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

Τοποθέτηση Εφαρμογών σε NUMA Αρχιτεκτονικές λαμβάνοντας υπόψη το κόστος μετάφρασης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Πολίτης Βασίλειος-Ορέστης

Επιβλέπων : Γκούμας Γεώργιος
Αναπληρωτής Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2021

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

Τοποθέτηση Εφαρμογών σε NUMA Αρχιτεκτονικές λαμβάνοντας υπόψη το κόστος μετάφρασης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Πολίτης Βασίλειος-Ορέστης

Επιβλέπων: Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14η Ιουλίου 2021.

.....	.....	.....
Γκούμας Γεώργιος	Κοζύρης Νεκτάριος	Πνευματικάτος Διονύσιος
Αν. Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2021

.....
Πολίτης Βασίλειος-Ορέστης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Παρά τα πολλά πλεονεκτήματα που προσφέρει η εικονική μνήμη, αποτελεί ταυτόχρονα σημαντική αιτία καθυστέρησης των προγραμμάτων, λόγω της απαίτησης που θέτει για μετάφραση διεύθυνσης σε κάθε εντολή μνήμης. Για την αντιμετώπιση του φαινομένου, έχουν συμπεριληφθεί στους επεξεργαστές κρυφές μνήμες μεταφράσεων, όπως το TLB. Η απεικόνιση της εικονικής μνήμης στην φυσική με μπλοκ μεγάλου μεγέθους βοηθά τις κρυφές μνήμες αυτές στο να μειώσουν το κόστος μετάφρασης αποτελεσματικότερα. Από την άλλη, είναι ευρέως διαδεδομένα τα συστήματα κοινής μνήμης NUMA, όπου οι πυρήνες και η μνήμη του συστήματος διαμοιράζονται σε κόμβους, που συνδέονται μεταξύ τους μέσω ενός δικτύου διασύνδεσης. Βασικός κανόνας για την καλή επίδοση μίας εφαρμογής είναι τα δεδομένα της να τοποθετούνται στον ίδιο κόμβο με τα νήματα της, ώστε να αποφεύγεται η διάσχιση του δικτύου.
Στην παρούσα διπλωματική εργασία, αρχικά δείχνουμε ότι σε συνθήκες κατακερματισμένης τοπικής μνήμης, η πολιτική συν-τοποθέτησης μνήμης/νημάτων δεν είναι πάντα η βέλτιστη. Βασιζόμενοι σε αυτή την παρατήρηση, υλοποιούμε συστήματα που επιτρέπουν την απεικόνιση μνήμης μίας εφαρμογής και σε μακρινούς κόμβους, όταν αυτό κρίνεται απαραίτητο. Τα πειραματικά αποτελέσματα δείχνουν ότι τα συστήματα αυτά σε αρκετές περιπτώσεις πετυχαίνουν βελτίωση της επίδοσης, ενώ σπάνια την βλάπτουν.

Λέξεις κλειδιά
NUMA, εικονική μνήμη, κόστος μετάφρασης διευθύνσεων, πυρήνας Linux

Abstract

Despite its many advantages, virtual memory is also the cause of significant execution overhead because of the requirement it imposes for address translation in every memory access. As a countermeasure, cache memories storing recently used translations, such as TLB, are included in cpu cores. Large contiguous virtual to physical memory mappings are important for translation caches to reduce address translation overhead more effectively.
On the other side, NUMA has been and still is a popular computer architecture. In such systems, memory and cpu cores are grouped in discrete nodes, connected by an interconnect network, and forming a shared-memory system. Common wisdom for NUMA machines is to collocate application thread and data on a single node as to avoid costly remote memory accesses.
In this diploma thesis, we first show that under local memory fragmentation, thread and data collocation policy is not always optimal. Based on that observation, we implement systems that allow remote memory mappings when local node memory suffers from external fragmentation. Our experimental results show that such systems are in some cases indeed beneficial, while they rarely hurt performance.

Keywords
NUMA, virtual memory, address translation overhead, Linux kernel

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τους γονείς μου που μου έδωσαν και με το παραπάνω, τη δυνατότητα να σπουδάσω.
Ευχαριστώ επίσης τους καθηγητές της σχολής για τις γνώσεις και το ενδιαφέρον που μου μετέφεραν, ειδικά τον κύριο Γιώργο Γκούμα ως επιβλέποντα αυτής της εργασίας.
Αισθάνομαι τυχερός που καθ' όλη τη διάρκεια εκπόνησης της εργασίας είχα την συνεχή επικοινωνία, συμβουλές, καθοδήγηση και γενικότερα πολύτιμη βοήθεια της Χλόης Αλβέρτη, την οποία ευχαριστώ θερμά.
Τέλος, ευχαριστώ τον κύριο Βασίλη Καρακώστα για την συμμετοχή του στην επίβλεψη.

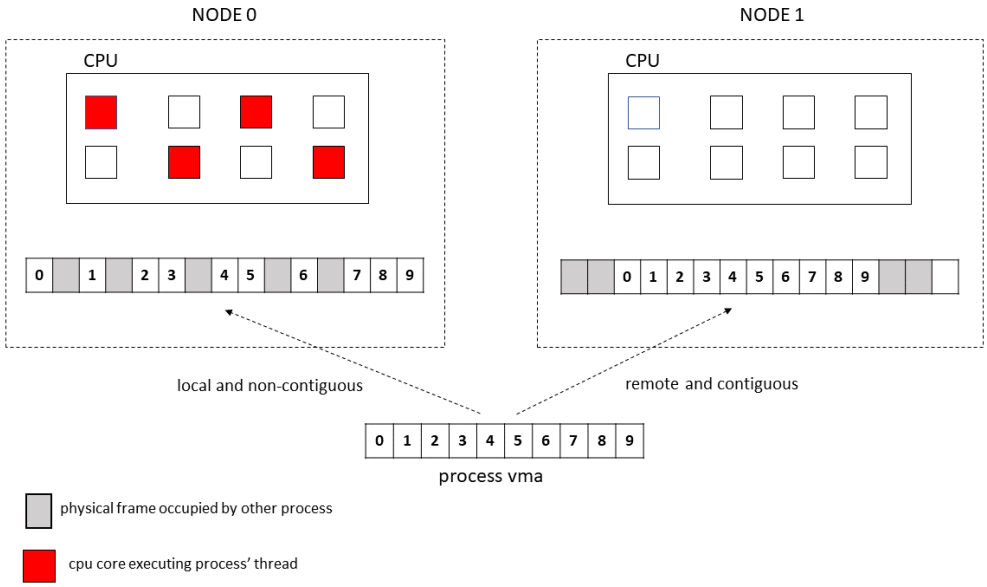
Πίνακας Περιεχομένων

Κατάλογος Σχημάτων

Κεφάλαιο 1 Εισαγωγή

Σήμερα είναι ευρέως διαδεδομένα τα συστήματα κοινής μνήμης NUMA, όπου επεξεργαστές και μνήμη οργανώνονται σε κόμβους που συνδέονται μεταξύ τους με ένα δίκτυο διασύνδεσης. Οι εφαρμογές, εφ' όσον το επιτρέπουν οι απαιτήσεις τους σε μνήμη και πυρήνες, εκτελούνται εντός ενός μόνο κόμβου προκειμένου να αποφευχθούν επιπλέον κόστη λόγω απομακρυσμένων προσβάσεων κύριας μνήμης.
Από την άλλη είναι ευρέως διαδεδομένη ως γνωστόν η εικονική μνήμη, λόγω των πολυάριθμων πλεονεκτημάτων που προσφέρει. Η επιτάχυνση της μετάφρασης διευθύνσεων απαιτεί την χρήση μεγάλων σελίδων για την απεικόνιση της μνήμης, ενώ έχουν επίσης αναπτυχθεί συστήματα (βλέπε [1], [9] και [10]) που κατασκευάζουν larger-than-a-page mappings και παρακινούν την ανάπτυξη hardware που εκμεταλλεύεται τη συνεχόμενη απεικόνιση σελίδων και επιταχύνει τη μετάφραση ακόμα περισσότερο.
Όταν μία εφαρμογή ενός NUMA κόμβου δρομολογείται προς εκτέλεση σε κάποιον από τους κόμβους του συστήματος, είναι πιθανό η τοπική μνήμη τη στιγμή εκείνη να υποφέρει από το φαινόμενο του εξωτερικού κατακερματισμού. Σαν αποτέλεσμα η εικονική μνήμη της εφαρμογής, ερχόμενη στην φυσική μνήμη του τοπικού κόμβου θα σπάσει σε πολλά μικρά κομμάτια, είτε μικρά πλαίσια ή μικρά physical ranges[1]. Από την άλλη, οι μνήμες μακρινών κόμβων πιθανώς να μην πάσχουν από εξωτερικό κατακερματισμό εκείνη τη στιγμή και θα μπορούσαν να φιλοξενήσουν την μνήμη της εφαρμογής με μεγάλες συνεχόμενες απεικονίσεις, καθιστώντας τις κρυφές μνήμες μετάφρασης διευθύνσεων πιο αποδοτικές. Ένα παράδειγμα των δύο εναλλακτικών τοποθετήσεων μνήμης φαίνεται στο σχήμα 1.1.
Σχηματίζεται με άλλα λόγια ένα tradeoff. Από τη μία πλευρά έχουμε 100% τοπική τοποθέτηση μνήμης και αποφυγή διάσχισης του δικτύου διασύνδεσης, αλλά χρήση μικρών πλαισίων ή ranges και αυξημένο κόστος μετάφρασης. Από την άλλη, έχουμε μεγάλες συνεχόμενες απεικονίσεις μνήμης και μειωμένο κόστος μετάφρασης με τίμημα κάποιες ακριβές απομακρυσμένες προσβάσεις κύριας μνήμης.

[1] Ως physical range ορίζεται ένα εύρος διαδοχικών buddy blocks μέγιστου μεγέθους (βλέπε και [1]).



Σχήμα 1.1: Αν προτιμηθεί η μνήμη της διεργασίας να μείνει στον τοπικό κόμβο, θα χρειαστούν 6 κομμάτια φυσικής μνήμης για να απεικονίσουν το vma. Αν επιτραπεί η απεικόνιση και στον μακρινό κόμβο 1, τότε θα χρειαστεί μόνο 1 κομμάτι.

Στην εργασία μελετάμε το tradeoff που μόλις αναφέρθηκε, ειδικότερα σε εικονικά περιβάλλοντα εκτέλεσης, όπου το κόστος μετάφρασης γίνεται πιο σημαντικό. Πιο συγκεκριμένα, στο **κεφάλαιο 2** γίνεται μία πιο λεπτομερής περιγραφή της NUMA αρχιτεκτονικής και του συστήματος εικονικής μνήμης, ενώ επίσης αναφέρεται και σχετική έρευνα. Στο **κεφάλαιο 3** μελετάμε μέσω διαφόρων πειραματικών εκτελέσεων και μετρήσεων το tradeoff contiguity vs NUMA locality και προσπαθούμε να καταλήξουμε σε συμπεράσματα ως προς το τι είναι πιο σημαντικό. Στο **κεφάλαιο 4** παρουσιάζουμε δύο υλοποιήσεις που αφορούν την εκτέλεση εφαρμογών ενός NUMA κόμβου και που δίνουν προτεραιότητα στη χρήση μεγάλων σελίδων και στη συνεχόμενη απεικόνιση σελίδων αντίστοιχα. Η πρώτη υλοποίηση είναι μία απλή βιβλιοθήκη χώρου χρήστη που επιτρέπει στις διεργασίες να απεικονίζουν την μνήμη τους πάντα με μεγάλα πλαίσια (huge frames), αγνοώντας αν χρειαστεί την τοπικότητα των δεδομένων. Η δεύτερη είναι μία επέκταση του πυρήνα CAPaging [1], ώστε τα vmas των διεργασιών να τοποθετούνται και σε remote κόμβους αν αυτό κρίνεται απαραίτητο για την επίτευξη καλού contiguity. Περιλαμβάνεται επίσης η σχεδίαση και υλοποίηση ενός απλού profiler χώρου χρήστη, που παρακολουθεί δυναμικά την εκτέλεση των διεργασιών και επεμβαίνει αν διαπιστώσει ότι το κόστος των remote memory accesses ξεπερνά το όφελος που παράγουν οι μεγάλες σελίδες. Στο **κεφάλαιο 5** δοκιμάζουμε τις υλοποιήσεις μας σε ένα σύνολο εφαρμογών και αξιολογούμε την αποτελεσματικότητα τους. Τέλος, στο **κεφάλαιο 6** παρουσιάζονται σύντομα ιδέες για μελλοντική εργασία πάνω στο θέμα που ασχοληθήκαμε.

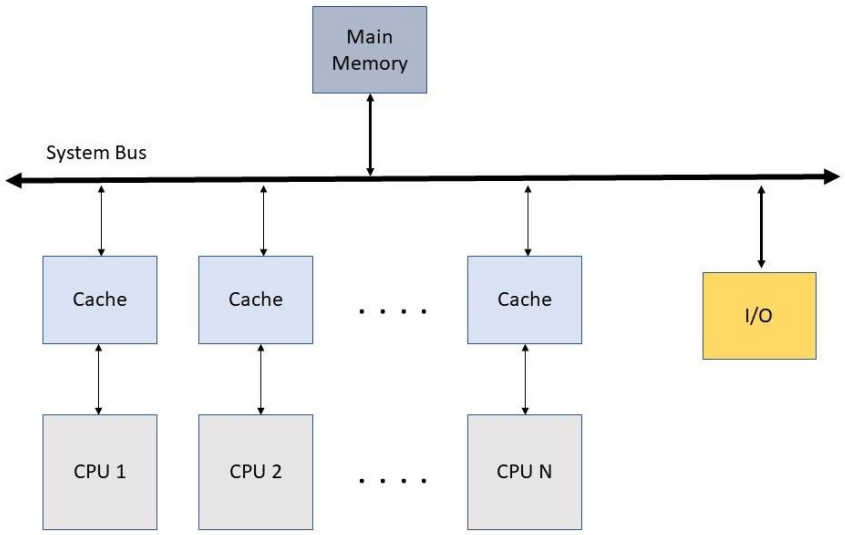
Κεφάλαιο 2 Θεωρητικό Υπόβαθρο

2.1 Συστήματα κοινής και μη κοινής μνήμης

Οι παράλληλες αρχιτεκτονικές μπορούν να χωριστούν σε δύο μεγάλες κατηγορίες. Από την μία υπάρχουν τα συστήματα κοινής μνήμης (shared memory systems) στα οποία όλη η διαθέσιμη μνήμη είναι άμεσα προσπελάσιμη από όλους τους επεξεργαστές μέσω απλών εντολών. Από την άλλη στα συστήματα μη κοινής μνήμης (non-shared memory systems) κάθε επεξεργαστής βλέπει ένα συγκεκριμένο κομμάτι μνήμης το οποίο μπορεί να διαβάσει και να γράψει άμεσα μέσω των εντολών του. Η πρόσβαση στα υπόλοιπα κομμάτια γίνεται με ανταλλαγή μηνυμάτων πάνω από ένα δίκτυο διασύνδεσης και έχει σημαντικά μεγαλύτερο κόστος.

Στα συστήματα κοινής μνήμης μπορεί να γίνει ένας περαιτέρω διαχωρισμός σε δύο βασικές υποκατηγορίες:

- Symmetric Multiprocessor (SMP). Αποτελείται από πλήθος επεξεργαστών που συνδέονται όλοι μέσω ενός κοινού διαύλου δεδομένων στην ίδια μονάδα μνήμης. Έχουν τον ίδιο χρόνο πρόσβασης κύριας μνήμης κάτι που δίνει στην αρχιτεκτονική και την ονομασία UMA (Uniform Memory Access) και ισότιμη πρόσβαση σε όλες τις μονάδες εισόδου-εξόδου. Από την άλλη, κάθε επεξεργαστής διαθέτει την δική του ιδιωτική cache, που έχει ως συνέπεια να δημιουργούνται πολλαπλά αντίγραφα του ίδιου δεδομένου στις κρυφές μνήμες. Αυτό δημιουργεί την ανάγκη για πρωτόκολλα συνοχής κρυφής μνήμης, τα οποία υλοποιούνται σε επίπεδο υλικού, όπως το MESI.



Σχήμα 2.1: Symmetric Multiprocessor System

- Non-Uniform-Memory-Access (NUMA) μηχανήματα. Οι επεξεργαστές και η μνήμη του συστήματος οργανώνονται σε κόμβους που συνδέονται μεταξύ τους μέσω ενός δικτύου διασύνδεσης. Κάθε επεξεργαστής μπορεί να διαβάσει και να γράψει όλη τη μνήμη, αλλά οι προσβάσεις στην μνήμη του κόμβου του έχουν μικρότερη καθυστέρηση. Και εδώ είναι απαραίτητα τα πρωτόκολλα συνάφειας κρυφής μνήμης, τα μηνύματα του οποίου χρειάζεται να διασχίζουν και το δίκτυο διασύνδεσης. Λόγω της αποκεντροποιημένης οργάνωσης τους, κλιμακώνουν καλύτερα σε σχέση με τις SMP αρχιτεκτονικές και μπορούν να περιλαμβάνουν μεγαλύτερο πλήθος

επεξεργαστών χωρίς να εμφανίζονται φαινόμενα συμφόρησης. Μιας και στην εργασία αυτή ασχολούμαστε αποκλειστικά με αυτού του τύπου συστήματα, περιγράφονται με περισσότερες λεπτομέρειες στο υποκεφάλαιο 1.2.

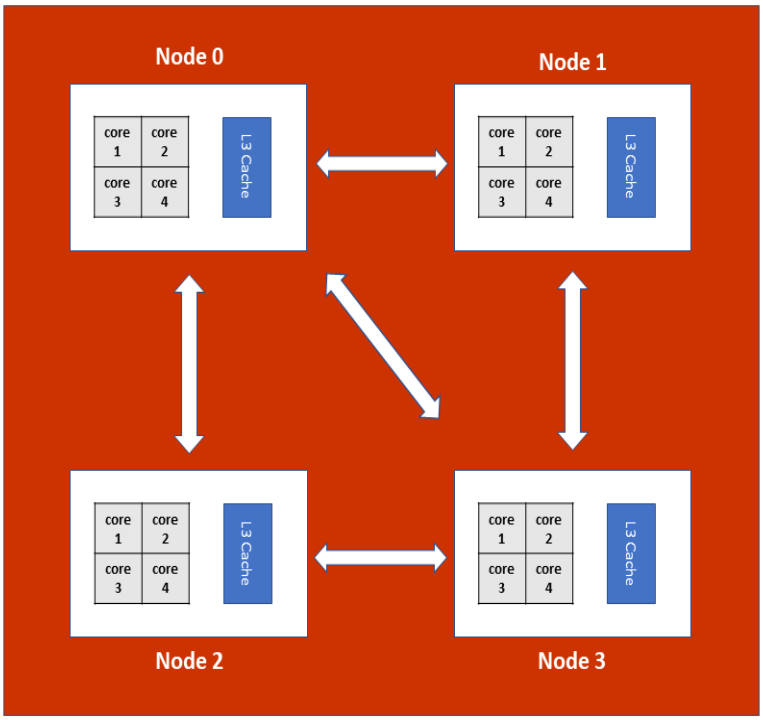
Ένας σημαντικός περιορισμός των συστημάτων κοινής μνήμης αφορά την κλιμακωσιμότητα. Λόγω της ανάγκης συγχρονισμού μεταξύ των κρυφών μνημών των επεξεργαστών, όσο αυξάνει των πλήθος τους αυξάνει και το πλήθος των μηνυμάτων του πρωτοκόλλου συνάφειας. Από κάποιο σημείο και έπειτα, η ενσωμάτωση περισσότερων επεξεργαστών καθίσταται μη πρακτική.

Τα συστήματα μη κοινής μνήμης από την άλλη, απαλλαγμένα από την απαίτηση συγχρονισμού, μπορούν να συμπεριλάβουν πολύ μεγαλύτερο αριθμό επεξεργαστών, δημιουργώντας τεράστια υπολογιστική ισχύ. Κάποιες βασικές κατηγορίες είναι οι εξής:

- Computer Cluster - Είναι μία ομάδα πολλών αυτόνομων υπολογιστών οι οποίοι συνδέονται μεταξύ τους μέσω δικτύου και συνεργάζονται για να λύσουν ένα κοινό πρόβλημα. Οι υπολογιστές βρίσκονται στον ίδιο φυσικό χώρο και για τη σύνδεση τους χρησιμοποιούνται πρότυπα συστήματα διασύνδεσης όπως InfinyBand και Gigabit Ethernet.
- Massively Parallel Processors (MPP) - Έχουν παρόμοια χαρακτηριστικά με ένα Computer Cluster. Ωστόσο, για την επικοινωνία μεταξύ των επεξεργαστών χρησιμοποιούνται εξειδικευμένα πρωτόκολλα διασύνδεσης. Επίσης, διαθέτουν μεγαλύτερη συνολική υπολογιστική ισχύ μιας και ενσωματώνουν μεγαλύτερο αριθμό επεξεργαστικών στοιχείων. Κάθε αυτόνομη μονάδα έχει το δικό της ξεχωριστό αντίγραφο λειτουργικού συστήματος και εκτελούμενης εφαρμογής.
- Grid Computing – Ανεξάρτητοι ο ένας από τον άλλο υπολογιστές που δεν βρίσκονται στον ίδιο φυσικό χώρο αλλά συνδέονται μεταξύ τους μέσω διαδικτύου, συνεργάζονται για να λύσουν ένα κοινό πρόβλημα. Συνήθως, καταναλώνουν κύκλους επεξεργασίας, όταν οι υπολογιστικές μονάδες-συνιστώσες είναι κατά τα άλλα αδρανείς.

2.2 Η Αρχιτεκτονική NUMA

Στις NUMA αρχιτεκτονικές η μνήμη και οι επεξεργαστές του συστήματος οργανώνονται σε υπολογιστικούς **κόμβους** (Numa Nodes). Κάθε κόμβος μπορεί να θεωρηθεί ισοδύναμος ενός συμμετρικού πολυεπεξεργαστή (Symmetric Multiprocessor) και συνδέεται με τους υπόλοιπους μέσω ενός δικτύου διασύνδεσης (Interconnect Network), σχηματίζοντας ένα ενιαίο υπολογιστικό σύστημα **κοινής μνήμης**. Χαρακτηριστικό γνώρισμα της αρχιτεκτονικής, απ' όπου παίρνει και την ονομασία της (Non Uniform Memory Access), είναι ότι ο χρόνος πρόσβασης στην κύρια μνήμη δεν είναι σταθερός αλλά εξαρτάται από την απόσταση του αιτούμενου επεξεργαστή και του κόμβου όπου αποθηκεύονται τα ζητούμενα δεδομένα. Αν, για παράδειγμα, τα δεδομένα που ζητά ο επεξεργαστής βρίσκονται στην μνήμη του ίδιου κόμβου (local access), τότε θα είναι διαθέσιμα νωρίτερα σε σχέση με το αν βρίσκονταν στην μνήμη ενός απομακρυσμένου κόμβου (remote access). Με την πάροδο του χρόνου και λόγω της βελτίωσης των δικτύων διασύνδεσης, η διαφορά στο κόστος μεταξύ local και remote memory access έχει μειωθεί σημαντικά, ωστόσο παραμένει μη αμελητέα.



Σχήμα 2.2: Παράδειγμα ενός NUMA μηχανήματος με 4 κόμβους. Κάθε κόμβος διαθέτει 4 πυρήνες και μία κοινή L3 cache μνήμη. Οι κόμβοι συνδέονται μεταξύ τους μέσω ενός δικτύου διασύνδεσης

Το κυρίως κίνητρο για την αρχιτεκτονική NUMA είναι η αποφυγή του συνωστισμού που παρατηρείται στις SMP αρχιτεκτονικές, όπου όλοι οι επεξεργαστές του υπολογιστή συνδέονται στον ίδιο δίαυλο μνήμης και ανταγωνίζονται για το εύρος ζώνης του. Σε ένα σύστημα NUMA, λόγω της οργάνωσης σε κόμβους και κατ' επέκταση της αποκεντροποίησης, μπορεί να προστίθεται μεγαλύτερο πλήθος επεξεργαστών χωρίς να δημιουργούνται τέτοια φαινόμενα συνωστισμού.

Αποστάσεις κόμβων

Οι αποστάσεις μεταξύ των κόμβων του συστήματος μπορούν να περιγραφούν μέσω ενός δισδιάστατου πίνακα. Μία προκαθορισμένη τιμή ορίζεται ώστε να υποδεικνύει το κόστος πρόσβασης στον τοπικό κόμβο. Την τιμή αυτή παίρνουν όλα τα στοιχεία της διαγωνίου. Οι υπόλοιπες τιμές του πίνακα υποδεικνύουν το κόστος επικοινωνίας μεταξύ κάθε ζεύγους κόμβων και εξαρτώνται από από το πλήθος και το είδος των διαύλων (links) που τους ενώνουν. Ο πίνακας αποστάσεων δεν είναι απαραίτητο να είναι συμμετρικός. Παρακάτω δίνεται ένα παράδειγμα.

Node	*0*	*1*	*2*	*3*
0	10	15	20	20
1	15	10	20	20
2	20	20	10	15
3	20	20	15	10

Η τιμή 10 υποδεικνύει τοπική πρόσβαση, η οποία κοστίζει το ίδιο ανεξαρτήτως κόμβου. Παρατηρούμε ότι η προσπέλαση της μνήμης κόμβου 0 από τον κόμβο 1 (στοιχείο [1,0]) είναι 1.5 φορές πιο αργή σε σχέση με την τοπική πρόσβαση, ενώ η προσπέλαση του ίδιου κόμβου από τους 2 και 3 είναι δύο φορές πιο αργή.

Ιδιαιτερότητες NUMA αρχιτεκτονικών

Υπάρχουν κάποιες συνθήκες που απορρέουν από τα χαρακτηριστικά της αρχιτεκτονικής και πρέπει να ικανοποιούνται από τα προγράμματα προκειμένου να επιτυγχάνεται καλή επίδοση.

Η πιο σημαντική προϋπόθεση είναι τα δεδομένα ενός προγράμματος να τοποθετούνται στην μνήμη του κόμβου όπου εκτελούνται και τα νήματα του (pages-threads colocation). Έτσι, οι προσβάσεις μνήμης που αστοχούν στην ιεραρχία caches θα πληρώνουν το μικρότερο δυνατό κόστος, αφού θα εξυπηρετούνται από την τοπική μονάδα μνήμης και δεν θα χρειάζεται να διασχίσουν το δίκτυο διασύνδεσης. Τονίζεται βέβαια ότι στην περίπτωση που τα δεδομένα που χρησιμοποιεί ένα πρόγραμμα διατηρούνται σε ικανοποιητικό βαθμό στην κρυφή μνήμη του επεξεργαστή, τότε η ιδιότητα αυτή παύει να είναι τόσο σημαντική.

Από την άλλη, στην περίπτωση εφαρμογών ευαίσθητων στο main memory bandwidth, είναι σημαντικό να μοιραστούν τα δεδομένα σε περισσότερους του ενός κόμβους ούτως ώστε να μεγιστοποιείται το διαθέσιμο εύρος ζώνης. Διαφορετικά, αν όλη η μνήμη συγκεντρωθεί σε ένα κόμβο, είναι πιθανό να προκύψουν επιπλέον κόστη λόγω συγχρωτισμού και να καθυστερήσει κατά πολύ περισσότερο η ολοκλήρωση του προγράμματος. Η τελευταία προϋπόθεση έχει μεγαλύτερη σημασία σε εφαρμογές περισσότερων του ενός κόμβου (όταν τα νήματα είναι περισσότερα από τους πυρήνες ενός node). Υπάρχουν ωστόσο και περιπτώσεις εφαρμογών ενός κόμβου, όπου λόγω συμφόρησης της τοπικής μνήμης είναι προτιμότερο ένα ποσοστό των δεδομένων να σταλεί σε μακρινό κόμβο για αύξηση του συνολικού διαθέσιμου bandwidth και αποφυγή καθυστερήσεων στην ουρά του τοπικού memory controller.

Η σημασία των δύο επιθυμητών ιδιοτήτων που αναφέρθηκαν παραπάνω, της τοπικότητας μνήμης και της αύξησης του bandwidth , ποικίλει ανάλογα με την χρονική περίοδο. Κατά την δεκαετία του 1990, μία απομακρυσμένη πρόσβαση ήταν από 4 έως 10 φορές πιο αργή σε σχέση με μία τοπική. Συνεπώς, η τοπικότητα αποτελούσε τον κυριότερο παράγοντα επίδοσης και πρωταρχικό στόχο. Με την βελτίωση των δικτύων διασύνδεσης πλέον αυτό έχει αλλάξει, με τα νεότερα συστήματα να επιβαρύνουν τις απομακρυσμένες προσβάσεις σε ποσοστό μέχρι 30%. Ως αποτέλεσμα, η κατανομή των δεδομένων και η αποφυγή συμφόρησης στο δίκτυο και τους ελεγκτές μνήμης αποτελεί υπολογίσιμο παράγοντα.

NUMA πολιτικές και Συστήματα αναπροσαρμογής τοποθέτησης μνήμης και νημάτων

Προκειμένου να επιτευχθούν σε ικανοποιητικό βαθμό οι στόχοι τοπικών προσβάσεων και αύξησης bandwidth, έχουν αναπτυχθεί διάφορα συστήματα λογισμικού.

Από τη μία, υπάρχουν εκείνα που ενσωματώνονται στο λειτουργικό σύστημα και μέσω αλγόριθμων και μετρικών αποφασίζουν τα ίδια για το πώς θα τοποθετηθεί η μνήμη της εφαρμογής στους κόμβους. Είναι πιο σύνθετα στην ανάπτυξη τους, καθώς επιχειρούν να συμπεριλάβουν στην λογική τους όλα τα διαφορετικά είδη εφαρμογών που θα κληθεί να εκτελέσει το υποκείμενο μηχάνημα. Λαμβάνοντας αυτοματοποιημένες αποφάσεις, εξοικονομούν προγραμματιστικό χρόνο, ενώ παράλληλα καθιστούν το υπολογιστικό σύστημα περισσότερο αυτάρκες.

Από την άλλη, υπάρχουν βιβλιοθήκες που παρέχουν τον απαραίτητο μηχανισμό και μέσω προγραμματιστικών διεπαφών αφήνουν τον ίδιο τον προγραμματιστή να αποφασίσει σε ποιο κόμβο μνήμης θα αντιστοιχιστούν οι σελίδες δεδομένων του προγράμματος του. Γνωρίζοντας τα ιδιαίτερα χαρακτηριστικά της εφαρμογής και εστιάζοντας σε αυτά, ο προγραμματιστής μπορεί να πετύχει την βέλτιστη τοποθέτηση. Για παράδειγμα, στο Linux η σχετική βιβλιοθήκη ονομάζεται libnuma [2], και προσφέρει πλήθος χρήσιμων συναρτήσεων στους προγραμματιστές.

Η υιοθέτηση αυτοματοποιημένων τεχνικών εντός των βασικών εκδόσεων των λειτουργικών συστημάτων βρίσκεται ακόμα σε αρχικό στάδιο. Τα περισσότερα παρέχουν κατ΄ελάχιστον δύο βασικές πολιτικές.

Η πρώτη ονομάζεται **First-Touch (**ή **Local)** και σύμφωνα με αυτήν, μία σελίδα αντιστοιχίζεται στον κόμβο απ΄όπου προέρχεται η πρώτη αναφορά (εξ' ου και η ονομασία first-touch). Είναι απλή πολιτική και κατά κανόνα ιδανική για εφαρμογές ενός κόμβου. Ωστόσο, δεν μπορεί να εγγυηθεί υψηλό ποσοστό τοπικών προσβάσεων για εφαρμογές περισσότερων κόμβων. Είναι χαρακτηριστικό το παράδειγμα όπου ένα νήμα της εφαρμογής αρχικοποιεί δομές δεδομένων οι οποίες όμως στην συνέχεια χρησιμοποιούνται από νήματα που εκτελούνται σε άλλο κόμβο.

Η δεύτερη βασική πολιτική ονομάζεται **Interleave** και μοιράζει με λογική round-robin τις σελίδες δεδομένων της εφαρμογής σε ένα προεπιλεγμένο σύνολο κόμβων. Δεν λαμβάνεται καθόλου υπόψιν η τοπικότητα των δεδομένων και σκοπός της είναι η αποφυγή συμφόρησης και η αξιοποίηση όλου του διαθέσιμου bandwidth. Θα μπορούσε να είναι μία καλή πολιτική για εφαρμογές που επεκτείνονται σε πολλές κόμβους και που η πλειοψηφία των δεδομένων συγκεντρώνει εξίσου αναφορές από όλα τα εμπλεκόμενα nodes.

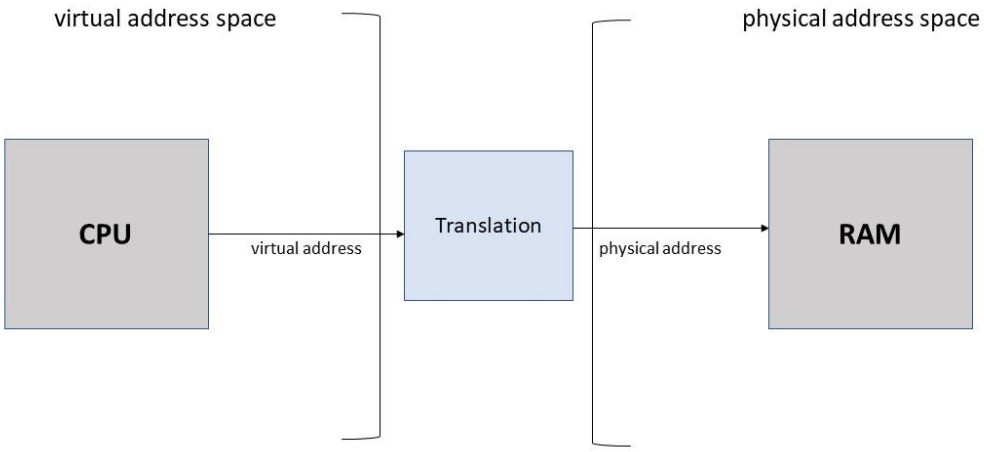
Στον χρήστη του συστήματος δίνονται εργαλεία γραμμής εντολών με τα οποία μπορεί να επιλέξει με ποια πολιτική θα εκτελεστεί η εφαρμογή του, με την First-Touch συνήθως να αποτελεί την προεπιλογή. Στο Linux το γνωστότερο τέτοιο εργαλείο είναι το numactl [3]. Επισημαίνεται ότι οι δύο πολιτικές που μόλις αναφέρθηκαν, δεν αναπροσαρμόζουν δυναμικά τις αποφάσεις τους ανάλογα με το μοτίβο πρόσβασης της εφαρμογής στην κύρια μνήμη και υπάρχουν πολλές περιπτώσεις στις οποίες η τοποθέτηση που πετυχαίνουν απέχει πολύ από τη βέλτιστη.

Υπάρχουν και κάποιες πιο εξελιγμένες και σύνθετες προσεγγίσεις. Στον πυρήνα του Linux για παράδειγμα, έχει προστεθεί η πολιτική AutoNUMA [4], που στηρίζεται σε δύο κύρια σημεία. Κατά το πρώτο, ο πυρήνας διατηρεί για κάθε διεργασία ένα πίνακα μετρητών, με έναν μετρητή ανά κόμβο. Κάθε φορά που η διεργασία πραγματοποιεί μία πρόσβαση στην κύρια μνήμη, αυξάνεται ο μετρητής του κόμβου όπου βρίσκεται το δεδομένο. Με χρήση των μετρητών, ο πυρήνας ανακαλύπτει ποιος κόμβος περιέχει την πλειοψηφία της μνήμης της διεργασίας και έχει την δυνατότητα να μεταφέρει τα νήματα της εκεί. Κατά το δεύτερο σημείο, ο πυρήνας προσπαθεί, πάλι μέσω συλλογής

στατιστικών, να εντοπίσει διεργασίες που μοιράζονται κοινά δεδομένα (για παράδειγμα μία βιβλιοθήκη κώδικα ή τα δεδομένα ενός αρχείου) και να τις ομαδοποιήσει σε έναν cluster. Έπειτα, προσπαθεί να μεταφέρει όλες τις διεργασίες του κάθε cluster στον κόμβο που φιλοξενεί την πλειοψηφία της κοινής μνήμης.

2.3 Εικονική μνήμη

Σύμφωνα με τα συστήματα εικονικής μνήμης, οι διευθύνσεις δεδομένων ενός προγράμματος εντολών είναι **εικονικές**, που σημαίνει ότι δεν χρησιμοποιούνται απευθείας για δεικτοδότηση. Πρέπει πρώτα να περάσουν από ένα στάδιο μετάφρασης, ώστε να προκύψουν οι αντίστοιχες **φυσικές** διευθύνσεις που δείχνουν την θέση φυσικής μνήμης όπου περιέχονται τα δεδομένα. Έτσι λοιπόν προκύπτουν δύο πεδία διευθύνσεων, το εικονικό (virtual address space) και το φυσικό (physical address space), όπως φαίνεται και στο παρακάτω σχήμα.

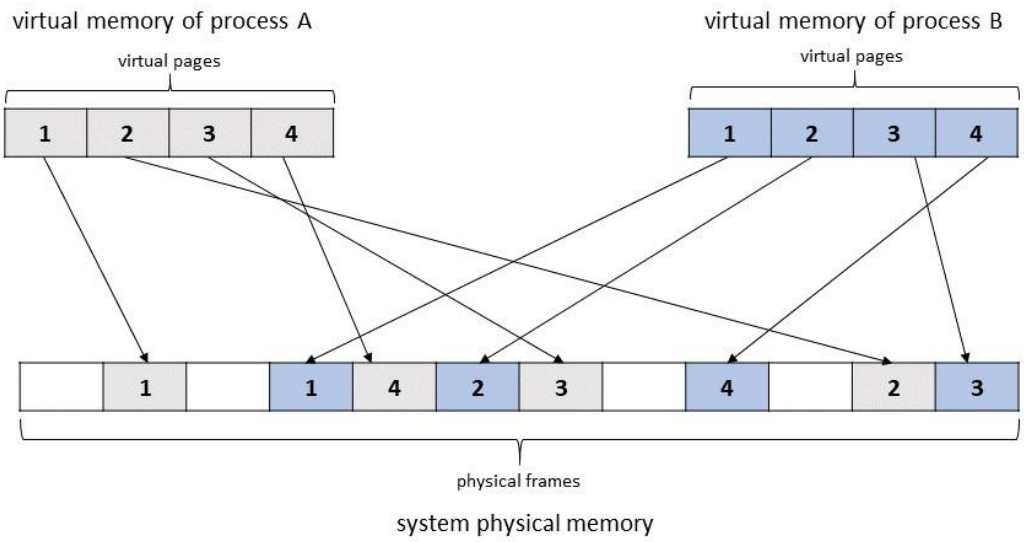


Σχήμα 2.3: Εικονικό και φυσικό πεδίο διευθύνσεων

Το μέγεθος του εικονικού χώρου εξαρτάται από τον επεξεργαστή και πιο συγκεκριμένα τον αριθμό των bits που χρησιμοποιεί για να κωδικοποιήσει μία διεύθυνση μνήμης. Για παράδειγμα, εικονικές διευθύνσεις των 32 bit αντιστοιχούν σε εικονικό χώρο $2^{32} = 4GB$. Από την άλλη το μέγεθος τους φυσικού χώρου ταυτίζεται με το μέγεθος της κύριας μνήμης του συστήματος.

Τα πεδία διευθύνσεων σπάνε σε μπλοκ σταθερού μεγέθους. Στο εικονικό πεδίο τα μπλοκ ονομάζονται **σελίδες** (pages) και στο φυσικό **πλαίσια** (frames). Η μετάφραση γίνεται σε επίπεδο σελίδας, δηλαδή μία ολόκληρη σελίδας μνήμης μίας διεργασίας αντιστοιχίζεται (απεικονίζεται) σε ένα πλαίσιο. Παράδειγμα δίνεται στο σχήμα 2.4.

Η ουσία της εικονικής μνήμης βρίσκεται στην διαφοροποίηση μεταξύ εικονικής σελίδας και φυσικού πλαισίου. Με όρο σελίδα αναφερόμαστε στα δεδομένα της διεργασίας που περιέχονται σε αυτή, ανεξάρτητα από το φυσικό μέσο στο οποίο αποθηκεύονται (κύρια μνήμη, δίσκος κτλ). Με τον όρο πλαίσιο αναφερόμαστε σε ένα συνεχόμενο κομμάτι (chunk) κύριας μνήμης το οποίο θα αποθηκεύσει με την πάροδο του χρόνου διαφορετικές σελίδες δεδομένων.



Σχήμα 2.4: Μετάφραση σελίδων εικονικής μνήμης σε πλαίσια φυσικής μνήμης

Αν και το κόστος μετάφρασης για κάθε εντολή αναφοράς στην μνήμη εισάγει ένα μη αμελητέο overhead στην εκτέλεση των προγραμμάτων, υπάρχουν πολλοί λόγοι για την χρήση της εικονικής μνήμης.

- Λόγω του επιπέδου αφαίρεσης που εισάγεται μεταξύ προγραμμάτων και φυσικών πόρων μνήμης, η κάθε διεργασία έχει το δικό της απομονωμένο πεδίο διευθύνσεων, ξεχωριστό από αυτό κάθε άλλης συνεκτελούμενης διεργασίας. Με άλλα λόγια, ένα πρόγραμμα έχει την "ψευδαίσθηση" ότι εκτελείται μόνο του στο σύστημα και πως όλη η διαθέσιμη μνήμη του ανήκει.
- Όταν ο εικονικός χώρος διευθύνσεων έχει μεγαλύτερο μέγεθος από τον φυσικό, το πρόγραμμα έχει στη διάθεση του περισσότερη ποσότητα μνήμης από όση παρέχει η κύρια μνήμη συστήματος. Στις περιπτώσεις αυτές και αν το πρόγραμμα χρειάζεται πράγματι αυτή την επιπλέον ποσότητα, επιστρατεύεται η μονάδα δευτερεύουσας αποθήκευσης, ώστε να υποστηρίξει όσα δεδομένα δεν χωράνε στην κύρια μνήμη.
- Μπορούν να τεθούν ξεχωριστοί κανόνες ελέγχου πρόσβασης για κάθε σελίδα μνήμης του προγράμματος. Με άλλα λόγια, κάθε σελίδα φέρει και τους δικούς της περιορισμούς ως προς την πρόσβαση στα δεδομένα που περιέχει. Οι κανόνες μπορούν να τεθούν από την ίδια τη διεργασία ή και από το λειτουργικό σύστημα.
- Με την κατάλληλη μέριμνα, η εικονική μνήμη μπορεί να δράσει ως ένα μηχανισμός δια-διεργασιακής επικοινωνίας (interprocess communication). Πιο συγκεκριμένα, αν δύο οι περισσότερες διεργασίες αντιστοιχίσουν κάποιες από τις σελίδες τους στα ίδια πλαίσια κύριας μνήμης, τότε έχουν ένα τρόπο να μοιράζονται δεδομένα και να επικοινωνούν μεταξύ τους.
- Μέσω κατάλληλης κλήσης συστήματος, ένα πρόγραμμα μπορεί να απεικονίσει κομμάτι της εικονικής μνήμης του στα δεδομένα ενός αρχείου. Αυτό συνεπάγεται ότι με τις καθιερωμένες εντολές πρόσβασης κύριας μνήμης, το πρόγραμμα μπορεί να διαβάζει και να γράφει τα δεδομένα του αρχείου.
- Δημοφιλή αρχεία δεδομένων και βιβλιοθήκες κώδικα αποθηκεύονται στην κύρια μνήμη με ένα μοναδικό αντίγραφο και αυτό μοιράζεται μεταξύ των διεργασιών που θέλουν να τα χρησιμοποιήσουν μέσω κατάλληλων απεικονίσεων. Έτσι, επιτυγχάνεται μεγάλη εξοικονόμηση πόρων κύριας μνήμης.

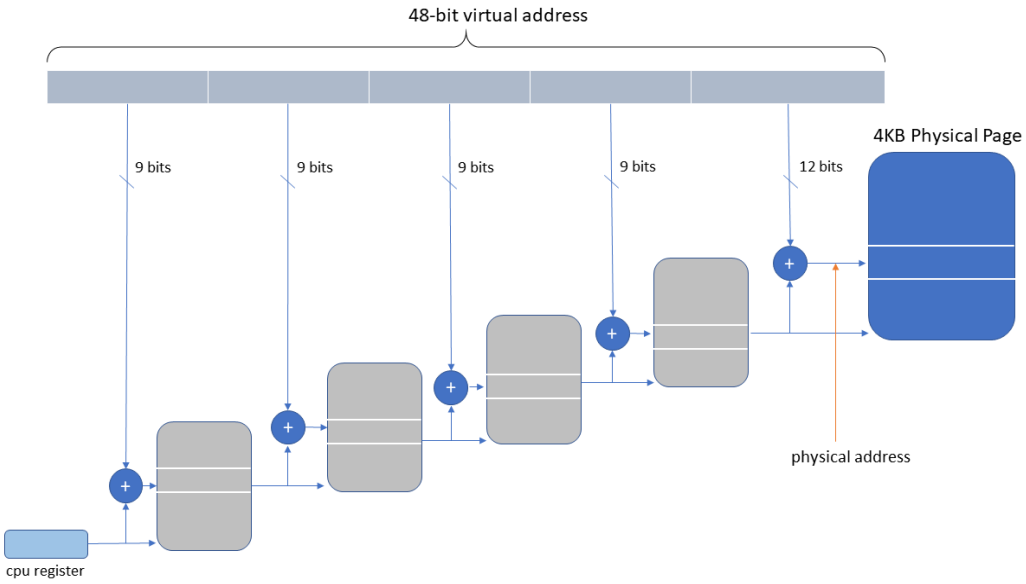
- Σημαντική εξοικονόμηση επιτυγχάνεται και στις περιπτώσεις όπου εφαρμογές αιτούνται ποσότητες μνήμης αλλά τελικά δεν τις χρησιμοποιούν. Χάρη στην εικονική μνήμη και την τεχνική του demand paging, για αυτά τα αιτήματα μνήμης δεν θα δεσμευθούν ποτέ φυσικά πλαίσια.

Πίνακες σελίδων (page tables)

Είναι απαραίτητη μία δομή δεδομένων ανά διεργασία που θα αποθηκεύει τις μεταφράσεις των σελίδων στα πλαίσια. Η δομή θα δέχεται ως είσοδο τον αριθμό εικονικής σελίδας (virtual page number) και να δίνει στην έξοδο τον αριθμό πλαισίου (page frame number ή pfn).

Η πιο απλή λύση είναι ένας γραμμικός πίνακας, τόσων στοιχείων όσο και το πλήθος των εικονικών σελίδων. Το στοιχείο στην θέση a θα περιέχει την μετάφραση (αριθμό πλαισίου) της σελίδας με αριθμό a. Υποθέτοντας εικονικό χώρο των 32 bit και μέγεθος σελίδας 4KB, προκύπτει πλήθος εικονικών σελίδων ίσο με $2^{32-12} = 2^{20}$ (αφού χρειάζονται 12 bits για την μετατόπιση εντός σελίδας). Αν θεωρήσουμε ότι κάθε μετάφραση χρειάζεται 32 bits (20 για τον αριθμό πλαισίου συν κάποια bit ελέγχου πρόσβασης), τότε δεσμεύονται $2^{20} * 4\ bytes = 4MB$ φυσικής μνήμης για κάθε διεργασία. Ακόμα χειρότερα για εικονικούς χώρους των 48 bit (πχ x86_64 αρχιτεκτονική), θα χρειάζονταν $2^{36} * 4 = 256GB$ ανά διεργασία, που είναι φυσικά μη ρεαλιστικό.

Για το λόγο αυτό χρησιμοποιούνται τα page tables (πίνακες σελίδων), μία δομή δέντρου το βασικό πλεονέκτημα της οποίας είναι ότι δεσμεύεται φυσική μνήμη μόνο για μεταφράσεις σελίδων τις οποίες χρησιμοποιεί η διεργασία. Η μετάφραση γίνεται διασχίζοντας τα page tables από την ρίζα μέχρι κάποιο φύλλο, χρησιμοποιώντας τμήματα του υπό μετάφραση αριθμού σελίδας για την επιλογή των ενδιάμεσων κόμβων (σχήμα 2.5).

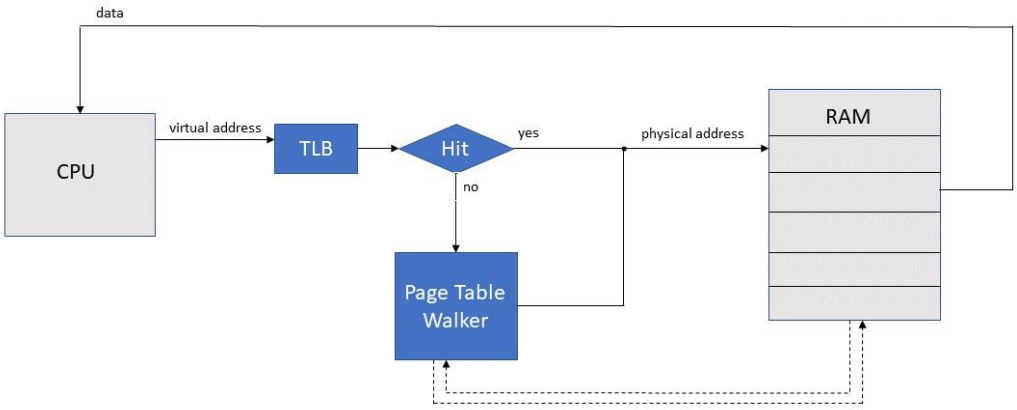


Σχήμα 2.5: Παράδειγμα διάσχισης των πινάκων σελίδων για εικονικές διευθύνσεις 48 bit και μέγεθος σελίδας 4KB. Ένας καταχωρητής του επεξεργαστή αποθηκεύει την φυσική διεύθυνση του κόμβου-ρίζας των page tables της τρέχουσας διεργασίας.

Τα page tables εξοικονομούν μεν σημαντική ποσότητα μνήμης, αυξάνουν όμως το χρόνο μετάφρασης μιας και η διάσχιση τους χρειάζεται αρκετές αναφορές στην μνήμη. Ο ακριβής αριθμός αναφορών δεν είναι σταθερός από σύστημα και σύστημα και εξαρτάται από το πλήθος bit των εικονικών διευθύνσεων και το μέγεθος σελίδας. Για παράδειγμα, η αρχιτεκτονική x86_64 χρησιμοποιεί εικονικές διευθύνσεις 48 bit και υποστηρίζει μεγέθη σελίδας 4KB, 2MB και σε κάποιες περιπτώσεις 1GB. Για το πρώτο μέγεθος (4 KB) η μετάφραση χρειάζεται τέσσερις αναφορές μνήμης (όπως φαίνεται στο σχήμα 2.5), για το δεύτερο (2 MB) τρεις και για το τρίτο (1 GB) δύο. Επίσης, είναι σημαντικό να αναφερθεί ότι η διάσχιση των page tables χρησιμοποιεί τις caches του επεξεργαστή και έτσι μπορεί να διώξει (evict) χρήσιμα δεδομένα του προγράμματος (cache pollution).
Επισημαίνεται ότι η μετάφραση διευθύνσεων γίνεται από την μονάδα διαχείρισης μνήμης (Memory Management Unit – MMU) του επεξεργαστή που διασχίζει σε επίπεδο hardware τα page tables. Ο σχηματισμός των μεταφράσεων (δηλαδή η αντιστοίχιση σελίδων με πλαίσια) γίνεται από το λειτουργικό σύστημα κατά την πρώτη αναφορά της διεργασίας στην σελίδα, εξασφαλίζοντας ότι θα δεσμευθούν πλαίσια μόνο για τις σελίδες που χρησιμοποιούνται (demand paging). Τέλος τονίζεται ότι αν και τα μεγαλύτερα μεγέθη σελίδας μειώνουν, όπως είδαμε, το κόστος μετάφρασης, αυξάνουν την πιθανότητα εσωτερικού κατακερματισμού (internal fragmentation).

Κρυφή μνήμη μεταφράσεων (TLB)

Προκειμένου να επιταχυνθεί η μετάφραση διευθύνσεων, συμπεριλαμβάνεται στην MMU του επεξεργαστή μία ιεραρχία κρυφών μνημών που αποθηκεύει μεταφράσεις της εκτελούμενης διεργασίας και στην ουσία λειτουργεί σαν cache των page tables που βρίσκονται στην κύρια μνήμη. Η cache αυτή ονομάζεται Translation Lookaside Buffer (TLB) και όταν ο επεξεργαστής εκτελεί μία εντολή μνήμης αναζητείται η μετάφραση διεύθυνσης πρώτα σε αυτή, προτού ξεκινήσει η χρονοβόρα διάσχιση των page tables. Τα παραπάνω φαίνονται και στο επόμενο σχήμα.



Σχήμα 2.6: Σε περίπτωση ευστοχίας της κρυφής μνήμης TLB, η φυσική διεύθυνση του δεδομένου είναι άμεσα διαθέσιμη. Διαφορετικά, η μετάφραση αναζητείται μέσω της διάσχισης των page tables.

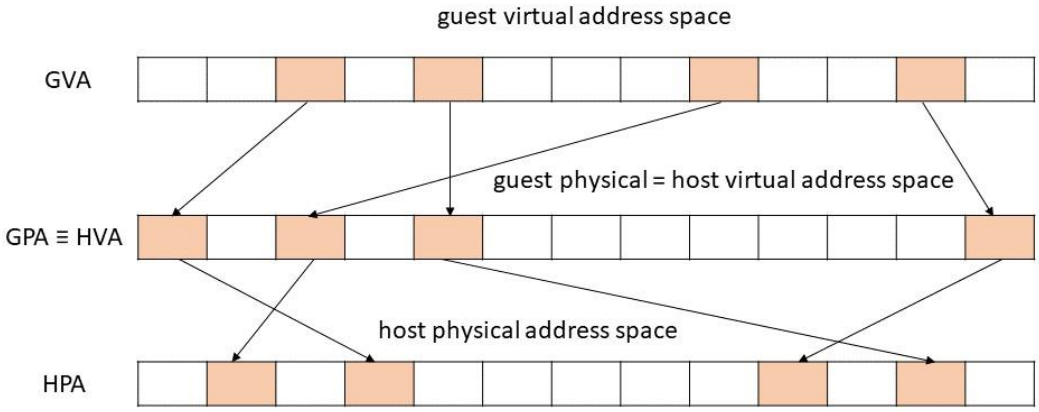
Η αποθήκευση μεταφράσεων σε κρυφές μνήμες συνήθως οργανώνεται σε δύο επίπεδα. Στο πρώτο υπάρχουν δύο μικρά αλλά πολύ γρήγορα L1 TLBs, ένα για σελίδες εντολών (ITLB) και ένα για σελίδες δεδομένων (DTLB) και στο δεύτερο ένα μεγαλύτερο και αργότερο L2 TLB.
Επίσης, υπάρχουν δύο τρόποι για τον χειρισμό των διαφορετικών μεγεθών σελίδων που υποστηρίζονται από την κάθε αρχιτεκτονική. Σύμφωνα με τον πρώτο, το TLB χωρίζεται σε ανεξάρτητα κομμάτια, ένα για κάθε μέγεθος, τα οποία αναζητούνται παράλληλα για την εύρεση μετάφρασης. Σύμφωνα με τον δεύτερο τρόπο, υπάρχει μία μονάδα TLB για όλα τα μεγέθη και οι εγγραφές έχουν σταθερό μήκος, με ένα επιπλέον πεδίο να υποδεικνύει το μέγεθος σελίδας για κάθε αποθηκευμένη μετάφραση.

Μετάφραση διευθύνσεων σε εικονικό μηχάνημα

Σε ένα μηχανικό μηχάνημα έχουμε περισσότερους από δύο χώρους διευθύνσεων:

- guest εικονικός χώρος: σε αυτόν ανήκουν οι εικονικές διευθύνσεις των διεργασιών που εκτελούνται μέσα στο εικονικό μηχάνημα (GVA)
- guest φυσικός χώρος ≡ host εικονικός χώρος: η μνήμη που το guest OS αντιλαμβάνεται ως φυσική αλλά στην πραγματικότητα είναι εικονική μνήμη στο host μηχάνημα (GPA ≡ HVA)
- host φυσικός χώρος: η φυσική μνήμη (RAM) συστήματος (HPA)

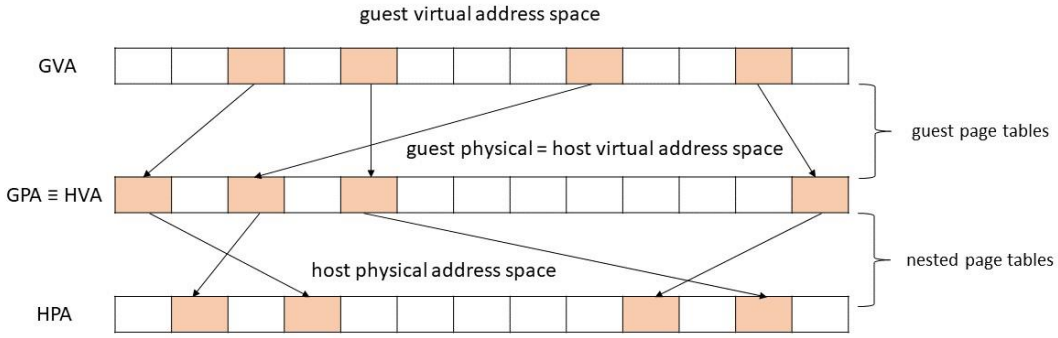
Ο διαχωρισμός των διάφορων πεδίων διευθύνσεων φαίνεται και στο παρακάτω σχήμα.



Σχήμα 2.7: Το guest ΛΣ απεικονίζει από τον guest εικονικό στον guest φυσικό χώρο. Το host ΛΣ απεικονίζει την φυσική μνήμη του εικονικού μηχανήματος στην πραγματική μνήμη.

Υπάρχουν δύο βασικοί τρόποι για την διαχείριση της εικονικής μνήμης μίας διεργασίας που τρέχει μέσα σε ένα virtual machine, ένας που βασίζεται σε λογισμικό και ένας που βασίζεται σε επεκτάσεις υλικού. Σύμφωνα με τον πρώτο, οι μεταφράσεις της διεργασίας περιέχονται σε page tables που απεικονίζουν απευθείας από τον guest εικονικό στον host φυσικό χώρο διευθύνσεων (GVA → HPA) και διασχίζονται κατά τα γνωστά από τον page table walker του υπάρχοντος υλικού. Τα pages tables αυτά διαχειρίζεται ο Virtual Machine Hypervisor (VMM). Όμως είναι απαραίτητη μία ακόμα δομή πινάκων σελίδων που να αποθηκεύει μεταφράσεις από guest εικονικές σε guest φυσικές διευθύνσεις (GVA → GPA), την οποία θα διαχειρίζεται το guest ΛΣ

ώστε να έχει την εντύπωση ότι εκτελείται σε ένα πραγματικό μηχάνημα. Η τεχνική αυτή ονομάζεται Shadow Page Tables και απαιτεί για τον συγχρονισμό των δύο δομών page tables να δίνεται με μεγάλη συχνότητα ο έλεγχος του μηχανήματος στον Hypervisor με αποτέλεσμα να εισάγονται μεγάλες καθυστερήσεις.
Γι' αυτό έχουν ενσωματωθεί επεκτάσεις υλικού που αποφεύγουν αυτές τις χρονοβόρες παρεμβάσεις και χειρίζονται την εικονική μνήμη αποκλειστικά σε επίπεδο hardware. Ο extended page table walker διασχίζει τα **guest page tables**, όπου για κάθε guest φυσική διεύθυνση που προκύπτει στην πορεία, γίνεται αυτόματα μία φωλιασμένη διάσχιση των **nested page tables**. Τα nested page tables είναι στον έλεγχο του host ΛΣ και απεικονίζουν τη φυσική μνήμη του εικονικού μηχανήματος στην φυσική μνήμη του πραγματικού.

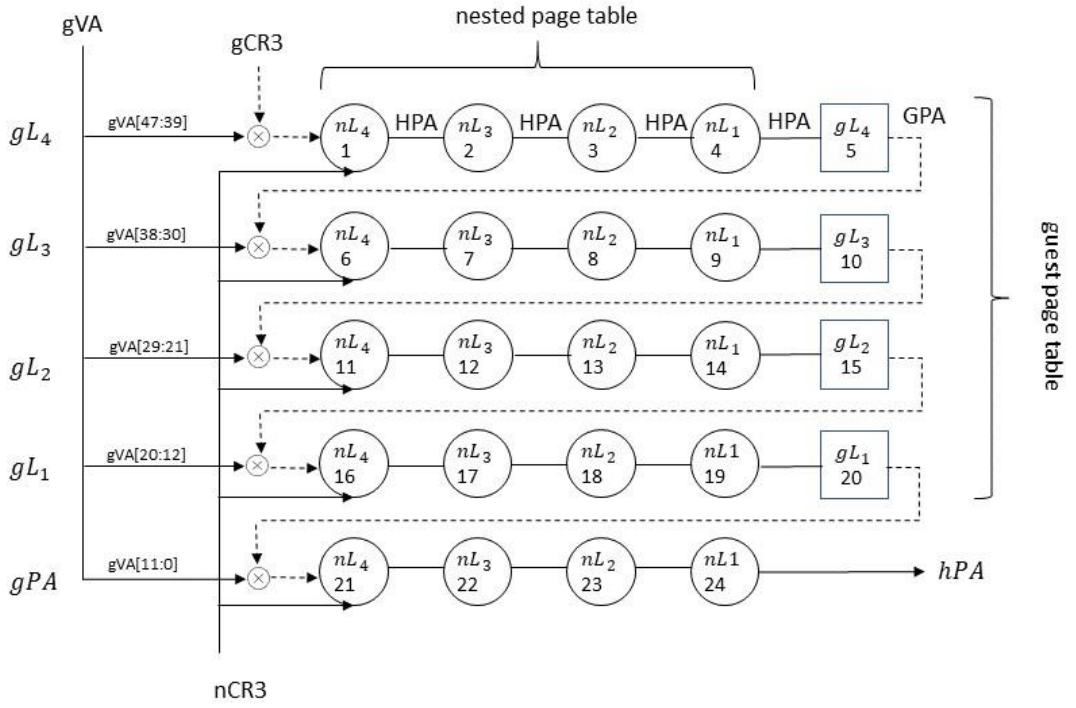


Σχήμα 2.8: Τα guest page tables βρίσκονται στον έλεγχο του guest OS που θεωρεί ότι εκτελείται σε ένα πραγματικό μηχάνημα. Τα nested pages tables ελέγχονται από το host OS και χρησιμοποιούνται από τον extended page table walker για να μεταφράζει guest φυσικές σε host φυσικές διευθύνσεις κατά τη διάσχιση των guest page tables.

Αν τα guest page tables αποτελούνται από n επίπεδα και τα nested από m, τότε ένα extended page table walk χρειάζεται:

1. n αναφορές μνήμης για την διάσχιση των guest page tables
2. m αναφορές μνήμης ανά φωλιασμένη διάσχιση των nested page tables (σύνολο $n \times m$ αναφορές)
3. n αναφορές μνήμης για την τελική μετάφραση της guest φυσικής διεύθυνσης του δεδομένου στην host φυσική.

Με βάση τα παραπάνω, συνολικά απαιτούνται $n \times m + n + m$ memory accesses ανά extended page table walk. Η όλη διαδικασία διάσχισης και ο υπολογισμός του κόστους γίνονται ευκολότερα κατανοητά μέσω του επόμενου σχήματος που παρατίθεται αυτούσιο από την εργασία στο [8].



Σχήμα 2.9: Διάσχιση των extended pages tables για τη μετάφραση μίας guest εικονικής σε host φυσική διεύθυνση. Για κάθε guest φυσική διεύθυνση που προκύπτει κατά τη διάσχιση των guest page tables, γίνεται αυτόματα μία ένθετη διάσχιση των nested page tables.

Καταλήγοντας, παρ' ότι οι επεκτάσεις υλικού επιτυγχάνουν σημαντική μείωση του κόστους μετάφρασης σε σχέση με τις software τεχνικές, εξακολουθούν να εισάγουν σημαντικές καθυστερήσεις που οφείλονται στον μεγάλο αριθμό προσβάσεων μνήμης που χρειάζεται ένα extended page table walk.

2.4 Σχετική Έρευνα

NUMA αρχιτεκτονικές

Στο υποκεφάλαιο 2.2 παρουσιάστηκαν κάποιες βασικές πολιτικές μνήμης στις οποίες έχει πρόσβαση ο χρήστης ενός NUMA μηχανήματος. Ωστόσο, υπάρχει και πλήθος άλλων σχετικών ερευνητικών συστημάτων που προτείνονται κατά καιρούς.

Στο [5] οι συγγραφείς παρουσιάζουν ένα σύστημα το οποίο παρακολουθεί δυναμικά την εκτέλεση ενός προγράμματος και βασιζόμενο σε μετρικές που υπολογίζει περιοδικά, αναπροσαρμόζει την τοποθέτηση μνήμης της εφαρμογής στους κόμβους. Παίρνει τα απαραίτητα στατιστικά για τον υπολογισμό των μετρικών κάνοντας δειγματοληψία στις εντολές του προγράμματος και διαβάζοντας μετρητές επίδοσης υλικού. Οι μηχανισμοί που έχουν σχεδιασθεί και στους οποίους βασίζεται το σύστημα, είναι η τοποθέτηση σελίδων στον ίδιο κόμβο με τα νήματα που τις προσπελαύνουν (**page co-location**), η μεταφορά σελίδων από υπερφορτωμένους κόμβους μνήμης σε κόμβους με λιγότερο φόρτο (**page interleaving**) και η δημιουργία πολλαπλών αντιγράφων για δημοφιλείς σελίδες που δέχονται αναφορές από νήματα πολλών διαφορετικών κόμβων (**page replication**). Η πρώτη και η τρίτη τεχνική αυξάνουν το ποσοστό των προσβάσεων στην τοπική μνήμη και μειώνουν την κίνηση στο δίκτυο διασύνδεσης, ενώ η δεύτερη αντιμετωπίζει φαινόμενα ανισοκατανομής και συμφόρησης. Τέλος, το σύστημα συμπεριλαμβάνει ένα αλγόριθμο που δρομολογεί νήματα που μοιράζονται δεδομένα σε επεξεργαστές του ίδιου κόμβου, στον βαθμό ωστόσο που τα νήματα δεν καταλήγουν να ανταγωνίζονται επιθετικά για τους πόρους του κόμβου.

Στο [27] προτείνεται ένα σύστημα σταθμισμένης κατανομής της μνήμης της εφαρμογής στους κόμβους. Πιο συγκεκριμένα, για πλήθος κόμβων N, σε κάθε έναν ανατίθεται βάρος w_i, ώστε $\sum_1^N w_i = 1$. Κάθε κόμβος αναλαμβάνει τέτοιο ποσοστό σελίδων μνήμης της εφαρμογής, όσο υποδηλώνει το βάρος του. Γίνεται η υπόθεση ότι τα νήματα εκτελούνται σε ένα υποσύνολο $W \subset \{1,2, \dots N\}$ των διαθέσιμων κόμβων οι οποίοι ονομάζεται κόμβοι-εργάτες (worker-nodes). Ωστόσο, και οι μη-εργάτες κόμβοι (non-worker nodes) αναλαμβάνουν να φιλοξενήσουν κάποιες από τις σελίδες μνήμης, ούτως ώστε να αξιοποιηθεί το όλο το διαθέσιμο εύρος ζώνης που παρέχει το μηχάνημα. Το βάρος w_i που ανατίθεται σε ένα κόμβο (εργάτη ή μη) εξαρτάται από το χαμηλότερο εύρος ζώνης μεταξύ αυτών που παρέχει στους κόμβους εργάτες. Πιο συγκεκριμένα, αν ως $b(n_i \to n_j)$ υποδηλώσουμε το εύρος ζώνης με το οποίο ο κόμβος j δέχεται δεδομένα από τον i, τότε $\min b_{n_i} = \min_{j \in W} b(n_i \to n_j)$. Το βάρος τότε ορίζεται ως $w_i = \frac{\min b_{n_i}}{\sum_{j=1}^N \min b_{n_j}}$.

Δημιουργείται έτσι η κατανομή $(w_1, w_2, \dots, w_N)$ με βάση την οποία μοιράζεται η μνήμη της εφαρμογής σε κόμβους. Επιπλέον, στην εργασία γίνεται μία κατηγοριοποίηση των εφαρμογών σε αυτές που είναι ευαίσθητες στο εύρος ζώνης και σε αυτές που είναι ευαίσθητες στον χρόνο πρόσβασης μνήμης (memory bandwidth vs memory access latency sensitivity). Οι εφαρμογές της πρώτης κατηγορίας ευνοούνται άμεσα από το σύστημα ως έχει μιας και αυτό προσπαθεί να μεγιστοποιήσει το εύρος ζώνης. Οι εφαρμογές της δεύτερης κατηγορίας ίσως έβλεπαν καλύτερη επίδοση αν κάποιο ποσοστό των σελίδων που έχουν ανατεθεί σε μη-εργάτες κόμβους μεταφέρονταν στις μνήμες των κόμβων-εργατών, αφού τότε θα αυξανόταν το ποσοστό των τοπικών προσβάσεων σε σχέση με τις απομακρυσμένες (και θα μειωνόταν ο χρόνος πρόσβασης μνήμης). Επειδή το σύστημα δεν γνωρίζει από πριν σε ποια από τις δύο κατηγορίες ανήκει μία εφαρμογή, κατά το χρόνο εκτέλεσης της, αφαιρεί δυναμικά από τους μη-εργάτες κόμβους ένα ποσοστό των σελίδων μνήμης που τους έχει ανατεθεί (με τέτοιο τρόπο ώστε η μνήμη που θα παραμείνει σε αυτούς να σέβεται τη σχέση των βαρών

τους) και την αναθέτει στους κόμβους-εργάτες (πάλι με τρόπο που καθορίζεται από τα βάρη τους). Η διαδικασία αυτή επαναλαμβάνεται περιοδικά, ενώ στο μεσοδιάστημα παρακολουθείται μέσω μετρητών επίδοσης ο αριθμός των άεργων κύκλων ανά δευτερόλεπτο (stalled cycles per second). Όσο παρατηρείται μείωση της μετρικής αυτής, η ανακατανομή μνήμης από μη-εργάτες σε εργάτες κόμβους συνεχίζεται. Μόλις παρατηρηθεί σταθεροποίηση, μπορεί να θεωρηθεί ότι έχει επιτευχθεί ένα τοπικό ελάχιστο (του ρυθμού άεργων κύκλων) και η διαδικασία σταματά.

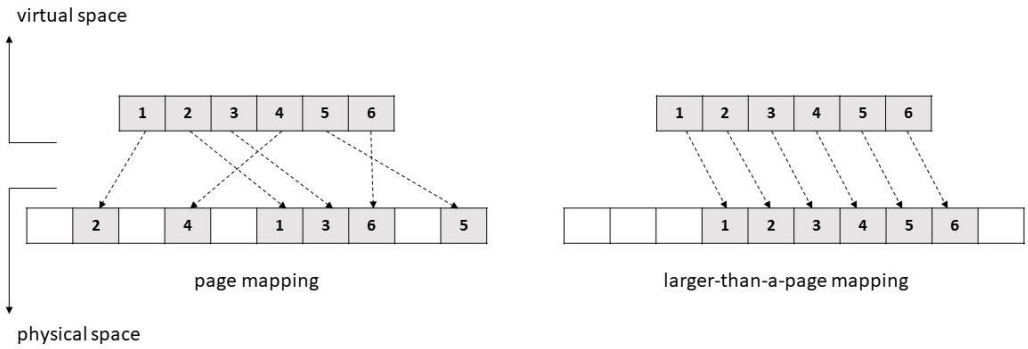
Ο Tam και οι συνεργάτες του [6] εντοπίζουν και προσπαθούν να επιλύσουν ένα εγγενές πρόβλημα στα συστήματα πολυεπεξεργασίας: νήματα που επεξεργάζονται τα ίδια δεδομένα αλλά εκτελούνται σε πυρήνες συνδεδεμένους σε διαφορετική ιεραρχία κρυφών μνημών, θα καταλήξουν να ακυρώνουν το ένα τα μπλοκ κρυφής μνήμης του άλλου λόγω του πρωτοκόλλου συνάφειας. Κάτι τέτοιο, όπως γίνεται κατανοητό, μπορεί να έχει πολύ σημαντικές συνέπειες στην επίδοση της εφαρμογής μιας και θα αυξήσει κατά πολύ τον ρυθμό αστοχίας κρυφής μνήμης. Προτείνουν ένα σύστημα που εντοπίζει δυναμικά τα νήματα που μοιράζονται δεδομένα, τα ομαδοποιεί σε clusters και προσπαθεί να δρομολογήσει τον κάθε cluster σε πυρήνες με κοινά επίπεδα cache. Για τον εντοπισμό των συνεργαζόμενων νημάτων, η μνήμη της εφαρμογής «σπάει» σε περιοχές (memory regions) μεγέθους ίσου με το μέγεθος μπλοκ κρυφής μνήμης επιπέδου 2 (L2 cache line size). Επειδή, η παρακολούθηση ολόκληρης της μνήμης της εφαρμογής θα είχε πολύ μεγάλο κόστος, οι συγγραφείς επιλέγουν να κάνουν δειγματοληψία, διαλέγοντας 256 περιοχές μνήμης οι οποίες θα βρίσκονται υπό παρακολούθηση. Οι περιοχές αυτές επιλέγονται κατά κάποιο τρόπο τυχαία. Για κάθε νήμα της εφαρμογής, το σύστημα διατηρεί ένα διάνυσμα 256 στοιχείων, όπου κάθε στοιχείο αντιστοιχεί σε μία από τις 256 επιλεγμένες περιοχές. Όταν ένα νήμα κάνει αναφορά σε μία διεύθυνση μνήμης και τα δεδομένα καταφθάνουν από μία απομακρυσμένη cache (βλέπε πρωτόκολλο συνάφειας), τότε εφόσον η εμπλεκόμενη διεύθυνση ανήκει σε κάποια από τις 256 επιλεγμένες περιοχές, η αντίστοιχη θέση του διανύσματος του νήματος αυξάνεται κατά 1. Τονίζεται σε αυτό το σημείο, ότι για να περιοριστεί το κόστος της παραπάνω διαδικασίας σε αποδεκτά επίπεδα, γίνεται δειγματοληψία στις εντολές μνήμης (δηλαδή για κάθε N εντολές εξετάζεται μόνο η μία). Έχοντας συλλέξει αρκετά δείγματα, το σύστημα συγκρίνει επαναληπτικά τα διανύσματα των νημάτων (χρησιμοποιώντας το εσωτερικό γινόμενο ως μέτρο ομοιότητας) και τα ομαδοποιεί σε clusters. Έπειτα τα νήματα των clusters ανατίθεται σε επεξεργαστές που μοιράζονται την ίδια ιεραρχία μνήμης, με τρόπο που σέβεται την αρχή εξισορρόπησης φόρτου σε όλους τους διαθέσιμους πόρους. Τονίζεται ότι το σύστημα που μόλις αναλύθηκε στοχεύει σε SMP συστήματα. Συμπεριλαμβάνεται σε αυτή την ενότητα γιατί οι ιδέες που υλοποιεί μπορούν με τον κατάλληλο σχεδιασμό να επεκταθούν και για την υποστήριξη NUMA αρχιτεκτονικών, αλλά και για την απλότητα και κομψότητα του.

Ο Lepers και οι συνεργάτες του [7] επικεντρώνονται στην ασυμμετρία που είναι πιθανό να παρουσιάζει το δίκτυο διασύνδεσης ενός NUMA μηχανήματος. Σε ένα ασύμμετρο δίκτυο, κάθε σύνδεσμος (link) μεταξύ δύο κόμβων δεν έχει απαραίτητα το ίδιο εύρος ζώνης με τους υπόλοιπους. Επιπλέον, μπορεί να παρουσιάζει διαφορετική ταχύτητα για κάθε μία από τις δύο κατευθύνσεις (έως και να είναι μονής κατεύθυνσης). Τέλος, δεν είναι απαραίτητα κοινόχρηστος, αλλά μπορεί να δεσμεύεται για χρήση από συγκεκριμένους κόμβους. Ως συνέπεια των παραπάνω, δύο κόμβοι μπορεί να αντιλαμβάνονται διαφορετικό εύρος ζώνης ο ένας ως προς τον άλλον. Επιπλέον, το μονοπάτι χαμηλότερου κόστους επικοινωνίας μεταξύ τους δεν καθορίζεται από το πλήθος των συνδέσμων, αλλά από το εύρος ζώνης που παρέχουν. Δεδομένου ότι τα ασύμμετρα δίκτυα διασύνδεσης παρέχουν μεγαλύτερη ευελιξία στην κατασκευή

(ειδικότερα όσο αυξάνεται το πλήθος των κόμβων ανά μηχάνημα), οι αλγόριθμοι δρομολόγησης που λαμβάνουν υπόψιν τα χαρακτηριστικά τους (asymmetry-aware scheduling) αποκτούν ιδιαίτερη σημασία. Στην εργασία αρχικά διεξάγονται κάποια πειράματα που καταδεικνύουν την επίδραση που έχει η επιλογή των κόμβων εκτέλεσης μία εφαρμογής στον χρόνο επικοινωνίας των νημάτων και συνεπώς στην συνολική επίδοση. Σε ένα μηχάνημα 8 κόμβων, 20 εφαρμογές εκτελούνται ξεχωριστά σε κάθε μία δυνατή επιλογή 3 κόμβων. Για μία από τις εφαρμογές η διαφορά επίδοσης μεταξύ βέλτιστης και χείριστης τοποθέτησης είναι 237%. Ο μέσος όρος διαφοράς είναι 40%. Η μεγάλη αυτή τιμή έγκειται στο ότι στις «καλές» τοποθετήσεις, οι κόμβοι που επιλέγονται συνδέονται μέσω διαύλων υψηλού εύρους ζώνης και οι επεξεργαστές τους ξοδεύουν λιγότερο χρόνο για περιμένοντας δεδομένα από μακρινούς κόμβους. Διεξάγεται ένα ακόμα πείραμα, στο οποίο μία εφαρμογή ευαίσθητη στην επιλογή των κόμβων της, εκτελείται σε όλες τις πιθανές δυάδες. Παρατηρείται ότι κάποιες δυάδες κόμβων με απόσταση δύο συνδέσμων επιτυγχάνουν ταχύτερη εκτέλεση από κάποιες άλλες με απόσταση ενός μόνο συνδέσμου. Καταδεικνύεται έτσι ότι στα ασύμμετρα δίκτυα, το βέλτιστο μονοπάτι μεταξύ δύο κόμβων δεν είναι απαραίτητα αυτό με το μικρότερό αριθμό συνδέσμων. Βασιζόμενοι στις παραπάνω παρατηρήσεις, οι συγγραφείς αναπτύσσουν ένα σύστημα αυτόματης τοποθέτησης εφαρμογών σε κόμβους που λαμβάνει μεταξύ άλλων υπόψιν και τα χαρακτηριστικά ασυμμετρίας του δικτύου διασύνδεσης. Πιο συγκεκριμένα, κατά το χρόνο εκτέλεσης της εφαρμογής και με χρήση μετρητών επίδοσης, εντοπίζονται τα νήματα που κάνουν χρήση κοινών δεδομένων και ομαδοποιούνται σε clusters (με απώτερο σκοπό όλα τα νήματα ενός cluster να εκτελεστούν στον ίδιο κόμβο). Για κάθε cluster υπολογίζεται μία μετρική που υποδεικνύει πόση κίνηση παράγουν τα νήματα του στο δίκτυο διασύνδεσης. Έπειτα, από όλες τις δυνατές τοποθετήσεις των clusters σε κόμβους, προκρίνονται εκείνες που δρομολογούν clusters με έντονες απαιτήσεις μνήμης σε «καλά δικτυωμένους» κόμβους (που δέχονται δηλαδή δεδομένα με υψηλό εύρος ζώνης). Προς αξιολόγηση του συστήματος, συγκρίνεται η επίδοση που έχουν διάφορες εφαρμογές όταν εκτελούνται με αυτό και όταν εκτελούνται με «χειροκίνητη» ανάθεση των νημάτων στην βέλτιστη τοποθέτηση (η οποία εντοπίζεται δοκιμάζοντας όλες τις πιθανές). Στις περισσότερες των περιπτώσεων, το σύστημα επιτυγχάνει σχεδόν ισάξια επίδοση, κάτι που σημαίνει ότι καταφέρνει να εντοπίζει δυναμικά και αυτόματα μία ευνοϊκή για την εφαρμογή τοποθέτηση που σέβεται τα χαρακτηριστικά ασυμμετρίας του δικτύου.

Εικονική μνήμη και larger-than-a-page translation schemes

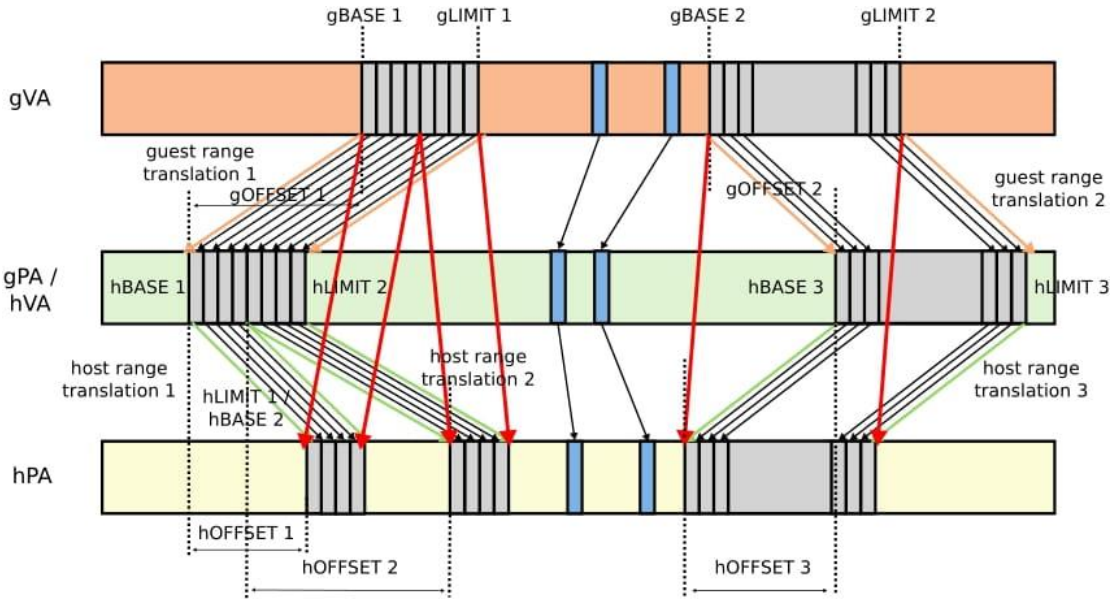
Η σελιδοποίηση μνήμης που αναλύθηκε στο υποκεφάλαιο 2.3 περιορίζει την μετάφραση σε επίπεδο σελίδας. Η συνεχόμενη απεικόνιση γειτονικών εικονικών σελίδων σε γειτονικά φυσικά πλαίσια (μία ιδιότητα γνωστή ως **contiguity**) μπορεί να συμπτύξει όλες αυτές τις σελίδες σε ένα μεγαλύτερο (larger-than-a-page) μπλοκ διευθύνσεων που απεικονίζεται στο φυσικό χώρο ως μία μονάδα και με την ίδια μετατόπιση. Αν μία τέτοια ιδιότητα αξιοποιηθεί κατάλληλα από μία επέκταση υλικού (πχ μία επέκταση της TLB cache), είναι δυνατόν να αυξηθεί κατά πολύ το εύρος (και κατ' επέκταση το ποσοστό ευστοχίας) της κρυφής μνήμης μεταφράσεων και να μειωθεί το address translation overhead των εφαρμογών. Λόγω αυτού λοιπόν έχουν αναπτυχθεί συστήματα που κατασκευάζουν τέτοιες συνεχόμενες απεικονίσεις από τον εικονικό στον φυσικό χώρο διευθύνσεων (larger-than-a-page mappings). Το σχήμα 2.10 δίνει ένα παράδειγμα της λειτουργίας τους.



Σχήμα 2.10: Αριστερά οι σελίδες απεικονίζονται διάσπαρτα σε πλαίσια φυσικής μνήμης. Δεξιά, κάθε σελίδα αποθηκεύεται σε πλαίσια με χρήση του ίδιου offset σχηματίζοντας μία συνεχόμενη απεικόνιση.

Το Translation Ranger [9] είναι μία επέκταση του Linux, η οποία μέσω ενός kernel daemon που τρέχει στο παρασκήνιο συνενώνει τα vmas των διεργασιών σε γειτονικά πλαίσια φυσικής μνήμης (page frame coalescing). Ο daemon επιλέγει για κάθε vma ένα anchor point το οποίο ορίζεται ως το ζεύγος μίας σελίδας του vma και ενός πλαισίου μνήμης και ουσιαστικά καθορίζει το physical memory region όπου θα αποθηκευτεί το vma. Έχοντας επιλέξει το anchor point, κατασκευάζεται η συνεχής απεικόνιση του vma μέσω διαδοχικών μεταφορών και ανταλλαγών πλαισίων.
Το σύστημα χειρίζεται τη φυσική μνήμη ως ένα σύνολο από coalesced και uncoalesced memory regions, όπου τα πρώτα αντιπροσωπεύουν περιοχές μνήμης που ήδη απεικονίζουν συνεχόμενα ένα vma. Επιλέγοντας τα anchor points μέσα στα uncoalesced memory regions, αποφεύγεται να καταστραφεί το contiguity που έχει παραχθεί προηγουμένως.

Το σύστημα Redundant Memory Mappings (RMM) [10] είναι πάλι μία επέκταση του Linux που εξασφαλίζει την συνεχή απεικόνιση μέσω memory reservation. Πιο συγκεκριμένα, κατά το πρώτο page fault εντός ενός vma δεσμεύεται ένα συνεχόμενο εύρος πλαισίων ώστε να το απεικονίσει συνεχόμενα. Η τεχνική αναφέρεται και ως eager allocation (πρόθυμη εκχώρηση μνήμης). Για εικονικά περιβάλλοντα εκτέλεσης, ο πυρήνας RMM πρέπει να εκτελείται και στο host και στο guest μηχάνημα. Στο σχήμα 2.11 φαίνεται ο σχηματισμός ranges σε εικονικό περιβάλλον εκτέλεσης, πρώτα από το guest virtual address space στο guest physical address space και μετά από το guest physical στο host physical address space.



Σχήμα 2.11

Πιθανώς ο βαθμός contiguity να μειωθεί λίγο λόγω του επιπλέον πεδίου διευθύνσεων (πχ στο σχήμα 2.11 το αριστερό range μεταφράζεται αυτούσιο στον guest φυσικό χώρο, ενώ στον host φυσικό σπάει σε δύο μικρότερα ranges). Τονίζεται ότι τα ranges μπορούν να έχουν αυθαίρετα μεγάλο μέγεθος και δεν υπόκεινται σε περιορισμούς ως προς το alignment.
Στην ίδια εργασία, προτείνεται επίσης η επέκταση υλικού Range TLB που αποθηκεύει μεταφράσεις σε επίπεδο range, και όχι σελίδας, και φαίνεται στο σχήμα 2.12.

RANGE TLB

gBASE	gLIMIT	2D OFFSET	PERMS

Σχήμα 2.12

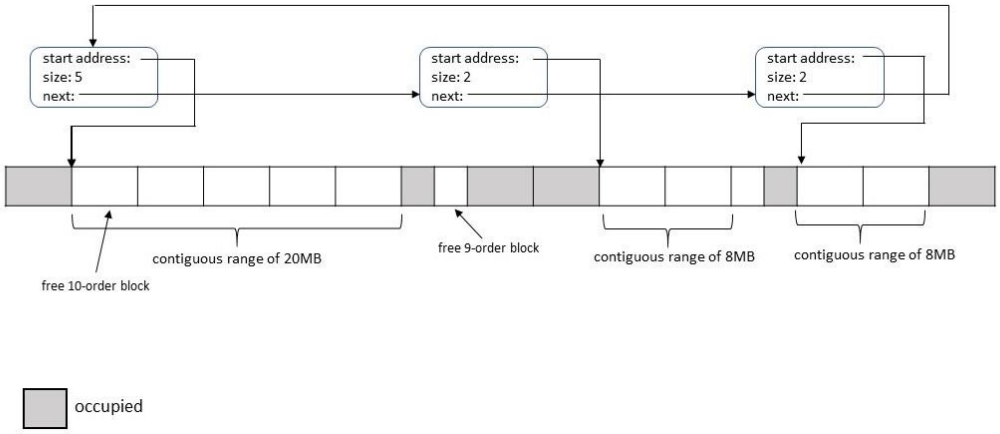
Κάθε εγγραφή του RangeTLB αποτελείται από 4 πεδία:

1. Την guest εικονική διεύθυνση αρχής του range (gBASE)
2. Την guest εικονική διεύθυνση τέλους του range (gLIMIT)
3. Το offset με το οποίο απεικονίζεται το range από τον guest εικονικό στον host φυσικό χώρο διευθύνσεων
4. Τα access control bits του range

Το σημαντικό είναι ότι κάθε εγγραφή του RangeTLB περιέχει την μετάφραση για ένα κομμάτι εικονικής μνήμης πολύ μεγαλύτερο από μία σελίδα. Συνεπώς, λίγα μόνο entries μπορεί να αρκούν για την κάλυψη ολόκληρης της μνήμης της διεργασίας και κατ' επέκταση το ποσοστό RangeTLB miss να είναι πολύ χαμηλό (ιδανικά 0%). Επειδή ο χρόνος πρόσβασης του RangeTLB υπολογίζεται να είναι περίπου ίδιος με τον χρόνο

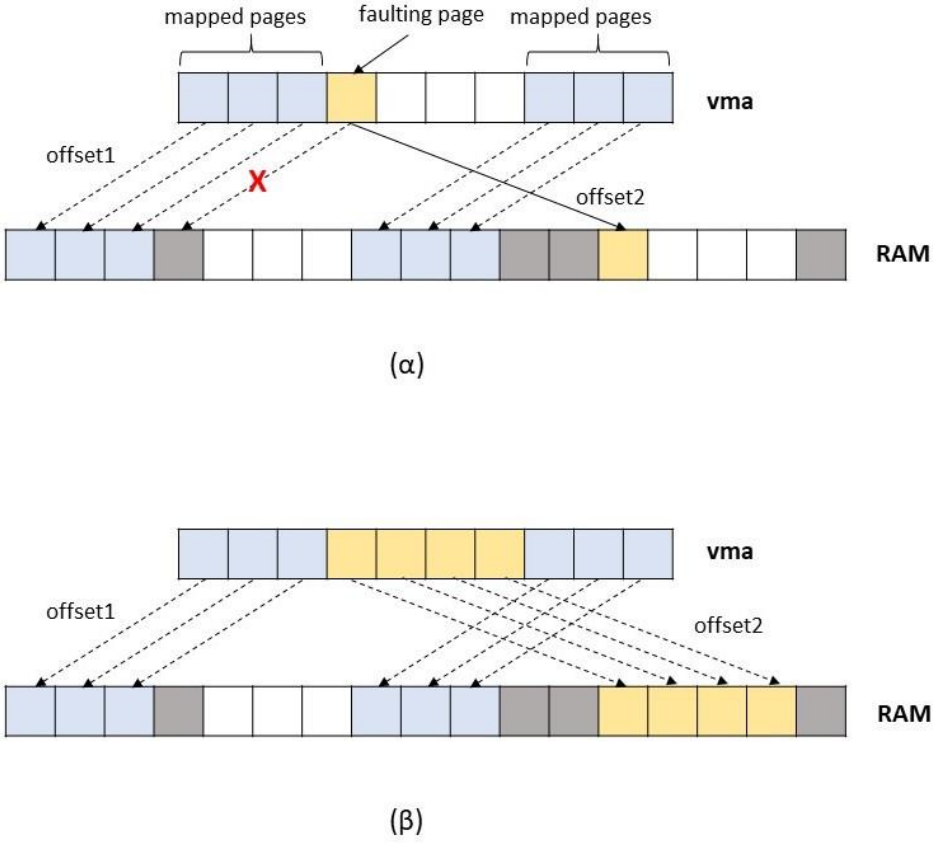
πρόσβασης του απλού TLB, το κόστος μετάφρασης διευθύνσεων πρακτικά εξαφανίζεται.
Τονίζεται επίσης, ότι κατ' αντιστοιχία με τα page tables είναι απαραίτητη μία δομή range page tables που θα αποθηκεύει τις μεταφράσεις όλων των ranges και η οποία θα είναι στον έλεγχο του ΛΣ. Τα range page tables θα διασχίζονται αυτόματα όταν συμβαίνει RangeTLB miss από τον range page table walker που πρέπει να προστεθεί στο υλικό κατ' αναλογία με τον γνωστό page table walker. Για υποστήριξη εικονικών μηχανήματων, θα πρέπει να γίνει επίσης η προσθήκη ενός nested range page table walker.

Το CAPaging [1] είναι επίσης μία επέκταση του Linux που παράγει contiguity. Μία από τις βασικές του δομές είναι ο χάρτης μνήμης, μία κυκλική λίστα που αποθηκεύει τις διευθύνσεις αρχής των ελεύθερων κομματιών φυσικής μνήμης (contiguous physical ranges). Κάθε range αποτελείται από πολλά γειτονικά buddy blocks μέγιστης τάξης. Ένα απλουστευμένο παράδειγμα του contiguity map δίνεται στο παρακάτω σχήμα.



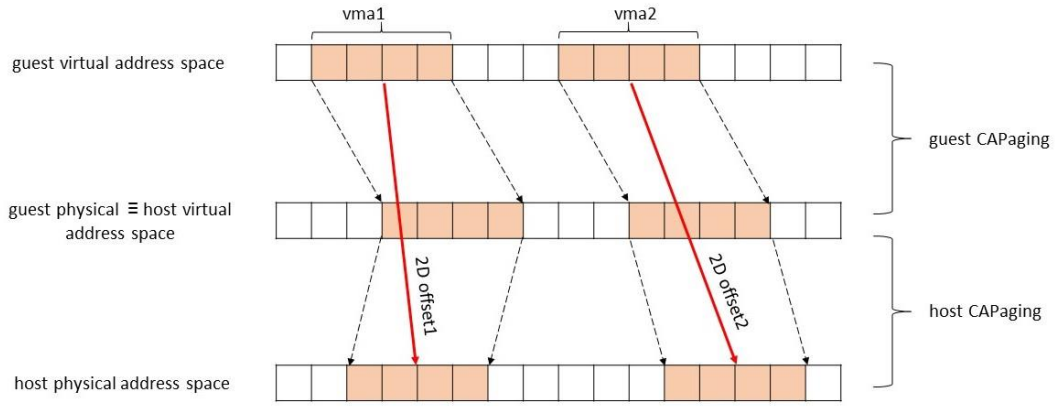
Σχήμα 2.13: Στιγμιότυπο της φυσικής μνήμης και του χάρτη contiguity map.

Όταν συμβαίνει το πρώτο page fault ενός vma, ο πυρήνας CAPaging αναζητά στον χάρτη μνήμης ένα physical range με μέγεθος μεγαλύτερο ή ίσο με το μέγεθος του vma. Το offset μεταξύ physical range και vma αποθηκεύεται στη δομή που περιγράφει το δεύτερο (vm_area_struct). Στα επόμενα page faults του vma, το CAPaging χρησιμοποιεί το offset ώστε να βρει το πλαίσιο που θα επεκτείνει τη συνεχή απεικόνιση και προσπαθεί να το δεσμεύσει. Επειδή το πλαίσιο-στόχος μπορεί να μην είναι ελεύθερο, το σύστημα επιτρέπει την χρήση περισσότερων του ενός offset ανά vma. Σε αυτή την περίπτωση, το τμήμα του vma όπου ανήκει η faulting σελίδα, αντιστοιχίζεται με ένα νέο range της λίστας contiguity map και απεικονίζεται με ένα νέο offset (σχήμα 2.14).
Σε αυτό το σημείο τονίζεται ότι το σύστημα διατηρεί έναν δείκτη, τον next-fit pointer, που δείχνει το range του contiguity map απ' όπου πρέπει να ξεκινήσει η επόμενη αναζήτηση. Κάθε φορά που ένα range επιλέγεται προς αντιστοίχιση ενός vma, ο next-fit pointer τίθεται να δείχνει στο αμέσως επόμενο της λίστας. Έτσι, κατά την επόμενη αναζήτηση range, αυτό που πόλις επιλέχθηκε θα εξεταστεί τελευταίο. Χάρη σε αυτό τον μηχανισμό, το σύστημα «δίνει χρόνο» στο vma να απεικονίσει όλες τις σελίδες του στο range επιλέχθηκε γι' αυτό, πριν ξεκινήσει ο ανταγωνισμός και από άλλα vmas.



Σχήμα 2.14: (α) Το πλαίσιο στόχος για την σελίδα 4 δεν είναι διαθέσιμο, οπότε το κομμάτι του vma που αποτελείται από τις σελίδες 4-7 απεικονίζεται στη φυσική μνήμη με διαφορετικό offset (β) Οι σελίδες 5-7 απεικονίζονται με βάση το νέο offset.

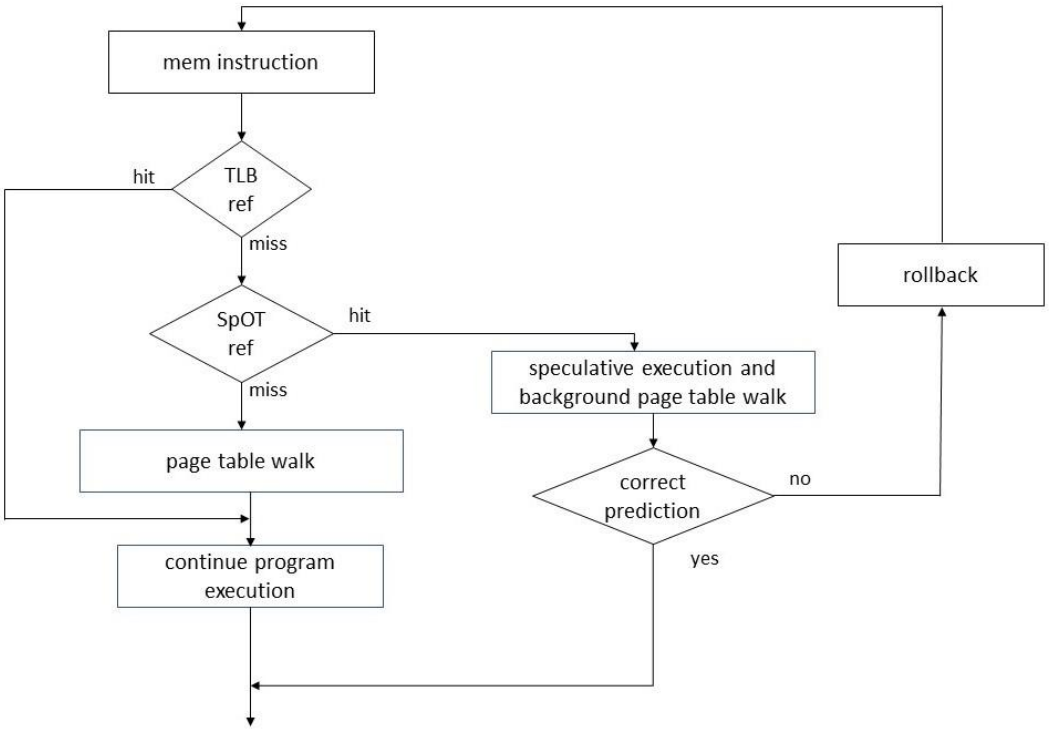
Για εκτέλεση προγραμμάτων σε εικονικό περιβάλλον, ο CAPaging πυρήνας εκτελείται ανεξάρτητα και στο εικονικό και στο φυσικό μηχάνημα. Το guest στιγμιότυπο κατασκευάζει συνεχόμενα mappings από το guest virtual στο guest physical address space, ενώ το host στιγμιότυπο από το host virtual στο host physical address space. Τα δύο στιγμιότυπα δηλαδή συνεργάζονται **αυτόματα** για τον σχηματισμό συνεχόμενων απεικονίσεων από το guest virtual στο host physical χώρο διευθύνσεων, όπως φαίνεται στο σχήμα 2.15.



Σχήμα 2.15: Ο guest CAPaging πυρήνας κατασκευάζει συνεχόμενα mappings μεταξύ guest virtual και physical address space και ο host πυρήνας μεταξύ host virtual και physical address space. Έτσι, σχηματίζονται με αυτόματο τρόπο συνεχόμενα mappings μεταξύ guest virtual και host physical address space.

Επιπλέον, στην εργασία προτείνεται η επέκταση υλικού SpOT (Speculative Offset-based Translation), που αξιοποιεί το contiguity που παράγεται από το CAPaging για να μειώσει το κόστος μετάφρασης. Το SpOT είναι μία PC-indexed[2] cache μνήμη που αποθηκεύει offsets μεταξύ vma και physical ranges. Όταν μία εντολή μνήμης προκαλεί TLB miss, γίνεται αναζήτηση στο SpOT με βάση την τρέχουσα τιμή του PC. Αν συμβεί SpOT hit, τότε γίνεται πρόβλεψη του ζητούμενου αριθμού πλαισίου ως $pfn = vpn + offset_{SpOT}$. Χρησιμοποιώντας τον PC ως δείκτη στο SpOT, κάθε εντολή μνήμης του προγράμματος συσχετίζεται έμμεσα με ένα vma, θεωρώντας ότι η εντολή θα κάνει την πλειοψηφία των αναφορών της σε αυτό το vma, όπως πράγματι ισχύει στις περισσότερες περιπτώσεις.
Επειδή ένα vma μπορεί να απεικονιστεί με περισσότερα του ενός offset ή κάποια εντολή μνήμης να κάνει αναφορές σε διαφορετικά vmas, είναι πιθανό το SpOT να κάνει κάποιες λανθασμένες προβλέψεις μετάφρασης (mispredictions). Γι' αυτό κάθε φορά που γίνεται μία πρόβλεψη μετά από TLB miss, ο επεξεργαστής την χρησιμοποιεί μεν χωρίς να περιμένει το αποτέλεσμα του page table walk, εισέρχεται όμως σε speculative execution mode. Στο background γίνεται από την MMU η διάσχιση των page tables για εύρεση της πραγματικής μετάφρασης. Όταν ολοκληρωθεί η διάσχιση, επιβεβαιώνεται η ορθότητα η μη της πρόβλεψης. Αν ήταν σωστή, ο επεξεργαστής κατάφερε να κρύψει το κόστος του TLB miss. Διαφορετικά, πρέπει να αντιστρέψει όσες λειτουργίες εκτέλεσε στο μεσοδιάστημα και να ξεκινήσει πάλι την εκτέλεση από την εντολή που προκάλεσε το TLB miss με την σωστή μετάφραση αυτή τη φορά. Το διάγραμμα ροής του τρόπου λειτουργίας δίνεται στο σχήμα 2.16.
Τονίζεται τέλος ότι το SpOT, για να μειώσει το ποσοστό mispredictions, περιλαμβάνει μετρητές εμπιστοσύνης για κάθε εγγραφή του.

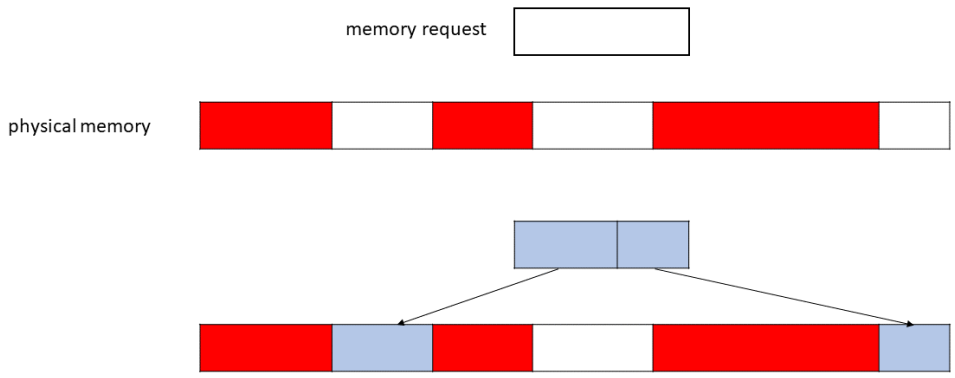
[2] PC είναι ο Program Counter καταχωρητής του επεξεργαστή



Σχήμα 2.16: Διάγραμμα ροής για την μετάφραση διευθύνσεων με την επέκταση υλικού SpOT.

Κεφάλαιο 3 Κίνητρο εργασίας

Στα υπολογιστικά συστήματα είναι γνωστό και σύνηθες το φαινόμενο κατακερματισμού της κύριας μνήμης (memory fragmentation) [11]. Ο κατακερματισμός μπορεί να είναι είτε εσωτερικός ή εξωτερικός. Στην εργασία ασχολούμαστε με τον δεύτερο, σύμφωνα με τον οποίο η ελεύθερη μνήμη του συστήματος σπάει σε πολλά μικρά κομμάτια, μεταξύ των οποίων παρεμβάλλονται άλλα δεσμευμένα. Έτσι, ένα αίτημα μίας διεργασίας για μνήμη πιθανώς να μην μπορεί να εξυπηρετηθεί (ή τουλάχιστον να μην μπορεί να εξυπηρετηθεί συνεχόμενα), παρ' ότι η συνολική ποσότητα ελεύθερης μνήμης επαρκεί. Ένα παράδειγμα φαίνεται στο παρακάτω σχήμα.

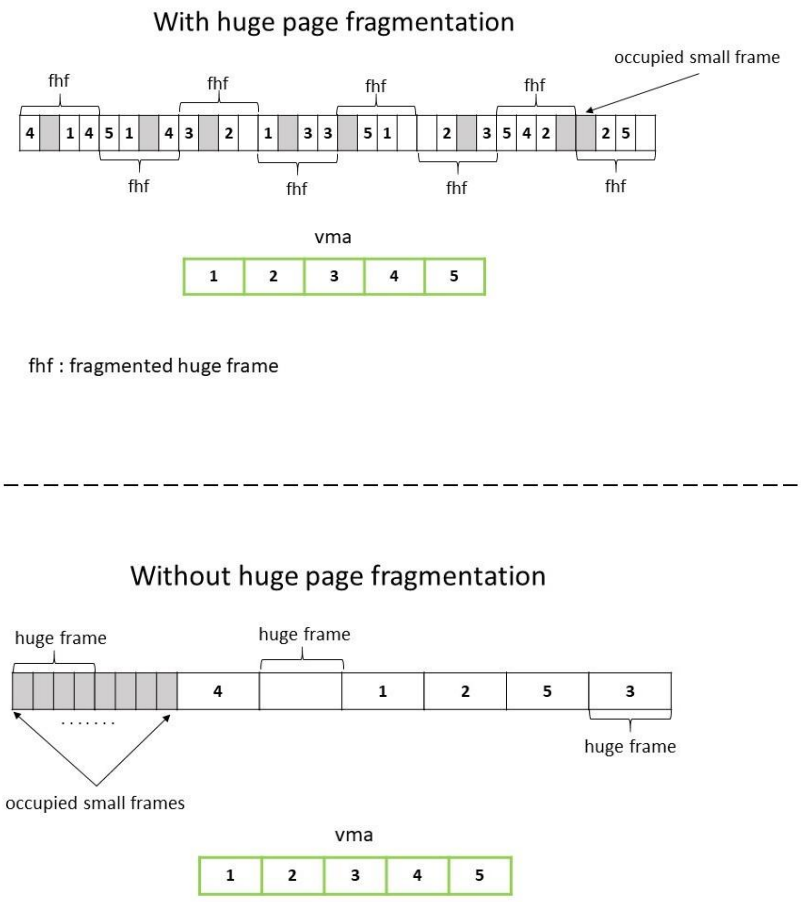


Σχήμα 3.1: Λόγω external fragmentation, το αίτημα μνήμης δεν μπορεί να εξυπηρετηθεί μονοκόμματα και πρέπει να σπάσει σε μη συνεχόμενα chunks.

3.1 Tradeoff Huge Pages vs NUMA Locality

Στο ΛΣ Linux όπου η εκχώρηση φυσικής μνήμης γίνεται με πλαίσια των 4KB και 2MB[3], το φαινόμενο external fragmentation εμφανίζεται όταν λίγα μόνο από τα 512 πλαίσια 4K που αποτελούν ένα πλαίσιο 2M είναι δεσμευμένα, και τα υπόλοιπα είναι ελεύθερα. Για την ακρίβεια ακόμα και ένα πλαίσιο-συστατικό να είναι δεσμευμένο, το μεγάλο πλαίσιο δεν μπορεί να χρησιμοποιηθεί για την απεικόνιση μίας 2MB σελίδας. Στο σχήμα 3.2 δίνεται ένα παράδειγμα του πώς απεικονίζεται ένα vma στην μνήμη με και χωρίς huge page fragmentation, θεωρώντας ότι ένα μεγάλο πλαίσιο αποτελείται από 4 μικρά.

[3] Ορισμένα μηχανήματα υποστηρίζουν πλαίσια 1GB αλλά δεν ασχολούμαστε με αυτά.



Σχήμα 3.2: Απεικόνιση ενός πίνακα στην φυσική μνήμη με και χωρίς huge page fragmentation

Τονίζεται ότι το Linux χρησιμοποιεί τον αλγόριθμό Buddy Memory Allocation [23] για την εκχώρηση πλαισίων μνήμης, ο οποίος όταν δέχεται αίτημα για ένα 4K πλαίσιο, προσπαθεί να βρει ένα που να ανήκει σε ένα ήδη fragmented 2M πλαίσιο, ούτως ώστε να αποφύγει τον κατακερματισμό. Ωστόσο καθώς πολλές διεργασίες εκτελούνται ταυτόχρονα δεσμεύοντας και ελευθερώνοντας πλαίσια φυσικής μνήμης, το φαινόμενο huge page fragmentation είναι πιθανό να συμβεί.
Στο σχήμα 4.3 φαίνεται, μέσα από τις πληροφορίες που δίνει το αρχείο /proc/buddyinfo του /proc filesystem [12], το παράδειγμα ακραίου huge page fragmentation της μνήμης του κόμβου 1 ενός NUMA μηχανήματος.
Τονίζεται σε αυτό το σημείο ότι η απεικόνιση της μνήμης μίας εφαρμογής με μεγάλα πλαίσια είναι σε ορισμένες περιπτώσεις πολύ σημαντική γιατί έτσι αυξάνεται το εύρος κάλυψης της TLB cache (TLB reach). Αυτό έχει σαν αποτέλεσμα να αυξάνεται το ποσοστό ευστοχίας TLB και να γίνονται λιγότερα page table walks. Οι εφαρμογές που εξαρτώνται περισσότερο από το μέγεθος πλαισίου είναι κυρίως αυτές που κάνουν πολλές αναφορές μνήμης και χρειάζονται μεταφράσεις διευθύνσεων με μεγαλύτερη συχνότητα.

```
bill@debian:~/liblinear-2.42$ numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9
node 0 size: 64419 MB
node 0 free: 63956 MB
node 1 cpus: 10 11 12 13 14 15 16 17 18 19
node 1 size: 64509 MB
node 1 free: 64160 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
bill@debian:~/liblinear-2.42$ cat /proc/buddyinfo
Node 0, zone      DMA      1      0      0      0      2      1      1      0      1      1      3
Node 0, zone    DMA32      4      3      3      4      3      3      6      6      3      2    738
Node 0, zone   Normal     99    156    142    123     37      9      2      0      1      2  15240
Node 1, zone   Normal  24442  24597  24587  24582  24584  24580  24578  24575  24575      2   3775
```

Σχήμα 3.3: Ο κόμβος 1 έχει συνολική ποσότητα ελεύθερης μνήμης 64G όπως φαίνεται από το output της εντολής numactl -H. Ωστόσο, μόνο τα περίπου 15G από αυτά είναι διαθέσιμα σε 2M πλαίσια και όλα τα υπόλοιπα είναι διαθέσιμα σε 4K πλαίσια.

Όταν τελειώνουν τα μεγάλα διαθέσιμα πλαίσια ενός NUMA κόμβου, μία εφαρμογή που εκτελείται σε αυτόν έχει δύο επιλογές. Είτε να συνεχίσει να απεικονίζει την μνήμη της τοπικά αλλά με μικρά πλαίσια ή να αρχίσει να χρησιμοποιεί τα μεγάλα πλαίσια μακρινών κόμβων. Στην πρώτη περίπτωση, έχει το πλεονέκτημα ότι όλες οι προσβάσεις κύριας μνήμης θα είναι τοπικές και θα αποφεύγεται το κόστος διάσχισης του δικτύου διασύνδεσης. Στην δεύτερη περίπτωση, απεικονίζοντας όλη τη μνήμη με μεγάλα πλαίσια, αυξάνεται το ποσοστό ευστοχίας TLB και μειώνεται το address translation overhead.
Για να μελετήσουμε ποια επιλογή είναι αποδοτικότερη από τις δύο παραπάνω, διεξάγουμε κάποιες πειραματικές εκτελέσεις προγραμμάτων. Πιο συγκεκριμένα, κάθε ένα από μία σειρά από benchmarks εκτελούμε 5 φορές όπως περιγράφεται παρακάτω:

1. **Local_2M**: όλη η μνήμη του προγράμματος απεικονίζεται στον τοπικό κόμβο και με χρήση αποκλειστικά μεγάλων πλαισίων
2. **Local_2M_4K**: όλη η μνήμη απεικονίζεται στον τοπικό κόμβο, η μισή ωστόσο με μεγάλα πλαίσια και η άλλη μισή με μικρά
3. **Local_4K**: όλη η μνήμη στον τοπικό κόμβο και με μικρά πλαίσια
4. **Interleave_2M**: η μνήμη αναμιγνύεται μεταξύ δύο κόμβων με εφαρμογή της Interleave πολιτικής και η απεικόνιση γίνεται αποκλειστικά με μεγάλα πλαίσια
5. **Remote_2M**: όλη η μνήμη στον μακρινό κόμβο και με μεγάλα πλαίσια

Τονίζεται ότι όλες οι εκτελέσεις γίνονται σε **εικονικό περιβάλλον** όπου το κόστος των TLB misses είναι μεγαλύτερο, όπως αναφέρεται και στην ενότητα 2.3.

Για κάθε εκτέλεση μετράμε μέσω performance counters[4]

1. τους συνολικούς κύκλους επεξεργασίας (T)
2. τα stalls λόγω L2 cache miss (S)[5]
3. τους κύκλους διάσχισης των page tables (PW)

Στον πίνακα 3.1 δίνεται συγκεντρωτικά μία συνοπτική περιγραφή των performance events που χρησιμοποιήθηκαν για να υπολογιστούν τα παραπάνω στατιστικά. Περισσότερες πληροφορίες συμπεριλαμβάνονται στο Software Development Manual της Intel [26].

Event	**Κωδική Ονομασία**	**Περιγραφή**
CPU cycles	CPU_CLK_UNHALTED. THREAD_P	Κύκλοι ρολογιού κατά τους οποίους ο επεξεργαστής δεν βρίσκεται σε κατάσταση halt
L2 cache miss stalls	CYCLE_ACTIVITY. STALLS_L2_PENDING	Execution stalls κατά τα οποία ένα L2 cache miss είναι ενεργό
page walk cycles for data TLB load misses	DTLB_LOAD_MISSES.W ALK_DURATION	Κύκλοι διάσχισης των page tables λόγω data TLB load misses
page walk cycles for data TLB store misses	DTLB_STORE_MISSES. WALK_DURATION	Κύκλοι διάσχισης των page tables λόγω data TLB store misses

*Πίνακας 3.1 Περιγραφή των performance events αρχιτεκτονικής **Broadwell** που χρησιμοποιήθηκαν στις πειραματικές εκτελέσεις του κεφαλαίου 3. Κάποια από αυτά χρησιμοποιούμε επίσης στο κεφάλαιο 5.*

[4] Για τα προγράμματα xsbench και liblinear-train που έχουν μία αρκετά χρονοβόρα φάση αρχικοποίησης χρησιμοποιήθηκε η βιβλιοθήκη PAPI [25] ώστε οι μετρήσεις να αφορούν μόνο την φάση υπολογισμού. Για το hashjoin οι μετρήσεις αφορούν ολόκληρη τη διάρκεια εκτέλεσης.

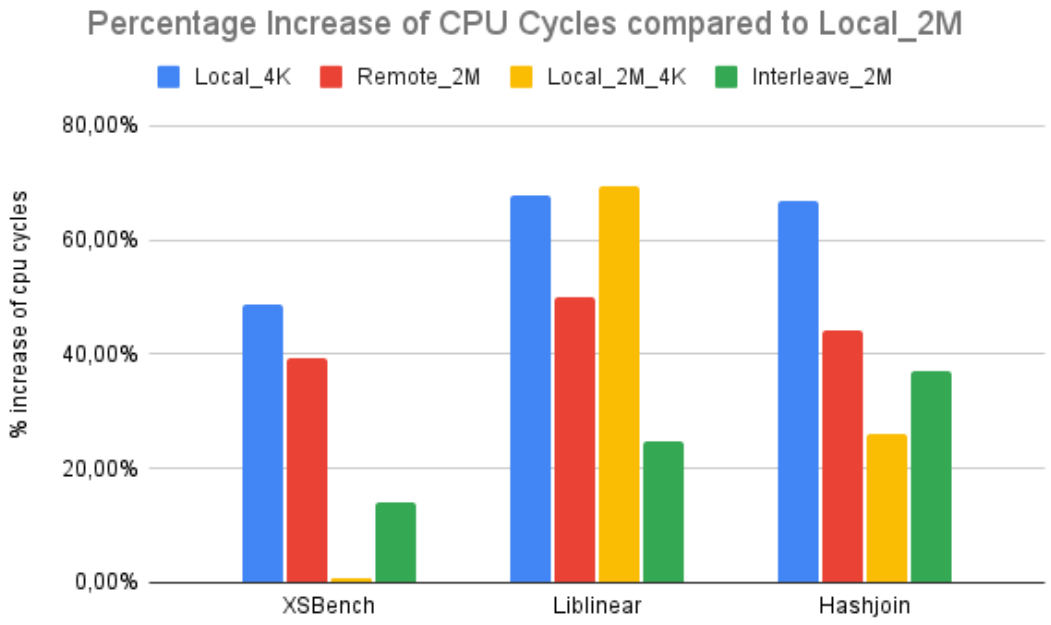
[5] Θέλουμε να μετρήσουμε τους κύκλους που ξοδεύει η κάθε εφαρμογή περιμένοντας δεδομένα από την *κύρια* μνήμη, οπότε ιδανικά θα χρειαζόμασταν ένα performance event για τα execution stalls λόγω **last level** cache misses. Ωστόσο, η μικροαρχιτεκτονική Broadwell του μηχανήματος μας δεν το διαθέτει, συνεπώς χρησιμοποιήσαμε προσεγγιστικά το event των execution stalls λόγω L2 cache miss.

Επειδή η εκτέλεση Local_2M έχει 100% ποσοστό τοπικών προσβάσεων κύριας μνήμης και 100% κάλυψη μνήμης με 2M πλαίσια, θεωρούμε τις τιμές των counters αυτής της εκτέλεσης ιδανικές και υπολογίζουμε την ποσοστιαία αύξηση που προκαλούν επί αυτών οι υπόλοιπες εκτελέσεις. Πιο συγκεκριμένα, υπολογίζουμε την ποσοστιαία αύξηση:

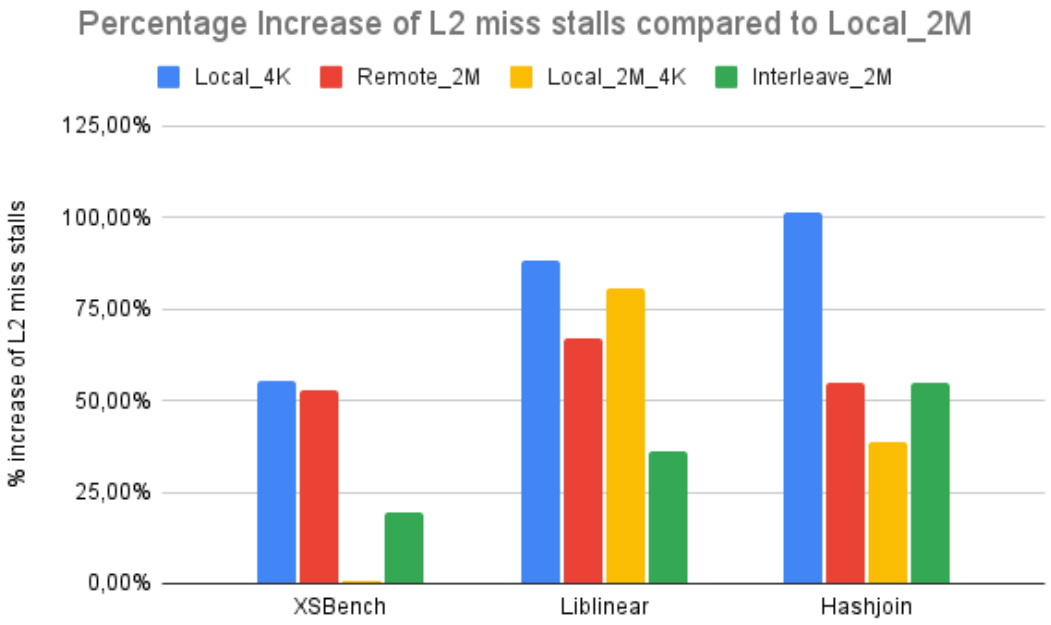
1. των συνολικών κύκλων επεξεργασίας
$$\frac{T_x - T_{Local\_2M}}{T_{Local\_2M}}$$
2. των stalls λόγω L2 Cache miss
$$\frac{S_x - S_{Local\_2M}}{S_{Local\_2M}}$$
3. των κύκλων διάσχισης των page tables
$$\frac{PW_x - PW_{Local\_2M}}{PW_{Local\_2M}}$$

όπου $x \in \{Local\_2M\_4K,\ Interleave\_2M,\ Local\_4K,\ Remote\_2M\}$

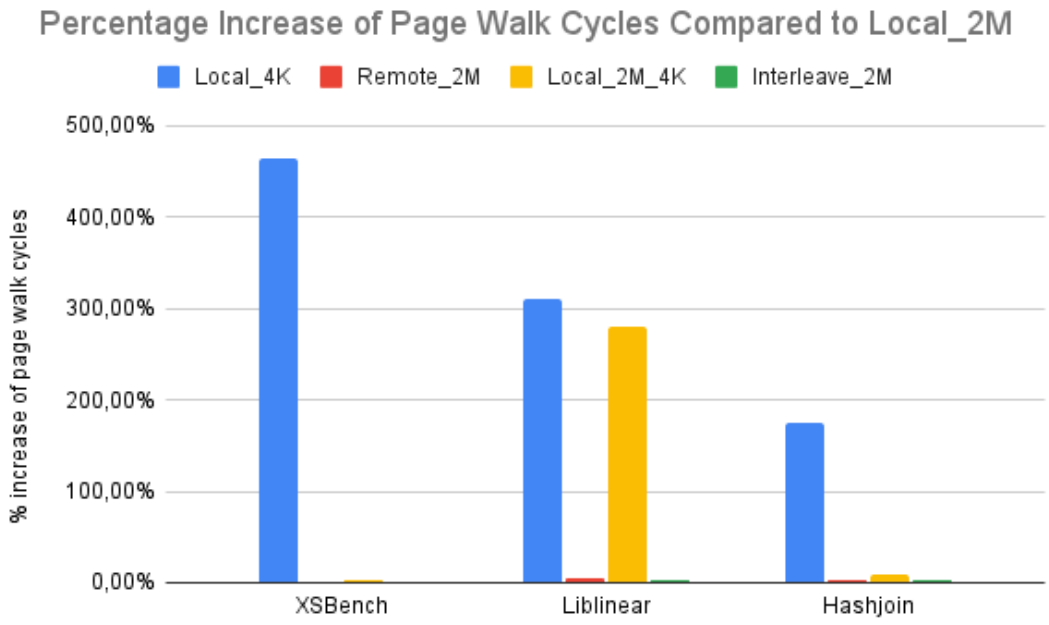
Τα αποτελέσματα που προέκυψαν φαίνονται παρακάτω στα διαγράμματα 3.4, 3.5 και 3.6.



Σχήμα 3.4: Ποσοστιαία αύξηση των κύκλων CPU σε σχέση με την Local_2M εκτέλεση.



Σχήμα 3.5: Ποσοστιαία αύξηση των L2 Cache Miss Stalls σε σχέση με την Local_2M εκτέλεση.

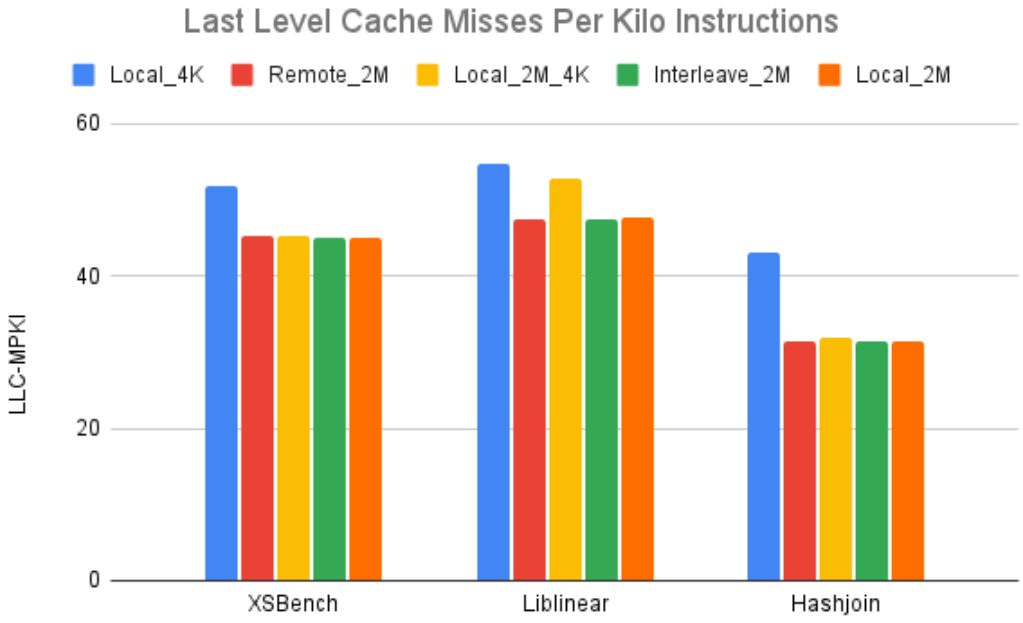


Σχήμα 3.6: Ποσοστιαία αύξηση των page walk cycles σε σχέση με την Local_2M εκτέλεση.

Παρατηρήσεις

- Στο διάγραμμα 3.4 βλέπουμε ότι σε όλα τα προγράμματα η εκτέλεση Local_4K (μπλε μπάρα) προκαλεί μεγαλύτερη αύξηση κύκλων CPU σε σύγκριση με την Remote_2M (κόκκινη μπάρα). Αυτό αποτελεί μία ένδειξη ότι η χρήση huge pages είναι πιο σημαντική από το NUMA locality.
- Η Local_2M_4K εκτέλεση προκαλεί μεγάλη αύξηση κύκλων CPU στο πρόγραμμα Liblinear-train, μικρή στο Hashjoin και μηδενική στο XSBench (κίτρινες μπάρες διαγράμματος 3.4). Αυτό έχει την εξήγηση ότι στο Liblinear-train, το hot κομμάτι μνήμης[6] είναι από τα τελευταία που αρχικοποιούνται και έτσι όταν έρθει η ώρα να απεικονιστεί στη φυσική μνήμη, τα μεγάλα πλαίσια του τοπικού κόμβου έχουν ήδη εξαντληθεί. Συνεπώς, αποθηκεύεται με μικρά πλαίσια και προκαλούνται πολλά TLB misses. Από την άλλη στα XSBench και Hashjoin, το hot κομμάτι μνήμης είναι το πρώτο που αρχικοποιείται και έτσι προλαβαίνει τα μεγάλα πλαίσια. Για τα υπόλοιπα κομμάτια μνήμης των προγραμμάτων αυτών, το μέγεθος σελίδας που χρησιμοποιείται για την απεικόνιση τους δεν φαίνεται να επηρεάζει την επίδοση του TLB.
- Στο σχήμα 3.5 βλέπουμε ότι οι εκτελέσεις Remote_2M και Interleave_2M αυξάνουν τα L2 Cache Miss Stalls όπως είναι αναμενόμενο αφού έχουν 100% και 50% ποσοστό απομακρυσμένων προσβάσεων κύριας μνήμης αντίστοιχα. Από την άλλη, όπως φαίνεται στο σχήμα 3.6, δεν αυξάνουν τους κύκλους διάσχισης των page tables σε σχέση με την Local_2M εκτέλεση, κάτι που είναι αναμενόμενο αφού σε όλες τις περιπτώσεις χρησιμοποιούνται μόνο μεγάλα πλαίσια μνήμης.
- Στις εκτελέσεις που χρησιμοποιούνται μικρά πλαίσια (Local_2M_4K και Local_4K) αυξάνονται όπως αναμένεται οι κύκλοι διάσχισης των page tables αφού συμβαίνουν περισσότερα TLB misses (σχήμα 3.6). Το ενδιαφέρον είναι ότι στις ίδιες εκτελέσεις αυξάνονται και τα L2 Cache Miss Stalls (σχήμα 3.5), το οποίο είναι μη αναμενόμενο αφού όλη η μνήμη παραμένει στον τοπικό κόμβο και δεν εισάγονται remote accesses. Για να το εξηγήσουμε, μετράμε την μετρική LLC-MPKI (Last Level Cache Misses Per Kilo Instructions) για κάθε εκτέλεση:

[6] Hot κομμάτι μνήμης θεωρούμε εκείνο που δέχεται τις περισσότερες αναφορές και που είναι υπεύθυνο για τα περισσότερα TLB και Cache misses.



Σχήμα 3.7: LLC-MPKI. Παρατηρούμε αύξηση της μετρικής στις περιπτώσεις που χρησιμοποιούνται μικρά πλαίσια

Παρατηρούμε ότι όταν χρησιμοποιείται το μικρό μέγεθος πλαισίου, το LLC-MPKI αυξάνεται. Το μικρό μέγεθος πλαισίου συνεπάγεται 1) μεγαλύτερο ποσοστό αστοχίας TLB και 2) περισσότερα memory references ανά TLB miss αφού αυξάνουν τα επίπεδα των page tables. Κατ' επέκταση, ο page table walker ενεργοποιείται συχνότερα και με περισσότερο φόρτο και ασκεί μεγαλύτερη πίεση στα επίπεδα cache, διώχνοντας χρήσιμα cache lines του προγράμματος (cache pollution). Γι' αυτό συμβαίνει η αύξηση των L2 Cache Miss Stalls που επισημάνθηκε παραπάνω για τις εκτελέσεις Local_4K και Local_2M_4K.

Σαν σύνοψη, βλέπουμε ότι η χρήση μικρών πλαισίων κρύβει δύο διαφορετικά overheads. Πρώτον, την αύξηση των TLB misses και του κόστους διάσχισης των page tables και δεύτερον, την αύξηση των Cache Miss Stalls λόγω του φαινομένου cache pollution από τον page table walker. Από την άλλη η απεικόνιση μνήμης με μεγάλα πλαίσια μακρινού κόμβου, εισάγει ένα μόνο overhead, την αύξηση των Cache Miss Stalls λόγω των remote memory accesses.
Συμπερασματικά, κατά την εκτέλεση εφαρμογών σε εικονικό περιβάλλον, η χρήση μεγάλων σελίδων για την απεικόνιση της μνήμης δείχνει να έχει μεγαλύτερη σημασία από την τοπικότητα των δεδομένων.

3.2 Tradeoff Memory Contiguity vs NUMA Locality

Ακόμα και με τη χρήση μεγάλων πλαισίων, υπάρχουν εφαρμογές για τις οποίες το address translation overhead παραμένει μη αμελητέο. Γι' αυτό έχουν αναπτυχθεί τα συνεργατικά συστήματα υλικού/λογισμικού που περιγράφονται στην ενότητα 2.4 του κεφαλαίου 2, στα οποία το λογισμικό (πχ CAPaging) απεικονίζει σελίδες συνεχόμενα από τον εικονικό στο φυσικό χώρο διευθύνσεων (contiguity) και το υλικό (πχ RangeTLB) αξιοποιεί αυτή την ιδιότητα για να μειώσει κι άλλο το translation overhead. Σε περιπτώσεις που ο κόμβος εκτέλεσης μίας εφαρμογής έχει υποστεί εξωτερικό κατακερματισμό και τα ελεύθερα κομμάτια μνήμης του έχουν μικρό μέγεθος, τίθεται το ερώτημα αν αξίζει να αξιοποιηθούν τα ελεύθερα, μη κατακερματισμένα κομμάτια φυσικής μνήμης μακρινών κόμβων, ούτως ώστε να παραχθεί καλύτερο contiguity. Το πλεονέκτημα αυτή της επιλογής είναι ότι οι επεκτάσεις υλικού όπως το RangeTLB, λόγω υψηλότερου contiguity, θα μπορούν να μειώσουν αποτελεσματικότερα το address translation overhead, ενώ από την άλλη το μειονέκτημα είναι ότι το κόστος κύριας μνήμης θα αυξηθεί αφού κάποια κομμάτια θα έχουν αποθηκευτεί σε remote κόμβους και οι προσβάσεις σε αυτά θα διασχίζουν το δίκτυο διασύνδεσης.

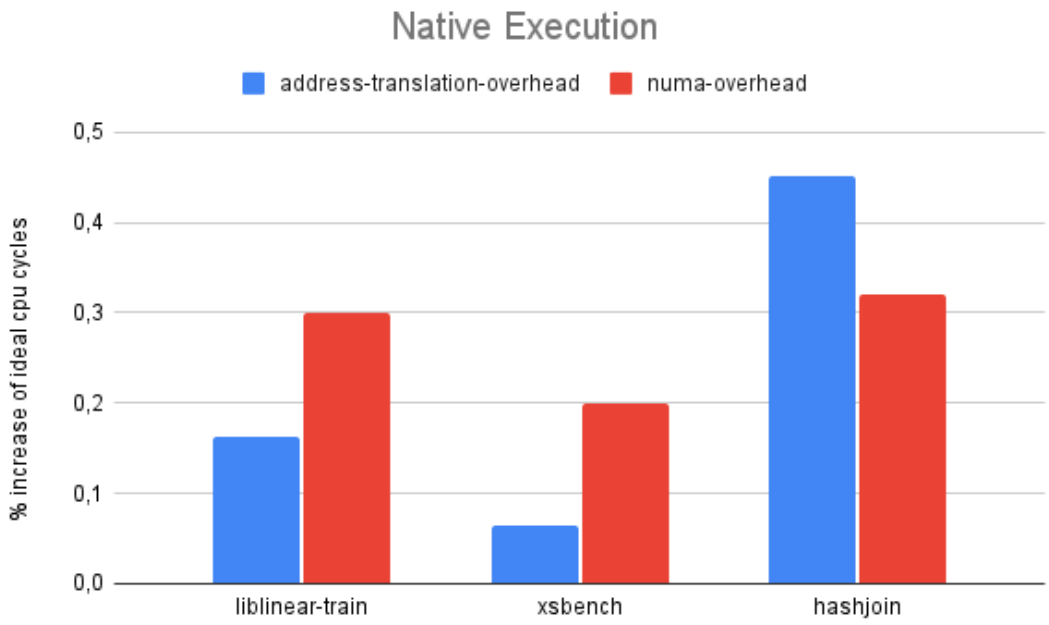
Για να μελετήσουμε το tradeoff Contiguity vs NUMA Locality που περιγράψαμε, αρχικά ορίζουμε την ποσότητα του ιδανικού χρόνου εκτέλεσης, T_{ideal}, ως το πλήθος των κύκλων που θα χρειαζόταν το πρόγραμμα για να ολοκληρωθεί αν είχε μηδενικό κόστος μετάφρασης διευθύνσεων (πχ 100% TLB hit ratio) και 100% τοπικές προσβάσεις κύριας μνήμης. Για να βρούμε το T_{ideal} ενός προγράμματος, το εκτελούμε με χρήση της Local πολιτικής μνήμης και αφαιρούμε από τους συνολικούς κύκλους εκτέλεσης τους κύκλους εκείνους που η μονάδα διαχείρισης TLB misses ήταν ενεργή. Έχοντας ορίσεις το T_{ideal} όπως περιγράφεται παραπάνω, τρέχουμε μία σειρά από προγράμματα σε φυσικό και εικονικό περιβάλλον εκτέλεσης και υπολογίζουμε τις εξής ποσότητες:

1. Address-Translation-Overhead – η ποσοστιαία αύξηση επί των ιδανικών κύκλων εκτέλεσης λόγω μετάφρασης διευθύνσεων. Υπολογίζεται ως
$$O_{Trans} = \frac{PWC}{T_{ideal}}$$
όπου PWC (*PageWalkCycles*) είναι οι κύκλοι που η μονάδα διαχείρισης των TLB misses είναι ενεργή. Σημειώνεται ότι η μνήμη σε αυτή την εκτέλεση απεικονίζεται με μεγάλα πλαίσια

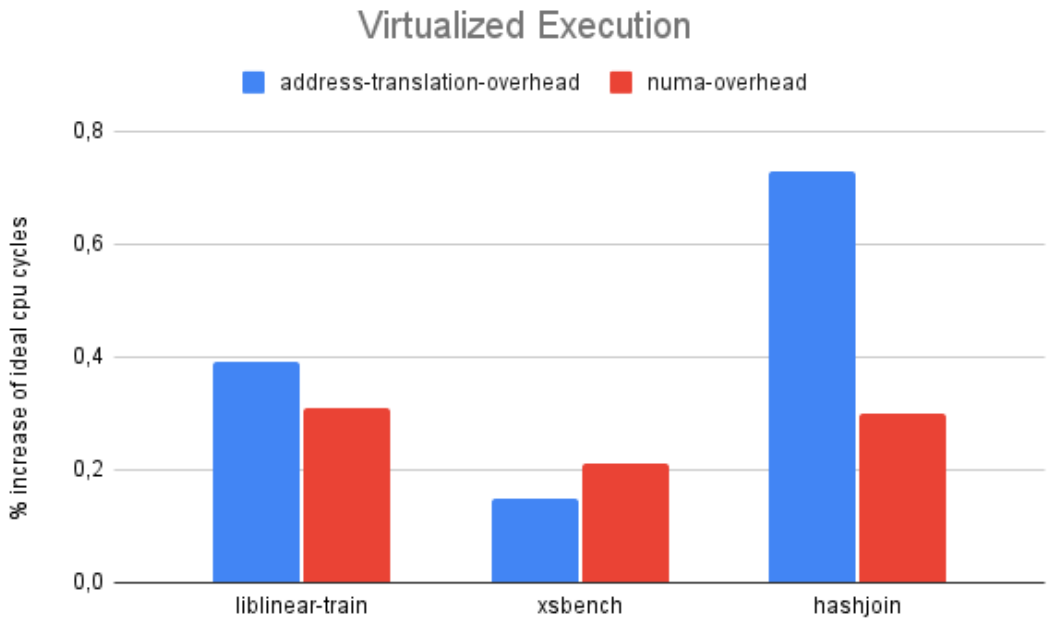
2. NUMA-Overhead – η ποσοστιαία αύξηση των ιδανικών κύκλων εκτέλεσης όταν το 50% της μνήμης αποθηκεύεται στον remote κόμβο. Υπολογίζεται ως
$$O_{NUMA} = \frac{T_{Interleave} - T_{Local}}{T_{ideal}}$$
όπου T_{policy} είναι ο χρόνος εκτέλεσης του προγράμματος σε κύκλους όταν εφαρμόζεται η $policy$ πολιτική μνήμης.

Επισημαίνεται ότι όσο υψηλότερο είναι το translation overhead τόσο περισσότερη σημασία αποκτά το Contiguity, ενώ όσο υψηλότερο είναι το NUMA overhead τόσο σημαντικότερο γίνεται το NUMA Locality. Βρίσκοντας ένα πρόγραμμα με υψηλό translation και χαμηλό NUMA overhead, θα σήμαινε ότι αυτό το πρόγραμμα προτιμά συνεχόμενα και remote mappings παρά local και fragmented (πχ σχήμα 1.1 του κεφαλαίου 1), μιας και μία επέκταση υλικού, όπως το RangeTLB, θα μπορούσε να

μειώσει το υψηλό translation overhead, ενώ παράλληλα το τίμημα που θα πληρωνόταν λόγω παράκαμψης του NUMA Locality θα ήταν χαμηλό.
Τονίζεται ωστόσο ότι οι δύο ποσότητες είναι ως ένα βαθμό συσχετισμένες υπό την έννοια ότι όσο υψηλότερο translation overhead έχει ένα πρόγραμμα, τόσες περισσότερες αναφορές μνήμης κάνει και συνεπώς τόσο περισσότερα cache misses θα έχει. Όμως ένα πρόγραμμα με πολλά cache misses είναι ευαίσθητο στην επιλογή του κόμβου που αποθηκεύει την μνήμη του, δηλαδή έχει υψηλό NUMA overhead. Από την άλλη, δύο παράγοντες που μπορεί να διαφοροποιήσουν τα παραπάνω overheads είναι ότι 1) ένα TLB miss είναι συνήθως ακριβότερο από ένα cache miss (ιδίως σε εικονικό περιβάλλον) και 2) το ποσοστό ευστοχίας TLB συνήθως είναι υψηλότερο από το ποσοστό ευστοχίας cache.
Έχοντας αναφέρει τα παραπάνω η σύγκριση των δύο overhead ανά πρόγραμμα φαίνεται στα σχήματα 3.7 και 3.8.



Σχήμα 3.8: Σύγκριση address translation και NUMA overhead ανά πρόγραμμα όταν η εκτέλεση γίνεται απευθείας στο φυσικό μηχάνημα



Σχήμα 3.9: Σύγκριση address translation και NUMA overhead ανά πρόγραμμα όταν η εκτέλεση γίνεται μέσα σε εικονικό μηχάνημα

Βλέπουμε ότι στην εικονική εκτέλεση το κόστος μετάφρασης των προγραμμάτων Liblinear-Train και Hashjoin ξεπερνά το κόστος απομακρυσμένων προσβάσεων μνήμης. Μάλιστα, στο Hashjoin το κόστος μετάφρασης είναι παραπάνω από διπλάσιο. Αντίθετα, στην εκτέλεση στο πραγματικό μηχάνημα, όπου τα TLB misses είναι φθηνότερα, βλέπουμε ότι αντιστρέφεται η σύγκριση για το Liblinear-train, ενώ για το Hashjoin παρ' ότι το translation overhead μειώνεται παραμένει μεγαλύτερο του NUMA.

Συμπερασματικά, το γεγονός ότι το Liblinear-train που είναι μία πραγματική εφαρμογή (σε αντίθεση με το Hashjoin που είναι microbenchmark), κατά την εκτέλεση σε εικονικό περιβάλλον έχει υψηλότερο κόστος μετάφρασης από κόστος απομακρυσμένων προσβάσεων μνήμης, μας δίνει κίνητρο ώστε να υλοποιήσουμε μία επέκταση του συστήματος CAPaging. Η επέκταση αυτή όταν διαπιστώνει ότι η μνήμη του τοπικού κόμβου εκτέλεσης μίας διεργασίας έχει υποστεί εξωτερικό κατακερματισμό, θα εξετάζει κομμάτια φυσικής μνήμης και μακρινών κόμβων για την τοποθέτηση της μνήμης.

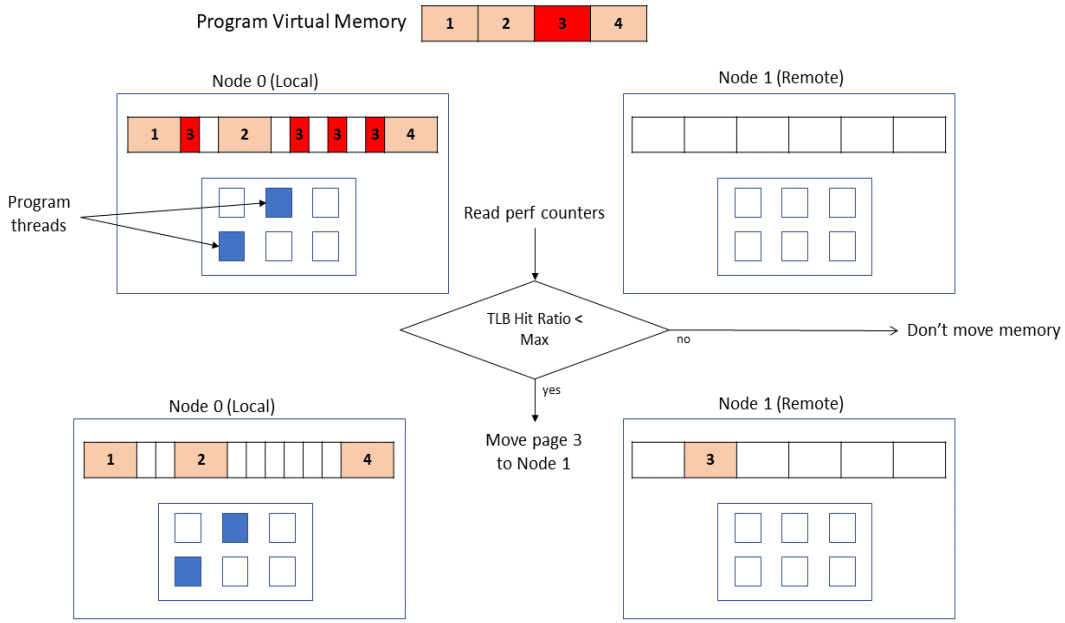
Κεφάλαιο 4 Υλοποίηση

4.1 Remote Huge Pages (RHP)

Τα αποτελέσματα της ενότητας «Tradeoff Huge Pages vs NUMA Locality» του κεφαλαίου 3 μας δίνουν κίνητρο για την ανάπτυξη ενός συστήματος που θα εντοπίζει κατά τη διάρκεια εκτέλεσης ενός προγράμματος αν κάποιες σελίδες του απεικονίζονται στη φυσική μνήμη με μικρά πλαίσια και θα αποφασίζει αν η επίδοση θα βελτιωνόταν με τη μεταφορά των σελίδων εκείνων σε έναν απομακρυσμένο κόμβο, όπου θα απεικονισθούν με μεγάλα πλαίσια. Σε μία τέτοια ιδέα θα ήταν απαραίτητο ένα στάδιο offline εκτέλεσης[7] του προγράμματος κατά το οποίο θα υπολογίζεται η μέγιστη δυνατή τιμή του TLB Hit Ratio (απεικόνιση μνήμης με 2M πλαίσια). Στις επόμενες εκτελέσεις του προγράμματος, το σύστημα θα παρακολουθεί δυναμικά μέσω performance counters το ποσοστό ευστοχίας TLB. Αν απέχει πολύ από την μέγιστη τιμή του, τότε

1. θα εντοπίζονται οι σελίδες που έχουν απεικονιστεί στον τοπικό κόμβο με μικρά πλαίσια
2. θα μεταφέρονται οι παραπάνω σελίδες στην μνήμη ενός απομακρυσμένου κόμβου, όπου η απεικόνιση θα γίνεται με το μεγάλο μέγεθος πλαισίου.

Η περιγραφή του εν λόγω συστήματος απεικονίζεται και στο ακόλουθο σχήμα.

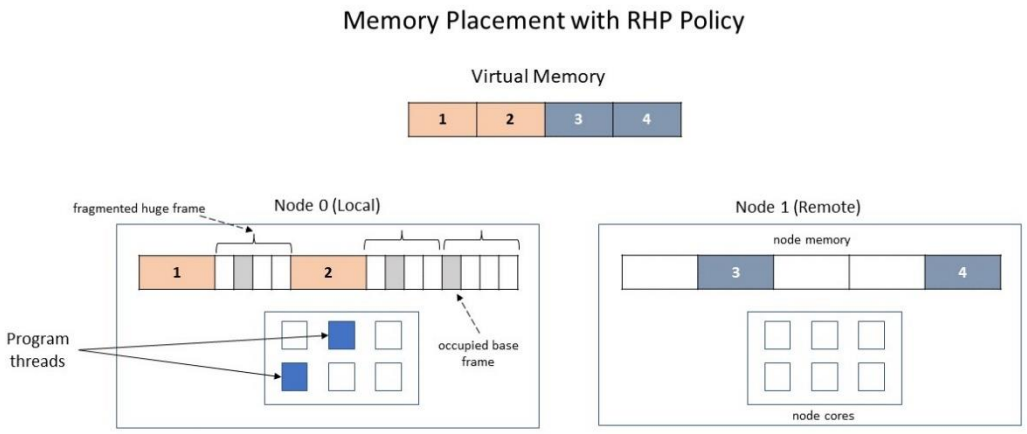


Σχήμα 4.1: Η μνήμη του προγράμματος απεικονίζεται αρχικά στον τοπικό κόμβο με χρήση και κάποιων μικρών πλαισίων. Στη συνέχεια προσδιορίζεται το ποσοστό ευστοχίας TLB. Αν το ποσοστό είναι μακριά από την μέγιστη τιμή του (η οποία είναι γνωστή μέσω offline εκτέλεσης), τότε γίνεται μεταφορά της σελίδας 3 στον μακρινό κόμβο, όπου απεικονίζεται σε ένα μεγάλο πλαίσιο.

Δεν μπορέσαμε να βρούμε στο ΛΣ Linux κάποια σχετική κλήση συστήματος που μαζί με την μεταφορά μνήμης από κόμβο σε κόμβο θα εγγυάται/επιβάλλει την χρήση μεγάλου φυσικού πλαισίου για την νέα απεικόνιση. Επομένως, δεν μπορέσαμε να προχωρήσουμε την υλοποίηση της παραπάνω ιδέας.

[7] Offline είναι μία εκτέλεση στην οποία δεν μας ενδιαφέρει πρωτίστως το output του προγράμματος αλλά η εξαγωγή στατιστικών και μετρικών επίδοσης.

Η επόμενη σκέψη για το σύστημα μας (την οποία εντέλει και υλοποιήσαμε) είναι η ακόλουθη: απεικονίζουμε την μνήμη της εφαρμογής εξ' αρχής στους κόμβους εκείνους που έχουν διαθέσιμα μεγάλα πλαίσια. Φυσικά πρώτα εξετάζεται ο τοπικός κόμβος και αν το πλήθος ελευθέρων 2M πλαισίων του δεν επαρκεί για την κάλυψη του 100% των απαιτήσεων μνήμης της εφαρμογής, τότε γίνεται αναζήτηση για μεγάλα πλαίσια και στους μακρινούς κόμβους. Με αυτόν τον τρόπο, εξασφαλίζεται ότι χρησιμοποιείται καθολικά το μεγάλο μέγεθος πλαισίου και η μετρική TLB Hit Ratio θα πάρει την μέγιστη τιμή της. Το μειονέκτημα είναι βέβαια ότι κάποιες σελίδες μπορεί να χρειαστεί να σταλούν σε μακρινούς κόμβους και έτσι οι προσβάσεις σε αυτές που αστοχούν στην cache μνήμη θα έχουν πρόσθετο κόστος. Στην πολιτική απεικόνισης μνήμης που μόλις περιγράψαμε δίνεται για λόγους αναφοράς η ονομασία **Remote Huge Pages** (**RHP**)[8]. Ένα παράδειγμα τοποθέτησης μνήμης με βάση αυτή δίνεται στο σχήμα 4.2.



Σχήμα 4.2: Ο τοπικός κόμβος 0 έχει ελεύθερη μνήμη για ολόκληρο τον πίνακα, αλλά μεγάλα πλαίσια μόνο για τον μισό. Συνεπώς, η πολιτική RCM στέλνει δύο μεγάλες σελίδες του πίνακα στην μνήμη του κόμβου 1.

Την πολιτική RHP υλοποιούμε ως μία μικρή βιβλιοθήκη σε γλώσσα C. Η βασική δομή της βιβλιοθήκης είναι αυτή που αποθηκεύει χρήσιμες πληροφορίες για έναν κόμβο του συστήματος:

```
struct node_desc {
        int id;
        int distance_local;
        unsigned long huge_page_bytes;
};
```

Τα πεδία του struct χρησιμεύουν ως εξής:

- id: το μοναδικό αναγνωριστικό που χρησιμοποιεί το λειτουργικό σύστημα για τον κόμβο

[8] Ο πηγαίος κώδικας της πολιτικής RHP μαζί με το profiling σύστημα που περιγράφεται στην ενότητα 5.3, βρίσκονται στη διεύθυνση https://github.com/bpol9/RemoteHugePages

- distance_local: μία αριθμητική τιμή που υποδηλώνει «πόσο μακριά» βρίσκεται ο εν λόγω κόμβος από τον κόμβο εκτέλεσης της διεργασίας. Η τιμή είναι ανάλογη της απόστασης και δίνεται από το λειτουργικό σύστημα
- huge_page_bytes: Η συνολική ποσότητα ελεύθερης μνήμης που έχει ο κόμβος σε μεγάλα πλαίσια

Μία χρήσιμη συνάρτηση που ορίζεται στον κώδικα της βιβλιοθήκης είναι η get_free_huge_page_bytes που υπολογίζει για κάθε κόμβο την ποσότητα μνήμης που μπορεί να αποθηκεύσει με μεγάλα πλαίσια. Για να υπολογίσει την παραπάνω ποσότητα χρησιμοποιεί τις πληροφορίες που παρέχει το αρχείο /proc/buddyinfo του /proc filesystem [12].
Τέλος, η βασικότερη συνάρτηση του συστήματος είναι η malloc_with_huge_pages την οποία πρέπει να χρησιμοποιήσει ο προγραμματιστής στον πηγαίο κώδικα της εφαρμογής του αντί των γνωστών συναρτήσεων εκχώρησης μνήμης (πχ malloc). Η malloc_with_huge_pages αρχικά καλεί την get_free_huge_page_bytes για να πάρει μία πρόσφατη τιμή της χωρητικότητας κάθε κόμβου σε huge pages. Έπειτα καλεί την συνάρτηση posix_mem_align για να προωθήσει το αίτημα μνήμης στο ΛΣ και της επιστρέφεται η αρχική διεύθυνση ενός πίνακα δεδομένων. Έπειτα μέσα σε ένα while loop, κάνει bind συνεχόμενα τμήματα του πίνακα δεδομένων σε κόμβους που έχουν εκείνη τη στιγμή διαθέσιμα huge pages. Η λειτουργία του while loop συνοπτικά δίνεται παρακάτω:

```
// nodes is an array of struct node_desc structs and
// it is sorted according to the distance_local field
bytes_left = request_size;
while (bytes_left > 0 && i < num_numa_nodes) {
  if (nodes[i].huge_page_bytes > 0) {
    bind_len = nodes[i].huge_page_bytes > bytes_left ? bytes_left :
nodes[i].huge_page_bytes;
    nodemask = 1 << nodes[i].id;
    if (mbind(start, bind_len, MPOL_PREFERRED, &nodemask, 8, 0) == 0)
{
      start += bind_len;
      bytes_left -= bind_len;
    }
  }
  ++i;
}
```

Για το binding ενός τμήματος του πίνακα με ένα συγκεκριμένο NUMA κόμβο, χρησιμοποιείται όπως φαίνεται η κλήση συστήματος mbind.

Τέλος, όσον αφορά την εφαρμογή της RHP πολιτικής σε ένα πρόγραμμα, υπάρχουν δύο εναλλακτικές:

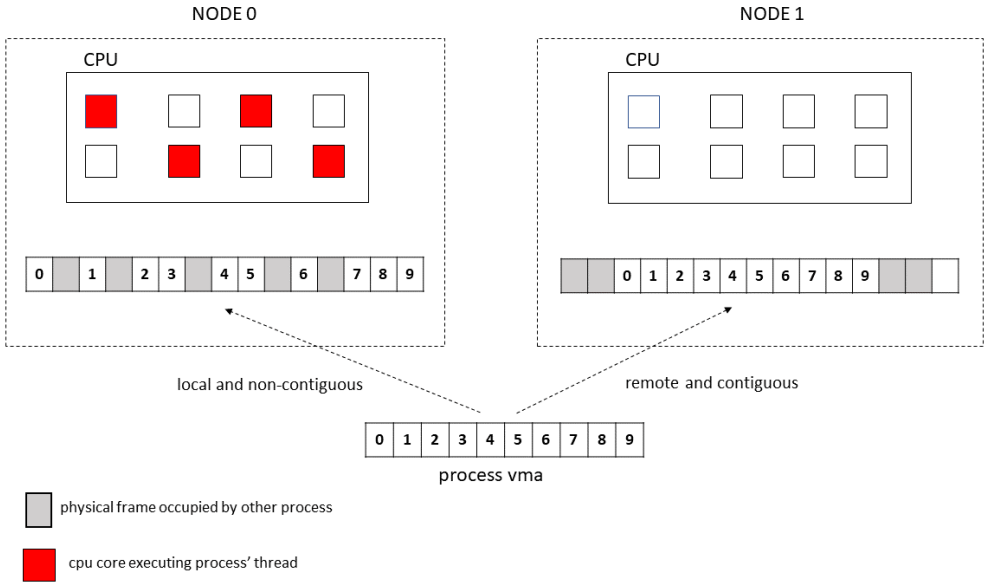
1. Μπορεί ο προγραμματιστής, όπως έχει ήδη αναφερθεί, να αντικαταστήσει απευθείας στον πηγαίο κώδικα του προγράμματος του τις κλήσεις στις συναρτήσεις εκχώρησης μνήμης (πχ malloc) με κλήσεις στην συνάρτηση malloc_with_huge_pages και έπειτα να μεταγλωττίσει εκ νέου το πρόγραμμα.

2. Ο κώδικας RHP μπορεί να μεταγλωττιστεί ως ένα αρχείο μοιραζόμενης βιβλιοθήκης (rhp.so). Θέτοντας την μεταβλητή περιβάλλοντος LD_PRELOAD στην κατάλληλη τιμή ακριβώς πριν την εκτέλεση της εφαρμογής, είναι δυνατόν οι κλήσεις που κάνει το πρόγραμμα στην συνάρτηση malloc (ή άλλη αντίστοιχη) να αντιστοιχιστούν με τον κώδικα της συνάρτησης **malloc_with_huge_pages** του αρχείου rhp.so. Στην περίπτωση αυτή, δεν χρειάζεται να τροποποιήσει ο προγραμματιστής τον κώδικα του προγράμματος του.

Περισσότερες λεπτομέρειες σχετικά με την χρήση της δεύτερης εναλλακτικής υπάρχουν στο [24].

4.2 Remote Contiguous Mappings (RCM)

Το αποτελέσματα της ενότητας «Tradeoff Contiguity vs NUMA Locality» του κεφαλαίου 3, μας έδωσαν κίνητρο να πειραματιστούμε με μία επέκταση του συστήματος CAPaging που θα δίνει προτεραιότητα στην συνεχόμενη απεικόνιση μνήμης (contiguity) σε σχέση με το NUMA locality. Πιο συγκεκριμένα, σε περιπτώσεις που ο τοπικός κόμβος μίας εφαρμογής έχει υποστεί fragmentation, τα ελεύθερα δηλαδή κομμάτια μνήμης του έχουν μικρό μέγεθος, κάποια vmas της διεργασίας θα τοποθετούνται στην μνήμη μακρινού κόμβου με μεγαλύτερα κομμάτια φυσικής μνήμης. Σαν αποτέλεσμα, το συνολικό contiguity που παράγεται θα είναι καλύτερο. Στην επέκταση αυτή δίνουμε για λόγους αναφοράς το όνομα **Remote Contiguous Mappings (RCM)**[9]. Το σχήμα 4.3 εξηγεί με ένα παράδειγμα της λειτουργία της.



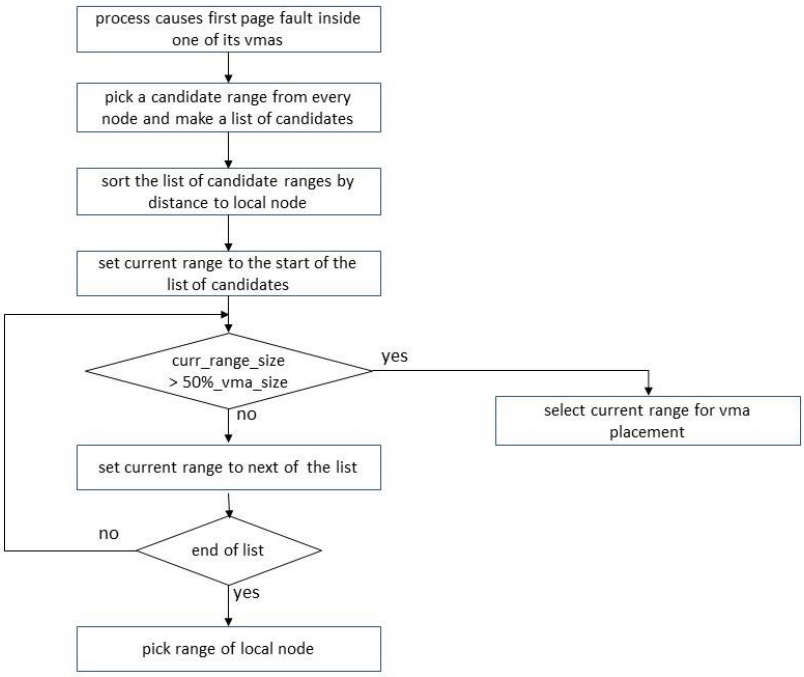
Σχήμα 4.3: Η μνήμη του τοπικού κόμβου 0 έχει υποστεί κατακερματισμό. Ο default CAPaging πυρήνας θα τοποθετήσει το vma στον τοπικό κόμβο, όπου και θα χρειαστεί να σπάσει σε 6 κομμάτια φυσικής μνήμης. Από την άλλη, η επέκταση RCM εξετάζει την μνήμη και του κόμβου 1, όπου βρίσκει ένα συνεχόμενο μεγάλο κομμάτι και αποφασίζει να τοποθετήσει το vma εκεί για επίτευξη υψηλότερου contiguity.

Επιλέξαμε ένα απλό κριτήριο για την επιλογή του physical range όπου θα τοποθετείται ένα vma. Πιο συγκεκριμένα, θέλουμε το range να έχει μέγεθος ίσο με τουλάχιστον το 50% του vma[10]. Φυσικά, πρώτα θα αναζητηθεί ο τοπικός κόμβος για την εύρεση ενός τέτοιου, κι αν αυτό δεν είναι δυνατό, τότε θα εξετάζονται και τα ranges μακρινών κόμβων.

[9] Ο πηγαίος κώδικας της επέκτασης RCM του πυρήνα CAPaging βρίσκεται στην διεύθυνση https://github.com/bpol9/RemoteContiguousMappings

[10] Το 50% είναι μία default τιμή, παραμετροποιήσιμη δυναμικά από τον χρήστη του συστήματος μέσω αρχείου στο /proc filesystem.

Όσον αφορά στην υλοποίηση της επέκτασης RCM μέσα στον CAPaging πυρήνα, εντοπίσαμε μέσα στον κώδικα το σημείο όπου λαμβάνεται η απόφαση τοποθέτησης, δηλαδή επιλέγεται το range φυσικής μνήμης με το οποίο θα αντιστοιχιστεί το vma. Ως είχε ο κώδικας, εξέταζε τα ranges φυσικής μνήμης μόνο του τοπικού κόμβου για να βρει ένα αρκετά μεγάλο ώστε να χωρέσει ολόκληρο το vma. Αλλάξαμε αυτή τη λογική ούτως ώστε να εξετάζονται οι λίστες ranges όλων των κόμβων, τοπικού και απομακρυσμένων, και να επιλέγεται από κάθε κόμβο ένα υποψήφιο range για την τοποθέτηση του vma. Έπειτα, η λίστα των υποψηφίων ταξινομείται ως προς την απόσταση του κόμβου-ιδιοκτήτη του εκάστοτε range από τον τοπικό κόμβο και διατρέχεται από την αρχή της για την εύρεση ενός υποψηφίου που θα ικανοποιεί το κριτήριο επιλογής. Λόγω της ταξινόμησης που αναφέρθηκε, το υποψήφιο range του τοπικού κόμβου εξετάζεται πρώτο. Τα παραπάνω περιγράφονται και στο διάγραμμα ροής του σχήματος 4.4.



Σχήμα 4.4: Η λογική με βάση την οποία η επέκταση RCM επιλέγει το range φυσικής μνήμης όπου θα τοποθετηθεί το vma

4.3 RHP Profiler

Σκοπός του profiler[11] είναι να παρακολουθεί δυναμικά τις διεργασίες που εκτελούνται με την πολιτική απεικόνισης RHP της ενότητας 4.1 και να εκτιμά

1. $gain_{RHP}$ – την μείωση του address translation overhead αποκλειστικά λόγω της RHP πολιτικής
2. $cost_{RHP}$ – το κόστος των remote memory accesses λόγω παράκαμψης του NUMA locality

Αν η ποσότητα 2 ξεπερνά την ποσότητα 1, τότε ο profiler καλώντας την συνάρτηση migrate_pages [13], μαζεύει όλη τη μνήμη της διεργασίας στον τοπικό κόμβο. Διαφορετικά, την αφήνει ως έχει.

Για να υπολογίσει την ποσότητα $gain_{RHP}$ ο profiler, πρέπει να γνωρίζει την ευαισθησία της εφαρμογής στο μέγεθος πλαισίου και η οποία ορίζεται ως

$$S = \frac{T_{4K} - T_{2M}}{T_{2M}}$$

όπου T_{4K} και T_{2M} είναι οι χρόνοι εκτέλεσης της εφαρμογής σε κύκλους όταν χρησιμοποιούνται 4K και 2M πλαίσια αντίστοιχα. Η ευαισθησία S της εφαρμογής είναι γνωστή στον profiler από πριν και έχει υπολογιστεί μέσω δύο offline εκτελέσεων.

Για να υπολογίσει την ποσότητα $cost_{RHP}$, ο profiler χρειάζεται το κόστος κύριας μνήμης που πληρώνει η εφαρμογή όταν **εκτελείται με όλη της τη μνήμη στον τοπικό κόμβο** και που ορίζεται ως

$$M = \frac{Stalls_{ram}}{T_{2M}}$$

όπου $Stalls_{ram}$ είναι τα stalls του επεξεργαστή λόγω αναμονής δεδομένων από την κύρια μνήμη. Το κόστος M της εφαρμογής υπολογίζεται επίσης μέσω offline εκτέλεσης και δίνεται στον profiler ως δεδομένο.

Η λειτουργία του profiler έχει ως εξής: σκανάρονται περιοδικά οι διεργασίες του συστήματος και εκείνες για τις οποίες είναι γνωστές οι ποσότητες S και M (δηλαδή έχουν περάσει το στάδιο των offline εκτελέσεων) επιλέγονται για περαιτέρω παρακολούθηση. Οι υπόλοιπες αγνοούνται.

Οι επιλεγμένες διεργασίες κρατούνται σε μία λίστα μέχρι να ολοκληρώσουν την φάση αρχικοποίησης τους. Μόλις μία διεργασία εισέλθει στην φάση υπολογισμού, γίνονται τα εξής:

1. Υπολογίζεται το ποσοστό μνήμης που έχει απεικονιστεί στον remote κόμβο λόγω RHP και που αν έμενε στον τοπικό θα απεικονιζόταν με 4K πλαίσια (έστω RM)
2. Ενεργοποιούνται οι performance counters στους πυρήνες που εκτελείται η διεργασία για τα events cpu-cycle και ram-stall, όπου ram-stall είναι οι άεργοι κύκλοι του επεξεργαστή λόγω αναμονής δεδομένων από την κύρια μνήμη
3. Η διεργασία αφήνεται να τρέξει για ένα προκαθορισμένο χρονικό διάστημα ώστε οι event counters να πάρουν αντιπροσωπευτικές τιμές
4. Διαβάζονται οι counters για τα events του βήματος 2
5. Υπολογίζεται το κέρδος $gain_{RHP}$ ως

   $$gain_{RHP} = RM * S$$

   όπου RM είναι το ποσοστό απομακρυσμένης μνήμης του βήματος 1 και S είναι η ευαισθησία της εφαρμογής στο μέγεθος σελίδας

[11] Ο πηγαίος κώδικας του RHP profiler βρίσκεται στη διεύθυνση https://github.com/bpol9/RemoteHugePages

6. Υπολογίζεται το κόστος $cost_{RHP}$ ως

$$cost_{RHP} = \frac{ram\_stalls}{cpu\_cycles} - M$$

όπου Mτο κόστος κύριας μνήμης της εφαρμογής όταν εκτελείται με την local πολιτική μνήμης
7. Αν $cost_{RHP} > gain_{RHP}$, τότε η μνήμη της διεργασίας που έχει σταλεί σε remote κόμβους μεταφέρεται στον τοπικό, διαφορετικά αφήνεται ως έχει

4.4 RCM Profiler

Καταρχάς, υποθέτουμε ότι το μηχάνημα διαθέτει την επέκταση υλικού RangeTLB που περιγράφεται στο κεφάλαιο 2.
Σε αυτή την περίπτωση, η λειτουργία του profiler θα είναι να εντοπίζει διεργασίες με remote contiguous memory mappings και να κάνει πάλι εκτίμηση κέρδους και κόστους. Το κέρδος αφορά την μείωση του address translation overhead λόγω κατασκευής μεγαλύτερων mappings και καλύτερης αξιοποίησης του RangeTLB, ενώ το κόστος κατά τα γνωστά είναι το NUMA overhead λόγω remote memory accesses.
Στην περίπτωση αυτή, ο profiler μπορεί να εκτιμήσει το κέρδος και το κόστος με χρήση μετρητών που μπορεί να διαβάσει κατά την εκτέλεση της εφαρμογής σε online χρόνο και συνεπώς, μπορούμε να παραλείψουμε το στάδιο των offline εκτελέσεων. Τονίζεται ωστόσο ότι επειδή πρόκειται για έναν υποθετικό profiler που δεν μπορεί να υλοποιηθεί (λόγω της απουσίας RangeTLB στο υλικό), δεν μπορούμε να ελέγξουμε την αποτελεσματικότητα του.
Πιο συγκεκριμένα όσον αφορά την λειτουργία του, για κάθε διεργασία που παρακολουθείται γίνονται τα εξής:

1. Αναμονή μέχρι η διεργασία να τελειώσει την φάση αρχικοποίησης και να εισέλθει στην φάση υπολογισμού
2. Υπολογίζεται το ποσοστό μνήμης της διεργασίας που έχει απεικονιστεί με remote contiguous memory mappings (έστω RM)
3. Ενεργοποιούνται στους πυρήνες που εκτελείται η διεργασία οι performance counters για τα events range-TLB-hit και remote-memory-access
4. Μετά από ένα προκαθορισμένο χρονικό διάστημα διαβάζονται οι τιμές των counters
5. Υπολογίζεται το κέρδος σε κύκλους ως

$$gain_{RCM} = H_{RTLB} * RM * AvgC_{THP}$$

όπου

- H_{RTLB} είναι το πλήθος των RangeTLB hits
- $AvgC_{THP}$ είναι ο μέσος χρόνος διάσχιση των page tables σε κύκλους
- RM είναι το ποσοστό μνήμης που έχει απεικονιστεί με remote contiguous memory mappings και που υπολογίστηκε στο βήμα 2

6. Υπολογίζεται το NUMA κόστος σε κύκλους ως

$$cost_{RCM} = r * (T_r - T_l) * Freq$$

όπου

- r είναι το πλήθος των remote memory accesses
- T_r και T_l είναι ο χρόνος πρόσβασης στην κύρια μνήμη απομακρυσμένου και τοπικού κόμβου αντίστοιχα
- $Freq$ είναι η συχνότητα του επεξεργαστή

7. Αν $cost_{RCM} > gain_{RCM}$, όλα τα contiguous remote memory mappings της διεργασίας σπάνε και μεταφέρονται στον τοπικό κόμβο, διαφορετικά αφήνονται ως έχουν

Όσον αφορά τον υπολογισμό του κέρδους $gain_{RCM}$, προσπαθούμε μέσω της ποσότητας $H_{RTLB} * RM$ να υπολογίσουμε το πλήθος των RangeTLB hits που οφείλονται αποκλειστικά στην επέκταση RCM και που δεν θα συνέβαιναν αν η διεργασία εκτελούνταν πάνω από τον αρχικό CAPaging πυρήνα. Πολλαπλασιάζοντας μετά με το κόστος ενός RangeTLB miss προκύπτει μία εκτίμηση για το πλήθος των κύκλων μετάφρασης που κατάφερε να κρύψει η επέκταση RCM.
Όσον αφορά στον υπολογισμό του κόστους $cost_{RCM}$, μέσω της ποσότητας $r * (T_r - T_l)$ υπολογίζεται ο επιπλέον χρόνος σε δευτερόλεπτα που ξοδεύει η διεργασία για αναμονή δεδομένων από την κύρια μνήμη και μετά γίνεται ο πολλαπλασιασμός με την συχνότητα του επεξεργαστή $Freq$ για να μετατραπεί το κόστος σε κύκλους.

Κεφάλαιο 5 Αξιολόγηση

5.1 Περιγραφή μηχανήματος, benchmarks και εργαλείων αξιολόγησης

Στο κεφάλαιο αυτό αξιολογούμε την σύστημα χώρου χρήστη RHP και την επέκταση πυρήνα CAPaging RCM του που παρουσιάστηκαν στο κεφάλαιο 4. Για την δοκιμή των υλοποιήσεων χρησιμοποιήθηκε το παρακάτω σύνολο από benchmarks:

1. XSBench [14] – Μία εφαρμογή που απομονώνει και υλοποιεί έναν σημαντικό υπολογιστικό πυρήνα της μεθόδου Monte Carlo για την εξίσωση μεταφοράς νετρονίων. Αποτελεί χρήσιμο εργαλείο για την ανάλυση επίδοσης σε High Performance αρχιτεκτονικές.
2. Liblinear-train [15] – Μία βιβλιοθήκη ανοικτού κώδικα που πραγματοποιεί γραμμική ταξινόμηση (linear classification) σε σύνολα δεδομένων μεγάλης κλίμακας και υποστηρίζει τις μεθόδους linear regression και support-vector machine (SVM).
3. Hashjoin –Ένα microbenchmark που πραγματοποιεί προσβάσεις σε έναν μεγάλο πίνακα δεδομένων με random access pattern.
4. STEAM [16] – Ένα benchmark για την μέτρηση του εύρους ζώνης κύριας μνήμης.

Την εκτέλεση των benchmarks πραγματοποιούμε στο μηχάνημα broady2 του εργαστηρίου CSLab, μικροαρχιτεκτονικής intel Broadwell και με τα εξής κύρια χαρακτηριστικά:

- 2 NUMA κόμβους με 10 φυσικούς πυρήνες και 128 GB μνήμης ανά κόμβο
- Κάθε πυρήνας έχει δυνατότητες Simultaneous Multithreading (SMT), δηλαδή μπορεί να εκτελεί ταυτόχρονα δύο νήματα εντολών[12]
- Μία L3 Cache μνήμη 25 MB ανά κόμβο που είναι κοινή για όλους τους πυρήνες του κόμβου
- Μία ιδιωτική L2 Cache μνήμη 256KB και μία ιδιωτική L1 Cache μνήμη 32KB ανά φυσικό πυρήνα
- 1 L1 Instruction TLB για 4K σελίδες με 64 entries, 1 L1 Data TLB για 4K σελίδες με 64 entries, 1 L1 Data TLB για 2M/4M σελίδες με 32 entries, 1 L1 Data TLB για 1 GB σελίδες με 4 entries και 1 L2 TLB για 4K/2M σελίδες με 1536 entries ανά φυσικό πυρήνα
- Ένα remote memory access χρειάζεται περίπου διπλάσιο χρόνο σε σχέση με ένα local

[12] Στη αξιολόγηση δεν χρησιμοποιούμε αυτή τη δυνατότητα, αναθέτοντας ένα μόνο νήμα σε κάθε φυσικό πυρήνα

Η εκτέλεση των benchmarks έγινε σε ένα NUMA εικονικό μηχάνημα καθ' εικόνα και ομοίωση του πραγματικού, για τη δημιουργία του οποίο χρησιμοποιήθηκε το QEMU [17]. Για τον προσδιορισμό της επίδοσης χρησιμοποιήθηκαν οι hardware performance counters του μηχανήματος και το Linux perf tool [18]. Χρησιμοποιήσαμε επίσης ένα custom fragmentation tool που πραγματοποιεί κατακερματισμό κύριας μνήμης είτε σε επίπεδο huge pages είτε σε επίπεδο larger-than-a-frame ranges και που δημιουργεί τα σενάρια fragmentation τοπικού κόμβου που στοχεύουν να αντιμετωπίσουν οι υλοποιήσεις μας. Τέλος, χρησιμοποιήθηκε το εργαλείο γραμμής εντολών numactl [3] για την τοποθέτηση των εφαρμογών στους επιθυμητούς κόμβους.

5.2 Αξιολόγηση του συστήματος Remote Huge Pages (RHP)

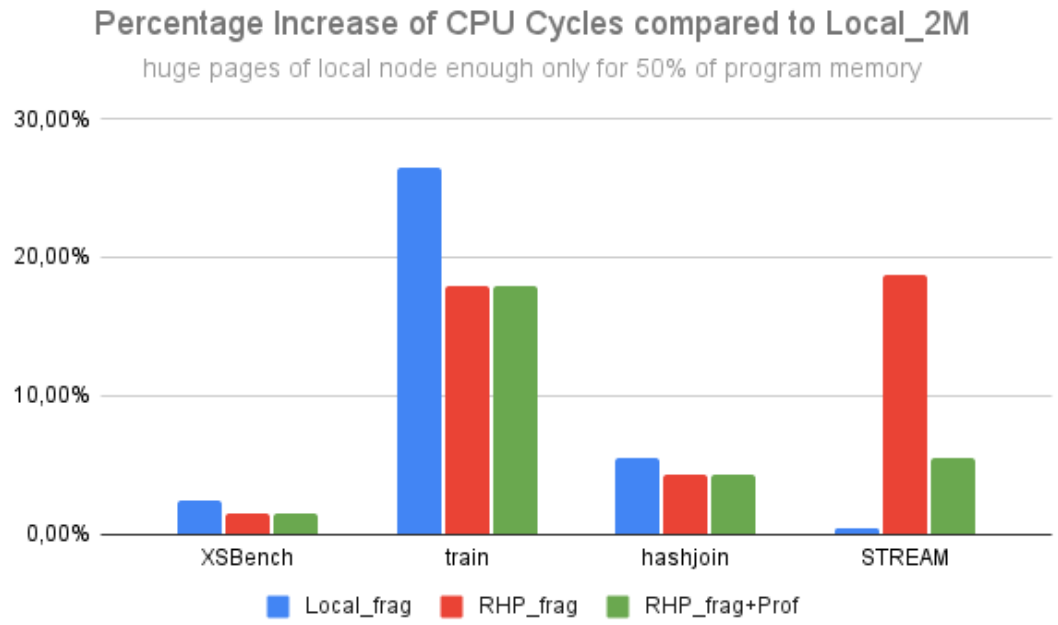
Εκτελούμε κάθε benchmark 4 φορές στο εικονικό μηχάνημα όπως περιγράφεται παρακάτω:

1. **Local_2M**: όλη η μνήμη του benchmark αποθηκεύεται στον τοπικό κόμβο και με 2MB πλαίσια.
2. **Local_frag**: πριν την εκτέλεση του benchmark και με χρήση του fragmentation tool, αφήνουμε στον τοπικό κόμβο τόσα ελεύθερα μεγάλα πλαίσια όσα φτάνουν για το μισό από τις απαιτήσεις μνήμης του benchmark. Στον κόμβο υπάρχει αρκετή μνήμη και για το υπόλοιπο μισό, αλλά αποτελείται από μικρά πλαίσια. Έτσι, όλη η μνήμη απεικονίζεται στον τοπικό κόμβο, η μισή με μεγάλα και η άλλη μισή με μικρά πλαίσια.
3. **RHP_frag**: χρησιμοποιούμε το fragmentation tool πριν την εκτέλεση του benchmark ακριβώς με τον ίδιο τρόπο με την περίπτωση Local_frag και παράλληλα χρησιμοποιούμε το σύστημα RHP για την απεικόνιση της μνήμης. Αυτό έχει σαν αποτέλεσμα 1) η μνήμη να απεικονίζεται αποκλειστικά με 2M πλαίσια και 2) οι σελίδες που δέχονται την πρώτη αναφορά αρκετά νωρίς να μένουν στον τοπικό κόμβο, ενώ οι υπόλοιπες να στέλνονται στον μακρινό.
4. **RHP_frag+Prof**: ίδια ακριβώς εκτέλεση με την RHP_frag, με την διαφορά ότι αυτή τη φορά ο RHP profiler που περιγράφεται στην ενότητα 4.3 είναι ενεργός και παρακολουθεί την εκτέλεση του benchmark.

Για κάθε διαφορετικό σενάριο μετράμε με χρήση του Linux perf tool

1. τους συνολικούς κύκλους εκτέλεσης
2. τα stalls λόγω L2 Cache miss
3. τα page walk cyles

και υπολογίζουμε την ποσοστιαία αύξηση της κάθε μετρικής σε σχέση με την Local_2M εκτέλεση. Τα αποτελέσματα φαίνονται στο σχήμα 5.1.



Σχήμα 5.1: Η αύξηση επί των ιδανικών κύκλων εκτέλεσης που προκαλούν οι δύο πολιτικές μνήμης (Local και RHP) όταν ο κόμβος εκτέλεσης τους προγράμματος έχει υποστεί external fragmentation.

Ξεκινώντας από το **XSBench**, βλέπουμε κατ' αρχάς ότι οι μπάρες RHP_frag και RHP_frag+Prof είναι πανομοιότυπες και αυτό γιατί ο profiler αποφασίζει να αφήσει την μνήμη με την αρχική της τοποθέτηση, οπότε πρόκειται ουσιαστικά για τις ίδιες εκτελέσεις. Επιπλέον, παρ΄ότι ο τοπικός κόμβος είναι κατακερματισμένος, παρατηρούμε πολύ μικρή αύξηση των κύκλων εκτέλεσης και με τις δύο πολιτικές (Local και RHP). Ως εξήγηση, το XSBench έχει ένα μόνο hot κομμάτι μνήμης που αρχικοποιείται από τα πρώτα, οπότε στην μεν Local_frag εκτέλεση προλαβαίνει να απεικονιστεί με μεγάλα πλαίσια, στην δε RHP_frag προλαβαίνει μεγάλα πλαίσια του τοπικού κόμβου και δεν χρειάζεται να σταλεί στον μακρινό. Συνεπώς, στην πρώτη περίπτωση αποφεύγονται τα αυξημένα TLB misses και στην δεύτερη τα remote memory accesses. Τα υπόλοιπα κομμάτια μνήμης δεν φαίνεται να επηρεάζουν ιδιαίτερα την επίδοση, ανεξάρτητα αν απεικονίζονται με μικρά πλαίσια ή στον μακρινό κόμβο, εξ' ου και η αμελητέα αύξηση των κύκλων εκτέλεσης. Καταλήγοντας, παρατηρούμε μία μικρή βελτίωση που πετυχαίνει η πολιτική RHP, που πιθανώς να οφείλεται και στον μειωμένο αριθμό page faults λόγω 100% απεικόνισης της μνήμης με μεγάλα πλαίσια.

Σχετικά με το **Hashjoin**, το computation phase του αποτελείται από διαδοχικά accesses σε δύο arrays, έστω A και B. Το array A διασχίζεται σειριακά και οι τιμές του δίνονται ως είσοδος σε μία συνάρτηση hashing. Το hash που προκύπτει χρησιμοποιείται ως δείκτης στο array B. Το κύριο loop εκτέλεσης δηλαδή είναι κάπως έτσι:

```
acc = 0;
for (i=0; i<iters_num; i++) {
   for (j=0; j<A.length; j++) {
      hash = hash_fun(A[j]);
      acc += B[hash];
   }
}
```

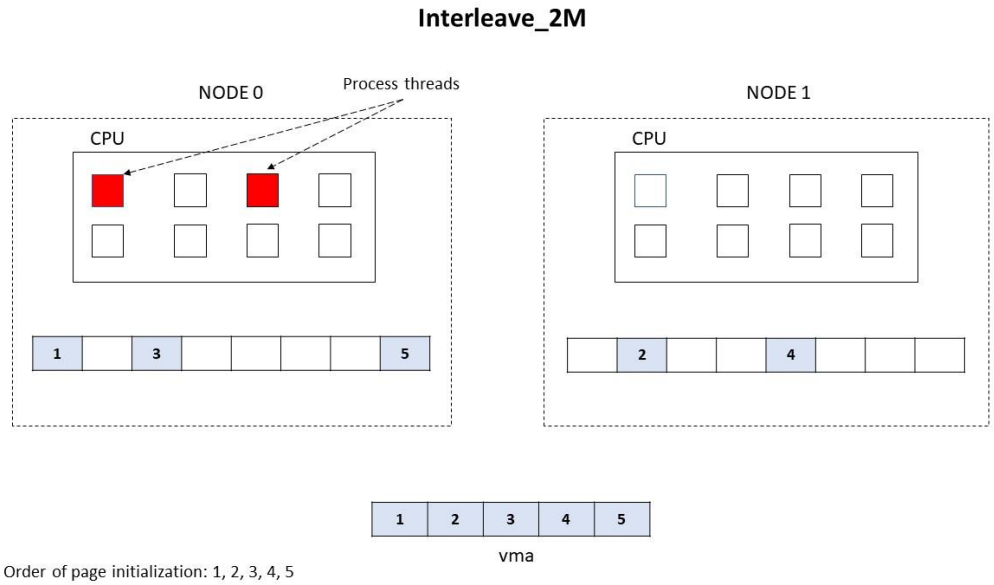
Ο πίνακας A δηλαδή δέχεται προσβάσεις με σειριακό access pattern και ο B με random access pattern. Μπορούμε να πούμε ότι ο B είναι το hot κομμάτι μνήμης του προγράμματος υπό την έννοια ότι προκαλεί τα περισσότερα cache και TLB misses. Ο πίνακας B επίσης τυχαίνει να αρχικοποιείται πρώτος μέσα στο πρόγραμμα, και για τους ίδιους λόγους που αναφέρονται και στην περίπτωση του XSBench, απεικονίζεται στον τοπικό κόμβο και με μεγάλα πλαίσια ανεξαρτήτως πολιτικής (Local ή RHP). Γι' αυτό και πάλι παρατηρείται μικρή αύξηση των κύκλων, παρά τον περιορισμένο αριθμό μεγάλων πλαισίων στον τοπικό κόμβο.

Η κατάσταση αλλάζει με το **Liblinear-train**. Το hot κομμάτι μνήμης είναι από τα τελευταία που αρχικοποιούνται. Γι' αυτό το λόγο στην Local_frag εκτέλεση απεικονίζεται με μικρά πλαίσια (αφού τα μεγάλα του τοπικού κόμβου έχουν ήδη δεσμευθεί όλα), προκαλώντας αύξηση του πλήθους αλλά και του κόστους των TLB misses. Από την άλλη, στην RHP_frag εκτέλεση το hot κομμάτι δεν προλαβαίνει τα μεγάλα πλαίσια του τοπικού κόμβου και στέλνεται στον μακρινό, εισάγοντας όμως έτσι remote memory accesses.

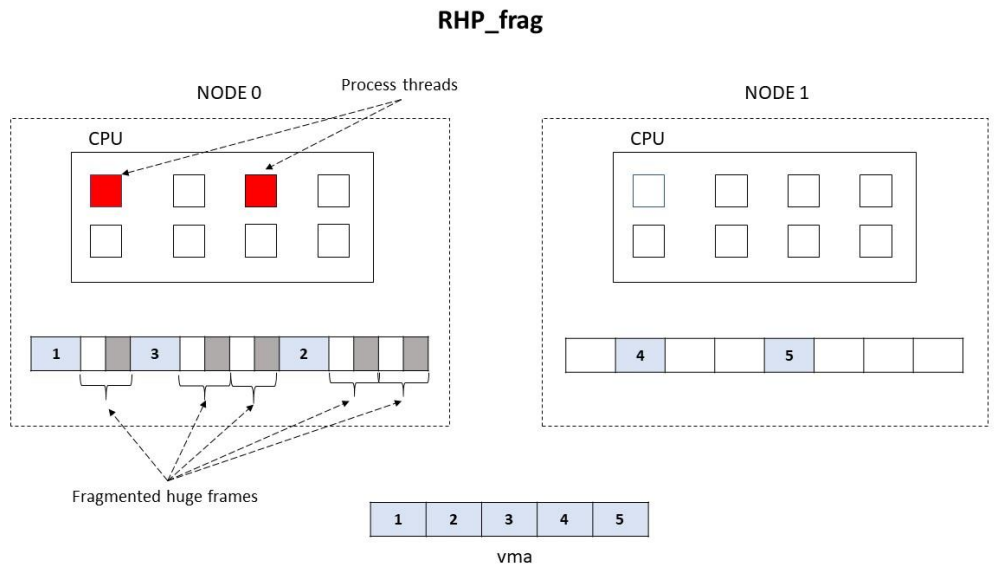
Παρ' ότι και στις δύο εκτελέσεις υπάρχει αύξηση των συνολικών κύκλων εκτέλεσης σε σχέση με την ιδανική Local_2M εκτέλεση, βλέπουμε ότι η Local_frag εισάγει overhead 26.53%, ενώ η RHP_frag 17.89%. Με άλλα λόγια, όταν η μνήμη του τοπικού κόμβου δεν έχει αρκετά huge pages, η RHP πολιτική μειώνει τον χρόνο εκτέλεσης κατά 7% σε σχέση με την default Local πολιτική.

Τέλος, όσον αφορά στο **STREAM**, χρησιμοποιεί ένα streaming access pattern το οποίο δεν κάνει επαναχρησιμοποίηση δεδομένων και οδηγεί σε μη αμελητέο cache miss ratio, ωστόσο το pattern αυτό ευνοεί πολύ υψηλό TLB hit ratio ακόμα και με 4K πλαίσια. Συνεπώς, είναι αναμενόμενο να συμβαίνει αύξηση στην RHP_frag εκτέλεση λόγω των remote memory accesses αλλά όχι στην Local_frag αφού το μέγεθος πλαισίου δεν επηρεάζει το ποσοστό ευστοχίας TLB. Το σημαντικό σε αυτή την περίπτωση είναι ότι ο profiler, βασιζόμενος και στις πληροφορίες που του δίνεται από τις offline εκτελέσεις, καταφέρνει να διαπιστώσει ότι η RHP πολιτική δεν είναι επωφελής για την συγκεκριμένη εφαρμογή και μεταφέρει δυναμικά την απομακρυσμένη μνήμη στον τοπικό κόμβο, ανακτώντας το μεγαλύτερο κομμάτι της χαμένης επίδοσης.

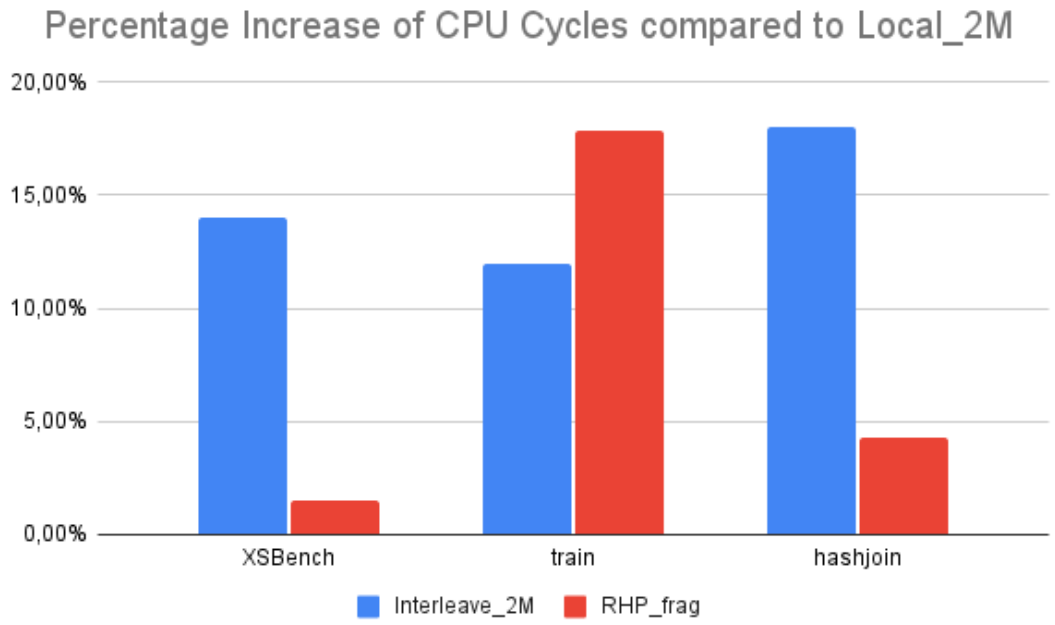
Για να στηρίξουμε την υπόθεση που κάναμε για το hot κομμάτι μνήμης των προγραμμάτων XSBench, Hashjoin και Liblinear-train, πραγματοποιούμε μία νέα εκτέλεση, την **Interleave_2M,** κατά την οποία εφαρμόζεται η Interleave πολιτική μνήμης και χρησιμοποιούνται παντού 2M πλαίσια για την απεικόνιση. Στη συνέχεια συγκρίνουμε τις εκτελέσεις RHP_frag και Interleave_2M, όπως φαίνεται στο σχήμα 5.4. Τονίζεται ότι αν και στις δύο περιπτώσεις στέλνεται το 50% της μνήμης στον remote κόμβο, διαφέρει κάθε φορά το ποιες σελίδες στέλνονται εκεί. Το σχήματα 5.2 και 5.3 δίνουν ένα παράδειγμα.



Σχήμα 5.2: Ο τρόπος απεικόνισης ενός πίνακα όταν εφαρμόζεται η Interleave πολιτική.



Σχήμα 5.3: Ο τρόπος απεικόνισης ενός πίνακα όταν εφαρμόζεται η RHP πολιτική και τα ελεύθερα huge pages του τοπικού κόμβου δεν αρκούν για όλο τον πίνακα. Η RHP απεικονίζει σελίδες του vma τοπικ΄και μόλις τα μεγάλα πλαίσια εξαντληθούν, οι υπόλοιπες σελίδες στέλνονται στον μακρινό κόμβο.



Σχήμα 5.4: Αύξηση των κύκλων επεξεργασίας ως προς την ιδανική Local_2M εκτέλεση

Κοιτώντας το διάγραμμα 5.4, οσον αφορά στο XSBench, στην RHP_frag εκτέλεση το hot κομμάτι όπως προαναφέρθηκε αποθηκεύεται όλο στον τοπικό κόμβο, γιατί αρχικοποιείται νωρίτερα σε σχέση με τα υπόλοιπα και προλαβαίνει τα local μεγάλα πλαίσια. Από την άλλη, στην Interleave_2M εκτέλεση το hot κομμάτι μνήμης «σπάει» εξ' ημισείας στους δύο κόμβους με αποτέλεσμα να γίνονται περισσότερα remote accesses, γι' αυτό και η Interleave_2M εκτέλεση προκαλεί μεγαλύτερη αύξηση των κύκλων CPU. Ίδια ακριβώς είναι και η περίπτωση του Hashjoin. Αντίθετα στο Liblinear-train, που το hot κομμάτι μνήμης αρχικοποιείται τελευταίο, κατά την RHP_frag εκτέλεση στέλνεται όλο στον μακρινό κόμβο (αφού δεν προλαβαίνει μεγάλα πλαίσια τοπικού κόμβου), ενώ στην Interleave_2M μόνο το μισό. Γι' αυτό και η RHP_frag έχει μεγαλύτερο χρόνο εκτέλεσης.

Καταλήγοντας, βλέπουμε ότι όταν ο τοπικός κόμβος έχει μεν ικανή ποσότητα ελεύθερης μνήμης αλλά περιορισμένο αριθμό μεγάλων πλαισίων, η πλειοψηφία των εφαρμογών έχει καλύτερη επίδοση με την RHP πολιτική σε σύγκριση με την Local, ειδικότερα όπως είδαμε όταν το hot κομμάτι μνήμης είναι από τα τελευταία που αρχικοποιούνται. Στις σπάνιες περιπτώσεις εφαρμογών, όπως το STREAM, που δεν έχουν ευαισθησία στο μέγεθος πλαισίου, ο profiler καταφέρνει να το αναγνωρίσει και να αποφύγει το overhead των remote memory accesses, μαζεύοντας εγκαίρως όλη τη μνήμη στον τοπικό κόμβο.

5.3 Αξιολόγηση της επέκτασης Remote Contiguous Mappings (RCM)

Αρχικά αναφέρεται ότι για να μπορέσουμε να κάνουμε αξιολόγηση της επέκτασης RCM, θεωρούμε ότι το υλικό υποστηρίζει τις επεκτάσεις RMM που περιγράφονται στο κεφάλαιο 2. Υπάρχει δηλαδή, μεταξύ άλλων, ένα RangeTLB που δίνει μεταφράσεις βασιζόμενο στα larger-than-a-page mappings που φτιάχνει ο πυρήνας και μειώνει το address translation overhead.

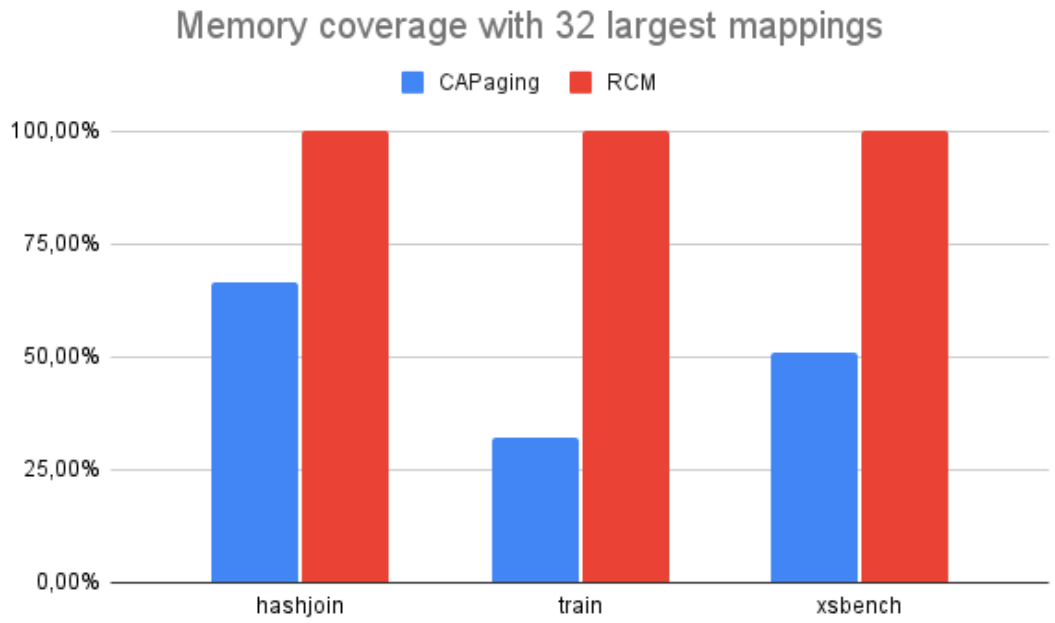
Αφήνουμε εκτός το πρόγραμμα STREAM, το οποίο όπως είδαμε έχει πολύ μικρό address translation overhead ακόμα και με 4K πλαίσια. Τα υπόλοιπα benchmarks που περιγράφονται στην αρχή του κεφαλαίου, εκτελούμε σε εικονικό μηχάνημα, 3 φορές το καθένα όπως περιγράφεται παρακάτω:

- ***THP*** εκτέλεση: όλη η μνήμη απεικονίζεται με 2M πλαίσια και στον τοπικό κόμβο (ισοδύναμη με τις Local_2M εκτελέσεις που έχουν αναφερθεί παραπάνω). Η εκτέλεση γίνεται πάνω από τον πυρήνα Linux, χωρίς την οποιαδήποτε επέκταση CAPaging
- ***CAPaging*** εκτέλεση: πριν την εκτέλεση του benchmark, προκαλείται κατακερματισμός στον τοπικό κόμβο με τέτοιο τρόπο, ώστε να υπάρχουν λίγα (όχι περισσότερα από 5), μεγάλα κομμάτια φυσικής μνήμης, η συνολική χωρητικότητα των οποίων αρκεί για το 50% των απαιτήσεων μνήμης του προγράμματος. Όλα τα υπόλοιπα κομμάτια της τοπικής μνήμης έχουν πολύ μικρό μέγεθος. Η εκτέλεση γίνεται πάνω από τον πυρήνα CAPaging
- ***RCM*** εκτέλεση: ο τοπικός κόμβος έχει υποστεί ίδιο ακριβώς fragmentation με την εκτέλεση CAPaging. Ωστόσο, αυτή τη φορά το πρόγραμμα εκτελείται πάνω από τον πυρήνα RCM, την επέκταση δηλαδή που υλοποιήσαμε και που χρησιμοποιεί κομμάτια μνήμης και του μακρινού κόμβου για την τοποθέτηση των vmas

Μία σημαντική ποσότητα που χρησιμοποιούμε ως βάση (baseline) για την αξιολόγηση είναι οι ιδανικοί κύκλοι εκτέλεσης (ideal cpu cycles), που ορίζονται ως οι κύκλοι επεξεργασίας που θα χρειαζόταν το πρόγραμμα για να ολοκληρωθεί αν είχε **μηδενικό κόστος μετάφρασης** διευθύνσεων (100% TLB hit ratio) και **100% local memory accesses**. Υπολογίζονται ως

$$\mathrm{T_{ideal}} = \mathrm{CpuCycles}_{THP} - \mathrm{PageWalkCycles}_{THP}$$

Για τις εκτελέσεις ***CAPaging*** και ***RCM***, μετράμε με το εργαλείο [20] το ποσοστό κάλυψης μνήμης που επιτυγχάνεται με τα 32 μεγαλύτερα mappings. Χρησιμοποιούμε τον αριθμό 32 γιατί τόσα είναι και τα entries του RangeTLB. Τα αποτελέσματα φαίνονται σχήμα 5.5.



Σχήμα 5.5: Το ποσοστό κάλυψης μνήμης με τα 32 μεγαλύτερα mappings για τους πυρήνες CAPaging και RCM. Η μνήμη του τοπικού κόμβου είχε υποστεί εξωτερικό κατακερματισμό πριν την εκτέλεση του κάθε benchmark

Αυτό που παρατηρούμε από το διάγραμμα είναι ότι ο κατακερματισμός της μνήμης τοπικού κόμβου μείωσε όπως αναμενόταν το ποσοστό κάλυψης των 32 largest mappings που πετυχαίνει ο CAPaging πυρήνας (μπλε μπάρα), ενώ η επέκταση RCM κατάφερε να το επαναφέρει χρησιμοποιώντας τα μεγάλα ranges και του μακρινού κόμβου (κόκκινη μπάρα).
Σημειώνεται ότι τα παραπάνω ποσοστά κάλυψης μνήμης αφορούν απεικονίσεις από τον guest virtual στον guest physical χώρο διευθύνσεων. Για να είναι πιο ακριβής η αξιολόγηση, θα θέλαμε να έχουμε την κάλυψη που πετυχαίνουν τα 32 μεγαλύτερα mappings από τον guest εικονικό στον host φυσικό χώρο διευθύνσεων. Για να το μετρήσουμε αυτό θα έπρεπε να εκτελεστεί ο πυρήνας CAPaging και στο host μηχάνημα, το οποίο δεν ήταν δυνατό. Θεωρώντας ότι ο host CAPaging πυρήνας θα απεικόνιζε συνεχόμενα την guest φυσική μνήμη στην host φυσική, κάνουμε την υπόθεση ότι οι τιμές memory coverage που πήραμε δεν απέχουν πολύ από τις πραγματικές και τις χρησιμοποιούμε για την αξιολόγηση της επίδοσης παρακάτω.

Περνώντας στο κομμάτι της αξιολόγησης, για κάθε σενάριο εκτέλεσης από τα 3 που προαναφέρθηκαν, θέλουμε να βρούμε τους επιπλέον κύκλους που εισάγουν πέραν των ιδανικών που ορίστηκαν παραπάνω.
Για την ***THP*** εκτέλεση που έχει 100% τοπικές προσβάσεις, το overhead επί των ιδανικών κύκλων οφείλεται αποκλειστικά στην μετάφραση διευθύνσεων και προκύπτει ως

$$O_{THP} = \frac{pwc}{T_{ideal}}$$

όπου pwc είναι οι συνολικοί κύκλοι διάσχισης των page tables όπως προκύπτουν από τους performance counters.

Για τις εκτελέσεις με τον ***CAPaging*** πυρήνα και την ***RCM*** επέκταση, θέλουμε με κάποιο τρόπο να βρούμε πόσοι θα ήταν οι κύκλοι μετάφρασης διευθύνσεων αν υπήρχε η επέκταση RangeTLB που θα αξιοποιούσε το contiguity που κατασκευάζεται. Το ιδανικό θα ήταν να το κάνουμε με τον τρόπο που γίνεται και στην εργασία [10], όπου χρησιμοποιείται το σύστημα BadgerTrap [19] για instrumentation των tlb misses και προσομοίωση του RangeTLB σε λογισμικό. Προσομοιώνονται δηλαδή τα RangeTLB accesses όταν συμβαίνει ένα TLB miss και μπορεί να υπολογιστεί ο ακριβής αριθμός των RangeTLB misses. Ο τύπος υπολογισμού των κύκλων μετάφρασης είναι δηλαδή

$$\mathrm{T_{cycles}} = \mathrm{M_{SIM}} * \mathrm{AvgC_{THP}}$$

όπου

- $\mathrm{M_{SIM}}$ είναι ο αριθμός των των TLB misses που δεν κατάφερε να κρύψει το RangeTLB, δηλαδή χονδρικά ο αριθμός RangeTLB misses
- $\mathrm{AvgC_{THP}}$ είναι ο μέσος χρόνος διάσχισης των page tables σε κύκλους όταν χρησιμοποιούνται μεγάλα πλαίσια

Ο παραπάνω ιδανικός τρόπος μέτρησης των κύκλων μετάφρασης δεν κατέστη δυνατός για τεχνικούς λόγους. Συνεπώς, κάνουμε την απλουστευμένη υπόθεση ότι το ποσοστό ευστοχίας του RangeTLB είναι ίσο με το ποσοστό κάλυψης μνήμης των 32 largest mappings. Έτσι, υπολογίζουμε τους κύκλους μετάφρασης ως

$$\mathrm{T_{cycles}} = (1 - \mathrm{cov_{32}}) * \mathrm{M_{TLB}} * \mathrm{AvgC_{THP}}$$

όπου

- cov_{32} είναι το ποσοστό κάλυψης μνήμης που επιτυγχάνεται με τα 32 largest mappings
- M_{TLB} είναι ο αριθμός των L2 TLB misses
- $AvgC_{THP}$ είναι ο μέσος χρόνος διάσχισης των page tables σε κύκλους και για 2M σελίδες

Ο CApaging πυρήνας δεν στέλνει μνήμη στον μακρινό κόμβο και συνεπώς δεν δημιουργεί καθόλου remote memory accesses. Συνεπώς, η ***CAPaging*** εκτέλεση έχει overhead επί των ιδανικών κύκλων

$$\mathrm{O_{CAPaging}} = \frac{\left((1-\mathrm{cov_{32}}) * \mathrm{M_{TLB}}\right) * \mathrm{AvgC_{THP}}}{\mathrm{T_{ideal}}}$$

Τονίζεται εδώ ότι λόγω local node memory fragmentation η τιμή του cov_32 στον παραπάνω τύπο θα παρεκκλίνει από την ιδανική τιμή 1 και συνεπώς για αρκετά TLB misses θα χρειασθεί page table walk για την μετάφραση. Αυτό είναι και το πρόβλημα εξάλλου που προσπαθεί να λύσει η επέκταση RCM.

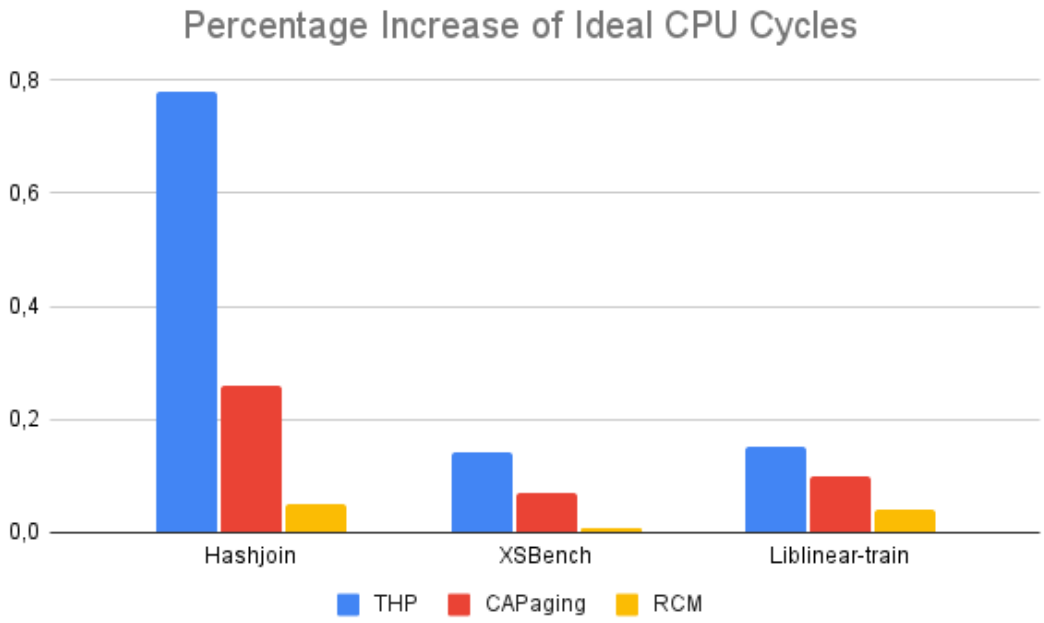
Για τον υπολογισμό overhead της εκτέλεσης ***RCM*** επί των ιδανικών κύκλων, πρέπει να συμπεριλάβουμε και το κόστος των remote memory accesses που προκύπτουν από την τοποθέτηση vmas στην μνήμη και του μακρινού κόμβου.

$$O_{RCM} = \frac{((1-\mathrm{cov}_{32}) * \mathrm{M}_{\mathrm{TLB}}) * \mathrm{AvgC}_{\mathrm{THP}} + T_{remote}}{\mathrm{T}_{\mathrm{ideal}}}$$

όπου

- T_{remote} είναι τα επιπλέον execution stalls λόγω remote memory accesses

Με βάση τους τύπους υπολογισμού O_{THP}, $O_{CAPaging}$ και O_{RCM}, η ποσοστιαία αύξηση επί των ιδανικών κύκλων εκτέλεσης για κάθε εκτέλεση φαίνεται στο σχήμα 5.6.



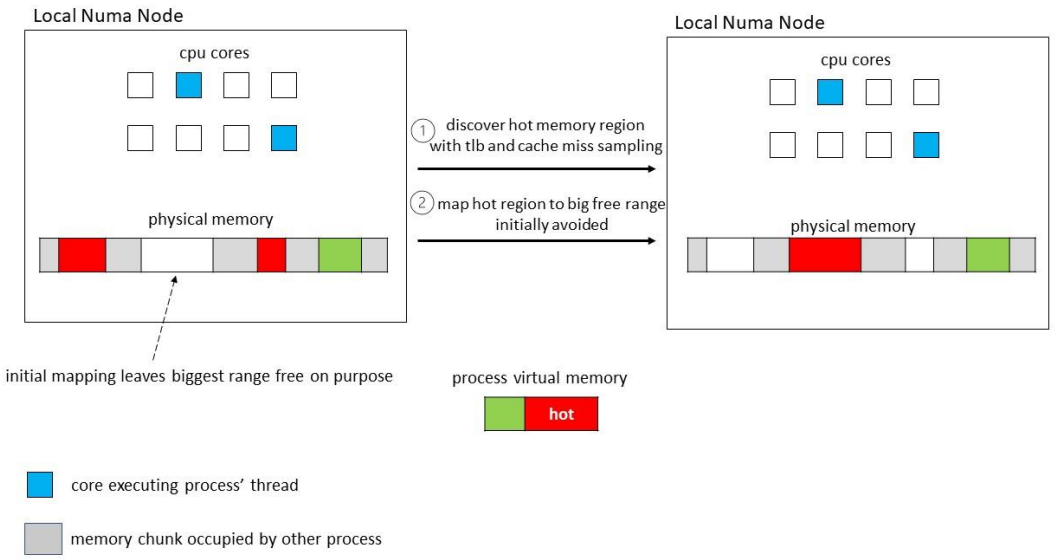
Σχήμα 5.6: Η ποσοστιαία αύξηση επί των ιδανικών κύκλων εκτέλεσης όταν το πρόγραμμα εκτελείται με 3 διαφορετικούς πυρήνες (Linux, CAPaging, επέκταση RCM). Η μνήμη του τοπικού κόμβου είχε υποστεί external fragmentation πριν την κάθε εκτέλεση.

Η ***THP*** εκτέλεση (μπλε μπάρα) αυξάνει την ποσότητα T_{ideal} κατά το μεγαλύτερο ποσοστό σε όλες τις περιπτώσεις, αφού ο default Linux πυρήνας δεν δημιουργεί larger-than-a-page mappings και το RangeTLB δεν αξιοποιείται. Σαν αποτέλεσμα, για όλα τα TLB misses χρειάζεται να γίνει διάσχιση των page tables. Ειδικά στο Hashjoin που έχει ένα random access pattern και συμβαίνουν πολλά TLB misses το ποσοστό αύξησης του T_{ideal} πλησιάζει το 80%.
Η αύξηση που προκαλεί η εκτέλεση με τον ***CAPaging*** πυρήνα (κόκκινη μπάρα) αντιστοιχεί στα TLB misses που δεν κατάφερε να κρύψει το RangeTLB λόγω μέτριου contiguity. Πιο συγκεκριμένα, λόγω κατακερματισμού του τοπικού κόμβου τα range entries του RangeTLB δεν είχαν μεγάλο μέγεθος και το ποσοστό ευστοχίας RangeTLB μειώθηκε.

Τέλος, κατά την εκτέλεση με την επέκταση ***RCM***, το coverage των 32 largest mappings επανήλθε στο 100% και αυτό έχει ως αποτέλεσμα 100% RangeTLB hit και 0% address translation overhead. Η αύξηση επί του ιδανικού χρόνου εκτέλεσης (κίτρινη μπάρα) οφείλεται αποκλειστικά στα remote memory accesses που προέκυψαν λόγω της τοποθέτησης κάποιων vmas των διεργασιών στον μακρινό κόμβο.
Συγκριτικά, βλέπουμε ότι η επέκταση RCM, σε συνθήκες κατακερματισμένου τοπικού κόμβου, προκαλεί την μικρότερη αύξηση επί του ιδανικού χρόνου εκτέλεσης για όλα τα προγράμματα και έχει δηλαδή την καλύτερη επίδοση.

Κεφάλαιο 6 Μελλοντική Εργασία

Σε πολλά σημεία της εργασίας φάνηκε ότι οι εφαρμογές συχνά έχουν ένα hot κομμάτι μνήμης (ή και περισσότερα) το οποίο ευθύνεται για το μεγαλύτερο ποσοστό Cache και TLB misses. Αυτό μπορεί να συμβαίνει είτε γιατί εκείνη η περιοχή μνήμης συγκεντρώνει περισσότερες αναφορές ή γιατί το access pattern της είναι τέτοιο που ευνοεί τα misses ή και για τα δύο. Είδαμε ότι το hot κομμάτι μνήμης είναι προτιμότερο να απεικονιστεί συνεχόμενα και ας σταλεί σε μακρινό κόμβο, παρά να μείνει στον τοπικό κόμβο και να απεικονιστεί με πολλά μικρά κομμάτια. Για παράδειγμα, στο διάγραμμα 5.1 του κεφαλαίου 5 και για την περίπτωση του Liblinear-train συγκεκριμένα, είδαμε ότι η πολιτική μνήμης RHP κατάφερε να σβήσει το overhead που εισάγει ο κατακερματισμός μεγάλων πλαισίων του τοπικού κόμβου στέλνοντας το hot κομμάτι στον μακρινό κόμβο. Παρ' όλα αυτά εισάγει ένα μη αμελητέο overhead λόγω remote memory accesses. Το ιδανικό σε αυτή την περίπτωση θα ήταν το hot κομμάτι μνήμης να απεικονιστεί και στον τοπικό κόμβο και με μεγάλα πλαίσια. Για να γίνει αυτό θα πρέπει τα λίγα ελεύθερα μεγάλα πλαίσια του τοπικού κόμβου να χρησιμοποιηθούν αποκλειστικά για την απεικόνιση του hot κομματιού και να μην σπαταληθούν για κομμάτια μνήμης που δεν τα χρειάζονται αλλά τυχαίνει να αρχικοποιούνται πρώτα και να τα δεσμεύουν. Έτσι, λοιπόν μια ιδέα για μελλοντική εργασία θα ήταν αρχικά η μνήμη της διεργασίας να απεικονίζεται με τέτοιο τρόπο που να αποφεύγεται η χρήση μεγάλων πλαισίων ή ranges ώστε να αποφευχθεί πιθανή σπατάλη τους. Με δειγματοληψία των TLB και Cache misses μέσω event-based sampling ([21], [22]) θα εντοπίζονται δυναμικά τα hot κομμάτια μνήμης της διεργασίας και έπειτα αυτά θα απεικονίζονται ξανά (remapping), αυτή τη φορά συνεχόμενα στα μεγάλα πλαίσια ή ranges που αρχικά αφέθηκαν ελεύθερα για το σκοπό αυτό. Το σχήμα 6.1 δίνει ένα παράδειγμα ενός τέτοιου συστήματος.



Σχήμα 6.1: Το μεγαλύτερο ελεύθερο κομμάτι μνήμης του τοπικού κόμβου αρχικά αφήνεται ελεύθερο, με σκοπό να απεικονιστεί συνεχόμενα σε αυτό και όταν εντοπιστεί, το hot κομμάτι εικονικής μνήμης της εφαρμογής.

Κεφάλαιο 7 Βιβλιογραφία

[1] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris, "Enhancing and Exploiting Contiguity for Fast Memory Virtualization", in Proceedings of the 47th Annual International Symposium on Computer Architecture, 2020. http://www.cslab.ece.ntua.gr/~vkarakos/papers/isca20_enhancing_and_exploiting_contiguity.pdf

[2] numa (3) - Linux manual page - https://man7.org/linux/man-pages/man3/numa.3.html

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

[4] Automatic NUMA Balancing - 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

[5] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova and Vivien Quema, "Large Pages May Be Harmful on NUMA Systems" in Proceedings of the USENIX Annual Technical Conference, 2014. https://www.usenix.org/system/files/conference/atc14/atc14-paper-gaud.pdf

[6] David Tam, Reza Azimi and Michael Stumm, "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors", in Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, 2007. http://www.cs.toronto.edu/~demke/2227/S.14/Papers/p47-tam.pdf

[7] Baptiste Lepers, Vivien Quema and Alexandra Fedorova, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters" in Proceedings of the USENIX Annual Technical Conference, 2015. https://www.usenix.org/system/files/conference/atc15/atc15-paper-lepers.pdf

[8] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini and Srilatha Manne, "Accelerating Two-Dimensional Page Walks for Virtualized Systems", in ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, 2008 https://pages.cs.wisc.edu/~remzi/Classes/838/Spring2013/Papers/p26-bhargava.pdf

[9] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee, "Translation Ranger: Operating System Support for Contiguity-Aware TLBs", in the 46th Annual International Symposium on Computer Architecture, 2019 https://www.cs.yale.edu/homes/abhishek/ziyan-isca19.pdf

[10] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, Osman Ünsal "Redundant Memory Mappings for Fast Access to Large Memories", in Proceedings of the 42nd Annual International Symposium on Computer Architecture, 2015. https://pages.cs.wisc.edu/~swift/papers/isca15-rmm.pdf

[11] Memory Fragmentation (Wikipedia) - https://en.wikipedia.org/wiki/Fragmentation_(computing)

[12] The /proc Filesystem - https://www.kernel.org/doc/html/latest/filesystems/proc.html

[13] migrate_pages (8) - Linux manual page, https://man7.org/linux/man-pages/man2/migrate_pages.2.html

[14] J. R. Tramm, A. R. Siegel, T. Islam and M. Schulz, "XSBench – The development and verification of a performance abstraction for Monte Carlo reactor analysis", in PHYSOR

2014 – The Role of Reactor Physics toward a Sustainable Future, Kyoto, 2014. https://www.mcs.anl.gov/papers/P5064-0114.pdf
[15] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang and C.-J. Lin, "LIBLINEAR: A library for large linear classification", Journal of Machine Learning Research, vol. 9, 2008 https://www.csie.ntu.edu.tw/~cjlin/liblinear/
[16] McCalpin, John D., 1995: "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Computer Society Technical Committee on Computer Arc hitecture (TCCA) Newsletter, December 1995. https://www.researchgate.net/publication/51992086_Memory_bandwidth_and_machine_balance_in_high_performance_computers
[17] QEMU: the FAST! processor emulator - https://www.qemu.org/
[18] Linux perf tool - https://perf.wiki.kernel.org/index.php/Main_Page
[19] J. Gandhi, A. Basu, M. D. Hill and M. M. Swift, "BadgerTrap: A Tool to Instrument x86_64 TLB Misses", SIGARCH Comput. Archit. News, vol. 42, no. 2, Sep 2014 https://research.cs.wisc.edu/multifacet/papers/can14_badgertrap.pdf
[20] CAPaging page-collect utility - https://github.com/cslab-ntua/contiguity-isca2020/blob/master/tools/CAPaging_scripts/collect-statistic/native/page-collect.cpp
[21] Intel Precise Event Based Sampling (Intel PEBS) - https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
[22] AMD Instruction Based Sampling (AMD IBS) - https://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf
[23] Buddy Memory Allocation (Wikipedia) - https://en.wikipedia.org/wiki/Buddy_memory_allocation
[24] 3 ways to override C malloc function - http://www.cs.cmu.edu/afs/cs/academic/class/15213-s03/src/interposition/mymalloc.c
[25] Performance Application Programming Interface (PAPI) Library - https://icl.utk.edu/papi/news/news.html?id=382
[26] Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3 - https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html
[27] D. Gureya, J. Neto, R. Karimi, J. Barreto, P. Bhatotia, V. Quema, R. Rodrigues, P. Romano and V. Vlassov, "Bandwidth-Aware Page Placement in NUMA Systems", in 34th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2020. https://arxiv.org/pdf/2003.03304.pdf