Εθνικό Μετσόβιο Πολυτεχνείο
Τμήμα Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

Ευέλικτες Πολιτικές Τοποθέτησης Δεδομένων σε NUMA Αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μιχαλάκη Ελένη-Αικατερίνη

Επιβλέπων: Γκούμας Γεώργιος
Αναπληρωτής Καθηγητής ΕΜΠ

Αθήνα, Φεβρουάριος 2021

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

Ευέλικτες Πολιτικές Τοποθέτησης Δεδομένων σε NUMA Αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μιχαλάκη Ελένη-Αικατερίνη

Επιβλέπων: Γκούμας Γεώργιος
Αναπληρωτής Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 16η Φεβρουαρίου 2021.

………………………………	………………………………	………………………………
Γκούμας Γεώργιος	Κοζύρης Νεκτάριος	Πνευματικάτος Διονύσιος
Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2021

....
Μιχαλάκη Ελένη - Αικατερίνη
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Ελένη Αικατερίνη Μιχαλάκη, 2021
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

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 την συγγραφέα.

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

Περίληψη

Στα σύγχρονα υπολογιστικά συστήματα χρησιμοποιείται ευρέως η αρχιτεκτονική NUMA (Non Uniform Memory Access), στην οποία οι επεξεργαστικοί πυρήνες και η μνήμη ενός μηχανήματος είναι διαμοιρασμένοι σε ”NUMA κόμβους”. Η βασική αρχή τέτοιων συστημάτων είναι ότι η πρόσβαση από έναν πυρήνα στην τοπική του μνήμη, δηλαδή αυτή που βρίσκεται στον ίδιο κόμβο με αυτόν, γίνεται γρηγορότερα από ότι σε απομακρυσμένη. Προκειμένου να πετύχουμε μέγιστη απόδοση κατά την εκτέλεση εφαρμογών σε NUMA συστήματα είθισται να τοποθετούμε τα δεδομένα όσο το δυνατόν πιο κοντά στον πυρήνα που τα προσπελαύνει. Παρ΄όλα αυτά, με την αύξηση των εφαρμογών που εκτελούνται σε ένα μηχάνημα και του πλήθους των δεδομένων που αυτές επεξεργάζονται, αυτή η προσέγγιση παύει να αποτελεί πάντα τη βέλτιστη. Η συμφόρηση του διαύλου μνήμης εμφανίζεται ως ένας σημαντικός παράγοντας για τη βέλτιστη τοποθέτηση με όρους επίδοσης.

Στην παρούσα διπλωματική εργασία μελετάμε το πρόβλημα αυτό στο λειτουργικό σύστημα Linux. Αρχικά δείχνουμε το πώς επηρεάζεται ο χρόνος εκτέλεσης μιας εφαρμογής όταν αυξάνεται ο φόρτος εργασίας ενός μηχανήματος. Στη συνέχεια αναφέρουμε τις πολιτικές δέσμευσης μνήμης τις οποίες διαθέτει το λειτουργικό και δείχνουμε ότι δεν μας παρέχουν αρκετή ευελιξία στην τοποθέτηση δεδομένων. Έτσι, προτείνουμε και υλοποιούμε μια νέα πολιτική που επιτρέπει λεπτομερή (fine grain) κατανομή δεδομένων στον τοπικό και στους απομακρυσμένους κόμβους μιας NUMA αρχιτεκτονικής. Τα πειραματικά αποτελέσματα δείχνουν ότι η ευελιξία που προσφέρει ο νέος μηχανισμός μπορεί να οδηγήσει υπό συνθήκες σε μείωση του χρόνου εκτέλεσης των εφαρμογών έως και κατά 38% σε σχέση με την προεπιλεγμένη.

Λέξεις κλειδιά

NUMA, πολιτικές μνήμης, διαχείριση μνήμης, πυρήνας Linux

Abstract

Modern computing systems widely use NUMA architectures, which organize the system's computing cores and memory into units called NUMA nodes. The basic principle of such systems is that the access time from a core to memory located within the same NUMA node, called a local memory access, is faster than to memory on a remote node. Because of this, it seems clear that in order to achieve maximum performance for applications running in NUMA systems we have to place data as close as possible to the cores that access them. However, with the growing number of applications executed on a machine and the increasingly large working set sizes, this approach of ”local allocation” is no longer optimal. Another factor that we need to consider when deciding on optimal placement is memory bandwidth congestion.

In this diploma thesis we study this problem on the Linux operating system. First, we show how increasing the workload of a system affects the execution time of an application. We then present the different policies of memory allocation available on Linux, and we explain how these do not provide us with enough flexibility in order to solve our problem. Thus, we suggest and implement a new policy, which enables us to define a more fine-grained distribution of data among the local and remote nodes. Our experimental results show that the flexibily of our mechanism can, in certain cases, result in execution time improvement by as much at 38% when compared to the default placement policy.

Keywords

NUMA, memory policies, memory management, Linux kernel

Ευχαριστίες

Με την ολοκλήρωση της παρούσας διπλωματικής εργασίας, και ταυτόχρονα των σπουδών μου στο ΕΜΠ, θα ήθελα να ευχαριστήσω τους γονείς μου για την αδιάκοπη στήριξή τους και για την εκπαίδευση που έλαβα χάρη σε αυτούς.

Ευχαριστώ επίσης τον κ. Γεώργιο Γκούμα για την επίβλεψη της διπλωματικής αυτής, καθώς και τη Χλόη Αλβέρτη και τον Βασίλη Καρακώστα για την βοήθεια και τις συμβουλές τους κατά την εκπόνησή της. Τους ευχαριστώ επίσης για την κατανόηση και την υπομονή που έδειξαν στις συχνές καθυστερήσεις που υπήρξαν λόγω της εργασίας μου και της παραμονής μου στο εξωτερικό.

Τέλος, ευχαριστώ τις φίλες και τους φίλους που είχα την τύχη να γνωρίσω στα χρόνια των σπουδών μου, και στέκονται δίπλα μου μέχρι και σήμερα.

Περιεχόμενα

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Κατάλογος πινάκων 12

Κατάλογος σχημάτων 13

1 Εισαγωγή 17
- 1.1 Σκοπός Εργασίας . . . 17
- 1.2 Συνεισφορά . . . 18
- 1.3 Οργάνωση Κειμένου . . . 19

2 Θεωρητικό Υπόβαθρο 21
- 2.1 NUMA . . . 21
- 2.2 NUMA στο Linux . . . 21
  - 2.2.1 Διαχείριση Μνήμης . . . 22
  - 2.2.2 Πολιτικές Μνήμης NUMA . . . 24
  - 2.2.3 Μεταφορά Σελίδων . . . 25
  - 2.2.4 Automatic NUMA Balancing . . . 26
  - 2.2.5 numactl . . . 27

3 Κίνητρο εργασίας 31

4 Υλοποίηση 35
- 4.1 Σύνοψη . . . 35
- 4.2 Περιγραφή Υλοποίησης . . . 35
- 4.3 Χρήση / Interface . . . 38
- 4.4 Μεταφορά Σελίδων . . . 39

5 Αξιολόγηση 41
- 5.1 Υποδομή . . . 41
- 5.2 Εργαλεία . . . 42
  - 5.2.1 perf . . . 42
- 5.3 Μεθοδολογία . . . 43
  - 5.3.1 Μετρικές . . . 44
- 5.4 Αποτελέσματα . . . 44

5.5 Συμπεράσματα 47

6 Σχετική Έρευνα **49**

7 Επίλογος **51**
7.1 Συμπεράσματα 51
7.2 Μελλοντικές Επεκτάσεις 51

8 Παράρτημα **53**
8.1 Προσθήκη MPOL_PERCENTAGE στον κώδικα του πυρήνα 4.19 (patch) 53
8.2 Προσθήκη -b/–percentage argument στο numactl (patch) 57
8.3 Παράδειγμα χρήσης move_pages 60

Βιβλιογραφία **63**

Κατάλογος πινάκων

3.1 Χρόνος εκτέλεσης 1 διεργασίας για διαφορετικό αριθμό διεργασιών που εκτελούνται ταυτόχρονα στο ίδιο NUMA node. 32
3.2 Χρόνος εκτέλεσης 1 διεργασίας για διαφορετικό αριθμό διεργασιών που εκτελούνται ταυτόχρονα στο απομακρυσμένο NUMA node. 33

5.1 Χαρακτηριστικά πειραματικού μηχανήματος (broady) 41
5.2 Πίνακας NUMA αποστάσεων broady 42
5.3 Μετροπρογράμματα 43
5.4 Αποτελέσματα LBM 45
5.5 Αποτελέσματα GemsFDTD 45
5.6 Αποτελέσματα MCF 46
5.7 Αποτελέσματα Omnetpp 47

Κατάλογος σχημάτων

2.1 Παράδειγμα αρχιτεκτονικής με 2 NUMA nodes 22
2.2 Memory zones σε ένα NUMA node 23
2.3 Παράδειγμα local allocation για 2 διεργασίες σε 2 NUMA nodes 24

3.1 Παράδειγμα memory congestion σε ένα NUMA node. 31
3.2 Ποσοστό αύξησης χρόνου εκτέλεσης διεργασίας για διαφορετικό αριθμό διεργασιών που εκτελούνται ταυτόχρονα στο ίδιο NUMA node. 32
3.3 Ποσοστό αύξησης χρόνου εκτέλεσης διεργασίας για διαφορετικό αριθμό διεργασιών που εκτελούνται ταυτόχρονα στο απομακρυσμένο NUMA node. 33

4.1 Παράδειγμα 2 διεργασιών P1 και P2 με πολιτική percentage 36

5.1 Αποτελέσματα LBM 45
5.2 Αποτελέσματα GemsFDTD 46
5.3 Αποτελέσματα MCF 46
5.4 Αποτελέσματα omnetpp 47

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός Εργασίας

Με την αύξηση του πλήθους και της ταχύτητας των πυρήνων που χρησιμοποιούνται στους σύγχρονους επεξεργαστές εμφανίζεται η ανάγκη για πιο αποδοτική διαχείριση της μνήμης. Πράγματι, οι επεξεργαστές γίνονται όλο και γρηγορότεροι ενώ δεν ισχύει το ίδιο και για τις ταχύτητες πρόσβασης στη μνήμη. Το φαινόμενο αυτό ονομάζεται "CPU - Memory Gap", και αναφέρεται στο "κενό" που εμφανίζεται μεταξύ της απόδοσης των δύο αυτών συστημάτων. Ένα αντίστοιχο πρόβλημα είναι ότι όλο και περισσότεροι πυρήνες προσπαθούν να προσπελάσουν την ίδια μνήμη, δημιουργώντας έτσι μεγαλύτερη κίνηση στις συνδέσεις και αυξάνοντας παραπάνω το χρόνο πρόσβασης σε αυτή.

Για την αντιμετώπιση των παραπάνω προβλημάτων τα σύγχρονα υπολογιστικά συστήματα χρησιμοποιούν αρχιτεκτονικές που αποτελούνται από πολλαπλούς "επεξεργαστικούς κόμβους" (processor nodes). Κάθε τέτοιος κόμβος περιλαμβάνει έναν πολυπύρηνο επεξεργαστή και μια τοπική μνήμη μαζί με έναν ή περισσότερους ελεγκτές μνήμης (memory controllers). Όλοι οι κόμβοι συνδέονται μεταξύ τους με κάποιο δίκτυο διασύνδεσης για να δημιουργηθεί έτσι τελικά ένα cache-coherent σύστημα όπου όλοι οι επεξεργαστικοί πυρήνες μπορούν να προσπελάσουν μνήμη σε κάθε node. Όπως φαίνεται, η πρόσβαση στη μνήμη που βρίσκεται στο ίδιο node με τον πυρήνα παίρνει λιγότερο χρόνο από ότι η πρόσβαση στη μνήμη διαφορετικού node, αφού η τελευταία θα πρέπει να γίνει μέσω του συστήματος διασύνδεσης. Για το λόγο αυτό τέτοιες αρχιτεκτονικές ονομάζονται Non-Uniform Memory Access (NUMA), δηλαδή η ταχύτητα πρόσβασης στη μνήμη εξαρτάται από την τοποθεσία μιας CPU σε σχέση με τη διεύθυνση μνήμης που θέλει να προσπελάσει.

Οι αρχιτεκτονικές αυτές λύνουν τα προβλήματα που αναφέραμε αφού πλέον η μνήμη βρίσκεται πιο "κοντά" σε κάποια CPU cores, και έτσι ο χρόνος πρόσβασής τους σε αυτή μειώνεται. Επίσης οι εφαρμογές μπορούν πλέον να εκτελούνται σε διαφορετικά nodes από τις υπόλοιπες και άρα εφαρμογές οι οποίες πραγματοποιούν πολλά accesses στη μνήμη δεν επηρεάζουν άλλες λόγω συμφόρησης στις συνδέσεις: η κάθε εφαρμογή μπορεί να προσπελαύνει την τοπική της μνήμη ανεξάρτητα από αυτές που εκτελούνται σε άλλα nodes. Γίνεται έτσι σαφές ότι θέλουμε οι εφαρμογές να εκτελούνται σε CPUs του node στο οποίο βρίσκεται η μνήμη τους για να πετύχουμε υψηλή απόδοση.

Παράλληλα, όμως, με την εξέλιξη στο hardware βλέπουμε να συμβαίνουν σημαντικές αλλαγές και στον τομέα του λογισμικού. Στα σύγχρονα κέντρα δεδομένων (data centers) εκτελούνται υπολογιστικά φορτία (workloads) που επεξεργάζονται όλο και με-

γαλύτερο όγκο δεδομένων. Ταυτόχρονα, με την χρήση εικονικοποίησης (virtualization) οι εξυπηρετητές (servers) είναι σε θέση να τρέχουν πολλές και ανεξάρτητες εφαρμογές ταυτόχρονα. Ο τρόπος διαμοιρασμού τέτοιων workloads σε NUMA nodes δεν είναι προφανής, αφού μπορεί για παράδειγμα κάποιες εφαρμογές να απαιτούν περισσότερους πόρους (CPU/μνήμη) για την εκτέλεσή τους από αυτούς που μπορεί να ικανοποιήσει ένα NUMA node. Παρατηρείται επίσης αύξηση του ανταγωνισμού (contention) μεταξύ διαφορετικών νημάτων που εκτελούνται στο ίδιο node και προσπαθούν να προσπελάσουν την ίδια μνήμη. Έτσι, η προσπάθεια για χρήση όσο το δυνατόν περισσότερης τοπικής μνήμης μπορεί σε ορισμένες περιπτώσεις όχι μόνο να μην αποτελεί βέλτιστη λύση, αλλά να επιφέρει παραπάνω καθυστερήσεις στην εκτέλεση των εφαρμογών.

Προκύπτει έτσι η ανάγκη για έρευνα πάνω στον τρόπο με τον οποίο τοποθετούνται οι εφαρμογές σε NUMA nodes και στο πού δεσμεύουν τη μνήμη τους, και για ανάπτυξη νέων μηχανισμών για αυτούς τους σκοπούς. Το θέμα αυτό παρουσιάζει ιδιαίτερο ενδιαφέρον καθώς θέλουμε να πετύχουμε την κατά το δύνατον βέλτιστη εκτέλεση των εφαρμογών χωρίς ταυτόχρονα να απαιτούμε περισσότερους πόρους ή/και να αφήνουμε κάποιους ανεκμετάλλευτους. Μια τέτοια ερευνητική εργασία είναι η [4], στην οποία οι συγγραφείς παρατηρούν ότι σε συστήματα που παρουσιάζουν υψηλή συμφόρηση σε ένα NUMA node η απόδοση μπορεί να βελτιωθεί εάν ένα ποσοστό της μνήμης γίνει allocate σε απομακρυσμένο NUMA node.

1.2 Συνεισφορά

Σε αυτή τη διπλωματική εργασία ασχολούμαστε με το λειτουργικό σύστημα Linux. Το Linux ως προεπιλογή προσπαθεί να τοποθετεί τη μνήμη μίας εφαρμογής τοπικά στο node στο οποίο αυτή εκτελείται. Παρέχει όμως και κάποιες εναλλακτικές πολιτικές NUMA, δηλαδή τρόπους τοποθέτησης της μνήμης μεταξύ των NUMA nodes, οι οποίες μπορούν να επιλεγούν από το χρήστη για μια διεργασία. Οι πολιτικές αυτές μας δίνουν τη δυνατότητα να επιλέξουμε ένα σύνολο nodes από τους οποίους θα δεσμεύεται η μνήμη, καθώς και το εάν θα επιτρέπεται η δέσμευση μνήμης σε διαφορετικό node, στην περίπτωση που οι επιλεγμένοι δεν μπορούν να ικανοποιήσουν τις απαιτήσεις της εφαρμογής. Δεν μπορούμε όμως να πειραματιστούμε με το ποσοστό της μνήμης που θα δεσμεύεται από κάθε node; μπορούμε μόνο να ζητήσουμε η μνήμη να ισοκατανέμεται μεταξύ του συνόλου που έχουμε ορίσει.

Με αφορμή τα παραπάνω, μελετάμε το πρόβλημα της μείωσης της απόδοσης ενός NUMA συστήματος καθώς αυξάνεται το workload και ταυτόχρονα ο ανταγωνισμός διαφορετικών νημάτων που εκτελούνται στο ίδιο NUMA node για πρόσβαση στην τοπική μνήμη. Αρχικά δείχνουμε το πώς μεταβάλλεται ο χρόνος εκτέλεσης μιας διεργασίας με την προσθήκη επιπλέον διεργασιών στο ίδιο node. Στη συνέχεια προτείνουμε και υλοποιούμε μια πολιτική δέσμευσης μνήμης (memory allocation) η οποία μας επιτρέπει να ορίσουμε το ποσοστό μνήμης το οποίο θα δεσμεύεται σε απομακρυσμένα NUMA nodes. Τέλος δείχνουμε το πώς η μέθοδος αυτή μπορεί, για ορισμένες εφαρμογές, να οδηγήσει σε βελτίωση της απόδοσης έως και 38% σε σχέση με την τοποθέτηση της μνήμης τοπικά στο node εκτέλεσης.

1.3 Οργάνωση Κειμένου

Κεφάλαιο 2: Θεωρητικό Υπόβαθρο

Στο κεφάλαιο αυτό εισάγουμε τα NUMA συστήματα και τις βασικές έννοιες σχετικά με αυτά. Στη συνέχεια περιγράφουμε αναλυτικά την υπάρχουσα υποστήριξη του NUMA στο λειτουργικό σύστημα Linux: τις πολιτικές μνήμης οι οποίες είναι διαθέσιμες, τα εργαλεία που παρέχονται στο χρήστη, τη μεταφορά σελίδων μνήμης καθώς και το Automatic NUMA Balancing.

Κεφάλαιο 3: Κίνητρο Εργασίας

Στο κεφάλαιο αυτό μελετάμε ένα πιθανό σενάριο εκτέλεσης διεργασιών το οποίο αποτέλεσε την αφορμή μας για επέκταση των πολιτικών και εργαλείων που παρουσιάσαμε στο θεωρητικό υπόβαθρο.

Κεφάλαιο 4: Υλοποίηση

Στο κεφάλαιο αυτό παρουσιάζουμε τη λύση μας στο πρόβλημα που δείξαμε στο προηγούμενο κεφάλαιο. Αρχικά μελετάμε το γιατί οι υπάρχουσες λύσεις δεν είναι αρκετές. Στη συνέχεια περιγράφουμε την υλοποίησή μας και δείχνουμε τον τρόπο με τον οποίο μπορεί να χρησιμοποιηθεί.

Κεφάλαιο 5: Αξιολόγηση

Στο κεφάλαιο αυτό αξιολογούμε το κατά πόσο η υλοποίησή μας μπορεί να αποτελέσει μια χρήσιμη εναλλακτική απέναντι στις υπάρχουσες, και σε ποιές περιπτώσεις συμβαίνει αυτό. Περιγράφουμε το μηχάνημα, τα εργαλεία και τα workloads που χρησιμοποιήσαμε, τα αποτελέσματά των πειραμάτων μας και προσπαθούμε να τα ερμηνεύσουμε.

Κεφάλαιο 6: Σχετική Έρευνα

Στο κεφάλαιο αυτό παρουσιάζουμε ερευνητικές εργασίες σχετικές με NUMA συστήματα και με το πρόβλημα το οποίο εμείς μελετάμε.

Κεφάλαιο 7: Επίλογος

Στο τελευταίο κεφάλαιο συνοψίζουμε την δουλειά μας, και προτείνουμε πιθανές μελλοντικές επεκτάσεις σε αυτή.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 NUMA

Ο όρος Non-Uniform Memory Access (Μη Ομοιόμορφη Πρόσβαση στη Μνήμη - NUMA) χρησιμοποιείται για να περιγράψει ένα μοντέλο μνήμης σε πολυεπεξεργαστικά υπολογιστικά συστήματα, σύμφωνα με το οποίο ο χρόνος πρόσβασης στη μνήμη εξαρτάται από την τοποθεσία της μνήμης σε σχέση με τον επεξεργαστή (CPU) ο οποίος την προσπελαύνει.

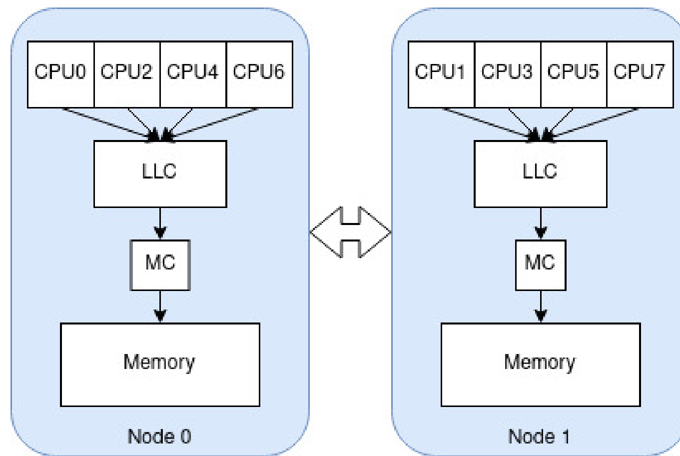
Ένα NUMA σύστημα μπορεί να θεωρηθεί ότι αποτελείται από μικρότερα SMP (Symmetric Multiprocessing - Συμμετρικά Πολυεπεξεργαστικά Συστήματα), καθένα από τα οποία περιέχει 1 ή περισσότερες CPUs, μνήμη και ελεγκτές μνήμης ή/και I/O buses. Κάθε ένα τέτοιο σύστημα ονομάζεται κόμβος NUMA (NUMA node). Τα nodes συνδέονται μεταξύ τους με κάποιο δίκτυο διασύνδεσης, και έτσι κάθε CPU μπορεί να προσπελάσει μνήμη η οποία ανήκει σε άλλο NUMA node. Με τον τρόπο αυτό δημιουργείται μια αρχιτεκτονική στην οποία κάθε CPU μπορεί να προσπελάσει τη μνήμη που βρίσκεται στο ίδιο NUMA node, ή ”τοπική μνήμη” (local memory) σε λιγότερο χρόνο από ότι αυτήν που βρίσκεται σε άλλα NUMA nodes (απομακρυσμένη - remote).

Οι αρχιτεκτονικές NUMA πετυχαίνουν έτσι κλιμακωσιμότητα σε επίπεδο μνήμης (scalable memory bandwidth [28]), αφού δεν υπάρχει ένας κεντρικός ελεγκτής μνήμης (memory controller) όπως επίσης και μειώνουν τη συμφόρηση που πρόκυπτει όταν πολλοί επεξεργαστές επιχειρούν να προσπελάσουν την ίδια μνήμη, αυξάνοντας έτσι την αποδοτικότητα του συστήματος [2]. Τα συστήματα NUMA χρησιμοποιούνται κυρίως σε data centers (κέντρα δεδομένων) τα οποία φιλοξενούν εφαρμογές που επεξεργάζονται μεγάλο όγκο δεδομένων.

Στο Σχήμα 2.1 φαίνεται ένα παράδειγμα μιας τέτοιας αρχιτεκτονικής με 2 NUMA nodes.

2.2 NUMA στο Linux

Ο πυρήνας του Linux (Linux kernel) υποστηρίζει NUMA πλατφόρμες από την έκδοση 2.4. Συγκεκριμένα υποστηρίζει τα λεγόμενα Cache Coherent NUMA (ccNUMA) συστήματα, στα οποία η συνολική μνήμη είναι προσβάσιμη από κάθε CPU, και οι cache των CPUs ή/και η διασύνδεση του συστήματος είναι υπεύθυνα για το cache coherence (συνέπεια) του συστήματος. Από εδώ και στο εξής με τον όρο NUMA θα αναφερόμαστε σε ccNUMA.



Σχήμα 2.1: Παράδειγμα αρχιτεκτονικής με 2 NUMA nodes

Το Linux υλοποιεί την έννοια του NUMA node σε επίπεδο λογισμικού. Για κάθε NUMA node που υπάρχει στο hardware του μηχανήματος, το Linux "βλέπει" ένα node σε επίπεδο software το οποίο επίσης περιέχει CPUs, μνήμη ή/και I/O buses. Όπως και με τα φυσικά nodes, η πρόσβαση στη μνήμη ενός κοντινότερου node γίνεται γρηγορότερα από ότι σε μια απομακρυσμένη.

Με αυτό τον τρόπο το Linux καταφέρνει να "κρύψει" από το χρήστη λεπτομέρειες σχετικά με την αρχιτεκτονική του hardware του μηχανήματος. Αξίζει να σημειωθεί εδώ ότι σε ορισμένες αρχιτεκτονικές η αντιστοίχηση software node - hardware node δεν είναι απαραίτητα 1-1. Για παράδειγμα, εάν κάποιο hardware node δεν διαθέτει δική του μνήμη το Linux δεν θα δημιουργήσει ένα software node για αυτό αλλά θα συμπεριλάβει τις CPUs του ως CPUs ενός άλλου software node το οποίο διαθέτει μνήμη. Το Linux υποστηρίζει επιπλέον την "προσομοίωση" επιπλέον NUMA nodes από αυτούς του hardware (NUMA emulation). Η λειτουργία αυτή είναι χρήσιμη γιατί μας επιτρέπει να δοκιμάσουμε NUMA εφαρμογές ακόμα και εάν δεν διαθέτουμε ένα NUMA σύστημα σε επίπεδο hardware.

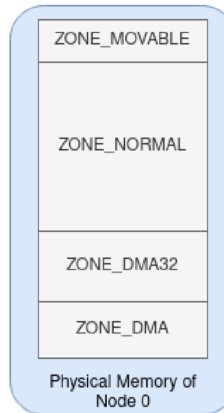
Στη συνέχεια του κειμένου όταν θα αναφερόμαστε στην έννοια NUMA node θα εννοούμε σε επίπεδο software, χωρίς να μας ενδιαφέρει ο σχεδιασμός του hardware του συστήματος.

2.2.1 Διαχείριση Μνήμης

Για κάθε node, το Linux δημιουργεί ένα ανεξάρτητο σύστημα διαχείρισης μνήμης το οποίο περιλαμβάνει πίνακες σελίδων, στατιστικά χρήσης και locks για έλεγχο πρόσβασης. Η μνήμη του κάθε node χωρίζεται στα λεγόμενα memory zones (ζώνες μνήμης) ανάλογα με το σκοπό για τον οποίο θα χρησιμοποιηθεί. Στην έκδοση 4.19 του πυρήνα, τα διαθέσιμα memory zones είναι τα `ZONE_DMA`, `ZONE_DMA32`, `ZONE_HIGHMEM`, `ZONE_NORMAL`, `ZONE_MOVABLE`, `ZONE_DEVICE`. Για παράδειγμα, μνήμη η οποία χρησιμοποιείται από συσκευές για Direct Memory Access θα ανήκει στη `ZONE_DMA`.

Ο αριθμός και τα είδη των memory zones που υπάρχουν σε ένα σύστημα εξαρτώνται από το hardware και από τον τρόπο με τον οποίο έχει ρυθμιστεί και μεταγλωττιστεί ο πυρήνας, και έτσι δεν συναντάμε όλα τα zones που αναφέραμε παραπάνω σε όλα τα

μηχανήματα. Στο Σχήμα 2.2 βλέπουμε ένα παράδειγμα ενός memory node του οποίου η διαθέσιμη μνήμη χωρίζεται σε 4 zones. Σε ένα Linux σύστημα μπορούμε να δούμε πληροφορίες σχετικά με τα zones και τα nodes διαβάζοντας το αρχείο `/proc/zoneinfo`.



Σχήμα 2.2: Memory zones σε ένα NUMA node

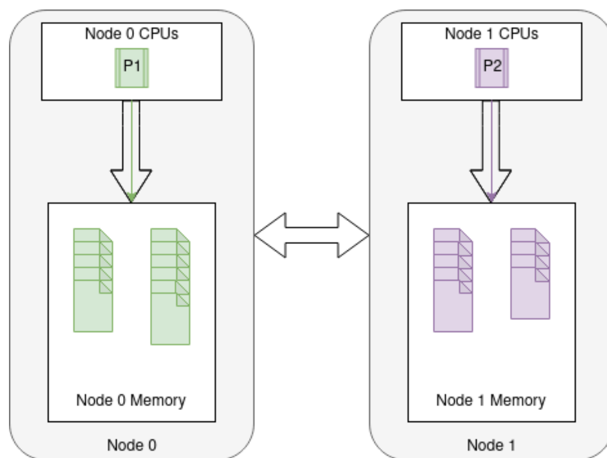
Στη διπλωματική αυτή ασχολούμαστε με το node στο οποίο θα γίνει το allocation και όχι με το zone, οπότε δε θα μπούμε σε παραπάνω λεπτομέρειες σχετικά με τα είδη των zones και τι allocations εξυπηρετεί το καθένα. Αξίζει να σημειώσουμε όμως ότι η πλειοψηφία των allocations γίνεται στο ZONE_NORMAL.

Για κάθε memory zone ο πυρήνας κατασκευάζει ένα ordered "zonelist", το οποίο είναι μια ταξινομημένη λίστα με όλα τα zones/nodes από τα οποία μπορεί να δεσμευτεί μνήμη στην περίπτωση που το ζητούμενο zone/node δεν μπορεί να χρησιμοποιηθεί (π.χ. επειδή δεν υπάρχει αρκετή ελεύθερη μνήμη). Επειδή κάθε node περιλαμβάνει 1 ή περισσότερα zones, η zonelist μπορεί να ταξινομηθεί με 2 τρόπους: είτε όταν αποτυγχάνει ένα allocation σε ένα zone ενός node να γίνεται fallback στο ίδιο zone διαφορετικού node, είτε σε άλλο zone αλλά στο ίδιο node. Το Linux ως προεπιλογή επιλέγει να ταξινομήσει τη zonelist ανά node, δηλαδή όταν αποτυγχάνει ένα allocation να δοκιμάζεται εκ νέου σε διαφορετικό zone του ίδιου node.

Μπορούμε τώρα να δούμε τι συμβαίνει όταν ο πυρήνας λάβει ένα αίτημα για δεσμεύση μνήμης (memory allocation) από μια CPU του συστήματος. Όπως είπαμε, η μνήμη του node στο οποίο ανήκει η κάθε CPU είναι και αυτή με τον μικρότερο χρόνο πρόσβασης. Επομένως είναι λογικό ο πυρήνας να προσπαθεί να δεσμεύσει τη μνήμη από το local node, πριν αναζητήσει εναλλακτικές. Η πολιτική αυτή ονομάζεται local allocation. Εάν αυτό αποτύχει, ο πυρήνας θα αναζητήσει στο zonelist την αμέσως επόμενη επιλογή zone/node.

Το local allocation φαίνεται και στο Σχήμα 2.3: οι διεργασίες P1 και P2 εκτελούνται στους κόμβους 0 και 1 αντίστοιχα, και η κάθε μια έχει δεσμεύσει μνήμη στο τοπικό node. Έτσι μειώνεται ο χρόνος πρόσβασης στη μνήμη και δεν υπάρχει κίνηση στο bus που συνδέει τα 2 nodes μεταξύ τους.

Η πολιτική local allocation προσπαθεί όσο γίνεται να πετύχει η πρόσβαση στη μνήμη να γίνεται τοπικά από τις CPUs του κάθε node στην αντίστοιχη μνήμη του ίδιου node και να αποφεύγει τη χρήση του διαύλου επικοινωνίας των nodes μεταξύ τους. Όμως, όπως γνωρίζουμε ήδη οι διεργασίες μπορεί να μεταφερθούν από μια CPU σε μια άλλη λόγω του scheduler. Στην περίπτωση που μια διεργασία εκτελείται σε μια CPU ενός node από το οποίο έχει δεσμεύσει μνήμη μεταφερθεί σε μια CPU ενός άλλου node, η πρόσβαση



Σχήμα 2.3: Παράδειγμα local allocation για 2 διεργασίες σε 2 NUMA nodes

στη μνήμη δε γίνεται πλέον τοπικά.

Για το σκοπό αυτό και για να μπορεί ο χρήστης να ελέγχει το πώς δεσμεύεται η μνήμη το Linux παρέχει λειτουργίες όπως: επιπλέον πολιτικές δεσμεύσης μνήμης εκτός από το local allocation, μεταφορά σελίδων μνήμης από ένα node σε ένα άλλο, και δυνατότητα επιλογής των CPUs στις οποίες θα εκτελεστεί μία διεργασία.

Παρακάτω θα περιγράψουμε αναλυτικά τις διαφορετικές πολιτικές δεσμεύσης μνήμης τις οποίες παρέχει το Linux, καθώς και τη μεταφορά σελίδων μεταξύ των nodes.

2.2.2 Πολιτικές Μνήμης NUMA

Η πολιτική μνήμης NUMA (NUMA memory policy) καθορίζει τον τρόπο με τον οποίο το Linux αποφασίζει από ποιό NUMA node θα προσπαθήσει να δεσμεύσει μνήμη σε ένα NUMA σύστημα. [18] Μπορεί να οριστεί από τον διαχειριστή του συστήματος με τη χρήση system calls που περιγράφουμε παρακάτω. Ένα NUMA policy αποτελείται στην ουσία από 3 πράγματα:

- Ένα ”mode”, το οποίο καθορίζει τη συμπεριφορά του policy
- Ένα σύνολο κόμβων (προαιρετικά)
- Διάφορα flags (προαιρετικά)

Τα memory policies που υποστηρίζονται είναι τα:

- **Preferred**: Αναπαρίσταται από το mode `MPOL_PREFERRED`. Σύμφωνα με αυτή την πολιτική ο πυρήνας προσπαθεί να δεσμεύσει μνήμη από έναν ή περισσότερους κόμβους οι οποίοι έχουν επιλεγεί από το χρήστη και είναι αποθηκευμένοι στο σύνολο κόμβων του policy. Στην περίπτωση που αυτό δεν είναι εφικτό, θα προσπαθήσει να βρει το κοντινότερο διαθέσιμο node και να χρησιμοποιήσει αυτό.

  Με βάση τα παραπάνω, βλέπουμε ότι η πολιτική local allocation είναι μια Preferred πολιτική όπου το προτιμώμενο node είναι το τοπικό.

- **Bind**: Αναπαρίσταται από το mode `MPOL_BIND`. Σύμφωνα με την πολιτική bind η δεσμεύση μνήμης πρέπει να γίνει από τους κόμβους που ανήκουν στο σύνολο κόμβων, αλλιώς θα αποτύχει.

- **Interleave**: Αναπαρίσταται από το mode MPOL_INTERLEAVED. Κατά την πολιτική interleave ο χρήστης ορίζει ένα σύνολο κόμβων και ο πυρήνας προσπαθεί να δεσμεύσει μνήμη εναλλάξ από κάθε κόμβο στο σύνολο αυτό. Μπορούμε με αυτόν τον τρόπο να πετύχουμε μια "ομοιόμορφη" κατανομή της μνήμης μιας εφαρμογής σε ένα σύνολο κόμβων.

Υπάρχει επίσης το mode MPOL_DEFAULT, το οποίο χρησιμοποιείται όταν θέλουμε να επιλέξουμε το default memory policy του συστήματος.

Εμβέλεια Πολιτικών Μνήμης

Ο πυρήνας υποστηρίζει την δυνατότητα να καθορίσουμε την εμβέλεια που έχει μια πολιτική μνήμης, καθώς και να ορίζουμε διαφορετικές πολιτικές με διαφορετικές εμβέλειές. Η εμβέλεια μπορεί να είναι μια από τις παρακάτω:

- **Default πολιτική συστήματος**: Η πολιτική αυτή είναι "hard coded" στον πυρήνα. Ορίζει το πώς θα γίνονται όλα τα allocations του συστήματος, εκτός και εάν έχουν οριστεί πιο συγκεκριμένες πολιτικές όπως περιγράφουμε παρακάτω. Η default πολιτική, όπως είπαμε και παραπάνω, είναι τα local allocations.

- **Πολιτική διεργασίας (task)**: Προαιρετικά, ο χρήστης μπορεί να επιλέξει την πολιτική μνήμης που θα ακολουθεί μια συγκεκριμένη διεργασία. Η πολιτική αυτή εφαρμόζεται σε ολόκληρο το χώρο διευθύνσεων της διεργασίας (εκτός και έαν έχει οριστεί πολιτική VMA - βλ. παρακάτω) και κληρονομείται μέσω των κλήσεων συστήματος `fork()`, `exec*()`. Στην περίπτωση πολυ-νηματισμού (multithreading) η πολιτική αυτή εφαρμόζεται μόνο στο task που την όρισε και σε όσα αυτό δημιουργεί. Επίσης, στην περίπτωση που η διεργασία έχει ήδη δεσμεύσει σελίδες στη μνήμη αυτές δεν επηρεάζονται από την αλλαγή πολιτικής. Η νέα πολιτική εφαρμόζεται για όλες τις σελίδες οι οποίες θα γίνουν allocate αφού αυτή οριστεί. Μπορούμε να δούμε την πολιτική μιας διεργασίας με την κλήση συστήματος `get_mempolicy()` [10] και να την αλλάξουμε με την `set_mempolicy()` [25].

- **Πολιτική VMA**: Ένα VMA (Virtual Memory Area) είναι ένα μέρος του εικονικού χώρου διευθύνσεων μιας διεργασίας. Μια διεργασία μπορεί να ορίσει μια συγκεκριμένη πολιτική για ένα συγκεκριμένο VMA, η οποία πολιτική θα έχει τότε προτεραιότητα απέναντι στην πολιτική της διεργασίας. Για όσα VMAs δεν έχει οριστεί συγκεκριμένη πολιτική, εφαρμόζεται η πολιτική διεργασίας. Εάν δεν έχει οριστεί αυτή εφαρμόζεται η default πολιτική του συστήματος. Η κλήση συστήματος `mbind()` [13] μπορεί να χρησιμοποιηθεί για να ορίσουμε την πολιτική ενός VMA.

- **Shared πολιτική**: Η πολιτική αυτή εφαρμόζεται σε memory objects τα οποία είναι shared μεταξύ διεργασιών, δηλαδή έχουν δημιουργηθεί με μια κλήση όπως `shmget()` ή `mmap(MAP_ANONYMOUS|MAP_SHARED)`.

Στη διπλωματική αυτή ασχολούμαστε με πολιτικές ανά διεργασία και όχι ανά VMA ή shared objects.

2.2.3 Μεταφορά Σελίδων

Το Linux υποστηρίζει τη μεταφορά σελίδων της μνήμης από ένα NUMA node σε ένα άλλο κατά τη διάρκεια εκτέλεσης μιας εφαρμογής. Αυτό σημαίνει ότι η εικονική μνήμη

που βλέπει η εφαρμογή δεν αλλάζει. Αυτό που αλλάζει είναι η φυσική τοποθεσία αυτών των σελίδων στα nodes.

Η αλλαγή της πολιτικής μνήμης μιας εφαρμογής όσο αυτή εκτελείται δεν επηρεάζει τις σελίδες τις οποίες η εφαρμογή έχει δεσμεύσει και υπάρχουν ήδη στη μνήμη. Δηλαδή, οι σελίδες αυτές δεν μεταφέρονται αυτόματα σε διαφορετικό κόμβο ανάλογα με το τι ορίζει η νέα πολιτική.

Στην περίπτωση που θέλουμε να μεταφέρουμε σελίδες από ένα κόμβο σε άλλον, μπορούμε να χρησιμοποιήσουμε τις παρακάτω κλήσεις συστήματος:

- `migrate_pages` [14]: Η κλήση αυτή μπορεί να χρησιμοποιηθεί εάν θέλουμε να μεταφέρουμε όλη τη μνήμη μιας εφαρμογής που βρίσκεται σε ένα σύνολο κόμβων σε ένα άλλο. Παίρνει ώς παραμέτρους το pid της εφαρμογής, το σύνολο των κόμβων από τους οποίους θέλουμε να μεταφέρουμε και το σύνολο των κόμβων προορισμού. Σελίδες της μνήμης της εφαρμογής οι οποίες βρίσκονται σε κόμβο που δεν είναι στο σύνολο που επιλέγουμε να μεταφέρουμε δεν επηρεάζονται.

- `move_pages` [16]: Η κλήση αυτή χρησιμοποιείται σε περιπτώσεις που θέλουμε να μεταφέρουμε ένα μέρος της μνήμης μιας εφαρμογής από ένα σύνολο κόμβων σε ένα άλλο. Συγκεκριμένα, εκτός από τα 2 σύνολα κόμβων και το pid της εφαρμογής παίρνει ως παράμετρο και το σύνολο των σελίδων που θέλουμε να μεταφέρουμε. Αυτό σημαίνει ότι για τη χρήση της πρέπει να γνωρίζουμε την κατανομή των σελίδων της εφαρμογής στη μνήμη. Είναι φανερό ότι η `move_pages` παρέχει μεγαλύτερη ευελιξία από την `migrate_pages`, είναι όμως δυσκολότερη στη χρήση.

- `mbind(... MF_MOVE | MF_MOVE_ALL)` [13]: Όπως είπαμε παραπάνω, η mbind χρησιμοποιείται όταν μια εφαρμογή θέλει να αλλάξει την πολιτική μνήμης κάποιων VMA της. Εάν στην mbind χρησιμοποιηθεί κάποιο από τα flags `MF_MOVE` ή `MF_MOVE_ALL`, τότε οι σελίδες του VMA αυτού θα μεταφερθούν κατάλληλα ώστε να υπακούουν πλέον στην νέα πολιτική.

Σημειώνουμε εδώ ότι η μεταφορά ορισμένων σελίδων δεν είναι εφικτή. Τέτοιες σελίδες μπορεί να είναι κάποιες οι οποίες χρησιμοποιούνται και από άλλες διεργασίες όταν ο χρήστης δεν έχει τα απαραίτητα δικαιώματα ή σελίδες που χρησιμοποιούνται από kernel threads. Οι παραπάνω κλήσεις συστήματος, πριν επιχειρήσουν τη μεταφορά των σελίδων, ελέγχουν εάν υπάρχει αυτή η δυνατότητα.

2.2.4 Automatic NUMA Balancing

Όπως προαναφέραμε, σε ένα NUMA σύστημα είναι γενικά προτιμότερο η πρόσβαση στη μνήμη να γίνεται όσο το δυνατόν περισσότερο στο ίδιο NUMA node. Για να το πετύχει αυτό το Linux, πέρα από το να χρησιμοποιεί local allocations ως default πολιτική, χρησιμοποιεί την τεχνική του Automatic NUMA Balancing [8]. Σύμφωνα με αυτήν, ο πυρήνας περιοδικά ελέγχει ποιά tasks κάνουν access διάφορες σελίδες που βρίσκονται στη μνήμη. Στη συνέχεια αποφασίζει εάν οι σελίδες αυτές πρέπει να μεταφερθούν σε διαφορετικό node, ώστε να βρίσκονται πιο κοντά στα tasks που τις προσπελαύνουν. Αντίστοιχα μπορεί να επιλέξει να μεταφέρει tasks σε διαφορετικές CPU.

Λόγω του τρόπου λειτουργίας της, η τεχνική αυτή μπορεί να επιφέρει επιπλέον overhead στην απόδοση του συστήματος. Υπάρχουν περιπτώσεις στις οποίες το NUMA balancing χρειάζεται να απενεργοποιηθεί, ή να ρυθμιστεί κατάλληλα. Αυτό μπορεί να γίνει μέσω του /proc filesystem [9].

Σε αυτή τη διπλωματική θεωρούμε ότι το NUMA balancing είναι απενεργοποιημένο, αφού θέλουμε να αποφύγουμε τη μεταφορά σελίδων σε διαφορετικά nodes από αυτά που έχουμε εμείς επιλέξει για τα πειράματά μας.

2.2.5 numactl

Για να μπορέσει ένας χρήστης να εκτελεί εντολές σχετικές με τις NUMA δυνατότητες ενός Linux συστήματος χωρίς να χρειάζεται να γράψει επιπλέον κώδικα που να χρησιμοποιεί τα system calls που αναφέραμε παραπάνω, μπορεί να χρησιμοποιηθεί το εργαλείο numactl [20] [21]. Το numactl μας δίνει τη δυνατότητα να εκτελέσουμε εφαρμογές με συγκεκριμένη πολιτική NUMA, καθώς και να λάβουμε πληροφορίες σχετικά με τη NUMA αρχιτεκτονική του συστήματος. Βασίζεται στη βιβλιοθήκη libnuma η οποία παρέχει μια προγραμματιστική διεπαφή (interface) για την ανάπτυξη εφαρμογών που χρησιμοποιούν τις NUMA δυνατότητες του πυρήνα του Linux. Κομμάτια του ίδιου project αποτελούν το numastat, για πληροφορίες σχετικά με τη μνήμη μιας εφαρμογής, το migratepages για μεταφορά σελίδων μιας εφαρμογής ενώ αυτή εκτελείται κ.α.

Παρακάτω αναλύουμε μερικούς τρόπους με τους οποίους μπορούμε να χρησιμοποιήσουμε το εργαλείο αυτό, οι οποίοι μας φάνηκαν χρήσιμοι σε αυτή την εργασία:

Πληροφορίες για το σύστημα:

Το numactl αποτελεί έναν εύκολο τρόπο να μελετήσουμε τη NUMA αρχιτεκτονική ενός συστήματος και το πόση μνήμη είναι δεσμευμένη σε κάθε node. Αυτό γίνεται μέσω της επιλογής `-H`/`-hardware`:

Listing 2.1: numactl -H

```
emichal@broady:~$ numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 44 45 46 47
    ↪ 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
node 0 size: 257843 MB
node 0 free: 245653 MB
node 1 cpus: 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
    ↪ 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
node 1 size: 258033 MB
node 1 free: 256337 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

Παραπάνω βλέπουμε ότι στο μηχάνημα broady έχουμε 2 NUMA nodes στο καθένα από τα οποία ανήκουν 44 cpus. Βλέπουμε επίσης το μέγεθος της μνήμης του κάθε node, πόση από αυτήν χρησιμοποιείται αυτή τη στιγμή καθώς και τις αποστάσεις μεταξύ των nodes.

Τοποθέτηση νημάτων σε επεξεργαστές

Το numactl μπορεί να χρησιμοποιηθεί για να επιλέξουμε τους πυρήνες (CPU cores) στους οποίους θα εκτελεστεί μια εφαρμογή, μέσω των επιλογών `-physcpubind= | -C <cpus>`. Μπορούμε ακόμη να επιλέξουμε όλες τις CPU ενός socket με `-cpunodebind= | -N <nodes>`. Η δυνατότητα αυτή είναι ιδιαίτερα χρήσιμη διότι περιορίζοντας της εφαρμογή μας σε συγκεκριμένες CPU μπορούμε να μελετήσουμε διαφορετικές πολιτικές NUMA με ακρίβεια και αποφεύγοντας τυχόν αλλαγές λόγω του scheduler.

Ρύθμιση NUMA πολιτικής εφαρμογών

Μέσω του numactl μπορούμε να τρέξουμε μια εφαρμογή χρησιμοποιώντας μια συγκεκριμένη πολιτική NUMA, όπως αυτές περιγράφηκαν σε προγούμενο κεφάλαιο. Για το σκοπό αυτό χρησιμοποιεί την κλήση συστήματος `set_mempolicy()` [25]. Συγκεκριμένα μας παρέχει τις παρακάτω επιλογές:

- **Local**: Η εφαρμογή δεσμεύει μνήμη από το NUMA node στις CPU του οποίου εκτελείται.

  ```
  $ numactl --localalloc COMMAND ARGS
  ```

- **Bind**: Η εφαρμογή δεσμεύει μνήμη μόνο από συγκεκριμένα NUMA nodes:

  ```
  $ numactl --membind=<nodes> COMMAND ARGS
  ```

- **Preferred**: Η εφαρμογή προσπαθεί να δεσμεύσει μνήμη μόνο από τα επιθυμητά NUMA nodes. Εάν αυτό δεν είναι εφικτό, θα επιχειρήσει να βρει το κοντινότερο node το οποίο έχει ελεύθερη μνήμη:

  ```
  $ numactl --preferred=<nodes> COMMAND ARGS
  ```

- **Interleave**: Η εφαρμογή δεσμεύει μνήμη από όλα τα δεδομένα nodes σύμφωνα με την πολιτική interleave:

  ```
  $ numactl --interleave=<nodes> COMMAND ARGS
  ```

Πληροφορίες για μνήμη εφαρμογής

Με χρήση του εργαλείου numastat [22] μπορούμε να πάρουμε πληροφορίες για τον τρόπο με τον οποίο κατανέμεται η μνήμη μιας εφαρμογής στα διάφορα NUMA nodes. Αυτό φαίνεται στο παρακάτω παράδειγμα:

```
$ numastat qemu

Per-node process memory usage (in MBs) for PID 174242 (qemu-system-x86)
                        Node 0          Node 1           Total
               --------------- --------------- ---------------
Huge                      0.00            0.00            0.00
Heap                      7.48            1.12            8.60
Stack                     0.01            0.01            0.02
Private                3819.29         3516.78         7336.07
-------------  --------------- --------------- ---------------
Total                  3826.78         3517.91         7344.69
```

Βλέπουμε ότι το qemu έχει δεσμεύσει συνολικά 7344MB μνήμης, από τα οποία τα 3517.91 έχουν δεσμευτεί στο Node 1 και τα υπόλοιπα 3826.78 στο Node 0.

Το numastat χωρίς κανένα όρισμα μπορεί να χρησιμοποιηθεί για τη συλλογή στατιστικών για όλο το σύστημα:

```
$ numastat
                           node0           node1
numa_hit                 8085696         1310149
numa_miss                  12534          108936
numa_foreign               12534          108936
interleave_hit            108974          108984
local_node               8085696         1310149
other_node                     0               0
```

Μεταφορά σελίδων εφαρμογής

Το migratepages [15] είναι ένα user space εργαλείο που μπορεί να χρησιμοποιηθεί για να μεταφέρουμε τις σελίδες μιας διεργασίας από ένα σύνολο NUMA nodes σε ένα άλλο, κατά τη διάρκεια εκτέλεσής της. Χρησιμοποιεί το system call migrate_pages [14]. Παίρνει ως ορίσματα το pid της διεργασίας και τα 2 σύνολα κόμβων (από - προς):

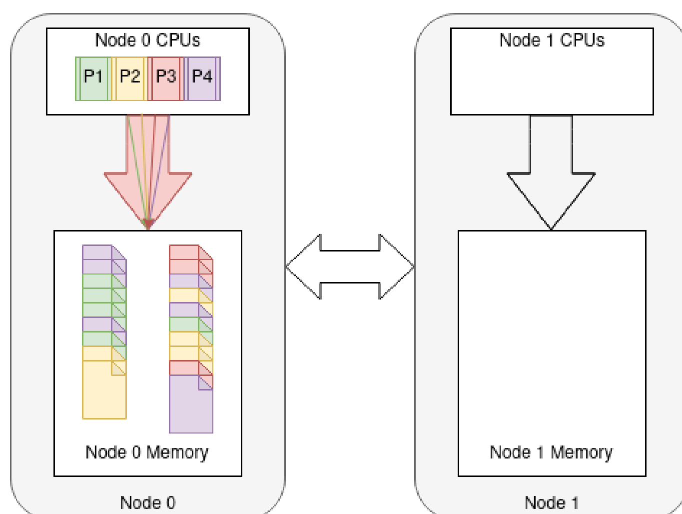
```
$ migratepages

usage: migratepages pid from-nodes to-nodes
```

Κεφάλαιο 3

Κίνητρο εργασίας

Όπως αναλύσαμε σε προηγούμενο κεφάλαιο, ο τρόπος με τον οποίο το Linux χρησιμοποιεί τις δυνατότητες των NUMA αρχιτεκτονικών προκειμένου να βελτιώσει την απόδοση των εφαρμογών που εκτελούνται σε αυτές είναι δεσμεύοντας μνήμη από το NUMA node στο οποίο εκτελείται η εφαρμογή, μειώνοντας έτσι το χρόνο πρόσβασης στη μνήμη (local allocation). Παρακάτω θα δούμε ότι υπάρχουν περιπτώσεις στις οποίες η δευσμεύση μνήμης κατ'αυτόν τον τρόπο δεν αποτελεί τη βέλτιστη λύση και μάλιστα μπορεί να μειώσει την απόδοση ενός συνόλου εφαρμογών.

Θα μελετήσουμε την περίπτωση όπου πολλές εφαρμογές εκτελούνται στο ίδιο NUMA node με local allocations. Ένα τέτοιο σενάριο είναι πολύ πιθανό να συναντηθεί σε servers ενός datacenter, όπου πολλές διαφορετικές εφαρμογές μπορούν να εκτελούνται ταυτόχρονα. Ένα παράδειγμα για αυτό το σενάριο φαίνεται στο Σχήμα 3.1, όπου οι διεργασίες P1, P2, P3, P4 εκτελούνται όλες στο node 0 και έχουν δεσμεύσει τη μνήμη τους σε αυτό.



Σχήμα 3.1: Παράδειγμα memory congestion σε ένα NUMA node.

Διεξάγουμε το εξής πείραμα: Σε ένα μηχάνημα με 2 NUMA nodes και 22 CPU cores / node (τα πλήρη χαρακτηριστικά του μηχανήματος αναφέρονται στον πίνακα

5.1) δοκιμάζουμε να τρέξουμε μια εφαρμογή ταξινόμησης ακεραίων (Integer Sort, NAS benchmarks [17]) σε μια τυχαία CPU του node 0 χωρίς να τρέχουν άλλες διεργασίες ταυτόχρονα, και μετράμε το χρόνο εκτέλεσής της. Στη συνέχεια δοκιμάζουμε να προσθέσουμε επιπλέον διεργασίες στις CPUs του ίδιου node, αρχικά 10 και στη συνέχεια 21, 43, 65 και 87. Συγκεκριμένα, προσθέτουμε διεργασίες stress [26] οι οποίες δεσμεύουν ένα μέρος της μνήμης και στη συνέχεια κάνουν τυχαία accesses σε αυτό, δημιουργώντας έτσι κίνηση μεταξύ των cpus και της μνήμης και άρα αυξάνοντας τη συμφόρηση στη σύνδεση αυτή. Ένα παράδειγμα τέτοιας διεργασίας φαίνεται στο 3.1, όπου εκτελούμε 1 thread που δεσμεύει 8GB μνήμης στην οποία κάνει access κάθε 4096 byte.

Όλες οι εφαρμογές δεσμεύουν μνήμη από το node στις CPU του οποίου εκτελούνται, δηλαδή το node 0. Κάθε φορά μετράμε το χρόνο εκτέλεσης της αρχικής μας εφαρμογής (Integer Sort) και υπολογίζουμε το ποσοστό μεταβολής από το base case, όταν δηλαδή αυτή εκτελείται μόνη της.

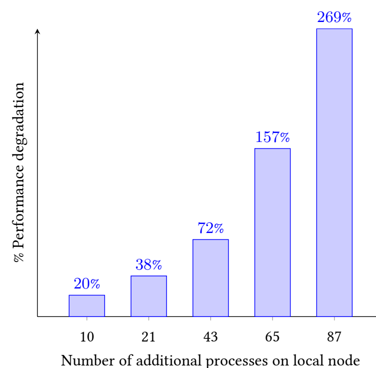
Listing 3.1: Παράδειγμα εκτέλεσης διεργασίας stress

```
$ stress -m 1 --vm-bytes 8000M --vm-keep --vm-stride 4096
```

Παρακάτω βλέπουμε τα αποτελέσματα αυτού του πειράματος:

Num. of additional processes on local node	Execution Time (s)	Exec. Time Degradation %
0	987	0%
10	1188	20%
21	1365	38%
43	1702	72%
65	2538	157%
87	3638	269%

Πίνακας 3.1: Χρόνος εκτέλεσης 1 διεργασίας για διαφορετικό αριθμό διεργασιών που εκτελούνται ταυτόχρονα στο ίδιο NUMA node.



Σχήμα 3.2: Ποσοστό αύξησης χρόνου εκτέλεσης διεργασίας για διαφορετικό αριθμό διεργασιών που εκτελούνται ταυτόχρονα στο ίδιο NUMA node.

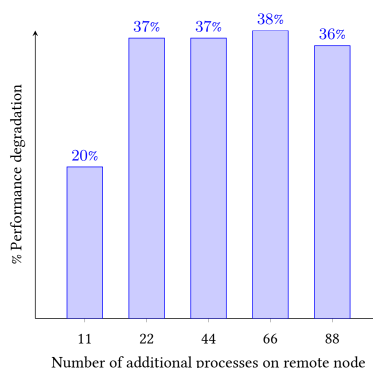
Βλέπουμε ότι ο χρόνος εκτέλεσης των εφαρμογών αυξάνεται κατά πολύ όσο αυξάνεται ο αριθμός των εφαρμογών που εκτελούνται ταυτόχρονα. Φυσικά το αποτέλεσμα αυτό μας φαίνεται αναμενόμενο επειδή όταν 2 ή περισσότερες εφαρμογές εκτελούνται στην ίδια CPU υπάρχει ανταγωνισμός μεταξύ τους σε επίπεδο χρονοδρομολόγησης. Επειδή όμως η μνήμη είναι ένας πόρος του συστήματος τον οποίο οι εφαρμογές μοιράζονται, εικάζουμε η απόδοση των εφαρμογών θα μπορούσε να βελτιωθεί εάν καταφέρναμε να μειώσουμε τον ανταγωνισμό μεταξύ των εφαρμογών σε επίπεδο μνήμης.

Για να δείξουμε ότι μέρος της μείωσης της απόδοσης της εφαρμογής οφείλεται σε συμφόρηση στο NUMA node, εκτελέσαμε το ακόλουθο πείραμα: Χρησιμοποιήσαμε πάλι την εφαρμογή Integer Sort, της οποίας έχουμε μετρήσει το χρόνο εκτέλεσης όταν εκτελείται μόνη της στο σύστημα. Στη συνέχεια τοποθετήσαμε στις CPU του node 1 πολλές διεργασίες stress, αντίστοιχες με αυτές του προηγούμενου πειράματος, οι οποίες όμως δεσμέυουν μνήμη από το node 0, από αυτό δηλαδή από το οποίο δεσμεύει και η εφαρμογή που μας ενδιαφέρει. Δοκιμάζουμε να αλλάξουμε το πλήθος αυτών των διεργασιών, και κάθε φορά μετράμε το χρόνο εκτέλεσης της εφαρμογής ταξινόμησης.

Με τον τρόπο αυτό πετυχαίνουμε να μην υπάρχει ανταγωνισμός των εφαρμογών για πόρους όπως CPU, αλλά αποκλειστικά λόγω μνήμης. Τα αποτελέσματα αυτού του πειράματος φαίνονται παρακάτω:

Num. of additional processes on remote node	Execution Time (s)	Exec. Time Degradation %
0	987	0%
11	1188	20%
22	1354	37%
44	1352	37%
66	1364	38%
88	1345	36%

Πίνακας 3.2: Χρόνος εκτέλεσης 1 διεργασίας για διαφορετικό αριθμό διεργασιών που εκτελούνται ταυτόχρονα στο απομακρυσμένο NUMA node.



Σχήμα 3.3: Ποσοστό αύξησης χρόνου εκτέλεσης διεργασίας για διαφορετικό αριθμό διεργασιών που εκτελούνται ταυτόχρονα στο απομακρυσμένο NUMA node.

Παρατηρούμε ότι ο χρόνος εκτέλεσης της εφαρμογής που μας ενδιαφέρει αυξάνεται όσο αυξάνουμε τον αριθμό των εφαρμογών που δεσμεύουν μνήμη στον ίδιο κόμβο με αυτήν, παρά το γεγονός ότι τρέχουν σε CPUs του άλλου node. Βλέπουμε έτσι ότι η συμφόρηση που δημιουργείται στο NUMA node οδηγεί σε μείωση της απόδοσης μιας εφαρμογής.

Στη συνέχεια αυτής της διπλωματικής θα μελετήσουμε έναν τρόπο με τον οποίο μπορούμε να λύσουμε αυτό το πρόβλημα. Συγκεκριμένα, θα δούμε ότι εάν επιλέξουμε να δεσμεύσουμε ένα μέρος της μνήμης των εφαρμογών σε ένα απομακρυσμένο NUMA node, μπορούμε να μειώσουμε τη συμφόρηση στο τοπικό και να πετύχουμε έτσι βελτίωση στην απόδοση των εφαρμογών.

Κεφάλαιο 4

Υλοποίηση

4.1 Σύνοψη

Σε προηγούμενο κεφάλαιο περιγράψαμε αναλυτικά τις δυνατότητες που μας προσφέρει το Linux σε NUMA αρχιτεκτονικές. Είδαμε επίσης ότι όταν έχουμε πολλές εφαρμογές να εκτελούνται στο ίδιο NUMA node υπάρχει σημαντική μείωση της απόδοσης του συστήματος όσο αυξάνεται ο αριθμός των εφαρμογών, και είδαμε ότι μέρος της μείωσης αυτής οφείλεται σε συμφόρηση στο NUMA node.

Σε αυτό το κεφάλαιο θα υλοποιήσουμε μια νέα πολιτική μνήμης στο Linux η οποία θεωρούμε ότι μπορεί να χρησιμοποιηθεί σε περιπτώσεις όπως η παραπάνω. Αφορμή για την υλοποίησή μας ήταν η έρευνα στο [4] όπου μελετήθηκε παρόμοιο πρόβλημα σε πλατφόρμα VMware ESXi. Οι συγγραφείς παρατήρησαν ότι στην περίπτωση ενός συστήματος που παρουσιάζει congestion, η απόδοση των εφαρμογών μπορεί να αυξηθεί εάν ένα ποσοστό της μνήμης τους γίνει allocate σε απομακρυσμένο κόμβο.

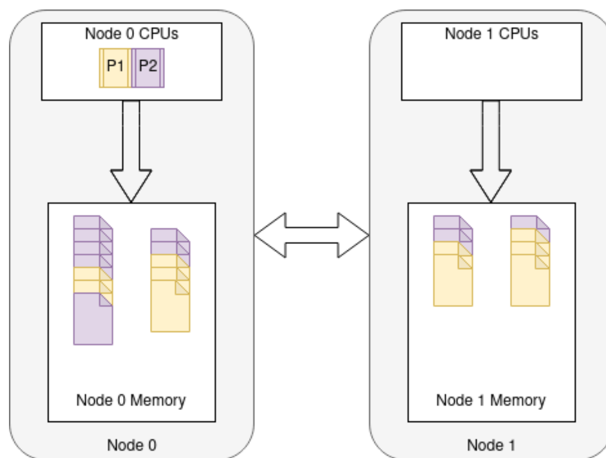
Παρατηρούμε ότι οι υπάρχουσες πολιτικές μνήμης του Linux, όπως αυτές περιγράφηκαν σε προηγούμενη ενότητα, δεν είναι αρκετές για να πειραματιστούμε με την παραπάνω ιδέα. Θα μπορούσαμε να πούμε ότι η πολιτική interleave είναι ίσως η πιο κοντινή σε αυτό που ζητάμε, αφού προσπαθεί να πετύχει μια ισοκατανομή του συνόλου της μνήμης της εφαρμογής στο σύνολο των κόμβων που της δίνουμε ως όρισμα. Κάτι τέτοιο όμως δεν είναι αρκετό, αφού δεν μπορούμε να μελετήσουμε περιπτώσεις όπως π.χ. το 30% της μνήμης να βρίσκεται remote και το υπόλοιπο 70% local.

Έτσι, προσθέσαμε στον πυρήνα του Linux μια νέα πολιτική η οποία μας επιτρέπει να δεσμεύσουμε ένα ποσοστό της μνήμης σε απομακρυσμένο node και το υπόλοιπο τοπικά. Θα ονομάζουμε την πολιτική αυτή percentage allocation. Στο Σχήμα 4.1 φαίνεται ένα παράδειγμα 2 διεργασιών που εκτελούνται σε ένα NUMA σύστημα με 2 nodes. Οι διεργασίες τρέχουν στο node 0 και έχουν δεσμεύσει ένα ποσοστό της μνήμης τους στο απομακρυσμένο node 1. Η P1 έχει το 50% της μνήμης της στο remote node, ενώ η P2 το 30%.

4.2 Περιγραφή Υλοποίησης

Κατά το σχεδιασμό της υλοποίησής μας αποφασίσαμε τα εξής:

1. Μας ενδιαφέρει η πολιτική μας να μπορεί να οριστεί ανά διεργασία, κατά την εκκίνησή της. Δηλαδή δεν ασχοληθήκαμε με το πώς θα μπορούσε να οριστεί ανά



Σχήμα 4.1: Παράδειγμα 2 διεργασιών P1 και P2 με πολιτική percentage

VMA ή να αλλάξει κατά της διάρκεια εκτέλεσης της διεργασίας.

2. Ο απομακρυσμένος κόμβος δεν χρειάζεται να είναι ίδιος για κάθε allocation. Για παράδειγμα, έστω ότι τρέχουμε μια διεργασία σε μηχάνημα με συνολικά 4 NUMA nodes και ορίζουμε ένα ποσοστό 30% για remote allocations. Τότε το ποσοστό αυτό μπορεί να γίνει allocate σε οποιαδήποτε από τα 3 remote nodes, τυχαία. Προφανώς, στην περίπτωση που το σύστημά μας διαθέτει 2 NUMA nodes, ολόκληρο το remote ποσοστό θα γίνει allocate στο μοναδικό remote node.

Θέλουμε, δηλαδή, ο χρήστης να μπορεί να ξεκινάει μια διεργασία χρησιμοποιώντας την πολιτική percentage στην οποία θα δίνει ένα συγκεκριμένο ποσοστό ως παράμετρο.

Η πολιτική μνήμης που προσθέσαμε αναπαρίσταται στον πυρήνα από το mode `MPOL_PERCENTAGE`. Παρακάτω περιγράφουμε συνοπτικά τις βασικές δομές και συναρτήσεις που χρησιμοποιήσαμε.

Αρχικά χρειάστηκε να προσθέσουμε κάποια επιπλέον πεδία σε 2 kernel structs: το `mempolicy` και το `task_struct`.

Το `mempolicy` struct χρησιμοποιείται για να αποθηκεύσει ένα memory policy και τυχόν πληροφορίες που αυτό χρειάζεται, όπως για παράδειγμα το preferred node εάν χρησιμοποιείται η πολιτική preferred. Εμείς χρειάστηκε να προσθέσουμε ένα επιπλέον πεδίο `perc`, για να αποθηκεύσουμε το επιθυμητό ποσοστό που θα γίνει allocate σε remote node.

Το `task_struct` είναι αυτό στο οποίο αποθηκεύονται όλες οι πληροφορίες για ένα task του συστήματος, για παράδειγμα το pid, το όνομα της διεργασίας κτλ. Σε αυτό το struct προσθέσαμε 2 πεδία: ένα για να μετράμε τον αριθμό των allocations που έχουν γίνει μέχρι τώρα (`perc_total`), και ένα για να μετράμε το πόσα από αυτά έχουν γίνει remotely (`perc_rem`). Σημειώνουμε εδώ ότι στο `task_struct` υπάρχει ήδη ένας δείκτης προς το struct mempolicy του αντίστοιχου task, το οποίο και χρησιμοποιείται για να θέσουμε μια πολιτική μνήμης ανά διεργασία. Επίσης, στον κώδικα του πυρήνα, μπορούμε να χρησιμοποιήσουμε το macro `current` για να αναφερθούμε στο `task_struct` του task το οποίο τρέχει.

Όταν λοιπόν ξεκινήσει η διεργασία, δημιουργούμε ένα `mempolicy` struct στο οποίο αποθηκεύουμε το επιθυμητό ποσοστό των remote allocations που έχει ορίσει ο χρήστης.

Στη συνέχεια και κατά τη διάρκεια εκτέλεσης της διεργασίας χρησιμοποιούμε τους μετρητές `perc_total` και `perc_rem` έτσι ώστε σε κάθε νέο allocation να μπορούμε να υπολογίσουμε το ποσοστό που έχει δεσμευτεί σε απομακρυσμένους κόμβους και να αποφασίζουμε εάν πρέπει να χρησιμοποιηθεί το τοπικό node ή κάποιο απομακρυσμένο. Στην περίπτωση που το allocation πρέπει να γίνει απομακρυσμένα, το node θα επιλεγεί τυχαία από συναρτήσεις του πυρήνα τις οποίες χρησιμοποιήσαμε ως έχουν, χωρίς δικές μας προσθήκες.

Δείχνουμε σε ψευδοκώδικα τις προσθήκες στις δομές `task_struct` και `mempolicy` καθώς και τις 2 βασικές συναρτήσεις του κωδικά μας: Τη SetPolicy που καλείται όταν ορίζεται η νέα πολιτική και την PercentageNode που καλείται σε κάθε allocation και επιστρέφει το node το οποίο πρέπει να χρησιμοποιηθεί.

Set percentage policy and return node for allocation

```
function SetPolicy(MPOL_PERCENTAGE, perc)
    struct mempolicy mpol ← mpol_new()          ▷ Create new mempolicy struct
    mpol.mode ← MPOL_PERCENTAGE
    mpol.perc ← perc
    current.perc_rem ← 0          ▷ current is a pointer to the current task struct
    current.perc_total ← 0
    current.mempolicy ← mpol
end function
function PercentageNode(a)
    if current.perc_total ≥ 100 then
        current.perc_rem ← 0
        current.perc_total ← 0
    end if
    current.perc_total ++
    if current.perc_rem < current.mempolicy.perc then
        current.perc_rem ++
        return random_node
    else
        return current_node
    end if
end function
```

/include/linux/sched.h

```
struct task_struct {
    ....
    struct mempolicy *mempolicy;
    ....
    +    unsigned long perc_rem;
    +    unsigned long perc_total;
}
```

Όπως φαίνεται παραπάνω, ανά 100 allocations κάνουμε τα `perc` από αυτά σε απομακρυσμένο κόμβο και τα υπόλοιπα στον τοπικό. Καταλήξαμε σε αυτή την λύση επειδή ήταν αρκετά απλή, αλλά θα μπορούσαμε να χρησιμοποιήσουμε μια διαφορετική τεχνική; για παράδειγμα η `PercentageNode` να επιστρέφει απομακρυσμένο κόμβο με

/include/linux/mempolicy.h

```
struct mempolicy {
    ....
    +    unsigned long perc; /* percentage */
}
```

πιθανότητα perc και τον τοπικό με 100-perc.

Επαναλαμβάνουμε εδώ η επιλογή του remote κόμβου γίνεται τυχαία. Σε ένα σύστημα με 2 NUMA nodes, όπως θα δούμε στη συνέχεια, ο απομακρυσμένος κόμβος είναι πάντα ο ίδιος και δεν χρειάζεται να οριστεί από τον χρήστη. Στην περίπτωση που θέλουμε να μπορεί ο χρήστης να ορίσει τον ή τους απομακρυσμένους κόμβους στους οποίους θα γίνει allocate το δεδομένο ποσοστό θα χρειαστεί να γίνουν επιπλέον μετατροπές στον κώδικα ώστε η πολιτική percentage να δέχεται παραπάνω arguments.

Ο κώδικάς μας με τις προσθήκες στον πυρήνα είναι διαθέσιμος στο [12], ενώ οι προσθήκες στο numactl στο [19].

4.3 Χρήση / Interface

Για τον ορισμό του NUMA policy μιας διεργασίας χρησιμοποιείται το εργαλείο numactl [20]. Χρειάστηκε, επομένως, πέρα από τις παραπάνω προσθήκες στον κώδικα του πυρήνα, να κάνουμε τροποποιήσεις στον κώδικα του numactl ώστε να υποστηρίζει τη νέα πολιτική. Προσθέσαμε έτσι το argument -b/-percentage το οποίο θέτει την πολιτική της εφαρμογής σε MPOL_PERCENTAGE με ποσοστό αυτό που ορίζει ο χρήστης.

Το numactl χρησιμοποιεί την κλήση συστήματος set_mempolicy [25] για τον ορισμό της πολιτικής μνήμης μιας διεργασίας. Προτιμήσαμε κι εμείς να χρησιμοποιήσουμε αυτήν την κλήση και να μην προσθέσουμε μια νέα, οπότε χρησιμοποιήσαμε το τελευταίο όρισμα της set_mempolicy για να δώσουμε το ποσοστό των remote allocations. Ο τρόπος κλήσης της set_mempolicy για να ορίσουμε την πολιτική percentage φαίνεται στο 4.1.

Listing 4.1: Κλήση set_mempolicy

```
set_mempolicy(MPOL_PERCENTAGE, NULL, perc};
```

Σε αυτό το σημείο πρέπει να τονίσουμε ότι για να χρησιμοποιηθεί η πολιτική percentage χρειάζεται να απενεργοποιηθεί το NUMA balancing στον πυρήνα [8], καθώς η τεχνική αυτή μπορεί να μεταφέρει περιοδικά σελίδες από τα απομακρυσμένα nodes στο τοπικό και να αλλάξει το ποσοστό όπως το έχουμε ορίσει.

Στο 4.2 φαίνεται ένα παράδειγμα χρήσης της πολιτικής μας με το numactl. Εκτελούμε μια διεργασία stress η οποία θα κάνει allocate 8000MB μνήμης. Τα ορίσματα -C 5 -b 30 του numactl είναι αυτά τα οποία ρυθμίζουν τη CPU στην οποία θα τρέξει η διεργασία (CPU 5) και το ποσοστό των remote allocations (30%). Το μηχάνημά μας έχει 2 NUMA nodes και η CPU 5 ανήκει στο node 0, όπως βλέπουμε στο 4.3 . Περιμένουμε λοιπόν να δούμε ότι το 70% των 8000MB θα γίνει allocate στο node 0 και το υπόλοιπο 30% στο node 1. Αυτό μπορεί να επιβεβαιωθεί χρησιμοποιώντας το εργαλείο numastat, όπως φαίνεται στο 4.4.

Listing 4.2: Εφαρμογή πολιτικής percentage

```
$ numactl -C 5 -b 30 stress -m 1 --vm-bytes 8000M --vm-keep --vm-stride 4096
```

Listing 4.3: Χαρακτηριστικά μηχανήματος

```
$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
node 0 size: 209595 MB
node 0 free: 203660 MB
node 1 cpus: 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
node 1 size: 209635 MB
node 1 free: 207070 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

Listing 4.4: Κατανομή μνήμης σε nodes με remote percentage 30%

```
$ numastat -p 2927

Per-node process memory usage (in MBs) for PID 2927 (stress)
                           Node 0          Node 1           Total
                  --------------- --------------- ---------------
Huge                         0.00            0.00            0.00
Heap                         0.00            0.00            0.00
Stack                        0.00            0.01            0.01
Private                   5600.38         2400.08         8000.46
----------------  --------------- --------------- ---------------
Total                     5600.38         2400.09         8000.48
```

4.4 Μεταφορά Σελίδων

Η νέα πολιτική percentage μας επιτρέπει να ρυθμίσουμε μια διεργασία να δεσμεύσει ένα ποσοστό της μνήμης της σε remote NUMA nodes, αλλά αυτό μπορεί να γίνει μόνο κατά την εκκίνηση της διεργασίας. Μας ενδιαφέρει τώρα να δούμε έαν υπάρχει τρόπος το ποσοστό αυτό να αλλάξει κατά της διάρκεια εκτέλεσης, αφού κάτι τέτοιο μας προσφέρει μεγαλύτερη ευελιξία.

Όπως ήδη αναφέραμε, το Linux μας παρέχει 2 κλήσεις συστήματος για μεταφορά σελίδων: τη migrate_pages, η οποία μεταφέρει ολόκληρη τη μνήμη της εφαρμογής, και τη move_pages [16] που χρησιμοποιείται για τη μεταφορά συγκεκριμένων σελίδων της εφαρμογής. Υπάρχει επίσης το εργαλείο migratepages, το οποίο αποτελεί μέρος του numactl project, που χρησιμοποιεί το migrate_pages system call και μας επιτρέπει τη μεταφορά ολόκληρης της μνήμης μιας εκτελούμενης εφαρμογής χωρίς να χρειαστεί να γράψουμε επιπλέον κώδικα.

Για τη μεταφορά ενός ποσοστού σελίδων, αρχικά σκεφτήκαμε να προσθέσουμε στη migrate_pages ένα επιπλέον όρισμα και να αλλάξουμε τον κώδικα του πυρήνα, αντίστοιχα με την δημιουργία του percentage policy. Κάτι τέτοιο όμως δεν είναι απαραίτητο από τη στιγμή που έχουμε στη διάθεσή μας τη move_pages. Μπορούμε να γράψουμε ένα C πρόγραμμα το οποίο βρίσκει όλες της σελίδες της επιθυμητής εφαρμογής διαβάζοντας το `/proc/<pid>/maps` [27], και, αφού επιλέξει ένα ποσοστό αυτών καλεί τη move_pages για τη μεταφορά τους.

Στο Παράρτημα αυτής της εργασίας παραθέτουμε ένα παράδειγμα χρήσης της move_pages, το οποίο μπορεί να μετατραπεί κατάλληλα ώστε να επιτρέπει τη μεταφορά ενός ποσοστού της μνήμης.

Κεφάλαιο 5

Αξιολόγηση

5.1 Υποδομή

Για την αξιολόγηση της υλοποίησής μας χρησιμοποιήσαμε το μηχάνημα broady. Πρόκειται για ένα Intel μηχάνημα με 44 επεξεργαστικούς πυρήνες (CPU cores) οι οποίοι υποστηρίζουν πολυ-νηματισμό (multi-threading) με 2 νήματα ο καθένας. Συνολικά δηλαδή υπάρχουν 88 εικονικές CPU (vCPUs). Ο κάθε πυρήνας διαθέτει 64KB L1 cache (32KB instruction + 32KB data) και 256KB L2 cache.

Το broady είναι βασισμένο σε NUMA αρχιτεκτονική και διαθέτει 2 NUMA κόμβους (NUMA nodes). Κάθε NUMA node αντιστοιχεί σε 22 φυσικές CPU ή 44 εικονικές. Η LLC (Last-Level Cache) ή L3 cache είναι κοινή για όλες τις CPUs ενός NUMA node και έχει μέγεθος 56320KB. Το μηχάνημα διαθέτει επίσης συνολικά 512 GM μνήμης RAM, ή 256 GB για κάθε NUMA node. Το λειτουργικό σύστημα είναι Debian GNU/Linux 8.3 (jessie) και η έκδοση του πυρήνα είναι η 4.7.

Τα χαρακτηριστικά συνοψίζονται στον παρακάτω πίνακα:

CPU Model	Intel(R) Xeon(R) CPU E5-2699 v4
CPUs	88
Threads/core	2
Cores/socket	22
Sockets	2
Core freq.	2.20GHz
NUMA nodes	2 (0-1)
L1d cache	32KB
L1i cache	32KB
L2 cache	256KB
L3 cache	56320KB
Total RAM	512GB

Πίνακας 5.1: Χαρακτηριστικά πειραματικού μηχανήματος (broady)

Οι αποστάσεις μεταξύ των NUMA nodes, όπως μετρήθηκαν με τη χρήση του numactl, είναι:

	Node 0	Node 1
Node 0	10	21
Node 1	21	10

Πίνακας 5.2: Πίνακας NUMA αποστάσεων broady

Λόγω του ότι η υλοποίησή μας έγινε σε επίπεδο πυρήνα, προτιμήσαμε να χρησιμοποιήσουμε μια εικονική μηχανή (virtual machine - VM) η οποία εκτελείται μέσα στο broady και είναι ρυθμισμένη με τέτοιο τρόπο ώστε να έχει παρόμοια αρχιτεκτονική με το φυσικό μηχάνημα, εκτελώντας παράλληλα το δικό μας πυρήνα. Για την δημιουργία του VM χρησιμοποιήθηκε το λογισμικό QEMU emulator version 4.1.1 [24]σε συνδυασμό με το υποσύστημα KVM (Kernel-based Virtual Machine) [11] του πυρήνα του Linux.

Για να μιμηθούμε την αρχιτεκτονική του broady επιλέξαμε να δώσουμε στο VM 44 CPUS χωρισμένες σε 2 sockets και 2 NUMA nodes, με 208 GB μνήμης ανά node. Σε αυτό το σημείο χρειάστηκε προσοχή, επειδή το VM αποτελεί μια ακόμα διεργασία μέσα στο host μηχάνημα, να γίνουν οι κατάλληλες ρυθμίσεις στο QEMU ώστε να υπάρχει 1-προς-1 αντιστοιχία μεταξύ των CPUS του VM και αυτών του host, όπως και μεταξύ των NUMA nodes.

Στο VM χρησιμοποιήσαμε το λειτουργικό σύστημα Debian GNU/Linux 10 (buster) και τον πυρήνα 4.19 τροποποιημένο με τις δικές μας προσθήκες.

5.2 Εργαλεία

5.2.1 perf

Το perf [23], είναι ένα εργαλείο που παρέχει το Linux με σκοπό τη μελέτη της συμπεριφοράς συγκεκριμένων εφαρμογών ή και του συνολικού συστήματος. Βασίζεται στο perf_events interface του πυρήνα του Linux, και χρησιμοποιεί performance counters και tracepoints για τη συλλογή μετρήσεων κατά τη διάρκεια εκτέλεσης των εφαρμογών. Οι μετρήσεις μπορεί να αφορούν πράγματα όπως branch misses, cache misses/hits, page faults κ.α.

Το perf παρέχει διάφορες υπο-εντολές οι οποίες μπορούν να χρησιμοποιηθούν για διαφορετικούς σκοπούς, για παράδειγμα μέτρηση γεγονότων για μια εφαρμογή, καταγραφή κλήσεων κάποιου kernel function, δειγματοληψία γεγονότων ανά ένα χρονικό διάστημα κ.α. Εμείς χρησιμοποιήσαμε την υπο-εντολή stat για να πάρουμε πληροφορίες για την απόδοση της εφαρμογής που μας ενδιέφερε. Ένα παράδειγμα εκτέλεσης του perf stat είναι το εξής:

Στο παράδειγμα βλέπουμε πληροφορίες όπως χρόνο εκτέλεσης, IPC, branch misses. Εκτός από αυτά το perf υποστηρίζει πολλούς ακόμα μετρητές, κάποιοι από τους οποίους εξαρτώνται από το υλικό του μηχανήματος (hardware events) και άλλοι οι οποίοι είναι επιπέδου πυρήνα (software events). Μπορούμε να δούμε μια λίστα μετρητών οι οποίοι είναι διαθέσιμοι στο συστημά μας εκτελώντας την εντολή `perf list`.

Στην εργασία αυτή χρησιμοποιήσαμε την έκδοση 4.19.132 του perf για να μετρήσουμε το IPC (insn per cycle) και το χρόνο εκτέλεσης των εφαρμογών που μελετήσαμε.

Listing 5.1: perf stat

```
user@debian:~$ perf stat numactl -C 13 -b 30 ./omnetpp_r_base -c General -r 0
# started on Thu Nov 12 12:01:08 2020


 Performance counter stats for 'numactl -C 13 -b 30 ./omnetpp_r_base -c General -
    ↪ r 0':

        1003985.01 msec task-clock:u              #    0.332 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
             70501      page-faults:u             #    0.070 K/sec
     2173406361069      cycles:u                  #    2.165 GHz
     1046192946735      instructions:u            #    0.48  insn per cycle
      228061618827      branches:u                #  227.156 M/sec
        5003383712      branch-misses:u           #    2.19% of all branches

    3026.838518999 seconds time elapsed

    1008.655077000 seconds user
       0.296651000 seconds sys
```

5.3 Μεθοδολογία

Μας ενδιαφέρει να δούμε τη μεταβολή στην απόδοση ενός συνόλου εφαρμογών όταν ένα ποσοστό της μνήμης τους έχει δεσμευτεί στο απομακρυσμένο NUMA node, σε σχέση με την απόδοση των ίδιων εφαρμογών όταν όλη τους η μνήμη έχει δεσμευτεί στο τοπικό node (base case).

Επιλέξαμε να μελετήσουμε τις παρακάτω εφαρμογές από τα μετροπρογράμματα SPEC2017 και SPEC2006:

Όνομα	Κατηγορία
505.mcf_r	SPEC2017
519.lbm_r	SPEC2017
520.omnetpp_r	SPEC2017
459.GemsFDTD	SPEC2K6

Πίνακας 5.3: Μετροπρογράμματα

Τα μετροπρογράμματα αυτά επιλέχθηκαν έχοντας ως κριτήριο το ότι είναι αρκετά εξαρτημένες από τη μνήμη και, όπως είδαμε σε προηγούμενο κεφάλαιο, η απόδοσή τους μειώνεται σημαντικά όταν αυξάνεται ο αριθμός των διεργασιών που εκτελούνται ταυτόχρονα στο ίδιο NUMA node.

Προτιμήσαμε επίσης τα μετροπρογράμματα να είναι όλα single-threaded. Αυτό κάνει πιο εύκολη την τοποθέτησή τους στις CPUs και την παρακολούθηση της μνήμης τους για τους σκοπούς της ερευνάς μας. Παρ'όλα αυτά η πολιτική μας δοκιμάστηκε και σε multi-threaded εφαρμογές όπου και παρουσιάζει την αναμενόμενη συμπεριφορά.

Για κάθε μία από αυτές τις εφαρμογές, δοκιμάζουμε να τρέξουμε δεδομένο αριθμό διεργασιών σε όλες τις CPU του NUMA node 0, δεσμεύοντας κάθε φορά ενα ποσοστό

της συνολικής μνήμης στον απομακρυσμένο κόμβο (node 1). Με χρήση του εργαλείου perf μετράμε τον χρόνο εκτέλεσης (σε δευτερόλεπτα) και τον αριθμό των εντολών που εκτελούνται ανά κύκλο ρολογιού (Instructions Per Cycle - IPC) για κάθε μια διεργασία. Στη συνέχεια υπολογίζουμε το μέσο όρο όλων των χρόνων εκτέλεσης καθώς και των IPC. Τόσο το IPC όσο και ο χρόνος εκτέλεσης αποτελούν ευρέως διαδεδομένες μετρικές για την αξιολόγηση της απόδοσης μιας εφαρμογής.

Για λόγους απλότητας, στη συνέχεια θα αναφερόμαστε στον μέσο όρο των χρόνων εκτέλεσης όλων των εφαρμογών ως "Χρόνο Εκτέλεσης" (Execution Time) και στο μέσο όρο όλων των αριθμών εντολών ανά κύκλο ρολογιού ώς IPC.

5.3.1 Μετρικές

Δεδομένου του ότι μας ενδιαφέρει η συνολική απόδοση του συστήματος και όχι μιας μεμονωμένης εφαρμογής, η αξιολόγησή μας πρέπει να βασιστεί σε μετρικές οι οποίες λαμβάνουν υπ'όψιν κάθε διεργασία που εκτελείται. Θέλουμε επίσης οι μετρικές αυτές να δείχνουν τη μεταβολή της απόδοσης του συστήματος σε σχέση με το base case το οποίο είναι όταν όλες οι εφαρμογές που εκτελούνται σε CPUs του node 0 δεσμεύουν και όλη τους τη μνήμη από αυτό.

Με βάση αυτά τα κριτήρια επιλέχθηκαν οι:

1. Ποσοστό βελτίωσης χρόνου εκτέλεσης:

$$ExecTimeImp_{perc} = \frac{\overline{ExecTime_0} - \overline{ExecTime_{perc}}}{\overline{ExecTime_0}} * 100\%$$

   Το ποσοστό βελτίωσης του μέσου χρόνου εκτέλεσης (Execution Time Improvement) μας δείχνει τη διαφορά στο χρόνο εκτέλεσης των εφαρμογών σε σχέση με το base case. Για τον υπολογισμό του χρησιμοποιούμε κάθε φορά τους μέσους όρους των χρόνων εκτέλεσης όλων των διεργασιών (instances του κάθε benchmark) που εκτελούνται για ένα δεδομένο ποσοστό $perc$. Θετικές τιμές εκφράζουν βελτίωση (μείωση) του μέσου χρόνου εκτέλεσης, ενώ αντίθετα αρνητικές δείχνουν επιπλέον αύξηση σε σχέση με το base case.

2. Weighted Speedup:

$$WeightedSpeedup_{perc} = \frac{1}{N} \sum_{i=1}^{N} \frac{IPC_{i,perc}}{IPC_{i,0}}$$

   Το Weighted Speedup [1]δείχνει τη μεταβολή στο IPC. Εκφράζεται ως το άθροισμα των IPC μιας διεργασίας όταν ένα ποσοστό perc έχει δεσμευτεί στον απομακρυσμένο κόμβο προς το αντίστοιχο IPC στο basecase. Εάν WeightedSpeedup > 1 η απόδοση των εφαρμογών βελτιώθηκε, αλλιώς η επίδραση του remote allocation ήταν αρνητική.

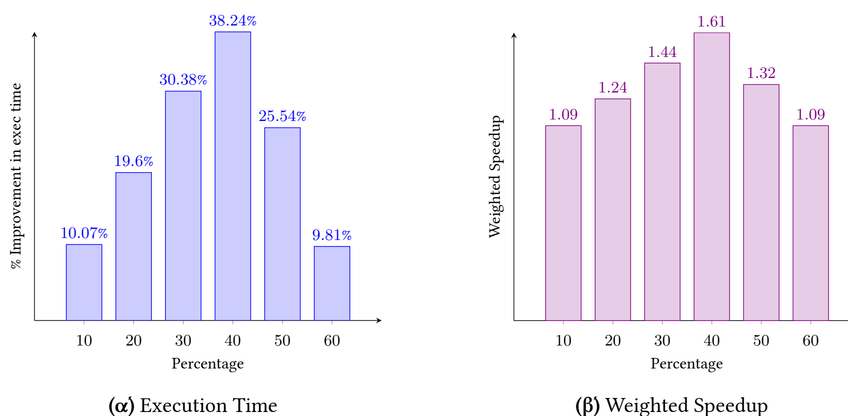
5.4 Αποτελέσματα

Για κάθε εκτέλεση του benchmark καταγράφουμε το ποσοστό της μνήμης το οποίο δεσμεύτηκε στον απομακρυσμένο κόμβο,το μέσο χρόνο εκτέλεσης (Execution Time), το ποσοστό βελτίωσης του χρόνου σε σχέση με το base case (Exec. Time Improvement), το IPC και το Weighted Speedup.

Παρακάτω παρουσιάζουμε πίνακες και γραφήματα με τα αποτελέσματα.

Allocation Percentage	Execution Time (s)	Exec. Time Improvement %	IPC	Weighted Speedup
0	8005	0	0.34	1
10	7199	10.07%	0.37	1.09
20	6436	19.60%	0.42	1.24
30	5573	30.38%	0.49	1.44
40	**4944**	**38.24%**	**0.55**	**1.61**
50	5961	25.54%	0.45	1.32
60	7220	9.81%	0.37	1.09
70	8444	-5.49%	0.32	0.94
100	12140	-51.66%	0.22	0.65

Πίνακας 5.4: Αποτελέσματα LBM



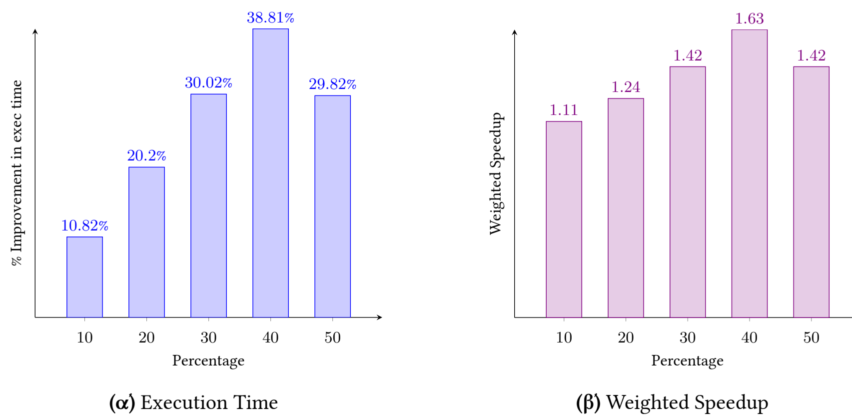
(α) Execution Time

(β) Weighted Speedup

Σχήμα 5.1: Αποτελέσματα LBM

Allocation Percentage	Execution Time (s)	Exec. Time Improvement %	IPC	Weighted Speedup
0	5273	0	0.53	1
10	4702	10.82%	0.59	1.11
20	4207	20.20%	0.65	1.24
30	3690	30.02%	0.75	1.42
40	**3227**	**38.81%**	**0.86**	**1.63**
50	3700	29.82%	0.75	1.42

Πίνακας 5.5: Αποτελέσματα GemsFDTD



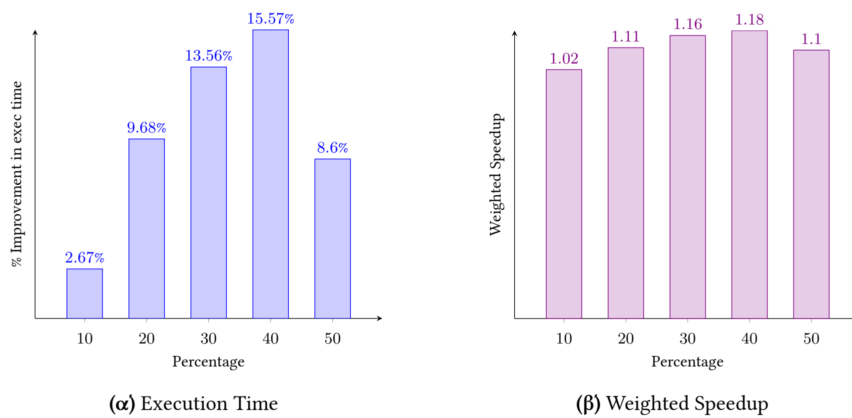
(α) Execution Time

(β) Weighted Speedup

Σχήμα 5.2: Αποτελέσματα GemsFDTD

Allocation Percentage	Execution Time (s)	Exec. Time Improvement %	IPC	Weighted Speedup
0	3638	0	0.49	1
10	3543	2.67%	0.5	1.02
20	3286	9.68%	0.54	1.11
30	3145	13.56%	0.56	1.16
40	**3071**	**15.57%**	**0.58**	**1.18**
50	3325	8.60%	0.54	1.10

Πίνακας 5.6: Αποτελέσματα MCF



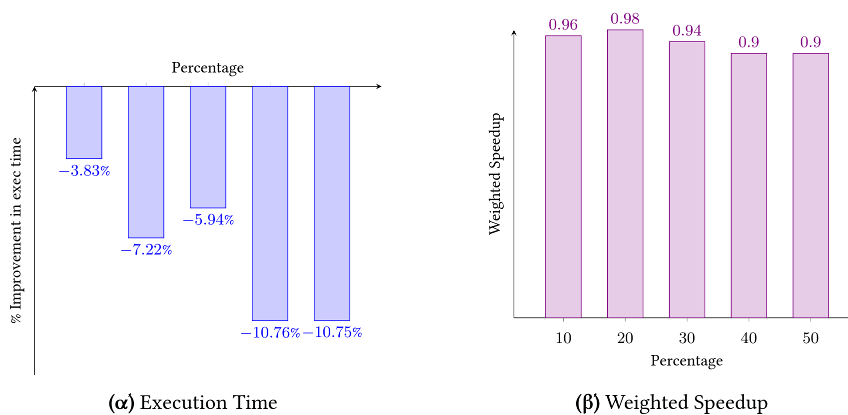
(α) Execution Time

(β) Weighted Speedup

Σχήμα 5.3: Αποτελέσματα MCF

Allocation Percentage	Execution Time (s)	Exec. Time Improvement %	IPC	Weighted Speedup
0	3906	0	0.5	1
10	4056	-3.83%	0.48	0.96
20	4188	-7.22%	0.49	0.98
30	4138	-5.94%	0.47	0.94
40	4327	-10.76%	0.45	0.9
50	4326	-10.75%	0.45	0.9

Πίνακας 5.7: Αποτελέσματα Omnetpp



(α) Execution Time

(β) Weighted Speedup

Σχήμα 5.4: Αποτελέσματα omnetpp

5.5 Συμπεράσματα

Παρατηρούμε ότι σε 3 benchmarks (MCF, LBM, GemsFDTD) μπορούμε να πετύχουμε βελτίωση της απόδοσης του συστήματος δεσμεύοντας ποσοστά από 10% έως 50% της συνολικής μνήμης σε απομακρυσμένο NUMA node. Συγκεκριμένα, η μέγιστη απόδοση για όλα τα benchmarks επιτυγχάνεται για ποσοστό 40%, όπου μπορούμε να έχουμε μείωση του συνολικού χρόνου εκτέλεσης έως και 39%. Για ποσοστά άνω του 60% παρατηρούμε ότι η απόδοση του συστήματος μειώνεται αισθητά σε σχέση με το base case.

Το αποτέλεσμα αυτό είναι λογικό αφού για μεγάλα ποσοστά μνήμης αρχίζει να παρουσιάζεται συμφόρηση στον απομακρυσμένο κόμβο, πράγμα που σε συνδυασμό με το μεγαλύτερο χρόνο πρόσβασης για απομακρυσμένη μνήμη οδηγεί σε σημαντική αύξηση του χρόνου εκτέλεσης.

Αντίθετα, στο omnetpp είδαμε ότι η δεσμεύση απομακρυσμένης μνήμης δεν οδηγεί σε μείωση αλλά σε αύξηση του συνολικού χρόνου εκτέλεσης, και άρα η πολιτική μνήμης percentage δεν μπορεί να μας βοηθήσει να βελτιώσουμε την απόδοση. Αυτό μπορεί να οφείλεται στο ότι το omnetpp δεν εξαρτάται από τη μνήμη τόσο όσο τα υπόλοιπα benchmarks, και άρα η μείωση της συμφόρησης στο NUMA node δεν είναι αρκετή για να έχουμε καλύτερη απόδοση. Μια άλλη ένδειξη που μας οδηγεί σε αυτό το συμπέρασμα είναι ότι ο χρόνος εκτέλεσης αυξάνεται αρκετά αργά όταν αυξάνουμε το ποσοστό, δηλαδή η εξάρτηση από τη μνήμη δεν φαίνεται τόσο έντονα όσο στα υπόλοιπα benchmarks.

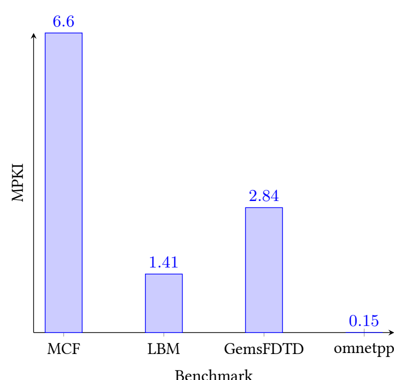
Μπορούμε να δούμε πόσο εξαρτώμενη είναι μια εφαρμογή από τη μνήμη χρησιμο-

ποιώντας τη μετρική MPKI (Misses Per Kilo Instructions) η οποία ορίζεται ως ο αριθμός των σφαλμάτων που παρουσιάζονται στη Last Level Cache ανά 1000 εντολές:

$$MPKI = \frac{LLCLoadMisses + LLCStoreMisses}{TotalInstructions} * 1000$$

Η μετρική MPKI χρησιμοποιείται συχνά για τη μελέτη της συμπεριφοράς μιας εφαρμογής όσον αφορά τη μνήμη σε μια δεδομένη αρχιτεκτονική. Όσο μικρότερη τιμή έχει, τόσο λιγότερα misses συμβαίνουν στην LLC και άρα χρειάζεται να γίνονται λιγότερα accesses στην κύρια μνήμη. Επομένως, σε μια δεδομένη αρχιτεκτονική, μπορούμε να πούμε ότι όσο μεγαλύτερο MPKI έχει μια εφαρμογή, τόσο περισσότερο εξαρτάται από την κύρια μνήμη, και άρα ο χρόνος εκτέλεσής της εφαρμογής μπορεί να βελτιωθεί εάν μειωθεί ο χρόνος πρόσβασης στη μνήμη.

Με τη χρήση του εργαλείου perf υπολογίσαμε το MPKI για κάθε ένα από τα μετροπρογράμματα:



Σχήμα 5.5: MPKI per benchmark

Πράγματι, βλέπουμε ότι τα MCF, LBM, GemsFDTD έχουν σημαντικά μεγαλύτερο MPKI σε σχέση με το omnetpp.

Συμπεραίνουμε λοιπόν ότι σε ένα σύνολο εφαρμογών που παρουσιάζουν σημαντική εξάρτηση από τη μνήμη, η νέα πολιτική percentage μπορεί φανεί χρήσιμη προκειμένου να μειώθεί η συμφόρηση ενός NUMA node και να βελτιωθεί η συνολική απόδοση των εφαρμογών.

Κεφάλαιο 6

Σχετική Έρευνα

Λόγω των αυξημένων δυνατοτήτων τους, οι NUMA αρχιτεκτονικές κυριαρχούν πλέον στα σύγχρονα υπολογιστικά συστήματα. Παρ'όλα αυτά, η αξιοποίηση ενός NUMA συστήματος ώστε να πετύχουμε την καλύτερη δυνατή επίδοση των εφαρμογών που εκτελούνται σε αυτό δεν αποτελεί ένα τετριμμένο πρόβλημα. Με την ταυτόχρονη αύξηση του πλήθους των εφαρμογών που μπορούν να εκτελούνται σε ένα μηχάνημα, των δεδομένων που χρησιμοποιούν οι εφαρμογές καθώς και της πολυπλοκότητας των NUMA αρχιτεκτονικών, γίνεται ξεκάθαρο ότι η αντίληψη για μέγιστη τοπικότητα των δεδομένων γίνεται πλέον παρωχημένη.

Έχει γίνει έτσι εκτενής έρευνα πάνω στο πώς επηρεάζονται οι εφαρμογές από την τοποθέτησή τους στα NUMA nodes καθώς και σε τρόπους τοποθέτησής τους οι οποίοι δεν πετυχαίνουν απαραίτητα τη μέγιστη τοπικότητα. Παραθέτουμε εδώ μερικές εργασίες οι οποίες ασχολούνται με αυτά τα θέματα και οι οποίες μας φάνηκαν ενδιαφέρουσες στην έρευνά μας.

Στο [2] οι Z. Majo και T. R. Gross δείχνουν ότι σε περιπτώσεις μηχανημάτων με υψηλό φόρτο εργασίας ο δίαυλος επικοινωνίας μεταξύ των NUMA nodes μπορεί να έχει καλύτερη απόδοση από τον τοπικό, on-chip memory controller. Έτσι οι συνηθισμένες πρακτικές της τοπικότητας δεδομένων (data locality) δεν είναι πλέον βέλτιστες. Αυτό είναι ακριβώς το φαινόμενο που παρατηρήσαμε και εμείς.

Στο [3] οι συγγραφείς, βασισμένοι στην παρατήρηση ότι τυχόν συμφόρηση σε ένα NUMA node επιφέρει καθυστερήσεις στην εκτέλεση των εφαρμογών, προτείνουν τον αλγόριθμο Carrefour για την τοποθέτηση της μνήμης. Ο αλγόριθμος αυτός χρησιμοποιεί υπάρχουσες τεχνικές όπως μεταφορά και αντιγραφή σελίδων της μνήμης τις οποίες όμως εφαρμόζει αφού πάρει αποφάσεις βασισμένες στη συνολική απόδοση του συστήματος και τη συμφόρηση που παρουσιάζει.

Ο αλγόριθμος Carrefour χρησιμοποιείται, μαζί με άλλες πολιτικές, στο [5] όπου οι συγγραφείς μελετούν το πως μια εφαρμογή σε ένα περιβάλλον εικονικοποίησης επηρεάζεται από το NUMA policy που χρησιμοποιείται. Συγκεκριμένα δείχνουν ότι μια πολιτική που προσπαθεί να πετύχει όσο το δυνατόν μεγαλύτερη τοπικότητα δεδομένων δεν είναι πάντα η βέλτιστη, και έτσι υλοποιούν επιπλέον πολιτικές για τον επόπτη (hypervisor) Xen.

Στο [4] οι συγγραφείς προσπαθούν να λύσουν το πρόβλημα μείωσης της απόδοσης ενός συνόλου εφαρμογών όσο αυξάνεται ο αριθμός τους δοκιμάζοντας τη δέσμευση ενός ποσοστού της μνήμης σε απομακρυσμένα NUMA memory nodes, στην πλατφόρμα VMware ESXi. Αφού δείξουν ότι αυτή η προσέγγιση έχει θετικά αποτελέσματα, τη

χρησιμοποιούν σε ενα ”Congestion-Aware Memory Allocation (CAMA)” μηχανισμό ο οποίος αποφασίζει πώς θα γίνονται τα allocations μιας νέας εφαρμογής στο σύστημα με βάση την παρούσα κατάστασή του και το πόσο congested είναι. Επίσης υλοποιούν το ”Congestion-Aware Page Migration (CAPM)”, δηλαδή την αυτόματη μεταφορά σελίδων μιας εφαρμογής κατά τη διάρκεια εκτέλεσής της βάση της συμφόρησης του συστήματος. Δείχνουν ότι τα CAMA και CAPM μπορούν να χρησιμοποιηθούν παράλληλα σε ένα σύστημα και να επιφέρουν βελτίωση της απόδοσής έως και 9.5%. Η εργασία αυτή σχετίζεται άμεσα με τη δουλειά μας σε αυτή τη διπλωματική, καθώς εμείς υλοποιήσαμε σε Linux μια αντίστοιχη πολιτική όπου ορίζουμε το ποσοστό των remote allocations.

Αντίστοιχη έρευνα έχει γίνει και στις λεγόμενες ασύμμετρες τοπολογίες NUMA. Στις τοπολογίες αυτές οι αποστάσεις μεταξύ των κόμβων δεν είναι απαραίτητα συμμετρικές, για παράδειγμα ένα thread το οποίο εκτελείται στο node 0 έχει διαφορετικό bandwith για πρόσβαση στη μνήμη του node 1 από το bandwidth ενός thread που εκτελείται στο node 1 και προσπελαύνει τη μνήμη του node 0. Επίσης, σε ασύμμετρες τοπολογίες μπορούμε να έχουμε πολλά NUMA nodes, κάποια από τα οποία είναι άμεσα συνδεδεμένα ενώ άλλα επικοινωνούν με multi-hop μονοπάτια. Μια συμμετρική τοπολογία μπορεί επίσης να συμπεριφέρεται σαν ασύμμετρη λόγω της συμφόρησης που μπορεί να παρουσιάζεται σε ένα ή περισσότερα nodes.

Με ασύμμετρα NUMA συστήματα, και συγκεκριμένα με πολυνηματικές, memory-intensive εφαρμογές που εκτελούνται σε τέτοια, ασχολούνται τα [6] και [7].

Στο [7] παρουσιάζεται η πολιτική `bw-interleaved` η οποία λαμβάνοντας υπ'όψιν τα bandwidths μεταξύ των κόμβων υπολογίζει τα ιδανικά ”βάρη” για το allocation σε καθέναν από αυτούς και στη συνέχεια δεσμεύει μνήμη από κάθε node ανάλογα με το βάρος του. Η υλοποίηση γίνεται ως μέρος της βιβλιοθήκης libnuma.

Αντίστοιχα στο [6] παρουσιάζεται ένας αλγόριθμος για Asymmetry-Aware Page Placement (AAPP) που χτίζει ένα μοντέλο της πιθανής συμπεριφοράς μιας εφαρμογής και με βάση αυτό υπολογίζει τα βάρη που πρέπει να χρησιμοποιηθούν στο interleave allocation. Ο αλγόριθμος υλοποιείται σαν εναλλακτική των `mmap` και `malloc`.

Οι 2 αυτές εργασίες κατανέμουν τη μνήμη στα διαφορετικά nodes με μη ομοιόμορφο τρόπο, σε αντίθεση με την πολιτική interleave. Σχετίζονται έτσι στενά με την πολιτική percentage, αφού και εμείς πετυχαίνουμε διαφορετικό ποσοστό remote και local allocations.

Κεφάλαιο 7

Επίλογος

7.1 Συμπεράσματα

Ο στόχος της διπλωματικής αυτής ήταν να φτιάξουμε μια πολιτική μνήμης η οποία να μας επιτρέπει να δεσμεύσουμε συγκεκριμένο ποσοστό της μνήμης μιας διεργασίας σε έναν απομακρυσμένο κόμβο και να μελετήσουμε κατά πόσο αυτή μπορεί να φανεί χρήσιμη σε περιπτώσεις όπου ένα υπολογιστικό σύστημα παρουσιάζει congestion. Πράγματι, η πολιτική percentage που δημιουργήσαμε μας επιτρέπει να δεσμεύσουμε διαφορετικά ποσοστά της μνήμης απομακρυσμένα και, όπως είδαμε στο Κεφάλαιο 5, μπορεί να βελτιώσει την απόδοση ενός συνόλου εφαρμογών.

7.2 Μελλοντικές Επεκτάσεις

- **Congestion-aware memory allocation/migration**: Η πολιτική percentage είναι αρκετά εύελικτη επειδή μας επιτρέπει να πειραματιστούμε με το ποσοστό που γίνεται allocate στον απομακρυσμένο κόμβο. Για το λόγο αυτό θα είχε ενδιαφέρον να χρησιμοποιηθεί σε κάποιο σύστημα το οποίο θα εντοπίζει πότε η μνήμη ενός node εμφανίζει congestion, και αντίστοιχα να μεταβάλλει το ποσοστό των remote allocations για τις νέες ή/και τις ήδη εκτελούμενες διεργασίες.

- **Μελέτη περισσότερων workloads και διαφορετικών NUMA αρχιτεκτονικών**: Σε αυτή την εργασία δοκιμάσαμε την πολιτική percentage σε αρχιτεκτονική με 2 NUMA nodes, και με συγκεκριμένα μετροπρογράμματα. Θεωρούμε ότι θα είναι χρήσιμη η δοκιμή της πολιτικής σε αρχιτεκτονικές με παραπάνω NUMA nodes και με επιπλέον workloads, τα οποία θα μπορούσαν να είναι ανομοιογενή (πολλές διαφορετικές εφαρμογές αντί για πολλά instances της ίδιας εφαρμογής) ή/και multi-threaded.

- **Επιλογή του απομακρυσμένου κόμβου**: Μια πιθανή επέκταση της πολιτικής μας είναι η δυνατότητα επιλογής του κόμβου από τον οποίο θα γίνει allocate το δεδομένο ποσοστό, ή ακόμα και η δυνατότητα ορισμού περισσότερων από ενός απομακρυσμένους κόμβους (στην περίπτωση μηχανήματος με περισσότερους από 2).

- **Μεταφορά σελίδων ή/και εφαρμογή του policy ταυτόχρονα με τη μεταφορά**: Όπως αναφέραμε, η μεταφορά ενός ποσοστού σελίδων μνήμης μεταξύ 2 ή περισσότερων nodes μπορεί να γίνει με χρήση της κλήσης συστήματος move_pages.

Δεν είναι όμως δυνατό να μεταφερθεί ένα ποσοστό σελίδων μιας διεργασίας και ταυτόχρονα να αλλάξει η πολιτική μνήμης της σε percentage. Φυσικά μπορούμε πρώτα να μεταφέρουμε τις σελίδες και στη συνέχεια να αλλάξουμε την πολιτική μνήμης ή και αντίστροφα, αλλά θεωρούμε πως θα είχε ενδιαφέρον να μελετηθεί εάν μπορούμε να κάνουμε τις 2 αυτές λειτουργίες με 1 μόνο κλήση συστήματος. Κάτι τέτοιο θα μπορούσε να είναι χρήσιμο σε ένα congestion-aware σύστημα όπως αυτό που περιγράψαμε παραπάνω.

Κεφάλαιο 8

Παράρτημα

8.1 Προσθήκη MPOL_PERCENTAGE στον κώδικα του πυρήνα 4.19 (patch)

```
diff --git a/include/linux/mempolicy.h b/include/linux/mempolicy.h
index 5228c62..c7f71ba 100644
--- a/include/linux/mempolicy.h
+++ b/include/linux/mempolicy.h
@@ -50,6 +50,7 @@ struct mempolicy {
                short            preferred_node; /* preferred */
                nodemask_t       nodes;          /* interleave/bind */
                /* undefined for default */
+               unsigned long    perc;           /* percentage */
        } v;
        union {
                nodemask_t cpuset_mems_allowed; /* relative to these nodes */
diff --git a/include/linux/sched.h b/include/linux/sched.h
index c69f308..f48d156 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -1010,6 +1010,8 @@ struct task_struct {
        struct mempolicy                *mempolicy;
        short                           il_prev;
        short                           pref_node_fork;
+       unsigned long                   perc_rem;
+       unsigned long                   perc_total;
 #endif
 #ifdef CONFIG_NUMA_BALANCING
        int                             numa_scan_seq;
diff --git a/include/uapi/linux/mempolicy.h b/include/uapi/linux/mempolicy.h
index 3354774..707966a 100644
--- a/include/uapi/linux/mempolicy.h
+++ b/include/uapi/linux/mempolicy.h
@@ -22,6 +22,7 @@ enum {
        MPOL_BIND,
        MPOL_INTERLEAVE,
        MPOL_LOCAL,
+       MPOL_PERCENTAGE,
        MPOL_MAX,       /* always last member of enum */
 };

diff --git a/mm/mempolicy.c b/mm/mempolicy.c
```

```
index 3cd27c1..481eb1b 100644
--- a/mm/mempolicy.c
+++ b/mm/mempolicy.c
@@ -191,6 +191,18 @@ static int mpol_new_bind(struct mempolicy *pol, const
    ↪ nodemask_t *nodes)
        return 0;
 }

+static int mpol_new_percentage(struct mempolicy *pol, const nodemask_t *nodes)
+{
+        int n;
+        for_each_online_node(n) {
+                if (n == numa_mem_id()) {
+                        printk(KERN_ALERT "mpol_new_percentage:␣current␣node:␣%d
    ↪ \n",
+                                n);
+                }
+        }
+        return 0;
+}
+
 /*
  * mpol_set_nodemask is called after mpol_new() to set up the nodemask, if
  * any, for the new policy.  mpol_new() has already validated the nodes
@@ -213,7 +225,9 @@ static int mpol_set_nodemask(struct mempolicy *pol,
                  cpuset_current_mems_allowed, node_states[N_MEMORY]);

        VM_BUG_ON(!nodes);
-       if (pol->mode == MPOL_PREFERRED && nodes_empty(*nodes))
+        if (pol->mode == MPOL_PERCENTAGE)
+                nodes = NULL;
+       else if (pol->mode == MPOL_PREFERRED && nodes_empty(*nodes))
                nodes = NULL;   /* explicit local allocation */
        else {
                if (pol->flags & MPOL_F_RELATIVE_NODES)
@@ -271,6 +285,9 @@ static struct mempolicy *mpol_new(unsigned short mode,
    ↪ unsigned short flags,
                    (flags & MPOL_F_RELATIVE_NODES))
                        return ERR_PTR(-EINVAL);
                mode = MPOL_PREFERRED;
+        } else if (mode == MPOL_PERCENTAGE) {
+                // Accept empty nodemask
+                ;
        } else if (nodes_empty(*nodes))
                return ERR_PTR(-EINVAL);
        policy = kmem_cache_alloc(policy_cache, GFP_KERNEL);
@@ -401,6 +418,10 @@ static const struct mempolicy_operations mpol_ops[MPOL_MAX]
    ↪  = {
                .create = mpol_new_bind,
                .rebind = mpol_rebind_nodemask,
        },
+        [MPOL_PERCENTAGE] = {
+                .create = mpol_new_percentage,
+                .rebind = mpol_rebind_nodemask,
+        },
 };

 static int migrate_page_add(struct page *page, struct list_head *pagelist,
@@ -791,7 +812,7 @@ static int mbind_range(struct mm_struct *mm, unsigned long
    ↪ start,

 /* Set the process memory policy */
```

```
 static long do_set_mempolicy(unsigned short mode, unsigned short flags,
-                             nodemask_t *nodes)
+                             nodemask_t *nodes, unsigned long percentage)
 {
        struct mempolicy *new, *old;
        NODEMASK_SCRATCH(scratch);
@@ -817,6 +838,13 @@ static long do_set_mempolicy(unsigned short mode, unsigned
    ↪ short flags,
        current->mempolicy = new;
        if (new && new->mode == MPOL_INTERLEAVE)
                current->il_prev = MAX_NUMNODES-1;
+        if (new && new->mode == MPOL_PERCENTAGE) {
+                current->perc_rem = 0;
+                current->perc_total = 0;
+                new->v.perc = percentage;
+                printk(KERN_ALERT "mpol_new_percentage:␣percentage␣set:␣%ld\n",
+                        new->v.perc);
+        }
        task_unlock(current);
        mpol_put(old);
        ret = 0;
@@ -839,6 +867,8 @@ static void get_policy_nodemask(struct mempolicy *p,
    ↪ nodemask_t *nodes)
        switch (p->mode) {
        case MPOL_BIND:
                /* Fall through */
+        case MPOL_PERCENTAGE:
+                /* Fall through */
        case MPOL_INTERLEAVE:
                *nodes = p->v.nodes;
                break;
@@ -1423,17 +1453,23 @@ static long kernel_set_mempolicy(int mode, const
    ↪ unsigned long __user *nmask,
        int err;
        nodemask_t nodes;
        unsigned short flags;
+        unsigned long percentage = 0;

        flags = mode & MPOL_MODE_FLAGS;
        mode &= ~MPOL_MODE_FLAGS;
        if ((unsigned int)mode >= MPOL_MAX)
                return -EINVAL;
+        if ((unsigned int)mode == MPOL_PERCENTAGE) {
+                //MPOL_PERCENTAGE
+                percentage = maxnode;
+                maxnode = 1;
+        }
        if ((flags & MPOL_F_STATIC_NODES) && (flags & MPOL_F_RELATIVE_NODES))
                return -EINVAL;
        err = get_nodes(&nodes, nmask, maxnode);
        if (err)
                return err;
-       return do_set_mempolicy(mode, flags, &nodes);
+       return do_set_mempolicy(mode, flags, &nodes, percentage);
 }

 SYSCALL_DEFINE3(set_mempolicy, int, mode, const unsigned long __user *, nmask,
@@ -1822,6 +1858,30 @@ static unsigned interleave_nodes(struct mempolicy *policy
    ↪ )
        return next;
 }
```

```
+/* Return a node id for MPOL_PERCENTAGE */
+static int percentage_node(struct mempolicy *policy)
+{
+        struct task_struct *me = current;
+        int ran;
+
+        do {
+            ran = node_random(&current->mems_allowed);
+        } while (ran == numa_mem_id() || !(node_online(ran)));
+
+        if (me->perc_total >= 100) {
+            me->perc_total = 0;
+            me->perc_rem = 0;
+        }
+        me->perc_total++;
+        if (me->perc_rem < policy->v.perc) {
+            // Keep allocating on remote
+            me->perc_rem++;
+            return ran;
+        } else {
+            return numa_mem_id();
+        }
+}
+
 /*
  * Depending on the memory policy provide a node from which to allocate the
  * next slab entry.
@@ -1863,6 +1923,10 @@ unsigned int mempolicy_slab_node(void)
                return z->zone ? zone_to_nid(z->zone) : node;
        }

+        case MPOL_PERCENTAGE: {
+                return percentage_node(policy);
+        }
+
        default:
                BUG();
        }
@@ -2102,6 +2166,14 @@ alloc_pages_vma(gfp_t gfp, int order, struct
    ↪ vm_area_struct *vma,
                goto out;
        }

+        if (pol->mode == MPOL_PERCENTAGE) {
+                unsigned nid;
+                nid = percentage_node(pol);
+                page = __alloc_pages(gfp, order, nid);
+                mpol_cond_put(pol);
+                goto out;
+        }
+
        if (unlikely(IS_ENABLED(CONFIG_TRANSPARENT_HUGEPAGE) && hugepage)) {
                int hpage_node = node;

@@ -2263,6 +2335,8 @@ bool __mpol_equal(struct mempolicy *a, struct mempolicy *b
    ↪ )
        switch (a->mode) {
        case MPOL_BIND:
                /* Fall through */
+        case MPOL_PERCENTAGE:
+                /* Fall through */
        case MPOL_INTERLEAVE:
```

```
                return !!nodes_equal(a->v.nodes, b->v.nodes);
        case MPOL_PREFERRED:
@@ -2758,7 +2832,7 @@ void __init numa_policy_init(void)
        if (unlikely(nodes_empty(interleave_nodes)))
                node_set(prefer, interleave_nodes);

-       if (do_set_mempolicy(MPOL_INTERLEAVE, 0, &interleave_nodes))
+       if (do_set_mempolicy(MPOL_INTERLEAVE, 0, &interleave_nodes, 0))
                pr_err("%s: interleaving failed\n", __func__);

        check_numabalancing_enable();
@@ -2767,7 +2841,7 @@ void __init numa_policy_init(void)
 /* Reset policy of current process to default */
 void numa_default_policy(void)
 {
-       do_set_mempolicy(MPOL_DEFAULT, 0, NULL);
+       do_set_mempolicy(MPOL_DEFAULT, 0, NULL, 0);
 }

 /*
@@ -2784,6 +2858,7 @@ static const char * const policy_modes[] =
        [MPOL_BIND]       = "bind",
        [MPOL_INTERLEAVE] = "interleave",
        [MPOL_LOCAL]      = "local",
+        [MPOL_PERCENTAGE] = "percentage",
 };


@@ -2960,6 +3035,8 @@ void mpol_to_str(char *buffer, int maxlen, struct
    ↪ mempolicy *pol)
        case MPOL_INTERLEAVE:
                nodes = pol->v.nodes;
                break;
+        case MPOL_PERCENTAGE:
+                break;
        default:
                WARN_ON_ONCE(1);
                snprintf(p, maxlen, "unknown");
```

8.2 Προσθήκη -b/--percentage argument στο numactl (patch)

```
diff --git a/libnuma.c b/libnuma.c
index 88f479b..68192b4 100644
--- a/libnuma.c
+++ b/libnuma.c
@@ -1792,6 +1792,20 @@ void numa_set_localalloc(void)
        setpol(MPOL_DEFAULT, numa_no_nodes_ptr);
 }

+/**
+  *  Use MPOL_PERCENTAGE:
+  *  calls set_mempolicy with the desired percentage
+  *  as the third argument (maxnode).
+  */
+void numa_set_percentage(unsigned long perc)
+{
+        printf("Setting policy %d with perc %ld\n", MPOL_PERCENTAGE, perc);
+        struct bitmask *bmp;
```

```
+        bmp = numa_no_nodes_ptr;
+        if (set_mempolicy(MPOL_PERCENTAGE, bmp->maskp, perc) < 0)
+            numa_error("set_mempolicy");
+}
+
 SYMVER("numa_bind_v1", "numa_bind@libnuma_1.1")
 void numa_bind_v1(const nodemask_t *nodemask)
 {
diff --git a/numa.h b/numa.h
index 288c2ca..c41abc9 100644
--- a/numa.h
+++ b/numa.h
@@ -192,6 +192,9 @@ void numa_set_localalloc(void);
 /* Only allocate memory from the nodes set in mask. 0 to turn off */
 void numa_set_membind(struct bitmask *nodemask);

+/* Allocate a percentage of total memory on a remote node */
+void numa_set_percentage(unsigned long);
+
 /* Return current membind */
 struct bitmask *numa_get_membind(void);

diff --git a/numactl.c b/numactl.c
index df9dbcb..03d1db9 100644
--- a/numactl.c
+++ b/numactl.c
@@ -45,6 +45,7 @@ struct option opts[] = {
        {"membind", 1, 0, 'm'},
        {"show", 0, 0, 's' },
        {"localalloc", 0,0, 'l'},
+        {"percentage", 1,0, 'b'},
        {"hardware", 0,0,'H' },

        {"shm", 1, 0, 'S'},
@@ -67,7 +68,8 @@ void usage(void)
        fprintf(stderr,
                "usage: numactl [--all | -a] [--interleave= | -i <nodes>] [--
    ↪ preferred= | -p <node>]\n"
                "                                [--physcpubind= | -C <cpus>] [--
    ↪ cpunodebind= | -N <nodes>]\n"
-               "                                [--membind= | -m <nodes>] [--
    ↪ localalloc | -l] command args ...\n"
+               "                [--membind= | -m <nodes>] [--localalloc | -l]\n"
+                "                [--percentage= | -b <perc>] command args ...\n"
                "       numactl [--show | -s]\n"
                "       numactl [--hardware | -H]\n"
                "       numactl [--length | -l <length>] [--offset | -o <offset
    ↪ >] [--shmmode | -M <shmmode>]\n"
@@ -86,6 +88,7 @@ void usage(void)
                "  block:PATH the node of block device path\n"
                "  pci:[seg:]bus:dev[:func] The node of a PCI device\n"
                "<cpus> is a comma delimited list of cpu numbers or A-B ranges
    ↪ or all\n"
+                "<perc> is an integer defining the percentage to be allocated
    ↪ on a remote node\n"
                "all ranges can be inverted with !\n"
                "all numbers and ranges can be made cpuset-relative with +\n"
                "the old --cpubind argument is deprecated.\n"
@@ -413,9 +416,18 @@ static struct bitmask *numactl_parse_nodestring(char *s,
    ↪ int flag)
                return numa_parse_nodestring(s);
 }
```

```
+int numactl_parse_percentage(char *s)
+{
+       int percent;
+       percent = atoi(s);
+       return percent;
+}
+
+
 int main(int ac, char **av)
 {
        int c, i, nnodes=0;
+        unsigned long percentage;
        long node=-1;
        char *end;
        char shortopts[array_len(opts)*2 + 1];
@@ -451,6 +463,18 @@ int main(int ac, char **av)
                                numa_set_interleave_mask(mask);
                        checkerror("setting interleave mask");
                        break;
+                case 'b': /* --percentage */
+                        checknuma();
+                        //FIXME check value of perc
+                        percentage = numactl_parse_percentage(optarg);
+                        setpolicy(MPOL_PERCENTAGE);
+                        errno = 0;
+                        if (shmfd >= 0)
+                                printf("Not supported");
+                        else
+                                numa_set_percentage(percentage);
+                        checkerror("percentage allocation");
+                        break;
                case 'N': /* --cpunodebind */
                case 'c': /* --cpubind */
                        dontshm("-c/--cpubind/--cpunodebind");
diff --git a/numaif.h b/numaif.h
index 91aa230..b89b259 100644
--- a/numaif.h
+++ b/numaif.h
@@ -27,7 +27,8 @@ extern long move_pages(int pid, unsigned long count,
 #define MPOL_BIND        2
 #define MPOL_INTERLEAVE  3
 #define MPOL_LOCAL       4
-#define MPOL_MAX         5
+#define MPOL_PERCENTAGE  5
+#define MPOL_MAX         6

 /* Flags for get_mem_policy */
 #define MPOL_F_NODE    (1<<0)   /* return next il node or node of address */
diff --git a/util.c b/util.c
index a41818f..80574a8 100644
--- a/util.c
+++ b/util.c
@@ -90,6 +90,7 @@ static struct policy {
        { "membind",    MPOL_BIND, },
        { "preferred",   MPOL_PREFERRED, },
        { "default",    MPOL_DEFAULT, 1 },
+        { "percentage", MPOL_PERCENTAGE, },
        { NULL },
 };

diff --git a/versions.ldscript b/versions.ldscript
```

```
index 23074a0..5d287f3 100644
--- a/versions.ldscript
+++ b/versions.ldscript
@@ -54,6 +54,7 @@ libnuma_1.1 {
     numa_set_localalloc;
     numa_set_membind;
     numa_set_preferred;
+    numa_set_percentage;
     numa_set_strict;
     numa_setlocal_memory;
     numa_tonode_memory;
```

8.3 Παράδειγμα χρήσης move_pages

```
/*
 * Find all pages of a process by reading /proc/<pid>/maps, and
 * move them to a given NUMA node.
 *
 * Link with -lnuma.
 */

#include <errno.h>
#include <numaif.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define FILENAMELEN         256
#define LINELEN             256
#define PAGE_SIZE           4096
#define PROC_DIR_NAME       "/proc"
#define MAPS_NAME           "maps"

void usage(void)
{
        fprintf(stderr,
                "usage: collect_and_move <pid> <target_node>\n");
        exit(1);
}

int main(int argc, char *argv[])
{
        FILE *m   = NULL;
        int n, pid, target_node, ret = 0;
        int pagesize = getpagesize();
        char m_name[FILENAMELEN];
        char line[LINELEN];

        if (argc != 3)
                usage();

        // argv[1] = PID of the process to move
        pid = atoi(argv[1]);

        // argv[2] = target node
        target_node = atoi(argv[2]);

        // Open pid/maps file
```

```
    sprintf(m_name, ”%s/%d/%s”, PROC_DIR_NAME, pid, MAPS_NAME);
    m = fopen(m_name, ”r”);
    if (m == NULL)
    {
            printf(”Unable␣to␣open␣%s␣for␣reading␣(errno=%d)␣(5).n”, m_name,
↪ errno);
            return -1;
    }

    // Structures for move_pages
    void** page_addr_new = NULL;
    void** page_addr = NULL;
    int* status;
    int* nodes;
    unsigned long num_pages_total = 0;
    unsigned long num_pages = 0;
    unsigned long i;

    // Iterate over lines in the maps file
    // For each new line, add the pages to page_addr
    // in order to avoid calling move_pages multiple times
    while (fgets(line, LINELEN, m) != NULL)
    {
            unsigned long vm_start, vm_end;

            // Read start and end addresses
            n = sscanf(line, ”%lX-%lX”, &vm_start, &vm_end);
            if (n != 2)
            {
                printf(”Invalid␣line␣read␣from␣%s:␣%s␣(6)n”, m_name, line);
                continue;
            }

            // Count total number of pages
            num_pages = (vm_end - vm_start) / PAGE_SIZE;

            if (num_pages > 0)
            {
                    void* index = (void *) ((((long)vm_start) & ~((long)(
↪ pagesize - 1))) + pagesize);
                    void** page_addr_tmp = malloc(sizeof(char *) * num_pages
↪ );

                    for (i = 0; i < num_pages; i++) {
                            page_addr_tmp[i] = index + i * pagesize;
                    }

                    // Create a new array, larger by num_pages
                    page_addr_new = malloc(sizeof(char *) * (num_pages_total
↪  + num_pages));

                    // If this is the first iteration we only have to copy
↪ the new array
                    if (!page_addr) {
                            memcpy(page_addr_new, page_addr_tmp, num_pages *
↪  (sizeof(char *)));
                    } else {
                            // First copy previous values
                            memcpy(page_addr_new, page_addr, num_pages_total
↪  * sizeof(char *));
                            // Add new pages
                            memcpy(page_addr_new + num_pages_total,
```

```
    ↪ page_addr_tmp, num_pages * sizeof(char *));
                        }

                        // Increase total number of pages and save this array
                        num_pages_total += num_pages;
                        page_addr = page_addr_new;

                }
        }  /* while */

        num_pages = num_pages_total;

        // Create status and nodes arrays
        status = malloc(sizeof(int *) * num_pages);
        nodes = malloc(sizeof(int *) * num_pages);
        for (i = 0; i < num_pages; i++) {
                nodes[i] = target_node;
                status[i] = -123;
        }

        printf("Total number of pages = %ld. Starting move_pages...\n", num_pages)
    ↪ ;

        // Call move_pages
        ret = move_pages(pid, num_pages, page_addr, nodes, status, 0);
        if (ret < 0)
                perror("move_pages");

        // Count total number of errors
        unsigned long errors = 0;
        for (i = 0; i < num_pages; i++)
        {
                if (status[i] < 0) {
                        errors++;
                } else if (nodes[i] != target_node) {
                        errors++;
                }
        }

        printf("Total number of errors = %lu\n", errors);

        if (m != NULL)
        {
                fclose(m);
                m = NULL;
        }

        return ret;
}
```

Βιβλιογραφία

[1] D. M. Tullsen και J. A. Brown. “Handling long-latency loads in a simultaneous multithreading processor”. Στο: *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. 2001, σσ. 318–327. DOI: 10.1109/MICRO.2001.991129.

[2] Zoltan Majo και Thomas Gross. “Memory System Performance in a NUMA Multicore Multiprocessor”. Στο: Ιαν. 2011, σ. 12. DOI: 10.1145/1987816.1987832.

[3] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma και Mark Roth. “Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems”. Στο: τόμ. 48. Απρ. 2013, σσ. 381–394. DOI: 10.1145/2499368.2451157.

[4] J. B. Kotra, S. Kim, K. Madduri και M. T. Kandemir. “Congestion-aware memory management on NUMA platforms: A VMware ESXi case study”. Στο: *2017 IEEE International Symposium on Workload Characterization (IISWC)*. 2017, σσ. 146–155. DOI: 10.1109/IISWC.2017.8167772.

[5] Gauthier Voron, Gaël Thomas, Vivien Quema και Pierre Sens. “An interface to implement NUMA policies in the Xen hypervisor”. Στο: *Twelfth European Conference on Computer Systems, EuroSys 2017*. Belgrade, Serbia, Απρ. 2017, σ. 15. URL: https://hal.inria.fr/hal-01515359.

[6] David Gureya, Rodrigo Rodrigues, Paolo Romano, Pramod Bhatotia, Vivien Quema και Joao Barreto. “Asymmetry-aware Page Placement for Contemporary NUMA Architectures”. English. Στο: The 8th Workshop on Systems for Multi-core and Heterogeneous Architectures ; Conference date: 23-04-2018 Through 23-04-2018. Μαρ. 2018. URL: https://sites.google.com/site/sfma2018eurosys/.

[7] D. Gureya, J. Neto, R. Karimi, J. Barreto, P. Bhatotia, V. Quema, R. Rodrigues, P. Romano και V. Vlassov. “Bandwidth-Aware Page Placement in NUMA”. Στο: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, Μάι. 2020, σσ. 546–556. DOI: 10.1109/IPDPS47924.2020.00063.

[8] *Automatic NUMA Balancing*. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-numa-auto_numa_balancing.

[9] *Documentation for /proc/sys/kernel: numa_balancing.* URL: https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#numa-balancing.

[10] *get_mempolicy(2) - Linux manual page.* URL: https://man7.org/linux/man-pages/man2/get_mempolicy.2.html.

[11] *KVM: Kernel Virtual Machine.* URL: https://www.linux-kvm.org/page/Main_Page.

[12] *Linux kernel with NUMA percentage policy Github repo.* URL: https://github.com/chipflake/linux-numa-percentage.

[13] *mbind(2) - Linux manual page.* URL: https://man7.org/linux/man-pages/man2/mbind.2.html.

[14] *migrate_pages(2) - Linux manual page.* URL: https://man7.org/linux/man-pages/man2/migrate_pages.2.html.

[15] *migratepages(8) - Linux manual page.* URL: https://man7.org/linux/man-pages/man8/migratepages.8.html.

[16] *move_pages(2) - Linux manual page.* URL: https://man7.org/linux/man-pages/man2/move_pages.2.html.

[17] *NAS Parallel Benchmarks.* URL: https://www.nas.nasa.gov/publications/npb.html.

[18] *NUMA Memory Policy.* URL: https://www.kernel.org/doc/html/latest/admin-guide/mm/numa_memory_policy.html.

[19] *numactl-percentage Github repo.* URL: https://github.com/chipflake/numactl-percentage.

[20] *numactl: NUMA support for Linux.* URL: https://github.com/numactl/numactl.

[21] *numactl(8) - Linux manual page.* URL: https://man7.org/linux/man-pages/man8/numactl.8.html.

[22] *numastat(8) - Linux manual page.* URL: https://man7.org/linux/man-pages/man8/numastat.8.html.

[23] *perf: Linux profiling with performance counters.* URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[24] *QEMU: the FAST! processor emulator.* URL: https://www.qemu.org/.

[25] *set_mempolicy(2) - Linux manual page.* URL: https://man7.org/linux/man-pages/man2/set_mempolicy.2.html.

[26] *stress(1): impose load on/stress test systems.* URL: https://linux.die.net/man/1/stress.

[27] *The /proc Filesystem.* URL: https://www.kernel.org/doc/html/latest/filesystems/proc.html.

[28] *What is NUMA?* URL: https://www.kernel.org/doc/html/latest/vm/numa.html.