Support for different service levels through transparent migration of pages in distributed memory systems

Emmanouil Skordalakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Assistant Prof. *Polyvios Pratikakis*

Thesis CoAdvisor: Doctor *Manolis Marazakis*

This work has been performed at and supported by the Foundation of Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Support for different service levels through transparent migration of pages in distributed memory systems

Thesis submitted by
Emmanouil Skordalakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Emmanouil Skordalakis

Committee approvals:

____
Manolis Marazakis
Doctor, Thesis CoAdvisor

____
Polyvios Pratikakis
Assistant Professor, Thesis Supervisor

____
Manolis GH Katevenis
Professor, Committee Member

____
Angelos Bilas
Professor, Committee Member

Departmental approval: ______________________
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, September 2019

Support for different service levels through transparent migration of pages in distributed memory systems

Abstract

With the constant evolution of high performance applications, their memory requirement is rapidly increasing. As a result, the demand for more memory on computer nodes of large clusters running those applications, continuously rises. However, an individual computer node has limits in terms of memory capacity. Typically, by running several processes of different computational and memory requirements on a cluster, creates fluctuating workloads among the computer nodes. Hence, several nodes use most of their memory, while others have unused memory where other Nodes with a heavy memory workload could potentially exploit it.

Consequently, the concept of remote memory management has become the subject for research by many organizations, which have implemented varying techniques for reading and writing data on remote memory. Although using remote memory practically increases the total available memory of a computer node, accessing data remotely, can critically minimize performance due to the data travelling through the network interconnection of the cluster. Furthermore, software APIs that are implemented to give processes access to remote memory, primarily can be complex, and secondly the responsibility for remote memory allocation and fair remote memory sharing among processes, is assigned to processes themselves, which can be quite complicated, especially when many processes are running simultaneously on the same computer node.

In our thesis, we present the Page Migration System(PMS), which monitors main memory usage of the computer nodes on a cluster, and moves infrequently accessed data of a process from the memory of a computer node with heavy memory workload, to the unused memory of a remote computer node of the same cluster node with a lighter memory workload. The key features of the PMS is that it transparently moves LRU pages of processes to remote memory while using a fairness algorithm when choosing processes and the memory pages among that processes running on the same computer node. What's more, remote memory is mapped on the local node, allowing the OS to cache remote data. To be precise, a read and/or write on remote memory happens when we get a cache miss. Cacheability offers better performance when there are less misses, by reducing network transfers. Finally the system is able to return memory pages locally if the overall node memory usage drops, or if the access frequency of those memory pages increases.

We evaluate the PMS using several benchmarks that stress the CPU in terms of memory access. We use benchmarks that perform raw serial access on arrays of around a Gigabyte in size and thus cause cache eviction frequently, essentially moving more data through the network. That way we can measure the performance drop of a process due to memory access in the worst case scenario. We also run cache blocking benchmarks that exploit temporal locality, and we show that we get a better performance that way by reducing operations on remote memory. Finally

we observe the behaviour and performance on real HPC applications using the PMS.

Υποστήριξη διαφορετικών επιπέδων υπηρεσιών μέσω διάφανης μετακίνησης σελίδων σε κατανεμημένα συστήματα μνήμης

Περίληψη

Με την συνεχή εξέλιξη εφαρμογών υψηλής επίδοσης, αυξάνεται ραγδαία και η ανάγκη τους για περισσότερη μνήμη. Ως αποτέλεσμα, γίνεται όλο πιο επιτακτική η απαίτηση για περισσότερη μνήμη στους υπολογιστικούς κόμβους ίδιας συστάδας που εκτελούν τις παραπάνω εφαρμογές. Ωστόσο, ένας μεμονωμένος υπολογιστικός κόμβος έχει όρια ως προς τη χωρητικότητα μνήμης που μπορεί να διαθέτει. Τυπικά, με την εκτέλεση πολλαπλών εφαρμογών οι οποίες διαφέρουν σε υπολογιστικές και χωρικές απαιτήσεις, δημιουργήται μεταβαλλόμενο φορτίο εργασίας μεταξύ των υπολογιστικών κόμβων. Επομένως, κάποιοι από αυτούς τους κόμβους χρησιμοποιούν το μεγαλύτερο ποσοστό μνήμης τους ενώ κάποιοι άλλοι έχουν ανεκμετάλλευτη χωρητικότητα μνήμης η οποία θα μπορούσε δυνητικά να χρησιμοποιηθεί από εκείνους με μεγαλύτερο φορτίο μνήμης.

Συνεπώς η έννοια της διαχείρησης απομακρυσμένης μνήμης, είναι αντικείμενο έρευνας για πολλούς οργανισμούς, οι οποίοι έχουν υλοποιήσει ποικίλες τεχνικές για ανάγνωση και εγγραφή δεδομένων σε απομακρυσμένη μνήμη. Παρόλο που η χρήση απομακρυσμένης μνήμης πρακτικά αυξάνει την συνολικά διαθέσιμη μνήμη ενός υπολογιστικού κόμβου, η πρόσβαση σε απομακρυσμένα δεδομένα μπορεί να μειώσει δραμάτικα την απόδοση μιας εφαρμογής, εξαιτίας της μεταφοράς αυτών των δεδομένων μέσω της ενδοσύνδεσης δικτύου της συστάδας. Επιπλέον, οι διεπαφές εφαρμογών λογισμικού που υλοποιούνται για να δώσουν πρόσβαση σε απομακρυσμένη μνήμη, αρχικά μπορεί να είναι περίπλοκες ως προς τη χρήση τους, και έπειτα η ευθύνη για την δέσμευση απομακρυσμένης μνήμης και την προτεραιότητα σε αυτήν μεταξύ των εφαρμογών, ανάγεται στις ίδιες τις εφαρμογές, παρόλο που έχει αρκετή πολυπλοκότητα, ειδικά όταν ένας μεγάλος αριθμός εφαρμογών εκτελείται ταυτόχρονα στον ίδιο υπολογιστικό κόμβο.

Στη διπλωματική μας εργασία, παρουσιάζουμε το σύστημα μετανάστευσης σελίδων (PMS), το οποίο επιβλέπει την χρήση κύριας μνήμης των υπολογιστικών κόμβων σε μία συστάδα, και μετακινεί δεδομένα αραιής πρόσβασης μιας διεργασίας, από την μνήμη ενός υπολογστικού κόμβου με αυξημένο φορτίο μνήμης, σε ελεύθερη μνήμη ενός απομακρυσμένου υπολογιστικού κόμβου της ίδιας συστάδας με χαμηλό φορτίο μνήμης. Τα βασικά χαρακτηριστικά του PMS είναι η διάφανη μεταφορά LRU σελίδων μιας διεργασίας σε απομακρυσμένη μνήμη ενώ παράλληλα χρησιμοποιείται ένας αλγόριθμος δικαιοσύνης για την επιλογή διεργασιών και σελίδων μνήμης μεταξύ αυτών των διεργασιών που εκτελούνται στον ίδιο υπολογιστικό κόμβο. Επιπλέον, η απομακρυσμένη μνήμη χαρτογραφείται στον τοπικό κόμβο από το λειτουργικό σύστημα και άρα τα απομακρυσμένα δεδομένα μπορούν να βρίσκονται στην κρυφή μνήμη. Για την ακρίβεια, μια ανάγνωση και/ή εγγραφή στην απομακρυσμένη μνήμη συμβαίνει όταν έχουμε αστοχία στην κρυφή μνήμη. Αυτή η δυνατότητα αυξάνει την απόδοση μιας

διεργασίας όταν δεν υπάρχουν πολλές αστοχίες, λόγω της μείωσης μεταφοράς δεδομένων στο δίκτυο. Τέλος, Το σύστημα επιτρέπει την επιστροφή σελίδων μνήμης τοπικά όταν ελευθερωθεί μνήμη στον κόμβο, ή αυξηθεί η συχνότητα πρόσβασης σε αυτές τις σελιδες.

Αξιολογούμε το PMS εκτελώντας εφαρμογές οι οποίες επικεντρώνται στην προσπέλαση μεγάλου όγκου δεδομένων από τη μνήμη. Χρησιμοποιούμε εφαρμογές που κάνουν σειριακή προσπέλαση σε συνεχόμενες θέσεις μνήμης σε πίνακες της τάξης του 1 Gigabyte και έτσι βγάζουμε συχνά δεδομένα από την κρυφη μνήμη με αποτέλεσμα την συνεχή μεταφορά δεδομένων μέσω δικτύου. Με αυτόν τον τρόπο μπορούμε να μετρήσουμε την μείωση απόδοσης της εφαρμογής λόγω της προσπέλασης σε απομακρυσμένη μνήμη στη χειρότερη περίπτωση. Επίσης εκτελούμε εφαρμογές που έχουν αυξημένη χωρική τοπικότητα στην κρυφή μνήμη, και έτσι δείχνουμε ότι το ποσοστό απόδοσης της εφαρμογής αυξάνεται λόγω της μειωμένης πρόσβασης στην απομακρυσμένη μνήμη. Τέλος, παρατηρούμε την συμπεριφορά και απόδοση σε πραγματικές εφαρμογές υψηλής επίδοσης, χρησιμοποιώντας το PMS.

Acknowledgements

I would like to thank my CoAdvisor Dr. Manolis Marazakis for his guidance and support to implement this research project for my Master's degree. I would also like to thank my Supervisor, Prof. Polyvios Pratikakis for giving me insights and valuable advice during this whole period trying to complete the Master. I would like to give a special thanks to my colleagues Pantelis Xirouchakis and Nikolaos Dimou for providing me with information about the hardware design which I implemented my project on, Vasilis Flouris, Polydoros Petrakis and Nikos Kossifidis for their share of Knowledge in the Linux Kernel and the Software utilities that were used to realise this project, and of course everyone in the Computer Architecture and VLSI(CARV) lab for their support this whole time.

Finally, I would like to thank the Foundation for Research and Technology - Hellas(FORTH), Institute of Computer Science (ICS), for the funding of this work, which was provided by the ExaNoDe research project supported by the European Commission under the "Horizon 2020 Framework Programme" with grant Number 671578, and the EUROEXA research project supported by the European Commission under the "Horizon 2020 Framework Programme" with grant Number 754337.

Contents

Table of Contents i

List of Tables iii

List of Figures v

1 Introduction 1

1.1 Prerequisites and Assumptions 2
1.2 Contributions 2

2 Related Work 3

2.1 On demand Remote Memory user-level access 3
2.2 Disaggregated Clusters 3
2.3 Hardware-based remote memory access 4

3 Design 7

3.1 Exploitation of the Hardware design 7
3.2 Kernel Modifications 9
3.3 Page Migration System design 10

4 Implementation 13

4.1 The page-migration-monitor 14
4.1.1 The daemon algorithm 14
4.1.2 The process selection algorithm 16
4.1.3 The daemon process data table 18
4.2 The pmm-sender library 19
4.3 Patching the Linux Kernel 20
4.4 The page-migrator Kernel Module 22

5 Evaluation 25

5.1 Testbed 25
5.2 Benchmark: Lmbench 26
5.2.1 Lmbench Setup 26
5.2.2 Lmbench Results 27

5.3 Benchmark: Stream . . . 28
5.3.1 Stream Setup . . . 29
5.3.2 Stream Results . . . 29
5.4 Benchmark: Himeno . . . 38
5.4.1 Himeno Setup . . . 38
5.4.2 Himeno Results . . . 38
5.5 Benchmark: StencilProbe . . . 39
5.5.1 Stencilprobe Setup . . . 40
5.5.2 Stencilprobe Results . . . 40
5.6 Benchmark: HPL . . . 45
5.6.1 HPL Setup . . . 46
5.6.2 HPL Results . . . 47
5.7 Benchmark: CSparse . . . 49
5.7.1 CSparse Setup . . . 49
5.7.2 CSparse Results . . . 50

6 Conclusion **53**
6.1 Summary . . . 53
6.2 Future Work . . . 54

Bibliography **55**

List of Tables

5.1 Lmbench Memory Read Latency, Array size: 256MB pt1 27
5.2 Lmbench Memory Read Latency, Array size: 256MB pt2 27
5.3 Stream Copy algorithm, Array size 512MB 30
5.4 Stream Copy algorithm, Array size 256MB 30
5.5 Stream Scale algorithm, Array size 512MB 32
5.6 Stream Scale algorithm, Array size 256MB 32
5.7 Stream Add algorithm, Array size 512MB 34
5.8 Stream Add algorithm, Array size 256MB 34
5.9 Stream Triad algorithm, Array size 512MB 36
5.10 Stream Triad algorithm, Array size 256MB 36

List of Figures

3.1 Data transfer paths for local writes (lw) and remote writes (rw). . 9
3.2 Page Migration System Implementation Overview. 11

4.1 Flowchart of event processing by the Page Migration Monitor. . . . 16
4.2 Flowchart of the process selection algorithm. 17
4.3 Overview of the process data table. 18
4.4 the inter-process communication between the daemon and processes using the library through a FIFO. 20
4.5 Flowchart of the page-migrator Kernel Module. 23

5.1 Testbed Overview. 26
5.2 Lmbench: Memory Read Latency for local and remote configuration 28
5.3 Stream Copy algorithm. Local configuration vs one array remote configurations, migrating used memory for 1 to 4 threads. 31
5.4 Stream Copy algorithm. Local configuration vs two arrays remote configurations, migrating used memory for 1 to 4 threads. 31
5.5 Stream Scale algorithm. Local configuration vs one array remote configurations, migrating used memory for 1 to 4 threads. 33
5.6 Stream Scale algorithm. Local configuration vs two arrays remote configurations, migrating used memory for 1 to 4 threads. 33
5.7 Stream Add algorithm. Local configuration vs one array remote configurations, migrating used memory for 1 to 4 threads. 35
5.8 Stream Add algorithm. Local configuration vs two arrays remote configurations, migrating used memory for 1 to 4 threads. 35
5.9 Stream Triad algorithm. Local configuration vs one array remote configurations, migrating used memory for 1 to 4 threads. 37
5.10 Stream Triad algorithm. Local configuration vs two arrays remote configurations, migrating used memory for 1 to 4 threads. 37
5.11 Himeno Benchmark. MFLOPS for local and remote configurations for 1, 2, 3 and 4 threads. 39
5.12 Stencilprobe blocked algorithm with worst cache block. Slowdown of remote configurations compared to local configuration. 41
5.13 Stencilprobe blocked algorithm with optimal cache block. Slowdown of remote configurations compared to local configuration. 41

5.14 Stencilprobe time skewing algorithm with worst cache block. Slowdown of remote configurations compared to local configuration. . . 42
5.15 Stencilprobe time skewing algorithm with optimal cache block. Slowdown of remote configurations compared to local configuration. . . 43
5.16 Stencilprobe circular queue algorithm with worst cache block. Slowdown of remote configurations compared to local configuration. . . 44
5.17 Stencilprobe circular queue algorithm with optimal cache block. Slowdown of remote configurations compared to local configuration. 44
5.18 Stencilprobe cache oblivious algorithm with worst cache block. Slowdown of remote configurations compared to local configuration. . . 45
5.19 Hpl data split between local and remote memory on the 4 configurations. 47
5.20 HPL benchmark n=11585, PxQ=1x4, 4 processes run. Slowdown of remote configurations compared to local configuration. 48
5.21 ct20stif and nd3k matrices composition. 50
5.22 CSparse demo 2 on ct20stif matrix. Slowdown of remote configuration compared to local configuration. 51
5.23 CSparse demo 2 on nd3k matrix. Slowdown of remote configuration compared to local configuration. 52

Chapter 1

Introduction

High Performance Computing platforms are constantly evolving in an effort to satisfy the demand of contemporary scientific applications. Contemporary scientific applications have among others, immense requirements in terms of computational power and memory capacity. Nowadays, HPC infrastructure of large-scale systems is consisted of hundreds or even thousands of compute nodes which communicate using a fast interconnect. Multiple heterogeneous applications often run in parallel on those infrastructures, generating fluctuated memory workloads in the compute nodes of a system. This necessitates the ability to exploit the unused memory of nodes in favour of nodes that have exhausted their memory.

The concept of allocating remote memory is well known to the HPC world, and many projects have been developed to realize that concept. Nonetheless, there are several drawbacks that emerge from using remote memory. First, when reading or writing data at remote memory, there is an increasing latency due to the data travelling through the network. Second, most projects that are implemented for remote memory allocation consist of complex APIs, that a process has to use, and define what memory should be allocated remotely. Additionally, many applications do not perform computations on all of its data during the whole duration of their lifetime. This means that as long as data are intensively used, it is best for them to reside in local memory for faster read and write access, however if those data are idle, they essentially occupy memory that could be used by other sources.

In this Master thesis the author has implemented and describes the Page Migration System (abbr. PMS) that essentially increases a node memory capacity by mapping unused remote memory of another node locally. The PMS is adapts to the local memory workload of the node and when the workload is heavy, it will move data of processes to mapped remote memory. On the other hand, in the case of light memory workload and if there are migrated data to mapped remote memory, the PMS can return those data locally for better process performance. The procedure of page migration is transparent for processes, while the PMS uses a priority policy among those processes when choosing one to migrate remotely or return locally its pages. When choosing pages among a process, the PMS uses the

global LRU page policy to only migrate LRU pages to remote memory. A user process informs the PMS of its allocated pages and can also suggest which pages are critical and thus should not be migrated. After the migration procedure has completed, a process retains the original virtual address of its pages and data that now reside in remote memory, are still cacheable locally.

1.1 Prerequisites and Assumptions

The PMS utilises several software and hardware resources. To realise the implementation of the system, we assume an existing hardware distributed system that consists of several compute nodes. Every node in that system contains its independent operating system of Linux distributions, as well as the maximum amount of main memory that one can install on it. Each node has read and write access on the memory of the rest of the nodes in this distributed system, by utilising a Partitioned Global Address Space (PGAS) hardware design, already implemented in the system. The PGAS is one of the prerequisites that can potentially enable remote memory to be cacheable to the local compute node.

1.2 Contributions

The author of this Master Thesis has contributed in the development of the PMS in the Following three key aspects. First is the implementation of the software components of the PMS. A daemon process which is the core component of the system, scheduling actions and storing important information, a software library used by process to send relative information to the daemon process, and a Kernel module, responsible for moving pages from local memory to remote memory and vice versa. Secondly, the Kernel of the Operating system was modified accordingly to support mapping of remote memory and allow pages to be moved there. At last, the author verified the functionality and evaluated the performance of the PMS by executing several benchmark applications.

Chapter 2

Related Work

2.1 On demand Remote Memory user-level access

"Remote Regions" [19] presented at Usenix ATC 2018 by VMWARE exports memory to remote hosts as files through a simple interface. A process allocates remote memory on demand, and that memory always resides in the node that was allocated. Then, a remote host reads or writes the allocated memory using an API similar file operations (i.e. read, write, etc.). Because of this, remote regions is not transparent in terms of read/write and memory allocation unlike the Page Migration System. Finally, Remote Regions do not provide cacheable remote memory. Therefore, a process cannot exploit cache blocking techniques for better performance.

"Designing Far Memory Data structures" [20] presented at HotOS 2019 by Microsoft, VMWARE and HP focus on the idea that only the core content of the data structure resides in remote memory, while the path to the core content of the structure can be calculated in the cache of the local computer node. This method reduces the network traffic, since most of the time, only one round trip is required to fetch the required data from remote memory. Operations on those data structures are performed using indirect addressing where a local pointer dereferences to the actual physical address on remote memory, scatter-gather operations, and notification operations which inform a node that data on remote memory have been altered. Far memory data structures include custom made Maps, Queues and Vectors. Unlike the PMS, there is no transparency when accessing remote memory.

2.2 Disaggregated Clusters

"LegoOS" [23] presented at Usenix OSDI 2018 by Purdue University, is an Operating System designed to work on disaggregated computing systems. Unlike a monolithic system where the processor, main memory and storage hardware devices are contained in the same physical machine, in a disaggregated system, there

are several distributed machines where one would contain only a processor, another would contain only the memory and another the storage. LegoOS works on top of those distributed hardware resources and utilizes them all together through a NIC. Users get access to a virtual Node, and LegoOS can distribute hardware resources dynamically to each virtual node according to their needs. Essentially it is a completely different form of cluster designed to improve resource utilization, elasticity, heterogeneity and failure isolation since each hardware component can operate or fail on its own and its resource allocation is independent from other components. On the part of remote memory management, it offers complete transparency, since the remote memory allocation is managed by the hardware and operating system. However it suffers performance-wise compared to a monolithic system, since communication through hardware resources happens using the NIC instead of an internal bus. The system is completely transparent in terms of remote memory management. The disadvantage compared to the page migration system is that the whole memory of a node is essentially remote memory, thus every main memory access experiences higher latency due to the data transmission over the network interface

"dReDBox" [14] presented at IEEE 2016 by IBM, UOT, FORTH, AUEB, HPN, VOS and BSC, is a disaggregated cluster like LegoOS, however it also combines the capabilities of monolithic compute nodes. The basic idea is that the main components of the cluster are monolithic compute nodes where each node contains its individual Processor, Memory and Storage, however the cluster also includes physical machines that purely contain physical memory or storage. A User has access to each node through a virtual machine, and drReDBox is capable of supplying each node with additional memory or storage from those physical machines. Similarly to the PMS, a node has access to its own local memory and remote memory, where in the dReDBox remote memory given dynamically by the hypervisor of the disaggregated cluster. However dReDBox does not utilize any policy that selectively distributes data of processes to remote memory based on a priority policy, hence there is no way to ensure that a critical process will access its data on local memory.

2.3 Hardware-based remote memory access

Project PBerry [13] presented at HotOS'19 by VMWARE and ETH Zürich exploits the Virtual Memory subsystem to provide access to remote memory. A computer node contains an FPGA programmed to directly access remote memory. In particular there is a coherent link between the last level cache of the CPU and the FPGA. When the CPU generates a last level cache miss, and the requested data do not reside in local memory, the CPU will request the data from the FPGA. The FPGA will transfer a whole page where the cache line is part of, from remote memory to local memory and if there is no available local memory, the FPGA will have to transfer a local page to remote memory first. However, the FPGA will only

fetch the cache line at first, and will asynchronously transfer the rest of the page to local memory in order to avoid page faults and improve latency. Like the PMS, there is transparency in data transfers, however, a big disadvantage compared to the PMS is, that there is no priority protocol among many processes that run on the same computer node, meaning that the FPGA can evict pages of critical processes that run on the system to remote memory, and delay their completion time.

KRAM is a remote swap space, implemented in the CARV laboratory of FORTH. The KRAM uses the (unused) memory of some remote node in order to create a ram disk. Each node consists of a kernel-space module that maps remote memory of another node as an I/O device in order to accomplish the task. For memory transfer to/from remote memory, an RDMA engine and/or memcpy are used. In contrast to PM System, KRAM uses a different interface, which is the Linux swapper, to swap out memory pages to a remote node, when RAM usage is at its limit. Similarly to PM System, KRAM uses a transparent method to exploit remote memory of a compute node. Although KRAM uses an LRU policy to swap out pages, processes are not able to demand their data to remain in the local memory. More specifically, pages that are frequently accessed should be swapped in the local memory, while pages that are infrequently accessed should swapped out. In the PMS, the kernel of the local node is able to directly map remote memory, which allows remote read and write operations. Although The KRAM is fully transparent when accessing remote memory, the core difference with the PMS is that when a process wants to access data in remote memory, a page fault is generated and a whole page must be transferred through the network interface. On the other hand, when using the PMS and a process wants to access remote memory, if there is a cache miss on the last level cache, the system only requires to fetch a cache line from remote memory.

Chapter 3

Design

In principle, we need to know the capabilities that the hardware design of the distributed system offers, and build the page migration system in conjunction with that design. Essentially our system must be able to to migrate memory pages transparently. To accomplish that objective, it must be aware of the memory pages that are allocated by user-space processes running on a computer node. Furthermore, a software mechanism is required to copy data from a part of physical memory to another. Finally, if data are migrated to remote memory, read and write operations take longer to complete, since data have to travel through the network of the distributed system. For this purpose the system should cleverly decide what pages to migrate among several processes. That algorithm of the system must consider the general system memory usage, the rate that a page is read or written, and be fair among processes regarding which pages shall be chosen for migration.

3.1 Exploitation of the Hardware design

Our design is built on top of a distributed system that consists of several computer nodes. Those nodes contain a Zynq UltraScale+ MPSoC with an FPGA programmed to communicate with each other through a network switch. The network interconnection design is developed in the context of the ExaNeSt [10] project. The hardware design supports zero-copy read and write operations to remote memory using a global address space among the computer nodes. If a read or write operation is initiated, and the physical address contains a hex prefix, instead of reading or writing in the local physical memory, the request is sent to the PL of the computer node. The PL uses that prefix to determine the remote node where the remote read/write operation must be executed on the memory of. The system boots from an SD card. That SD card also contains a device tree that informs the OS, where each hardware chip exists on board and how to use it. For the 2GB of physical memory, we instruct the OS through the device tree to map only the 1768MB and label the rest 256MB as reserved memory. Reserved memory is not

mapped by the Kernel, so there are no means that the system can read or write on that memory unless custom read or write system calls are used.

The above implementation of the hardware design allows us to hotplug [17] physical memory of a computer node to another computer node. When memory is hotplugged in a node, the kernel of that node recognizes new memory, makes new memory management tables, and makes sysfs files for new memory's operation. This means that we can exploit the Kernel's software utilities to perform operations on remote pages and we also gain the advantage that the data which reside in remote memory are cacheable. This gives us two significant benefits. First of all, the user will not require to specifically request a remote read or write operation on data. The Kernel will initiate those operations during a cache invalidation. Figure 3.1 depicts the paths a local and remote write will take in the hardware after a cache invalidation. In detail, for a local/remote write to initiate, data must be removed from cache and be written to memory. The physical address of that data will enter the Cache Coherent Interconnect (CCI). The CCI determines if the physical address is local or remote. If the address contains eight extra MS bits, it is remote, else it is local. For a local write, the data and address will be transferred to the local DDR Memory controller and finally to the physical memory. For a remote write, the CCI will send the data and address to Programmable Logic (PL), the PL will send them to the network, and the network switch will send them to the corresponding remote node. The PL of the remote node will send the data and address now converted to local, to its DDR Memory controller to be written to remote physical memory. This fulfills the objective to make remote operations transparent for the user. The second benefit is that we can use cache blocking techniques to perform read/write operations on remote data, in cache. This equals to less network traffic, which will minimise performance loss of a process using remote memory.

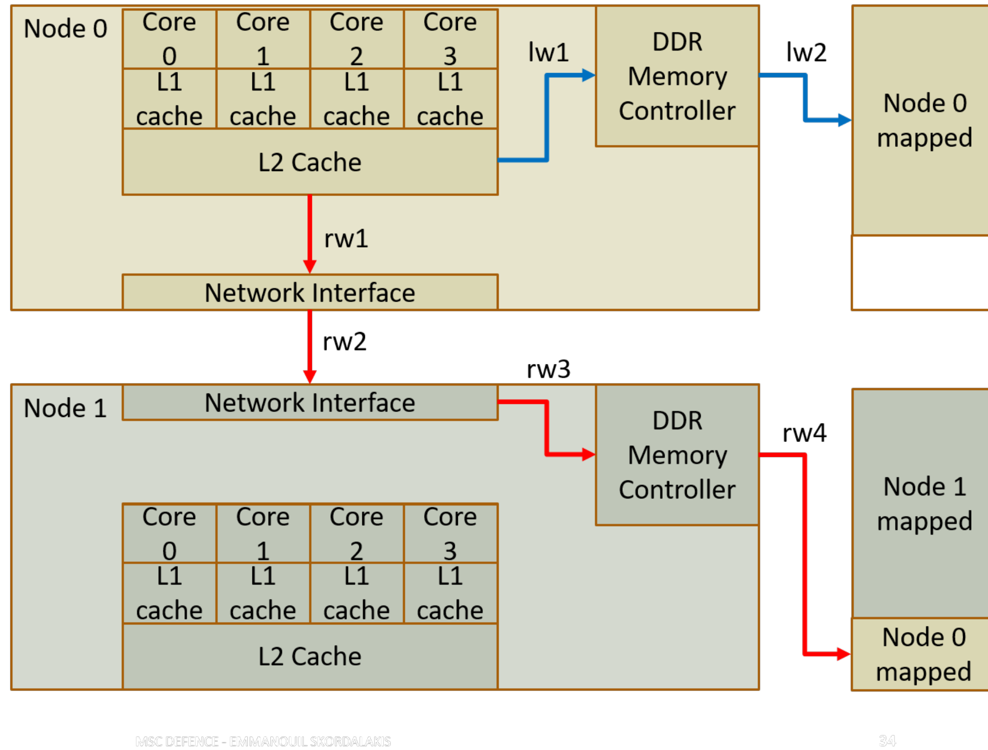

Figure 3.1: Data transfer paths for local writes (lw) and remote writes (rw).

3.2 Kernel Modifications

As we mentioned in the previous section, we can hot-lug remote memory to a computer node. We use the Linux Kernel version 4.9 since it supports memory hotplug through patching [2]. The kernel divides memory into blocks called zones which represent ranges within memory. When a memory allocation mechanism is used, the Kernel looks up free memory space in those zones, and allocates the requested size. We want to create an entry for a new zone, so that when we hotplug remote memory, the Kernel registers it there. We must also isolate that zone from all allocation mechanisms so that the only way a process can have access to remote memory will be when the page migration system migrates its memory pages there.

The process of migrating memory pages requires the execution of several tasks, such as blocking the page's access from other processes while data are copied from one page to the other, reverting the page table entries so that they point to the new page, etc. The Kernel already contains such utilities [16], however their main purpose is to be used for memory compaction, or for handling poisoned pages, and therefore they are not exported to be used by outer modules. We can export and modify several of those Kernel utilities and use them in collaboration, in order to achieve page migration. Since this operation can only be done in Kernel-space, we

create a module to perform the task.

3.3 Page Migration System design

As we discussed in the first paragraph of this chapter, we need to be aware of the pages that are allocated by the processes, and select among those pages for migration. In the /proc path of the root file-system, there are files that contain information of every process that runs on the system along with the virtual and physical addresses of memory that is allocated by them. In the /proc/kpageflags file there are bit-strings which describe the state of memory pages of all processes running in the system. By reading a bit-string of a specific memory page, we can determine if a page belongs in the LRU list, is poisoned, unevictable etc. The /proc path addresses however also point to instruction memory which would not be wise to migrate to remote memory for better performance. For our prototype design, we implement simple inter-process communication between a process and our system where a process can send the system the virtual memory of data that are allocated at run-time using a software library. The system can then match that memory with the process entries in the file-system and find the physical addresses that could be candidates for migration.

The core of the page migration system is a daemon process that stores information on migrated pages and their processes. The daemon works as an event-driven application. Since we implement inter-process communication, the first thing that should be done by the daemon is to read incoming messages from other processes. Those messages can contain allocated memory information and requests of the processes as to how the daemon should handle that memory. Next step for the daemon is to read the current memory workload. The daemon should have two threshold values defined. One threshold when compared to current system memory usage would mean that current memory workload is high enough so that the computer node requires more memory capacity and the other threshold would mean that current memory workload is light for the current node. In case of a high memory workload, the daemon must gather the physical addresses of local pages that are appropriate to migrate to remote memory. In case of a low memory workload, the system can gather the physical addresses of migrated pages and return them locally, since now there is enough unused local memory to satisfy memory requirements for all processes running the system.

Once the physical addresses are gathered, we need a way to move them from local memory to remote and vice versa. That memory is already mapped by the Kernel. As we explained in the previous section. We can export several Kernel utilities that perform page migration for us. To use them, we implement a character device driver that can acquire physical pages on demand by writing them with the daemon process to a character device file created by the driver, and then perform the migration of pages. The driver must fulfill all of the possible safety properties when moving memory pages, such as the case that a process suddenly terminates

or that the physical address requested to migrate, is actually invalid. After the migration procedure is complete, the driver is required to send relevant information to the daemon process, as to how many and which pages successfully migrated, or if the migration procedure failed. Figure 3.2 presents an overview of the Page Migration System implementation.

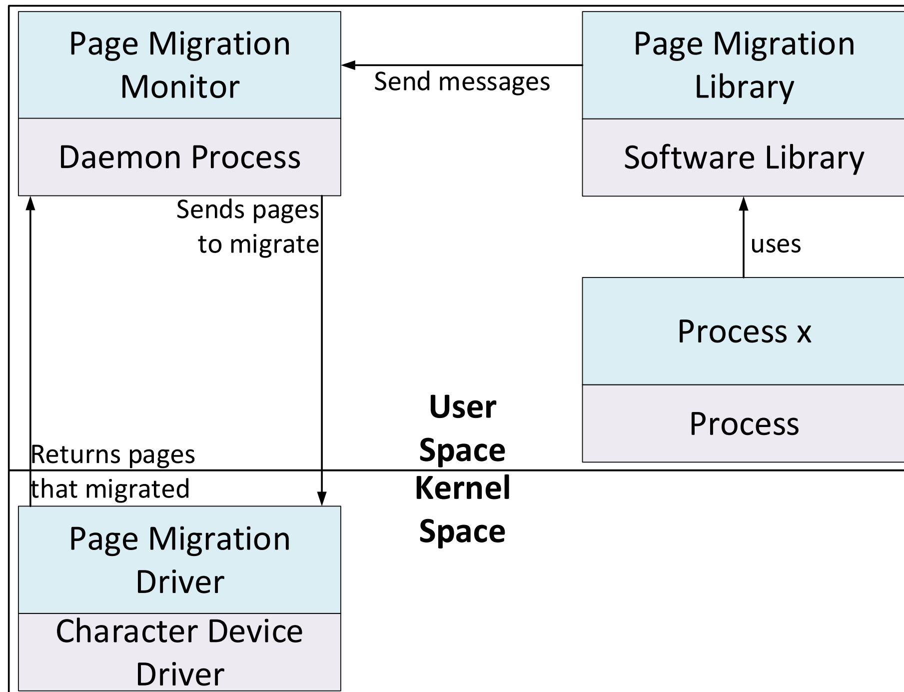

Figure 3.2: Page Migration System Implementation Overview.

Chapter 4

Implementation

The PM system consists of several software components that communicate with each other in order to transparently migrate pages in a distributed system. In our implementation, the system consists of compute nodes that have isolated part of their memory. The isolated memory section is not mapped by the kernel and therefore the standard allocation mechanisms in user and kernel space will not allocate memory from that section. Our system hot-plugs isolated memory from remote nodes, into our local node. The hotplugged remote memory is mapped by the local node and is registered to an isolated memory zone which is specifically defined for the page migration system. Default allocation mechanisms in user and kernel space cannot allocate memory in that zone. The migration system uses that zone to point at hot-plugged remote memory, and move LRU pages there.

More precisely, a daemon process is continuously active on the system, and inspects the main memory usage. When free main memory percentage is dropped below a specified threshold, the daemon iterates over registered processes and chooses LRU memory pages that are allocated by those processes. When it finishes collecting page frame numbers that are candidates for migration, it sends that data to a kernel module through file communication. The kernel module is then responsible for performing the task of migration. When the module completes the process, it sends the frame numbers of pages that were migrated, back to the daemon on success, or an appropriate return value if migration could not take place. On success, the daemon stores information of pages that were moved. The daemon can only move pages of processes that are registered to it. In order for a process to register to the daemon, a user space software library with a simple API must be included, that the process can use to send information to the daemon, such as the process id, allocated memory that can be migrated etc.

4.1 The page-migration-monitor

We now describe the Page Migration Monitor, which is one of the most basic software components of the PM System. The page migration monitor is a user-space daemon that continuously runs on the system performing several sequential steps of operations, and its main role is to monitor system's memory usage. During the initialization of the daemon, two thresholds are specified as parameters. The first threshold is called migration percentage threshold. That is, when free available RAM goes below this threshold, the daemon will try to migrate local pages that are infrequently used to remote nodes. The second threshold is called return percentage threshold, and its value should be greater than the value of the first threshold. Whenever the amount free RAM goes above this threshold, the daemon will try to bring pages that are migrated to remote nodes to the local memory. After determining which pages should be migrated, the Page Migration Monitor will send pages' frame numbers to a kernel module via file communication, so the latter can perform the actual migration of the memory. The daemon also checks a FIFO file at every event, for any incoming messages from processes that use the library to send information. That information can be either process data for registration, or memory with the potential to be migrated.

4.1.1 The daemon algorithm

Figure 4.1 depicts the operations taken by the Page Migration Monitor.

- The daemon reads the FIFO file of an incoming message. A message contains data including a value that determines its type. In case a message has arrived, the daemon checks the message type and performs the appropriate action, depending on the message type.
  - Process registration: A new process requests to be registered to the daemon. This message contains the process id and a priority value that the daemon uses in order to determine which process's pages should be migrated first. A process with a higher priority value will have its pages moved at a later time, than a process with a lower priority value.
  - Process sends memory: A process previously registered, sends contiguous virtual memory that has the potential to be migrated, in case of the system memory reaching the migration threshold. This message contains the process id, a virtual address, and a size that is a number of sequential memory bytes. The virtual address is the starting memory of those bytes.
  - Process release memory request: A process requests to release memory specified. This message contains the process id, and a virtual address that has been sent before by a "Process sends memory" message. The daemon will clear all data from its data structures regarding that memory including information about migrated pages in that memory range.

This message should only be sent by a process right before the process deallocates that memory.

- Process lock memory request: A process requests to keep memory specified, locally. This message contains the process id, and a virtual address that has been sent before by a "Process sends memory" message. The daemon will mark the memory starting with the specified virtual address as locked. If that memory has been moved remotely, the daemon will add the process data struct in a queue, and on the next event will try to bring its memory locally. The process struct will remain in that queue until all pages have returned. After that, the daemon will never migrate pages of that process again.
- Process termination: A process sends a cleanup message and the daemon removes any stored data of the particular process. This message contains the process id. This message should only be sent by a process that terminates immediately after that.

- The daemon serves pending lock memory requests. All processes that requested to lock their memory, will have their data struct stored in a queue. The daemon will return all pages requested by the processes in the queue locally, and will move to the next step when the queue is empty.
- The daemon reads the system memory usage, and determines if memory has reached one of the two thresholds. If free Ram percentage is below the specified migration threshold, and the memory zone has enough free space to receive pages:
  - The daemon will try to find a process, appropriate to migrate its pages. To determine which process will be selected, the daemon takes into account, the process' priority value, and how many pages each process has already sent.
  - After an appropriate process has been selected, the daemon will scan the memory pages sent by that process, and use the /proc entries in the file-system to determine which of those pages are in the LRU list. LRU pages will be stored in a buffer.
  - The daemon will then send that buffer to the page-migration module, to migrate those pages to the memory zone.
  - On completion, the daemon receives a modified buffer from the module that contains only the pages that were successfully migrated and the daemon will update the process data struct accordingly.
- If free Ram percentage is above the specified return threshold, and there are pages that are already migrated to the memory zone:
  - the daemon will try to find a process, appropriate to return its pages. To determine which process will be selected, the daemon takes into

account, the process' priority value, and how many pages each process has already sent.

- After an appropriate process has been selected, the daemon will add an amount of migrated page frame numbers of that process, in a buffer.
- The daemon will then send that buffer to the page-migration module, to return those pages from the memory zone.
- On completion, the daemon receives a modified buffer from the module that contains only the pages that were successfully returned and the daemon will update the process data struct accordingly.

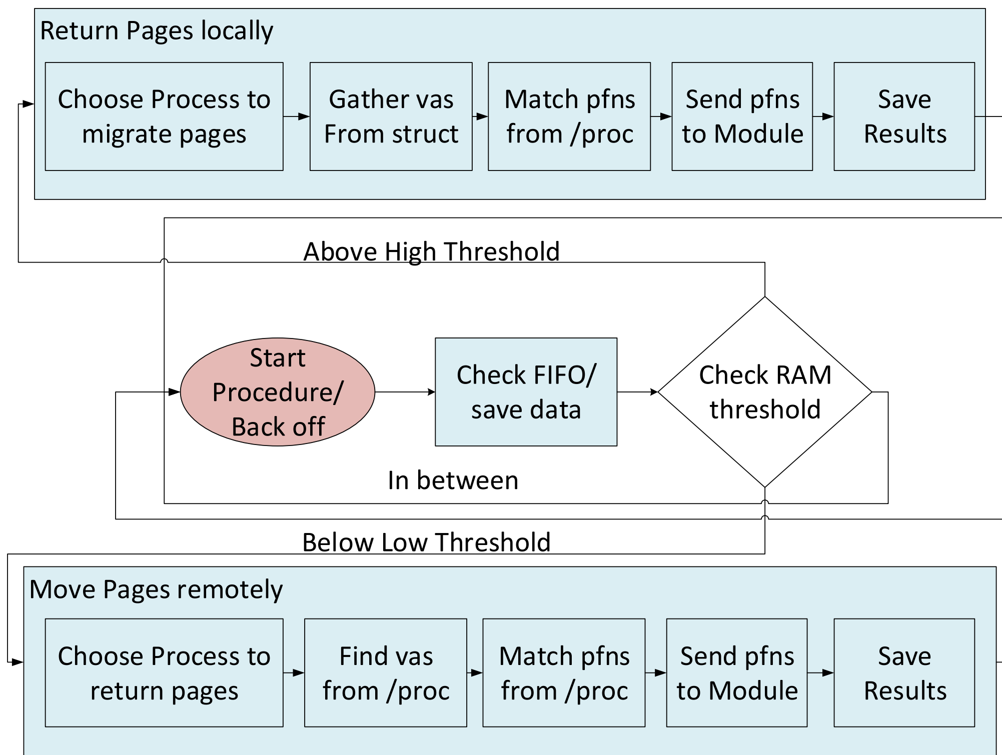

Figure 4.1: Flowchart of event processing by the Page Migration Monitor.

4.1.2 The process selection algorithm

Figure 4.2 presents the flowchart of the process selection algorithm. For the daemon to choose an appropriate process to move pages from one memory region to another, an algorithm is used which takes two parameters. A priority value, which is an integer value given by a process to the daemon through the registration message, and a migration threshold value which is incremented or decremented by

a value equal to the maximum amount of pages that can migrated at the same time. For this prototype we set that value to 8192 which is the maximum number of pages that the daemon can send to the module with one system call. The migration threshold value and the priority value start at 0(zero).

When the daemon searches through the registered processes to find a candidate for migration, it will start with searching all processes which have the current priority value. If a process has not migrated more pages than the threshold value, it is chosen as a candidate and the algorithm returns that process. If none of the processes of that priority meet that criterion, The priority value is incremented by one, and processes of that priority will be scanned. Every time the priority value changes, the algorithm examines if process of all priorities have been searched. If that is true, the migration threshold value is increased so that process can meet the threshold criterion. The last thing that happens a after the priority value is changed is that the daemon checks the RAM usage once more. If the usage is drops below the critical percentage, we exit the algorithm since we do not need to migrate the pages anymore.

The algorithm required for finding a process when we need to return pages locally is similar. The differences is that we decrement the priority value and migration threshold value when that is required.

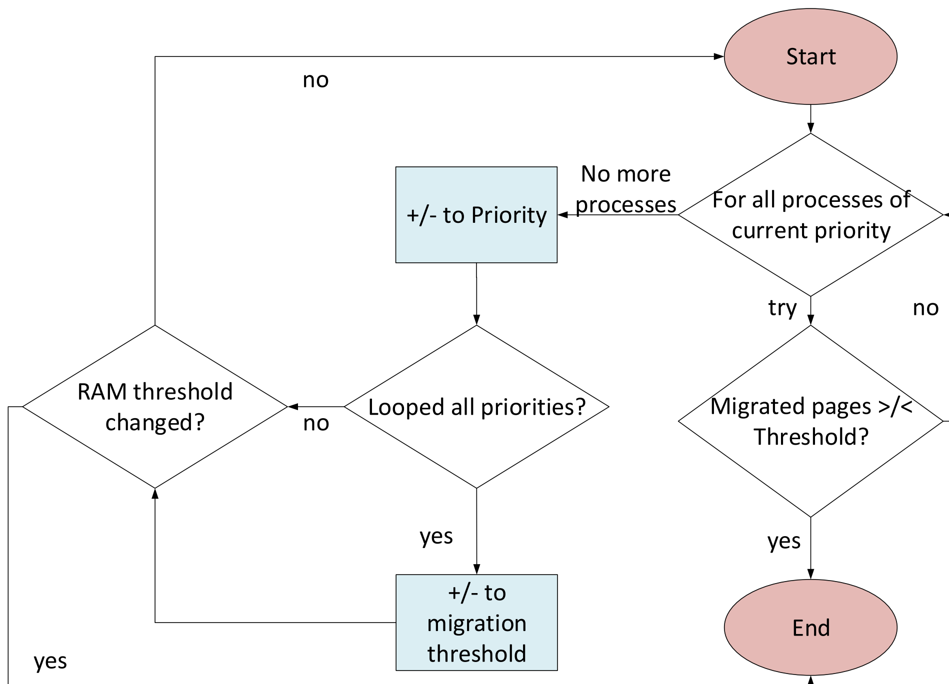

Figure 4.2: Flowchart of the process selection algorithm.

4.1.3 The daemon process data table

For the appropriate management of the processes that have been registered to the daemon, we require to store important information regarding the available remote memory, memory pages of registered processes as well as their memory location. The main data structure of the page migration daemon is the process data table. The table contains several numeric values used for the process selection algorithm such as the memory thresholds and priority values.

Figure 4.3 illustrates an overview of the process data table. The table consists of an array of size equal to available priority values. Every index of the array points to a double linked list where every node contains data of a particular process. Those data include registered memory ranges that are allocated by a process, an array of virtual addresses that point to migrated physical addresses and miscellaneous data such as the process id, priority etc. During every event, the daemon requires to perform an extensive search among the process data in order to fill a buffer with page frame numbers that will be sent to the module for page migration. In detail, for every page that is chosen, the array of virtual addresses must be iterated, in order to know if that page already resides in remote memory or not. In order to minimize the iterations, we store all virtual addresses in a sorted array. By using a sorted array, we can perform binary search and thus, if we want to validate that M pages exist in an array of N migrated pages, the time complexity of that process happens in $O(MlogN)$. Furthermore, when a migration procedure finishes, the pages that were recently migrated are also sorted. If we want to add M new pages to an array of N migrated pages, the time complexity of that process happens in $O(M + N)$.

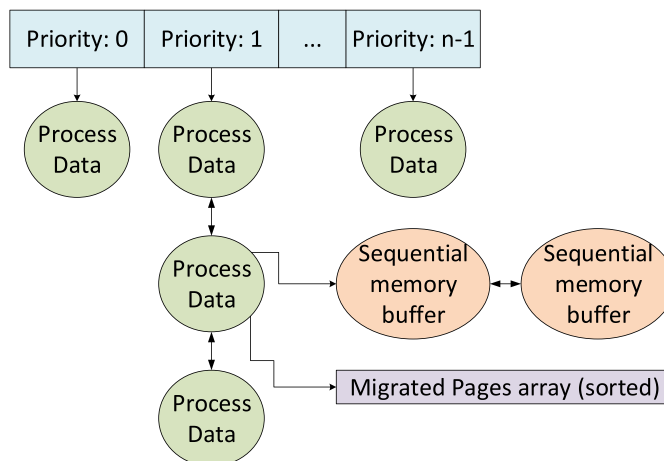

Figure 4.3: Overview of the process data table.

4.2 The pmm-sender library

The pmm-sender is user-space software library that a process must include and use its methods, in order to send sufficient information to the daemon, to perform migration on its pages. The library communicates with the daemon through a FIFO file. Figure 4.4 illustrates the inter-process communication between the daemon and processes using the library. The file is only written by the library and is only read by the daemon. The library implements the following API:

- **int pmm_init(uint16_t priority):** This method must be called before any other method of the library and is used to register a process to the daemon. The "priority argument" is used to declare the priority of the process in terms of how often its pages will be migrated, if required. A process with a higher priority will have its pages moved less frequently than a process with lower priority, if both of those processes are registered to the daemon. Current values range from 1 to 5. The priority and the process id are written to the FIFO file.

- **int pmm_send_buffer(void* virt_addr, long unsigned bytes):** This method is called, when the calling process wants to send memory to the daemon that is suitable for possible migration. The first argument is the virtual address at the starting byte of that memory, and the second argument specifies the size of that memory in sequential bytes. The method rounds up the bytes to make them multiple of a page size. Those 2 values then along with the process id, are written to the FIFO file.

- **int pmm_release_buffer(void* virt_addr):** This method is called when the calling process wants the daemon to unregister the memory that was registered before with the pmm_send_buffer call. The daemon removes any information regarding that memory such as the memory range and the pages of that range that are migrated. The daemon will not try to return any migrated pages locally, if that method is called. It is the calling process's responsibility to free the memory after the pmm_release_buffer call has returned.

- **int pmm_request_pages_lock(void* virt_addr):** This method tells the daemon that sequential memory that was sent with the previous method, and begins at the virtual address specified in the first argument, is considered critical and should be prioritized to return locally. At that point, that memory is protected by the daemon, and will not be migrated again. The virtual address along with the process id are written to the FIFO file.

- **int pmm_cleanup(void):** When the process calls this method, it sends termination information to the daemon. This means that the daemon will deregister the process and any information on pages that were stored. If this

method is called, it should happen right before the process terminates. However, if a process terminates without calling that method, and the daemon results in moving its pages, it will determine that it is not running anymore, and clear its data on its own value.

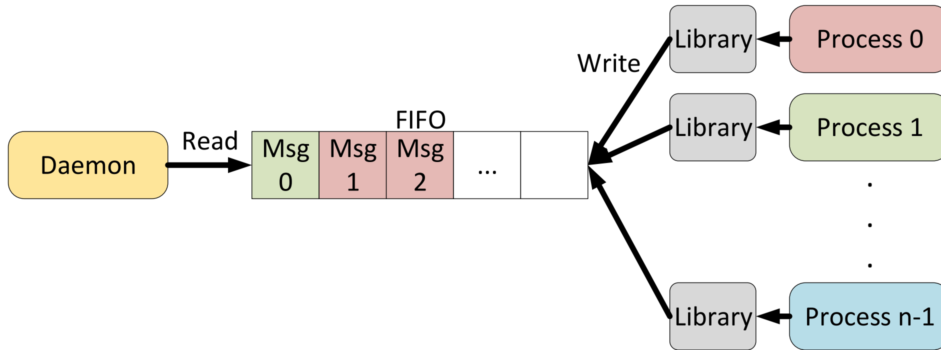

Figure 4.4: the inter-process communication between the daemon and processes using the library through a FIFO.

4.3 Patching the Linux Kernel

By patching the Linux Kernel, we enable remote memory to be used from our operating system after it is hotplugged. Furthermore we export and modify several functions that allow us to migrate memory pages to the hotplugged remote memory or return them to local memory if they have already been migrated.

In order to make remote memory accessible to the Operating System, we must integrate it to the Kernel data structures that manage system memory. The Linux Kernel has an architecture independent way of describing physical memory [11]. As mentioned in the previous chapter 3.2 memory is divided into blocks called zones. By gathering information regarding memory management from several books on the Linux Kernel [5] [18] [4], we defined a new memory zone. We made changes to the following files in the Linux Kernel to successfully create a new memory zone and modify the hotplug patch, to add memory that is being hotplugged to that specific memory zone.

- **include/linux/mmzone.h** In this file we add an extra entry in the enumeration type for zones.
- **mm/page_alloc.c** In this file we add the name of the zone as an array of characters. When reading the /proc/buddyinfo file, that name will appear for the newly created zone.
- **arch/arm64/mm/init.c** This is the initialization file for the whole memory system. Here we define the range of physical memory as defined in the device

tree at boot time, that should be adapted to each zone. Since the zone for the migration system needs to only acquire hotplugged memory, we set it to zero range, but we need to define it anyway, so that the operating system creates the basic data structures for the zone. In this file, the patch for the hotplug is added. We make a minor modification to the hotplug patch, by changing the zone where the memory would be hotplugged, to our special zone.

- **include/linux/gfp.h** In this header file, several flags are defined so that when added to page allocation calls, they tell the operating system to allocate memory from a specific memory zone. There is also the GFP-zone-table which is a bit-string that is used in a complex bit-wise operation along with the GFP flag of the page allocation call. The result of that operation is the zone that will be used for memory allocation. We defined our GFP flag accordingly so that when it is used in the allocation call, it will return the zone used for page migration. We also defined our zone in a way that a different GFP flag will never return that zone. This ensures that any memory allocation call by a third party will not allocate memory in our zone.

For the page migration to happen, we have exported several functions from the kernel, which are used for moving, isolating and referencing pages. Below we explain the use of the functions we have exported from the kernel.

- **isolate_lru_page:** Increases the references to the page so that it cannot vanish while the page migration occurs. It also prevents the swapper or other scans to encounter the page.

- **migrate_pages:** This is the core function that we require in order to move a list of pages. As derived from kernel Documentation, this function performs many tasks including, locking page accesses from any source while data is moving, allocating the new pages, coping data from old pages to new pages, and freeing old pages from memory. This particular function uses as parameters a page-list of page structures that will be moved from one memory zone to another. There is also a parameter of type enum migrate_reason which is used by the operating system to understand the reason why migrate_pages was called, like for memory compaction, memory failure etc. We added our own entry into that enumeration type so that when we call migrate_pages from our Kernel module, the procedure will follow the path that we want.

4.4 The page-migrator Kernel Module

The page migrator is a kernel Module, waiting to receive data from the page migration monitor. When the module file is written by the daemon, the Module receives a buffer of page frame numbers. The buffer also contains a bit, of which its value indicates whether its pages should be migrated to the memory zone, or be returned from the memory zone. Figure 4.5 presents the operations that the page-migrator Kernel Module performs when the daemon schedules a migration of pages. In depth, the Module performs the following actions:

- The Module scans the whole buffer of page frame numbers. Because of the limited address space of the kernel, we use a smaller buffer in the module, where we copy parts of the bigger, user buffer. When we finish scanning each part, we copy back to the user the resulting buffer, and we move to the next part until we have scanned the whole buffer. The first part of the buffer also contains the bit which indicates whether migrate the pages to the zone, or return them from the zone to local memory.
- For every page that we read from the buffer, we check there are still references to that page. If that is true, we get the struct page of that pfn.
- Next, we increase the reference of the page by one. This is required before we perform page isolation, so that even if no-one else refers to that page, we can still access the page struct. We then perform isolation. If isolation succeeds, we add that page struct to a page list.
- When the whole buffer is scanned, and the page list is not empty, we call migrate_pages to complete the operation.

If the process of copying the buffer or the migration procedure fails, the module returns an error value to tell the daemon to restart the process. In any other case, the module returns the number of pages that were successfully migrated and their frame numbers in the modified buffer.

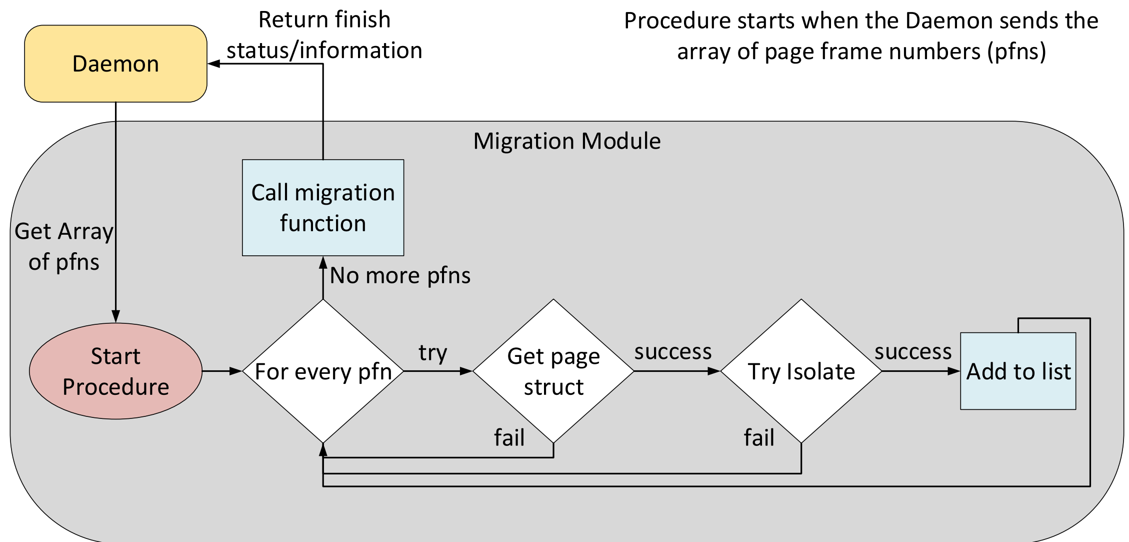

Figure 4.5: Flowchart of the page-migrator Kernel Module.

Chapter 5

Evaluation

Our goal is to test the resilience of our system when it is used on applications that stress the memory usage, observe the effect of performance on processes that have their data moved to remote memory through the Page Migration System, demonstrate that by migrating infrequently used data, we can achieved reduced latency compared to when we migrated continuously used data, and also ascertain how much temporal locality in processes can minimise their latency on remote memory access.

Consequently, we evaluate the Page Migration System on several benchmarks that fulfill the prerequisites mentioned above. We run the benchmarks using several different configurations and options, gather the results and analyze them.

5.1 Testbed

Figure 5.1 presents the overview of the testbed. Our testbed consists of 2 computer nodes and a network switch. Each computer node uses a Zynq UltraScale+ MPSoC containing four ARM Cortex A53 Cores and a XCZU9EG-2FFVC900 FPGA, 2GB of DDR4 SDRAM where 256MB are unmapped by the node, 16GB Micro SD for storage and Linux OS with 4.9 Kernel installed. The switch also uses a Zynq UltraScale+ MPSoC programmed as a network node for remote data transfers between the two computer nodes. The network interconnection design is developed in the context of the ExaNeSt project.

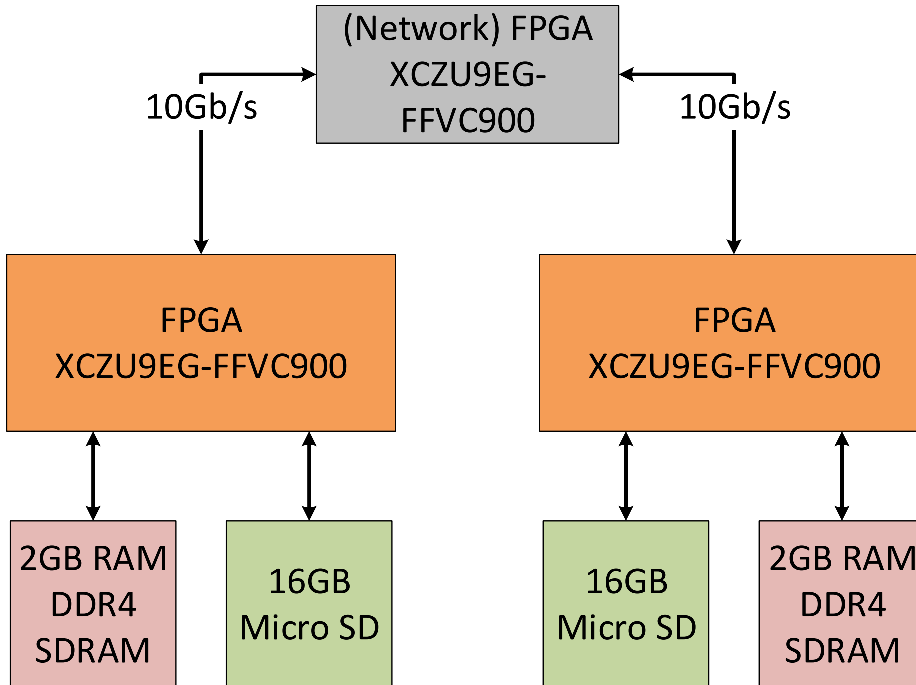

Figure 5.1: Testbed Overview.

5.2 Benchmark: Lmbench

Lmbench [22] is a suite of portable, ANSI/C micro-benchmarks for UNIX/POSIX. In general, it measures two key features: latency and bandwidth. We use lmbench to measure the memory latency when accessing remote memory compared to local memory. For memory bandwidth, lmbench uses the Stream benchmark, which we use in the next section 5.3.

For memory latency, we use the *lat_mem_rd* tool of lmbench which counts nanoseconds per load operation in an array, by defining an array size in MB, and a stride value in Bytes which signifies the gap between sequential memory accesses in the array. A larger stride value will cause more latency due to a greater amount of cache misses. When a load operation generates a last level cache(LLC) miss, and the requested data reside in remote memory, the system will have to wait for the local pl to send the read request to the remote pl and wait for the data to arrive through the NIC. That equals a round-trip in the NIC.

5.2.1 Lmbench Setup

We want to compare the memory read latency of the local DRAM with the equivalent of the remote DRAM. . we perform several runs which we divide into local

and remote sets. In the local sets of runs, data reside in local DRAM, while in the remote sets of runs, data reside in remote DRAM. Since we have 256MB available remote memory in our prototype testbed, we set the array size 256MB in all runs. For every set we define strides of 16, 64, 256, 1K, 4K, 16K, 32K and 64K bytes. In our case the latency did not increase when we measured beyond a 64K stride value, so we display the results up to 64K. Right before initialization we call pmm_init to register the process to the daemon, then we call pmm_send_buffer to send the daemon the array that we want it to migrate when RAM drops below the specified threshold, and right before the benchmark terminates, we call pmm_cleanup to deregister the process from the daemon.

5.2.2 Lmbench Results

Figure 5.2 and Tables 5.1 and 5.2 below, show the results. We observe that when we increase the stride from 256 bytes to 1KB, the latency increases at an exceptional rate compared to the other pairs of strides both at the local and remote configuration, but even more at the remote configuration due to the NIC latency. We assume that this happens due to the prefetcher of the cache. A cache line in l1 and l2 cache of the ARM Cortex a53 has a fixed size of 64 bytes. If the latency increases exponentially after the 256-byte stride, that would probably mean that the prefetcher adds about three more cache lines after a cache miss, and thus if we use a stride of more than 256 bytes, we greatly increase the rate of cache misses. This is verified in Chapter 6, Section 6.6.2 of The ARM Technical Reference Manual for Cortex-A53 [3] where it is written that the prefetcher recognizes a sequence of data cache misses at a fixed stride pattern that lies in four cache lines, plus or minus.

Stride	16	64	256	1024
local(ns/ld)	5.502	21.908	27.027	102.777
remote(ns/ld)	108.039	430.769	512.248	2003.609

Table 5.1: Lmbench Memory Read Latency, Array size: 256MB pt1

Stride	4096	16384	65536
local(ns/ld)	108.039	127.048	132.976
remote(ns/ld)	2010.261	2020.660	2043.296

Table 5.2: Lmbench Memory Read Latency, Array size: 256MB pt2

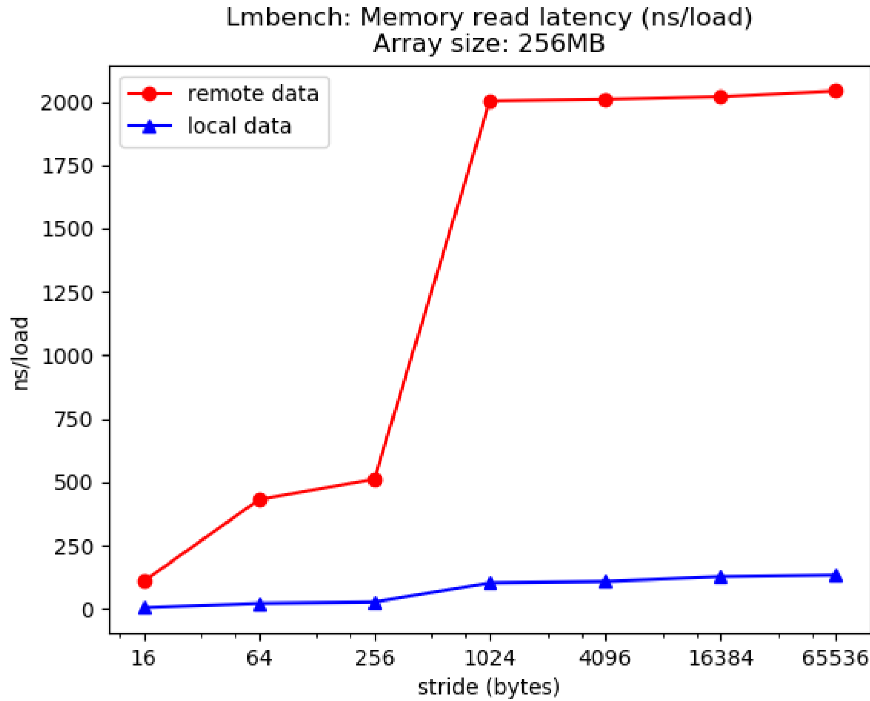

Figure 5.2: Lmbench: Memory Read Latency for local and remote configuration

5.3 Benchmark: Stream

The STREAM [21] benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. The benchmark allocates three one-dimensional equally sized arrays a, b and c. and iterates over them performing four functions over every index. Copy, Scale, Add and Triad. Copy. After the arrays are initialized, the above for algorithms are executed in order:

- Copy: $c[i] = a[i],\ for\ i \leftarrow 0\ to\ array_elements$
- Scale: $b[i] = c[i] * const,\ for\ i \leftarrow 0\ to\ array_elements$
- Add: $c[i] = a[i] + b[i],\ for\ i \leftarrow 0\ to\ array_elements$
- Triad: $a[i] = b[i] + c[i] * const,\ for\ i \leftarrow 0\ to\ array_elements$

We made a small modification by converting all three of the arrays from one-dimensional to two-dimensional. The reason was to display a simple example, where rarely used or unused allocated data of a process that is migrated to remote memory, and will have minimal to zero performance drop, compared to keeping every data in memory locally. When Stream starts, the user can specify which

parts of the arrays will be computed. There are 3 options. 'Odd', 'even' or 'all'. 'Odd' computes only the columns of odd index for every array, 'even' computes only the columns of even index, and 'all' computes the whole array.

5.3.1 Stream Setup

For our tests we used two different array sizes. Since the available remote memory of our computer nodes is 256MB, we defined the array size such as, that a whole array can be migrated to remote memory. For every run we test a good scenario where we migrate part of the arrays that is not used for computation, and a bad scenario where we migrate part of the arrays that will be computed. For our first setup, we want to migrate one array at a time. We set each array to have the size of 512MB for a total memory usage of 1536MB. Since we only compute half of the whole arrays, each half part is exactly 256MB and can fit into remote memory. For our second setup we want to migrate two arrays into remote memory, so we set the size of the arrays to 256MB for a total of 768MB. Each half part of the array that will be computed, is now 128MB and thus we can fit two arrays in remote memory. For every run we migrate the odd columns, of the arrays, but in the good scenario, we perform computations on the even columns, while on the bad scenario we do that for the odd columns. Finally, we run every configuration for 1, 2, 3 and 4 threads.

5.3.2 Stream Results

For the good scenarios where we migrated only unused memory, performance was equal to having all of our data in local memory as expected, so we omitted those results from the thesis. Next we analyze the results for the bad scenarios, where we migrate used memory.

For the Copy algorithm the results are shown on tables 5.3 and 5.4 and on figures 5.3 and 5.4. Every iteration performs $c[i] = a[i],\ for\ i \leftarrow 0\ to\ array_elements$. This means that we have one load operation on a, and one store operation on c for every iteration. For Figure 5.3, migrating the b array gives us performance equal to the local configuration. That is because in the Copy algorithm, b is not used at all, and so every computation is performed locally. Migrating the c array gives better performance than migrating array a. Let's examine array a first. As we explained in section 5.2,for a load operation that generates an LLC miss, we pay latency for a round-trip to NIC fetching a cache line. During eviction, since data are not modified, there is no need for the system to write them back to external memory. When we migrate the array c on the other hand, if we generate an LLC miss, a request from the local PL will be sent to remote PL to bring data from remote memory. When the line is fetched, in will be stored in the Store buffer of the processor as described in Chapter 2 in section 2.1.6 of the Arm a53 TRM [3], but since we only overwrite the $c[i]$ without reading its data, the store operation will allocate enough space in the store buffer for the incoming cache line, write

the data of that double word in the cache line, and the rest of the line will be brought from remote memory asynchronously. That way we do not wait for the round-trip to complete, and since all sequential data of the same line and nearby lines are only overwritten and are never read, we never need to wait for the data to come from remote memory. During line eviction from the LLC, there is usually a write buffer, also called an eviction buffer or a victim buffer, where evicted data are stored there, and at a later time or when the buffer is full, lines in that buffer will be written asynchronously in main memory. Consequently, migrating the c array is faster than migrating the a array, due to a having more latency because of more round-trips in the NIC and due to the fact that the system will have to wait on all those round-trips.

For Figure 5.4, migrating the b and c arrays gives the best performance compared to the other 2 remote configurations, since the b array is not used in the Copy algorithm, and migrating c performs better than a as explained in the previous paragraph.

We notice that no matter how many threads we use when accessing remote memory, performance does not improve. We speculate that this happens due to the fact that with our prototype the NIC uses one network path for every transaction and we are already throughput-bound by one thread.

Configuration	1 Thread	2 Threads	3 Threads	4 Threads
local(MB/s)	5015.8	7471.8	8939.2	9659.1
a remote(MB/s)	64.5	64.6	64.6	64.6
b remote(MB/s)	4993.6	7571.4	9037.8	9708.7
c remote(MB/s)	518.0	518.1	516.7	517.9

Table 5.3: Stream Copy algorithm, Array size 512MB

Configuration	1 Thread	2 Threads	3 Threads	4 Threads
local(MB/s)	4960.9	7551.0	8995.0	9726.7
a,b remote(MB/s)	64.8	64.9	64.9	64.8
b,c remote(MB/s)	520.0	520.4	518.9	520.3
a,c remote(MB/s)	64.4	64.4	64.1	47.7

Table 5.4: Stream Copy algorithm, Array size 256MB

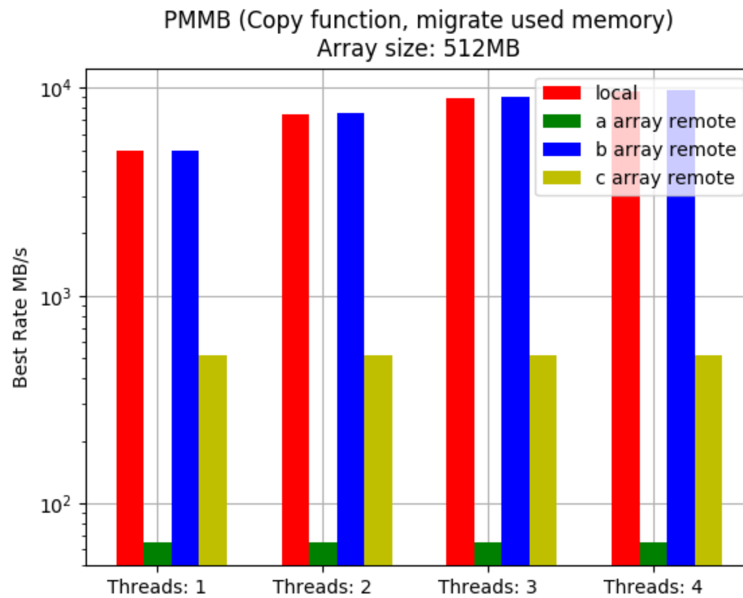

Figure 5.3: Stream Copy algorithm. Local configuration vs one array remote configurations, migrating used memory for 1 to 4 threads.

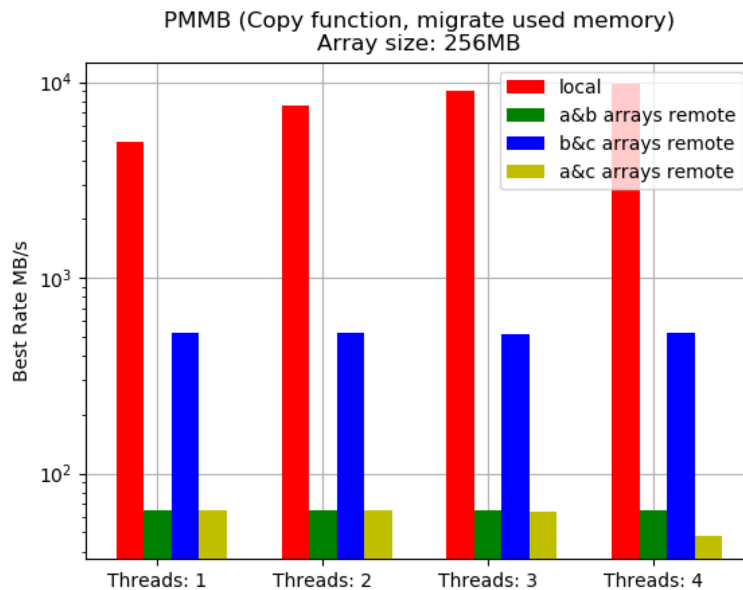

Figure 5.4: Stream Copy algorithm. Local configuration vs two arrays remote configurations, migrating used memory for 1 to 4 threads.

For the Scale algorithm the results are shown in Tables 5.5 and 5.6 and in Figures 5.5 and 5.6. Every iteration performs: $b[i] = c[i] * const,\ for\ i \leftarrow 0\ to\ array_elements$. This means that we have one load operation on c, and one store operation on b for every iteration. The results for these test runs are similar to the Copy algorithm, with the difference that now the b is the array being written instead of c, and c is the array being read instead of a. Migrating only the array a, has no performance drop since it is not used.

Configuration	1 Thread	2 Threads	3 Threads	4 Threads
local(MB/s)	2014.3	3994.2	5767.0	7676.5
a remote(MB/s)	2011.9	3982.9	5774.5	7626.4
b remote(MB/s)	517.4	517.9	517.1	518.2
c remote(MB/s)	64.5	64.6	64.6	64.6

Table 5.5: Stream Scale algorithm, Array size 512MB

Configuration	1 Thread	2 Threads	3 Threads	4 Threads
local(MB/s)	2021.5	4025.7	5772.3	7608.7
a,b remote(MB/s)	515.8	516.8	515.6	516.9
b,c remote(MB/s)	64.2	64.2	63.8	48.8
a,c remote(MB/s)	64.3	64.4	64.4	64.4

Table 5.6: Stream Scale algorithm, Array size 256MB

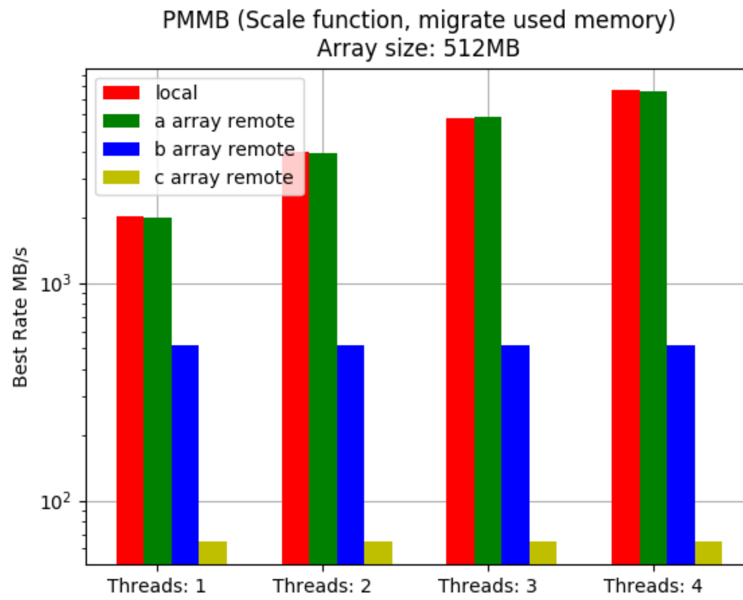

Figure 5.5: Stream Scale algorithm. Local configuration vs one array remote configurations, migrating used memory for 1 to 4 threads.

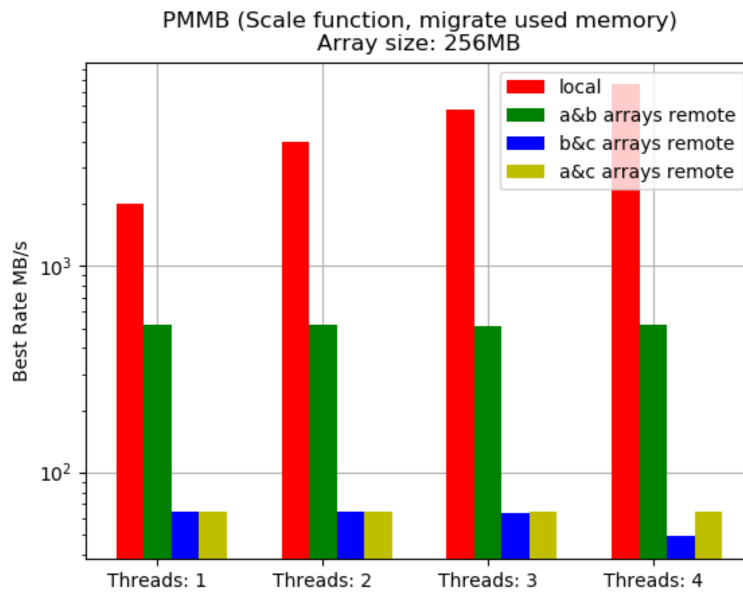

Figure 5.6: Stream Scale algorithm. Local configuration vs two arrays remote configurations, migrating used memory for 1 to 4 threads.

For the Add algorithm the results are shown in Tables 5.7 and 5.8 and in Figures 5.7 and 5.8. Every iteration performs: $c[i] = a[i]+b[i],\ for\ i \leftarrow 0\ to\ array_elements$. In this algorithm the array c is written while the arrays a and b are read. Like the previous algorithms we experience better bandwidth when we migrate the c array than the other two. In the configurations where we migrate two arrays, migration of $b + c$ and $a + c$ get double bandwidth than $a + b$ because of the c array.

Configuration	1 Thread	2 Threads	3 Threads	4 Threads
local(MB/s)	2404.8	4362.4	5554.1	7239.4
a remote(MB/s)	96.8	96.9	96.7	96.6
b remote(MB/s)	96.6	96.9	96.8	96.7
c remote(MB/s)	774.4	775.4	774.3	774.1

Table 5.7: Stream Add algorithm, Array size 512MB

Configuration	1 Thread	2 Threads	3 Threads	4 Threads
local(MB/s)	2405.4	4427.1	5592.7	7333.2
a,b remote(MB/s)	48.4	48.4	48.4	48.4
b,c remote(MB/s)	95.6	95.6	95.5	95.1
a,c remote(MB/s)	96.7	96.7	96.1	95.7

Table 5.8: Stream Add algorithm, Array size 256MB

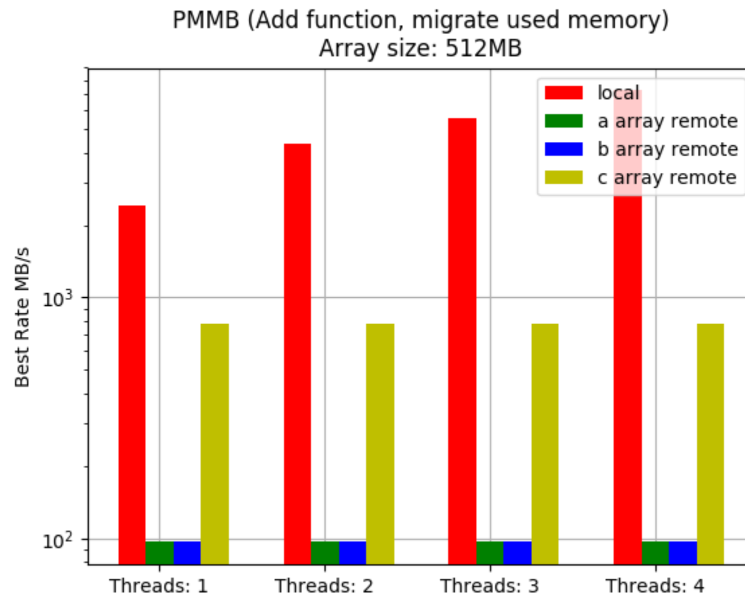

Figure 5.7: Stream Add algorithm. Local configuration vs one array remote configurations, migrating used memory for 1 to 4 threads.

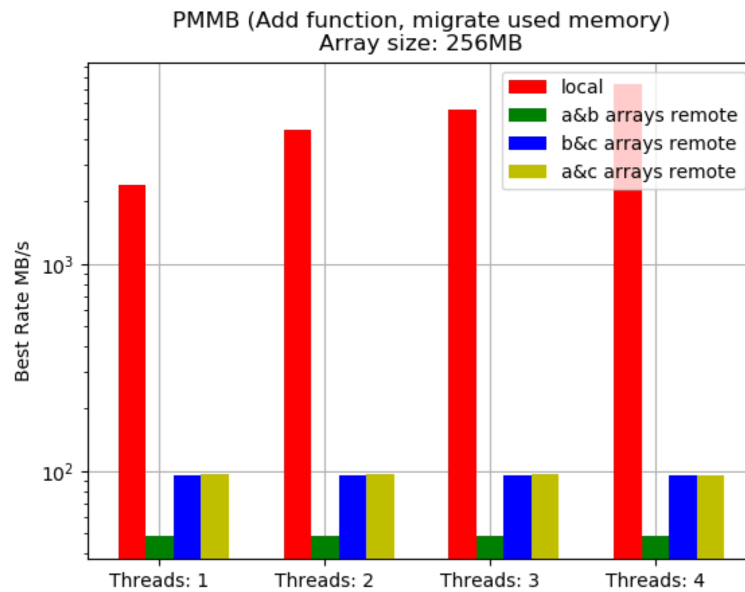

Figure 5.8: Stream Add algorithm. Local configuration vs two arrays remote configurations, migrating used memory for 1 to 4 threads.

For the Triad algorithm the results are shown in Tables 5.9 and 5.10 and in Figures 5.9 and 5.10. Every iteration performs: $a[i] = b[i] + c[i] * const,\ for\ i \leftarrow 0\ to\ array_elements$. In this algorithm the array a is written while the arrays b and c are read. Here we experience better bandwidth when we migrate the a array than the other two. In the configurations where we migrate two arrays, migration of $a + c$ and $a + b$ get double bandwidth than $b + c$ because of the a array.

Configuration	1 Thread	2 Threads	3 Threads	4 Threads
local(MB/s)	1957.6	3847.3	5393.6	6802.7
a remote(MB/s)	773.2	775.1	774.7	776.9
b remote(MB/s)	96.6	96.9	96.8	96.7
c remote(MB/s)	96.7	96.8	96.7	96.6

Table 5.9: Stream Triad algorithm, Array size 512MB

Configuration	1 Thread	2 Threads	3 Threads	4 Threads
local(MB/s)	1957.8	3839.0	5393.7	6764.8
a,b remote(MB/s)	95.8	96.0	95.5	95.1
b,c remote(MB/s)	48.3	48.4	48.4	48.4
a,c remote(MB/s)	96.0	96.0	95.4	95.1

Table 5.10: Stream Triad algorithm, Array size 256MB

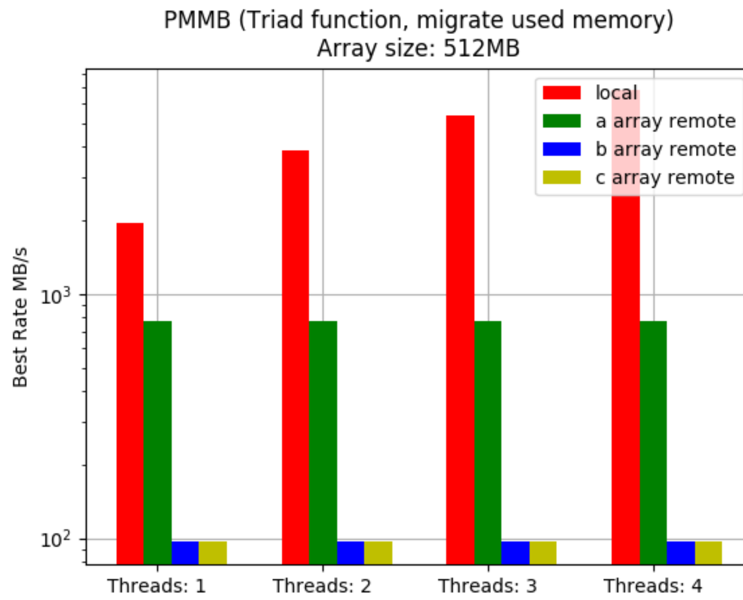

Figure 5.9: Stream Triad algorithm. Local configuration vs one array remote configurations, migrating used memory for 1 to 4 threads.

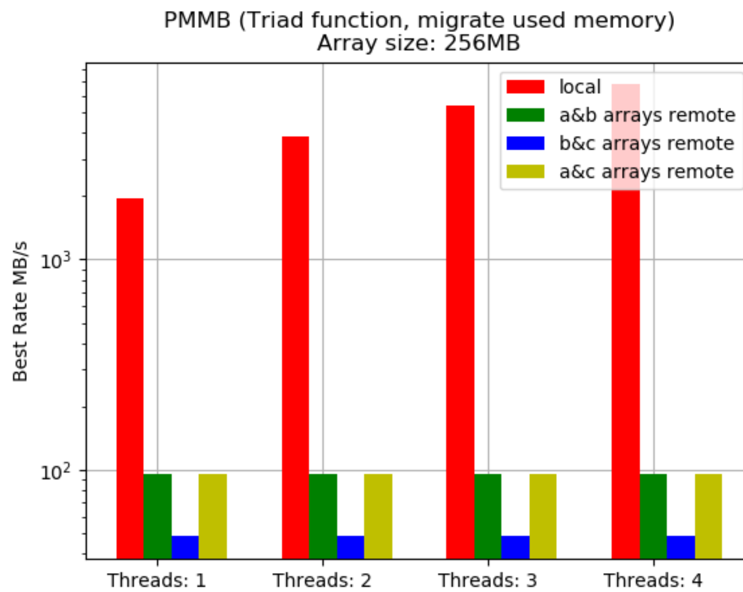

Figure 5.10: Stream Triad algorithm. Local configuration vs two arrays remote configurations, migrating used memory for 1 to 4 threads.

5.4 Benchmark: Himeno

The Himeno [12] benchmark evaluates performance of incompressible fluid analysis code. It takes measurements to proceed major loops in solving the Poisson's equation solution using the Jacobi iteration method. Effectively, Himeno allocates several 3-Dimensional float arrays and performs computations on them, using a stencil algorithm. There are different implementations of that benchmark regarding the memory allocation and the parallelism. We chose the dynamic allocation with Open-MP threads version. Himeno runs the loop of the algorithm 3 times on initialization phase to calculate the processor frequency, and then determines how many times the loop should run at running phase, so that the benchmark takes about a minute to finish. In the end, the benchmark prints the MFLOPs rate per second, for the minute it ran.

The benchmark contains five configurations for different array sizes, extra-small, small, middle, large, and extra-large. The middle configuration allocates 128x128x256 arrays which result in 300MB memory usage, which is low considering the available memory of our compute nodes, while the next available configuration of large, allocates 256x256x512 arrays which ends up using more than the available memory of the node. To run the benchmark based on our needs, we created a custom configuration that allocates 224x224x448 arrays essentially using 1.2GB of RAM.

5.4.1 Himeno Setup

Except from the configuration modification, we also call the API functions of our library in the benchmark. Right before initialization we call pmm_init to register the process to the daemon, then we call pmm_send_buffer to send the daemon the arrays that we want it to migrate when RAM drops below the specified threshold, and right before the benchmark terminates, we call pmm_cleanup to deregister the process from the daemon. For our configuration, four arrays p, bnd, wrk1 and wrk2 have 85.5MB size except for the arrays a, b and c where a is exactly 4 times larger than the normal array size, and b and c which are 3 times larger than the normal array size. This allows us to fully migrate almost 3 of the smaller arrays, b or c alone, and the 3/4 of a. We ran every possible combination, along with keeping all data locally for 1, 2, 3 and 4 threads, and we show the results of the local run, the best memory migration run and the worst memory migration run for every thread option.

5.4.2 Himeno Results

Figure 5.11 below, shows the results. Although the local configuration calculates more MFLOPS than any remote configuration, the remote configuration of migrating bnd, wrk1, and wrk2 arrays, achieves performance close to the local configuration for 1, 2 and 3 threads. By examining the source code, we observe that

for every iteration of the algorithm, the iteration index of bnd and wrk1 arrays is only read once, and the index of wrk2 array is read and written once. On the other hand, p array has 20 read accesses and 1 write access, and arrays a, b and c have 3 read accesses. This means that bnd, wrk1, and wrk2 are more rarely used than the other arrays, so for the remote configurations, the process executes less remote read and write operations over the network interconnection. The b configuration gets a boost in performance as we increase the number of threads to the point that the configuration of 4 threads gives better performance than the bnd, wrk1, wrk2 configuration. Deductively, this happens because there is increased cache locality for array b due to the way the arrays are accessed.

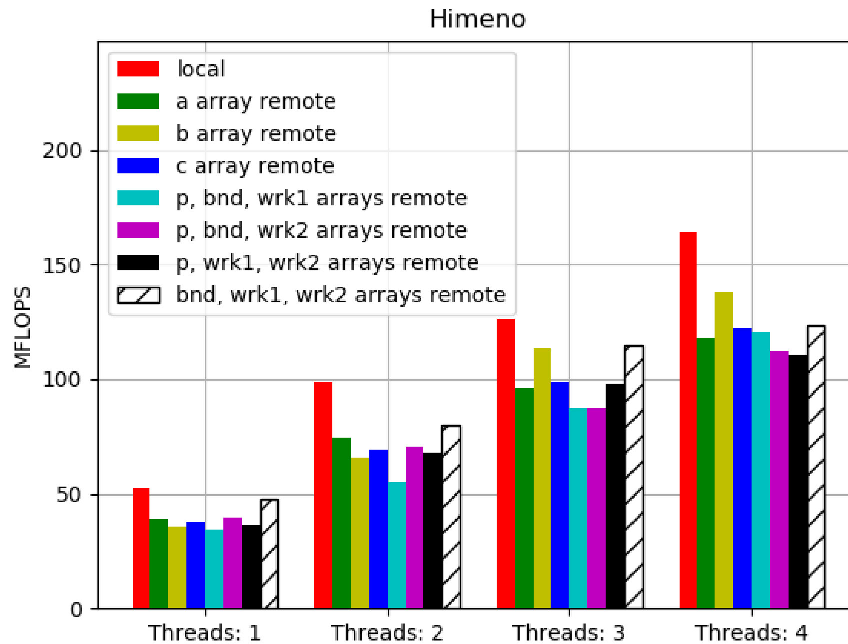

Figure 5.11: Himeno Benchmark. MFLOPS for local and remote configurations for 1, 2, 3 and 4 threads.

5.5 Benchmark: StencilProbe

Stencilprobe [15] is a small, self-contained serial micro-benchmark that was developed as a tool to explore the behaviour of grid-based computations. Effectively, it mimics the kernels of applications that use stencils on regular grids. In this way, it can simulate the memory access patterns and performance of large applications, as well as be used as a testbed for potential optimizations, without having to port or modify the entire application. Stencilprobe dynamically allocates 2 3-Dimensional

arrays and uses a 7-point 3D von Neumann style algorithm. It also uses a time step configuration, where the algorithm will run as many times as the time step parameter is specified. Finally, the Stencilprobe implements four different algorithms, 2D cache blocking, cache oblivious, time skewing and circular queue, that calculate the same result, each using a different cache blocking technique.

For this benchmark we want to take advantage of the cache blocking techniques and reduce the latency created by remote memory access, by keeping remote data in cache as much as possible. In each algorithm, cache block size is specified by the user except from the cache oblivious algorithm, where it uses recursion to perform cache blocking implicitly. Stencilprobe dynamically allocates two arrays A0 and Anext. On the first time step A0 is used for reading stencil data and the results are written to Anext. On every next time step, the two arrays swap the roles they had in the previous time step.

5.5.1 Stencilprobe Setup

In order to show the effect, a good cache blocking technique would have on our system, we did test runs with every cache block combination possible on each algorithm, where A0 and Anext each are of size 512x256x256. The tests were run with all data locally. Then, for every algorithm we chose the cache block configuration giving the optimal and worst performance. For every configuration of array size 512x256x256 and 6 time steps, we ran the benchmark keeping all data local, migrating the array A0 and migrating the array Anext. Right before initialization we call pmm_init to register the process to the daemon, then we call pmm_send_buffer to send the daemon the array that we want it to migrate when RAM drops below the specified threshold, and right before the benchmark terminates, we call pmm_cleanup to deregister the process from the daemon.

5.5.2 Stencilprobe Results

Figure 5.12 to Figure 5.18 below, illustrate the results for each configuration. The blocked configuration requires the user to define a 2D cache block. In Figure 5.12 and Figure 5.13, we observe that using the worst cache block of 16x512, we get a slowdown of 2.708X (157.746 seconds) for A0 configuration and 2.597X (151.263 seconds) compared to the local configuration of 58.246 seconds, while for the optimal cache block of 256x32 we only get a slowdown of 1.221X (55.097 seconds) for A0 configuration and 1.225X (55.287 seconds) compared to the local configuration of 45.133 seconds. Cache locality with this algorithm gives us a significant boost in performance.

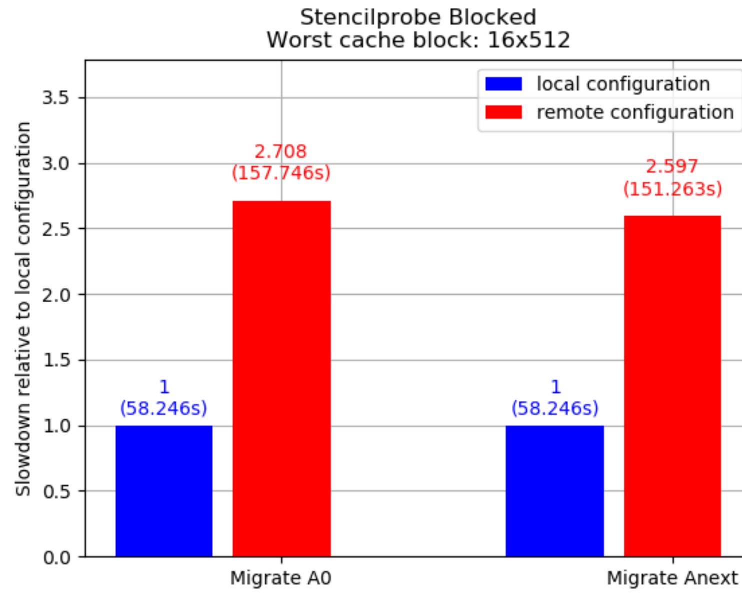

Figure 5.12: Stencilprobe blocked algorithm with worst cache block. Slowdown of remote configurations compared to local configuration.

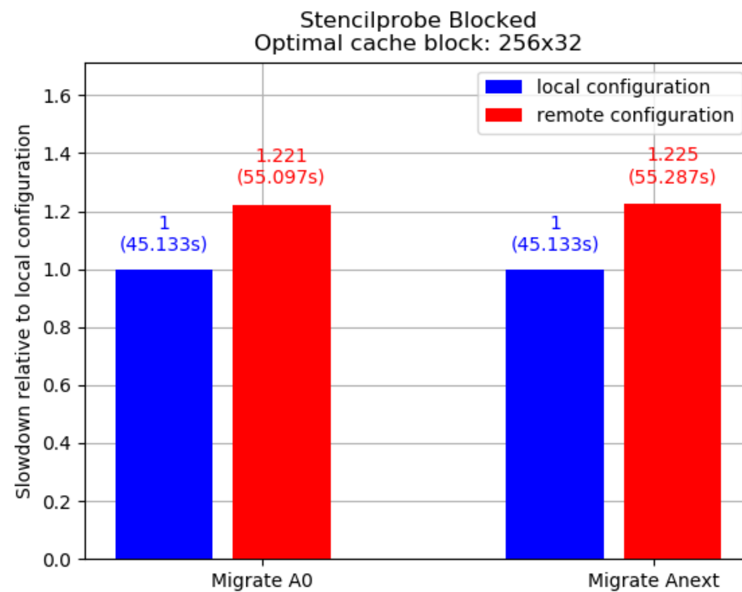

Figure 5.13: Stencilprobe blocked algorithm with optimal cache block. Slowdown of remote configurations compared to local configuration.

The time skewing configuration requires the user to define a 3D cache block where every dimension of the cache block must be a multiple of the corresponding dimension of array size -2 (e.g. here we have y:256 array size, so the y cache block must be a multiple of 254). In Figure 5.14 and Figure 5.15, we observe that using the worst cache block of 1x1x127, we get a slowdown of 4.551X (676.096 seconds) for A0 configuration and 4.525X (672.343 seconds) compared to the local configuration of 148.573 seconds, while for the optimal cache block of 255x2x2 we only get a slowdown of 1.293X (58.154 seconds) for A0 configuration and 1.274X (57.299 seconds) compared to the local configuration of 44.975 seconds. Like the blocked configuration, cache locality with this algorithm also increases performance.

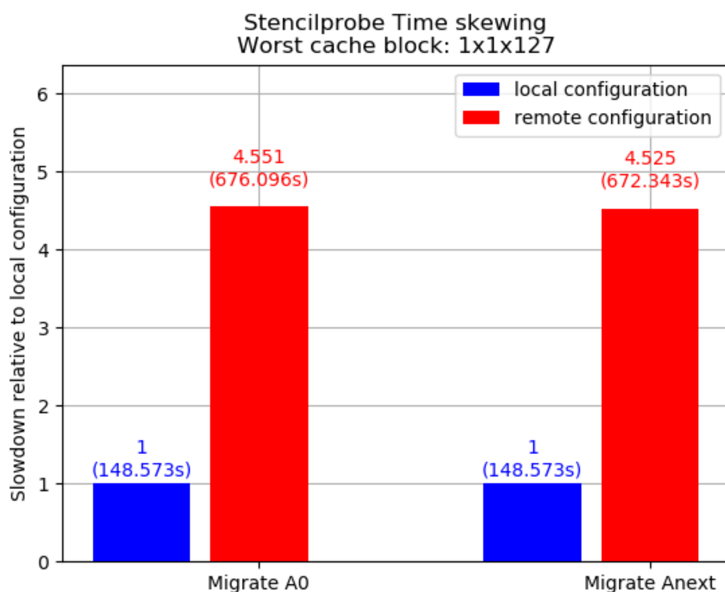

Figure 5.14: Stencilprobe time skewing algorithm with worst cache block. Slowdown of remote configurations compared to local configuration.

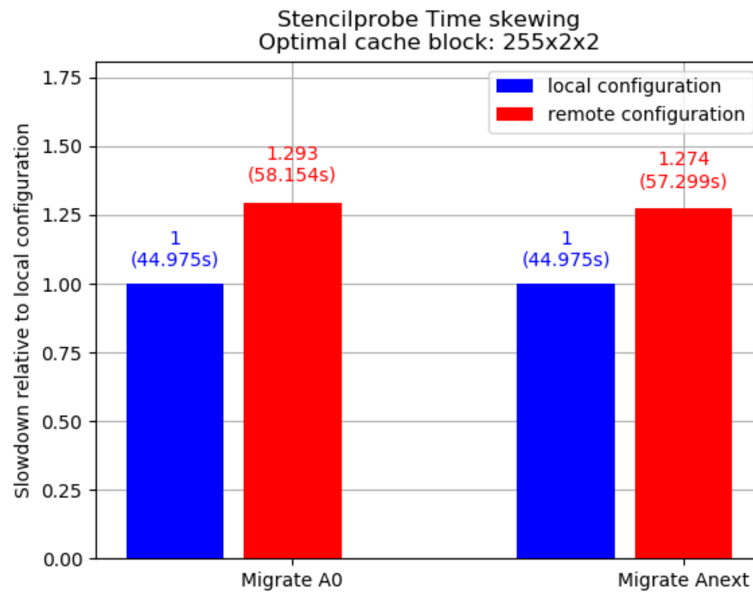

Figure 5.15: Stencilprobe time skewing algorithm with optimal cache block. Slowdown of remote configurations compared to local configuration.

The circular queue configuration requires the user, to only set the y dimension cache block. In Figure 5.16 and Figure 5.17, we observe that using the worst cache block of y:254, we only get a slowdown of 1.116X (323.201 seconds) for A0 configuration and almost no slowdown of 289.674 seconds compared to the local configuration of 289.576 seconds, while for the optimal cache block of y:1 we only get a slowdown of 1.164X (59.542 seconds) for A0 configuration and almost no slowdown of 51.193 seconds compared to the local configuration of 51.174 seconds. For the Circular queue, we observe that even with the worst cache block, the remote configuration's performance is close to the local. This could mean that we have good cache locality even for y:1, but the algorithm in general performs more read/write operations and thus the performance of both the local and remote configuration drop dramatically.

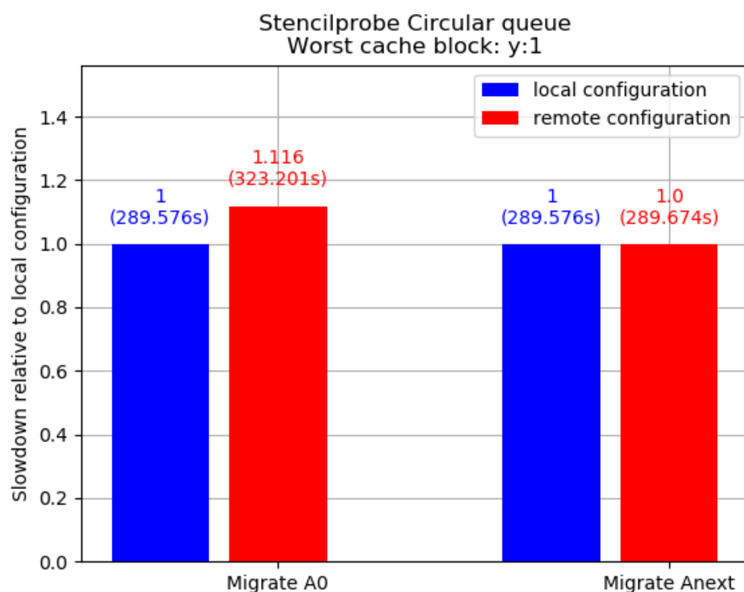

Figure 5.16: Stencilprobe circular queue algorithm with worst cache block. Slowdown of remote configurations compared to local configuration.

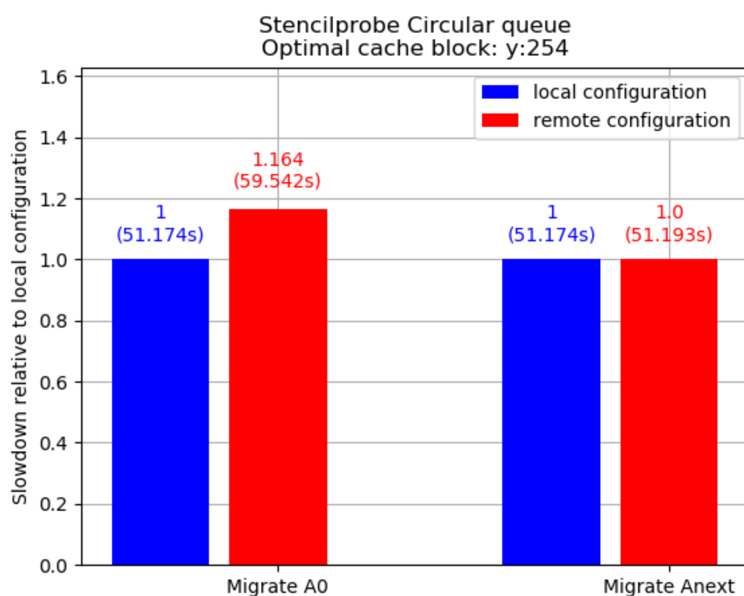

Figure 5.17: Stencilprobe circular queue algorithm with optimal cache block. Slowdown of remote configurations compared to local configuration.

For the cache oblivious configuration, we cannot define the cache block size. Instead, the algorithm performs cache blocking implicitly, using recursion. In Figure 5.18, we observe we only get a slowdown of 1.13X (76.157 seconds) for A0 configuration and 1.121X (75.541 seconds) compared to the local configuration of 67.408 seconds.

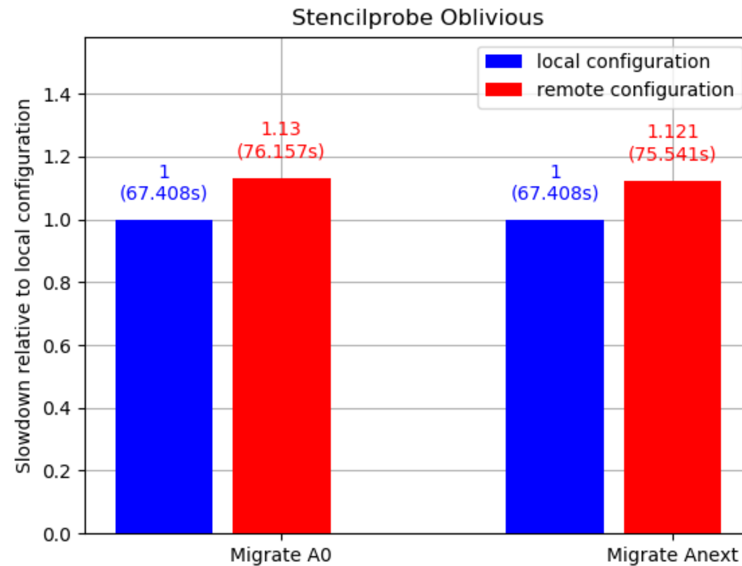

Figure 5.18: Stencilprobe cache oblivious algorithm with worst cache block. Slowdown of remote configurations compared to local configuration.

5.6 Benchmark: HPL

HPL [1] is a software package that solves a random dense linear system in double precision arithmetic on distributed-memory computers using MPI. Effectively it solves a linear system of order n: $Ax = b$ by first computing the LU factorization with row partial pivoting of the n-by-n+1 coefficient matrix $[Ab] = [[L, U]y]$. The data is distributed onto a two-dimensional P-by-Q grid of processes according to the block-cyclic scheme to ensure ”good” load balance as well as the scalability of the algorithm. The n-by-n+1 coefficient matrix is first logically partitioned into nb-by-nb blocks, that are cyclically ”dealt” onto the P-by-Q process grid. This is done in both dimensions of the matrix.

The source contains an HPL.dat input file where the user can specify the benchmark configuration. Parameters such as the matrix order size, the block size or the P-by-Q grid size as well as algorithm modifications are configurable. The user can also set multiple configurations of the same parameter, like multiple orders of P-by-Q grids. The benchmark will then run every possible combination

of the parameters that were set.

5.6.1 HPL Setup

We configured the HPL.dat accordingly to define a problem suited to our testbed. Since we have 256MB remote memory available on a remote node, we defined a two-dimensional matrix of order 11585. This corresponds to 1GB problem size which means that one quarter of the array can fit to remote memory. We defined a 1x4 P-by-Q grid since our computer node contains 4 cores, so every index of the grid would be calculated by a different core, hence every core has a data-set about 256MB data. We performed 5 runs with this configuration. For the first run we keep all of our matrix to local memory. For the second run, one of the four cores sends its whole data-set to remote memory. For the third run two of the four cores send half of their data-set to remote memory. For the fourth run, all four cores send one quarter of their data-set to remote memory. Finally, for the last run we migrated parts of different size of the data-set for each core. We migrated 128MB for the first core, 64MB for the second core and 32MB for the third and fourth core Right before initialization we call pmm_init to register the process to the daemon, then we call pmm_send_buffer from the required MPI processes, to send to the daemon the memory that we want it to migrate when RAM drops below the specified threshold. Right before the benchmark terminates, we call pmm_cleanup to deregister the process from the daemon. Figure 5.19 below illustrates the way we split data to remote memory in each configuration

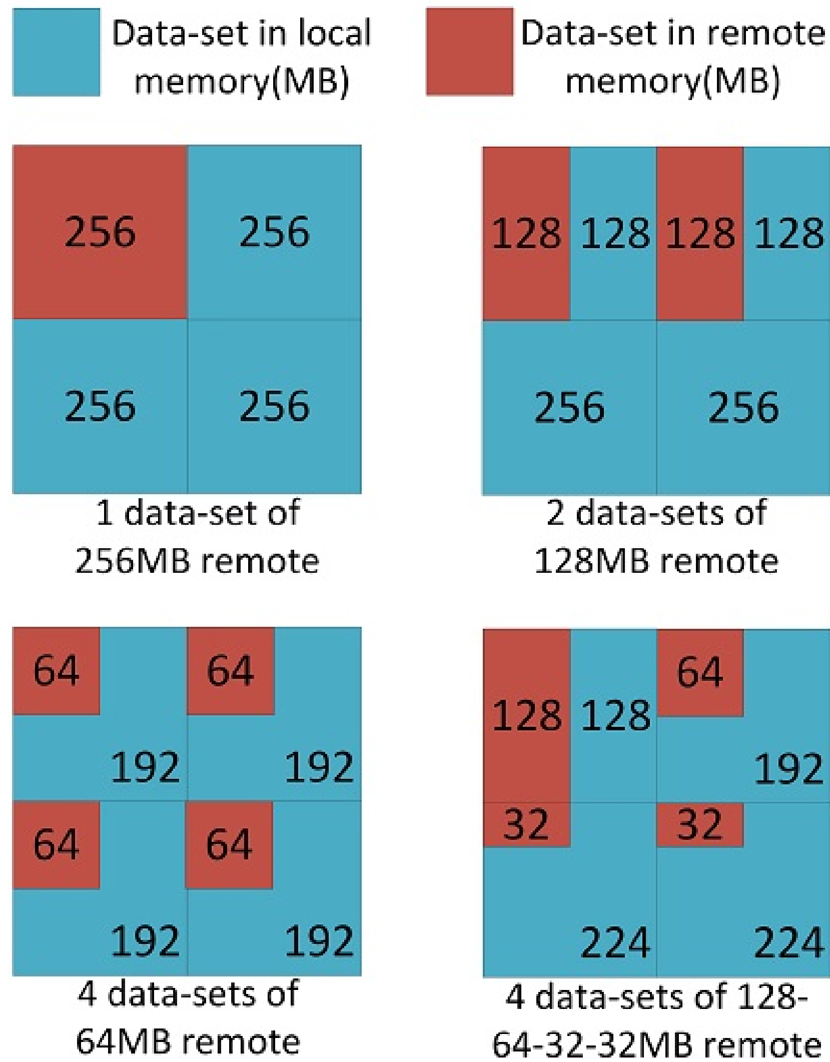

Figure 5.19: Hpl data split between local and remote memory on the 4 configurations.

5.6.2 HPL Results

Figure 5.20 below depicts the slowdown of the three remote configurations compared to baseline local configuration. The remote configuration with the biggest slowdown of X3.46 is the one where we migrate the whole data-set of 256MB from only one core. As we split the migration among the data-sets of processes we experience a lower slowdown. The configuration where we migrate a data-set of 128MB from two processes gets a better performance than the previous remote configuration with a X2.33 slowdown compared to the baseline, while migrating a data-set of 64MB from all four processes gets the best performance between the

remote configurations with only a X1.8 slowdown compared to the baseline. The run with the different migrated size for each core gets a X2.27 slowdown which is close to the performance of the second run.

By deduction, a process that has its data-set in remote memory, takes longer to calculate its share than a process with the data-set in local memory because of the interconnection latency. We assume that in the first remote configuration, the one process with the remote data-set of 256MB needs a lot more time to perform its computations than the other three processes which have their data in local memory, so the other three processes are waiting the first process to broadcast its data to them most of the time. In the last case where we split the migrated data among the processes, all four processes take about the same time to calculate their share and since each data-set has 64MB of remote data and 196MB of local data, they finish faster than the first process of the first remote configuration. The performance loss of the last run is expected, as there is a core with a migrated data-set of 128MB which is the largest among the other three cores. We deduce that it takes the same time to perform operations as the two cores in the second run which have the same migrated size, and the other three cores are waiting for this core to broadcast and receive data.

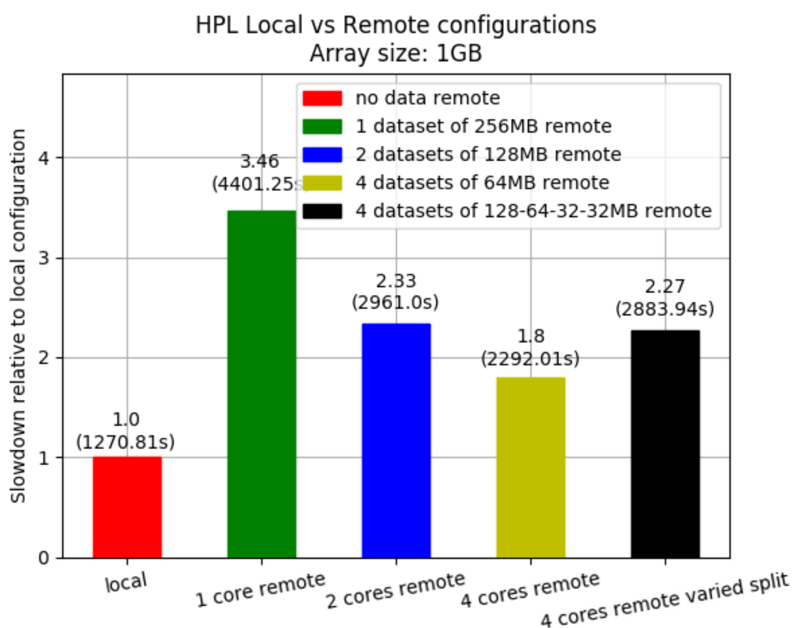

Figure 5.20: HPL benchmark n=11585, PxQ=1x4, 4 processes run. Slowdown of remote configurations compared to local configuration.

5.7 Benchmark: CSparse

CSparse [6] is a C library which implements a number of direct methods for sparse linear systems. CSparse is using the compressed sparse column Data Structure [9] to store the values of a matrix. The main advantage of using such a data structure is that the data-set requires to only contain the non zero(nnz) values of the matrix and the extents of columns, and row indices for each value, thus with greater sparsity, we save more memory in percentage, compared to storing the actual matrix with the zero values. The CSparse library is contained in SuitSparse [7], which is a suite for Sparse Matrix algorithms in Matlab.

Inside the source of the C library, three interesting demo applications can be found that solve several basic matrix equations. We used the second demo file of the CSparse suite which receives a file as input that contains the non-zero values of the sparse matrix as well as its indices, then performs several different algorithms which solve the linear system $x = A\backslash b$ where A is the input matrix and b is a vector generated at run-time by the demo. We picked 2 matrices that can be solved using the QR decomposition by calculating the permutation matrix $P = A'A$, the LU decomposition by calculating the permutation matrices $P = A'A$, $P = S'S$ and the Cholesky decomposition using the permutation matrices $P = A' + A$ where S is a symbolic analysis data structure of the QR or the LU decomposition.

5.7.1 CSparse Setup

To evaluate our system on the CSparse demo suite, we downloaded two matrices from the SuitSparse Matrix Collection [8],which contains a huge database of sparse matrices that can be used. The first matrix is called "ct20 engine block stiffness matrix" of the boeing group. It is a symmetric square matrix of order 52329 with 2600295 non zero values and is considered a structural problem. The second matrix is called "nd3k" of the ND group. It is a symmetric square matrix of order 9000 with 3279690 non zero values and is considered a problem of the 2D/3D category. Figure 5.21 below illustrates the composition of the two matrices.

CSparse was a little more complicated to tweak compared to the previous benchmarks. Right before initialization we call pmm_init to register the process to the daemon. For every one of the five algorithms that CSparse uses to solve the $x = A\backslash b$ system, it allocates memory of the data-set during the algorithm, and frees all allocated memory at the end of the algorithm. Not only that, but because of the data structures that CSparse is using, the allocation of the data-set is not a single big chunk of data, but several small chunks that take place in different timestamps during the algorithm. We analyzed the code and found the lines where the allocation of those chunks happens. We then called pmm_send_buffer after every allocation. More specifically, we sent to the daemon the nnz of the A, P, L, U matrices as well as their indices. We let the page migration system decide which pages to migrate based on LRU. Since the demo frees the memory at the end of the algorithm and allocates new memory at the beginning of the next

algorithm, we called pmm_release_buffer before each free call. At the end we call pmm_cleanup to deregister the process from the daemon.

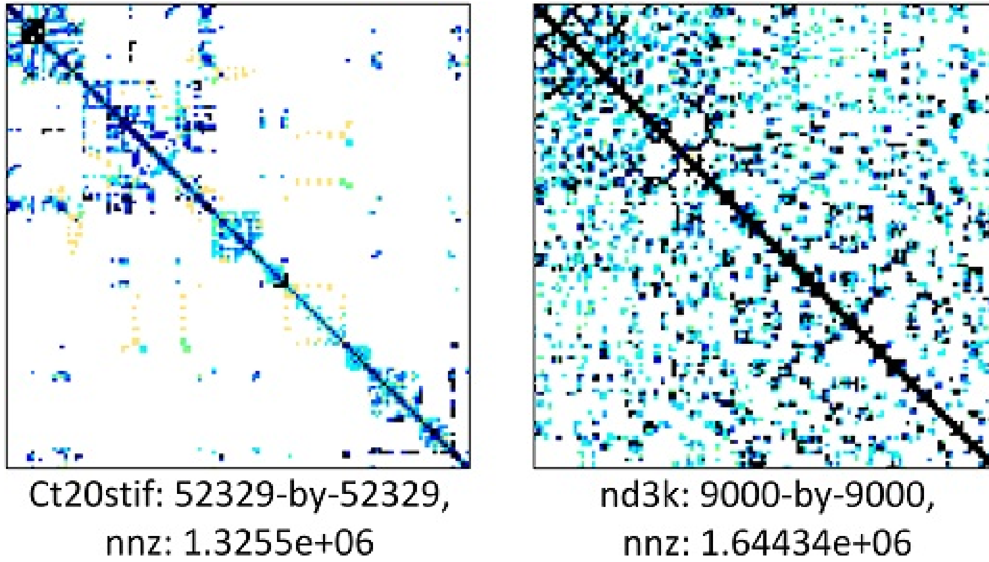

Figure 5.21: ct20stif and nd3k matrices composition.

5.7.2 CSparse Results

Figures 5.22 and 5.23 below show the running time of the five algorithms in the demo for the two matrices that were used. What we understand from the graphs is that the matrix composition combined with the algorithm that is used, play an important role in cache locality on operations on compressed matrix data structures. Although ct20stif is a matrix of higher order than nd3k, because it has less nnz, it takes less time than nd3k to finish each test. For the local configuration, the QR decomposition finishes in 1578.40 seconds for the ct20stif matrix while it takes 5874.90 seconds for the nd3k. In the $QR : P = A'A$ decomposition, the remote configuration for the ct20stif has a X1.2 slowdown compared to to the local configuration while the remote configuration for the nd3k has a X1.7 slowdown compared to the local configuration. In contrast, in the $LU : P = A'A$ decomposition, the remote configuration for the nd3k performs better than ct20stiff, with a X1.87 slowdown for the former compared to a X2.17 slowdown for the latter. For the rest of the algorithms we observe that slowdowns range from X1.01 to X1.04. Again we confirm that for the last three algorithms due to temporal locality we take advantage of remote memory with the cost of only 1% to 4% performance drop.

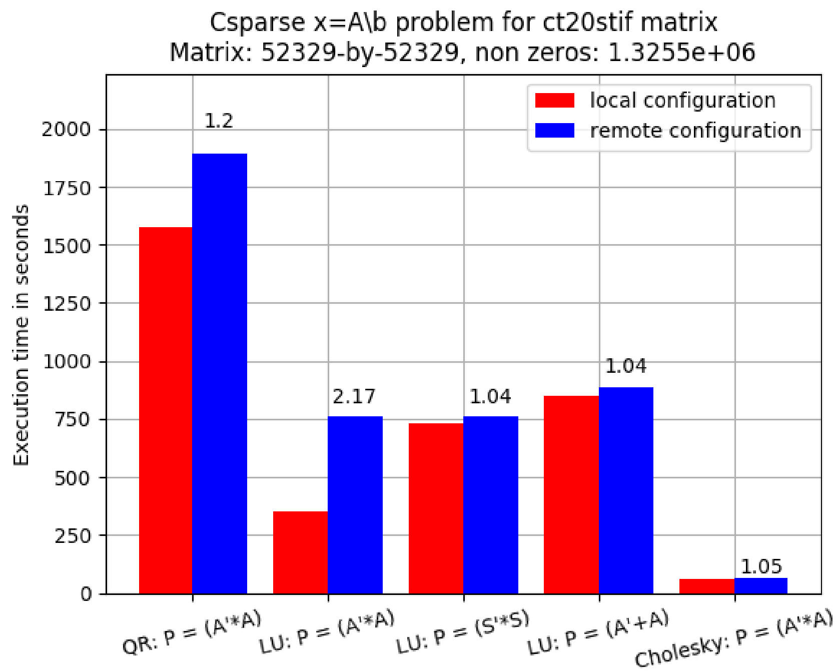

Figure 5.22: CSparse demo 2 on ct20stif matrix. Slowdown of remote configuration compared to local configuration.

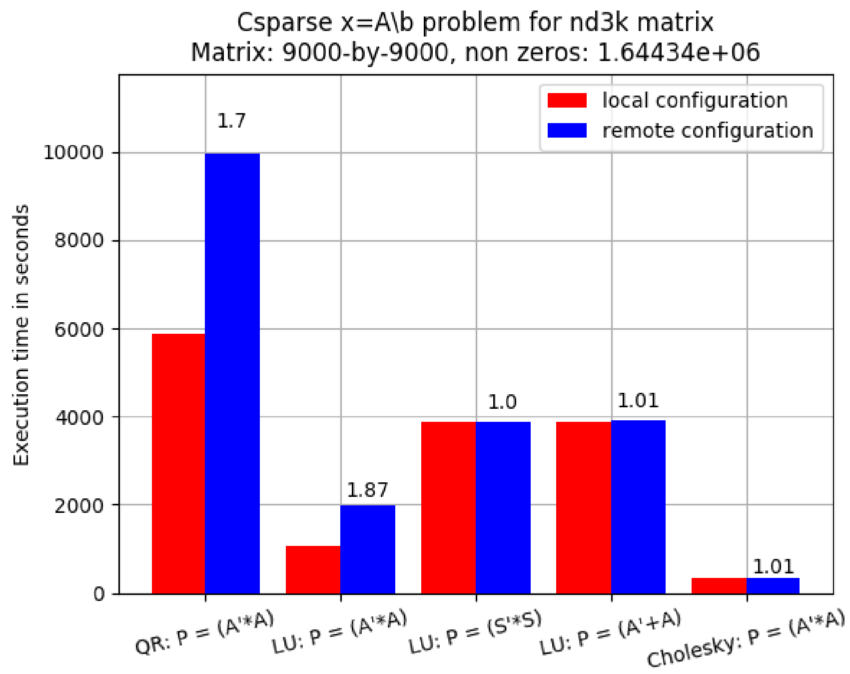

Figure 5.23: CSparse demo 2 on nd3k matrix. Slowdown of remote configuration compared to local configuration.

Chapter 6

Conclusion

6.1 Summary

The constant evolution of HPC clusters renders the increase of memory capacity, a necessity. However, an individual computer node has limited capabilities for memory expansion. The utilization of unused memory from nearby computer nodes in the same cluster, is an effective way to increase the available memory for individual nodes, bypassing their memory capacity constraints, without the requirement for additional resources. Because of that, there have been many research projects that try to exploit remote memory allocation.

The page migration system, is an implementation that monitors the main memory usage of computer nodes in a cluster, and provides a node with a high memory workload, additional remote memory from a remote node in the same cluster. The benefits of the page migration system offer transparency with remote read and write operations, and data are cacheable which exploits temporal locality for better system performance. A process running on the system, only needs to inform the page migration daemon about the memory that it has allocated through a simplistic API. Similarly, the API offers extra utilities so that the daemon has a different behaviour on the calling process's pages. The daemon runs a fairness algorithm regarding the page selection. The algorithm takes into account the LRU page policy, and a priority value among processes. Processes with lower priority will have their pages migrated remotely earlier and/or returned locally later than processes with higher priority, when and if it is required.

As we have demonstrated in our evaluation chapter, moving data to remote memory hinders the process's performance due to the remote operations taking longer to complete. However, in cases where we can take advantage of cache locality, or migrate data that are less frequently used, performance loss is significantly lowered, while we still get the benefits of increased memory capacity.

6.2 Future Work

The current implementation presented in this thesis is a prototype version aiming to demonstrate the benefits of the PMS on clusters with fluctuated memory workloads. We currently work on upgrading the system to add more functionality and investigate on several possible additions which are listed below:

1. Port work to the latest prototype QFDB. Each QFDB node contains four Zynq UltraScale+ MPSoCs instead of one and 16GB of RAM split equally among the four MPSoCs. There are also differences in the hardware design of the PL. For example there is no part of the physical memory that is considered reserved by the device tree. We plan on finding a way to isolate memory from the OS, unmap it and then make it available for other nodes to hotplug it. This will require a more complex implementation, but will allow us to share dynamic sizes of memory with the other nodes.

2. Implement memory hot-remove. The big challenge with memory hot-removal is the fact that we need to ensure that there is no allocated memory on the region when the procedure takes place. Since we managed to add the region in a memory zone and isolate it from the rest of the system, we have full control of allocated memory inside the region as well as the ability to migrate pages to another region if required.

3. Add communication between daemons on the computer nodes of the cluster. This will allow the nodes to share memory between each other and avoid collisions in case multiple nodes try to hot-plug the same memory region. The mailbox driver which is an API already implemented for our distributed system that provides communication between user-space processes, can be used for that purpose.

4. Implement complete transparency on the Page Migration System. We aim to remove the library that allows for inter-process communication between the daemon and the other processes. Instead, we want add the daemon a functionality that allows it to dynamically search for allocated data among all the processes that run on the system. The daemon should be able to distinguish startup processes of high importance for the Operating System from other user-processes and choose to move data from the latter. The daemon should also identify pages of processes and refrain from moving critical data like instruction memory.

Bibliography

[1] J. Dongarra A. Cleary A. Petitet, R. C. Whaley. "hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers". https://www.netlib.org/benchmark/hpl/.

[2] Scott Branden" "Andrea Reale, Maciej Bielski. "memory hotplug support for arm64 - complete patchset". https://lwn.net/Articles/719710/.

[3] ARM. *ARM Cortex-A53 MPCore Processor: Technical Reference Manual*, r0p3 edition, April 2014. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500e/DDI0500E_cortex_a53_r0p3_trm.pdf.

[4] Raghu Bharadwaj. Memory management and allocators. In *Mastering Linux Kernel Development*, chapter 4. Packt Publishing Ltd, 1st edition, October 2017.

[5] Marco Cesati Daniel P. Bovet. Memory management. In *Understanding the Linux Kernel*, chapter 8. O'Reilly, 3rd edition, November 2005.

[6] Timothy A. Davis. "csparse: A concise sparse matrix package in c". https://people.sc.fsu.edu/~jburkardt/c_src/csparse/csparse.html.

[7] Timothy A. Davis. "suitsparce. a suite of sparse matrix software". http://faculty.cse.tamu.edu/davis/suitesparse.html.

[8] Timothy A. Davis. "suitsparce matrix collection". https://sparse.tamu.edu/.

[9] Jack Dongarra. "describing physical memory". http://netlib.org/linalg/html_templates/node92.html.

[10] http://exanest.eu/.

[11] Mel Gorman. "compressed column storage (ccs)". https://www.kernel.org/doc/gorman/html/understand/understand005.html.

[12] Dr. Ryutaro Himeno. "himeno benchmark — isc, riken". http://accc.riken.jp/en/supercom/documents/himenobmt/.

[13] Aasheesh Kolli Andreas Nowatzyk Jayneel Gandhi Onur Mutlu Pratap Subrahmanyam" "Irina Calciu, Ivan Puddu. "project pberry: Fpga acceleration for remote memory". https://dlp5.acm.org/citation.cfm?id=3321424.

[14] D. Pnevmatikatos G. Zervas D. Theodoropoulos I. Koutsopoulos K. Hasharoni D. Raho C. Pinto F. Espina S. Lopez-Buedo Q. Chen§ M. Nemirovsky D. Roca H. Klosx T. Berends" "K. Katrinis, D. Syrivelis. "dredbox: Disaggregated data center in a box". http://www.dredbox.eu/.

[15] Shoaib Kamil. "stencilprobe: A microbenchmark for stencil applications". http://people.csail.mit.edu/skamil/projects/stencilprobe/.

[16] The kernel development community. "page migration". https://www.kernel.org/doc/html/latest/vm/page_migration.html.

[17] The kernel development community. "memory hotplug", 2018. https://www.kernel.org/doc/html/latest/admin-guide/mm/memory-hotplug.html.

[18] Robert Love. Memory management. In *Linux Kernel Development*, chapter 12. Addison-Wesley Professional, 3rd edition, June 2010.

[19] Irina Calciu Xavier Deguillard Jayneel Gandhi Stanko Novakovic Arun Ramanathan Pratap Subrahmanyam Lalith Suresh Kiran Tati Rajesh Venkatasubramanian-Michael Wei" "Marcos Aguilera, Nadav Amit. "remote regions: a simple abstraction for remote memory". https://research.vmware.com/publications/remote-regions-a-simple-abstraction-for-remote-memory.

[20] Stanko Novakovic Sharad Singhal" "Marcos K. Aguilera, Kimberly Keeton. "designing far memory data structures: Think outside the box". https://www.microsoft.com/en-us/research/publication/designing-far-memory-data-structures-think-outside-the-box/.

[21] John D. McCalpin. "stream: Sustainable memory bandwidth in high performance computers". https://www.cs.virginia.edu/stream/.

[22] Larry McVoy. "lmbench - tools for performance analysis". http://lmbench.sourceforge.net/.

[23] Yilun Chen "Yizhou Shan, Yutong Huang and Yiying Zhang". "legoos: A disseminated, distributed os for hardware resource disaggregation". https://www.usenix.org/conference/osdi18/presentation/shan.