ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Memory Management in Hybrid DRAM/NVM Systems using LSTMs

Κωνσταντίνος Γ. Σταυρακάκης
Α.Μ. : 03116155

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα
Φεβρουάριος 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Memory Management in Hybrid DRAM/NVM Systems using LSTMs

Κωνσταντίνος Γ. Σταυρακάκης
Α.Μ. : 03116155

Επιβλέπων **:** Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Τριμελής Επιτροπή Εξέτασης

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
....	....	....
Δημήτριος Σούντρης Καθηγητής ΕΜΠ	Παναγιώτης Τσανάκας Καθηγητής ΕΜΠ	Γεώργιος Γκούμας Αναπληρωτής Καθηγητής ΕΜΠ

Ημερομηνία Εξέτασης:
18 Φεβρουαρίου 2022

Copyright © - All rights reserved Σταυρακάκης Κωνσταντίνος, 2022.
Με επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ΄ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

(Υπογραφή)

....

Σταυρακάκης Κωνσταντίνος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
©2022 - All rights reserved.

Περίληψη

Είναι πλέον ευρεία η ενσωμάτωση των τεχνολογιών ετερογενών μνημών στα σύγχρονα υπολογιστικά συστήματα προκειμένου να αντιμετωπιστεί η συνεχώς αυξανόμενη ανάγκη των αναδυόμενων εφαρμογών για όλο και μεγαλύτερο μέγεθος κύριας μνήμης. Αυτές οι νέες τεχνολογίες παρουσιάζουν αρκετές διαφορές μεταξύ τους όσον αφορά το μέγεθος, την καθυστέρηση πρόσβασης αλλά και το εύρος ζώνης. Αυτή η γενικότερη ετερογένεια των νέων συστημάτων καθώς και οι ίδιες οι ιδιαιτερότητες των νέων εφαρμογών καθιστούν ανεπαρκείς τις σύγχρονες πρακτικές διαχείρισης μνήμης.

Σε αυτή τη διπλωματική σχεδιάσαμε και αξιολογήσαμε έναν δρομολογητή, ο οποιος έξυπνα τοποθετεί τα δεδομένα των εφαρμογών, σε επίπεδο Σελίδων Μνήμης, στα διάφορα στοιχεία μνήμης του συστήματος χρησιμοποιώντας Τεχνητά Νευρωνικά Δίκτυα. Ο δρομολογητής που προτείνουμε συνδυάζει τη χρήση LSTM Δικτύων με τις υπάρχουσες μεθόδους διαβάθμισης των δεδομένων που στηρίζονται στο *Ιστορικό* των δεδομένων αυτών. Ο δρομολογητής μας έξυπνα χρησιμοποιεί μηχανική μάθηση μόνο για ένα υποσύνολο σελίδων, οι οποίες αν μετακινηθούν στο σωστό χρόνο θα επιτευχθεί σημαντική αύξηση της απόδοσης. Ακόμα χρησιμοποιώντας την τεχνική K-Means για τον χωρισμό του Πεδίου Διευθύνσεων σε Συστάδες, ο χρονοδρομολογητής ενισχύει το πλήθος των πληροφοριών με βάση το οποίο θα παίρνονται οι αποφάσεις για την απομάκρυνση δεδομένων απο τη DRAM. Έτσι βλέπουμε ότι επιτυγχάνεται κατά μέσο όρο μια αύξηση της απόδοσης περίπου 10% σύμφωνα με τη διαδικασία αξιολόγησης που ακολουθήσαμε. Επίσης, επιβεβαιώσαμε ότι ο προτεινόμενος δρομολογητής μπορεί να γεφυρώσει ικανοποιητικά το χάσμα απόδοσης μεταξύ των σύγχρονων λύσεων και ενός δρομολογητήμάντη με a priori γνώση της συμπεριφοράς των δεδομένων. Και τέλος, αξιολογώντας ενεργειακά τον δρομολογητή που σχεδιάσαμε προκύπτει ότι θα μπορούσε να είναι μια αξιόλογη πρόταση για συστήματα τα οποία ειναι σχεδιαστικά προσανατολισμένα στη χαμηλή κατανάλωση ενέργειας.

Λέξεις Κλειδιά — Συστήματα Ετερογενών Μνημών, Μηχανική Μάθηση, Νευρωνικά Δίκτυα, LSTM, K-Means, NVM, Επαναληπτικά Νευρωνικά Δίκτυα, Δρομολόγηση Σελίδων, DRAM

Abstract

Heterogeneous memory technologies have been widely used in effort to address the ever-increasing demands of modern applications for larger main memory capacity. The new technologies showcase vastly greater differences in terms of capacity, latencies and bandwidth. This heterogeneity along with the the greater irregularity of emerging workloads, render state-of-the-art memory management solutions insufficient; thus calling for more intelligent methods.

In this diploma Thesis, we design and evaluate a scheduler which intelligently places application data, on a Page granularity, across hybrid memory components using Artificial Neural Networks. The proposed Scheduler combines intelligent page placement decisions leveraging LSTM networks with existing history-based data tiering methods. The scheduler focuses the machine learning on a page subset whose timely movement will reveal most application performance improvement, while being mindful of computation resources. K-Means address space clustering is also utilized to augment the eviction policy used by the proposed scheduler in order to provide application performance boost. That boost is on average 10% according to our evaluation process. Our performance evaluation also indicates that the proposed Scheduler significantly reduces the performance gap between existing solutions and an oracle scheduler with a priori knowledge of the page access patterns, while being a potential candidate for designing low-power oriented Hybrid Memory Systems as well.

Keywords — Heterogeneous Memory Systems, Machine Learning, Long Short Term Memory Networks, K-Means, Non Volatile Memory, Recurrent Neural Networks, Page Scheduling, DRAM

Ευχαριστίες

Αρχικά, ευχαριστώ θερμά τον καθηγητή μου, κ. Δημήτριο Σούντρη, για την εμπιστοσύνη που μου έδειξε από την πρώτη μας επικοινωνία μέχρι και σήμερα. Επίσης, ένα μεγάλο ευχαριστώ στους υποψήφιους διδάκτορες κ.κ. Μασούρο και Κατσαραγάκη για την εξαιρετική συνεργασία μας και την ακούραστη καθοδήγηση που μου προσέφεραν. Τέλος, ευχαριστώ απο καρδιάς τους γονείς μου, τον αδερφό μου, και τους φίλους μου για την απεριόριστη στήριξη και ώθηση που μου έδωσαν καθ΄ όλη την ακαδημαϊκή μου πορεία.

Contents

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Contents 11

Εκτεταμένη Περίληψη 15
Εισαγωγή 15
Persistent Memory και Διαχείριση Μνήμης 17
Χρονοδρομολόγηση και Μηχανική Μάθηση 19
Σχεδίαση 22
Υλοποίηση 24
Αξιολόγηση 27
Σύνοψη και Μελλοντική Δουλειά 29

1 Introduction 31
Thesis Topic 31
Motivation 32
Approach and Contributions 33
Thesis Overview 34

2 Related Work 35
Hardware Solutions 35
Software Solutions 35
Machine Learning Solutions 36

3 Persistent Memory & Memory Management 39
Persistent Memory 39
Page Migration 42
Page Migration across NUMA-nodes 43
Page Migration in Hybrid Memory Systems 44
Machine Intelligence based solution 46

Page Migration Challenges 47
Implementation Overhead 47
Data Retrieval 48
Page Movement 49

4 Machine Learning & Deep Neural Networks 51
Machine Learning Background 51
Types of Machine Learning 51
Artificial Neural Networks 53
Recurrent Neural Networks 60
LSTMs 61
Page Scheduling as a Machine-Learning problem 63
Reinforcement Learning Approach 63
Recurrent Neural Network Approach 63
Neural Network Input 65
Deltas Prediction 65
Per Page Prediction Approach 68

5 Proposed Page Scheduler Architecture 71
Critical Metrics 71
Page Scheduler Overview 74
Page Scheduler Components 76
Page Selector 76
Access Count Predictors 79
DRAM Eviction Policy 80

6 Technical Implementation 83
Benchmark workloads 83
Collecting Memory Access Traces 85
Hybrid Memory System Simulator 86
Recurrent Neural Networks Details 86
Neural Network Input 86
Neural Network Configuration 89

7 Experimental Evaluation 91
RNN Prediction Accuracy 91
Application Performance 93
Eviction Policy 95
Energy Consumption 96

8 **Conclusions** **105**
Thesis Summary 105
Future Work 106

Εκτεταμένη Περίληψη

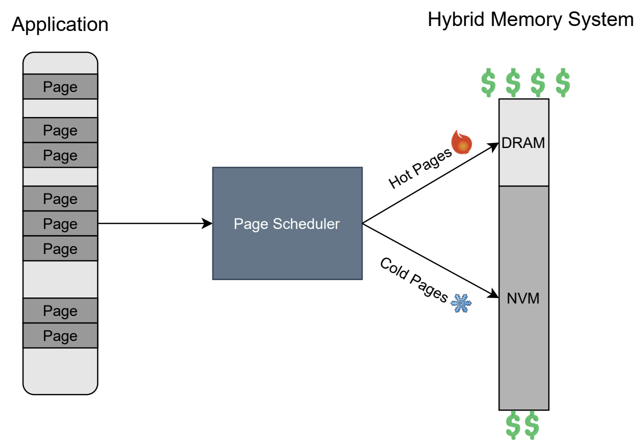
Εισαγωγή

Τα τελευταία χρόνια παρατηρείται εκτεταμένη διείσδυση της Μηχανικής Μάθησης σε κάθε κλάδο. Η Μηχανική μάθηση αποτελεί παράγοντα καινοτομίας σε ένα μεγάλο φάσμα εφαρμογών που περιλαμβάνει απο εμπορικά προϊόντα έως και ιατρικές εφαρμογές. Η ευρύτητα αυτού του φάσματος, σε συνδυασμό με το γεγονός ότι η απόδοση των μοντέλων που αναπτύσσονται με τεχνικές μηχανικής μάθησης είναι αρκετά καλή μοντελοποιώντας εφαρμογές που παραδοσιακά θα ήταν αρκετά περίπλοκο να μοντελοποιηθούν, ωθούν στην ραγδαία ανάπτυξη της. Ταυτόχρονα, στον κλάδο της αρχιτεκτονικής υπολογιστών η προόδος που προβλέπεται απο το νόμο του Μοορε φαίνεται ότι σταδιακά παύει να ακολουθεί το εκθετικό μοτίβο αύξησης που ακολουθούσε έως τώρα, ενώ το χάσμα επιδόσεων μεταξύ μνήμης και επεξεργαστή δεν εχει γεφυρωθεί ακόμα. Οι δύο αυτές τάσεις ,δηλαδή η εξέλιξη της Μηχανικής Μάθησης και τα ίδια τα προβλήματα που υπάρχουν στην Αρχιτεκτονικη Υπολογιστων, ωθούν προς μια συνδυαστική αξιοποιήση μεθόδων και τεχνικών, έτσι ώστε η μηχανική μάθηση να υποστηρίζεται αλλα κυρίως να υποστηρίζει την αρχιτεκτονική.

Είναι γνωστό ότι τα σύγχρονα υπολογιστικά συστήματα σχεδιάζονται χρησιμοποιώντας ετερογενή συστατικά μνήμης. Αυτές οι μνήμες συχνά είτε εξυπηρετούν στην αύξηση της χωρητικότητας της κύριας μνήμης, δηλαδή σαν επέκταση της DRAM, είτε αξιοποιούνται ως κρυφές-μνήμες (caches) της κύριας μνήμης. Αυτά τα υβριδικά συστήματα μνήμης συνοδεύονται εκ φύσεως με κάποιους σχεδιαστικούς συμβιβασμούς. Συνήθως η μακρύτερη στην ιεραρχία μνήμη, δηλαδή μνήμη που βρίσκεται πιο μακρία απο την Επεξεργαστική Μονάδα, έχει μεγαλύτερη χωρητικότητα αποθήκευσης αλλά ταυτόχρονα έχει και μεγαλύτερη καθυστέρηση πρόσβασης (latency) και μειωμένο εύρος ζώνης (bandwidth).

Στη συγκεκριμένη διπλωματική εργασία θα περιοριστούμε σε υπολογιστικά συστήματα τα οποία αξιοποιούν την Persistent Memory ως επέκταση της κύριας μνήμης. Για το σχεδιασμό αυτών των συστημάτων είναι ιδιαίτερα σημαντικό να ληφθούν υπόψιν το μεγαλύτερο latency και το μειωμένο bandwidth που παρουσίαζει αυτή η μνήμη σε σχέση με τη DRAM. Στη συγκεκριμένη περίπτωση αυτό που μας ενδιαφέρει είναι η αποδοτική υλοποιήση ενός Χρονοδρομολογητή Σελίδων Μνήμης (Page Scheduler), δηλαδή η αποδοτική υλοποίηση της μονάδας που αναλαμβάνει

τη διαχείριση της μνήμης του λειτουργικού συστήματος αλλά και των εν εκτελέση προγραμμάτων. Ο χρονοδρομολογητής σελίδων θα είναι υπεύθυνος για τη μεταφορά σελίδων μνήμης απο και προς τα διάφορα ετερογενή συστατικά μνήμης που απαρτίζουν το σύστημα μας. Κύριο σκοπό θα έχει οι σελίδες μνήμης που θεωρούνται *hot*, δηλαδή προσπελαύνονται συχνά, να βρίσκονται στα υψηλής απόδοσης στοιχεία μνήμης που διαθέτει το σύστημα μας, δηλαδή στη DRAM, ενώ λιγότερο σημαντικές σελίδες μνήμης *cold* να βρίσκονται στην Persistent Memory. Μια διαγραμματική επεξέγηση του πρόβληματος που προσπαθούμε να αντιμετωπίσουμε σε αυτή τη διπλωματική φαίνεται στο διάγραμμα 1.



Εικόνα 1: Χρονοδρομολογητής Σελίδων σε ένα Hybrid Memory System

Κίνητρο και συναφείς προσεγγίσεις

Πολλοί ερευνητές έως τώρα έχουν αποπειραθεί να δώσουν μια λύση στο πρόβλημα της κατηγοριοποίησης των σελίδων μνήμης και στην κατάλληλη τοποθέτηση τους στα διάφορα ετερογενή στοιχεία μνήμης. Πρόκειται σίγουρα για μία δύσκολη διαδικασία καθώς πρέπει να ληφθούν υπόψιν και το μοτίβο προσπέλασης μνήμης που ακολουθεί μία εφαρμογή όσο και οι παράμετροι εκτέλεσης της εφαρμογής (μέγεθος του input, strong/weak scaling κλπ). Οι περισσότεροι ερευνητές έχουν προτείνει λύσεις στο παραπάνω πρόβλημα που μπορούν να ενσωματωθούν στο επίπεδο του hardware, των Compilers, του Λειτουργικού Συστήματος και του περιβάλλοντος εκτέλεσης [1, 2, 3, 4, 5, 6, 7]. Συχνά οι ερευνητές σε αυτές τις προσεγγίσεις χρησιμοποιούν πληροφορίες που σχετίζονται μονάχα με το ιστορικό προσπέλασης των σελίδων μνήμης. Συγκεκριμένα οι σύγχρονες τεχνικές που χρησιμοποιούνται στη δυναμική διαχείριση σελίδων σε επίπεδο συστήματος χρησιμοποιούν την πρόσφατη παρατηρούμενη συμπεριφορά των σελίδων προκειμένου να παρθούν αποφάσεις την μελλοντική τοποθέτηση τους.

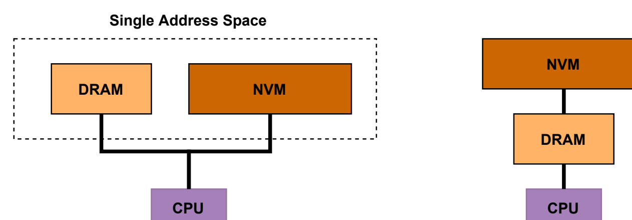
Σε αυτή τη διπλωματική εργασία θα ακολουθηθεί η συλλογιστική πορεία των άρθρων *Learning Memory Access Patterns* [8] και *Kleio: A hybrid Memory Page*

Scheduler [9]. Σκοπός μας είναι η μελέτη και κατασκευή ενός Χρονοδομολογητή Σελίδων Μνήμης χρησιμοποιώντας Μηχανική Μάθηση, ο οποίος θα επιτυγχάνει καλύτερη απόδοση απο τις σύγχρονες μεθόδους οι οποίες στηρίζονται αποκλειστικά στην ιστορική παρατήρηση προσβάσεων στη μνήμη των εφαρμογών. Θα προσπαθήσουμε να απαντήσουμε σε ερωτήσεις που αφορούν το πως θα πετύχουμε μια αποδοτική λύση, δηλαδή πως θα καταφέρουμε να τοποθετήσουμε όσο πιο ιδανικά γίνεται τα σωστά δεδομένα σε επίπεδο σελίδων (**4 KiB** [10]) στα *σωστά* στοιχεία μνήμης. Ενώ ταυτόχρονα αναζητούμε και μια εφικτή λύση, χρησιμοποιώντας περιορισμένους επεξεργαστικούς πόρους για την κατασκευή των μοντέλων μηχανικής μάθησης που θα χρειαστούμε.

Persistent Memory και Διαχείριση Μνήμης

H Persistent memory (NVMM) είναι μια σχετικά νεο-αφιχθήσα στον χώρο των υπολογιστικών συστημάτων μη πτητική μνήμη, η οποία προσφέρει διευθυνσιοδότηση σε επίπεδο byte και είναι άμεσα προσπελάσιμη από τον επεξεργαστή όπως η DRAM. Διάφορες τεχνολογίες μπορούν να ενταχθούν στην κατηγορία της Persistent Memory όπως η Phase Change Memory (PCM) [11], η Spin-Transfer Torque RAM [12], και η 3D-XPoint. H Persistent Memory έχει υψηλή πυκνότητα και χαμηλό κόστος ανά bit, ενώ παράλληλα η καθυστέρηση πρόσβασης (access latency) είναι στην ίδια τάξη μεγέθους με τη DRAM, αισθητά όμως μεγαλύτερη. Ιδιαίτερο χαρακτηριστικό της Persistent Memory είναι η ασυμμετρία στην καθυστέρηση πρόσβασης μεταξύ των αιτημάτων διαβάσματος και γραψίματος. Τα αιτήματα γραψίματος είναι αισθητά πιο αργά και παράλληλα φαίνεται να είναι πεπερασμένα και περιοριστικά για το χρόνο ζωής της μνήμης (finite write endurance). Ακόμα αξιοσημείωτο χαρακτηριστικό της είναι και η μικρή κατανάλωση ενέργειας όταν βρίσκεται σε αδράνεια συγκριτικά με την DRAM, πράγμα που την καθιστά καλή επιλογή για τον σχεδιασμό low-power συστημάτων.

Η μνήμη αυτή βρίσκεται στο ίδιο επίπεδο ιεραρχίας με την DRAM και χρησιμοποιείται είτε ως επέκταση της κύριας μνήμης είτε ως κρυφή μνήμη της DRAM όπως φαίνεται στην εικόνα 2. Εμείς θα ασχοληθούμε μόνο με το πρώτο σενάριο.



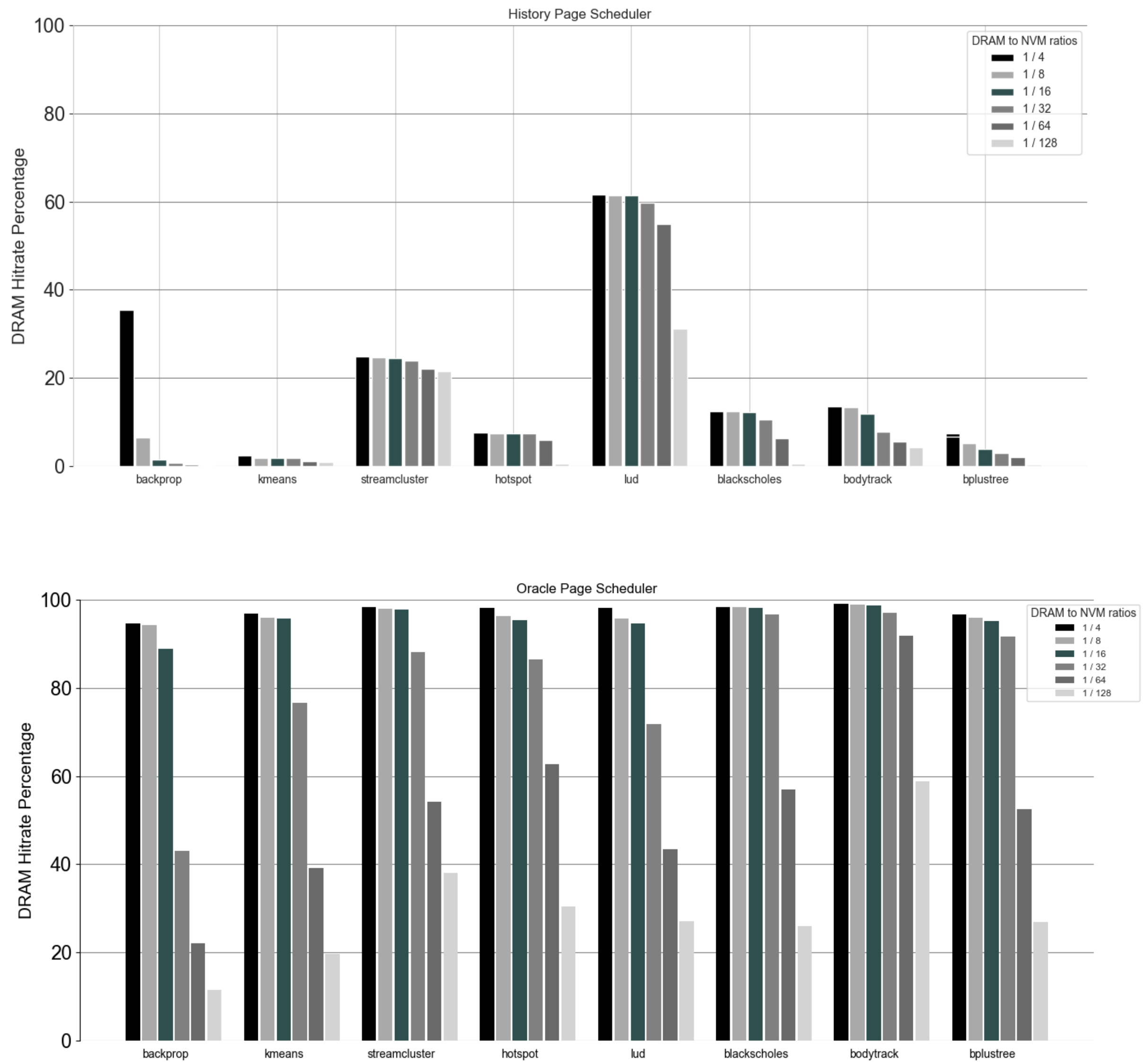
Εικόνα 2: NVM σαν επέκταση κύριας μνήμης και σαν κρυφή μνήμη της DRAM

Πρακτικά στο πρώτο σενάριο το σύστημα *βλέπει* ένα διευρυμένο Address Space το οποίο αποτελείται, σε hardware και από DRAM αλλά και απο NVM. Ωστόσο

αυτά τα δύο είδη μνήμης δεν έχουν τα ίδια τεχνικά χαρακτηριστικά. Για αυτό το λόγο η διαχείριση μνήμης σε αυτά τα συστήματα είναι ιδιαιτέρη, και παρουσίαζει ιδιαίτερη ομοιότητα με τη διαχείριση μνήμης που απαιτείται στα συγχρονα υπολογιστικά συστήματα που διαθέτουν NUMA nodes. Αυτό που επιθυμούμε σε αυτά τα συστήματα είναι κατα τη διάρκεια της εκτέλεσης των προγραμμάτων να μπορούν να παρθούν αποφάσεις για τη μεταφορά Σελίδων Μνήμης απο και προς τα διαφορετικά συστατικά μνήμης, με σκόπο να εκμεταλλευτούμε το γεγονός ότι η DRAM είναι αρκετά ταχύτερη από την NVM ώστε να αυξηθεί η συνολική απόδοση του συστήματος. Αυτό που προτείνεται στη βιβλιογραφία ως λύση μέχρι αυτή τη στιγμή είναι η χρήση ενός Χρονοδρομολογητή ο οποίος θα στηρίζεται αποκλειστικά στο *Ιστορικό* των σελίδων [13] (History Page Scheduler). Σύμφώνα με αυτήν την προσέγγιση παρατηρείται η συμπεριφορά της σελίδας στο πρόσφατο παρελθόν (π.χ. στα τελευταία 5 δευτερόλεπτα ή στην τελευταία *Περίοδο Χρονοδρομολογησης*), και με βάση αυτήν την συμπεριφορά αποφασίζεται σε ποιο συστατικό μνήμης πρέπει να τοποθετηθεί η σελίδα. Η υλοποίηση αυτής της προσέγγισης σε επίπεδο συστήματος είναι σχετικά εύκολη και συνήθως χρησιμοποιείται το system call move_pages(), έτσι ώστε οι σελίδες που ο History Page Scheduler κρίνει ότι θα είναί *hot* στο μέλλον τοποθετούνται στη DRAM μέχρι αυτή να γεμίσει. Προφανώς για να είναι αποτελεσματικός ο History Page Scheduler θα πρέπει να ελπίζουμε ότι κάθε σελίδα που χαρακτηρίστηκε ως *hot* με βάση την προηγούμενη Εποχή Χρονοδρομολόγησης θα παραμείνει hot και στο μέλλον.

Συγκρίνουμε τον History Page Scheduler με έναν Χρονοδρομολογητή-Μάντη (Oracle Page Scheduler) ο οποίος έχει a priori γνώση για τη συμπεριφορά κάθε σελίδας για διάφορες εφαρμογές για πολλαπλές αναλογίες DRAM:NVM. Πρακτικά ο Χρονοδρομολογητής-Μάντης είναι η μέγιστη επίδοση που μπορεί ρεαλιστικά να επιτευχθεί, πρόκειται για τον ιδανικό Χρονοδρομολογητή Σελίδων. Η σύγκριση έγινε με βάση το DRAM hit-rate, δηλαδή πόσα αιτήματα κατά την εκτέλεση του προγράμματος εξυπηρετήθηκαν απο την DRAM. Προφανώς όσο μεγαλύτερο είναι αυτό το ποσοστό τόσο το καλύτερο. Η σύγκριση φαίνεται στην εικόνα 3 όπου γίνεται ξεκάθαρο το χάσμα στην απόδοση μεταξύ του τι θα μπορούσε ιδανικά να επιτευχθεί από άποψη απόδοσης (κάτω γραφική παράσταση) και τι επιτυγχάνει ο History Page Scheduler (επάνω γραφική παράσταση).

Αυτές οι δυο γραφικές παραστάσεις μας οδηγούν στην ανάγκη να βρούμε τρόπους να γεφυρώσουμε αυτό το χάσμα απόδοσης μεταξύ της απλοϊκής προσέγγισης του History Page Scheduler και του ιδανικού Oracle Page Scheduler. Κύριος λόγος της ανεπαρκούς απόδοσης του πρώτου υποπτευόμαστε ότι ειναι οι λανθασμένες προβλέψεις που κάνει όσον αφορά την εκτίμηση του *hotness* κάθε σελίδας. Το γεγονός ότι στηρίζεται αποκλειστικά και μόνο σε πρόσφατες πληροφορίες για την συμπεριφορά των σελίδων είναι αυτό που οδηγεί στην ανικανότητα του να εντοπίσει πιο σύνθετα μοτίβα προσβάσεων μνήμης. Στηριζόμενοι σε αυτήν



Εικόνα 3: History and Oracle Page Schedulers DRAM hitrate. Για κάθε εφαρμογή έχει υπολογιστεί το DRAM hitrate για διάφορες αναλογίες DRAM προς NVM. Δηλαδή η συμπεριφόρα αν η DRAM μπορούσε να φιλοξενήσει το x% των συνολικών Σελίδων μνήμης και η NVM τις υπόλοιπες.

την παρατήρηση αλλά και στην αντίστοιχη δουλεία των [8, 9] θα ενσωματώσουμε τεχνικές μηχανικής μάθησης στη διαδικασία της Χρονοδρομολόγησης ώστε να μπορούν να εντοπιστούν οι πιο σύνθετες συμπεριφορές ορισμένων σελίδων που αδυνατεί να εντοπίσει ο History Page Scheduler. Η μηχανική μάθηση μας προσφέρει μηχανισμούς χειρισμού χρονικών δεδομένων, όπου εντοπίζονται εξαρτήσεις τόσο μακροπρόθεσμες όσο και βραχυπρόθεσμες.

Χρονοδρομολόγηση και Μηχανική Μάθηση

Για το πρόβλημα μας υπάρχουν διάφορες τεχνικές Μηχανικής Μάθησης που θα μπορούσαμε να εξετάσουμε. Στη συγκεκριμένη διπλωματική περιοριστήκαμε στην

εξέταση δύο εναλλακτικών, στην χρήση Ενισχυτικής Μάθησης, και στη χρήση Βαθειών Τεχνητών Νευρωνικών Δίκτύων.

Ενισχυτικη Μάθηση

Σε μια πρώτη ματιά η ενισχυτική μάθηση, δηλαδή η χρήση ενός agent ο οποίος μαθαίνει μέσα απο τις αποφάσεις που παίρνει σε ένα καλά ορισμένο περιβάλλον ώστε να μεγιστοποιήσει μια συνάρτηση κέρδους μοιάζει να ταιριάζει στο πρόβλημα μας. Θα μπορούσε ως agent να θεωρηθεί ο Χρονοδρομολογητης ο οποίος θα μαθαίνει το μοτίβο προσβάσεων στη μνήμη, θα παίρνει αποφάσεις για την τοποθέτηση των σελίδων με σκοπό να ελαχιστοποιήσει το runtime μέσω της επίτευξης υψηλού DRAM hit-rate. Γρήγορα όμως απορρίψαμε αυτήν την προσέγγιση διότι αποδείχθηκε μη εφαρμόσιμη. Έστω ότι μια εφαρμογή έχει N σελίδες και δύο συστατικά μνήμης, ο Agent-Scheduler πρέπει να διαλέξει ανάμεσα σε 2^N τρόπους να τοποθετήσει αυτές τις σελίδες. Βλέπουμε υπάρχει εκθετική αύξηση του πεδίου το προβλήματος με την αύξηση των σελίδων.

Επαναληπτικά Νευρωνικά Δίκτυα

Μετά την εγκατάλειψη της ίδεας της Ενισχυτικής Μάθησης προτού καν εφαρμοστεί πρακτικά, προχωρήσαμε στην εξέταση του κατα πόσο θα μπορούσαμε να χρησιμοποιήσουμε Επαναλητπικά Νευρωνικά Δίκτυα για το πρόβλημα μας. Διαπιστώσαμε πως τα Επαναληπτικά Νευρωνικά Δίκτυα (Recurrent Neural Networks - RNNs) λόγω της ικανότητας τους να βρίσκουν μακροχρόνιες και βραχυχρόνιες εξαρτήσεις μεταξύ των δεδομένων αλλά και κυρίως λόγω της γραμμικής αύξησης του πεδίου του προβλήματος (αντί για εκθετική) με την αύξηση των σελίδων μνήμης, είναι κατάλληλα για την επίλυση του προβλήματος μας. Ο χρονοδρομολογητης θα μπορούσε να χρησιμοποιεί ένα Επαναληπτικό Νευρωνικό Δίκτυο ώστε να προβλέπει μελλοντικές προσβάσεις μιας σελίδας μνήμης, χρησιμοποιώντας για την εκπαίδευση του τις προηγούμενες προσβάσεις στη μνήμη. Με βάση τώρα αυτές τις προβλέψεις που αφορούν την μελλοντική συμπεριφορά των σελίδων θα μπορούσαν αυτές να διαταχθούν και να τοποθετηθούν ανάλογα είτε στη DRAM είτε στην NVM.

Είσοδος Νευρωνικού Δίκτυο

Πέρα απο την επιλογή της τεχνικής που αποφασίσαμε να ακολουθήσουμε, ένα από τα πιο σημαντικά σχεδιαστικά βήματα είναι η επιλογή της εισόδου που θα δοθεί στο Νευρωνικό Δίκτυο. Προφανώς ως είσοδος θα χρησιμοποιηθούν δεδομένα που αφορούν τις προσβάσεις μνήμης μιας Σελίδας, αλλά πως ακριβώς αυτή θα διαμορφωθεί θα επηρεάσει σημαντικά τόσο την ακρίβεια των προβλέψεων όσο και τον χρόνο εκπαίδευσης του Νευρωνικού Δικτύου. Σε αυτό το σημείο εξετάσαμε

δύο επιλογές ως προς την τροποποιήση των δεδομένων εισόδου του Νευρωνικού αλλά και την επιλογή για το ποια θα είναι η ίδια η πρόβλεψη που αυτό καλείται να κάνει.

Πρόβλεψη Deltas

Η πρώτη προσέγγιση η οποία είναι και η πρώτη που μας περνά απο το μυαλό είναι να τροφοδοτηθεί ως είσοδος το ίχνος προσβάσεων στη μνήμη (memory access trace) ως έχεί όπως στο Hashemi *et al.*[8]. Σε αυτήν την προσέγγιση το Νευρωνικό προσπαθεί να κάνει προβλέψεις για το **ποιες** σελίδες μνήμης θα προσπελαστούν μελλοντικα. Έτσι το πρόβλημα το χειριζόμαστε σαν πρόβλημα κατηγοριοποίησης παρόμοια με αυτά στον τομέα της Επεξεργασίας Φυσικής Γλώσσας (π.χ. κάνε μία πρόβλεψη για την επόμενη λέξη μέσα από ένα λεξικό). Ωστόσο το γεγονός ότι ένα σύγχρονο σύστημα έχει 2^{64} θέσεις μνήμης και επειδή σε αυτά τα προβλήματα το διάνυσμα εξόδου είναι συνήθως ίσο με το μέγεθος του λεξικού μας οδηγεί στη χρήση των deltas αντί για τις πραγματικές τιμές των θέσεων μνήμης. Ως delta τη χρονική στιγμη N ορίζουμε τη διαφορά μεταξύ της διευθύνσεως μνήμης τη χρονική στιγμή N με τη διεύθυνση μνήμης τη στιγμή $N-1$.

$$Delta_N = Addr_N - Addr_{N-1}$$

Όμως προέκυψαν αρκετά ουσιαστικά προβλήματα με αυτήν την προσέγγιση τα οποία την καθιστούν μάλλον ανέφικτη. Αρχικά ακόμα και με τη χρήση των deltas το μέγεθος του ίχνους εισόδου με βάση το οποίο θα εκπαιδευτεί το Νευρωνικό Δίκτυο είναι τεράστιο, φτάνει μέχρι και την τάξη των μερικών δισεκατομμυρίων. Αυτό κάνει την διαδικασία της εκπαίδευσης απαγορευτικά μεγάλη. Ακόμα, το πιο σημαντικό πρόβλημα αυτής της προσέγγισης μάλλον θα πρέπει να θεωρηθεί η χαμηλή ακρίβεια στις προβλέψεις. Το ότι το μέγεθος εξόδου είναι τόσο μεγαλό καθιστά το μοντέλο ανίκανο να πραγματοποιήσει σωστές προβλέψεις. Το άλλο βασικό τεχνικό πρόβλημα αυτής της υλοποιήσης είναι η χαμηλή ακρίβεια προβλέψεων εξαιτίας της κανονικοποιήσης των δεδομένων εισόδου. Είναι σύνηθες προτού τα δεδομένα δωθούν στο μοντέλο για εκπαίδευση αυτά να κανονικοποιούνται. Ωστόσο επειδή μιλάμε για διευθύνσεις μνήμης που μπορούν να πάρουν τιμές απο 0 έως 2^{64} και επειδή συνήθως οι εφαρμογές δεν *απλώνονται* σε ολόκληρο το address space, η μονή ακρίβεια κινητής υποδιαστολης 32-bit οδηγούν σε σημαντικά δεδομένα να θεωρούνται ως θόρυβος. Με αποτέλεσμα το τελικό μοντελο που έχει εκπαιδευτεί σε αυτά τα δεδομένα να μην μπορεί να πραγματοποιήσει σωστές προβλέψεις. Ενώ αν προσπαθήσουμε να αντιμετωπίσουμε αυτό το πρόβλημα μέσω clustering του address space όπως προτείνεται και στο [8] πρέπει να αντιμετωπίσουμε το πρόβλημα από τη χρήση ASLR (Address Space Layout Randomization). Όταν μαζεουούμε ίχνη προσβάσεων στη μνήμη ακόμα και για τις ίδιες εφαρμογές αυτά έχουν διαφορετικό layout στην virtual memory με αποτέλεσμα το εκπαιδευμένο μοντέλο να μην μπορεί να κάνει ακριβείς προβλέψεις.

Πρόβλεψη ανά Σελίδα

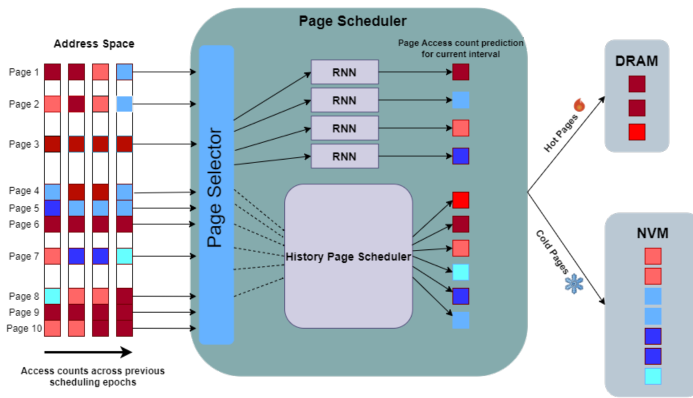
Η άλλη προσέγγιση που τελικά επιλέχθηκε είναι η πραγματοποιήση προβλέψεων ανά σελίδα. Αποφεύγουμε το πρόβλημα του **ποια** σελίδα θα προσπελαστεί (λόγω όλων των δυσκολιών που αναφέρθηκαν). Αντιθέτως διαλέγουμε το Νευρωνικό να απαντάει στην ερώτηση **πότε** θα προσπελαστεί η σελίδα μνήμης. Θα προβλέπει πόσες φορές θα προσπελαστεί μια σελίδα σε μια περίοδο χρονοδρομολόγησης. Για κάθε σελίδα που μας ενδιαφέρει θα εκπαιδεύουμε ένα Νευρωνικό Δίκτυο στο οποίο θα τροφοδοτούμε την αλληλουχία παρελθοντικών προσβάσεων που αφορούν αυτήν την σελίδα και το Νευρωνικο θα προβλέπει πόσες φορές αυτή η σελίδα θα προσπελαστεί στην επόμενη Εποχή/Περίοδο χρονοδρομολόγησης. Βλέπουμε λοιπόν ότι αυτή η προσέγγιση ταιριάζει με την περιγραφή του προβλήματος μας και πράγματι θα μας δώσει λύση σε αυτά που ζητάμε. Επίσης, με τη χρήση ενός νευρωνικού ανά σελίδα μειώνεται σημαντικά το μέγεθος του πεδίου του προβλήματος με αποτέλεσμα να οδηγηθούμε σε ακριβείς προβλέψεις. Και τέλος, επειδή μπορούμε να διαλέξουμε για ποιες σελίδες θέλουμε να εκπαιδεύσουμε Νευρωνικά Δίκτυα, μπορούμε να μειώσουμε αισθητά το overhead της διαδικασίας της εκπαίδευσης.

Σχεδίαση

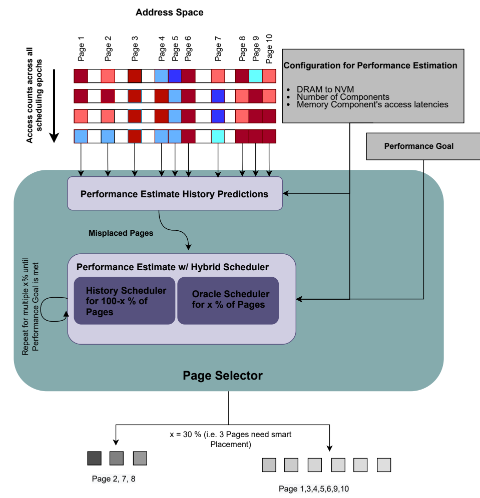
Ο χρονοδρομολογητής σελίδων (σχήμα 4) που σχεδιάσαμε σε κάθε *Εποχή Χρονοδρομολόγησης* (Scheduling Epoch) θα καλείται να κάνει ορισμένες ενέργειες.

- Με τη χρήση του *Επιλογέα Σελίδων* (σχήμα 5) θα εντοπίζονται οι σελίδες που επηρεάζουν περισσότερο την απόδοση.
- Έπειτα απο αυτήν την επιλογή οι Σελίδες θα έχουν χωριστεί σε δύο υποσύνολα. Στο πρώτο ανήκουν αυτές που απαιτούν ιδιαίτερη/έξυπνη διαχείριση και επηρεάζουν σημαντικά τη συνολική απόδοση και στο δεύτερο οι υπόλοιπες. Για κάθε σελίδα που ανήκει στο πρώτο υποσύνολο θα εκπαιδευέται ένα Επαναληπτικό Νευρωνικό Δίκτυο (stacked LSTM) το οποίο θα κάνει προβλέψεις για το **πόσες** φορές θα προσπελαστεί η εκάστοτε Σελίδα στην επόμενη *Εποχή*. Για τις σελίδες του δεύτερου υποσυνόλου θα χρησιμοποιηθεί ο History Page Scheduler.
- Εφόσον τώρα έχουμε για κάθε σελίδα έχουμε τις εκτιμώμενες φορές που θα προσπελαστεί στην επόμενη περίοδο δρομολόγησης μπορούμε να τις διατάξουμε κατα φθίνουσα σειρά και να τις τοποθετήσουμε στην DRAM μεχρι να γεμίσει και τις υπόλοιπες στην πιο αργή NVM.

Ο χρονοδρομολογητής που σχεδιάσαμε πρακτικά αποτελείται απο τρία βασικά συστατικά. Το πρώτο είναι ο Επιλογέας Σελίδων (εικόνα 5), τα μέσα πρόβλεψης



Εικόνα 4: Page Scheduler Overview



Εικόνα 5: Page Selector Overview

του αριθμού των μελλοντικών προσπελάσεων των Σελίδων (History Page Scheduler και Επαναληπτικά Νευρωνικά Νευρωνικά Δίκτυα), και η DRAM eviction policy δηλαδή ποιες σελίδες που βρίσκονται στη DRAM θα φύγουν για να μπουν άλλες στη θέση τους.

Επιλογέας Σελίδων

Ο Επιλογέας Σελίδων (σχ. 5) χρησιμοποιείται για να βρεθεί ποιες σελίδες απαιτούν χρήση Μηχανικής Μάθησης και ποιες όχι. Για να βρεθεί αυτό για κάθε Σελίδα λαμβάνονται υπόψιν τα εξής δύο. Πρώτον πόσες φορές προσπελάστηκε μία Σελίδα γιατί προφανώς οι Σελίδες που προσπελάστηκαν πολλές φορές θα είναι και αυτές που θα επηρεάσουν τη συνολική απόδοση. Δεύτερον για κάθε σελίδα βλέπουμε πόσο καλά μπορεί να τη διαχειριστεί ο History Page Scheduler. Προφανώς αν μία σελίδα μπορεί να τη διαχειριστεί σωστά βρίσκοντας το μοτίβο προσβάσεων της δεν χρειάζεται να εκπαιδευτεί ένα Νευρωνικό Δίκτυο για αυτή τη σελίδα. Αντίθετα αν ο History Page Scheduler τοποθετεί μια σελίδα στην NVM ενώ αυτή ιδανικά θα έπρεπε να βρίσκεται στη DRAM λέμε ότι αυτή η Σελίδα έγινε *misplace* και μάλλον θα πρέπει να αναλάβει τη διαχείριση της ένα RNN. Τα παραπάνω συνοψίζονται στις εξής σχέσεις που προσδιορίζουν ακριβώς το πως λαμβάνει τις αποφάσεις ο Επιλογέας Σελίδων. Για κάθε Σελίδα X υπολογίζεται το εξής για τις Περιόδους 0...N

$$\mathrm{Profit}(x) = \sum_{i=0}^{\mathrm{N}} Accesses_i(x) * Misplacement_i(x) \tag{1}$$

$Misplacement_i(x)$ παίρνει την τιμή 1 αν Σελίδα X τοποθετήθηκε λάθος απο τον History Page Scheduler την περίοδο ι, ενώ αν τοποθετήθηκε σωστά παίρνει την τιμή 0. Και $Accesses_i(x)$ είναι οι φορές που προσπελάστηκε η Σελίδα X την περίοδο ι και υπολογίζεται ως εξής:

$$Accesses_i(x) = 3 * Writes_i(x) + Reads_i(x) \tag{2}$$

Μέσα Πρόβλεψης

Όπως ειπώθηκε ήδη ως μέσα πρόβλεψης για τον αριθμό προσπελάσεων μια Σελίδας στο άμεσο μέλλον θα χρησιμοποιηθούν για κάποιες Σελίδες Επαναληπτικά Νευρωνικά Δίκτυα και για κάποιες ένας History Page Scheduler. Τη λειτουργία του History Page Scheduler είναι εύκολο να την αντιληφθεί κανείς, καθώς έχει ήδη αναφερθεί. Σε αυτό το σημείο θα δούμε μια αφηρημένη εικόνα για τη δομή των RNNs. Για κάθε σελίδα που επιλέγει ο Επιλογέας Σελίδων θα εκπαιδεύτεται ένα Επαναληπτικό Νευρωνικό Δίτκυο που θα έχει 2 LSTM layers τα οποία ενώνονται με ένα Dense Layer απο το οποίο θα προέρχεται και η μία έξοδος-πρόβλεψη.

Πολιτική Αδειάσματος DRAM

Όσον αφορά τώρα το Eviction Policy επιλέχθηκε να μην χρησιμοποιηθεί μια απλή ουρά LRU. Εκμεταλλευόμενοι την ιδεά για Address Space Clustering από το Hashemi et al. [8], υλοποιούμε μια ενισχυμένη LRU πολιτική αδειάσματος τηςDRAM, όπου θα λαμβάνεται υπόψιν και το cluster στο οποίο ανήκει κάθε σελίδα. Οπότε σε περιόδους όπου ένα cluster είναι ιδιαίτερα ενεργό, θα αποφεύγεται να αδειάζει η DRAM απο Σελίδες που φέρουν το clusterID του συγκεκρμένου cluster. Αντίθετα θα προτιμούνται να απομακρύνονται απο τη DRAM σελίδες που δεν προσπελάστηκαν πρόσφατα και ανήκουν σε άλλα clusters του Address Space.

Υλοποίηση

Για την αξιολόγηση της υλοποίησης του Χρονοδρομολογητή που σχεδιάσαμε χρειάστηκε να μαζέψουμε ίχνη προσβάσεων στη μνήμη απο διάφορες εφαρμογές. Οι εφαρμογές που εξετάστηκαν φαίνονται στον πίνακα 1. Για να συλλεχθούν τα ίχνη προσβάσεων στην κύρια μνήμη αυτών των εφαρμογών χρησιμοποιήθηκε το binary instrumentation εργαλείο *Intel Pin 3.13-98189* [14]. Αφότου κατασκευάσαμε ένα pintool το οποίο χρησιμοποιεί ένα Cache Simulator που προσομοιώνει ένα πραγματικό σύστημα μπορέσαμε για κάθε εφαρμογή να πάρουμε ένα ίχνος απο Last Level Cache Misses το οποίο είχε την εξής μορφή.

Thread ID , Timestamp , Operation , Virtual Address

Για κάθε Virtual Address που υπάρχει στο trace καθε εφαρμογής που συλλέχθηκε μπορούμε να βρούμε σε ποια Σελίδα Μνήμης (4KB) αυτή αντιστοιχεί. Χρησιμοποιώντας τώρα αυτά τα traces θέλουμε να ελέγξουμε πως θα συμπεριφερόταν έναν Σύστημα με διαφορετικά είδη μνήμης σε μια ορισμένη αναλογία αν χρησιμοποιούσε τον Χρονοδρομολογητη που σχεδιάσαμε. Πρακτικά αυτό που θέλουμε να κάνουμε είναι profiling του trace που συλλεχθηκε. Για αυτό το λόγο κατασκευάστηκε ένας Profiler ο οποίος θα παίρνει ως είσοδο το trace και διάφορες

Workload	Benchmark Suite	Domain
streamcluster	PARSEC	Data Mining
lud	Rodinia 3.1	Linear Algebra
backprop	Rodinia 3.1	Machine Learning
kmeans	Rodinia 3.1	Data Mining
bplutstree	Rodinia 3.1	Graph Theory
bodytrack	PARSEC	Computer Vision
blackscholes	PARSEC	Finance
hotspot	Rodinia 3.1	Physics Simulation

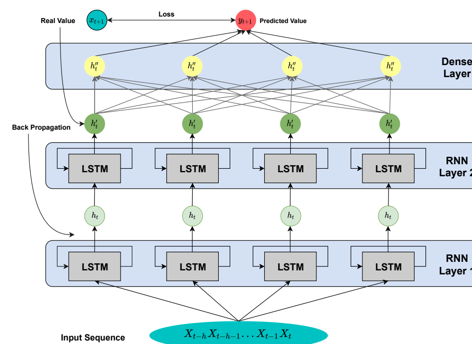
Πίνακας 1: Workloads

άλλες παραμέτρους, όπως π.χ. αναλογία DRAM προς NVM, και θα μπορεί να μας δώσει πληροφορίες για το πως θα ανταποκρινόταν το Προσομοιωμένο συστήμα με τη συγκεκριμένη είσοδο. Θα μπορεί να μας δώσει πληροφορίες για το πόσο πολυ΄χρησιμοποιήθηκε η DRAM και πόσο η NVM. Μέσα σε αυτόν τον Profiler κατασκευάσαμε και τον δικο μας Χρονοδρομολογητή για να τον αξιολογήσουμε. Ο profiler/simulator αυτός φαίνεται στην εικόνα 7.

Όπως φαίνεται και από την εικόνα ο Χρονοδρομολογητής μας έχει ενσωματωθεί μέσα στο γενικότερο κατασκεύασμα του Profiler. Βασικό χαρακτηριστικό της υλοποιήσης του Χρονοδρομολογητή μας είναι τα Νευρωνικά Δίκτυα. Όπως έχει ήδη ειπωθεί στο Σχεδιασμό κατασκευάσουμε ένα Νευρωνικό ανα-Σελίδα. Η είσοδος αυτού του νευρωνικού συνδυάζει σε αντίθεση με το [9] πληροφορίες που αφορούν τόσο την ίδια τη Σελίδα όσο και κοντινές σε αυτήν σελίδες. Αυτό γίνεται επειδή υπάρχει σημαντική πληροφορία στην τοπικότητα των δεδομένων που θα βοηθήσει στο να πετύχουμε υψηλότερη ακρίβεια στα μοντέλα. Συγκεκριμένα η είσοδος για το Νευρωνικό Δίκτυο μιας Σελίδας X προκύπτει απο τον εξής τύπο

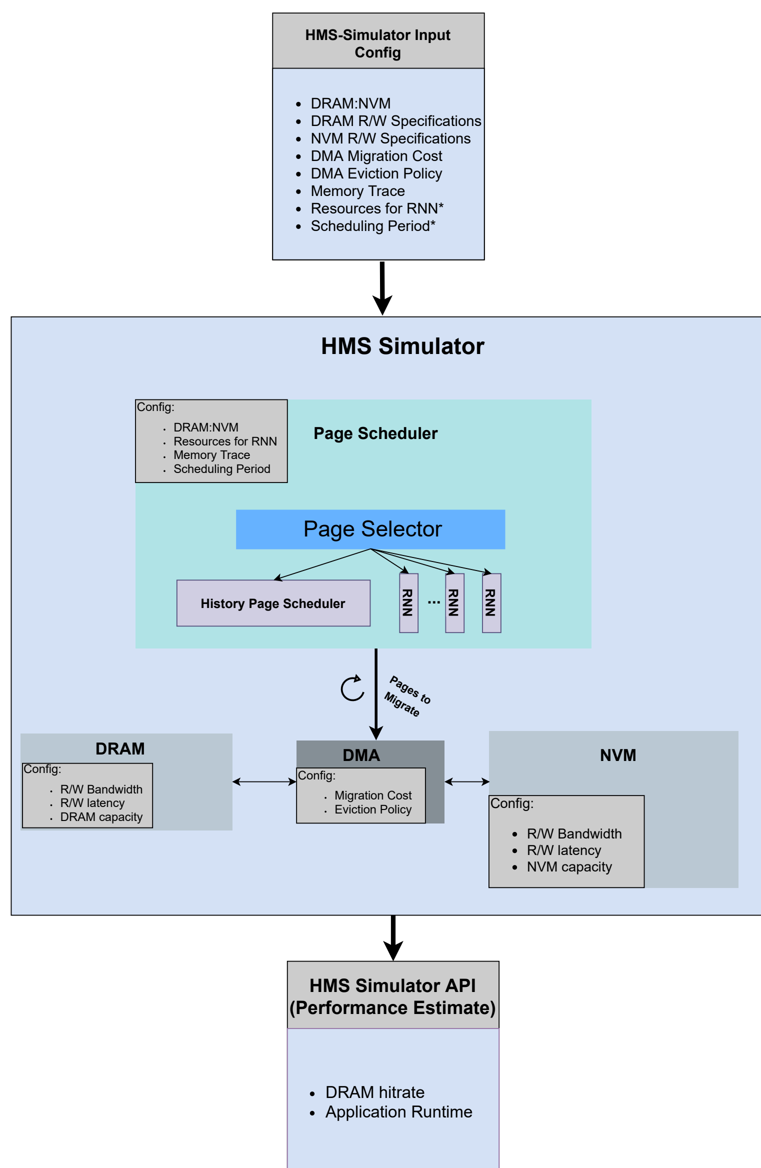
$$\text{Input}_i(x) = \sum_{j=-4}^{4} \frac{1}{|j+1|} * Accesses_i(x+j) \tag{3}$$

Δηλαδή για την είσοδο του Νευρωνικού αξιοποιούμε πληροφορίες για τους 8 κοντινότερους γείτονες μια σελίδας, ώστε να εκμεταλλευτούμε ότι για ακριβείς



Εικόνα 6: 2-Layer LSTM Neural Network Overview

προβλέψεις καλο είναι να χρησιμοποιήσουμε πληροφορίες σχετικές και με το Program Counter αλλά και με το Address Delta όπως έδειξαν και στο [8]. Όσον αφορά τώρα την έξοδο του Νευρωνικού, αυτό θα προβλέπει πόσες φορές θα προσπελαστεί η Σελίδα για την οποία έχει εκπαιδευτεί, στην επόμενη περίοδο δρομολόγησης. Στο τεχνικό κομμάτι τώρα το κάθε Νευρωνικό Δίκτυο έχει μορφή σαν αυτή της εικόνας 6. Χρησιμοποιήθηκαν 2 LSTM layers με **256** νευρώνες το κάθε ένα. Το history length ορίστηκε στα **20**, ενω τα 3/4 του συνόλου των δεδομένων χρησιμοποιήθηκε για την εκπαίδευση. Χρησιμοποιήθηκε ο **Adam Optimizer** ενώ το learning rate ορίστηκε στο **0.01**. Η διαδικασία της εκπαίδευσης σταματούσε αν το loss για τα δεδομένα επαλήθευσης δεν άλλαζει για **30** συνεχόμενες εποχές.

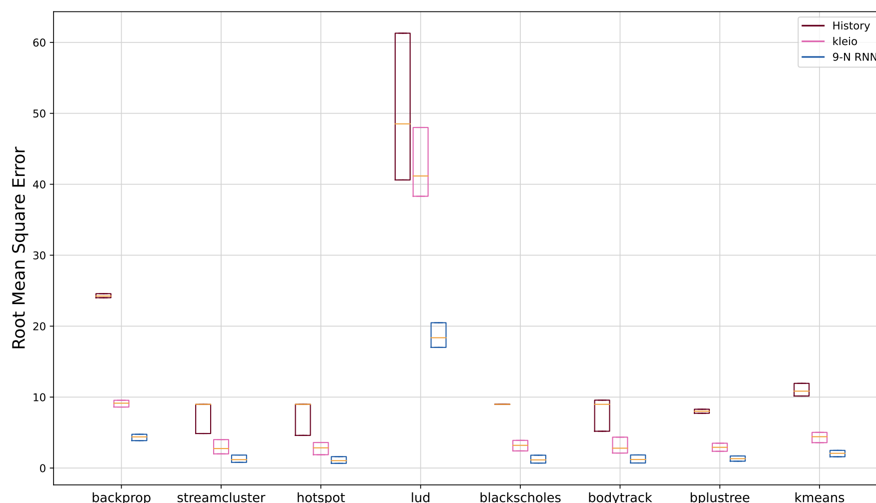


Εικόνα 7: Access Trace Profiler

Αξιολόγηση

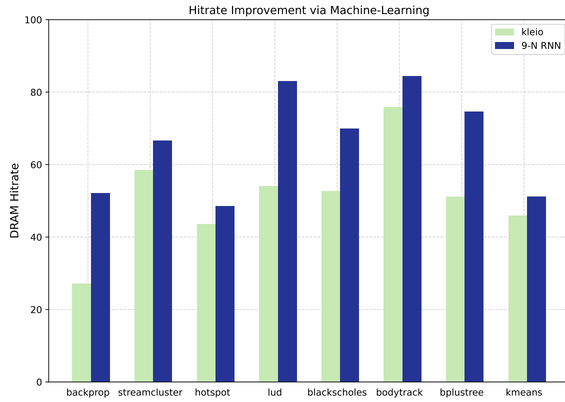
Χρησιμοποιώντας τώρα όσα κατασκευάσαμε προχωρήσαμε στην αξιολόγηση του Χρονοδρομολογητή. Αυτό που θέλουμε να δούμε είναι πόσο καλά κάνει την τοποθέτηση των Σελίδων Μνήμης ο Χρονοδρομολογητής μας για τα ίχνη που συλλέξαμε. Αυτό θα το δούμε με τη χρήση του Profiler που κατασκευάσαμε.

Πρώτο στάδιο της αξιολόγησης είναι να δούμε πόσο καλές είναι οι προβλέψεις που κάνουν τα Νευρωνικά Δίκτυα. Υποθέτουμε ότι οσο καλύτερες είναι οι προβλέψεις σε τοσο καλύτερες τοποθετήσεις Σελίδων θα οδηγηθούμε. Παρατηρώντας την γραφική παράσταση 8 καταλαβαίνουμε ότι η υλοποίηση μας κάνει λιγότερα λάθη στις προβλέψεις (οδηγούμαστε σε χαμηλότερο μέσο Root Mean Square Error) της σε σύγκριση με έναν History-Predictor αλλά και με την υλοποιήση που προτείνεται στο kleio [9].

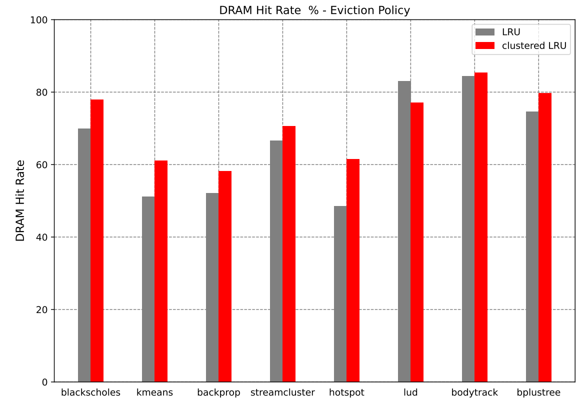


Εικόνα 8: Prediction Accuracy of the number of access counts across the scheduling intervals for the selected trained pages. History, kleio's RNNs and our RNNs are used as Access Count Predictors

Είδαμε στο πρώτο στάδιο ότι οδηγούμαστε σε χαμηλότερο σφάλμα προβλέψεων. Θέλουμε τώρα να δούμε αν αυτό μεταφράζεται και σε μεγαλύτερη απόδοση του Συστήματος. Για αυτό στο δεύτερο στάδιο της αξιολόγησης προχωρήσαμε στη μελέτη του κατά πόσο γεφυρώνεται το χάσμα μεταξύ του History Page Scheduler και του Oracle Page Scheduler. Μέσω του Profiler συγκρίναμε το DRAM hit-rate του Χρονοδρομολογητή μας με αυτό του kleio (Εικόνα 9 ο Χρονοδρομολογητής είναι ο μπλε και το kleio το πράσινο). Αυτό που παρατηρούμε είναι ότι η υλοποιήση μας φαίνεται να υπερτερεί προσφέροντας μεγαλύτερο DRAM hit-rate δηλαδή περισσότερα αιτήματα εξυπηρετήθηκαν από την DRAM σε σχέση με το kleio. Ακόμα, μπορεί με ασφάλεια να εξαχθεί ως συμπερεσμά ότι το χάσμα μεταξύ



Εικόνα 9: DRAM hit-rate for kleio and our Scheduler, normalized between 0% for History and 100% for Oracle Page Scheduler, RNNs for 100 pages and 1:8 DRAM to NVM ratio

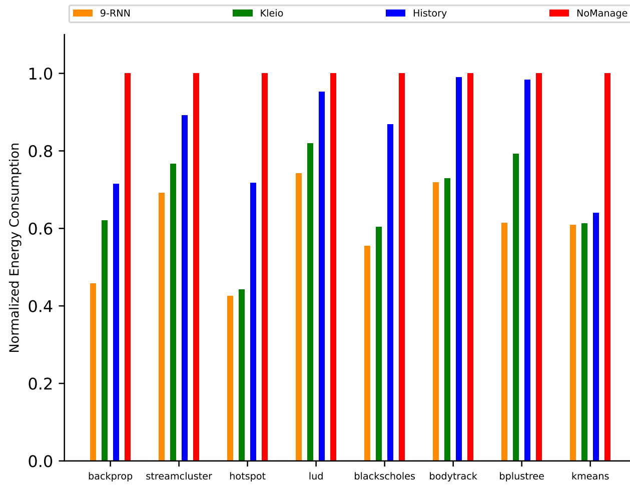


Εικόνα 10: DRAM hit-rate for LRU and enhanced-LRU, normalized between 0% for History and 100% for Oracle Page Scheduler, RNNs for 100 pages and 1:8 DRAM to NVM ratio

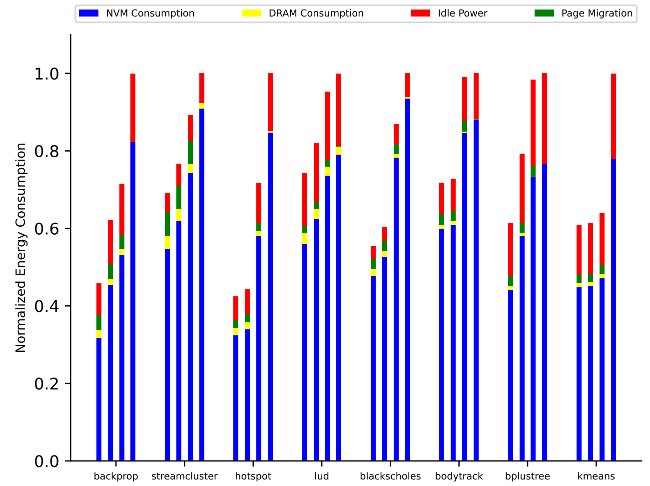
του History και του Oracle γεφυρώνεται ικανοποιητικά για τις περισσότερες εφαρμογές.

Το τρίτο στάδιο της αξίολογησης μας ήταν η επιλογή της διαφορετικής πολιτικής *αδειάσματος* (Eviction Policy) της DRAM. Θέλαμε να δούμε αν με το να δίνουμε προτεραιότητα να παραμένουν στη DRAM Σελίδες που ανήκουν σε ενεργά ανά την περίοδο clusters θα οδηγηθούμε σε μεγαλύτερο DRAM hitrate και κατέπέκταση καλύτερο performance. Τα αποτελέσματα της αξιολόγησης όπως φαίνονται απο την εικόνα 10 μας δείχνουν ότι υπάρχει μια αύξηση στην επίδοση για τις περισσότερες εφαρμογές, ωστόσο η πολυπλοκότητα της διαδικασίας του Address Clustering ίσως αντισταθμίζει το όποιο πλεονέκτημα (σε DRAM hitrate) μπορούμε να πάρουμε από αυτήν την σχεδιαστική επιλογή.

Το τελευταίο κομμάτι της αξιολόγησης αφορά την ενεργειακή απόδοση της υλοποίησης μας. Χρησιμοποιώντας τα μοντέλα που προτάθηκαν στο [15], συγκρίναμε μερικά διαφορετικά Συστήματα. Στο πρώτο δεν χρησιμοποιείται καμία τεχνική Χρονοδρομολόγησης σελίδων. Οι σελίδες τοποθετούνται στην αρχή της εκτελέσης της εφαρμογής στα διάφορα συστατικά μνήμης και δεν αλλάζουν θέση μέχρι το πέρας της. Στο δεύτερο χρησιμοποιείται ένας History Page Scheduler, στο τρίτο χρησιμοποείται ένας Δρομολογητής όπως αυτός που περιγράφεται στο kleio [9], ενώ στο τέταρτο χρησιμοποιείται ο δρομολογητής που κατασκευάσαμε εμείς. Να σημειωθεί ότι οι εφαρμογές διαφέρουν η κάθεμια ως προς την αναλογία Read και Write. Κάποιες είναι write-intensive, άλλες είναι read-intensive, ενώ άλλες είναι ισορροπημένες. Επίσης πρέπει να σημειωθεί ότι επειδή η μελέτη μας έγινε για ίχνη μικρού memory footprint η κατανάλωση ενέργειας από τη μνήμη θα επηρεαστεί κυρίως από τα NVM Read/Write operations και όχι τόσο από την ενέργεια αδρανειας της DRAM. Αυτό προκύπτει επειδή ο όρος $IdlePower = 451 * \frac{mW}{GB} * T$ για μικρές τιμές memory footprint είναι συγκρίσιμος ή μικρότερος



Εικόνα 11: Energy Comparison between NoManage, History, Kleio and Our Scheduler



Εικόνα 12: Energy Consumption details for NoManage, History, Kleio and Our Scheduler

με τον όρο που αφορά τα NVM operations ($E_{NVM} = 418.6 * W + 80.41 * R$), όπου W,R ο συνολικός αριθμός των Writes και Reads αντίστοιχα.

Παρατηρώντας την εικ. 11 όπου συγκρίνονται αυτά τα 4 Συστήματα καταλαβαίνουμε ότι η υλοποιήση μας οδηγεί σχεδόν σε όλες τις περιπτώσεις σε χαμηλότερη κατανάλωση ενέργειας. Αρκετά κοντά στην υλοποιήση μας είναι και ο Χρονοδρομολογητής του kleio, ενώ βλέπουμε οτι ο History Page Scheduler στις περισσότερες εφαρμογές έχει κατανάλωση ενέργειας κοντά σε αυτή του NoManage, δηλαδή του συστήματος που δεν φέρει δρομολογητή. Εξηγήσεις για το πως επιτεύχθηκαν οι μειώσεις στην ενέργεια μπορούν να αντληθούν απο την εικ. 12 όπου όπως περιμέναμε με την αύξηση του DRAM hitrate (και την ταυτόχρονη μειωση των operations που διεκπεραιώνει η NVM), οδηγηθήκαμε σε μικρότερη κατανάλωση ενέργειας από την NVM, ενώ σημαντικό είναι και το γεγονός ότι η ενέργεια που καταναλώνεται για να μεταφερθούν οι Σελίδες μεταξύ NVM και DRAM (Migration Cost) φαίνεται να μην εκτοξεύουν τη συνολική καταναλισκόμενη ενέργεια. Δηλαδή τα ωφέλη που αποκομοίζουμε από τη διαδικασία του Migration δεν αντισταθμίζονται απο το ενεργειακό κόστος που υπεισέρχεται με αυτό.

Σύνοψη και Μελλοντική Δουλειά

Τα δεδομένα των εφαρμογών συνεχώς αυξάνονται, ενώ το μοτίβο προσβάσεων στη μνήμη γίνεται όλο και πιο σύνθετο. Οι παραδοσιακές τεχνολογίες μνήμης δεν μπορούν να ανταπεξέλθουν σε αυτήν την αύξηση για αυτό νέες τεχνολογίες ενσωματώνονται στα υπολογιστικά συστήματα για να επιτευχθούν οι απαραίτητες αυξήσεις στην επίδοση. Έτσι έχει δημιουργηθεί μια ετερογένεια στα συστήματα μνήμης. Φαίνεται πως υπάρχει ένα χάσμα μεταξύ του σύγχρονου τρόπου χειρισ-

μού της διαχείρισης των πόρων στα συστήματα που φέρουν ετερογενείς μνήμες, και του τι θα μπορούσε να επιτευχθεί ιδανικά. Αυτό το χάσμα προσπαθήσαμε να γεφυρώσουμε σε αυτή τη διπλωματική χρησιμοποιώντας τεχνικές Μηχανικής Μάθησης.

Προτείναμε την κατασκευή ενός Χρονοδρομολογητή Σελίδων Μνήμης ο οποίος θα μπορεί να εντοπίσει ένα μικρό υποσύνολο σελίδων το οποίο αν το διαχειριστούμε με τη χρήση μηχανικής μάθησης αντι για μία συμβατική reactive προσέγγιση, θα οδηγηθούμε σε αυξημένες επιδόσεις των εφαρμογών. Για κάθε μια από αυτες τις σελίδες θα εκπαιδευτεί ένα Νευρωνικο Δίκτυο το οποίο θα μαθαίνει το μοτίβο πρόσβασης μνήμης της σελίδας. Για τις σελίδες που δεν ανήκουν σε αυτό το υποσύνολο θα χρησιμοποιείται η reactive προσέγγιση ενός History Page Scheduler. Έτσι είδαμε ότι το χάσμα που θέλαμε να γεφυρώσουμε πράγματι σε αρκετές περιπτώσεις γεφυρώθηκε σε ποσοστό 70% και ότι η κατανάλωση ενέργειας μειώνεται σημαντικά. Ενώ ακόμα παρατηρήσαμε ότι η πρόταση να χρησιμοποιηθεί μια ενισχυμένη πολιτική αδειάσματος της DRAM με τη χρήση της τεχνικής του clustering οδήγησε σε 10% κατα μέσο όρο αύξηση του DRAM hitrate σε σύγκριση με μια συμβατική LRU πολιτική. Σαν μελλοντική δουλειά προτείνουμε την επέκταση της μελέτης του Χρονοδρομολογητη για Huge Pages (2 MB) αντί για τις συνηθισμένες σελίδες μεγέθους 4 (KB). Ακόμα θα μπορούσε κανείς να προσπαθήσει να επεκτείνει αυτή τη δουλειά μελετώντας data objects μίας εφαρμογής, και κατα πόσο θα μπορούσαμε να εκμεταλλευτούμε τις σελίδες που ανήκουν στο ίδιο data object. Ακόμα ένα μονοπάτι σκέψης για μελλοντική μελέτη θα μπορούσε να είναι η χρήση τεχνικών μηχανικής μάθησης για διαχείριση μονάδων αποθήκευσης.

Chapter 1

Introduction

There seems to be a constantly increasing trend of introducing machine-learning based solutions in almost every single domain of human activities. Machine learning has been a key contributor to achieving innovations in a wide spectrum of applications ranging from simple commercial products to more complex medical applications. This wide spectrum of applications and the incredible ability of Machine Intelligence to model many traditionally complex problems seem to be the primary drivers of the growth spurt observed in the Artificial intelligence field. At the same time, the progress observed in the Computer Architecture field tends to stop following the exponential growth pattern described by Moore's law, and the performance gap between CPU and Memory Units does not seem to stop growing any time soon. These two trends, the Machine Intelligence advancements and these persistent problems in Computer Architecture, push towards a synergistic utilization of methods and techniques, in such a manner that Machine Intelligence and Computer Architecture design will mutually assist each other.

Thesis Topic

This thesis is solely dedicated to the use of Machine Intelligence, specifically the utilization of Neural Networks, in an attempt to provide improvements upon Page Scheduling techniques. We will focus primarily on hybrid memory systems, meaning modern computer systems that are comprised of both conventional DRAM and Persistent Memory components.

It is well established that modern systems are frequently designed using heterogeneous memory components. These memories are mainly used for one of two reasons. They are leveraged for either extending the capacity of main memory or for caching purposes. A system comprising of heterogeneous memory components comes with some natural trade-offs. Typically memory components which are deeper in the memory hierarchy (further from the Computing Unit) have higher storing capacity albeit at larger latency and reduced bandwidth.

In this thesis we will restrict ourselves to only working with systems that leverage the Persistent Memory for extending the main memory capacity. An important artifact of not only those systems but heterogeneous memory systems in general is addressing the limitations of increased latency and decreased bandwidth. In our case, we are mainly interested in designing an efficient page-scheduler in one HMS scenario. A page scheduler is the memory management layer of operating and runtime systems. It is responsible for the page migration across heterogeneous memory components. A well designed page scheduler insures that pages that are frequently accessed (*hot pages*) are readily available on the high performing memory modules of our system (DRAM), whereas the least important pages, those that are rarely accessed (*cold pages*) remain on the slower Persistent Memory. The use of the page scheduler module we are trying to construct is depicted in figure 1.1.

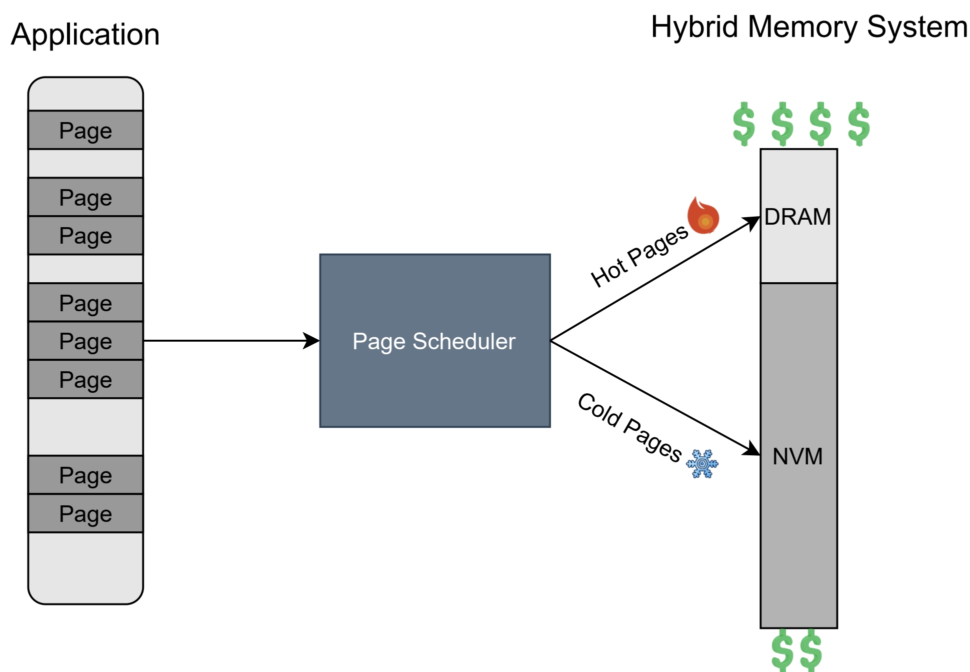


Figure 1.1: Page Scheduler tiering hot and cold pages to improve performance of and application.

Motivation

Many researchers have tried to address the challenge of tiering application pages and placing them into the memory components of a system accordingly. This is an undeniably intricate task, since the complex combination of access pattern of application pages and the runtime parameters of the application (input size, strong/weak scaling etc.) should be taken into consideration. Many researchers have considered solutions whose implementation can be integrated in

the hardware-,compiler-,Operating system-, and runtime-level [1, 2, 3, 4, 5, 6, 7]. There are a lot of similarities among these approaches. One of them is that they rely exclusively on historic information about page accesses. Specifically, the state-of-the-art in system level dynamic page management solution for Heterogeneous Memory Systems utilize the immediate observed behavior to make decisions on the best future page placement. However, as we can imagine and will prove later on this thesis, this naive policy is far from ideal when it comes to capturing the complex access pattern of modern applications.

Approach and Contributions

As far as this thesis is concerned, we will follow a similar thought process to the ones presented in the articles *Learning Memory Access Patterns* [8] and *Kleio: A Hybrid Memory Page Scheduler* [9]

The main goal of thesis is to study and construct a Page Scheduler specifically designed for a Hybrid Memory System utilizing Machine Intelligence. Our primary goal is to bridge the performance gap between the current state-of-the-art **History** and the ideal but infeasible **Oracle Page Scheduler**[1]. Briefly, our objective is to deliver a near-optimal data placement across the heterogeneous memory components on a page granularity (4KB in Linux based systems). We will try to address important questions concerning how to achieve a both practical and efficient solution. To be specific, we aim to find a solution which reasonably uses computational resources for the typically compute-intensive machine intelligence processing tasks without compromising on the efficacy to properly classify the application pages.

The specific topics we will try to address throughout this thesis are the following

- Performance gap in current solutions. Due to the recent arrival of the Persistent Memory technology, there has not been enough time to develop many algorithms applicable to this specific problem. As a result, the state-of-the-art history-based approach is relatively naive from an algorithmic perspective and does not seem capable of capturing complex access patterns.

- Scheduling interval/epoch selection. Using memory trace collection of several applications, we will try to find out how different scheduling intervals, meaning the amount of time between two data migration events, affect performance. A small scheduling interval means that the page scheduler is frequently called to make decisions about migrating the application pages.

[1]Oracle Page Scheduler uses a priori knowledge of the access pattern of the pages,meaning that it can migrate the indeed *hot pages* into the DRAM until capacity is full

- Machine Intelligence based scheduling. We will soon identify Recurrent Neural Networks as an effective and practical technique for the page scheduling problem, as it is also documented in the articles [8, 9]. We will try to make necessary adaptations to our problem so that we can achieve high accuracy leveraging RNN models, howbeit without neglecting the spatial and computational complexity of our approach. We will find out that training a Recurrent Neural Network on a per-page granularity can lead to high accuracy and significant performance improvements even when applied to a relatively small subset of application pages.

- Page Scheduler design. We design a Page scheduler after taking into consideration several performance metrics. The page-scheduler's approach will combine both the state-of-the-art history-based policy and Machine Intelligence, implemented using RNNs and more specifically LSTMs. We will try to quantify the performance improvements achieved, using a range of workloads from popular suites such as ***Rodinia 3.1*** [16] and ***PARSEC*** [17]. We will evaluate our Page-Scheduler compared to the current state-of-the-art implementations found in modern Hybrid Memory systems. Evaluation will revolve around both actual performance and energy consumption.

Thesis Overview

This thesis is organized in 8 chapters. In the second chapter we briefly summarize existing work related to our research. In the third chapter an informational background concerning Non-volatile Memory and Page Migration in modern Computer Systems is provided. The fourth chapter is dedicated to explaining briefly several types, techniques and use cases of Neural Networks and then we focus mainly on how Recurrent Neural Networks can be leveraged to provide solutions for our problem. In the fifth chapter, a detailed description of the implemented Page Scheduler is laid out, after outlining the important performance metrics that should be considered. In the sixth chapter there are the details when it comes to basic ideas, tools and design choices made concerning the actual implementation of the Page Scheduler. The seventh chapter is solely dedicated to presenting the results of the simulation and its performance evaluation. Finally, before conclusion is drawn, in the eighth chapter we clearly portray the contribution of this thesis and propose future research ideas.

Chapter 2

Related Work

Over the last years, a considerable amount of research has been carried out in order to address the resource management challenges present in Hybrid Memory Systems, and an array of interdisciplinary approaches have been employed for that purpose; many with noteworthy success. In this chapter, we summarize the recent advances in various aspects of hybrid memory management and we reference systems that use Machine Learning for the purpose of resource management.

Hardware Solutions

In this section, we reference memory management solutions in hybrid memory systems that are implemented by custom specialized hardware.

The authors of [18] introduce custom counters to monitor data accesses and enable threshold based data migration triggers. In addition, custom memory controller hardware is also proposed to enable support for page migration in non-volatile memories [19]. In [20], authors design a clustered architecture, which transparently manages hybrid memories configured in a combination of cache and flat organization, that outperforms prior work.

Apart from the purely hardware-level solutions a lot of researchers propose specialized hardware that assist existing software-level solutions by reducing critical resource overheads. A lot of operating system-level solutions [21, 22, 19] are suggesting hardware-assisted page hotness tracking.

Software Solutions

In this section, we summarize recent work revolving around the resource management of hybrid memory systems whose implementation is software based and spans either on Application Middleware or Operating System -level. Obviously, the amount of Software oriented work conducted is much more extensive

due to the fact that actual Hybrid Memory Systems have reached the market just recently.

Starting on the top of the stack, inside applications themselves, recent work optimizes the algorithmic design to perform more efficiently over the underlying hardware. The authors of [23] direct data placement for conjugate gradient, Fast Fourier Transform (FFT) and LU decomposition of a matrix by utilizing algorithm features and structures and common numerical operations. In addition, a lot of solutions proposed in recent work [24, 2, 7] highly suggest the development of custom data allocations APIs, that require application source code modifications, to improve not only the initial but the dynamic data placement of user-identified critical regions as well.

As far as the recent contributions at the user library-level are concerned, the most significant is probably Memkind [25]. It is a user extensible heap manager that can be leveraged by middleware solutions to improve performance over System with heterogeneous memory components. Various middleware-level solutions [2, 24] rely on application profiling of data access behaviors. Data tiering is optimized based on data movement cost models. Finally, Piccoli *et. al* [26] propose compiler analyses and code generation methods to migrate pages and improve data locality.

To conclude, operating system-level solutions rely on page access information available on kernel's page tables. This way, frequently accessed pages are identified and then are periodically migrated. These solutions create significant resource overheads, and often seem impractical.However, these overheads could be potentially reduced through hardware-assisted solutions. Most System-level solutions leverage existing NUMA-based page migration support or extend the NUMA-based data balancing policy [21, 19, 9, 27]. Another approach is the one proposed by the authors of [28], where a user interface, a user space library and a kernel space service is introduced to accelerate page migrations across heterogeneous memories.

Machine Learning Solutions

In this section we describe some of the machine intelligence approaches used in the system's community, focusing either on other relevant problems or just other aspects of data management.

There has been a lot of research regarding the usage of RNNs in the system software stack or in hardware. First, it was popularized by Hashemi *et. al* [8], which had a huge impact on the introduction of Machine Learning into the Computer Architecture field. Hashemi *et. al* [8] proposed leveraging Recurrent Neural Networks for the purpose of memory prefetching. The authors of [29]

utilize Recurrent Neural Network to learn I/O block level access patterns in order to optimize the performance of flash storage usage. The usage of RNNs have also been explored in Supercomputing environments. They are deployed in order to predict node failures and make decisions about timely migrating tasks on live nodes [30]. Finally, this thesis is heavily influenced by *kleio* [9] which pioneered the idea of utilizing Machine Learning to improve resource management in Hybrid Memory Systems. *Kleio* utilized Recurrent Neural Networks to timely migrate pages in Hybrid Memory Systems. Unlike *Kleio*, we deploy and train RNNs differently utilizing information that is essential for achieving high prediction accuracy and extend some of the Page Scheduling components to provide performance enhancements.

Chapter 3

Persistent Memory & Memory Management

Before engaging on the topic of efficient page migration across heterogeneous memory components in modern computer systems, there are several essential terms and concepts that should be carefully defined. This chapter provides an introductory overview of hardware and software technologies. We start by presenting the characteristics of non-volatile hardware, with primary focus on the Persistent memory module developed by Intel (Intel Optane Persistent Memory Module). Then we present a few concepts concerning general memory paging and page movement across different memory components on system level. This chapter concludes with the demonstration of several implementation challenges and difficulties that should be definitely be taken into consideration.

Persistent Memory

Typically, modern computer systems are design for a strict bifurcation of devices into memory and storage devices

Storage devices offer the highest capacity and lowest cost-per-bit for persistent storage. They are frequently implemented as block devices and thus cannot be accessed by CPU using Load/Store instructions. As a consequence storage devices are too slow for direct access and data has to be buffered into the main memory to accelerate application execution. A common technique in this context is page caching. The goal of page caching is to minimize data access to slower secondary storage devices by storing recently used pages in unused main memory. Whenever data from secondary storage is requested, the operating system first checks whether the requisite data is in the page cache. If that is not the case, the page containing the requested data has to be read from the slower storage device and is added to the page cache afterwards. When modifying data residing in the page cache, the whole memory page is marked as dirty. Periodically, all dirty marked pages are written back to the disk. Therefore,

small changes create a large I/O overhead as the whole page has to be written back to the secondary storage. For reference, the typical linux page is 4 KiB [10].

Memory on the other hand, is directly accessible by the CPU, but has smaller capacity and is more expensive that storage. Moreover, it only provides volatile storage, which means that data stored in memory is lost in the event of a power outage or a system crash.

Non-volatile main memory (NVMM) aims to bridge the gap between memory and storage by offering fast, persistent, byte-addressable memory. Various technologies can be considered as persistent memory hardware, such as Phase Change Memory (PCM) [11], Spin-Transfer Torque RAM (STT-RAM) [12], and 3D XPoint. All of them have in common that they offer a high density and low cost-per-bit while also being byte-addressable and achieving latency close to DRAM. The updated storage hierarchy, which now includes persistent memory is depicted in the following figure

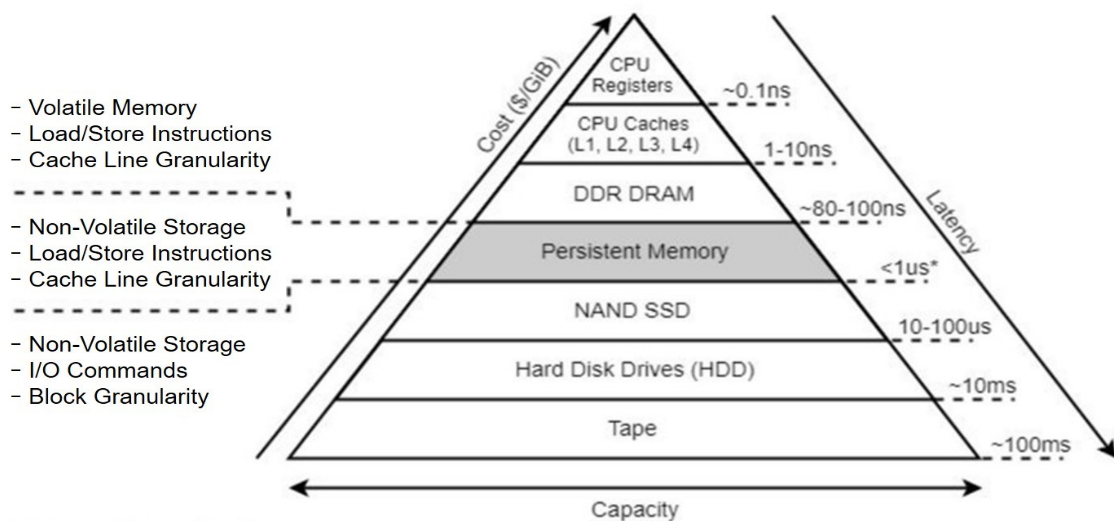


Figure 3.1: Pyramid of storage hierarchy with focus on latency capacity and cost. Persistent Memory closes the gap between Non-Volatile Memory and Volatile Memory.

The first scalable commercially available non-volatile memory hardware is Intel Optane DC Persistent Memory Module, which is based on the aforementioned 3D XPoint technology. We refer to it as DCPMM on the throughout the remainder of this chapter. The modules are available in three different capacities : 128GB, 256GB and 512GB per module.

Like conventional memory, DCPMMs are directly connected to the CPU's integrated Memory Controller (iMC) via the memory bus. A single iMC can support up to three DCPMMs. Hence, one processor can employ up to six DCPMMs across its two iMCs. The iMC is located inside the asynchronous DRAM Refresh Domain (ADR), which guarantees that data reaching this domain will survive a power failure. Internally, the iMC maintains read and write

pending queues for each DCPMM, ensuring that data is flushed to media on power failure. It should be emphasized that the ADR does not include the processor's caches. Sotres are consequently only persistent once they reach the iMC [31]. However, there is ongoing research in the area of enhanced ADR (eADR), which also includes the CPU caches [32].

To communicate with the DCPMM, the iMC uses a proprietary DDR-T protocol [33], which has a lot in common with the DDR4 standard but has been adapted to the peculiarities of non-volatile applications. Just like DDR4 (with ECC), the interface for DDR-T uses a 72-bit data bus and transfers data in cache line (64B) granularity between iMC and DCPMM [34]. Starting with Cascade Lake processor family, Intel added CPU support for the DDR-T protocol and consequently for DCPMM. Therefore, DCPMM support is not available on prior Intel CPU generations.

The DCPMM itself contains an onboard controller that coordinates the accesses to the 3D Xpoint media by performing wear-leveling and bad-block management. As the physical media access granularity of 3D Xpoint is 256B (XPLine) [31], the controller includes a small write-combining buffer in the size of 256B, coalescing adjacent 64B DDR-T writes into larger 256B media writes. As a result, the optimal access size for DCPMM is 256B [**?**, 34]. The communication between the iMC and DCPMM is depicted in the following figure.

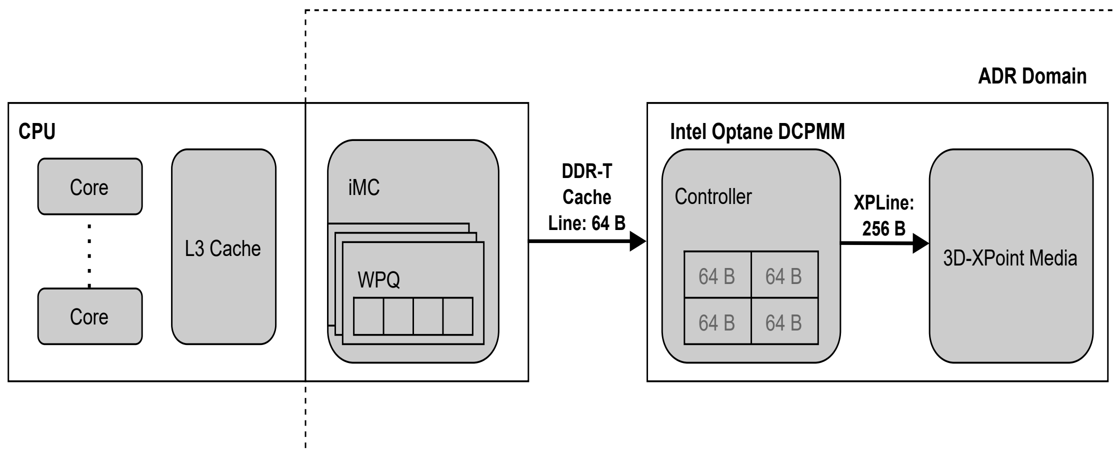


Figure 3.2: Communication Structure between CPU and DCPMM.

Latency and bandwidth are key memory technology parameters. Yang et. al [31] demonstrate in their evaluation of the DCPMM that the average read latency is two to three times higher than DRAM. Since both DRAM and DCPMM use the iMC to commit data to media, they perform similarly in terms of write latency [31]. Regarding the performance characteristics of a single Intel DCPMM DIMM, Intel specifies the sequential bandwidth for reads with 7.6 GB/s and for writes with 2.3 GB/s. As for the random bandwidth, Intel quan-

tifies the bandwidth with 2.4 GB/s for reads and 0.5 GB/s for writes. Several publications verify these numbers as well [31, 34].

When looking at the performance numbers, two things stand out: First the random bandwidth which is significantly lower compared to sequential bandwidth. Second, the performance of read and write operations is asymmetrical, with writes being the slower of the two. From this, it can be deduced that data structures with primarily random writes and a high write amplification should be avoided when working with DCPMMs. Before DCPMM became available commercially researchers used emulation to validate and test their non-volatile memory applications. These emulations often inject latency to data accesses and limit the overall bandwidth. However, the previously mentioned empirical analysis by Yang et al [31] indicates that these emulations have failed to reflect the distinctive properties of DCPMM. Characteristics like the internal 256B granularity and the asymmetrical performance of read/write operations were not incorparated into the prior emulations, resulting in less meaningful insights.

The DCPMM has two memory modes: Memory Mode and App Direct Mode [35]. In Memory Mode, the hardware acts as a larger volatile main memory. In this mode, DCPMM is transparent to the operating system and applications. To hide the longer latency and lower bandwidth, DRAM is frequently used as L4 Cache.

In App direct mode the DCPMM is directly exposed as non-volatile memory device separated from DRAM. For the operating system, the DCPMM and the DRAM appear as individual entities. Applications can now use the non-volatile memory either as an accelerated block device (Storage over App Direct Mode) or access it directly using CPU instructions on memory-mapped files (App Direct Mode).

In this thesis, we will solely use DCPMM in the first mode,the memory mode without focusing on its persistent capabilities.

Page Migration

Before diving into our scenario of interest, it would be really beneficial to obtain information about the techniques used in problems similar to ours. Therefore, we will first describe the Page Migration process in general Non-Uniform Memory Access (NUMA) systems and then move on to the more specific Heterogeneous Memory Systems comprised of both DRAM and NVM modules. When we refer to Page Migration, we mean the movement of the physical location of pages between nodes in a system while the process is running. This means that the virtual addresses of the executing process do not change. However, the

system rearranges the physical location of those pages.

Page Migration across NUMA-nodes

Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data is often associated strongly with certain tasks or users. An example of NUMA-System is depicted in the following figure.

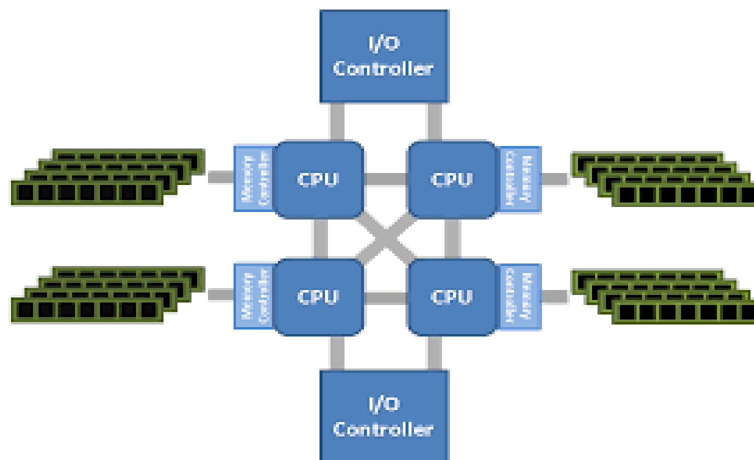


Figure 3.3: Architecture of NUMA system with 4 CPU nodes

NUMA Systems have been prevalent in the High Performance Computing field for many years. Most servers' architecture is NUMA based due to the overall speed increase and general performance improvements NUMA provides. However, the fact that every CPU node has different memory access latency depending on if the requisite data resides in local or remote memory module, entails several design challenges. As it is fairly obvious, an application will generally perform best when the threads of its processes are accessing memory on the same NUMA node as the threads are scheduled. Therefore, there is a need for *load balancing* across NUMA nodes. There are two ways that we can achieve the desired load balancing. We can either move tasks (which can be threads or processes) closer to the memory they are accessing, or we can move application data to memory closer to the tasks that reference it. Hence, we understand that in modern NUMA systems page migration across NUMA nodes is a fundamental aspect of achieving high performance.

Load balancing in NUMA systems occurs on the system level and is handled by the kernel. The most common implementation of load balancing consists of the following three steps :

- A task scanner periodically scans a portion of a task's address space and marks the memory to force a page fault [1] when the data is next accessed.
- The next access to the data will result in a NUMA Hinting Fault. Based on this fault, the data can be migrated to a memory node associated with the task accessing the memory.
- To keep a task, the CPU it is using and the memory it is accessing together, the scheduler groups tasks that share data.

The unmapping of data and page fault handling incurs overhead. However, commonly the overhead will be offset by threads accessing data associated with the CPU.

Page Migration in Hybrid Memory Systems

There is a clear similarity between NUMA systems and our field of interest, the Hybrid Memory Systems. Page migration in NUMA systems takes place in order to minimize the amount of Page Accesses of CPU-node to remote higher latency Memory modules. On the same note, in hybrid memory systems pages are migrated from the slower NVM to DRAM and vice versa in order to achieve a reduced average runtime latency. However there is a distinctive difference between the two, that forces us to think of different more intelligent solutions when it comes to the Hybrid Memory System (HMS) scenario.

A considerable amount of research concerning load balancing in NUMA systems revolves around moving threads or processes closer to the memory they are trying to access [36, 37]. However, in the HMS scenario, this approach is fundamentally inapplicable. Both Persistent Memory and DRAM have the same relative location to the CPU. The only load balancing technique left to exploit is the migration of pages across NVM and DRAM.

Following what is currently predominantly used in NUMA systems, most state-of-the-art Page Migration policies proposed by researchers rely on the use of historic information alone [13]. The page behavior is observed, and according to this immediate behavior a decision is being made on the best future page placement. The implementation of this policy on System-level is pretty straightforward, and this is probably the key reason that contributed to its popularity. A History-Page scheduler would periodically migrate pages, probably using the move_pages()[2] system call, such that those that are hot in the current scheduling epoch/interval, are allocated to DRAM until capacity is

[1] A page fault occurs when a program attempts to access a block of memory that is not stored in the physical memory, or RAM.

[2] move_pages - move individual pages of a process to another node

full. Of course, for this solution to be considered efficient we hope that these hot pages will remain hot in the next scheduling interval as well.

We compare the History Page scheduler implementation with an Oracle page scheduler, which uses a priori knowledge to periodically migrate application pages such that those that are indeed highly accessed, in the next scheduling epoch are placed in DRAM until capacity is full (Fig. 3.4). We observe that the there is a significant gap in the obtained versus the attainable application performance.

The main metric used to assess the performance of those two is the DRAM hitrate percentage. In other words, using several workloads (that will be thoroughly described later on) we measured how many main memory requests are going to be served by DRAM if a System utilizes a History Page Scheduler or an Oracle Page Scheduler. We used several DRAM to NVM ratios to assess the performance each time. We tested each workload in 8 different HMS scenarios (Table 3.1) in order to gain insight about how close or far is the performance (DRAM hit-rate) obtained by a History Page Scheduler and an Oracle Page scheduler. This examination was necessary. If we found out that the performance gap between the History Scheduler and the Oracular was insignificant, there would not be a need to come up with new Page Scheduling proposals.

For instance, we used the backprop workload and examined this application's behavior in case it is executed in the following simulated Hybrid Memory Systems:

Page Scheduler	**DRAM:NVM**
History	1:4 i.e. only 1/4 of memory footprint fits in DRAM
Oracle	1:4 i.e. only 1/4 of memory footprint fits in DRAM
History	1:8 i.e. only 1/8 of memory footprint fits in DRAM
Oracle	1:8 i.e. only 1/8 of memory footprint fits in DRAM
History	1:16 i.e. only 1/16 of memory footprint fits in DRAM
Oracle	1:16 i.e. only 1/16 of memory footprint fits in DRAM
History	1:32 i.e. only 1/32 of memory footprint fits in DRAM
Oracle	1:32 i.e. only 1/32 of memory footprint fits in DRAM
History	1:64 i.e. only 1/64 of memory footprint fits in DRAM
Oracle	1:64 i.e. only 1/64 of memory footprint fits in DRAM

Table 3.1

Obviously we expected performance drops (lower DRAM hitrate) as DRAM to NVM ratio decreases even for an Oracle Page Scheduler, due to the fact that not many applications pages fit in DRAM at a given time. An important point is also clearly illustrated. There are workloads that historic information is enough to achieve an acceptable data placement across the different memory modules. However, in most cases a Page scheduler based solely on historic in-

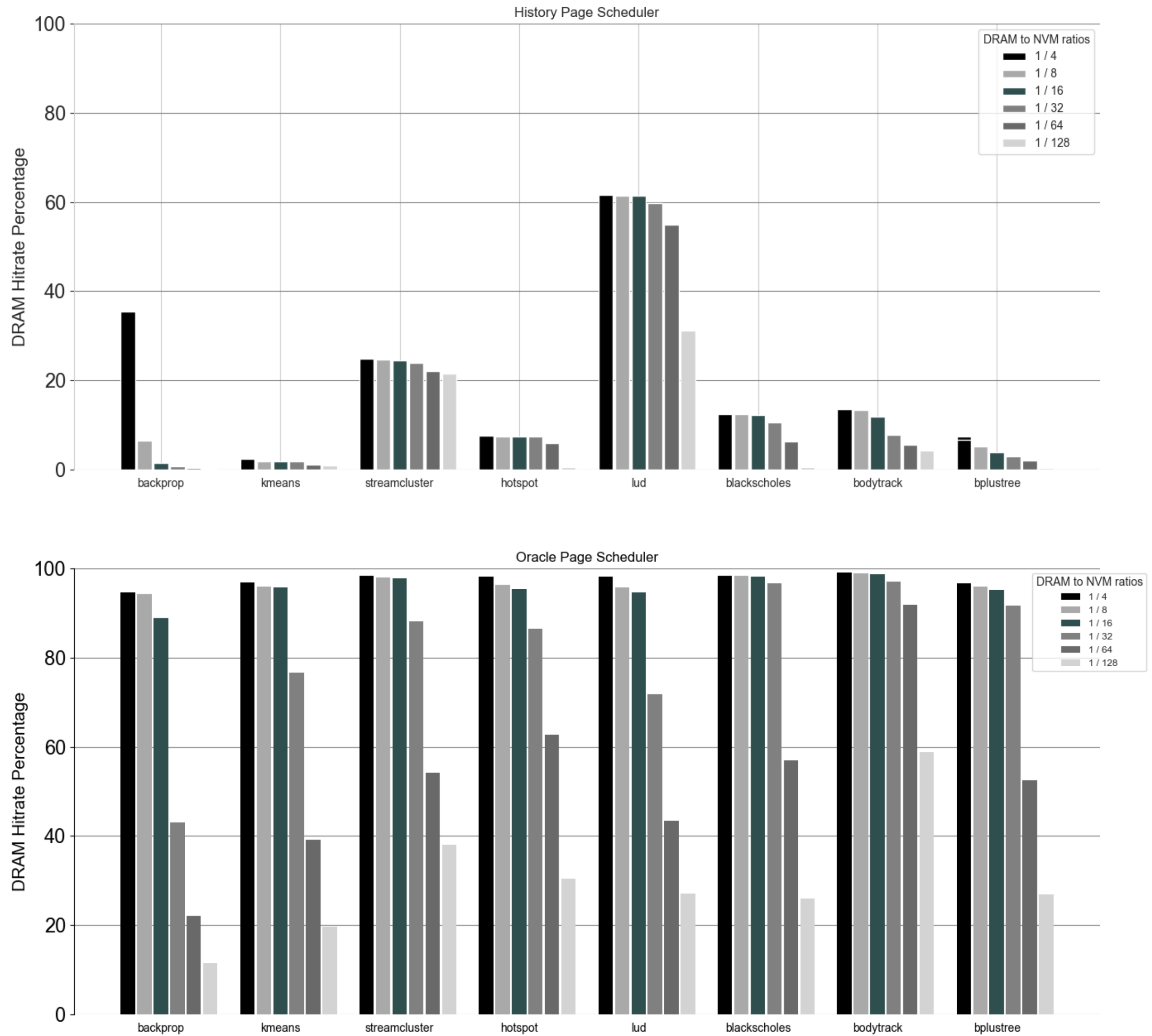


Figure 3.4: History and Oracle Page Schedulers DRAM hitrate for different workloads across variable DRAM to NVM ratios

formation is limited in the performance opportunities they can provide running on Hybrid Memory Systems. It is obvious that the gap between what a historic page scheduler achieves versus what is actually attainable is quite significant (Even 80-90% in some workloads). Constructing a scheduler that performs exactly as the oracle is of course unrealistic. A more realistic solution would likely require augmenting the state-of-the-art scheduler with more intelligent, predictive mechanisms.

Machine Intelligence based solution

As we mentioned above, it is clear that the immediately observed memory access behavior is not sufficient to capture the necessary information that allows correct future behavior predictions for making clever placement decisions. Yet,

we suspect that a larger window of accesses would probably allow the ability to capture historic information (long term access) while also leveraging recent accesses (short term access) for effective data placement.

There are a few design possibilities when it comes to augmenting the current state-of-the-art history page scheduler. For instance, we can use simple methods as Markov chains for handling the temporal aspect. Markov chains are among the most important stochastic processes. They are stochastic processes for which the description of the present state fully captures all the information that could influence the future evolution of the process. This approach is followed in this paper [38]. Another approach would be to utilize advanced techniques of machine intelligence. Machine intelligence provides mechanisms to handle temporal data, capturing both short and long term data dependencies. It seems to be a good fit to our problem, since there is a lot of ongoing research around techniques in Reinforcement Learning and Deep Neural network field, that would allows us to capture page access patterns.

Page Migration Challenges

Before designing and implementing a Page Scheduler, it is of utmost importance to highlight briefly the challenges that make the construction of an oracular one intricate. After taking into consideration the challenges and the difficulties, it would be much easier to conceive and evaluate a realistic technique of migrating pages.

Implementation Overhead

The most common techniques that are currently used in the context of migrating pages across memory components, are relatively easy to implement but are somewhat restricted when it comes to making accurate predictions. On the other hand, more enhanced techniques would require a considerable amount of metadata. These enhanced techniques are often based on Machine Learning models which can either reside in memory, or they can be implemented on-chip. Either way, using machine intelligence for achieving higher prediction accuracy comes with the cost of occupying a substantial chunk of our system's resources. Apart from that, a machine intelligence and generally a more complex scheduling approach would require some extra processing cycles, since inferring prediction models, in most cases, require some data processing, and thus the time and processing resources needed are probably non trivial.

Data Retrieval

An efficient and fully functioning Page Scheduler in Hybrid Memory Systems requires the acquisition and storage of essential data. In most cases, the Last Level Cache (LLC) Misses are processed and utilized to make design decisions. However, collecting data concerning LLC Misses is not trivial. Heretofore there is not hardware support to retrieve that data. There are a few approaches that researchers seem to take when the acquisition of such data is needed.

A relatively simple approach is to utilize the Page Table Entry Protection and Status bits.

Bit	Function
_PAGE_PRESENT	Page is resident in memory and not swapped out
_PAGE_PROTNONE	Page is resident but not accessible
_PAGE_RW	Set if the page may be written to
_PAGE_USER	Set if the page is accessible from user space
_PAGE_DIRTY	Set if the page is written to
_PAGE_ACCESSED	Set if the page is accessed

Table 3.2: Page Table Entry Protection and Status Bits.

Using a polling technique on the _PAGE_ACCESSED bit allows us to retrieve information about pages that are accessed by an application. Acquiring the data needed using this technique would require to periodically check the PTE status bit for the whole address space of a process, which is obviously unrealistic and incurs substantial overhead. This technique could be more lightweight, if used only for just a fraction of the address space, with the obvious consistency compromises that this entails. Apart from that, the other key problem with this technique is that the system's cache structure is not taken into consideration. Data collected this way might prove to be unreliable and unsafe to base our whole design analysis on.

Another approach is to use binary instrumentation. Binary instrumentation is the technique of modifying a binary program. Instructions are added, modified or deleted. By using dynamic code injection techniques, no special preparation or recompilation of the executable is necessary, since the instrumentation code is generated during the execution of the application. Many researchers prefer this approach due to the support of the development community and the high versatility that instrumentation offers. The resource overhead of binary instrumentation is certainly non negligible. However, the consistency of the results that this technique provides and the absence of actual hardware support are certainly the main contributors to binary instrumentation's popularity. The main instrumentation tool used to obtain memory accesses is Intel

Pin [39]. We will elaborate on Intel Pin and its use later on this Thesis.

Page Movement

Core functionality of the Page Scheduler we aim to construct is the Page Movement/Migration across different memory components. In other words, our Page Scheduler would have to periodically move pages from DRAM to Persistent Memory and vice versa. This is a complicated task due to the fact that Persistent Memory is relatively new and there is no special hardware support that allows seamless page migration from and to DRAM. Most research in this field has been conducted on emulators thus far. Therefore, there was no urge to take into consideration the difficulties that come with migrating pages in actual hybrid memory systems. In many cases, researchers either avoid mentioning the actual way of Migrating Pages considering it is a trivial task, or most commonly they use a system call supported by modern NUMA systems. *Move_pages()* is widely used, which is a system call that moves the specified pages of the process pid to the memory nodes specified by the nodes argument. However, *move_pages()* is designed to operate for basic NUMA nodes without taking into account the peculiarities of Persistent Memory.

Chapter 4

Machine Learning & Deep Neural Networks

Machine Learning is a subfield of artificial intelligence. It is predominantly defined as the capability of a machine to imitate intelligent human behavior. Artificial intelligence systems are used to perform complex tasks in a way that is similar to how humans solve problems. Sample data, known as *training data*, are used to build Machine Learning models, in order to make decisions without being deterministically programmed to do so. A subset of machine learning is closely related to computational statistics, which focuses on making predictions using computers; but not all machine learning is statistical learning. The study of mathematical optimization delivers methods, theory and application domains to the field of machine learning.

Machine Learning Background

Before trying to apply Machine Learning algorithms and techniques, it is crucial to define and elaborate on some key Machine Intelligence concepts.

Types of Machine Learning

As with any method, there are different ways to train machine learning algorithms, each with their own advantages and disadvantages. It is necessary to look at what kind of data each type of machine learning ingests, if we want to understand the pros and cons of each one. In ML, there are two kinds of data. On the one hand we have labeled data that has both machine-readable input and output parameters and require a lot of human labor for data labeling. On the other hand, we have unlabeled data that has only one or none of the parameters in a machine readable form. Therefore there is no need for human labor, but requires more complex solutions.

Three main machine learning methods are used today, even though there are also some types of machine learning algorithms that are used in very specific

use-cases.

Supervised Learning

Supervised learning is one of the most basic types of machine learning. Labeled data is used for the machine learning model training process. Accurate data labeling is required for this method to work properly and this is certainly not easy accomplish. However, Supervised Learning is extremely powerful in the right circumstances

In supervised learning, a small training dataset is given to the Machine Learning algorithm. This relatively small dataset serves to give an elementary idea of the problem, and data points to be dealt with and it is in most cases a part of a more extensive dataset. The training dataset shares similar characteristics with the final dataset. It also provides the algorithm with necessary parameters, which are accurately labeled and are required for the problem. Then, the algorithm tries to establish a cause and effect relationship between the dataset variables by fiding relationships between the parameters given. At the end of training, the model has an idea about the relationship between input and output and of how the data actually works. Supervised learning is mainly utilized in the following problems:

- Regression Problems : Input data are accompanied with the expected output variable which is often continuous. The trained model is used to predict this expected variable. A typical problem which falls into the category of regression is stock price prediction.

- Classification Problems : In this scenario the desired output variables are discrete. Classification refers to a predictive modeling problem where a class label is predicted for a given example of input data. Some examples of classification include spam detection, churn prediction, sentiment analysis, dog breed detection etc.

Unsupervised Learning

Unsupervised machine learning main advantage is the ability to work with unlabeled data. What that essentially means is that no human labor is required to make the data machine-readable. That allows much larger dataset to be worked on by the program. In supervised learning, the algorithm is able to find the exact nature of the relationship between two given data points. However, the absence of labels to work off of in unsupervised learning results in the creation of hidden structures. The algorithm perceives the relationship between data points in an abstract manner, without any human-given input

requirement. This hidden structure creation is exactly why unsupervised learning algorithms is so versatile. Unsupervised learning algorithms can adapt to the data by altering hidden structures instead of requiring a defined set and problem statement

Reinforcement Learning

Reinforcement learning draws inspiration from how human beings use the data they are exposed to to learn in their lives. Reinforcement learning features an algorithm which uses a trial-and-error method, in order to improve upon itself and learn from new data-situations. Outputs that are considered favorable are encouraged or *reinforced*, while non-favorable outputs are *punished*.

Based on the psychological concept of conditioning, the reinforcement learning algorithm is put in a work environment with an interpreter and a reward system. In every iteration of the algorithm, the interpreter is given the output result and then determines whether the outcome is favorable or not. If the programs finds the correct solution, the solution is reinforced by the interpreter by providing a reward to the algorithm. In case of not favorable outcome, the algorithms is forced to reiterate until a better result is found. In most cases, the effectiveness of the result is directly tied to the reward system.

Artificial Neural Networks

Artificial Neural Networks (ANNs), or as they are usually called (NNs), are computing systems. These systems are heavily inspired by the biological neural networks that constitute animal brains

What constitutes an Artificial Neural Network is a collection of connected units called artificial neurons, which broadly model the neurons in an actual biological brain. Each connection is similar to a synapse in a biological brain. A signal can be transmitted to other neurons. An artificial neuron receives a signal then processes it and can signal other neurons that are connected to it. What we refer as signal at a connection is a real number, and the output of each neuron is the result of applying some non-linear to the sum of its inputs. The connections are called edges. Neurons and edges are typically accompanied with a weight value that adjusts as learning proceeds. The weight either increases or reduces the *potency* of the signal at a connection. Neurons can have a threshold such that a signal is sent if and only if the aggregate signal is bigger than that threshold. Most commonly, neurons are aggregated into layers and each layer performs different transformations on its input. Signals travel from the input layer (first layer) to the output layer (last layer), after traversing the layers several times.

Artificial Neural Network Training

The training process of a Neural Network should be completed in a finite amount of time, while trying to minimize optimally the required computational resources. To achieve this, it is crucial to carefully tune a few training parameters such as several methods, tools and the training data. A critical attribute that the trained Neural Network should have, is the ability to generalize. The ability of the trained network to make accurate predictions shouldn't be confined exclusively to the train and control (test) data. The trained Network should be accurate and efficient for new data as well.

On a more practical level, training a neural network is nothing more than a repetitive series of fine-tuning network's parameters in order to achieve the desired output (prediction). The amount of repetitions which are often referred to as epochs influence the model's ability to accurately classify input data and generalize. Depending on the refresh of the parameters, training can be separated into two categories Batch and On-line learning. During batch learning, input data are divided in packets (batches) which are jointly used to compute the new updated weights of the Network. On the contrary, during On-line learning, every single *snapshot* of training data is used to refresh the network's weights. Training process mainly involves multiple computations of differences and derivatives, and it is heavily dependent on several parameters such as the number of training epochs, and activation, cost and optimization functions.

Activation Function

Activation Function is one of the most crucial elements of Deep Learning, due to the fact that it determines the output of every node and, by extension, the output of the whole network given a single input or a series of input data. Therefore, it affects every aspect of the model ranging from the computational efficiency of training to the network's ability to converge and accurately make predictions. For the purposes of completeness several activation functions are described below.

- Sigmoid Function: It is one the first activation functions that were used in Neural Network training. The sigmoid function is also known as a squashing function. Its domain is the set of all real numbers, while its range is (0, 1). If the input to the function is either a very large positive or a very large negative number, the output always remains between 0 and 1. The fact that, the sigmoid function's output scales in such a manner leads to small derivatives, which entails a significantly slower learning process. This is often referred to as Vanishing Gradient and is the root of many problems in the learning process

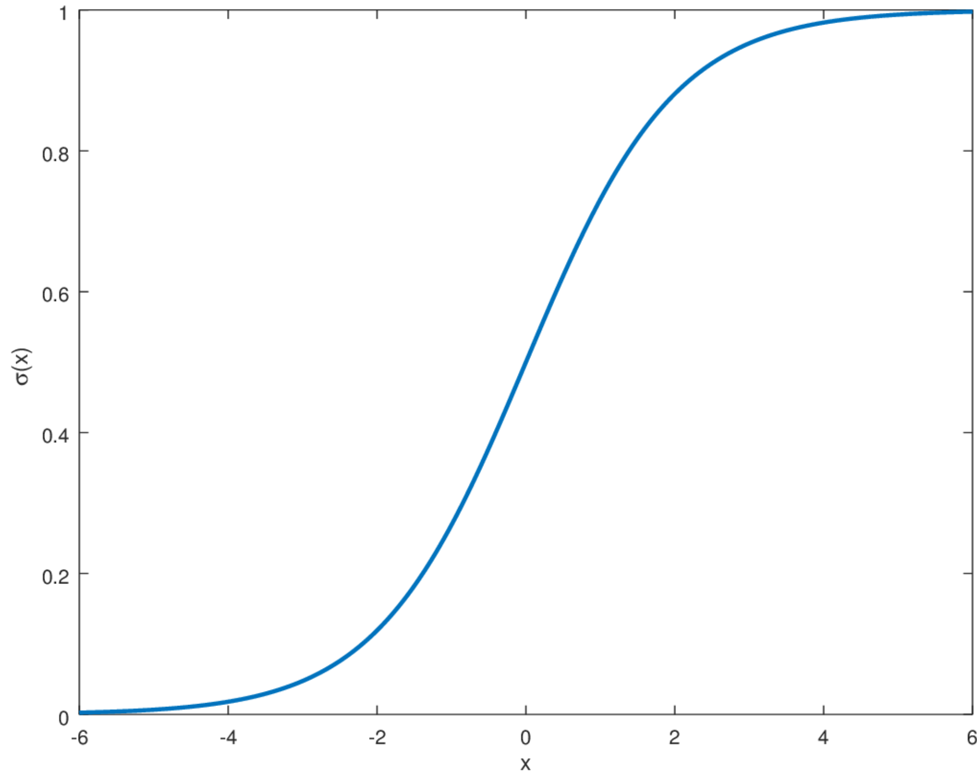


Figure 4.1: Sigmoid Function

Mathematically the output of the sigmoid function is computed as follows:

$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$

- Hyperbolic tangent: It is a continuous function which produces outputs in a scale of [-1,+1]. In other words, the hyperbolic tangent function produces output for every x value. Compared to the sigmoid function, values close to 0 are not changed. Hence, those values also contribute to the propagation process.This is advantage is the primary reason why this function is widely used in Recurrent Neural Networks. The Vanishing Gradient effect can be observed when using this function but to a smaller extent compared to the sigmoid function.

  Mathematically the output of the hyperbolic tangent function is computed as follows:

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Rectified Linear Unit - ReLU : It's the most popular and widely used activation function. Using ReLU, input data that are less than 0 are not

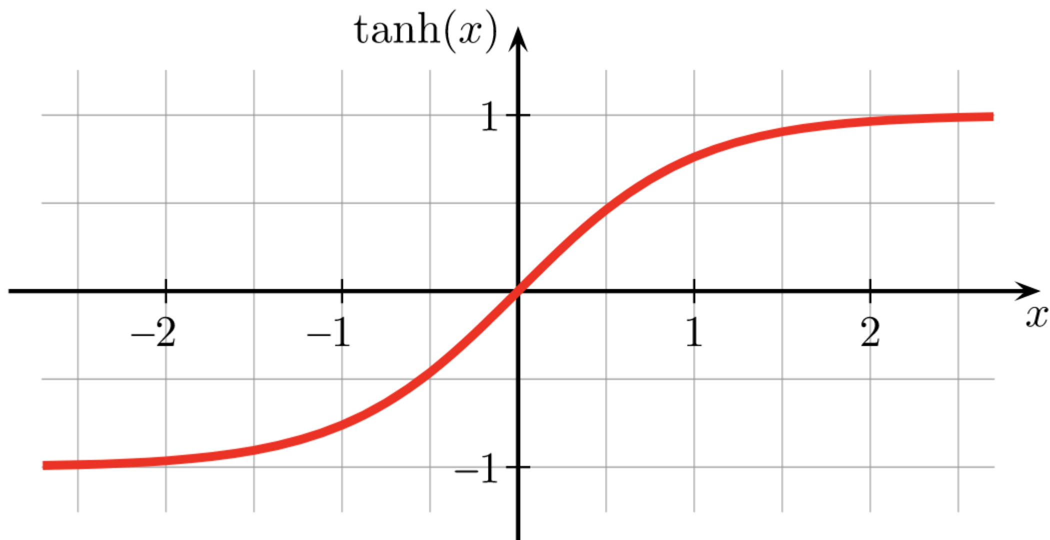


Figure 4.2: Hyperbolic Tangent Function

considered during the training process. This is clearly visible both from the math formula of ReLU, and the following figure.

Mathematically the output of ReLU function is computed as follows:

$$f(x) = max(0, x)$$

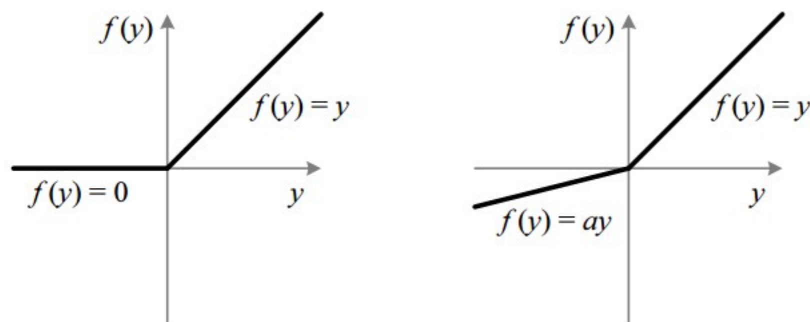


Figure 4.3: ReLU and Leaky ReLU

This particular behavior of ReLU, practically deactivates the neurons whose output value is less than zero. This way, the learning time is decreased, thus the efficiency of the network is significantly improved. That same property of ReLU also comes with a cost. For the negative values we have a horizontal segment which entails a derivative that is constant and zero. That means that there are no changes during the learning process. This restriction is well known as *dying ReLU*, which indicates the inability of using a neuron that has a negative value. To combat this problem, what is often used is a slightly modified version of ReLU, called Leaky ReLU. Leaky ReLU does not produce zero as output for the negative values. Instead, a linear function with a slight slope is used in order to enable neurons that have received a negative value to recover.

- Softmax: Softmax is a generalization of the sigmoid function. It is mainly used for classification problems which involve more than two classes. Essentially, it normalizes the output values. After applying softmax each component will be in the interval of [0,1] and the components will add up to 1, so that they can be interpreted as probabilities.

  Mathematically the output of softmax function is computed as follows:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Cost Function

A cost function is utilized as a control metric for the repetitive process of training. It essentially helps to evaluate how badly the prediction models are performing. In simple terms, a cost (or loss function) is a measure of how wrong the model is in terms of its ability to estimate the relationship between y and X. Typically, this is expressed as a distance between the actual and predicted value. The cost function is estimated by iteratively running the model to compare estimated predictions against the known values of y.

The most well-known cost function is Cross-Entropy Loss which mathematically is formulated as follows:

$$J(\theta) = -H(y,p) = -\sum_{i=0}^{\text{N}} \hat{y}_i \log(p_i, j) \tag{4.1}$$

N is the number of the different classes, y_i is the estimated value for the observation i and $p_i, j = p(\hat{y}_i = j|x)$ is the posterior probability of i to belong to class j. The equation provided above computes the cost of every input sample. For the total cost we simply need to compute the arithmetic mean of all the individual cost.

Another popular and widely used cost function is the Mean Squared Error (MSE) function. The mean squared error (MSE) tells you how close a regression line is to a set of points. It does this by taking the distances from the points to the regression line (these distances are the "errors") and squaring them. The squaring is necessary to remove any negative signs. It also gives more weight to larger differences. It's called the mean squared error as you're finding the average of a set of errors. The lower the MSE, the better the forecast.

$$J(\theta) = \frac{1}{n}\sum_{i=0}^{\text{M}}(Y_i - \hat{y}_i)^2 \tag{4.2}$$

This function takes into account purely arithmetic differences. This may cause a lot of problems when used for classification purposes, since classes most often than not are not ordered numbers.

Optimization Techniques and Algorithms

The process of minimizing (or maximizing) any mathematical expression is called optimization. The algorithms and methods used to change the attributes of a neural network (i.e. weights and learning rate) with the aim of reducing the losses are called Optimizers. They are used to solve optimization problems by minimizing the function. Various optimizers are researched within the last few couples of years each having its advantages and disadvantages.

The most common technique used for Artificial Neural Network training is Back Propagation. In fact, it is not a single technique but a group of methods that are used for optimization based on gradient. A common characteristic that these methods share, is the repetitive and recursive approach for computing the updated weights of the network. More specifically, back-propagation is used when training neural network models to calculate the gradient for each weight in the network model. The gradient is then used by an optimization algorithm to update the model weights. The whole process of back-propagation is subsequent to Feed-Forward, during which the network is given an input x and it produces an output y. The adjustment of every weight requires the computation of the gradient of the loss function with respect to the weight of every neuron k of the network $\frac{\partial J(\theta)}{\partial w_k}$. For this particular computation the chain rule (from calculus) is leveraged, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule. This group of optimization methods' popularity is fundamentally attributed to the fact that they provide an efficient weight computation, while offering quick readjustments to potential arbitrary bias value of a neuron.

Gradient descent is another well established optimization method. Gradient descent is an iterative optimization algorithm for finding the local minimum of the cost function using the gradients of the problem's parameters. The computation is repeated until convergence is reached or a pre-determined amount of iterations (Termination criteria) is completed. The network's parameters are updated based on the following mathematical formula:

$$\theta_{t+1} = \theta_t - \lambda \nabla_\theta J(\theta) \tag{4.3}$$

θ represents the amount of parameters of the network, λ represents the learning rate and $J(\theta))$ represents the cost function. In most cases, modified versions of the gradient descent algorithm are put to use which differ in the amount of data they use.

- Batch gradient descent, also called vanilla gradient descent. The error for each example within the dataset is calculated. However, the model is updated only after all training examples have been evaluated. This whole process is like a cycle and it's called a training epoch.

- Stochastic gradient descent (SGD) differs from Batch Gradient Descent in terms of doing this for each training example within the dataset. That means that the parameters are updated for each training example one by one. This can make Stochastic Gradient Descent much faster than Batch Gradient Descent depending of course on the problem. What is really advantageous in SGD is that the frequent updates allow us to have a pretty detailed rate of improvement.
- Mini-batch gradient descent combines the concepts of SGD and batch gradient descent. That's what makes Mini-batch the go-to method. The training dataset is split into smaller batches and updates are performed for each one of those batches. That is how it establishes a balance between the robustness of SGD and the efficiency of Batch Gradient Descent.

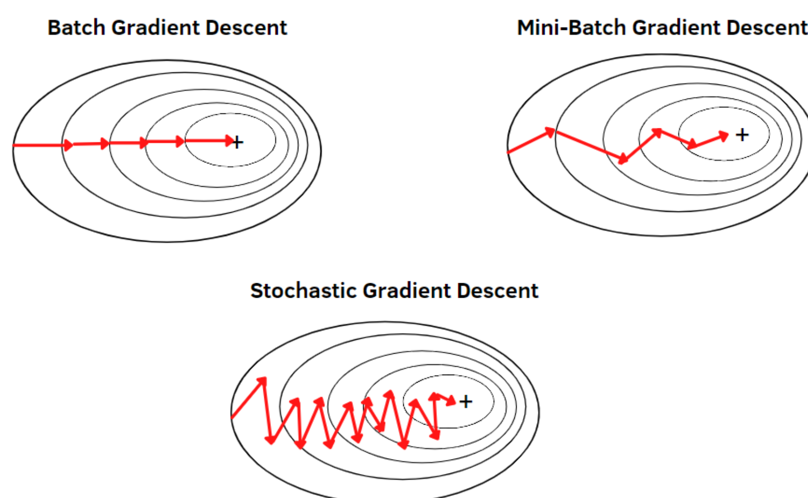


Figure 4.4: Convergence as reached in the three different Gradient Descent Scenarios

Undeniably, Gradient Descent algorithm has a few drawbacks as well. That leads to exploring further modifications of the vanilla algorithm. Its constant learning rate and the slow convergence in problems with a big amount of data (due to the large gradient variance) lead to new algorithms specifically adapted to those particular problems. Adam optimizer (Adaptive Moment Estimation) is based on SGD, but uses a changing learning rate for every parameter-weight of the network. It utilizes both first and second Moment of gradient, and simultaneously two Forget Variables are leveraged to avoid oscillation leading to faster convergence. A similar approach when it comes to learning rate is used by Adagrad as well. Adagrad decreases the learning rate faster for frequent parameters, and slower for infrequent parameters. Unfortunately, there are some cases where the effective learning rate drops significantly fast because we accumulate the gradients from the beginning of training. This might make the model reach a point where the learning process practically ends because

the learning rate is almost zero. This issue was mitigated by some algorithms that extend AdaGrad. Adadelta is such an algorithm, which requires no initial learning rate setting and is insensitive to hyper-parameters.

Recurrent Neural Networks

Many of the simplest neural networks are feed-forward networks. In Feed-Forward networks data flows in only one direction. Normally, the data flow begins from a set of input nodes which is connected to a set of the so-called hidden layer nodes. These hidden nodes then connect to a set of output nodes. A basic network architecture demonstrating this setup is shown in figure 4.5.

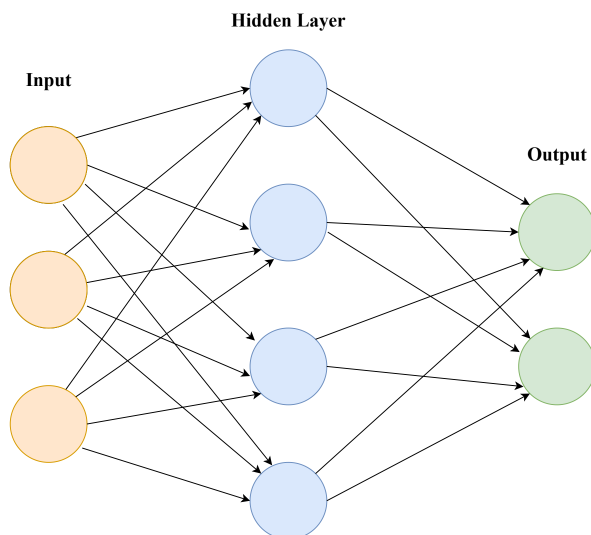


Figure 4.5: : A simple feed-forward neural network. Data only flows in one direction. Data goes from the input layer, to the hidden layer, to the output layer.

Often, we use feed-forward networks for tasks where the ordering of the data that will be used as input is not important. Image Classification is a good example of such a task. Typically, for these types of problems, there is not any relevant temporal relationship between different items in the database of images. That is mainly why Networks for Image Classification often randomize the order of the input training data each epoch. Obviously, this is done in order to refrain from inferring a relationship between the ordering of images which is actually non-existent. By contrast, Recurrent Neural Networks are fundamentally used when the temporal relationship between different items in a dataset is important. With RNNs, nodes in the hidden layer receive information both from the input nodes and from themselves as well. Information about what was seen in the past is saved and is used in order to make contextual judgements about what they are seeing in the present. Information can be actually shared across several time steps, and visualizing how information is passed from one time-step to the other is called RNN-unfolding. An example of this is shown in

figure 4.6.

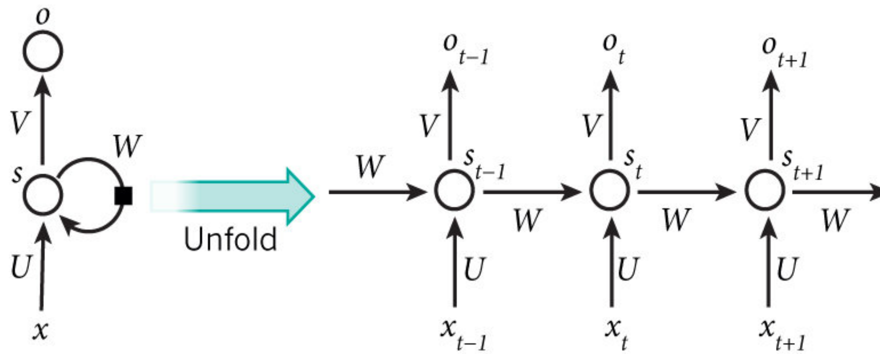


Figure 4.6: A simple RNN unfolded through time. x represents input data being sent to a hidden layer s. On a new timestep s_t, information W about the past is received from the s_{t-1} layer on the previous timestep. [40]

While the ability to share information across time-steps is very useful for being able to contextualize data being seen in the present, it has practical limitations. Neural networks are typically trained through backpropagation, where the errors in the weights of each node in the output layer are propagated backwards through the network towards the input layer. This process allows the weight values in each node to be tweaked so that the network can make better predictions in the future [41]. For RNNs, backpropagation also happens backwards through time. After the node weights are adjusted for s_t, the calculated errors in those weights will then be used to adjust s_{t-1}. The errors found in s_{t-1} will then be used to adjust s_{t-2}, and so on.

There is a serious problem that needs to be considered. This whole process can lead to an increasingly long chain of floating point multiplication, which range between (0,1). What that means is that eventually calculated error values will be *dragged* close enough to 0 that typical floating point resolution will no longer be sufficient to accurately track. Therefore, there is a limitation on how many time-steps can be kept track of when we use Recurrent Neural Networks. This is issue of shrinking errors is broadly known as Vanishing Gradient problem [42] The vanishing gradient issue might arise for problems that use as few as 100 time-steps. Many problems require a much larger number of time-steps, often they in the order of thousands. For this to be feasible, there is a necessity to address Vanishing Gradient.

LSTMs

In 1997, Long Short-Term Memory was proposed by Hochreiter and Schmidhuber as a direct way to address the vanishing gradient problem faced by Recurrent Neural Networks [43]. LSTMs are considered to be a direct evolution of Recurrent Neural Networks, and in fact they are used in many networks today.

LSTMs cells have what is referred to as gated memory. The memory contained previously in Recurrent Neural Networks flowed time-step to time-step unregulated. Now, LSTMs have a series of functions, called gates, that the data flows through. These gates are used to regulate the importance of data. Modern LSTMs have three such gates, the forget gate F, the input gate I, and the output gate O. Each one of these gate is in fact itself a simple Neural Network and it is automatically tuned during the overall training. LSTM memory is also widely known as cell state C.

The input gate decides what new information to add to C. The forget gate decides what information to throw away from C at the current time-step. The output gate simply decides what information to provide as output from the LSTM cell at each time-step. A detailed overview of the LSTM cell is shown in figure 4.7.

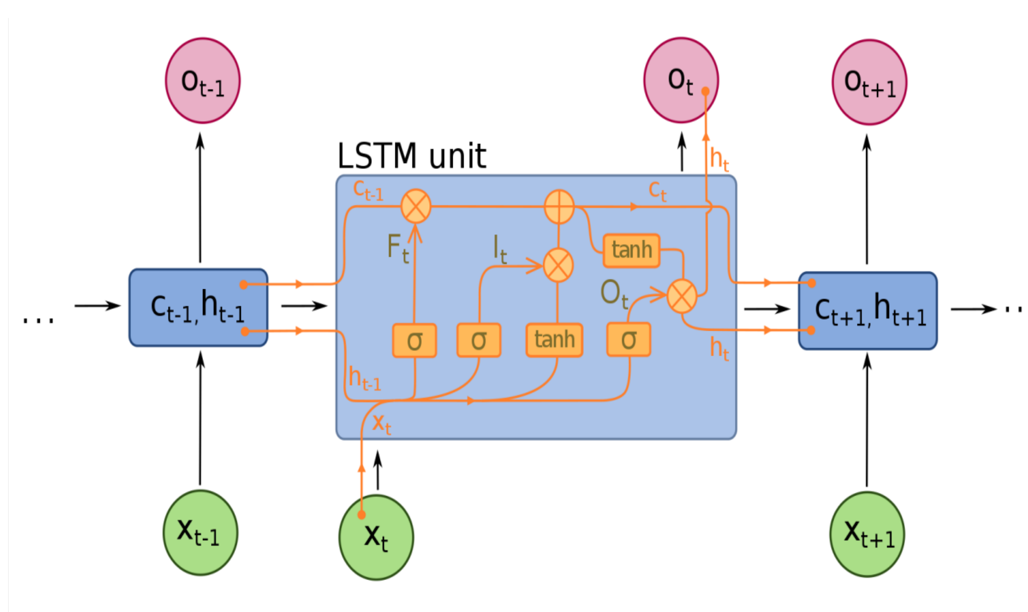


Figure 4.7: Overview of an LSTM cell. Data from both the input layer x_t and previous timestep h_{t-1} pass through the forget gate F_t, the input gate I_t , and the output gate O_t. The cell states, C_{t-1} and C_t, flow along the top of the LSTM cell, and act as its memory.

The most important part of LSTMs is the cell state C. Cell state is what allows information to flow between LSTM cells of thousands or even millions of time-steps. Vanishing Gradient problem is now removed, since over time information is changed only by addition via the input gate I or by removal via the forget gate F and this allows calculated errors to be carried back much further through time.

The advantages that LSTMs have over traditional Recurrent Neural Network is what made them really popular in many fields. For instance, they have proved to be a great choice for dealing with NLP problems such as Speech Recognition and text to text language translation where long sequences of data are used [44, 45].

Page Scheduling as a Machine-Learning problem

In this section, we explore the machine intelligence techniques that seem to be a good fit when designing a scheduler for application pages management over hybrid memory systems. As we observed in chapter 2, there is a need for more intelligent page placement decisions across scheduling epochs, since the vanilla History Page Scheduler seems incompetent to perform adequately enough prediction-wise.

Reinforcement Learning Approach

At first glance, deep reinforcement learning, the technique that enables an agent to learn through taking actions in a defined environment in order to maximize reward via the received feedback, seems like a good fit. The page scheduler could be interpreted as an agent with the purpose of learning the dynamic data layout that optimizes the application performance across its runtime. This can be achieved by the following course of actions. The agent periodically interrupts the execution of the application to take an action, that is to migrate pages across the memory components. Then the application continues its execution and during the next scheduling epoch the page scheduler receives its reward, that is the DRAM hit rate with the most recent page placement.

Although this approach of reinforcement learning seems highly compatible to the problem description of designing a hybrid memory page scheduler, it is fundamentally difficult to implement. The possible paths of action that the agent can take are prohibitively large. The page scheduler needs to act upon every single page. For instance, if we assume that we have two memory components and N pages, the agent needs to consider and choose from 2^N possible placements. As it is fairly obvious the problem space grows exponentially and depends on the number of the application pages. On account of this, we stopped considering the reinforcement learning approach for the context of our problem, despite the fact that there are several researchers that have used this technique in similar cases [46, 47].

Recurrent Neural Network Approach

Another Machine Intelligence approach, which seems suitable for the design of the hybrid memory page scheduler, is to use Recurrent Neural Networks. RNNs are able to find long-term dependencies in a sequence of data points and make predictions about data behavior, whereas in reinforcement learning interactions with the environment are the facilitators of learning.

This machine intelligence technique has already been leveraged to solve similar problems. It has been used for hardware memory prefetching [8] and in

Memory Management in hybrid Memory systems as well. This technique has a huge advantage over the reinforcement learning one. Different from reinforcement learning, where the problem space complexity grows exponentially with the number of pages, in the Recurrent Neural Network scenario it only grows linearly, thus it is more lightweight and not as resource-intensive.

In the context of the page scheduler, these data points can be the sequence of page accessed throughout an application execution time interval. The page scheduler can deploy an RNN in order to learn the page access pattern and make predictions about future page accesses. Using these predictions the page scheduler can now segregate the pages according to their *hotness* (access frequency). Pages that are predicted to be frequently accessed in the future are allocated to the lowest access latency memory technology (DRAM), and the remaining pages are allocated to the slower NVM.

A Recurrent Neural Network can be constructed via the combination of multiple LSTM neurons, which where thoroughly described in subsection 4, on a single layer, stacked LSTM layers together with regular Dense layers. The input sequence is split into subsequencies of history length h, in a rolling window fashion. During a training epoch all subsequencies are fed into the network, which then makes a single value prediction for each subsequence. Using the loss function which takes as a parameter the predicted and the actual value, difference is calculated and backpropagated into the network, resulting in weights and biases updates. Training is terminated when there is no reduction in loss, thus the network cannot make any predictions closer to the actual values. An abstract design of the LSTM we want to construct is depicted in figure 4.8.

We will elaborate on the layout and the selected hyper-parameters of the network we constructed and evaluated in the chapters that follow.

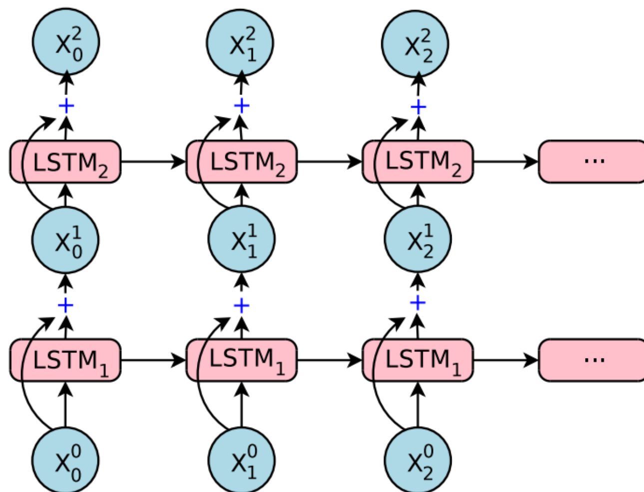


Figure 4.8: Example layout of an RNN using LSTM neurons

Neural Network Input

We concluded that Recurrent Neural Networks seem to be the best fit for the problem we want to deal with. One of the most important steps when trying to design a Neural Network is choosing the features that describe the problem and are to be used as inputs. There are several paths that can be taken when it comes to the format of the input sequence of the Recurrent Neural Network we want to construct. We will now discuss the representation of the data sequence related to memory access behaviors to be fed into the RNN and the interpretation of the predicted value. This step is crucial not only for the accuracy but for the training time of the generated model as well.

Before moving on the different approaches we can take, there is a need to explicitly define the following.

- **Input Data.** It is truly important to understand what type of data we have available for each application. The data we can obtain during an application's runtime is a memory trace, a sequence of the page accesses that were serviced from main memory and not the processor's hardware caches.

- **RNN's purpose (Learning Objective).** The aim of the RNN training is to be able to make predictions with respect to future memory accesses on page level granularity, so as to aggregate the accesses and then determine the ordering of heavily accessed pages. These predictions will happen periodically, when the page scheduler is invoked. The appropriate page migrations are determined and then executed.

- **Training Time.** Our main goal is to facilitate fast learning via reduced training times. Our machine intelligence based solution should definitely be practical and must operate within certain time and computational resource budgets. Learning can be certainly accelerated using technologies like GPUs, TPUs or custom RNNs accelerators, but we undoubtedly need to explore ways to enable rapid learning via the training methodology (e.g. input type, RNNs hyperparemeters).

Deltas Prediction

We will now describe the most intuitive approach and that was also explored by Hashemi et al. [8] for the purpose of prefetching future memory address accesses. In this approach the memory access trace is fed into the RNN as it is. The RNN looks at the input sequence and tries to predict the page that is going to be accessed next.

In this use case the problem is treated similar to Natural Language Processing classification problems (guess the next word from a whole vocabulary). However, there is notable problem with this. A modern system has 2^{64} possible memory locations, and classification networks typically output vectors of length equal to that of the class vocabulary size. Building a neural network with a vocabulary size as large as 2^{64} is currently completely infeasible because of limited memory resources. A vector of this length would take approximately 18 exabytes to store. Due to this, most researchers choose to predict address deltas, as opposed to the raw addresses themselves. An address delta at a timestep N is defined as the address at timestep N minus the address at timestep N -1 .

$$Delta_N = Addr_N - Addr_{N-1}$$

Using address deltas we manage to cut down on the number of possible classes that our network must predict. The previous equation can be used to convert virtual addresses into address deltas. An example of this is showed in the table 4.1.

PC	Virtual Address	Delta
0x400619	0x7EFEE09AC4A8	0x18
0x400619	0x7EFEE09AC4C0	0x40
0x400619	0x7EFEE09AC500	0x40
0x400619	0x7EFEE09AC540	0x40
0x400619	0x7EFEE09AC580	0x40

Table 4.1: Virtual addresses and their converted address deltas. Repeated address deltas shows that the program is accessing data at a strided interval.

This approach to feed the whole input sequence, even after transforming the virtual addresses to address deltas, has some serious limitations that should be definitely taken into account.

Training Time: More often than not the input trace contains millions of memory accesses. If we consider High Performance Computing applications they might even reach the order of billions. Training an RNN using the input data "as-is" is unrealistic, since the training time in such scenario is prohibitively large (in the order of several days).

Prediction Accuracy: The main limitation of this approach is probably the difficulty of achieving acceptable prediction accuracy. The output value space is significantly large, even when using address deltas instead of actual virtual addresses and that comes with the cost of low RNN prediction accuracy.

Model inaccurate due to Normalization: It is often beneficial when working with neural networks to normalize the different input features to be on the same scale. The primary reason for doing this is that each of the different

input is given equal preference during network training. If input features are on vastly different scales, then the network may end up ignoring some of them and that will probably lead to poor prediction accuracy. However, in our scenario normalizing virtual addresses is a complex task. A typical computer will have a 64-bit address space. Therefore, there are 2^{64} different possible address values. Normalizing values in such a large scale might be hindered by problems in floating point resolution. Normalized values for Neural Networks are often stored as 32-bit single precision floats. This particular level of precision is clearly inadequate when trying to accurately normalize specific address across such a large numeric region. After analyzing sample memory traces of several workloads, it was often observed that different areas of activity within the address space would be separated by large regions of empty space. (as seen in fig 4.9). Considering the sample trace in fig. 4.9, if normalization happens across the whole address space with a 32-bit float, then there will effectively be only three values. Even if sufficiently high resolution float was used, it would not lead to a better trained network. This is because Neural Network are fairly invariant to noisy data. Consequently, tiny changes in floating point numbers would look just like noise. Hence, normalizing address in the standard way will most certainly lead to a poor performing model.

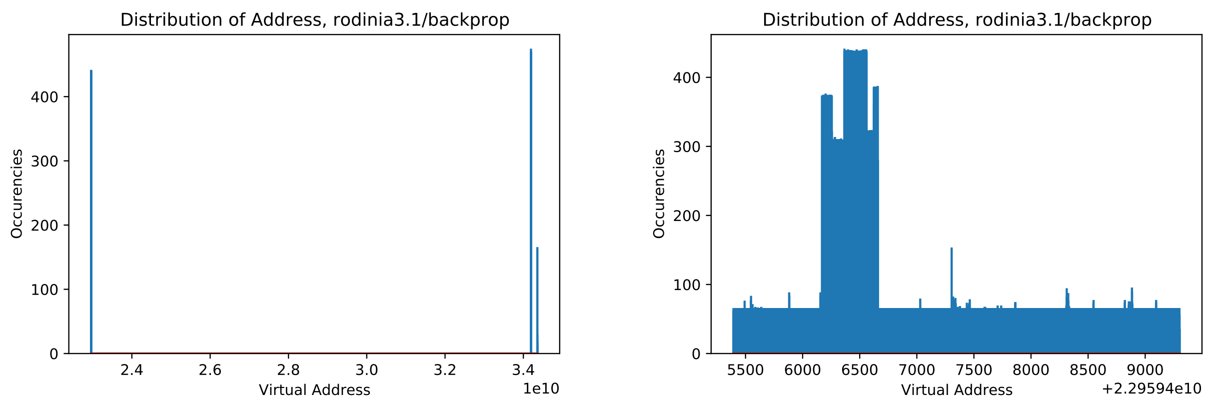


Figure 4.9: The first chart depicts the distribution of addresses on the whole address space, while the second chart depicts a zoomed in area. When viewing the whole address space, misses are very sparse. However, when zooming in on one of the bars it can be seen that it is very rich in information. Due to floating point resolution this information will be lost if normalization happens across the whole address space

Model inaccurate when using K-means Clustering: To combat the inaccuracy induced by the normalization of addresses in the standard way, a common practice, as presented in [8], is to reduce the output value space, by discretizing it into frequently appearing values (classes), and training different RNNs across clusters of the address space covered by the application. The popular K-Means algorithm was often used for clustering, and the number of clusters was chosen manually based on inspecting the distribution of addresses

histograms, on the whole address space.

Once clustering was complete, addresses were then normalized locally on a cluster by cluster basis. However, manually choosing the clusters will be affected by Address Space Layout Randomization (ASLR). ASLR is a security technique employed by most operating systems today that randomly offsets the starting locations in memory of many key data areas [48]. These areas will typically include the executable base, stack, heap, and libraries. This is done so that an attacker cannot reliably jump to a known location in memory where sensitive data might be stored. This results in different layouts in virtual memory when gathering memory traces, which in turn has an impact on the accuracy of the RNN model. The different layouts in virtual memory can easily be observed in figure 4.10.

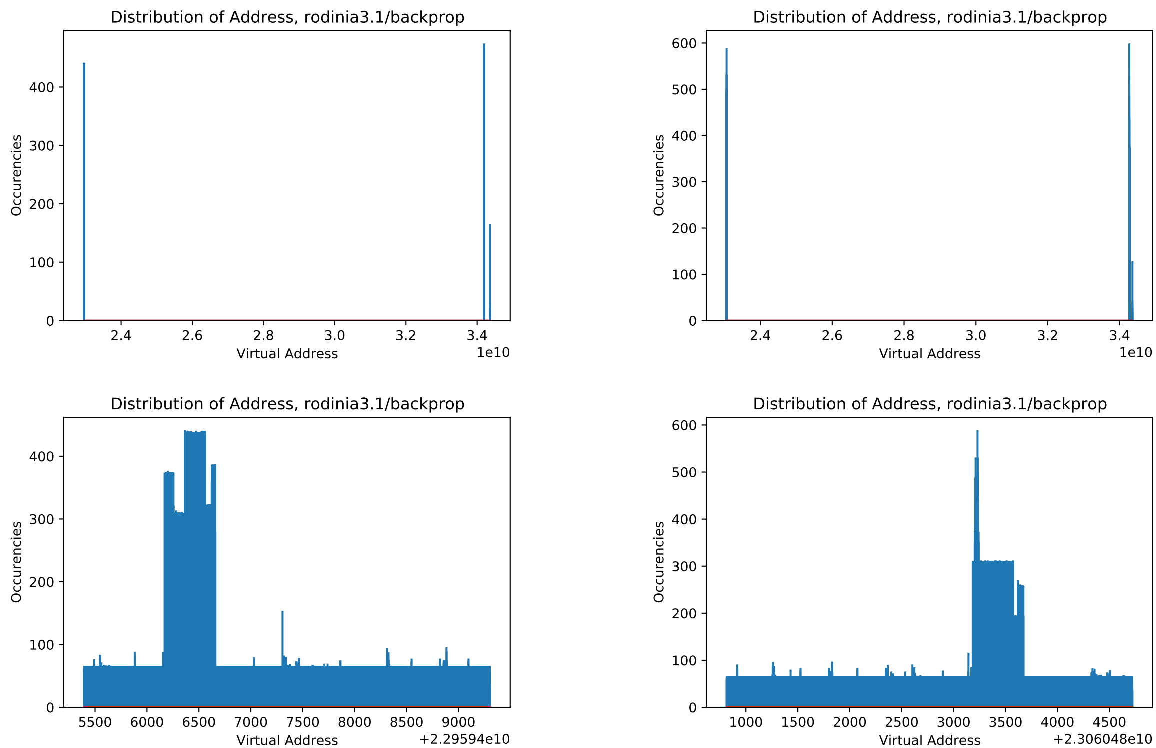


Figure 4.10: ASLR as it affects the address distribution of the backprop of rodinia suite. The top two charts show the distribution of addresses across the global address space. The bottom two charts show how things can change on the local level

Per Page Prediction Approach

The other approach that can be followed in order to tackle our problem, is to make Per Page predictions. We avoid predicting **which** page is going to be accessed next (Across Page Prediction) due to the aforementioned difficulties this approach entails. Instead, we explore the case of predicting **when** a page is going to be accessed next. More specifically, we try to predict how many times a page is going to be accessed in every scheduling epoch interval. We

train individual RNNs (per page RNN) for every application page we want. We feed the per-page RNN our page of interest's sequence of access counts across the scheduling epochs and predict the amount of accesses of this page is going to receive in the next scheduling epoch. There are several benefits that comes with transforming the input sequence in this manner.

Great fit to our problem's description: Our problem is to classify pages according to their *hotness*, using each page's access count on every scheduling interval, so as to order frequently accessed pages and appropriately migrate them across the memory components. Transforming our input trace following the Per Page Prediction Approach achieves exactly just that. The trained RNNs will provide an intelligent way to obtain information about a page's access count on a scheduling interval. Then utilizing that information application pages will be ordered and placed appropriately.

Low training overhead: Per Page Prediction comes with training a different RNN model per page. This is similar to having a single RNN model the makes predictions across all application pages, when the total number of pages is in the order of hundred thousands, since the input problem size typically remains the same. However, as it was observed earlier not all application pages are critical to application performance (huge empty spaces in the address space in figure 4.9). Besides, that is the reason clustering techniques of the address space into memory regions were implemented by Hashemi et al. [8] In a similar manner the number of RNN models and overall training time can be significantly reduced if we only train individual RNN models only for application-critical memory pages.

High Prediction Accuracy: The maximum number of accesses per epoch of a single page is probably not going to exceed the order of hundreds. Of course, that is heavily dependent upon the epoch duration and the hotness of the page. It is safe to say that the maximum number of page accesses per epoch is orders of magnitude less than problem space the intuitive approach (across pages predictions) needed to capture, normalize and predict. Thus, this output value range is probably more suitable for RNN training, and will most certainly lead to high prediction accuracy.

Chapter 5

Proposed Page Scheduler Architecture

This chapter provides a detailed description of a Page Scheduler for hybrid memory systems, which leverages the existing state-of-the-art data management solutions. On top of that, Page Scheduler optimizes application performance by delivering machine intelligence based placement decisions for a chosen subset of pages.

Critical Metrics

Before moving onto designing the components of the Page Scheduler there are several critical metrics that should be explored and considered. Analyzing application traces can provide us with deep insight into the parameters that influence application's performance.

Benefit Per Page

We want to design a Page Scheduler which uses both the existing state-of-the-art data management approach and Machine Intelligence for a selected number of pages. One subset of pages will be handled by the state-of-the-art solution which is pretty lightweight and straightforward to implement. The second subset of pages will be handled by deploying a Recurrent Neural Network model for each page in an effort to achieve high prediction accuracy, which would have been impossible to obtain using the state-of-the-art approach. However, choosing the amount of pages that need *smart* placement is an intricate task. We can not deploy a Recurrent Neural Network for every single page, as it is fairly obvious that the resource overhead would skyrocket.

After analyzing a few application traces we observed that there is no need to deploy a huge amount of Recurrent Neural Networks. We can achieve significant performance improvement when only deploying Neural Networks for a small subset of pages.

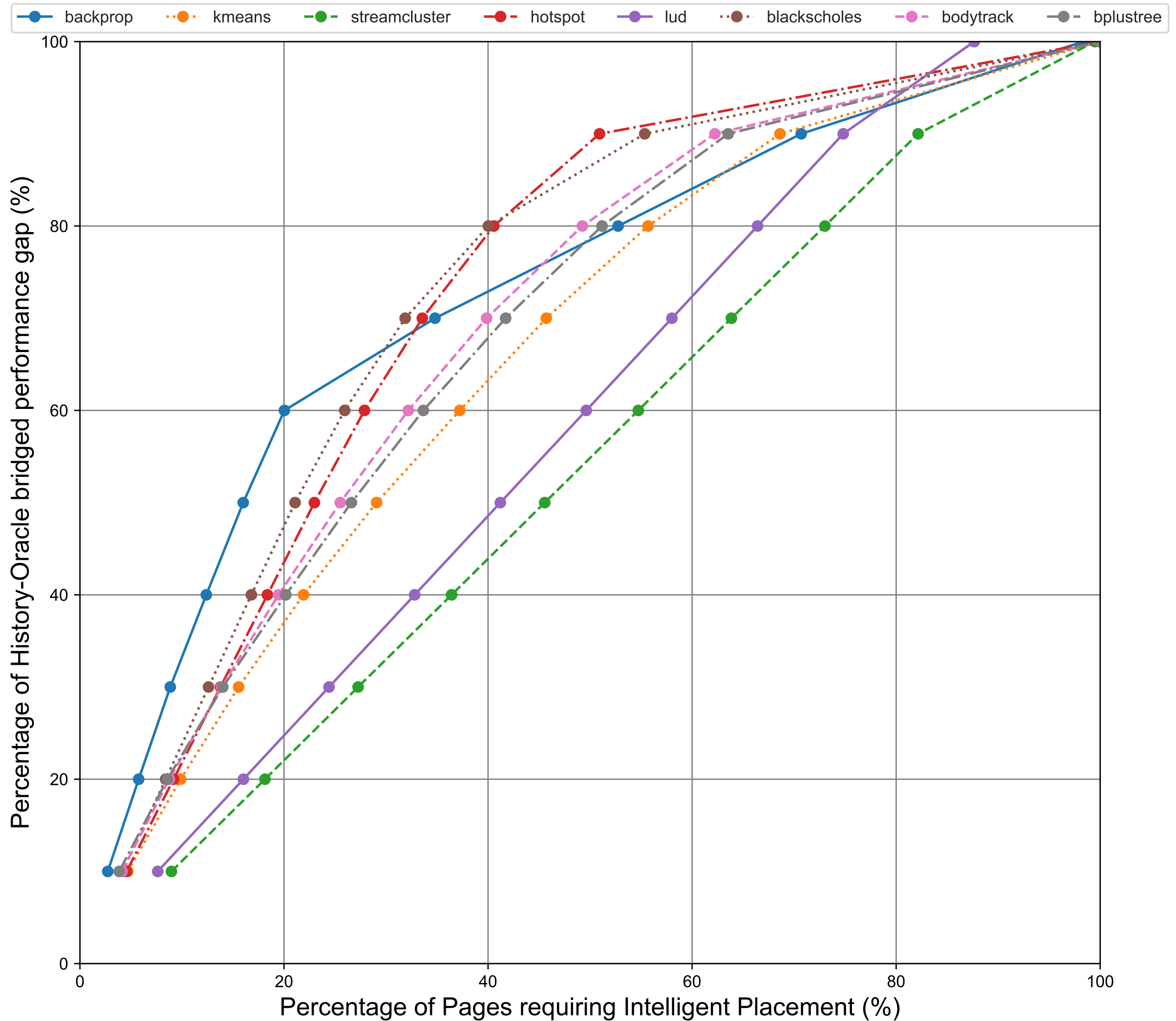


Figure 5.1

In Figure 5.1 we can observe the relationship between the percentage of pages placed across memory components using Machine Intelligence and the obtained performance speedup compared to solely using the state-of-the-art (history-based) approach. It is fairly obvious that in most cases this relationship doesn't follow a strict linear pattern. For instance, as far as the backprop application (rodinia 3.1 suite) is concerned, we can bridge the performance gap of the history-based and the oracular scheduler by 60% , by only handling 20% of the pages in an *intelligent* way. This seems to be the case for most of the workloads we studied. For the majority of them, we can achieve a significant performance speedup (30% , 40%) without having to deploy RNN models for a huge amount amount of pages. With an intelligent placement of a relatively small subset

of pages 10% or 15% of the total amount of pages used by the application, a significant boost in DRAM hitrate can be achieved.

Migration Frequency

There is certainly a need to explore how the frequency of the Page Scheduler would affect performance. The Scheduler we wish to construct would periodically migrate pages across the different memory components. We need to determine the duration of this period, so that we can achieve a high DRAM hitrate without migrating pages too often causing unnecessary bus congestion. However, choosing a proper migration frequency is a complex task.

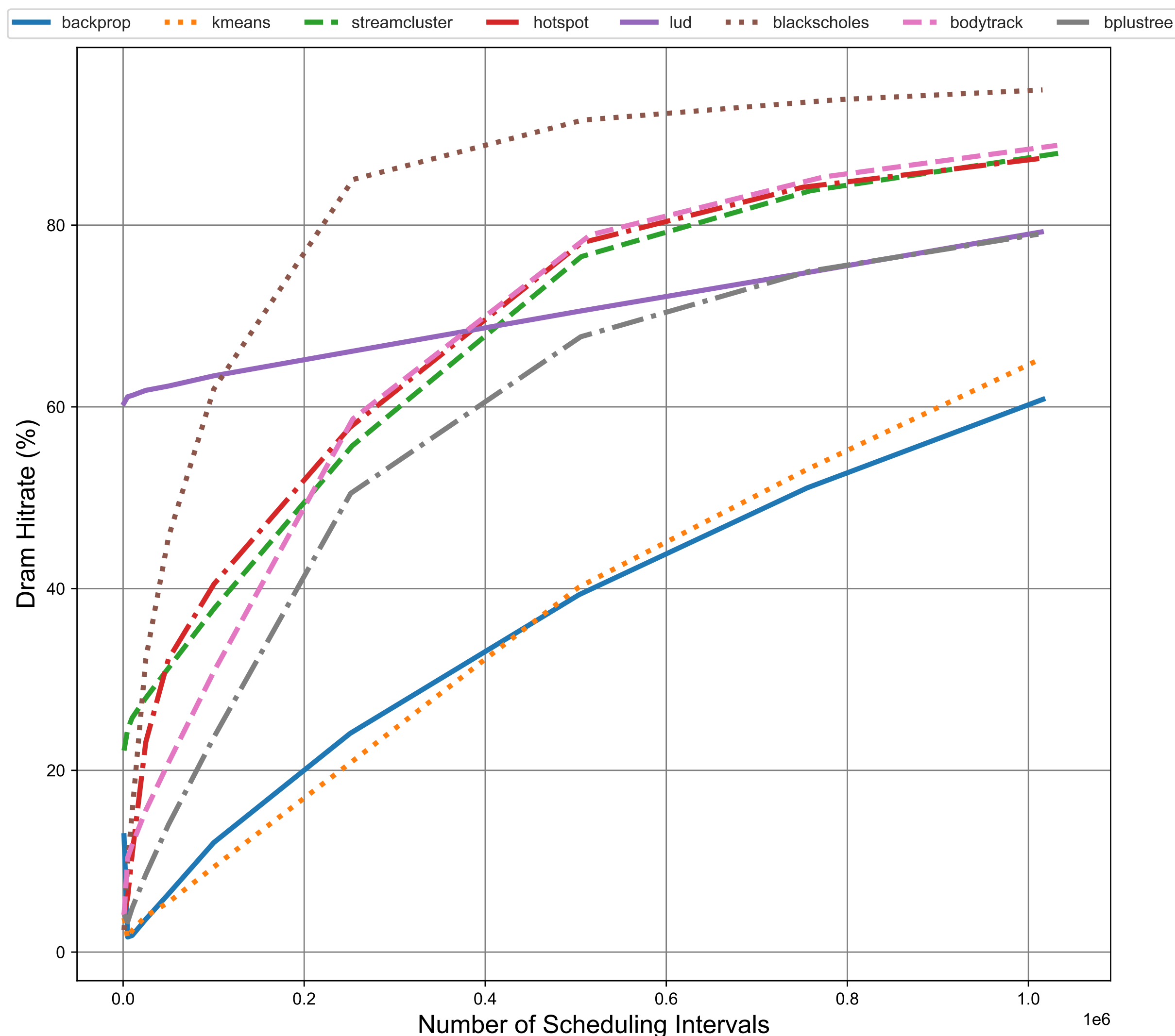


Figure 5.2

In figure 5.2 we can observe how different migration frequencies affect DRAM

hitrate. The scheduler used in this example is the naive history-based scheduler, which draws information exclusively from the current period and makes predictions about the page accesses for the next one.

It is pretty clear that choosing a short period (i.e. scheduler making placement decisions more frequently) results in a significantly higher DRAM hit-rate, despite the fact that the placement policy remained unchanged. Most of the workloads appear to behave in a similar manner. Ramping up the scheduler's frequency seems to have an immense impact to DRAM hit-rate for most of them. That is obvious considering the exponential trend line most of the workloads follow. Only lud workload's hit-rate seems to follow a linear trend, but that is totally expected since the hit-rate was relatively high even for low scheduling frequencies. However, choosing a really high migration frequency to achieve a high DRAM hit-rate is practically impossible, due to the fact that the frequent inference of the page scheduler as well as the huge amount of Page Movement across memory components would probably cancel out the performance boost acquired from the high DRAM hit-rate. Page migration and the scheduler's inference latency are certainly not trivial in real world scenarios. Therefore, these parameters should be also taken into account for our design.

Page Scheduler Overview

The proposed Page Scheduler is inferred on every scheduling epoch and is obliged to take some specific actions.

- Identify performance-critical application pages through the Page Selector component

- We now have two subsets of pages. The first one that has the critical Pages that demand special handling, and the second one with the remaining pages. An individual Long Short Term Memory network is trained for each critical page, in order to make predictions about the page access counts for the following scheduling epoch. For the rest of the pages, a History-Based approach is used. Thus, it is assumed that they preserve their access counts in the following epoch.

- We have now accumulated the per page access counts for the next scheduling epoch. Then pages are ordered in descending predicted access frequency order. The most frequently accessed are then allocated to DRAM until the capacity is full, while the rest of the pages are placed to the slower NVM.

Now, an overview of the proposed Page Scheduler is presented. The following

figure briefly describes the structure of the Page Scheduler.

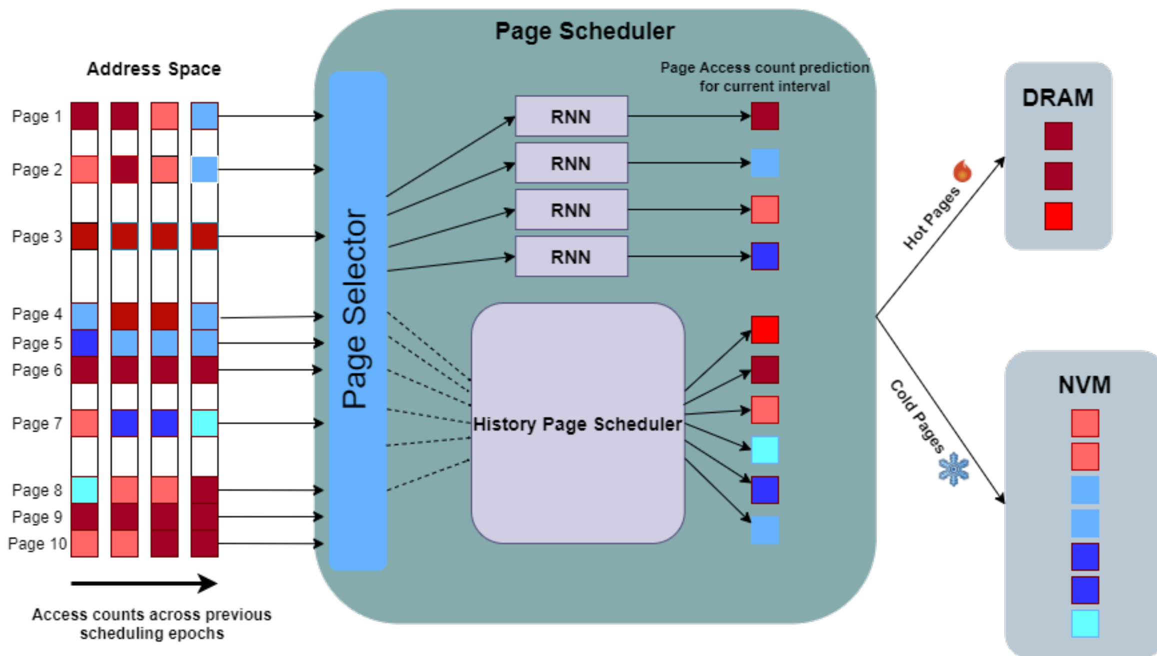


Figure 5.3

To extensively comprehend how the page scheduler operates, it is sufficient to observe the Figure 5.3. On the left part of the figure we can see the pages accessed by an application. Each column represents a period and each box represents a Page. A box with a *deep red* color means that a page in that period was accessed frequently. And boxes with blue-ish colors means that they were not accessed frequently. For example Page 4 was rarely accessed on the first period. Then for the 2nd and 3rd it was frequently accessed and then for the 4th period it went back to being rarely accessed. Now having that information we want to make decisions about the next period. What is going to be the behavior of page 4 in the following period? Is it going to be hot or cold? A plain history-based page scheduler would predict that it will have the exact same behavior as it did on the previous period. Therefore it would predict that it would be rarely accessed in the next period. The Page Scheduler proposed has a page selector component. Page selector component tries to identify which pages require special attention. For example in our scenario the page selector might decide that making a decision about page 4 using the history based approach is not sufficient and eventually will cause performance implications. Therefore the page selector will choose Page 4 as one of the Pages that need special attention. We make predictions about these particular pages using LSTMs (Recurrent Neural Networks) in order to achieve a higher prediction accuracy. In this particular scenario depicted in Figure 5.3 we can see that the Page selector decided that four of the pages require special attention and the other six don't.

That's why the predictions for those four pages are handled by four individual LSTM networks and the predictions for the remaining 6 pages are handled by the history page scheduler. Then, the pages are placed into the two different components according to those predictions.

Page Scheduler Components

The components of the proposed scheduler are clearly depicted in Figure 3.5, but they are not thoroughly described. At this point, we will dive into the details of the Page Selector and the Access Count Predictor components used in the proposed implementation.

Page Selector

The Page Selector is probably the most important component of the Page Scheduler. As it was mentioned in **??**, not all Pages have a strong effect on application's performance. Many pages are properly placed when using the lightweight history based approach. Thus, those pages do not need Machine Intelligence. On the other hand, there is a subset of pages that definitely require a more intelligent management. Therefore, an explicit segregation of the Pages should be performed. Its Page Selector's responsibility to distinguish which Pages require Machine Intelligence and which don't.

Ideally, if unlimited computing resources were available, every single Page would be managed using Machine Intelligence as shown in the following Figure.

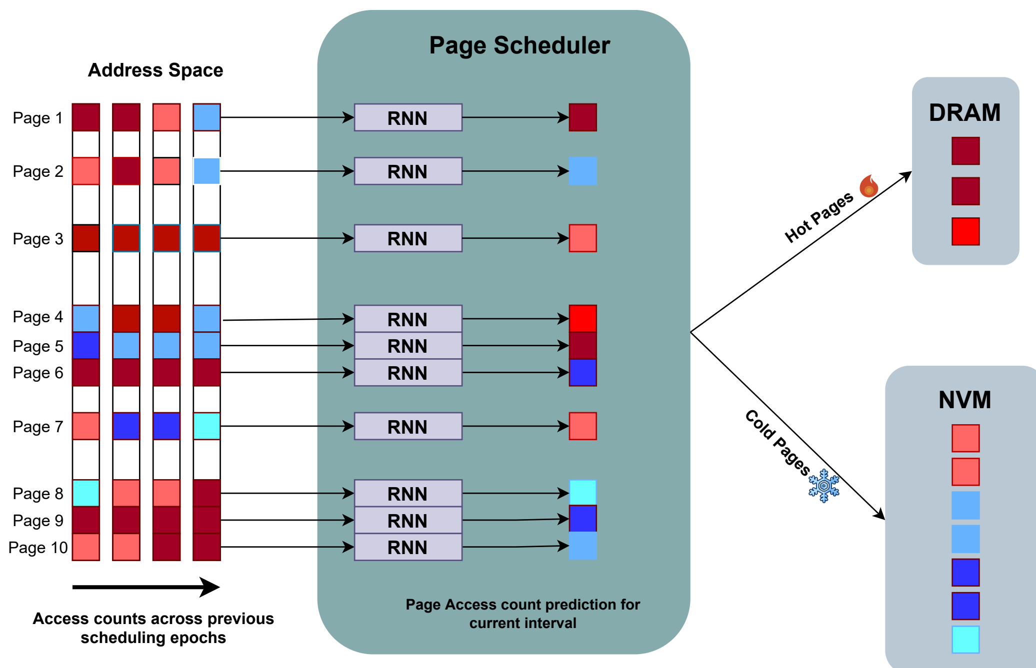


Figure 5.4

That way, by deploying a single RNN for each application Page, maximum prediction accuracy would be achieved. Obviously, this is totally unrealistic and certainly not scalable if we consider that HPC and Big Data applications can have millions of Pages.

As it was mentioned in 5.1, not all Pages have a strong effect on application's performance. Many pages are properly placed when using the lightweight history based approach. Thus, those pages do not need Machine Intelligence. On the other hand, there is a subset of pages that definitely require a more intelligent management. This leads to the conclusion that, a realistic implementation would be to apply Machine Intelligence techniques (RNNs) only on the subset of Pages whose timely DRAM allocation brings significant performance improvement. For the remaining Pages, the lightweight history-based solution would be incorporated.

This approach involves the explicit segregation of the Pages into two subsets according to their effect on performance. **Page Selector** is responsible for distinguishing which pages require Machine Intelligence and which don't. To divide the Pages into those two subsets, Page Selector takes into account two metrics for **every** page.

1. The amount of times a Page has been accessed while taking into consideration the read and write latency asymmetry in NVM module. Clearly, frequently accessed Pages have the most significant impact on performance and therefore, their proper management should be prioritized.

2. The ability of the history-based scheduler to place that Page correctly. If the lightweight History scheduler is able to properly place that page across the different memory components, there is no need to apply resource intensive Machine Intelligence techniques for that particular page, since there would be no additional benefit from a more accurate prediction. However, if the number of misplacements of that page from the history-based scheduler is not trivial; namely that page should have been placed in DRAM but it was falsely placed in NVM instead from the history scheduler, a Recurrent Neural Network should probably take over and manage the placement of that Page.

These two metrics are combined in the following formula for every Page X across scheduling periods i=0...N :

$$\text{Profit}(x) = \sum_{i=0}^{N} Accesses_i(x) * Misplacement_i(x) \tag{5.1}$$

$Accesses_i(x)$ is the amount of times Page **x** was accessed during the **i-th**

scheduling interval and is described as follows :

$$Accesses_i(x) = 3 * Writes_i(x) + Reads_i(x) \tag{5.2}$$

and $Misplacement_i(x)$ is a function which is defined:

$$Misplacement_i(x) = \begin{cases} 1 & \text{if Page x was misplaced during Period i} \\ 0 & \text{if Page x was properly placed during Period i} \end{cases}$$

A sample workflow of Page Selector component can be seen in the figure 5.5. Page Selector component utilizes models to estimate application performance by simulating the page scheduling process given of course a specific Hybrid Memory System Configuration as shown in the figure below. First, it generates an application runtime estimate when using history-based scheduling. This way, we capture information about the number of times each page was misplaced due to inaccurate Page Hotness estimation when using solely historical information. This information is then coupled together with the Page Hotness into the Profit factor as described above and the Page priority for machine-learning based management is determined. Now that the pages have been ordered, a performance curve similar to figure 5.1 generated and the smallest subset of pages that delivers acceptable performance is selected for RNN training.

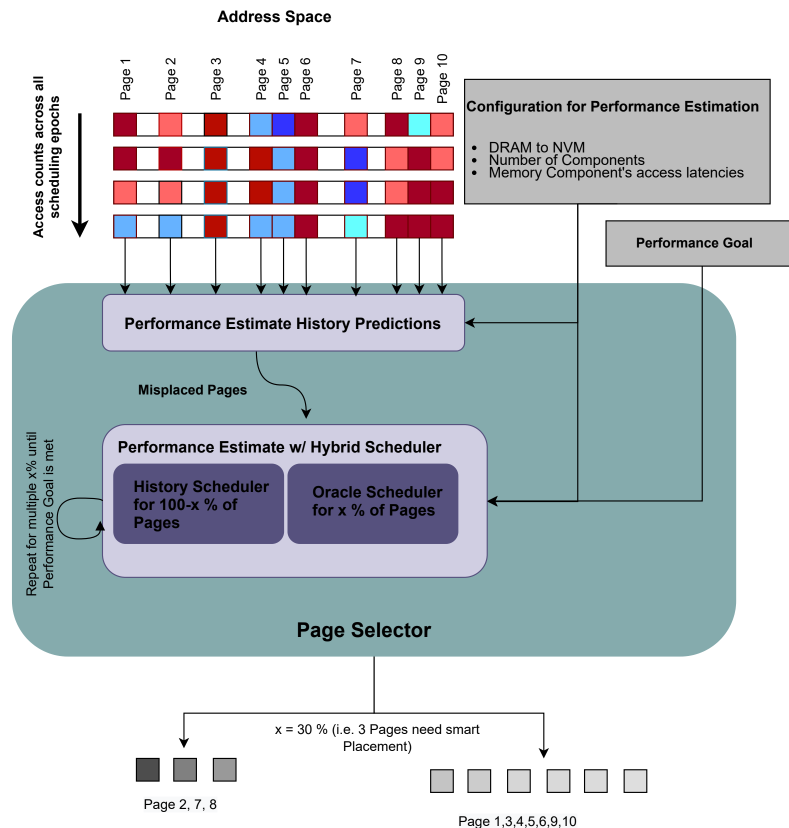


Figure 5.5: Page Selector component used to identify the subset of Pages that require Machine Learning based management

Access Count Predictors

Important components of the Page Scheduler we wish to construct are the Access Count Predictors. The functionality of the Access Count Predictors is pretty straightforward. After every scheduling interval, the Page Scheduler should estimate how many times every page is going to be accessed within the following interval. As it's been aforementioned, a subset of Pages require intelligent management and thus **RNN-based Predictors** are going to be used, while a **History-based Predictor** will make access count predictions for the remaining pages.

History-based Predictor

The history based Predictor has been already referenced a lot throughout this Thesis. It uses solely information of the previous scheduling interval to make predictions about the one that follows. Its implementation is pretty straightforward and it is widely explored in Computer Architecture concepts such as TLB or Cache-line prefetchers. A simple overview of a History-based Access Count Predictor can be seen in Figure 5.6.

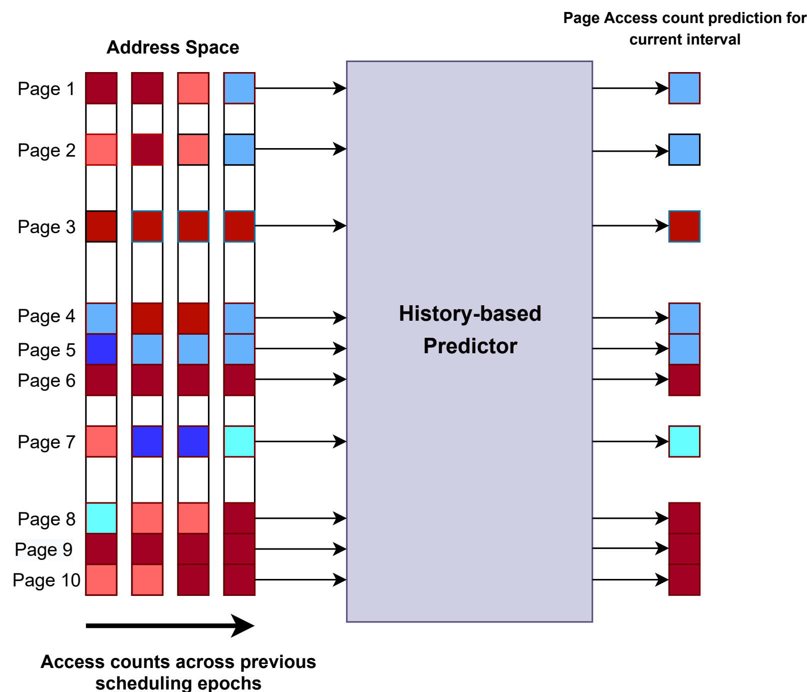


Figure 5.6: History-based Predictor: Predicts that a Page will be maintain the same amount of access within the next scheduling interval

RNN-based Predictor

For a selected subset of Pages, a different RNN model will be used for every page. Having a different model per page, when the total number of pages can

be in the order of hundred thousands, is similar to having a single RNN model that makes predictions across those pages, since the input problem size remains the same. For our models, we create networks based on LSTM cells. We stack two LSTM layers followed by one Dense layer. The simplified layout overview can be seen in Figure 5.7.

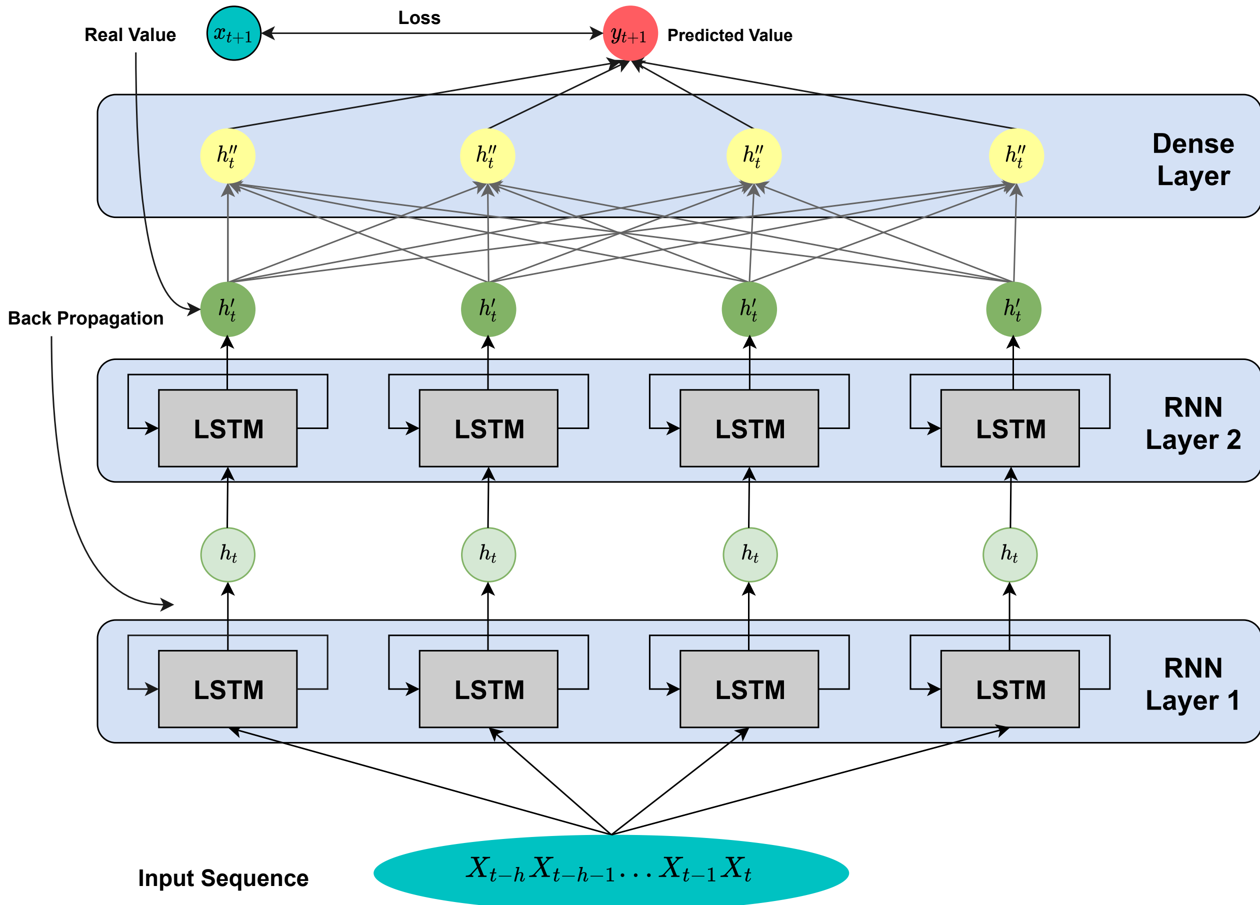


Figure 5.7: RNN-based Predictor: has two layers of LSTM cells followed by a dense layer

The specifics of the input sequence manipulation, and the hyper-parameters chosen for the models thoroughly described in Chapter 6 with other implementation details.

DRAM Eviction Policy

An essential part of a Page Scheduler we have not touched upon yet is the DRAM-page eviction policy. We need to come up with a way of deciding which element to evict.

Probably the most common and broadly used eviction policy is LRU (or Least Recently Used) which is really popular as a Cache Eviction Policy as well. Practically having an LRU policy means that if DRAM size has reached the maximum allocated capacity, the least recently accessed objects (pages) in DRAM will be evicted.

Our Page Scheduler will utilize an **enhanced-LRU** eviction policy. We will use a clustering technique of the address space into memory regions similar to *Hashemi et al* [8]. We observed that the distribution of addresses across the whole address space is very sparse, as it is depicted in figure 4.10. We opt to cluster the different addresses into discrete regions. The popular K-Means clustering alogirthm is used. The number of clusters is chosen manually based on inspecting the distribution of addresses histograms on the whole address space. For example based on the figure 4.10, after observing the global distribution of addresses of the *backprop* benchmark from rodinia 3.1 the address space will have three address clusters. After clustering the address space every page will have a distinct **clusterID** that will be used during the DRAM page eviction process.

During every scheduling epoch, we will maintain information about the most *active* address clusters. Pages that belong to the most active clusters will be prioritized to retain their position in DRAM during the eviction process. That way, we try to take advantage of the spatial and temporal locality of the data, based on the observation that a cluster that is highly active during a scheduling epoch will probably remain active within the scheduling epoch as well. Of course, the LRU policy is still the biggest contributor to the eviction process but now is combined with extra information that lead to better eviction decisions.

Chapter 6

Technical Implementation

Having formally defined the Page Scheduler in the previous chapter, here we describe the details of the technical implementation of our algorithm in length. Firstly, we briefly describe the set of the applications that were used for evaluation, and how the collection of memory access traces of those applications was realized. Then, we provide details about the lightweight Hybrid Memory System Simulation that was constructed in order to evaluate the Page Scheduler. Finally, we present all the details of the Neural Networks' layout, hyper-parameters and input data.

Benchmark workloads

We used a particular collection of programs for the evaluation of our Page Scheduler. Those programs span across domains with representative computation kernel and stress different components of the system. Some of the workloads used are from the PARSEC [17] and others are from Rodinia 3.1 [16] using the simlarge input sizes and the default input data sizes respectively.

Lud *(rodinia 3.1)* belongs to the domain of **Linear Algebra** : LUD (LU Decomposition) is an algorithm to calculate the solutions of a set of linear equations. The LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix. This application has many row-wise and column-wise interdependencies and requires significant optimization to achieve good parallel performance. LU Decomposition exhibits significant inter-thread sharing and row and column dependencies. [49]

Backprop *(rodinia 3.1)* belongs to the domain of **Machine Learning** : Backprop is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. The application is comprised of two phases: the Forward Phase, in which the activations are propagated from the input to the output layer, and the Backward Phase, in which the error between the

observed and requested values in the output layer is propagated backwards to adjust the weights and bias values. [16]

K-means *(rodinia 3.1)* belongs to the domain of **Data Mining**: K-means (KM) is a clustering algorithm used extensively in data mining. This identifies related points by associating each data point with its nearest cluster, computing new cluster centroids, and iterating until convergence. [16]

Hotspot *(rodinia 3.1)* belongs to the domain of **Physics Simulation**: HotSpot (HS) is a thermal simulation tool used for estimating processor temperature based on an architectural floor plan and simulated power measurements. [16]

Bplustree *(rodinia 3.1)* belongs to the domain of **Graph Theory**: BPlustree is an application that traverses B+trees. B+Tree represents sorted data that allows efficient insertion and removal of graph elements. [49]

Blackscholes *(PARSEC)* belongs to the domain of **Finance**: The blackscholes application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE) [17]

Bodytrack *(PARSEC)* belongs to the domain of **Computer Vision**: The bodytrack computer vision application is an Intel RMS workload which tracks a 3D pose of a marker-less human body with multiple cameras through an image sequence [17]

Streamcluster *(PARSEC)* belongs to the domain of **Data Mining**: This RMS kernel was developed by Princeton University and solves the online clustering problem: For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared distances (SSQ) metric. Stream clustering is a common operation where large amounts or continuously produced data has to be organized under real-time conditions, for example network intrusion detection, pattern recognition and data mining. [17]

As far as the memory footprint of those applications is concerned, it is in the order of couple of hundreds of MBs. Having such a significant memory footprint is of utmost importance, since we need to capture the use case where the data will span across multiple memory components due to DRAM capacity

limitations.

Collecting Memory Access Traces

One the most important parts of our technical implementation is the collection of the memory access traces for every single workload mentioned above. For each application, we need detailed traces of the data accesses that missed the Last Level of Processor's hardware caches (**LLC miss**) and thus resulted in main memory accesses.

For the workloads, we collected traces for memory accesses that miss the last level cache on a system with Intel(R) Xeon(R) Gold 5218R CPU clocked at 2.10GHz. As it was previously stated in section 3, unfortunately there is not a straightforward way to obtain those traces. As of now, there is not hardware support to obtain information about the data accesses that caused an LLC miss. For that very reason, we considered *Intel Pin 3.13-98189* [14] to acquire reliable traces, due to the fact that Pin is a dynamic binary instrumentation framework that enables the creation of dynamic program analysis tools. This allowed us to build a custom tool that would perform program analysis at the application's runtime.

For our custom pintool, we performed binary instrumentation on instruction-level granularity. Using a pin cache simulator consisting of L1-data , L1-Instruction, L2 and L3 caches of sizes that are representative of a real Hybrid Memory System, we filtered every instruction during the execution of every application and only retained information about those who resulted in a Main Memory access. For our custom pin cache simulator's cache sizes we used as reference the a hybrid memory system with the following specifications:

- L1 Data cache : 1.3 MiB
- L2 Instruction cache : 1.3 MiB
- L2 Cache : 40 MiB
- L3 Unified Cache : 55 MiB

The information included for each individual access has the following format:

Thread ID[1] , Timestamp , Operation , Virtual Address

For the purpose of our analysis we extract the 4 KB virtual page ID, that corresponds to the virtual memory address accessed and we group memory accesses into scheduling epoch interval similar to kleio [9].

[1]only single threaded applications were considered due to the lack of support provided by Pin

Hybrid Memory System Simulator

Our objective is to assess the performance of the Page Scheduler we designed. Ideally, the whole evaluation process would haven been conducted in a real Hybrid Memory System, but as of now this is nearly impossible. The lack of a system level API to migrate memory pages across the different memory components, combined with the fact that there is not an unanimously agreed and established process of capturing main memory accesses lead us to use a custom lightweight Hybrid Memory System Simulation.

We simulate a hybrid memory system that contains two different memory components. It contains a fast one (i.e. DRAM) and one with a higher access latency (i.e. NVM). The persistence attribute of NVM is not taken into consideration in our simulating environment, since we suppose that NVM is configured in Memory-Mode and serves as an extension of DRAM.

The capacity of our memory system is assumed to be the application's memory footprint similar to [9]. The ratio between NVM and DRAM is configurable in our simulator. For example when we instantiate our Simulator with DRAM to NVM ratio that is equal to 1:4, it means that DRAM will have a restricted capacity and will be able to host no more than 1/4 of the application's pages, while the other 3/4 will be serviced by the slower NVM.

Our simulator is constructed appropriately, so that it can accommodate our Page Scheduler. It is mainly used to obtain information about DRAM hit-rate. Specifically after initializing the HMS-Simulator and our Page Scheduler for a given Memory Access Trace we are given information about how many and which Page Accesses were serviced by DRAM and which were serviced by the slower NVM. Apart from DRAM hit-rate we are able to collect data that will assist on extrapolating the application runtime. Also, the DRAM page eviction policy is configurable and of course, we assume dedicated DMA engines that allow seamless page migration similar to [50, 9, 51]. In figure 6.1 we can see an abstract overview of the simulator we constructed.

Recurrent Neural Networks Details

Before diving into the detailed architecture (network's layers, loss function etc.) of the Neural Networks, we need to elaborate on the required manipulation of the data that will be used as input for training and validation.

Neural Network Input

As described earlier we will train and validate a Recurrent Neural Network for every selected page based on its effect on performance. We followed the

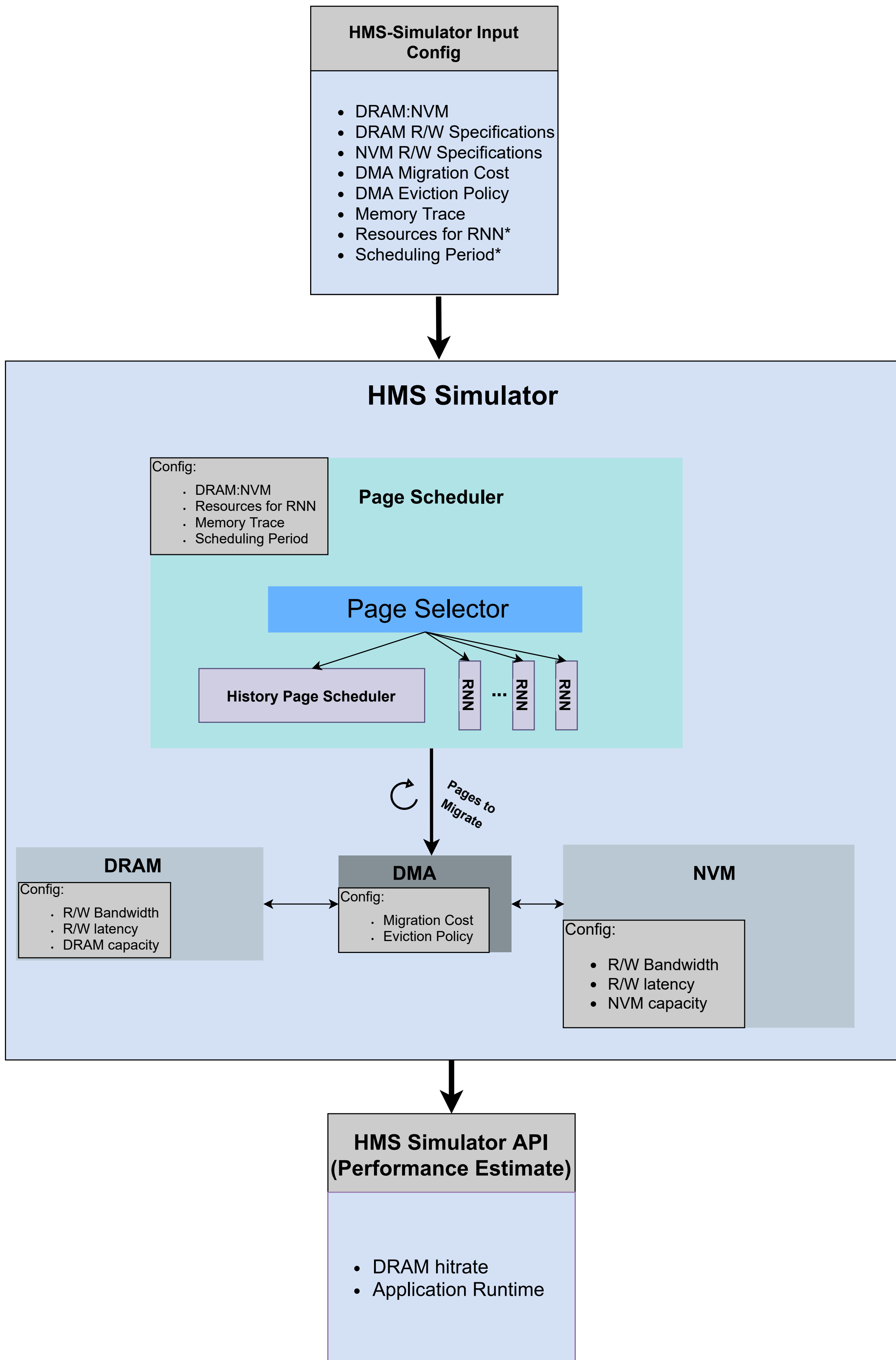


Figure 6.1: Python Profiler used to analyze the obtained Memory Access Traces

Per-Page approach and avoided making predictions across all pages for various reasons which are mentioned in section 4.

As far as the input of the RNN is concerned, we need to utilize as much information as possible in order to achieve high prediction accuracy. Unlike similar solutions that are proposed in literature [9], we cannot only use a sequence of per-page access counts during consecutive scheduling epochs because a lot of crucial information will be thrown away. Achieving high prediction accuracy based solely on the access counts of a **single** page is really difficult. To construct per-page models that have decent prediction accuracy, we need to use data that are representative of the page's behavior in relation to the application as a whole. This was also clearly pointed out by Hashemi *et al.* [8], where both **Program Counter** and **Delta** information was required to obtain accurate models. On account of this, for every single Recurrent Neural Network we use the sequence of access counts of our page of interest combined with the sequence of access count of its **8** closest page-neighbor during consecutive scheduling epochs. This way we can utilize information that encapsulate the spatio-temporal locality of our application without inducing too much computation overhead.

The input of a Recurrent Neural Network for page **x** is given by the following formula

$$\text{Input}_i(x) = \sum_{j=-4}^{4} \frac{1}{|j+1|} * Accesses_i(x+j) \tag{6.1}$$

The output of the per-page Recurrent Neural Network is the predicted number of accesses the page will receive during the next epoch. Our models do not need to be absolutely accurate when it comes to determining the exact amount of times a page is going to be accessed. They only need to be accurate enough so that our Page Scheduler can determine the hotness order across all pages. Thus, there is certainly room for the prediction to slightly differ from the actual number of accesses, provided that it will not influence the hotness order of the page and, by extension its placement decision on the particular scheduling interval.

We normalize the input sequence 6.1 between 0 and 1, since Recurrent Neural Network (and especially LSTMs) are performing much better in this case. Then, we denormalize the data for the final prediction. Different from [8] we do not need to make predictions over distinct integers which would increase the chance of mispredictions. This is not necessary for the purpose of our predictions, since we only need to derive information from the model about the *relative hotness* of a Page.

Neural Network Configuration

A simplified overview of the Neural Network's Configuration can be seen in Figure 5.7.

Hyperparameters

The network consist of two stacked LSTM layers with **256** neurons each, followed by a Dense Layer. The history length is **20**. Thus, the input data series is split in sequences of length 20, on a rolling window fashion. $\frac{3}{4}$ of the dataset is used as a training dataset, while the other $\frac{1}{4}$ is used for validation.

Now, a really important parameter of the Neural Network was the selection of the loss function. Using most of the popular loss function like *mean squared loss* resulted either in poor convergence or poor accuracy due to dataset imbalances. Therefore, a custom **weighted loss function** was implemented that resulted in weighing each label's contribution to the cost function inversely proportional to the frequency of the label. This is the sample code for the custom loss function

```
from keras import backend as K

def loss(y_true, y_pred,weights):
    weights = K.variable(weights)
    y_pred /= K.sum(y_pred, axis=-1, keepdims=True)
    y_pred = K.clip(y_pred, K.epsilon(), 1 - K.epsilon())
    loss = y_true * K.log(y_pred) * weights
    loss = -K.sum(loss, -1)
    return loss
```

The neural network tried to minimize this custom loss value between the predicted and actual values, using the **Adam** optimizer and a learning rate of **0.01**. The training stops if the loss for the validation dataset is not reduced for **30** consecutive training epochs.

Implementation

As far as the implementation is concerned, our RNN models should be compatible with our python-based Hybrid Memory System Simulation. For that reason, we use the Keras high level API [52]. We use the existing implementation for the LSTM neurons, the Adam optimizer, and model training. For the additional hyperparameters that are not explicitly determined above, the default values from Keras were used. The backend execution engine used is Tensorflow.

Chapter 7

Experimental Evaluation

In this chapter, we perform an experimental assessment of the Page Scheduler in order to evaluate its performance. We previously showed (in Figure 5.1) that assuming that we had oracular knowledge of the access counts of even a small fraction of the pages, the performance improvements of an application are pretty significant. Our goal is to approach oracular knowledge of the access counts, using Machine Intelligence. Therefore, in this chapter we will not only evaluate the actual performance improvements our Page Scheduler provides, but the prediction accuracy of the per-page RNNs as well. We showcase how close to the Oracle Page Scheduler (i.e. upper limit of performance) our Page Scheduler can perform. Finally, we touch on the performance boost induced by our **enhanced-LRU** policy described in Section 5 and the Estimated Energy Cost of our Page Scheduling proposal.

RNN Prediction Accuracy

As it's been clearly stated already, a fundamental component of the Page Scheduler are the Per-Page RNNs models. We use them to make better predictions for the access counts of a performance-critical pages in the next scheduling epoch compared to the naive approach of history-based predictions.

We now evaluate the prediction accuracy of the per page RNN training for the application workloads mentioned in section 6. For every workload we deploy RNNs for the **100** most performance-critical application pages as selected by the *Page Selector* (5). We will use the Root Mean Square Error (RMSE) to evaluate the prediction accuracy of our models. RMSE is widely used when it is a priority to emphasize on huge errors by penalizing our model. The RMSE formula is the following:

$$RMSE = \sqrt{(\frac{1}{n})\sum_{i=1}^{n}(y_i - x_i)^2} \tag{7.1}$$

Figure 7.1 depicts the distribution of RMSE (Root Mean Squared Error),

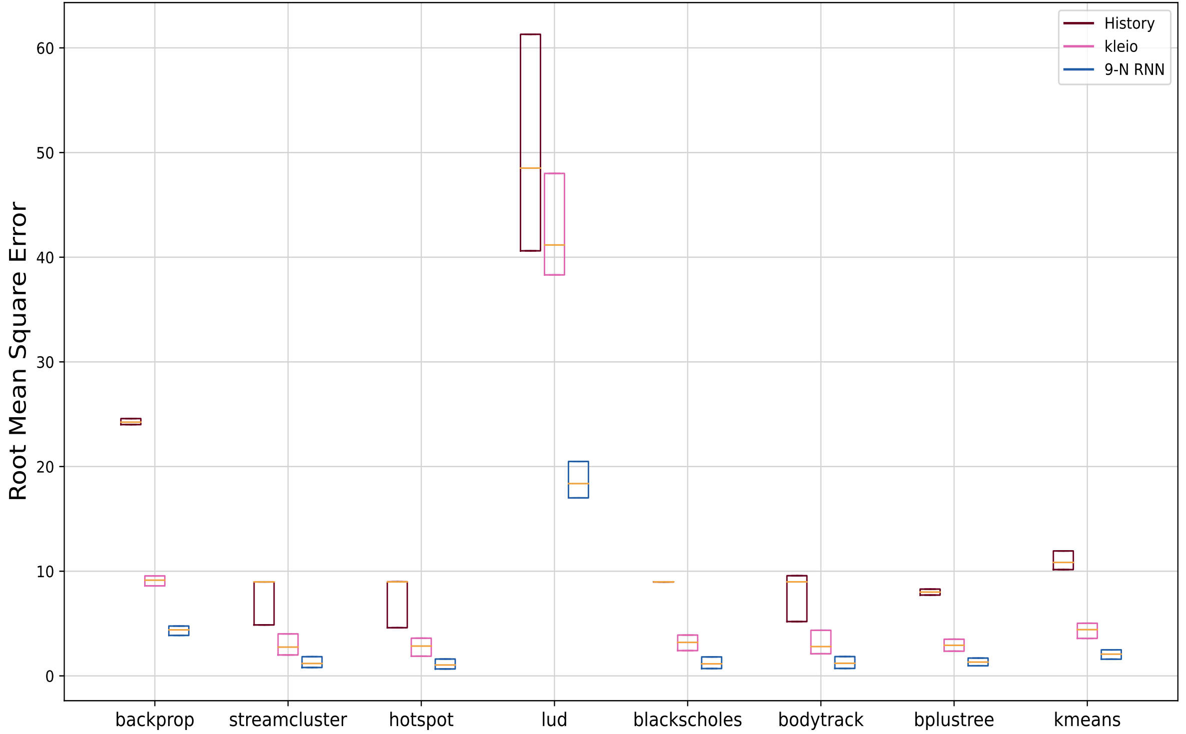


Figure 7.1: Prediction Accuracy of the number of access counts across the scheduling intervals for the selected trained pages. History, kleio's RNNs and our RNNs are used as Access Count Predictors

in boxplot representation, between the per epoch page access counts and the actual values. In our scenario the equation values in eq. 7.1 are as follows:

- n is the number of epochs
- y_i is the actual value
- x_i is the value predicted by our models

For example, mean RMSE of 50 means that the average Root Mean Square Error per epoch per page was 50. Obviously, the value of RMSE is not sufficient enough to evaluate the prediction accuracy of our model. RMSE that is seemingly low might actually result in wrong *Page Hotness* ordering and vice versa. A model with Root Mean Square Error that seems high may not negatively interfere with the global page hotness order and actual page placement. For that reason, we utilize a History Page Scheduler as a baseline evaluation model. We treat the decisions of the History Page Scheduler as predictions and plot the corresponding root mean square error. Obviously, the History Page Scheduler predicts that on the next epoch, a page will receive the same access

counts as to those of the current epoch. Apart from the history page scheduler we compare our implementation to the one proposed by [9] as described in the article. After following as closely as possible the implementation described in the paper, we deploy Neural Networks for the same 100 pages we deployed RNNs and evaluate the prediction accuracy achieved by those models.

After observing Figure 7.1, we understand that our proposed implementation of RNN model which is thoroughly described in section 6 leads to better page access count predictions compared to the naive history-based page scheduler and the scheduler proposed by [9]. Outperforming the prediction accuracy of history-based scheduler was certainly expected. What is quite noteworthy is that the different approach we followed concerning the neural network input may lead to better on-average predictions compared to kleio [9].

Application Performance

As it was aforementioned, the value of Root Mean Square Error of our models cannot be used as the only indicator of the application performance boost achieved by our Page Scheduler. For that very reason, we use a more representative metric which is the DRAM hit-rate (i.e. the amount of requests served by DRAM). DRAM hit-rate of our implementation can be extracted from the Hybrid Memory System Simulation described in 6.1. Obviously, the higher the Dram hit-rate, the more significant the application performance boost, due to the fact that more application requests are served by the significantly faster memory component.

We use the same setup as the one used in 7. We will evaluate the DRAM hit-rate for the application workloads mentioned in section 6. For every workload we deploy RNNs for the **100** most performance-critical application pages as selected by the *Page Selector* (5). The performance is normalized between 0% when all pages are managed by the *History Page Scheduler* and 100% when all the selected pages are managed by an *Oracle Page Scheduler*. In other words, our plot will depict if our Page Scheduler is able to bridge the performance gap between the lightweight approach of a History Page Scheduler and the optimal Page Scheduler with a priori knowledge. We also follow the Page Scheduler implementation description from [9] in attempt to compare it with our implementation. Thus, we also implement **kleio** and deploy RNNs for the first 100 important pages as selected by kleio's Page Selector component.

Figure 7.2 depicts the DRAM hit-rate improvement achieved by leveraging Machine Intelligence based predictions for a selected subset of pages. We can clearly observe for the majority of the workloads we can obtain at least 70% of the possible performance improvement. There are cases, such as *back-*

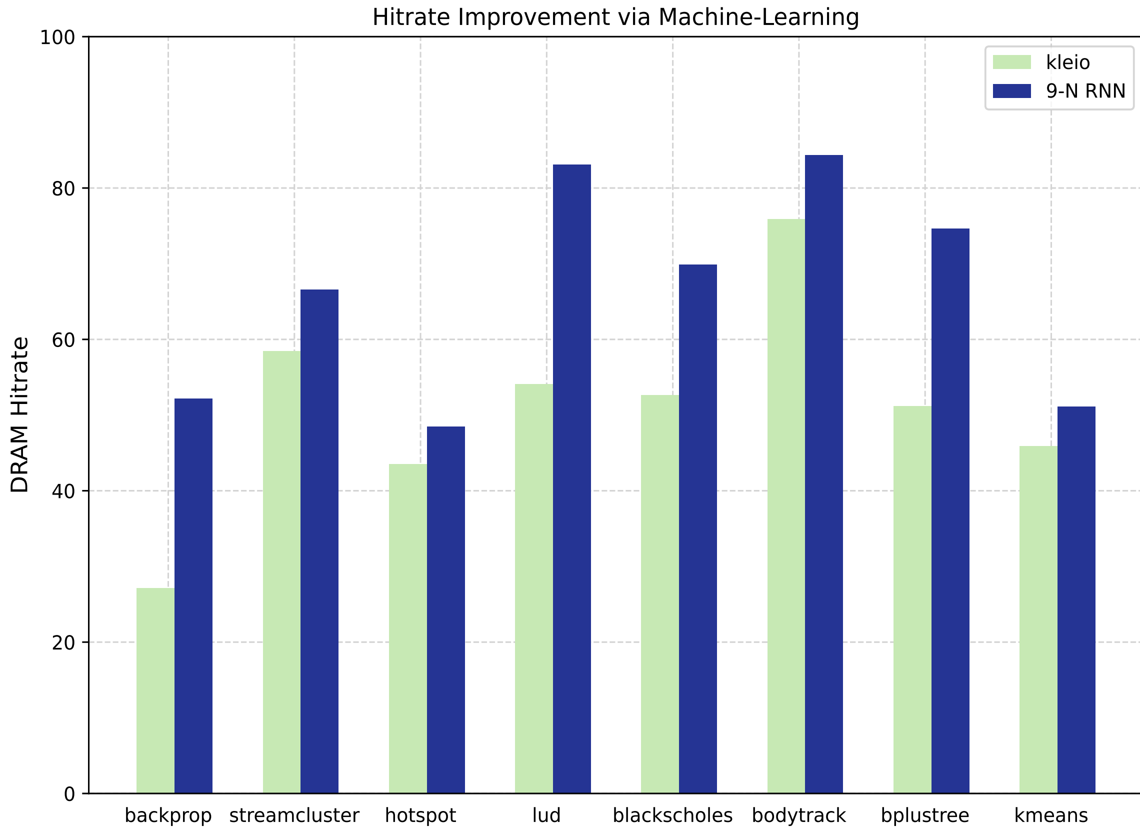


Figure 7.2

prop and *kmeans*, where only 50% of the possible performance speedup is achieved. That probably means that we need to deploy more RNNs in order to achieve a more significant performance boost. What is actually remarkable is that our implementation seems to outperform the *kleio-based* implementation for every workload. In some cases the performance speedup achieved by kleio is quite close to the one achieved by our page scheduler (*hotspot,kmeans,streamcluster,bodytrack*). However, there are cases where the performance gap between our implementation and kleio is more than 15-20% (*lud,bplustree,backprop*). This performance difference is certainly induced by the fundamentally different approach we followed for performance-critical page selection and the actual Neural Network implementation.

Overall, we can clearly observe that the prediction accuracy of the trained RNNs is such that it can deliver application performance similar to what would be possible with oracular knowledge of the page access counts.

Eviction Policy

We now proceed to evaluate another important component of our implementation. We will evaluate the performance boost we can obtain using the proposed enhanced-Least Recently Used policy and compare it with the plain LRU.

Our enhanced LRU policy (section 5) uses a clustering technique of the address space into memory regions. After clustering the address space each page is assigned a distinct clusterID which is used during the eviction process at the end of every scheduling epoch. Pages that belong to a memory region (cluster) that was highly active in the recent scheduling epochs are favored to carry on residing in DRAM. Pages that belong to the other not so active clusters are chosen to be evicted unless they have been accessed in the last scheduling epoch.

For the evaluation process we will use DRAM hitrate as a performance metric which can be easily extracted from the Hybrid Memory System Simulation described in 6.1. The setup is similar to the one described in 7. For every workload we deploy RNNs for the **100** most performance-critical application pages as selected by the *Page Selector* (5). The performance is normalized between 0% when all pages are managed by the *History Page Scheduler* and 100% when all the selected pages are managed by an *Oracle Page Scheduler*. Both History and Oracle Page Schedulers are using a Least Recently Used DRAM eviction policy.

In Figure 7.3 we can observe the performance achieved by our Page Scheduler in two implementation scenarios:

1. Least Recently Used DRAM eviction policy (Gray)
2. Clustering-based Least Recently Used DRAM eviction policy (Red)

It is fairly obvious that for the majority of the workloads using an address space clustering based LRU DRAM eviction policy allows us to obtain a higher DRAM hit-rate. The perfomance boost achieved by the proposed eviction policy is certainly not immense. However, in some cases it surpasses 10% (*kmeans,hotspot*). There are workloads that show only a slight performance enhancement (*bodytrack,bplustree,streamcluster*) and in one scenario (*lud*) the performance deteriorated when using the clustered-LRU policy.

Overall, we can assume that using a more sophisticated eviction policy in DRAM will deliver better application performance. However, there are several complications that should be addressed. For instance, as it is briefly mentioned in subsection 5, we are aware that address space clustering is not trivial.

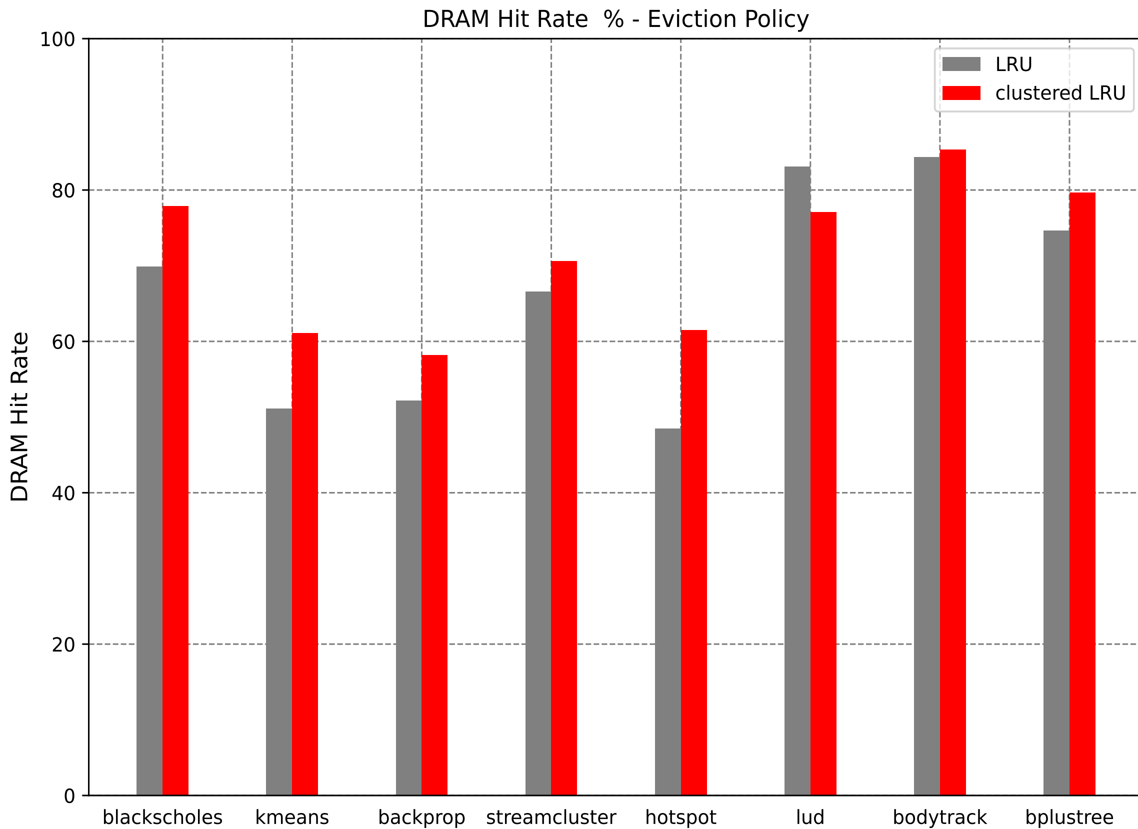


Figure 7.3

Energy Consumption

It is widely known that DRAM not only suffers from low capacity making it unable to meet the requirements for big data applications, but it also has high refresh power consumption which leads to the power usage of DRAM to be incredibly high even when system is idle. On the contrary, Non-Volatile Memory (NVM) besides having a large capacity it also has trivial idle energy consumption. These differences in energy consumption characteristics between DRAM and NVM leave a lot of room for low-power design exploration in Hybrid Memory Systems.

In this section we will evaluate how our implementation performs energy-wise. We also design NoManage, similar to [15], as a baseline candidate, in which no Page Scheduling is conducted. We assume page allocation is random in NoManage and the probability of assigning to NVM for each page is proportional to the ratio of NVM capacity to total memory size. We will calculate the energy cost of **four** Hybrid Memory Systems (excluding the cost for RNN training).

- **NoManage HMS:** A System which does not use Page Scheduler. Pages are allocated once and their placement remain the same during the application execution.
- **History HMS:** A System which uses a Page Scheduler making page placement decision based solely on historic information.
- **kleio HMS:** A System which uses a Page Scheduler. Historic information and Machine Intelligence are combined as described in [9]. RNNs are deployed for the **100** most performance-critical application pages as selected by the kleio's Page Selector.
- **9-RNN HMS:** A System which uses our proposed Page Scheduler. We deploy RNNs (as described in 6) for the **100** most performance-critical application pages as selected by the *Page Selector*.

Following a similar approach as presented in article [15] we will calculate the total Energy Cost for the four Hybrid Memory Systems. Energy consumption is mainly caused by reading and writing operations of the hybrid memory and the idle duration. We will use the workloads described in 6 which have quite different W/R ratios (Table 7.1);and thus help us obtain a good overview of the energy demands of each implementation.

Workload	Ratio
streamcluster	0.19
lud	0.88
backprop	1.05
kmeans	0.75
bplutstree	3.24
bodytrack	3.86
blackscholes	15.3
hotspot	45.7

Table 7.1: Workload Write/Read ratio

Energy Model

Before moving to the evaluation of the four memory systems we have to define the Energy Model we will be using to calculate the Energy cost for every single HMS. We will follow the same Energy model used in [15]. All of the notations used in the following definitions are presented in Table 7.2.

Table 7.2: Notation table for energy related symbols

P_i	A Page in a Hybrid Memory System
C_{NVM}	Energy Cost of page P_i in NVM
C_{DRAM}	Energy Cost of page P_i in DRAM
E_{NVM}^{read}	Reading energy consumption per page in NVM
E_{NVM}^{write}	Writing energy consumption per page in NVM
POW_{NVM}^{idle}	Idle power consumption per page in NVM
r_i	The number of readings for P_i in hybrid memory system
w_i	The number of writing for P_i in hybrid memory system
T	Time Period for page replacement in hybrid memory system
E_{DRAM}^{read}	Reading energy consumption per page in DRAM
E_{DRAM}^{write}	Writing energy consumption per page in DRAM
POW_{DRAM}^{idle}	Idle power consumption per page in DRAM
$C_{NVM--DRAM}$	Energy cost when P_i is replaced from NVM to DRAM
E_{extra}	Extra energy consumption during the migration of P_i

Definition 1. Given page P_i in NVM, C_{NVM} is defined to denote the energy cost during the time period of T

$$C_{NVM} = E_{NVM}^{read} * r_i + E_{NVM}^{write} * w_i + POW_{NVM}^{idle} * T \tag{7.2}$$

where E_{NVM}^{read} is the reading energy consumption of each page in NVM, E_{NVM}^{write} is the writing energy consumption of each page in NVM, r_i and w_i are the numbers of reading and writing P_i in NVM respectively. POW_{NVM}^{idle} is the idle power for each NVM page. In particular, POW_{NVM}^{idle} can be ignored because the idle energy consumption of NVM is much lower than that of DRAM.

Definition 2. Given page P_i in NVM, C_{DRAM} is defined to denote the energy cost during the time period of T

$$C_{DRAM} = E_{DRAM}^{read} * r_i + E_{DRAM}^{write} * w_i + POW_{DRAM}^{idle} * T \tag{7.3}$$

where E_{DRAM}^{read} is the reading energy consumption of each page in DRAM, E_{DRAM}^{write} is the writing energy consumption of each page in DRAM and POW_{DRAM}^{idle} is the idle power of each DRAM page.

Definition 3. Given page P_i , $C_{NVM--DRAM}$ is defined to denote the energy cost P_i replaced from NVM to DRAM.

$$C_{NVM--DRAM} = E_{extra} + (E_{NVM}^{read} + E_{DRAM}^{write}) \tag{7.4}$$

Definition 4. Given page P_i , $C_{DRAM--NVM}$ is defined to denote the energy cost P_i replaced from NVM to DRAM.

$$C_{DRAM--NVM} = E_{extra} + (E^{read}_{DRAM} + E^{write}_{NVM}) \tag{7.5}$$

E_{extra} is the extra energy consumption during migration process in addition to reading and writing consumptions. $(E^{read}_{NVM} + E^{write}_{DRAM})$ means the energy consumption of a page once migrating from NVM to DRAM. In particular, there are some extra energy consumptions in the page migration procedure, such as page wear energy consumption, cache energy consumption and so on. But these extra power consumptions are low enough to be ignored compared with the energy consumptions of page reading, writing and idle.

Evaluation Results

We now use the Hybrid Memory System Simulator described in section 6.1 to extract all the information required in order to use the aforementioned Energy Model with the Energy Parameters mentioned in Table 7.3 for the four Hybrid Memory Systems we wish to evaluate. All the Hybrid Memory Systems we evaluated have two different memory components NVM and DRAM with a 1:8 ratio. In other words, at a given moment only one eighth of the application's memory footprint reside in DRAM and the other seven eighths are located in NVM.

Before presenting the results of the evaluation, we need to mention that the memory footprint of our workloads is quite small (A few tens of MBs). Obviously, with such low memory footprint and short runtime, Leakage Power (Idle) of DRAM is not going to be the main contributor of the total energy consumed of the System. That is because the term: $IdlePower = 451 * \frac{mW}{GB} * T$ is relatively small compared to the energy consumed by the NVM and DRAM operations during runtime. If workloads with bigger memory footprints and longer execution time are examined, we might end up different results than the ones we present. That is also evident in [15] where for small memory footprints all the Page Placement policies performed similarly due to the fact that the main

Table 7.3: Evaluation Parameters of NVM and DRAM [15]

	PCM	DRAM
Writing Energy	418.6 nJ	12.7 nJ
Reading Energy	80.41 nJ	5.9 nJ
Leakage Power (Idle)	4.23 mW/GB	451 mW/GB

contributor to the system's total energy consumption were the write ($418.6nJ$) and read ($80.41nJ$) operations of NVM instead of the Leakage Power of DRAM. However, we can still draw some conclusions from the workloads with relatively small memory footprints. Those conclusions may not translate perfectly to bigger workloads in absolute numbers but they can certainly help us estimate if we are headed towards the right direction.

In figure 7.4 we can observe how the four different policies perform as far as the Energy Consumption is concerned. As expected NoManage performs poorly compared to the Systems which utilize a Page Scheduling policy. What we need to point out is that our Page Scheduler seems to outperform both Kleio and the History Page Scheduler for every workload. This was more or less expected due to the higher DRAM hitrate our implementation achieves; thus both the total amount of operations carried out by NVM and the application's execution time are decreased. Overall, figure 7.4 clearly indicates that utilizing a Page Scheduling policy results in a reduced system power consumption and that our Page Scheduler could be a considerable option for a low power oriented Hybrid Memory System design.

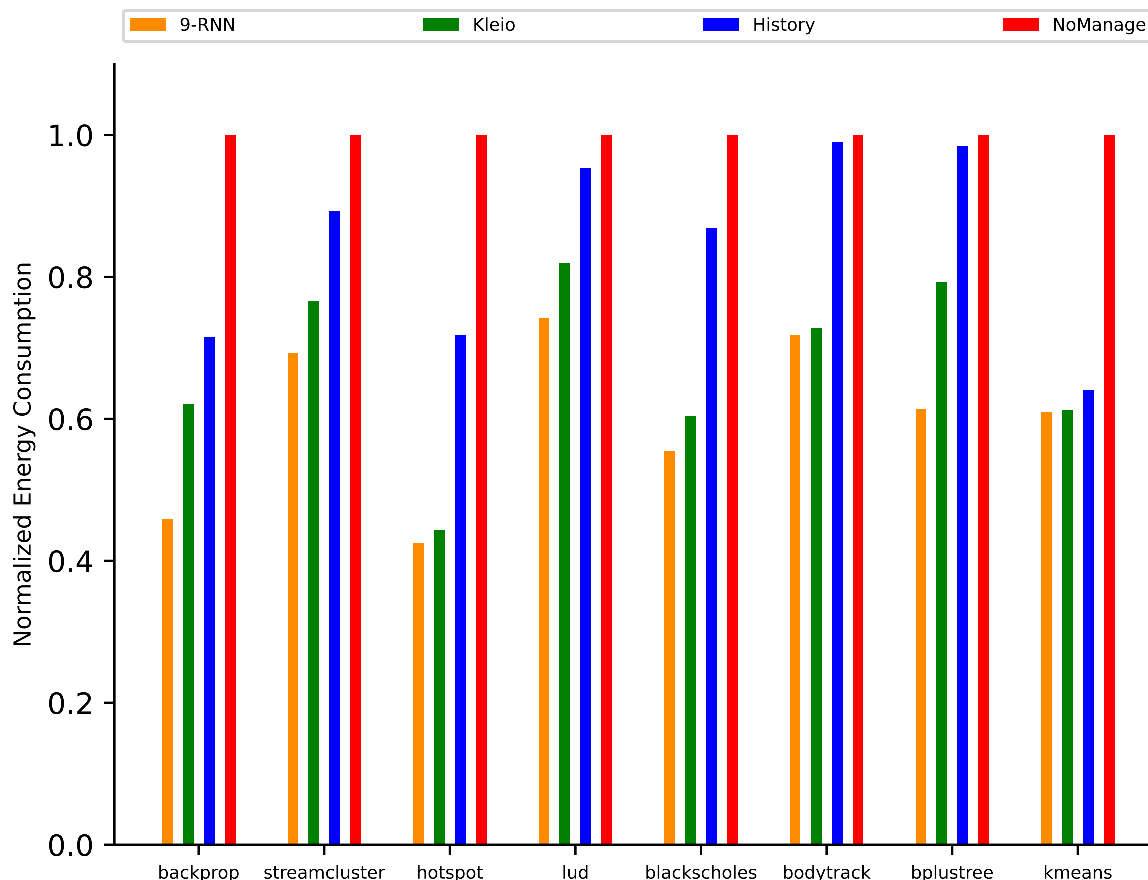


Figure 7.4

Although Figure 7.4 is certainly helpful and indicative of the performance differences between the different design approaches we evaluated, it is also essential to examine why each workload behaves the way it does and determine the primary contributors to the total Energy consumption. On that account, figure 7.5 is provided, which is practically an enriched version of 7.4. This figure provides information about how each memory component contributes to the summation of the system's energy consumption. It should be mentioned that for every workload, the first, second, third and fourth bar is the normalized energy consumption of our page scheduler, kleio, History and NoManage respectively (similar to Figure 7.4). It is clear that for every workload the most amount of energy is consumed by the NVM operations instead of the DRAM Idle Power due to the small memory footprint of the tested applications. It is also important to note that the Page Migration Energy cost does not seem to be rather significant, and that is probably the reason why Page Scheduling results in systems with lower power consumption.

Every application we used for evaluation is quite different from the others. Therefore, it is of utmost importance to examine each one individually, since we can probably draw distinct crucial information from every single one of them.

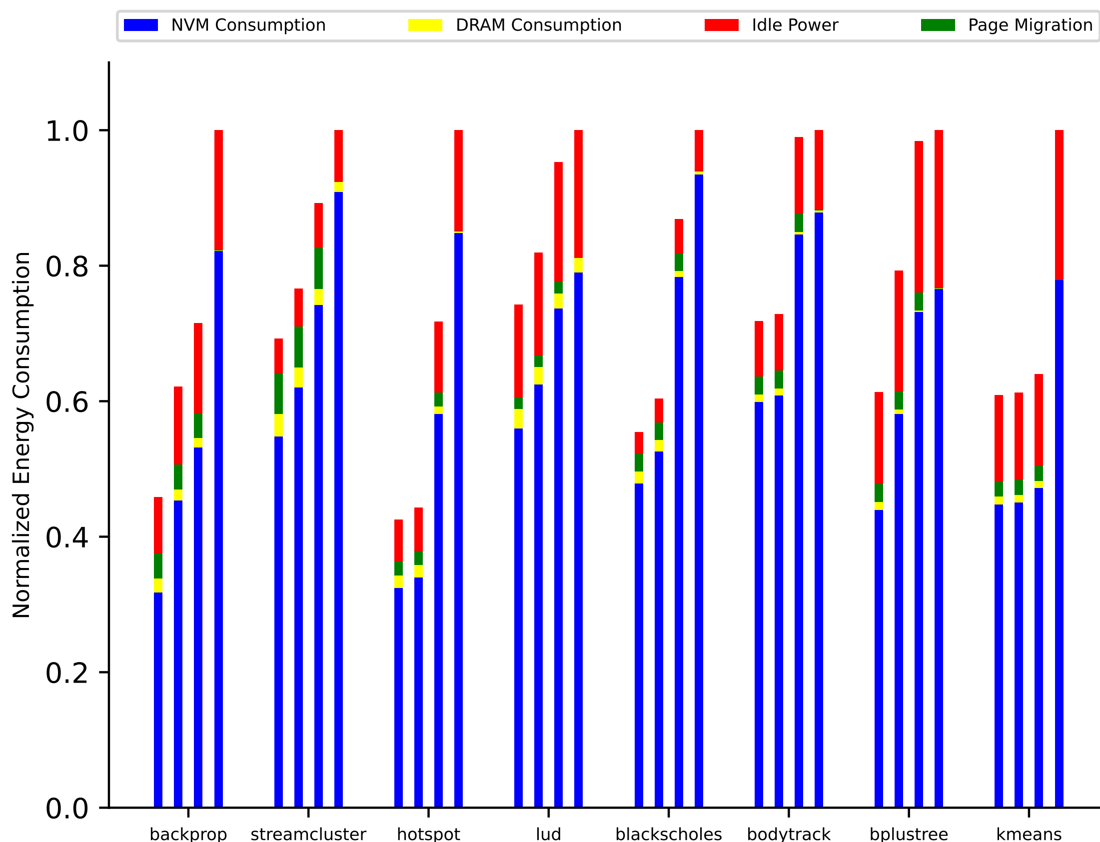


Figure 7.5

Backprop: Backprop's trace acquired has a balanced Read/Write ratio. Meaning that the total amount of Reads and Writes were almost equal. A big gap between History and NoManage can be observed which probably means that NoManage's initial page placement was such that it resulted in low DRAM utilization. Kleio's Energy Consumption is not significantly lower than History, which was expected due to the low DRAM hit-rate increase kleio achieved (seen in Figure 7.2). Our Page Scheduler seems to perform really well energy-wise for this particular workload.

Streamcluster: Streamcluster's trace acquired was read-heavy. That means that write operations were significantly less than read operations. Therefore, we did not expect huge energy consumption benefits, because write operations are the ones that are the primary cause of high energy consumption. We can see that History is not significantly better than NoManage, and kleio's Energy Consumption is relatively close to History but obviously lower. Our Page Scheduler achieves 25% decrease in normalized energy consumption, which is only 5% difference with kleio which is probably due to their performance similarities as showed in figure 7.2.

Hotspot: Hotspot's trace acquired was write-heavy. That means that write operations were significantly more than read operations. There is big gap between NoManage and History, and that is probably because History reduced the number of write requests served by NVM. There is also significant gap between kleio and History, while the Energy Consumption of our Page Scheduler and kleio's are almost equal. What is important to note here is the drop of DRAM Idle Power. NoManage seems to have almost double the DRAM Idle kleio and our Page Scheduler have, and that clearly has to do with the longer execution duration of NoManage.

lud: Lud's trace acquired has a balanced Read/Write ratio similar to backprop. History's Energy Consumption is really close to NoManage. That means that the initial Page Placement of NoManage did not result in poor DRAM utilization (clearly seen as DRAM consumption is not inexistent in NoManage's bar). After observing Figure 7.2 we were not surprised when kleio did not project huge energy consumption reduction. Our Page Scheduler is only 5% better energy-wise than kleio.

blackscholes: Blackscholes's trace acquired was write-heavy similar to hotspot. History's Energy Consumption is really close to NoManage due to migration cost. That means that the cost of Page movement between the different mem-

ory components in order to achieve high DRAM utilization is not trivial for this particular workload. Kleio and our Page Scheduler seem to behave similarly energy-wise achieving a significant reduction in Energy Consumption.

bodytrack: Bodytrack's trace acquired was also write-heavy. The write operations were certainly more than the read. However the write to read ratio was not as large as those in hotspot and blackscholes. Bodytrack shows quite similar behavior to blackscholes. The only noteworthy difference is that the energy reduction kleio and our Page Scheduler show is not as large compared to the ones in the hotspot and blackscholes applications.

B+Tree: B+Tree's trace acquired had also more write than read operations (3:1 ratio). Similar to bodytrack NoManage is not significantly worse than History, and kleio delivers a rather considerable (20%) decrease in energy consumption. What is really noteworthy in this workload is that our Page Scheduler performs much better energy-wise compared to both History and Kleio. That is probably due to high DRAM hit-rate (observed in fig. 7.2) and the prioritization we made for write operations in Profit equation (5.1, 5.2) in the Page Selector component (Section 5)

Kmeans: Kmeans Read-Write operation ratio is relatively balanced (reads are slightly more than writes). NoManage performs poorly energy-wise for this particular workload. DRAM utilization is really low; thus resulting in a longer runtime duration (High Idle Power). History manages to bring a 40% reduction in energy consumption, while our Page Scheduler and kleio do not seem to improve upon History's energy performance. That was expected to some degree considering that both did not provide huge DRAM hit-rate increase as seen in fig.7.2.

All things considered, we can make the assumption that our Page Scheduler can bring down the energy consumption significantly both decreasing by the total operations carried out by NVM and the runtime of the application. That is especially the case for workloads which are write-intensive. We can also estimate that our Page Scheduler, or an accurate Page Scheduler in general, can be utilized in a low-power oriented system even when workloads have big memory footprint. Only if huge workloads are accompanied with comparatively significant migration energy cost, the system is going to behave poorly. That is not highly probable, as it also evident in [15] where the different page placement approaches followed a similar energy consuming pattern for increasing application memory footprint.

Chapter 8

Conclusions

Thesis Summary

Application data sizes are constantly increasing, while the data access patterns of a wide range of application domains become more and more complex. Traditional memory hardware technologies fail to scale in the necessary capacities and speeds to accelerate modern analytics. Therefore, new hardware technologies are integrated in the memory subsystem to boost application performance and system cost efficiency. This new heterogeneity in the memory hardware and application data access behaviors cannot be handled using existing system-level resource management policies. There is gap between current state-of-the-art solutions in Hybrid Memory System memory management and what could be achieved optimally. In this thesis, we explore the effectiveness and practicality of using Machine Intelligence in Memory Management of a Hybrid Memory System.

We first point out that replacing the hybrid memory manager with one machine intelligent component, such as a reinforcement learning agent, is not scalable and robust to hardware changes. After taking into consideration that applications that execute over hybrid memory systems likely have massive memory footprints, our Page Scheduler identifies a small page subset, whose machine intelligence management boosts application performance. Then, for every single selected page a Recurrent Neural Network is deployed to learn page-level access patterns. For the remaining pages the lightweight history-based scheduling approach is used. In this way, the relative performance gap between existing and an oracular solution is bridged by on average 70% . Besides the use of machine intelligence for page access count predictions, we also explored how using a different eviction policy for DRAM can affect performance. We found out that using an LRU policy after dividing the address space into clusters performs on average 10% better than an LRU policy which does not consider the memory cluster a page belongs to.

Future Work

Both the conclusions drawn from the evaluation, and the assumptions and simplifications made during the implementation of this Thesis leave a lot of room for future work. There are various directions and ideas that someone can use to build upon, and enrich this thesis.

Page size: The Memory Manager (i.e. Page Scheduler) proposed in this Thesis utilizes machine learn methods to learn memory access patterns at the granularity of a page, assuming a 4 KB page size that is primarily used across systems. However, there are many emerging platforms using huge pages (2 MB page size) that show promising performance characteristics. We expect the memory footprints of emerging workloads to be massive. Therefore, we should definitely explore the idea of memory management at a huge page granularity, since the associated learning overhead would be reduced due to the reduction of the aggregate number of ML-models.

Data Objects: There are many solutions that mark memory regions and keep track of the corresponding application level data objects. Such information can be beneficial to better guide the page selection for machine learning-based management. Throughout this thesis, our proposal is on the system-level and is agnostic when it comes to the application data. It certainly worth exploring the two following ideas.

1. Will Machine learning based memory management benefit from clustering the pages that belong to the same data object
2. Can we use application level insights to decide which data object to manage with machine intelligence?

Online Learning: Applications often have multiple phases during their execution, periods of time where one type of activity is occurring followed by a period of time where a completely different activity is occurring. Often these different phases will have very different load patterns from each other. Training the LSTM in an online scenario would allow it to dynamically adjust its weights to be good as possible for the activity that's happening purely in the present. It would no longer have to be trained in a way where it had to generalize to the whole program at once.

ML-based Management in Data Storage: Throughout this Thesis we only focused on bringing a memory management solution in systems with heterogeneous memory hardware by adding machine intelligence. However, there

is certainly potential for the proposed approach of integrating machine learning methods to be extended to the management of storage technologies. The key design points of this Thesis could be followed and applied for managing data stored across storage-only hardware, memory and storage, as well as when considering data offloads to GPUs and accelerators.

Bibliography

[1] C. Chou, A. Jaleel, and M. K. Qureshi, "Batman: techniques for maximizing system bandwidth of memory systems with stacked-dram," *Proceedings of the International Symposium on Memory Systems*, 2017.

[2] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2901318.2901344

[3] D. Shen, X. Liu, and F. X. Lin, "Characterizing emerging heterogeneous memory," ser. ISMM 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 13–23. [Online]. Available: https://doi.org/10.1145/2926697.2926702

[4] T. D. Doudali and A. Gavrilovska, "Comerge: Toward efficient data placement in shared heterogeneous memory systems," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 251–261. [Online]. Available: https://doi.org/10.1145/3132402.3132418

[5] D. Shen, X. Liu, and F. X. Lin, "Characterizing emerging heterogeneous memory," *SIGPLAN Not.*, vol. 51, no. 11, p. 13–23, Jun. 2016. [Online]. Available: https://doi.org/10.1145/3241624.2926702

[6] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[7] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime data management on non-volatile memory-based heterogeneous main memory," *CoRR*, vol. abs/1705.00249, 2017. [Online]. Available: http://arxiv.org/abs/1705.00249

[8] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," 2018.

[9] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A hybrid memory page scheduler with machine intelligence," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 37–48. [Online]. Available: https://doi.org/10.1145/3307681.3325398

[10] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010.

[11] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[12] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano, "A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram," in *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.*, 2005, pp. 459–462.

[13] M. R. Meswani, S. Blagodurov, D. A. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 126–136, 2015.

[14] *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation.* New York, NY, USA: Association for Computing Machinery, 2005.

[15] Y. Zhang, J. Zhan, J. Yang, W. Jiang, L. Li, and Y. Li, "Energy-aware page replacement for nvm based hybrid main memory system," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017, pp. 1–6.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[17] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[18] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-based hybrid memory management," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 152–165.

[19] V. R. Kommareddy, S. D. Hammond, C. Hughes, A. Samih, and A. Awad, "Page migration support for disaggregated non-volatile memories," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 417–427. [Online]. Available: https://doi.org/10.1145/3357526.3357543

[20] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen, "Mempod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 433–444.

[21] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," *SIGPLAN Not.*, vol. 52, no. 4, p. 631–644, apr 2017. [Online]. Available: https://doi.org/10.1145/3093336.3037706

[22] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 126–136.

[23] P. Wu, D. Li, Z. Chen, J. S. Vetter, and S. Mittal, "Algorithm-directed data placement in explicitly managed non-volatile memory," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 141–152. [Online]. Available: https://doi.org/10.1145/2907294.2907321

[24] Y. Chen, I. B. Peng, Z. Peng, X. Liu, and B. Ren, "Atmem: Adaptive data placement in graph applications on heterogeneous memories," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 293–304. [Online]. Available: https://doi.org/10.1145/3368826.3377922

[25] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond, "memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies." [Online]. Available: https://www.osti.gov/biblio/1245908

[26] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira, "Compiler support for selective page migration in numa architectures," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 369–380. [Online]. Available: https://doi.org/10.1145/2628071.2628077

[27] Z. Duan, H. Liu, X. Liao, H. Jin, W. Jiang, and Y. Zhang, "Hinuma: Numa-aware data placement and migration in hybrid memory systems," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019, pp. 367–375.

[28] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," *SIGARCH Comput. Archit. News*, vol. 44, no. 2, p. 369–383, mar 2016. [Online]. Available: https://doi.org/10.1145/2980024.2872401

[29] C. Chakraborttii, V. Sinha, and H. Litz, "Ssd qos improvements through machine learning," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 511. [Online]. Available: https://doi.org/10.1145/3267809.3275453

[30] A. Das, F. Mueller, C. Siegel, and A. Vishnu, "Desh: Deep learning for system health prediction of lead times to failure in hpc," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 40–51. [Online]. Available: https://doi.org/10.1145/3208040.3208051

[31] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/yang

[32] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the idiosyncrasies of real persistent memory," *Proc. VLDB Endow.*, vol. 14, no. 4, p. 626–639, dec 2020. [Online]. Available: https://doi.org/10.14778/3436905.3436921

[33] V. Mironov, I. Chernykh, I. Kulikov, A. Moskovsky, E. Epifanovsky, and A. Kudryavtsev, “Performance evaluation of the intel optane dc memory with scientific benchmarks,” in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 2019, pp. 1–6.

[34] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. Soh, Z. Wang, Y. Xu, S. Dulloor, J. Zhao, and S. Swanson, “Basic performance measurements of the intel optane dc persistent memory module,” 03 2019.

[35] Intel, in *Intel® 64 and IA-32 Architectures Optimization Reference Manual.*, 2016.

[36] M.-L. Chiang and W.-L. Su, “Thread-aware mechanism to enhance inter-node load balancing for multithreaded applications on numa systems,” *Applied Sciences*, vol. 11, no. 14, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/14/6486

[37] “A new hardware counters based thread migration strategy for NUMA systems,” in *13th INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING AND APPLIED MATHEMATICS*, 2019.

[38] P. D. Sanzo, M. Sannicandro, B. Ciciani, and F. Quaglia, “Markov chain-based adaptive scheduling in software transactional memory,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 373–382.

[39] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” *SIGPLAN Not.*, vol. 40, no. 6, p. 190–200, jun 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065034

[40] S. Rajaram, P. Gupta, B. Andrassy, and T. Runkler, “Neural architectures for relation extraction within and across sentence boundaries in natural language text,” 04 2018.

[41] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient backprop,” in *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop.* Berlin, Heidelberg: Springer-Verlag, 1998, p. 9–50.

[42] S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 06, no. 02, pp. 107–116, 1998. [Online]. Available: https://doi.org/10.1142/S0218488598000094

[43] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 11 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[44] A. Graves, N. Jaitly, and A.-r. Mohamed, "Hybrid speech recognition with deep bidirectional lstm," in *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, 2013, pp. 273–278.

[45] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, p. 3104–3112.

[46] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *International Conference on Machine Learning.* PMLR, 2017, pp. 2430–2439.

[47] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 39–50, 2008.

[48] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 298–307. [Online]. Available: https://doi.org/10.1145/1030083.1030124

[49] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *IEEE International Symposium on Workload Characterization (IISWC'10)*, 2010, pp. 1–11.

[50] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris, "Shoal: Smart allocation and replication of memory for parallel programs," in *2015 USENIX Annual Technical Conference (USENIX ATC 15).* Santa Clara, CA: USENIX Association, Jul. 2015, pp. 263–276. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/kaestle

[51] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and*

Operating Systems, ser. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 369–383. [Online]. Available: https://doi.org/10.1145/2872362.2872401

[52] F. Chollet *et al.* (2015) Keras. [Online]. Available: https://github.com/fchollet/keras