bandwidth-Aware Tiered-Memory Management (BATMAN), a runtime mechanism that manages the distribution of memory accesses in a tiered-memory system by explicitly controlling data movement. When the latency of near and far memories (near would be like 3D-RAM and far would be like DRAM) is almost same but they have different bandwidths then using a traditional way of accessing data where you first bring to the closest memory and then read it is not the most optimal way of accessing data rather what we should do is read from both the memories and split the accesses in a way proportional to bandwidth. BATMAN in cache mode [6] (meaning data will be first brought to the near memory from the far memory and then accessed) BATMAN will control the near memory access rate by partially disabling the cache. If the near memory access rate is higher than target access rate (TAR) we disable some of the cache sets and if the near memory access rate is lower than TAR we enable those disabled cache sets this is how bandwidth is managed. BATMAN in flat mode [7] (meaning near memory is used as part of the memory space and relies on the OS to perform dynamic page migration for data locality) BATMAN that explicitly controls data movement to meet the TAR to systems in the flat mode. In flat mode page migration, in either direction from the NM to the FM or from the NM to the FM, is in the control of the OS. If the near memory access rate is higher than TAR we will move pages out of near memory to the far memory (referred to as page downgrade). If the far memory access rate is lower than TAR we will move pages out of the far memory to the near memory (referred to as page upgrade).

Coming to disaggregated memory itself we have mentioned previously what's the need for it and Kommareddy et al. [19] have mentioned a epoch-based page migration technique they state that since disaggregated memory systems are expected to be using dense,power-efficient Non-volatile memories (NVMs) e.g., Intel's and Micron's 3D X-Point[2] as their remote memory pool/pools and DRAM as their local memory and since NVMs are slower compared to DRAMs so there is a need of moving frequently accessed pages from global memory to local memory would benefit both node and system overall performance, due to the reduced contention at the global memory and the fast response time of local memory. Here authors have used epoch(a small time interval) based page migration.

Here during an epoch whichever pages were accessed the most also termed as hot pages are moved to local memory at the beginning of the next epoch if they pass a certain threshold also known as page migration threshold. Page migration also comes with it's own set of challenges as well such as TLB shootdowns (TLB shootdowns are necessary to maintain coherency of TLB entries) and setting a proper page migration threshold and the number of pages to migrate per epoch. To see what pages are hot they have employed PACT (Page access count table) which counts the accesses to each of the pages made by nodes and To keep track of cold pages they check for which pages were referenced recently we will discuss all this later in the report. But what the problem with that epoch-based approach is perfectly summed up by the statement made by Prodromou et al. [30] that migrating recently accessed hot pages results in better performance than migrating the "hottest" pages with accesses accrued over a longer period.

Hence Marko and Kavi et al. [16] in on-the-fly (OTF) page migration they present a Heterogeneous Memory Architecture (HMA) that relies on OTF migration of heavily accessed pages from slower memories to faster memories. Unlike epoch-based approaches, which migrate several pages together at regular epochs we migrate a page immediately when it becomes hot this helps us in achieving more hits from the fast memory. In OTF approach, migration can take place anytime, hence it's important that the program doesn't stop as soon as a page migration begins dealing with this problem of migration without stopping by devising a migration technique that takes place in the background with the help of a specialized hardware that is transparent to the user program and also OS called migration controller. Page migration requires changes to PTEs to reflect new physical address we call this process address mappings, invalidating TLB entries and cache entries they call this process address reconciliation while this is fine in epoch-based page migration it won't fly in OTF page migration cause this is too much of a overhead. Hence they propose a low overhead technique with the help of a special hardware migration controller that enables migration of pages while doing address reconciling for different pages in a cycle interleaving fashion, making it not necessary to perform the process in separate phases.

Chapter 4

Proposed Work

In this section, We discuss our proposed OTF page migration policy in detail that changes the threshold values based on the workload's memory access patterns.

4.1 Dynamic Page Migration

Since existing approaches use different methods for identifying hot and cold pages in local and remote memory, respectively, it can cause a lot of non-profitable migrations (Where the victim page in local memory itself is hotter than the hot page in remote memory). We propose an On-The-Fly adaptive page migration approach that uses the same hotness measure for all the pages, regardless of whether they are in local or remote memory. Dynamic page migration, as shown in Fig. 4.1, mainly consists of three steps: hot page identification using the same hotness measure, which helps us in identifying which page to migrate, page migration which is how a page should be migrated and dynamic adjustment of hotness parameters, which helps us in accommodating different memory access patterns and workloads by dynamically adjusting parameters based on how profitable a migration actually is to our system.

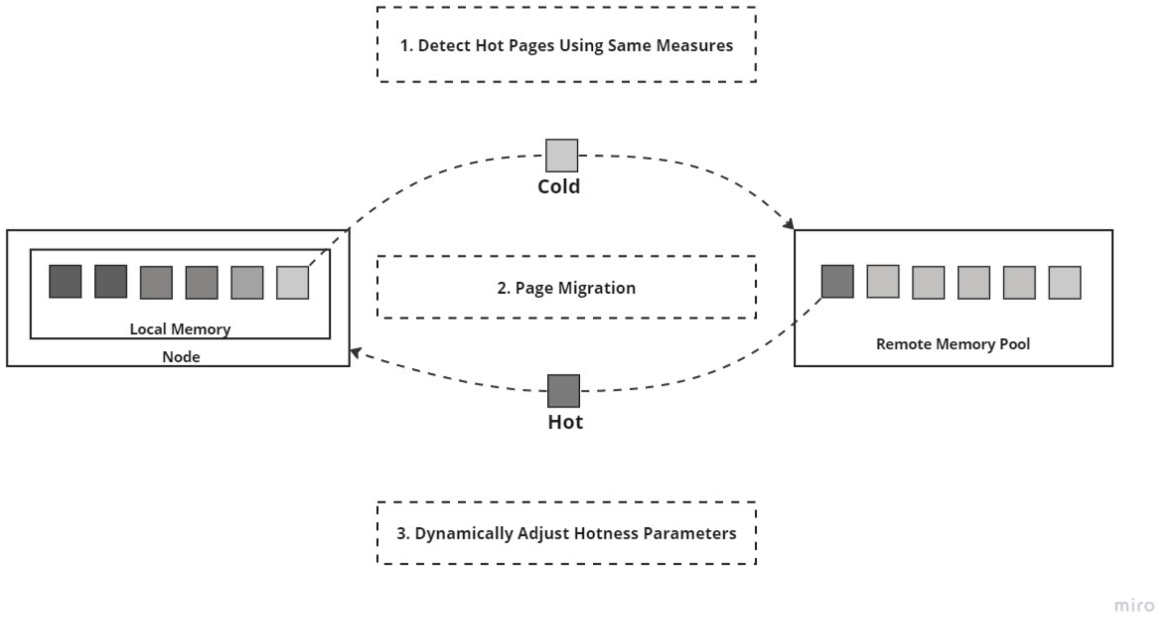
1. Detect Hot Pages Using Same Measures
Cold
Local Memory
Node
2. Page Migration
Remote Memory Pool
Hot
3. Dynamically Adjust Hotness Parameters
miro

Fig. 4.1 DPM Architecture

4.1.1 Identification of Hot Pages

Identification of hot pages is a critical part of any hot page migration approach. It helps us in quantifying the hotness/popularity of each page in local and remote memory. Other approaches basically equate page hotness directly to the number of page accesses. DPM also increments page hotness by one on each new memory access. But, based on how many cycles ago the current page was last accessed, the overall hotness of the page is determined. This is to accommodate for the fact that the page accessed more recently should be more popular than a page that was accessed some while ago even if the number of accesses to both the pages are almost equivalent. Thus, while updating the hotness of a page on a page access, DPM also uses a reduction/attenuation coefficient which helps us in lowering the popularity of pages on which the accesses occurred not so recently. As apparent in Equation (1), where $old_{hotness}$ and $current_{hotness}$ represent the old hotness and current hotness of each page, respectively for all pages in local and remote memory, t represents the reduction coefficient that changes based on access time locality meaning its value is inversely proportional to the number of cycles current page has not been accessed.

$$current_{hotness} = old_{hotness} * t + 1 \tag{1}$$

Equation (2) shows how DPM computes the reduction/attenuation coefficient t. In the Equation, $global_{access\_count}$ represents the total access count a node in our rack has made either to local or remote memory, $page_{access\_count}$ represents the $global_{access\_count}$ value that was recorded when this current page was last accessed, $local\_memory_{size}$ represents the number of pages in the current node. If all local memory pages are supposed be accessed in each cycle $\frac{global_{access\_count} - page_{access\_count}}{local\_memory_{size}}$ represents the number of cycles since a page has not been accessed. So, Finally, t represents the reduction coefficient or decay ratio for a page's access popularity depending upon how long it has been since this page was last accessed. It will be used to update the hotness of a page on each access to that particular page.

$$t = \frac{old_{hotness}}{old_{hotness} + \frac{global_{access\_count} - page_{access\_count}}{local\_memory_{size}}} \tag{2}$$

In, Summary, hot page identification is done not only through the number of accesses that have been made to a particular page, but the access time locality has also been taken into consideration, which allows one to find truly hot pages.

4.1.2 Page Migration

The page migration step migrates the pages, each using the same measure for hotness in local and remote memory. In remote memory, when a hot page is identified, it should be migrated to local memory so we can optimize average memory access latency. If the local memory has free unallocated pages, we can directly migrate the hot page to one of those locations, but if the local memory is full, a victim page will be found and evicted to make space for the hot page coming from a remote memory. In existing approaches identification of hot pages in remote memory and cold/victim pages in local memory is done through different measures. Like hot page might be found using counting the number of accesses, but victim page is found through if a page was accessed in the last cycle or not [19]. These approaches lead to a case where the hotness between the hot page and victim page can't be compared correctly among themselves as they are using different measures to find hot and victim pages, which might lead to many costly non-profitable migrations, which happens when the victim page itself is hotter than the hot page you brought from a remote memory. To solve this problem, We first of all use the same measure for computing hotness for all local and remote pages, which not only allows us to find hot and victim pages but also allows us to compare among the chosen hot page and victim page and make the right decision on whether a migration should even take place or not basically when the hotness of the victim page in local memory is found to be lower than the hotness of the page in remote memory only then the migration will take place.

To further reduce non-profitable page migrations, we set a couple of other thresholds one is *cold_page_migration_threshold* (CPMT) and *hot_page_migration_threshold* (HPMT). We use these two thresholds to prevent some other performance-degrading migrations. HPMT is used to prevent migration of pages that are not accessed/hot enough in remote memory, and CPMT saves us from evicting pages from local memory that are still hot or not cold enough to be evicted. Again keep in mind that CPMT and HPMT are just thresholds. We are still computing the hotness of every page using Equation (1). In Layman's terms, a page can be migrated from remote to local memory only when these three conditions are fulfilled. Firstly, the hotness of this hot page in remote memory must be greater than HPMT. Secondly, the hotness of the victim page you are about to evict from local memory must be smaller than CPMT. Thirdly hotness of the page in remote memory must be larger than the hotness of the victim page in local memory.

4.1.3 Dynamic Adjustment

Workloads change their memory access behavior overtime, and different workloads have different memory access patterns. So, We cannot keep HPMT and CPMT the same when executing different workloads and even throughout the execution of a single workload because it, at some point, might negate any performance benefit we might be able to get from using page migration. So, It is extremely crucial that these parameters change according to the memory access pattern of a workload. We adjust these thresholds (HPMT and CPMT) on it's own to suppress or promote page migration for a node according to the real time migration profit we may get from each migration. Each node in the rack will have its own HPMT and CPMT that will change independently of other nodes in the rack because the workloads executing on them might be different, or they might be in different stages of execution even if they are executing the same workload.
Migration profit can be defined as the total time saved by all the accesses that till now took place from this page after it has been migrated to local memory rather than all these accesses were to occur from global memory minus the time required for migrating the page from remote memory to local memory and time taken to perform TLB shootdown as well as TLB and cache re-access time for those invalidated TLB/cache entries and if a wrong page is transferred to remote memory than all the accesses that occur from that page in remote

memory are also a system overhead that needs to be taken into account represented by $R_{mem\_accesses\_time}$ in equation 4.1.3. In order to compute migration profit, we will definitely need to keep track of the number of page accesses to all the pages in local memory so that we will be able to calculate how much time we have saved after the page has been migrated to local memory. Equation (4.1.3) shows the migration profit (M) calculation for a page that's being accessed, and P_{count} is the number of page accesses that occurred from this page after it was migrated to local memory. $access_{latency\_remote}$ and $access_{latency\_local}$ represents the time taken to access a page in remote memory and local memory respectively. $P_{count} * (access_{latency\_remote} - access_{latency\_local})$ represents the time saved by transferring this page to local memory and letting all the accesses happen from local memory rather than them taking place from remote memory. $P_{migration\_time}$ represents the time taken to migrate the hot page from remote memory to local memory added with the time taken to migrate the victim page from local memory to remote memory it also takes into consideration if the victim page was empty to begin with then the time taken to move the page from local memory to remote memory won't be added. TLB re-access overhead is computed by using the value of TLB_miss cycles. The equation 4.1.3 is calculated using $access_{latency\_local}$ set to 50ns and $access_{latency\_remote}$ set to around 250ns. As for cache re-access overhead is local memory access time into the number of invalidated blocks due to migration. However, the system does not have any information regarding the number of invalidated cache blocks as it is tough to figure out the cache overhead considering multiple levels of caches and overlapped accesses to memory. We assume that adding a constant factor for it is not the right way to go and choose to remove this factor all together the thought process being while it is a expensive operation but it in itself will not be able to switch many page migrations from being beneficial to degrading.

$$M = P_{count} * (access_{latency\_remote} - access_{latency\_local}) - P_{migration\_time} - TLB_{shootdown\_time} - TLB_{re\_acc} - R_{mem\_accesses\_time}$$

Now at some point, the hot pages you brought from local memory will finally become cold as the number of accesses happening from them go down, and it's their turn to be replaced by some other hot page coming from remote memory. When that happens, and one of those pages is about to be evicted from local memory, at that point we go ahead and calculate the migration profit is calculated.

If the migration profit is less than or equal to zero. In that case, that means this migration just degraded the system performance cause the migration cost itself was greater than all the benefits we got from all the accesses that were made from this page after it had been brought to the local memory in this case what we assume is maybe the pages that we are bringing into local memory they might not be hot enough and the pages that we are evicting from the local memory maybe they are not cold enough and to accommodate for that fact we increment HPMT by one to prevent us from bringing certain pages to local memory from remote memory that are not even hot enough which leads to more pages staying in remote memory itself and decrement CPMT by one so more pages don't become cold in local memory or more pages don't get evicted from local memory.

If the migration profit is larger than zero, that means the benefit of migrating this page to local memory exceeded any and all costs associated with page migration. In this case, we will try to make more of such page migrations happen. In order to do that, we will increment the CPMT by one to make it easier for certain cold pages to be evicted from local memory so it becomes easier for the new hot pages to move in, and we will decrement the HPMT by one so more pages in remote memory become hot and get migrated to local memory.

Also, we do not always need to decrement/increment the HPMT or CPMT by one always cause that indicates that no matter how profitable a migration was to the system performance, it should be treated the same way to a page that barely crossed the $migration\_profit > 0$ mark and same goes for not profitable migrations. In order to deal with this depending on how good the migrations are, we could also increase the aggressiveness with which the HPMT and CPMT increments or decrements. We use two averages of $migration\_profit$, one is $average\_positive\_migration\_profit$ and the other being $average\_negative\_migration\_profit$ Both of these averages will be constantly updated with the calculation of each and every migration profit calculation if the migration profit we just calculated happens to be greater than $average\_positive\_migration\_profit$ then we increment HPMT and decrement CPMT by two else by one and similarly if my migration profit happens to lesser than the similarly calculated $average\_negative\_migration\_profit$ we decrement HPMT and increment CPMT by two else by one.

4.1.4 Victim Page Selection

Whenever a hot page is found in remote memory and is about to be migrated to local memory, OS at compute node needs to make sure that it can find a victim page where the swap can occur. The victim page selector either provides a free unallocated page if it's available or provides a list of pages that have not been accessed for a long time, also called cold pages. We should be quite sure that the victim page is not going to be accessed much after the migration cause then all those accesses will have to be served from remote memory, and if victim pages were to be chosen poorly, it would lead to a huge overhead. Migrating such pages will take away some, if not all, benefits of page migration. Our hotness measure already saves us from cases where we are considering just because a page had high hotness in the past. We won't keep on thinking it's still hot. This particularly was a problem with cases where they would simply use number of memory accesses as a hotness measure.

Another common method is using a clock replacement policy where the re-reference bit that's associated with each page is in memory. Although this way, we couldn't provide an extra chance to each page, still leaving a huge probability for it to be accessed again if it were to be chosen as a victim page just because of the reference bit, and on top of that, clock replacement policy doesn't ensure that this page does not have TLB entry or cached blocks, which will have to be invalidated after migration. We dealt with this limitation by using a cache-aware clock replacement that uses two bits: a cache bit and a reference bit. Whenever a page is accessed, both the cache and reference bit are set to 1. While selecting a victim, if the cache bit is set to 1 and the re-reference bit is not modified. We make the cache bit 0 first which gives more chances to recently accessed pages before we go ahead and select them as victims. We run the victim page selector as a background process which will initially have all the unused/unallocated pages, and the list is updated on every TLB or cache access. So we first find a victim page using this method and then go ahead and compare the hotness which we have computed with the same hotness measure for each page in local or remote memory. One problem that can occur with our case of hotness measure as well is what if a page became really hot in the past but has not been accessed recently we will still keep on thinking it's hot cause hotness of a page only changes when it's accessed so what we did to resolve this was we would find pages through using the above cache-aware method and then instead of comparing the hotness of this victim page directly to the hotness of the remote page but instead we first compute the hotness of the victim page once again using equation (1) as if the page is being accessed but actually it's not being accessed this way we will get a new updated hotness for this page and we will be able to correctly compare between our hot remote page and cold victim page otherwise it might have lead to a case where the victim page was very hot in the past and it didn't get accessed after that but when we got to it through cache-aware policy we and then tried to meet the condition where this victim page must be less hot than the remote hot page or it has hotness lesser the CPMT and failed in any of those conditions and stopped this victim page from going out of local memory when ideally it should have gone out so to counter that we compute the hotness once again and then compare to make the right decision.

4.1.5 Reducing Page Migration Latency

Transferring a page from remote memory to local memory incurs a per page transfer delay of about 1.75 µs. The transfer delay in itself can be handled when it comes to epoch-based page migration approaches as all the migrations take place at a certain time only but in OTF approach migration can take place anytime and during migration new memory requests cannot be sent due to the changes in the physical address of the pages currently undergoing the swap. Some other arrangement is indeed needed through which the CPU can

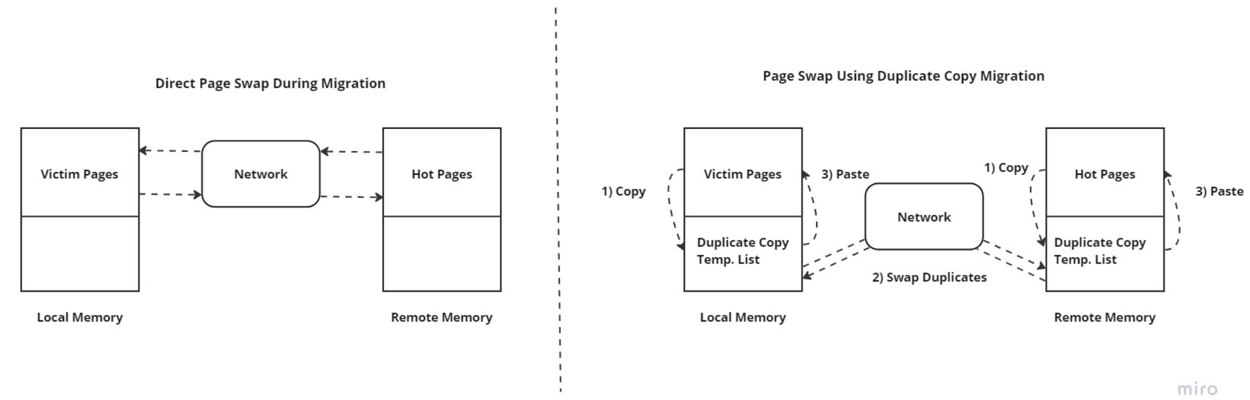

Fig. 4.2 Direct Page Migration vs Duplicate Copy Page Migration

still complete its memory accesses while the migration or page swap is happening. We solve this issue using a duplicate copy of the pages being swapped, as shown in Fig. 4.2, a migration flag that tells the OS that migration for this page has been started in the background and a list of pointers that stores the real physical addresses of these pages that are about to be swapped before their migration. The second duplicate copy is temporarily used to store the pages that are currently being migrated and stored at a fixed location in both the remote and local memory. These duplicate locations are marked as unusable by the system. The list of pointers actually stores the real physical addresses of a page before it was migrated and mapped to the physical address of the fixed location at which that page's copy was stored. Instead of directly starting the swapping of pages and immediately removing them from their respective memory locations, pages are set into a process of copying them into one of these free frames but are not yet removed from their old memory addresses. The migration flag is set to 1, which indicates that the CPU does not need to stall and can still access these pages at their respective old location. Instead of swapping the pages from their original locations, we go ahead and initiate the swap of these duplicate copy pages. Once the duplicate copy swap is complete, the migration flag is changed back to 0, which indicates to the CPU that it can stall. Migration is then, in fact, completed by performing a swap of pages present in duplicate copy to their correct swap locations both in remote and local memory using the list of pointers. Page transfer in the same memory will take a fraction of the time using memory-to-memory copy operation, specifically when compared to remote-to-local memory transfer using DMA. This way, we save ourselves from expensive migration operations.

4.1.6 Software and Hardware Overheads

The total overhead of the page migration approach we are proposing and the hardware support required for it is minimal. The table maintaining the hotness of each page. Each entry in it requires 64-bits to store the physical address of a page and a 32-bit counter. We maintain this information in a hash table which leads to it having a size of around 6KB per node. This can be done in the background. It also requires free memory space for the max number of pages that you want to migrate at one time. We keep that limited to 500 at one time so we keep 500 pages in local and remote memory free, and that allows us to do duplicate copy and swap and reduce the page migration delay. But the memory is in fact, unusable for any workload to access directly, which is a cost we have to bear.

Chapter 5

Experiment Environment

When it comes to disaggregated memory systems evaluation in itself is quite a challenging task as there is no standard simulation infrastructure available that can simulate multiple compute nodes and remote memory pools. We use a cycle-approximate disaggregated memory simulator that models a similar environment with many remote memory pools that is achieved by simulating a CMMU. In order to be able to correctly simulate the notion of time for remote memory accesses by multiple nodes all the remote memory pools, compute nodes, and other system components are synchronized by a single clock. Firstly we make use of Intel's PIN[25] platform and perform binary instrumentation that's used for workload analysis at instruction level granularity. Pintool generates per-thread traces for each executed-instruction that then get consumed on the fly for out of order simulation which uses multi-level cache and TLB hierarchies, as shown in table (5.1). we additionally also model a local memory management unit for each which uses 4-level page tables and manages the local memory address space for page allocation. Rest of the system delays for page faults, page-table walks, etc. are added accordingly for each request.

A CMMU is also simulated that allocates remote memory to the compute nodes in chunks of 4MB each on a request by the nodes's OS page fault handler. Once the remote memory is allocated to the compute node the compute node is allowed to freely allocate pages to that memory. An, interconnection network is simulated by following the network standards of a Gen-Z-enabled network [15],[21]. Finally we make use of a cycle-accurate DRAMSim2 [34] which is used for memory simulation and initialize multiple instances of DRAM, one for each local memory at nodes and for remote memory at remote pools. We modified DRAM-Sim2 to keep track of all the completed memory transactions at different memory units. The remote memory accesses from compute nodes are sent/forwarded to the interconnection network in the form of a network packet, which eventually reaches the pool, simulates the memory access, and gets sent as a response packet to the source. To simulate the multi-node environment, we instrument multiple workloads in parallel that simultaneously produces an instruction trace, which is used for simulation at one of the compute nodes. The simulation is managed using different multiple threads, one for each node, and the barriers are used to synchronize them with a single global clock.

Element	**Parameter**
CPU	1.2GHz, 8-core, 2-Issues/cycle
Core	50-InsQ, 64-RS, 192-ROB, 128-LSQ
ITLB	128-Ent., 8-Way, 10-Cyc
DTLB	64-Ent., 4-Way, 10-Cyc
STLB	1536-Ent, 12-Way, 20-Cyc, 60-Cyc Miss
L1 Cache	32KB(I/D), 8-Way, 4-Cyc, 64B block
L2 Cache	256KB, 4-Way, 12-Cyc, 64B block
L3 Cache	2MB per core shared, 16-Way, 25-Cyc, 64B block
Cache Type	Write-Back/Write-Allocate, Round-Robin
Compute Nodes	8-Nodes, 256MB Local Memory Per Node
Memory Pools	2-Pools, 32GB Per Remote Pool
Memory Type	DDR4 1200MHz x 2
Switch	400Gbps, 132MB Buffer, 20ns delay
NIC	100Gbps, 10ns Delay
Packet Size	64B-Request, 128B-Response/WriteBacks

Table 5.1 Simulation Parameters

Benchmark	**Description**
SimpleMOC(s)[10]	Light Water Reactor Simulation
miniFE[13]	Unstructured Implicit Finite Element Codes
Lulesh [17]	Unstructured Hydrodynamics
XSBench(1)[36]	Monte Carlo Neutron Transport Kernel
NAS:IS(C)[8]	Integer Sort Over Random Memory Access
NAS:MG(C)IS(C)[8]	Long and Short Distance Communication
NAS:SP(C)[8]	Scaler Penta Diagonal Solver

Table 5.2 Benchmarks

Chapter 6

Evaluation

In this section, we start by initially displaying the impact of page migration on system performance in terms of IPC (instructions per cycle). Then next we discuss the improvement in average memory access latency, and system throughput as a result of being able to perform more accesses from local memory only due to the addition of page migration in your system. As every page migration impacts system performance to some extent due to the extra overhead that means just performing more number of accesses from local memory in itself doesn't indicate true system behavior it also depends on how many page migrations had to occur in order to be able to achieve those high number of memory accesses from local memory, Thus we then observe the number of page migrations occur over the entire workload's execution to understand how it impacts system performance. Komareddy et al. performed an exhaustive search in their work using page access count table (PACT) [19] using a wide range of page migration parameters to suggest the best one for the same HPC mini-applications. We take it's two best performing candidates which we will use for comparison with parameter values quite far from each other. The first one is PACT_min which has *page_migration_threshold* as 10, *max_pages_to_migrate* as 10 and a compulsory epoch *migration_epoch_length* of 100K cycles. The second candidate PACT_max uses 10, 500, and 10M cycles for *page_migration_threshold*, *max_pages_to_migrate*, and *migration_epoch_length* respectively. Further we also implement work done by Scrbak et. al work regarding On-The-Fly page migration which has a static *page_migration_threshold* set to 32 as it was on average the best performing static threshold. Further, to observe the benefits of page migration, we use a local-remote alternate page allocation [18] for each consecutive/subsequent page fault in our system. The allocation makes remote and local memory usage at 50% each until the point there is no space left in local memory and the victim page selector has to swap out a page that it finds not recently used. For the benchmarks that have a large memory footprint, the victim page selector plays a big role to evict only the pages that are cold. Finally we perform simulations for a long time (1 billion instructions) so that we can realize the changing of memory access pattern and the adjustment of the page migration parameters by our proposed policy.

6.1 Impact of Page Migration on IPC

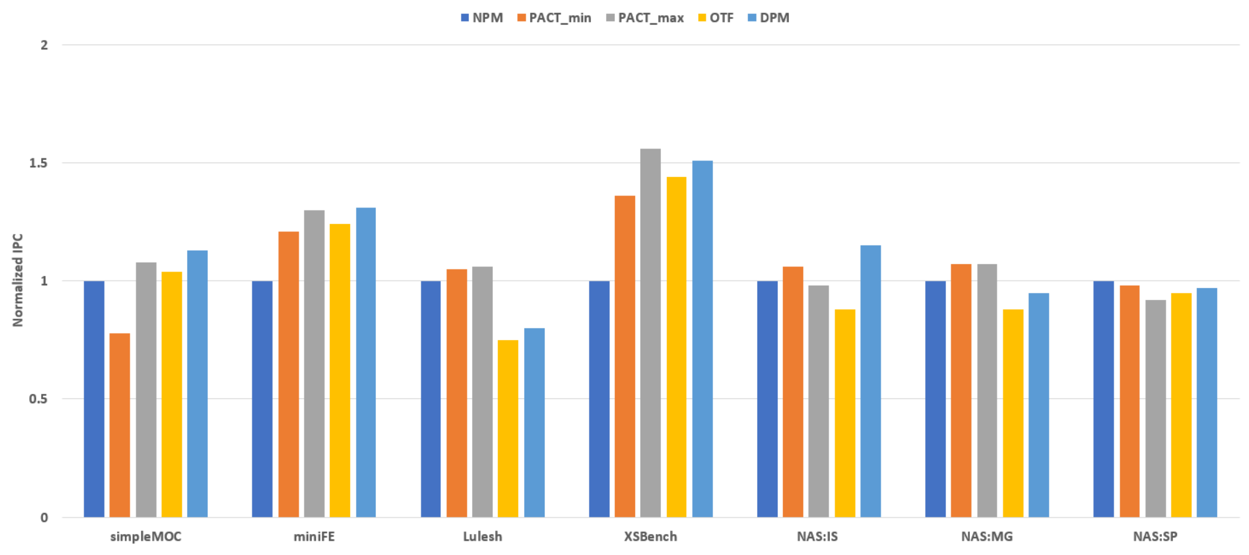

Fig. 6.1 System Speedup with PACT_min, PACT_max, OTF, and DPM compared to no page migration (Normalized IPC)

IPC (instructions per cycle) is one of the aspects of a system's performance its the average number of instructions executed per clock cycle. We see the impact of different hot page migration approaches on system performance as compared to the *no_page_migration* (NPM) in Fig. 6.1. As it is clearly apparent, migrating pages does not always lead to performance improvement due to the overheads involved with page migration as it can be seen with *PACT_min* in simpleMOC, OTF in Lulesh, NAS:IS, NAS:MG, and NAS:SP and OTF in NAS:IS, NAS:MG and NAS:SP etc reason for that being the page access patterns for NAS workloads is quite uniform as can be seen in Fig. 2.3 and do not let us exploit the usual locality of reference that we need in order to make page migration beneficial to our cause. In majority of the cases DPM is able to get better performance when compared to other page migration approaches except in XSBench, Lulesh and NAS:MG where *PACT_max* performs slightly better. NAS benchmarks are known for their uniform memory access patterns meaning it is quite migration unfriendly as can be seen in Fig. 2.3 even despite that fact our approach worked in one of the cases that is cause HPMT and CPMT adjust based on *migration_profit* which helped us in changing the migration parameters in such a way that it was able to adapt to this workload's memory access pattern whenever it could exploit it could find some pages that were being accessed more than the others.

6.2 Percentage of Local Memory Accesses Due to Page Migration

When hot pages are migrated from remote to local memory, it allows more memory accesses to be completed from local memory. As there is a overhead that is attached to each page migration more local memory accesses does not directly translate to better IPC, it can reduce average memory access latency hence reducing the network traffic at rack-level due to less remote memory requests. Fig 6.2 shows the percentage of local memory accesses completed by the workload throughout it's execution. NPM gets exactly 50% of the memory requests

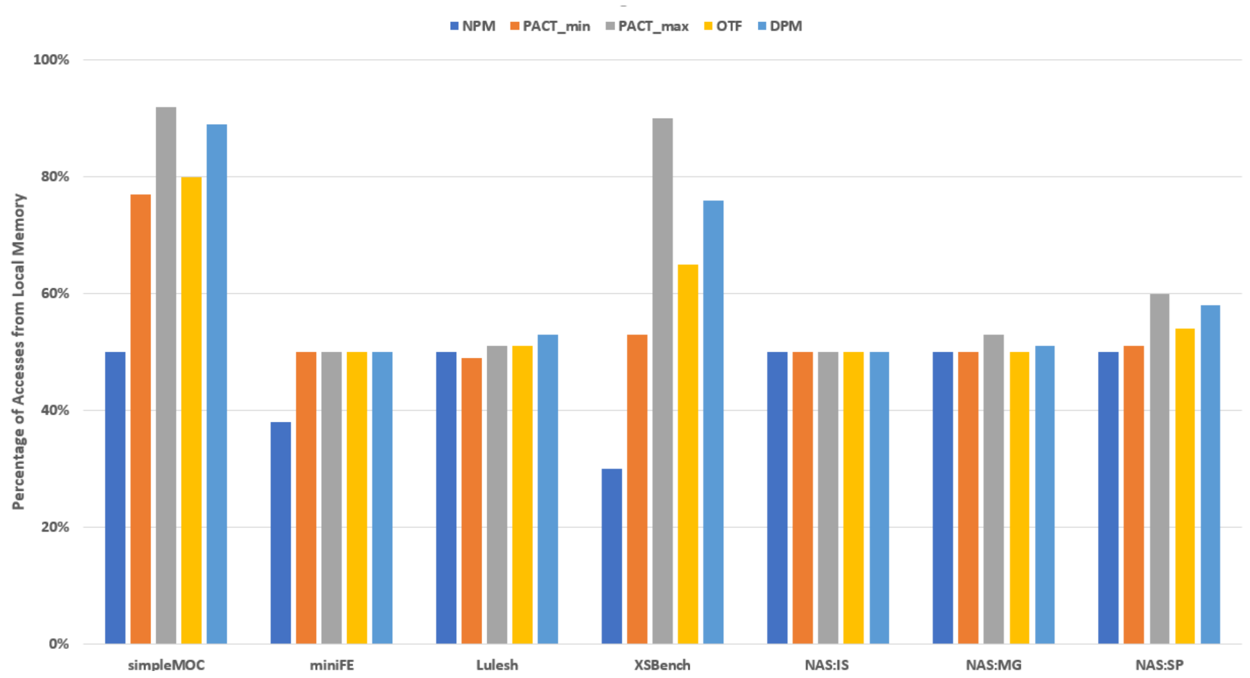

Fig. 6.2 Change in Percentage of Local Memory Accesses due to Page Migration

from local memory and rest of the 50% happen to be from remote memory with SimpleMOC, NAS, and Lulesh, as the memory footprint is not that large for these workloads. We can surely notice the positive impact of DPM in terms of choosing the correct hot pages to migrate to local memory so more accesses can occur from local memory it works especially well with NAS:SP, NAS:IS, simpleMOC and miniFE where it performs nearly the same as $PACT\_max$, For other three workloads, the impact of DPM is almost similar to other approaches except in case of XSBench it does degrade the performance in that case when compared to other approaches but it is still better than OTF in that case as well due to OTF only allowing static PMT. There is only a slight increase in local memory accesses when you look at NAS workloads and Lulesh when compared to NPM, with all migration policies. With Lulesh, the difference is less visible cause of the fact that huge number of memory accesses occur, and migrated pages get only a smaller number of accesses at local memory, which is also the reason behind fewer benefits in terms of IPC. When it comes to NAS(IS), even though the percentage of accesses at local and remote memory are exactly split in the middle the IPC improvement we get is due to slight increase in throughput which is discussed later in this chapter in section 6.4.

6.3 Impact on Average Memory Access Latency

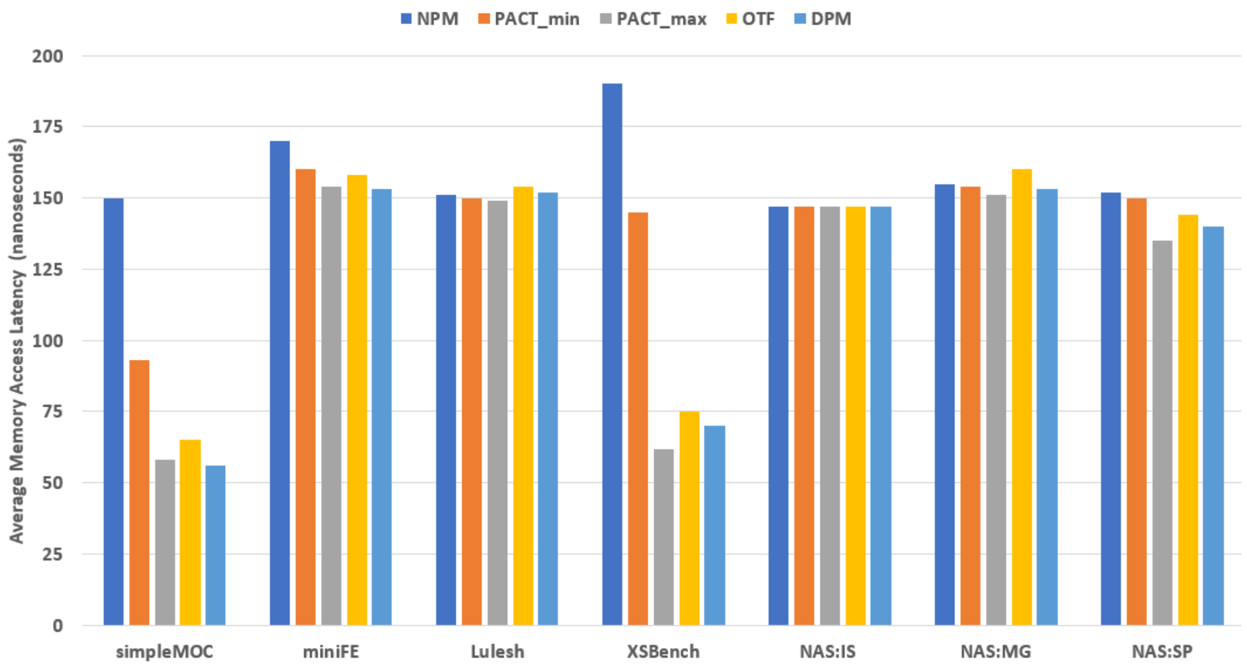

Fig. 6.3 Average Memory Access Latency

Now we measure average memory access latency over local and remote memory. Fig. 6.3 shows the decrement in average memory access latency by using each page migration approach when compared to NPM. Which is due to the elimination of most of the requests being addressed by remote memory. The average memory latency has been reduced significantly for most of the benchmarks, which indicates that page migration is a successful system optimization to reduce the rack-level network traffic. Lower traffic in turn also improves the access speed for the remaining remote accesses as their will be less contention at remote memory pools. As seen, DPM is able to bring down the average memory latency to some extent when compared to NPM,$PACT\_min$, $PACT\_max$, and OTF. DPM almost always performs better than OTF which shows that having a static PMT is actually degrading as it causes higher average memory access latency hence implying our approach of dynamically adjusting page migration parameters is actually benefiting us when it comes to average memory access latency.

6.4 Memory Throughput

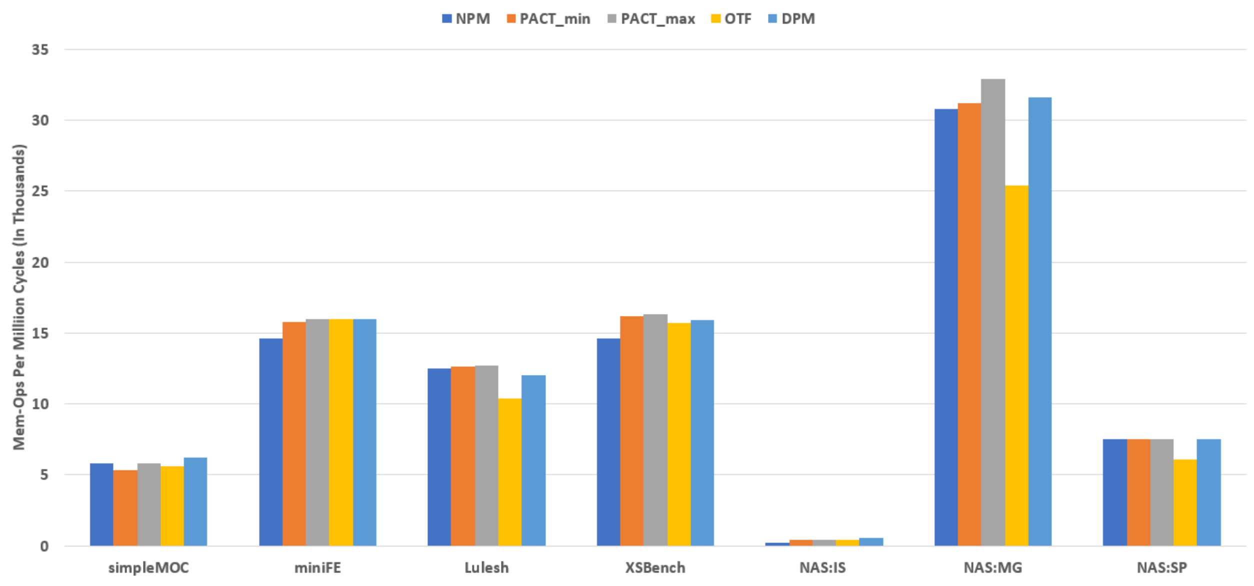

Fig. 6.4 Throughput in terms of memory operations completed per million cycles

Some workloads like NAS have memory accesses pattern in such a way that the ratio of local and remote memory accesses stay the same even after page migration and hence the average memory access latency does not change much. Analysis of memory throughput for such workloads can shine a light on the reasons behind their IPC improvements due to hot page migrations. We measure the throughput in terms of memory operations (mem-ops) per million cycles, shown in Fig. 6.4. Here, we can see that throughput has increased somewhat in simpleMOC, which in fact is also the reason behind it's IPC improvement. Although the number of local memory accesses is less in the case of simpleMOC when compared to $PACT\_max$, they are still enough to provide some IPC improvement however this behaviour as it can be seen in simpleMOC. Similarly NAS:IS while having similar local memory accesses and average memory access latency as other page migration policies an improvement in IPC can be seen cause of the increased throughput. It has to be also noticed that DPM almost always performs better than OTF so it is clearly evident that dynamically changing parameters during workload's execution is working well when compared to a static approach.

6.5 Total Number of Pages Migrated

The total number of pages migrated will be directly responsible for the amount of extra overhead in terms of re-accessing the invalidated enteries from cache and TLB. If we assume that a page is migrated from remote to local memory hoping that it would lead to more accesses from local memory. Upon migration since the physical address of this page has changed the virtual to physical address translation of this page must be invalidated in TLB and the same is done for cached blocks belonging to that particular page. On re-accessing that page you just brought into the local memory, the overhead will be added in the form of TLB miss time for the invalidated TLB entry and cache miss time for each and every cache block belonging to that page. Further, migrating more pages to local memory does not always boost memory performance cause the more pages you bring in from local memory the more pages you have to evict as well from local memory in order to be able to provide space for the incoming pages. Those pages in turn can also get some memory accesses once

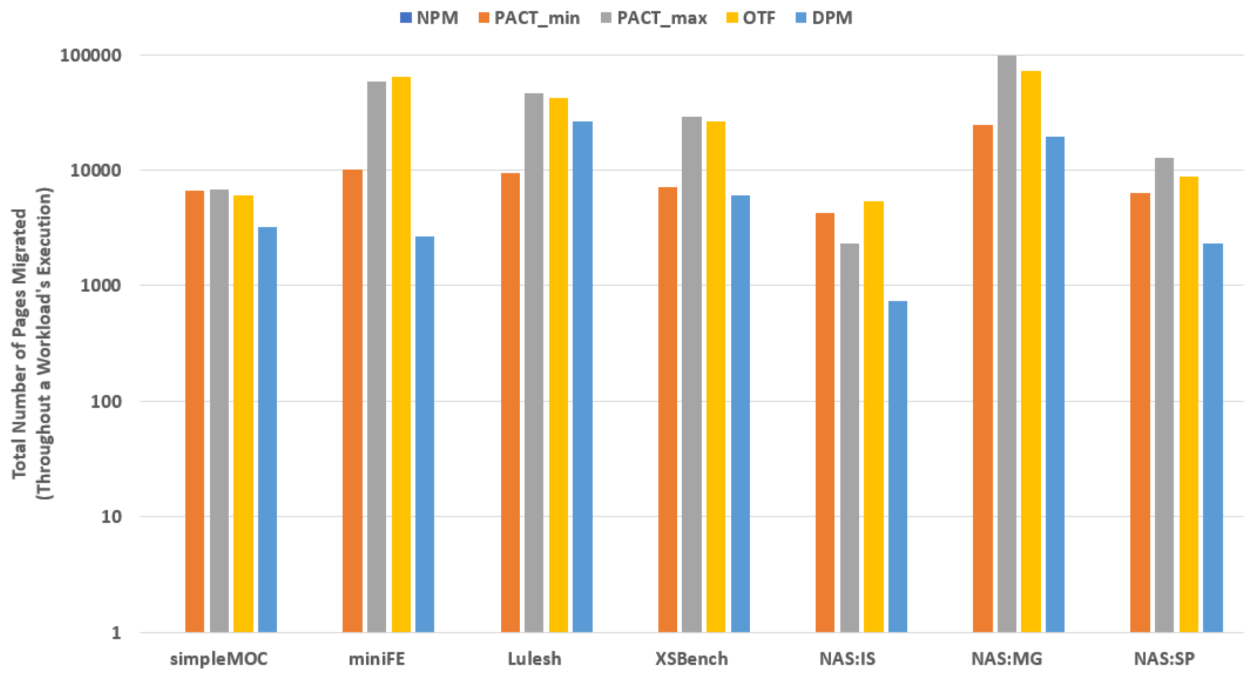

Fig. 6.5 Total Number of Pages Migrated Throughout Workload's Execution

swapped to remote memory, which reduces the benefits of page migration. We analyze the total number of pages migrated throughout each workload's execution with all the page migration policies shown in Fig.6.5. As expected because of our three conditions that need to be fulfilled in order for any page migration to occur as well as using the same measure to measure hotness of each and every page in remote and local memory it can be seen that the total number of pages migrations is lesser than other page migrations except for the case Lulesh where we are beaten by $PACT\_min$ by a small margin. Also it OTF which doesn't use HPMT and CPMT and only uses HPMT is consistantly performing more page migrations when compared to DPM while still not producing better results than our policy. NPM since no page migration is used obviously causes zero page migrations while the overhead of page migrations is not there in that case but exploiting most accessed pages by bringing them to local memory cannot be done there as well. So, Even though page migrations cause page migration overhead it overall still improves the system performance in most cases.

6.6 Evaluation of Migration Overhead

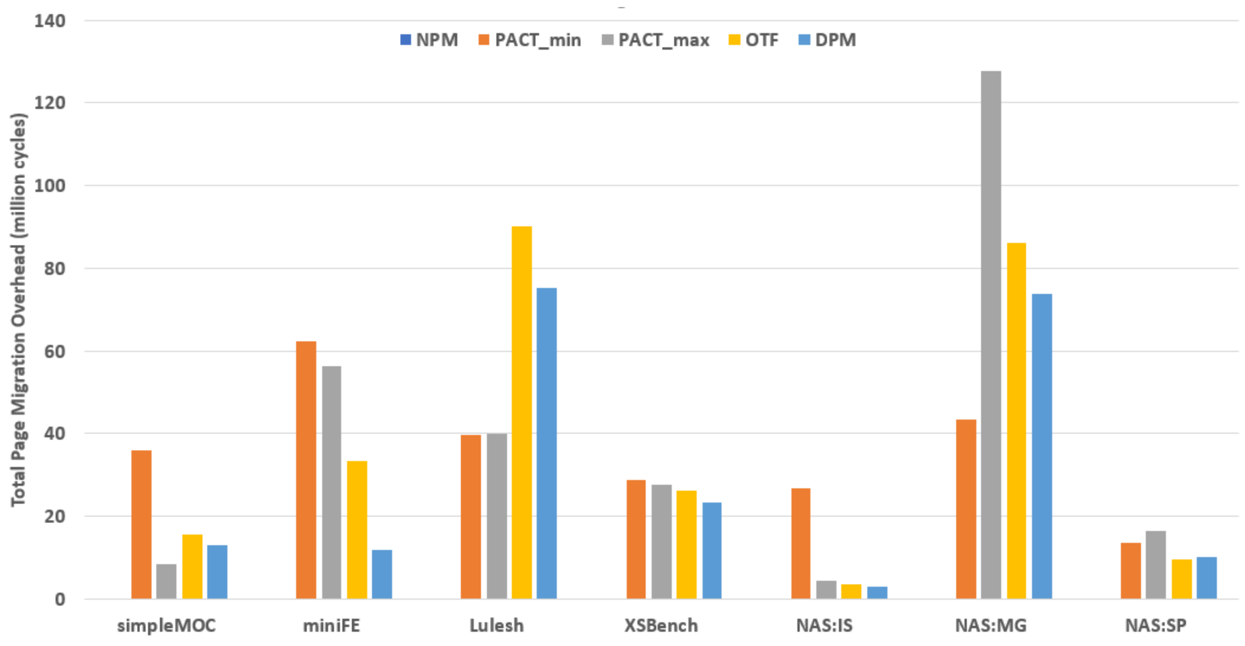

Fig. 6.6 Migration Overheads

To get a clear picture of the true benefit of using page migration we evaluate the total overhead of all the page migrations. The overhead in combination with benefits observed from memory accesses stats presents a precise comparison for page migration. We utilize $P_{count} * (access_{latency\_remote} - access_{latency\_local})$ as the benefit we get from each page migration. And for overhead it is equal to the sum of invalidation time, TLB re-access time, cache re-access time, time to access victim page that have been moved to remote memory, and memory to memory copy time for migrated pages. We calculate the overheads of all the page migartions perfromed throughout the simulation which is shown in Fig.6.6 as it is clearly visible since DPM leads to lower number of migrations in most workloads the absolute overhead due to migration is less compared to other page migration approahes only exceptions being simpleMOC and Lulesh.

Chapter 7

Conclusion and Future Works

Recently disaggregated memory system is a memory design for large-scale systems which can deal with under-utilization issues in HPC systems that deal with large datasets and is easily upgradable but it increases the average memory access time due to the presence of network interconnect between CPU and memory. However we saw that using some system-level optimizations it is possible to be able to bring down average memory access latency for at least for some of HPC workloads that have a non-uniform memory accesses. In this work, we proposed a On-The-Fly page migration policy in a disaggregated memory system. We initially discussed the hardware and software support that is needed for performing page migration in such a system. Our design made use of a Gen-Z interconnection network and a CMMU with a remote memory controller, for which there is quite a small overhead involved. We then discussed our page migration approach that can choose the hottest of pages to migrate and can dynamically adjust depending upon the profit we get from each migration. Further we study about decreasing the page migration latency using system overheads in detail and reduce the extra delays in the form of page transfer latency by using a duplicate copy migration. Also our victim pages selector makes optimal choices while selecting victim pages from local memory during page migration. We also added a couple of extra conditions using HPMT and CPMT to further reduce unneccesary migrations. All these add upto the performance of our proposed page migration, Which improves the system performance in some cases but also does not negatively impact the performance in migration unfriendly cases by much due to dynamically adjusting parameters. As for future prospects machine learning may be something that can be looked into if some light weight machine learning algorithm could predict memory accesses behaviour based on past memory accesses that might be able to optimize system performance even further.

References

[1] Hmcc. 2013. hmc specification 1.0. http://www.hybridmemorycube.org.

[2] Jim handy. 2015. understanding the intel/micron 3d xpoint memory. in proc. sdc.

[3] William Allcock, Bennett Bernardoni, Colleen Bertoni, Neil Getty, Joseph Insley, Michael E Papka, Silvio Rizzi, and Brian Toonen. Ram as a network managed resource. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 99–106. IEEE, 2018.

[4] Razvan Cheveresan, Matt Ramsay, Chris Feucht, and Ilya Sharapov. Characteristics of workloads used in high performance and technical computing. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 73–82, 2007.

[5] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram. In *Proceedings of the International Symposium on Memory Systems*, pages 268–280, 2017.

[6] Chiachen Chou, Aamer Jaleel, and Moinuddin K Qureshi. Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches. *ACM SIGARCH Computer Architecture News*, 43(3S):198–210, 2015.

[7] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *SC'10:*

Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–11. IEEE, 2010.

[8] Dalvan Griebler, Junior Loff, Gabriele Mencagli, Marco Danelutto, and Luiz Gustavo Fernandes. Efficient nas benchmark kernels with c++ parallel programming. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 733–740. IEEE, 2018.

[9] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[10] Geoffrey Gunow, John Tramm, Benoit Forget, Kord Smith, and Tim He. Simplemoc-a performance abstraction for 3d moc. 2015.

[11] Ahmad Hassan, Hans Vandierendonck, and Dimitrios S Nikolopoulos. Energy-efficient hybrid dram/nvm main memory. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 492–493. IEEE, 2015.

[12] Taekyung Heo, Yang Wang, Wei Cui, Jaehyuk Huh, and Lintao Zhang. Adaptive page migration policy with huge pages in tiered memory systems. *IEEE Transactions on Computers*, 71(1):53–68, 2020.

[13] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.

[14] Seokbin Hong, Won-Ok Kwon, and Myeong-Hoon Oh. Hardware implementation and analysis of gen-z protocol for memory-centric architecture. *IEEE Access*, 8:127244–127253, 2020.

[15] Seokbin Hong, Won-Ok Kwon, and Myeong-Hoon Oh. Hardware implementation and analysis of gen-z protocol for memory-centric architecture. *IEEE Access*, 8:127244–127253, 2020.

[16] Mahzabeen Islam, Shashank Adavally, Marko Scrbak, and Krishna Kavi. On-the-fly page migration and address reconciliation for heterogeneous memory systems. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 16(1):1–27, 2020.

[17] Ian Karlin, Jeff Keasler, and J Robert Neely. Lulesh 2.0 updates and changes. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013.

[18] Vamsee Reddy Kommareddy, Amro Awad, Clayton Hughes, and Simon David Hammond. Exploring allocation policies in disaggregated non-volatile memories. In *Proceedings of the Workshop on Memory Centric High Performance Computing*, pages 58–66, 2018.

[19] Vamsee Reddy Kommareddy, Simon David Hammond, Clayton Hughes, Ahmad Samih, and Amro Awad. Page migration support for disaggregated non-volatile memories. In *Proceedings of the International Symposium on Memory Systems*, pages 417–427, 2019.

[20] Wonok Kwon, Chanho Park, and Myeonghoon Oh. Gen-z memory pool system architecture. In *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1356–1360. IEEE, 2020.

[21] Wonok Kwon, Chanho Park, and Myeonghoon Oh. Gen-z memory pool system architecture. In *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1356–1360. IEEE, 2020.

[22] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 488–504, 2021.

[23] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news*, 37(3):267–278, 2009.

[24] Hai-Kun Liu, Di Chen, Hai Jin, Xiao-Fei Liao, Binsheng He, Kan Hu, and Yu Zhang. A survey of non-volatile main memory technologies: state-of-the-arts, practices, and future directions. *Journal of Computer Science and Technology*, 36:4–32, 2021.

[25] 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. *Acm sigplan notices*, 40(6):190–200, 2005.

[26] David Mosberger and Stephane Eranian. *IA-64 Linux kernel: design and implementation.* Prentice Hall PTR, 2001.

[27] Na Niu, Fangfa Fu, Bing Yang, Qiang Wang, Xinpeng Li, Fengchang Lai, and Jinxiang Wang. Pfha: A novel page migration algorithm for hybrid memory embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(10):1685–1692, 2021.

[28] Archit Patke, Haoran Qiu, Saurabh Jha, Srikumar Venugopal, Michele Gazzetti, Christian Pinto, Zbigniew Kalbarczyk, and Ravishankar Iyer. Evaluating hardware memory disaggregation under delay and contention. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1221–1227. IEEE, 2022.

[29] Ivy Peng, Roger Pearce, and Maya Gokhale. On the memory underutilization: Exploring disaggregated memory on hpc systems. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 183–190. IEEE, 2020.

[30] Andreas Prodromou, Mitesh Meswani, Nuwan Jayasena, Gabriel Loh, and Dean M Tullsen. Mempod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 433–444. IEEE, 2017.

[31] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*, pages 1–13, 2012.

[32] Thomas Repantis, Christos D Antonopoulos, Vana Kalogeraki, and Theodore S Papatheodorou. Dynamic page migration in software dsm systems. *TSP*, 48:95, 2004.

[33] Bogdan F Romanescu, Alvin R Lebeck, Daniel J Sorin, and Anne Bracy. Unified instruction/translation/data (unitd) coherence: One protocol to rule them all. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[34] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE computer architecture letters*, 10(1):16–19, 2011.

[35] Yujuan Tan, Baiping Wang, Zhichao Yan, Witawas Srisa-an, Xianzhang Chen, and Duo Liu. Apmigration: Improving performance of hybrid memory performance via an adaptive page migration method. *IEEE Transactions on Parallel and Distributed Systems*, 31(2):266–278, 2019.

[36] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[37] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–349. IEEE, 2011.

[38] Shuang Wu, Bei Wang, Ce Yang, Qinming He, and Jianhai Chen. A hot-page aware hybrid-copy migration method. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 220–221. IEEE, 2016.