



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

DPART: Deterministically Indexed Address Translation with Multiple Page Sizes via Partitioned Address Space

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ
ΑΘΗΝΑΓΟΡΑΣ - ΣΤΥΛΙΑΝΟΣ ΣΚΙΑΔΟΠΟΥΛΟΣ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
Αθήνα, Οκτώβριος 2019



Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

DPART: Deterministically Indexed Address Translation with Multiple Page Sizes via Partitioned Address Space

Διπλωματική Εργασία

του

Αθηναγόρα - Στυλιανού Σκιαδόπουλου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 15^η Οκτωβρίου, 2019.

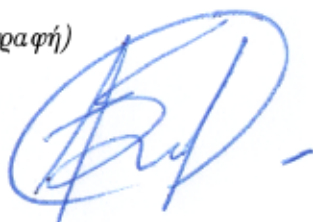
Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Γεώργιος Γκούμας
Επικ. Καθηγητής Ε.Μ.Π.

Δημήτριος Φωτάκης
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2019

(Υπογραφή)

A handwritten signature in blue ink, consisting of stylized, overlapping letters that appear to be 'Α', 'Σ', 'Κ', 'Α', 'Δ', 'Ο', 'Π', 'Ο', 'Υ', 'Λ', 'Ο', 'Σ'.

.....
Αθηνάγορας-Στυλιανός Σκιαδόπουλος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

© 2019– All rights reserved



Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Copyright © – All rights reserved Αθηναγόρας-Στυλιανός Σκιαδόπουλος, 2019.
Με επιφύλαξη παντός δικαιώματος.

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

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

Acknowledgements

The realization of this dissertation signifies the completion of my studies in National Technical University of Athens. As this journey comes to an end, I feel the necessity to express my thankfulness to some people, whose presence was a key factor to both the assertion of my thesis and my evolution in general.

First of all, I would like to express my deep gratitude to Prof. Nectarios Koziris for not only being an excellent supervisor of my thesis, but also for being a mentor and an example to me. Also, I would like to extend my appreciation to Mr Koziris, as him, alongside with Prof. Georgios Goumas, Konstantinos Nikas and Evangelos Koukis were the ones that deeply influenced me with their lectures, made me love the field of Computer Systems and inspired me to pursue a career in Computer Architecture despite being initially inclined towards more mathematically oriented fields.

Additionally, I would like to deeply thank members of CSLab Vasileios Karakostas, again Konstantinos Nikas and Chloe Alverti for providing me with any assistance I needed throughout my thesis. I remember Vasileios being in constant contact with me providing any research related help despite his busy schedule all year long, Konstantinos providing me with significant aid in accomplishing my goals, and Chloe delivering crucial technical knowledge and being always amenable to help. Also, I want to thank Stratos Psomadakis for his brief but critical contribution to my research.

Furthermore, I would like to express my thankfulness to people from my social circle and family for all their relentless support. Aside from making me feel lucky for being surrounded by them, the above people motivated me into achieving my goals and becoming a better person.

Athinagoras Skiadopoulos

Abstract

Virtual memory has been a vital contribution to computer systems, redefining memory utilization and substantially ameliorating computer programming experience. However, even being an inextricable element of modern architectures, virtual memory's address translation mechanism presents great performance overheads.

To overcome this, we propose DPART (**D**eterministically Indexed Address Translation with Multiple Page Sizes via **P**artitioned Address Space). Concretely, our proposed scheme partitions the virtual address space and allocates memory areas accordingly, so that a virtual address's most significant bits indicate the address's used page size. Leveraging this feature, we prototype a single set associative TLB structure accommodating translations from all page sizes. In addition, we describe a supporting page table managing to decode translations in less space.

Implemented in Linux 4.19, DPART achieves negligible TLB miss rates for most tested configurations and shows superiority against other schemes. Not limited in its TLB performance benefits, DPART characteristics promise low latencies, energy efficiency and reduced chip size.

Keywords

Virtual Memory, Address Translation, TLB

Περίληψη

Η εικονική μνήμη έχει αποτελέσει μία ζωτικής σημασίας συνεισφορά στα υπολογιστικά συστήματα, επαναπροσδιορίζοντας τη χρήση της μνήμης και βελτιώνοντας ουσιαστικά την προγραμματιστική εμπειρία. Ωστόσο, παρόλο που αποτελεί ένα αναπόσπαστο κομμάτι μοντέρνων αρχιτεκτονικών, ο μηχανισμός μετάφρασης διευθύνσεων της εικονικής μνήμης παρουσιάζει σημαντικό κόστος στην απόδοση.

Για να ξεπεραστεί αυτό το πρόβλημα, προτείνουμε το DPART (Ντετερμινιστικά Δεικτιοδοτημένη Μετάφραση Διευθύνσεων με Πολλαπλά Μεγέθη Σελιδών μέσω Διαμερισμένου Χώρου Διευθύνσεων). Συγκεκριμένα, το σχήμα που προτείνουμε διαμερίζει τον εικονικό χώρο διευθύνσεων και αναθέτει τμήματα μνήμης κατάλληλα έτσι ώστε τα δεξιότερα ψηφία μίας εικονικής διεύθυνσης να δηλώνουν το μέγεθος σελίδας που χρησιμοποιεί αυτή η διεύθυνση. Αξιοποιώντας αυτό το χαρακτηριστικό, κατασκευάζουμε ένα μοναδικό set associative TLB που φιλοξενεί μεταφράσεις από κάθε μέγεθος σελίδας. Επιπρόσθετα, περιγράφουμε ένα υποστηριζόμενο πίνακα σελιδών που καταφέρνει να συμπίπτει την αποθήκευση μεταφράσεων σε λιγότερο χώρο.

Υλοποιημένο σε πυρήνα Linux 4.19, το DPART καταφέρνει να έχει αμελητέες τιμές αστοχιών TLB για τους περισσότερους συνδυασμούς παραμέτρων TLB που αναλύθηκαν, επιδεικνύοντας ανωτερότητα σε σχέση με άλλα σχήματα. Χωρίς να περιορίζεται στα πλεονεκτήματα που απορρέουν από την απόδοσή του TLB του, τα χαρακτηριστικά του DPART υποσχονται χαμηλούς χρόνους διεκπαιρέωσης, ενεργειακή αποδοτικότητα και μειωμένο μέγεθος τσιπ.

Λέξεις Κλειδιά

Εικονική Μνήμη, Μετάφραση Διευθύνσεων, TLB

Contents

| | |
|--|-----------|
| Acknowledgements | i |
| Abstract | iii |
| Περίληψη | v |
| Contents | ix |
| List of Figures | xii |
| List of Tables | xiii |
| List of Listings | xv |
| 1 Εκτεταμένη Ελληνική Περίληψη (Extended Greek Summary) | 1 |
| 1.1 Εισαγωγή | 1 |
| 1.1.1 Θεωρητικό Υπόβαθρο | 1 |
| 1.1.2 Συνεισφορά Διπλωματικής | 2 |
| 1.2 Το Σχήμα DPART | 3 |
| 1.2.1 Διαμέριση Εικονικού Χώρου | 3 |
| 1.3 Το DPART TLB | 4 |
| 1.4 Ο Πίνακας Σελίδων του DPART | 7 |
| 1.5 Επιπλέον Βελτιώσεις | 8 |
| 1.6 Αξιολόγηση | 10 |
| 1.7 Επίλογος | 11 |
| 2 Introduction | 13 |
| 2.1 Brief Problem Formulation | 13 |
| 2.2 Contribution | 13 |

| | | |
|----------|--|-----------|
| 2.3 | Document Outline | 14 |
| 3 | Background | 15 |
| 3.1 | Virtual Memory Overview | 15 |
| 3.2 | Paging | 16 |
| 3.3 | Page Table | 17 |
| 3.4 | The TLB Mechanism | 18 |
| 3.5 | Virtual Memory Allocation | 19 |
| 4 | The DPART Scheme | 23 |
| 4.1 | Description of an Ideal Address Translation Scheme | 23 |
| 4.2 | Two Page Sizes Initiative | 25 |
| 4.3 | DPART | 29 |
| 4.3.1 | Virtual Address Space Partitioning | 29 |
| 4.3.2 | Usage of Multiple Page Sizes | 31 |
| 4.3.3 | DPART TLB | 35 |
| 4.3.4 | DPART Page Table | 37 |
| 4.4 | Further Optimizations | 42 |
| 4.4.1 | Skewing | 42 |
| 4.4.2 | The Heap Overhead | 46 |
| 4.5 | Restrictions | 46 |
| 5 | Experiments | 49 |
| 5.1 | Methodology | 49 |
| 5.1.1 | System | 49 |
| 5.1.2 | Simulating TLB Misses | 50 |
| 5.1.3 | Software TLB Simulators | 51 |
| 5.1.3.1 | Comparison of Different DPART Configurations | 51 |
| 5.1.3.2 | Comparison Between DPART and Other Address Translation Schemes | 52 |
| 5.2 | Benchmark Analysis | 54 |
| 5.3 | Results | 56 |
| 5.3.1 | First Results | 56 |
| 5.3.2 | OS Changes Execution Time Overhead | 56 |
| 5.3.3 | Comparison of Different DPART Configurations Results | 57 |
| 5.3.4 | Comparison Between DPART and Other Address Translation Schemes Results | 58 |

| | |
|---|-----------|
| 5.4 Discussion | 60 |
| 6 Related Work | 63 |
| 6.1 Range Based Schemes | 63 |
| 6.2 Leveraging Contiguity | 64 |
| 6.3 Prediction Based Mechanisms | 64 |
| 6.4 Address Space Partitioning | 66 |
| 6.5 Groundbreaking Approaches | 66 |
| 7 Conclusion and Future Work | 69 |
| 7.1 Future Work | 69 |
| 7.1.1 Further Implementation | 69 |
| 7.1.2 Further Evaluation | 70 |
| 7.2 Concluding Remarks | 71 |
| Bibliography | 73 |
| Appendices | 79 |

List of Figures

| | | |
|------|---|----|
| 1.1 | Συστατικά εικονικής διεύθυνσης | 2 |
| 1.2 | Διαμέριση εικονικής μνήμης | 3 |
| 1.3 | Εύρεση των τμημάτων tag, index και offset μίας εικονικής διεύθυνσης | 5 |
| 1.4 | Χρήση μασκών για εύρεση tag και offset | 6 |
| 1.5 | Δομή των καταχωρήσεων του TLB | 6 |
| 1.6 | Συστατικά διεύθυνσης που αντιστοιχεί σε σελίδα 32KB | 7 |
| 1.7 | Modified page table under the third approach | 8 |
| 1.8 | Παράδειγμα κατανομής μεταφράσεων στα set του TLB πριν (α) και μετά (β) τον μηχανισμό διάχυσης | 9 |
| 1.9 | Ρυθμός αστοχίας TLB για μεταβαλλόμενο (α) πλήθος καταχωρήσεων και (b) associativity | 11 |
| 3.1 | Virtual address components | 16 |
| 3.2 | Page walk in a 4 level page table | 18 |
| 3.3 | Virtual address TLB components | 19 |
| 3.4 | Memory regions bounds in GemsFDTD benchmark | 20 |
| 4.1 | Index bits position in different page sizes | 26 |
| 4.2 | Hypothetical isomorphic layer between virtual and physical address space | 26 |
| 4.3 | Address tag, index and offset identification with 2 supported page sizes | 27 |
| 4.4 | Tag and offset masks | 28 |
| 4.5 | Virtual address space partition | 29 |
| 4.6 | Virtual memory area search | 30 |
| 4.7 | Page size utilized for each partition | 33 |
| 4.8 | TLB entries structure | 35 |
| 4.9 | Address tag, index and offset identification for $2^n - 1$ supported page sizes | 36 |
| 4.10 | TLB mechanism | 37 |
| 4.11 | Part of page table | 38 |

| | | |
|------|---|----|
| 4.12 | Addresses page table components | 38 |
| 4.13 | 32KB page address page table components | 40 |
| 4.14 | Modified page table under the third approach | 41 |
| 4.15 | Top-down memory allocation in each partition | 42 |
| 4.16 | Bottom TLB sets accumulation (pages not in scale) | 43 |
| 4.17 | Skewing schemes for indices longer or shorter than partition bits | 44 |
| 4.18 | Page skewing (pages not in scale) | 45 |
| | | |
| 5.1 | System calls frequency with/without TCMalloc | 55 |
| 5.2 | Execution time with/without DPART OS modifications | 57 |
| 5.3 | TLB miss rates for variant partitioning bits | 58 |
| 5.4 | TLB miss rates for variant TLB (a) size and (b) associativity | 59 |
| 5.5 | Same associativity TLBs comparison | 60 |
| | | |
| A.1 | All possible 3-node chain trees violate the red-black tree properties | 81 |
| B.1 | TLB miss rates for variant partitioning bits | 82 |
| C.1 | TLB miss rates for 5 partitioning bits on variant size | 83 |
| D.1 | TLB miss rates for 5 partitioning bits on variant associativity | 84 |

List of Tables

| | | |
|-----|--|----|
| 1.1 | Διαθέσιμα μεγέθη σελίδων | 4 |
| 4.1 | Partition sizes for 4-level and 5-level page tables | 32 |
| 4.2 | Available page sizes under each partitioning scheme | 32 |
| 4.3 | Page assignment examples for each policy under 3 bits partitioning | 34 |
| 5.1 | System configuration | 50 |
| 5.2 | DPART simulations configurations | 51 |
| 5.3 | Simulation parameters on varying partition bits | 52 |
| 5.4 | Simulation parameters on varying TLB size | 53 |
| 5.5 | Simulation parameters on varying TLB associativity | 53 |

List of Listings

- 4.1 Skew a pseudocode 44
- 4.2 Skew b pseudocode 44
- 5.1 Tlb miss microbenchmark pseudocode 56

Chapter 1

Εκτεταμένη Ελληνική Περίληψη (Extended Greek Summary)

1.1 Εισαγωγή

1.1.1 Θεωρητικό Υπόβαθρο

Η εικονική μνήμη αποτελεί μία σημαντική συνεισφορά στον τομέα των υπολογιστικών συστημάτων, συμβάλλοντας στην αποδοτική και ασφαλή διαχείριση των πόρων μνήμης ενός συστήματος, και διευκολύνοντας τον προγραμματισμό. Η εικονική μνήμη αποτελεί έναν εικονικό χώρο διευθύνσεων για κάθε διεργασία, στον οποίο της παρέχεται η ψευδαίσθηση ότι σχεδόν άπειρη μνήμη είναι στη διάθεσή της, δίχως την παρέμβαση άλλων διεργασιών.

Για τη χρησιμοποίηση των φυσικών πόρων του μηχανήματος, το λειτουργικό σύστημα αναλαμβάνει την απεικόνιση των εικονικών (λογικών) διευθύνσεων σε φυσικές. Η διαδικασία αυτής της αντιστοίχισης ονομάζεται μετάφραση διευθύνσεων. Κάθε φορά που μία διεργασία χρειάζεται ένα δεδομένο της, ο κώδικας της δηλώνει ποια εικονική θέση μνήμης χρειάζεται, το λειτουργικό βρίσκει την απαιτούμενη μετάφραση, και έτσι λαμβάνεται τη φυσική θέση μνήμης που εμπεριέχει το ζητούμενο δεδομένο. Στη συνέχεια, το δεδομένο αυτό μπορεί να χρησιμοποιηθεί κατάλληλα από το πρόγραμμα.

Καθώς θα ήταν κοστοβόρο να αποθηκεύονται όλες οι μεταφράσεις ενός συστήματος ξεχωριστά, χρησιμοποιείται ο μηχανισμός της σελιδοποίησης. Συγκεκριμένα, 2^x μεταφράσεις οργανώνονται σε μία δομή η οποία ονομάζεται σελίδα. Αυτές οι μεταφράσεις είναι αναγκαίο να είναι συνεχόμενες τόσο σε εικονική όσο και σε φυσική μνήμη, και να είναι επίσης

ευθυγραμμισμένες (τα τελευταία x bits της πρώτης διεύθυνσης να είναι ίσα με 0). Με αυτόν τον τρόπο, είναι αρκετό να αποθηκευτούν οι μεταφράσεις μόνο των αριθμών σελίδων, όπως φαίνεται στο παρακάτω σχήμα.

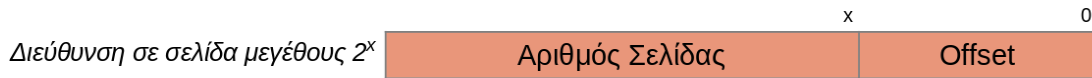


Figure 1.1: Συστατικά εικονικής διεύθυνσης

Μικρές σελίδες έχουν το πλεονέκτημα να απεικονίζουν επακριβώς τα δεδομένα που χρειάζονται από μία διεργασία, και είναι ευέλικτες ως προς τη συνύπαρξή τους στην κύρια μνήμη. Ωστόσο, καθώς λιγότερες μεταφράσεις μπορούν να υπάρξουν σε μία μικρή σελίδα. Συνεπώς, για εφαρμογές με υψηλή χρήση μνήμης όπου χρειάζονται περισσότερες σελίδες, το TLB δεν είναι ικανό να φιλοξενήσει όλες τις αναγκαίες μεταφράσεις και παρατηρείται πτώση στην απόδοσή του. Για αυτό χρησιμοποιούνται μεγαλύτερα μεγέθη σελίδων, που όμως έχουν τα δικά μειονεκτήματα.

Η δομή στην οποία αποθηκεύονται όλες οι μεταφράσεις είναι ο πίνακας σελίδων. Ο πίνακας σελίδων είναι μία δενδρική δομή, η οποία στα φύλλα της περιέχει τις ζητούμενες μεταφράσεις. Στα προηγούμενα επίπεδα, οι κόμβοι του δείχνουν προς το κατάλληλο κόμβο παιδί που πρέπει να ακολουθηθεί. Η δεικτοδότηση γίνεται διαμερίζοντας τον εικονικό αριθμό σελίδας σε τόσα μέρη όσα τα επίπεδα του πίνακα σελίδων.

Ο πίνακας σελίδων είναι μία δομή που επιφέρει υψηλούς χρόνους πρόσβασης, συνεπώς, για την εύρεση μίας μετάφρασης ελέγχεται πρώτα το TLB. Το TLB αποτελεί μία μικρότερη δομή που περιέχει ένα υποσύνολο με τις συχνότερα χρησιμοποιούμενες μεταφράσεις και προσφέρει σημαντικά μικρότερο χρόνο προσπέλασης. Το TLB επίσης δεικτοδοτείται με το index μέρος της διεύθυνσης, το οποίο καθορίζει ένα υποσύνολο του TLB στο οποίο μπορεί να κατοικεί η ζητούμενη μετάφραση.

1.1.2 Συνεισφορά Διπλωματικής

Ο μηχανισμός της μετάφρασης διευθύνσεων είναι συχνά ιδιαίτερα κοστοβόρος εξαιτίας του συχνά χαμηλού ρυθμού ευστοχίας του TLB. Επίσης σε ένα συμβατικό TLB, λόγω των διαφορετικών θέσεων του index μέρους μίας διεύθυνσης για τα διάφορα μεγέθη σελίδας, η υποστήριξη πολλαπλών μεγεθών είναι δύσκολη. Η παρούσα διπλωματική άρει αυτό το εμπόδιο με το σχήμα DPART. Το DPART προσφέρει ντετερμινιστικό indexing σε μετάφραση διευθύνσεων με πολλαπλά μεγέθη σελίδων, μέσω μιας διαμέρισης του εικονικού χώρου διευθύνσεων.

Το σχήμα μας συνοδεύεται από:

- Το DPART TLB, μία μοναδική δομή που εμπεριέχει μεταφράσεις από όλα τα μεγέθη σελίδων
- Υλοποίηση ενός πίνακα σελίδων που εξοικονομεί τον χώρο απεικόνισής του
- Μεταβολές στο λειτουργικό σύστημα Linux 4.19 για την υποστήριξη των μηχανισμών μας

1.2 Το Σχήμα DPART

1.2.1 Διαμέριση Εικονικού Χώρου

Το σχήμα μας επιδιώκει να καθιερώσει μία αμφιμονοσήμαντη αντιστοίχιση ανάμεσα στα n δεξιότερα bits (MSB) μίας εικονικής διεύθυνσης, και του μεγέθους σελίδας που εμπεριέχει τη διεύθυνση αυτή. Για να επιτευχθεί κάτι τέτοιο, ο εικονικός χώρος διευθύνσεων διαμερίζεται σε 2^n υποχώρους, όπου ο καθένας αντιπροσωπεύει ένα μέγεθος. Παρακάτω απεικονίζεται η περίπτωση για $n = 2$.

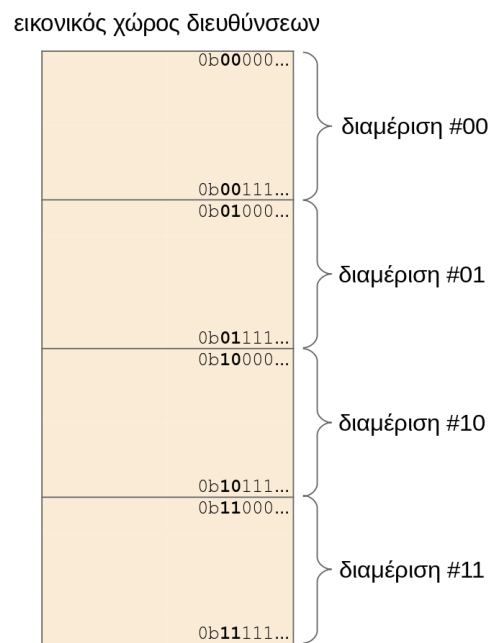


Figure 1.2: Διαμέριση εικονικής μνήμης

Όπως παρατηρείται, η τεχνική αυτή έχει ως αποτέλεσμα κάθε διαμέριση να προσδιορίζεται μοναδικά από τις τιμές των MSBs των διευθύνσεων που περιέχονται σε αυτήν. Για να

πετύχουμε τώρα την προαναφερθείσα σύνδεση διαμερίσεων και μεγέθους σελίδας, τροποποιούμε τις αντίστοιχες συναρτήσεις του Linux που είναι υπεύθυνες για την ανάθεση νέου εικονικού χώρου. Από τα διαθέσιμα μεγέθη σελίδας που παρέχονται, αναθέτουμε κάθε φορά το κατάλληλο, ανάλογα με την πολιτική που χρησιμοποιείται. Οι επιλογές που έχουν υλοποιηθεί είναι οι:

- LOWER
- CLOSER
- UPPER

των οποίων οι ορισμοί είναι οι εξής:

$$page_size_{lower}(length) = \sup_{ps \in avail_page_sizes} \{ps \mid ps \leq length\}$$

$$page_size_{closer}(length) = \arg \min_{ps \in avail_page_sizes} |\log_2(ps) - \log_2(length)|$$

$$page_size_{upper}(length) = \inf_{ps \in avail_page_sizes} \{ps \mid ps \geq length\}$$

Τέλος, ανάλογα με το πλήθος των MSBs που χρησιμοποιούνται, μας δίνεται μεγαλύτερη ευελιξία ως προς τα μεγέθη σελίδων που μπορούν να υποστηριχτούν. Αυτά αναφέρονται στον παρακάτω πίνακα:

| Πλήθος MSBs | Μεγέθη σελίδων |
|-------------|--|
| 0 | 4K |
| 2 | 4K, 2M, 1G |
| 3 | 4K, 32K, 256K, 2M, 16M, 128M, 1G |
| 4 | 4K, 8K, 32K, 64K, 256K, 512K, 2M, 4M, 16M, 32M, 128M, 256M, 1G, 2G, 8G |
| 5 | όλες οι δυνάμεις του δύο από 4K έως 32G |

Table 1.1: Διαθέσιμα μεγέθη σελίδων

1.3 Το DPART TLB

Έχοντας εξασφαλίσει τον προσδιορισμό του μεγέθους σελίδας από τα MSBs της εικονικής διεύθυνσης, μπορούμε να εκμεταλλευτούμε αυτή τη σύνδεση στην υλοποίηση του TLB του

σχήματός μας.

Για τη χρήση ενός TLB είναι απαραίτητα τα μεγέθη tag, index, offset της εικονικής διεύθυνσης. Το offset είναι με την απόσταση που έχει μία διεύθυνση από την αρχή της σελίδας που την περιέχει. Το index αφορά τα αμέσως επόμενα bits και είναι τα LSBs του αριθμού σελίδας που χρησιμοποιούνται για να δεικτοδοτηθεί το κατάλληλο set του TLB. Το tag αποτελεί το υπόλοιπο κομμάτι της εικονικής διεύθυνσης και χρειάζεται για να συγκριθεί με το περιεχόμενο ενός TLB entry έτσι ώστε να αποφανθεί εάν μία μετάφραση είναι η ζητούμενη.

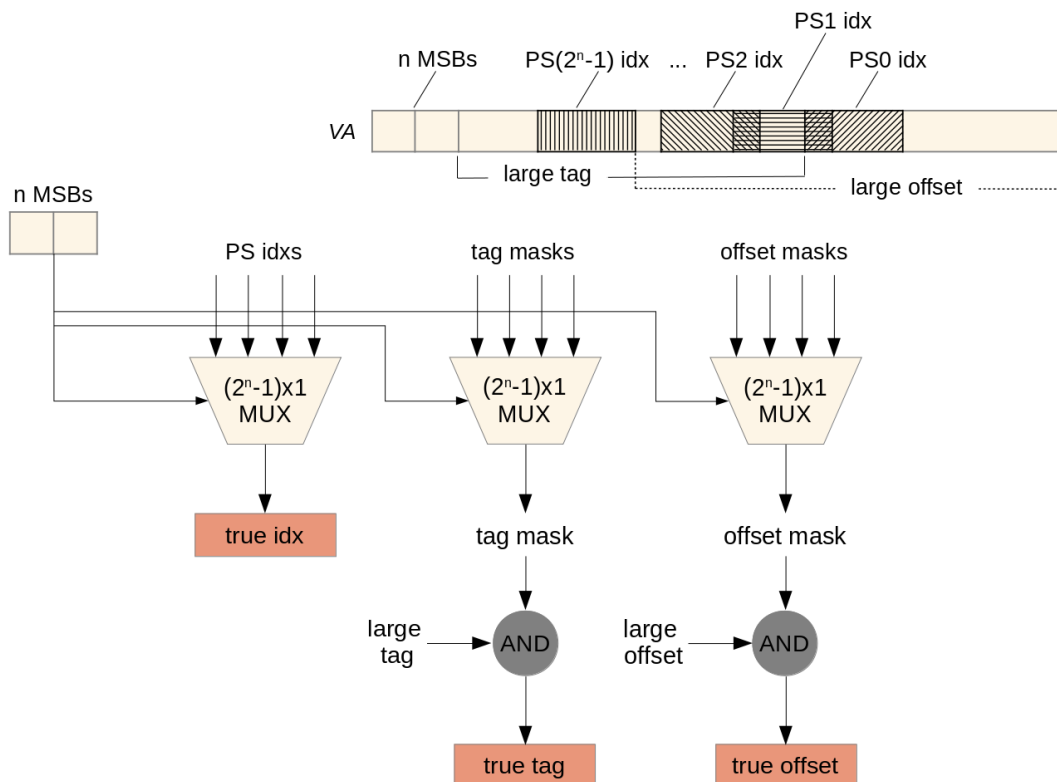


Figure 1.3: Εύρεση των τμημάτων tag, index και offset μίας εικονικής διεύθυνσης

Στο παραπάνω σχήμα παρουσιάζεται ο τρόπος χρήσης των MSBs της διεύθυνσης για τον προσδιορισμό αυτών των τριών ποσοτήτων. Τρεις πολυπλέκτες με επιλογή τα κατάλληλα MSBs της διεύθυνσης, κάθε φορά επιλέγουν είτε το κατάλληλο index, είτε την επιθυμητή μάσκα που θα εφαρμοστεί στο μεγαλύτερο δυνατό tag ή offset έτσι ώστε να παραχθούν τα κατάλληλα αντίστοιχα μεγέθη.

Παράδειγμα τέτοιων μασκών φαίνεται στο σχήμα που ακολουθεί:

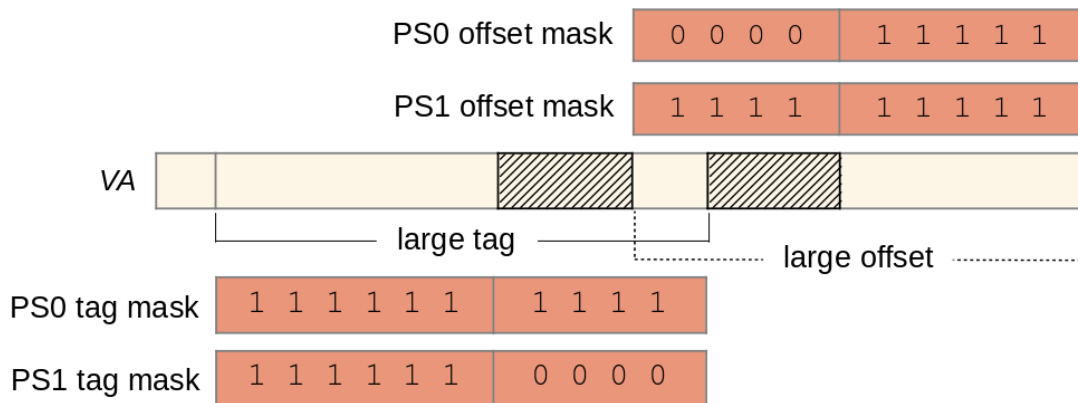


Figure 1.4: Χρήση масκών για εύρεση tag και offset

Η εφαρμογή αυτών των масκών έχει ως συνέπεια σε ένα zero padding του εξαγόμενου tag. Κάτι τέτοιο παρουσιάζει πλεονεκτήματα όσον αφορά την ομοιομορφία και τη μοναδικότητα των tags. Συγκεκριμένα, αρχικά κάθε tag, συνεπώς και κάθε καταχώρηση του TLB έχει ισοπληθή bits. Επιπλέον, μπορεί να αποδειχθεί ότι διευθύνσεις που ανήκουν σε διαφορετικές σελίδες θα απεικονίζονται και με διαφορετικά tags. Η απόδειξη αυτής της πρότασης χωρίζεται σε δύο σκέλη. Το πρώτο σκέλος αφορά διευθύνσεις που ανήκουν σε σελίδες ίδιου μεγέθους. Αυτή η περίπτωση είναι τετριμμένη καθώς διαφορετικές σελίδες θα έχουν διαφορετικά page numbers, κάτι που υποδηλώνει και διαφορετικά tags αφού αυτά αποτελούνται από ίσου πλήθους bits. Το δεύτερο σκέλος αφορά διευθύνσεις που ανήκουν σε σελίδες διαφορετικού μεγέθους. Σε αυτήν την περίπτωση, λόγω του zero padding που έχει εφαρμοστεί, τα MSBs των αντίστοιχων tags είναι ευθυγραμμισμένα. Όμως αυτά είναι διαφορετικά, εξαιτίας της ιδιότητας του partitioning.

Ο συνδυασμός των επιχειρημάτων της ομοιομορφίας και της μοναδικότητας έχει ως αποτέλεσμα να μπορούν μεταφράσεις από όλα τα μεγέθη σελίδων να συνυπάρχουν στο ίδιο TLB.

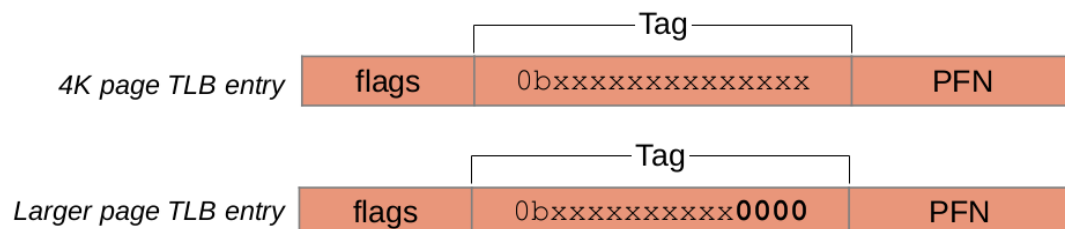


Figure 1.5: Δομή των καταχωρήσεων του TLB

1.4 Ο Πίνακας Σελίδων του DPART

Στην παρούσα ενότητα παρουσιάζεται ο πίνακας σελίδων που υποστηρίζει το σχήμα μας.

Στο σημείο αυτό σημειώνεται ότι το μέγεθος σελίδας είναι ένα χαρακτηριστικό του μηχανισμού που μεταχειρίζεται μία διεύθυνση, και όχι της ίδιας της διεύθυνσης. Αυτό επιτρέπει στον πίνακα σελίδων μας να λειτουργεί είτε αποκλειστικά με σελίδες 4KB, είτε με το υπάρχον σχήμα που υποστηρίζει Transparent Huge Pages [10].

| | PGD idx | PUD idx | PMD idx | PTE idx | offset |
|------------------|---------|---------|---------|---------|---------|
| VA in a 32K page | 9 bits | 9 bits | 9 bits | 6 bits | 15 bits |

Figure 1.6: Συστατικά διεύθυνσης που αντιστοιχεί σε σελίδα 32KB

Στην υλοποίηση που εκμεταλλεύεται τα μεγέθη σελίδων που προσφέρει το DPART, παρατηρούμε αρχικά ότι ο αριθμός σελίδας δεν είναι εν γένει πολλαπλάσιο του 9, όπως φαίνεται στο σχήμα 1.6. Κάθε σύνολο κόμβων του πίνακα σελίδων όμως που αποθηκεύεται σε μία σελίδα της μνήμης αποτελείται από 2^9 θέσεις. Αυτό όμως έχει ως συνέπεια, το τελευταίο επίπεδο του πίνακα περιέχει λιγότερα από 9 bits και οι σελίδες που βρίσκονται στα φύλλα του να έχουν λιγότερες από 2^9 καταχωρήσεις. Κάτι τέτοιο απεικονίζεται στο παρακάτω σχήμα 1.7.

Το γεγονός αυτό δεν αποτελεί μειονέκτημα αφού σελίδες του τελευταίου επιπέδου μπορούν πλέον να συμπυκνθούν εξοικονομώντας έτσι σημαντικό χώρο. Σημειώνεται, ότι το σχήμα αυτό είναι τουλάχιστον ανώτερο από τον πίνακα σελίδων των transparent huge pages.

Τα βήματα προσπέλασης του πίνακα σελίδων σε περίπτωση αστοχίας του TLB αναγράφονται παρακάτω:

- Αναγνώριση μεγέθους σελίδας μέσω των MSBs της διεύθυνσης
- Εξαγωγή των index για τα διάφορα επίπεδα και του offset
- Διάσχιση των επιπέδων ανάλογα με τα προηγούμενα indices
- Συνένωση φυσικού αριθμού σελίδας που βρέθηκε με το offset για τον προσδιορισμό της φυσικής διεύθυνσης
- Ενημέρωση του TLB με τον φυσικό αριθμό σελίδας

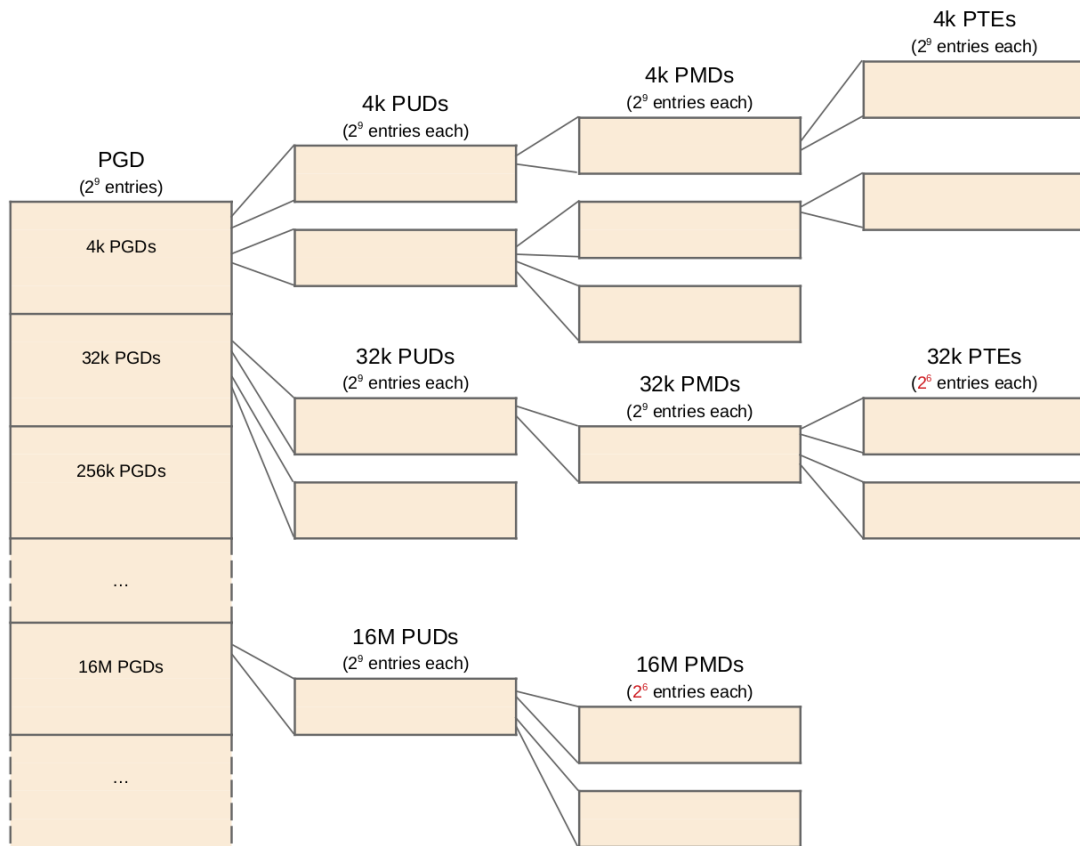


Figure 1.7: Modified page table under the third approach

1.5 Επιπλέον Βελτιώσεις

Στον πυρήνα του Linux χρησιμοποιείται ανάθεση διευθύνσεων από τις υψηλότερες προς τις χαμηλότερες. Κάτι τέτοιο τώρα μεταφράζεται σε κάθε partition ξεχωριστά. Ως αποτέλεσμα, εντοπίζεται συμφύρση στα τελευταία sets του TLB αφού οι πρώτες σελίδες που ανατίθενται από κάθε partition έχουν ίσο index (το μεγαλύτερο δυνατό). Το φαινόμενο αυτό απεικονίζεται στο σχήμα 1.8 (α).

Για την αντιμετώπισή του, προσπαθούμε να διαχύσουμε τις μεταφράσεις στα διάφορα sets εκμεταλλευόμενοι των διαφοροποιήσεων στα MSBs του κάθε partition. Για να πραγματοποιήσουμε αυτήν την διάχυση, εφαρμόζουμε κατάλληλη πράξη XOR ανάμεσα στα MSBs και το index. Το αποτέλεσμα φαίνεται στο σχήμα 1.8 (β).

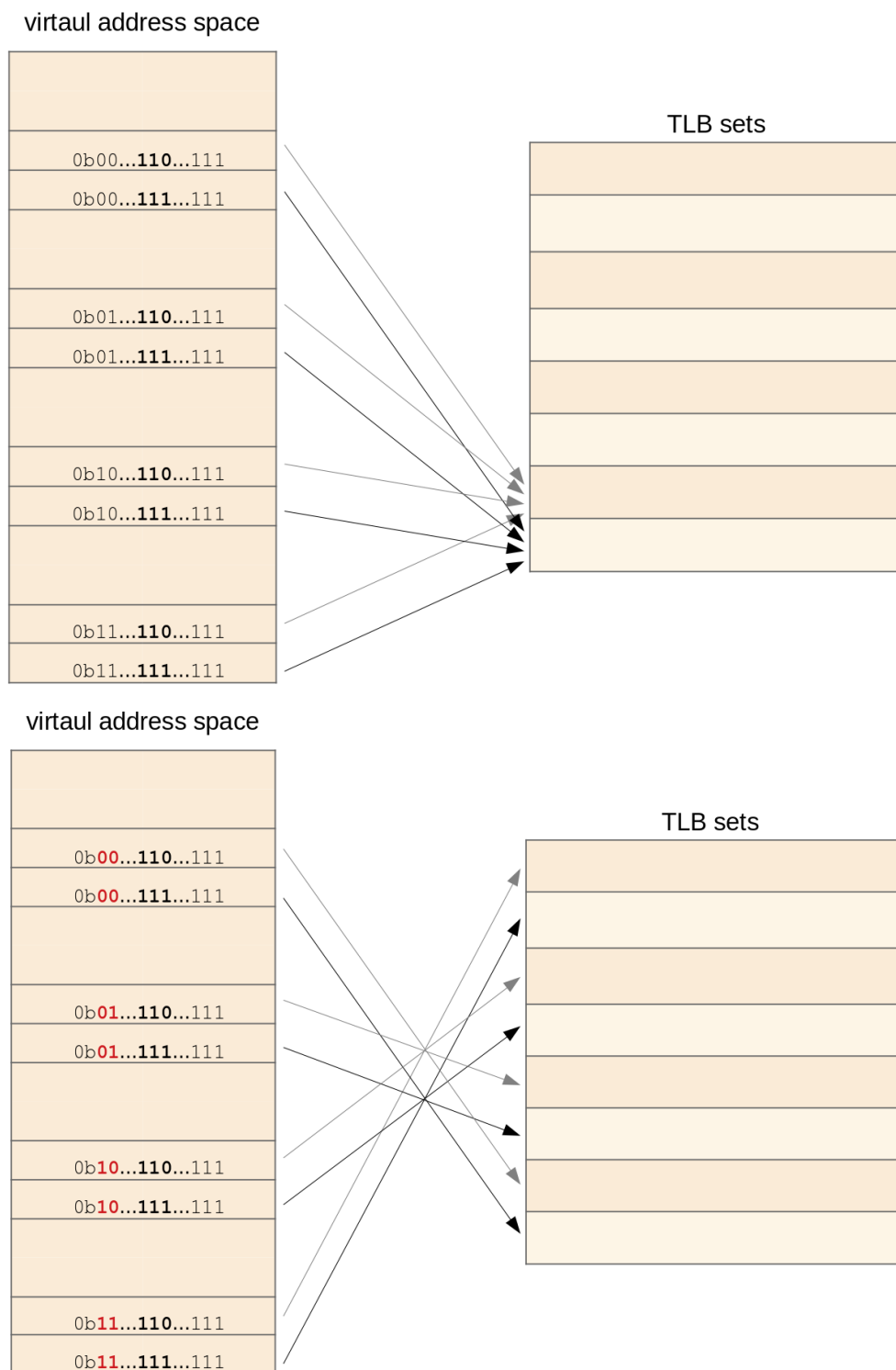


Figure 1.8: Παράδειγμα κατανομής μεταφράσεων στα set του TLB πριν (α) και μετά (β) τον μηχανισμό διάχυσης

Μία επιπλέον βελτίωση του DPART αφορά την απεικόνιση του σωρού. Συγκεκριμένα, ο σωρός σε πολλά προγράμματα παίρνει μεγάλες διαστάσεις και καθώς απεικονίζεται με σελίδες μικρού μεγέθους συνεπάγεται σε χαμηλές αποδόσεις. Για την επίλυση αυτού του προβλήματος, αποσκοπούμε στην απεικόνιση του σωρού σε κάποια διαμέριση του εικονικού χώρου που χρησιμοποιεί μεγαλύτερες σελίδες.

Κάτι τέτοιο στην παρούσα φάση έχει υλοποιηθεί μόνο σε επίπεδο προσομοίωσης. Σε ένα πραγματικό σύστημα, κάτι τέτοιο θα μπορούσε να επιτευχθεί με ποικίλους τρόπους. Αρχικά, θα μπορούσε εφόσον ο σωρός ξεπεράσει κάποιο κριτήριο μεγέθους, να μεταφερθεί τότε σε μία διαμέριση που χρησιμοποιεί μεγαλύτερο μέγεθος. Μία άλλη λύση εμπεριέχει την μείωση του ορίου κάτω από το οποίο χρησιμοποιείται ο σωρός, έτσι ώστε μεγάλες απεικονίσεις να πραγματοποιούνται από ανώνυμες μεταφράσεις. Τέλος, προτείνουμε την μεταβολή της αρχής του σωρού κατά την εκκίνηση της διεργασίας, έτσι ώστε να μην χρειαστεί η μεταφορά του σε διαφορετική διαμέριση αργότερα.

1.6 Αξιολόγηση

Σε αυτήν την ενότητα φαίνονται τα αποτελέσματα στα πειράματα που διεξήχθησαν σχετικά με το ποσοστό αστοχιών TLB. Σε σύγκριση τέθηκαν τα παρακάτω πέντε σχήματα:

- conventional
- hugepages split
- hugepages merged
- dpart_skew_b
- RMM

Το πρώτο σχήμα χρησιμοποιεί αποκλειστικά σελίδες 4KB. Τα δύο επόμενα αφορούν transparent huge pages σε ξεχωριστά ή κοινό TLB. Η δική μας μέθοδος χρησιμοποιεί 5 MSBs για partitioning, closer πολιτική, και είναι εξοπλισμένη με αμφότερες τις βελτιστοποιήσεις. Τέλος το RMM αποτελεί το άνω φράγμα της ανάλυσής μας και αποθηκεύει περιοχές αυθαίρετου μήκους σε μία fully associative δομή.

Όπως φαίνεται στα επόμενα σχήματα, το σχήμα μας καταφέρνει να προσεγγίσει την απόδοση του RMM, ενώ είναι συγκριτικά ανώτερο από τόσο το συμβατικό σχήμα, όσο και αυτό που κάνει χρήση THP.

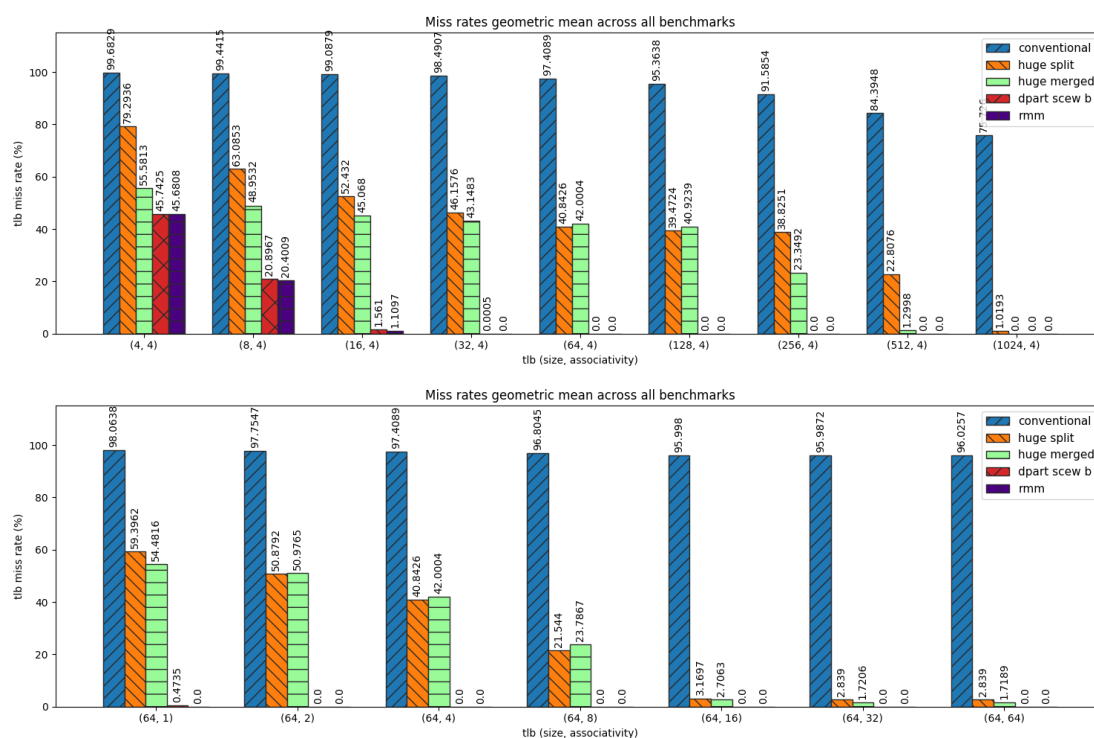


Figure 1.9: Ρυθμός αστοχίας TLB για μεταβαλλόμενο (α) πλήθος καταχωρήσεων και (β) associativity

1.7 Επίλογος

Η βελτιστοποίηση της εικονικής μνήμης είναι ένας από τους καθοριστικούς τομείς στην αύξηση της απόδοσης ενός υπολογιστικού συστήματος. Το σχήμα που προτείνουμε αποτελεί μία αποδοτική μέθοδο μετάφρασης διευθύνσεων. Υποστηρίζοντας πολλά μεγέθη σελίδων σε μία κοινή δομή TLB, το DPART καταφέρνει να έχει εξαιρετικά χαμηλά ποσοστά αστοχίας.

Εκτός από την επίδοση του TLB του όμως, το DPART εμπεριέχει επιπλέον πλεονεκτήματα. Η απλότητα του design του υπόσχεται χαμηλούς χρόνους απόκρισης και ενεργειακή κατανάλωση. Επιπλέον, η μοναδική δομή TLB συμβάλλει στη μείωση του μεγέθους του επεξεργαστή. Η αξιολόγηση των παραπάνω επιπλέον σημείων αποτελεί μελλοντική δουλειά.

Σε αυτήν εντάσσονται επίσης υλοποιήσεις που έρχονται να τελειοποιήσουν και να εξελίξουν τον μηχανισμό μας. Αυτές αφορούν στο κομμάτι της διαχείρισης φυσικής μνήμης, την ένταξη στο λογισμικό της βελτιωμένης απεικόνισης του σωρού, μηχανισμός που να καθορίζει το μέγεθος σελίδας που πρέπει να χρησιμοποιηθεί από μέσω πιο πλούσιων κριτηρίων, και τέλος υλοποίηση στο υλικό.

Chapter 2

Introduction

2.1 Brief Problem Formulation

Virtual memory has substantially contributed to the establishment of efficient and secure memory utilization, while also enhancing a programmer’s productivity. However, despite its incontrovertible benefits, in modern computer systems virtual memory’s address translation mechanism has become a major bottleneck in demanding applications’ performance.

Emerging technologies such Intel’s 3D XPoint [27], Hybrid Memory Cube (HMC) [51] and High Bandwidth Memory (HBM) [34] in combination with increasing capacity demands by heavy workload applications, highlight the importance of optimizing virtual memory while also maintaining affordable power consumption. Basu et al. have found that applications can spend up to 51% of execution cycles in TLB misses with 4KB pages and 10% of execution cycles using 2MB superpages [15]. Also, recent research has shown that NUMA systems, which are broadly used nowadays, may even experience performance degradation under large pages [25].

2.2 Contribution

In an effort to overcome most address translation schemes’ inability to encounter the aforementioned problems, we propose DPART (**D**eterministically Indexed Address Translation with Multiple Page Sizes via **P**artitioned Address Space). DPART manages to tackle the problem of address translation by a single TLB structure accommodating various page sizes, and a deterministic mechanism that defines a priori the page size of a specific translation. To achieve this, DPART performs a partition of the virtual ad-

dress space and allocates pages accordingly, so that their most significant bits indicate their used page size. Our software and hardware co-design scheme is transparent to the application, and includes:

- DPART TLB, a single deterministic set associative structure that may accommodate any power of two page size
- Outlines on the implementation of a page table optimizing its representation space
- Operating system support in Linux 4.19 enabling our mechanisms

We also provide software-based simulation results using our modified, to be supported by Linux kernel 4, version of BadgerTrap TLB miss instrumentation tool [24]. We find that DPART's TLB miss rates are negligible for most size configurations. This results in vastly outperforming conventional TLB and the THP mechanism [10] and behaving similarly to RMM [35], while being set associative, in contrast to RMM's Range TLB which is limited by full associativity and as a result lack of scalability.

2.3 Document Outline

This section denotes the organization of the rest of this document. Chapter 3 covers the background needed to fully understand the concepts presented in following chapters. Chapter 4 is the core part of this thesis analyzing DPART. Through this process, a two page size case elucidating our approach is presented, which is followed by our full system accompanied by its respective software mechanisms, TLB and page table. In the end of this chapter, two optimizations are provided boosting the performance of our method. Chapter 5 presents the results of our experiments while previously providing their methodology. This chapter ends with a section discussing DPART's performance and other probable superior attributes not illustrated by our simulations. A chapter devoted to related work is included in 6, enumerating and discussing other address translation schemes in current bibliography, and comparing them to DPART. Concluding the main-matter, chapter 7 presents future work and final concluding remarks. Finally, a bibliography chapter and an appendix are included in the end of the document.

Chapter 3

Background

This chapter covers the background needed to comprehend this work. Initially, a description of virtual memory is provided (3.1). In Section 3.2 the paging mechanism is explained, and afterwards the page table and TLB structures are examined in 3.3 and 3.4 respectively. Later-on in 3.5, we overview some Linux structures and operations, useful for our approach.

3.1 Virtual Memory Overview

Before virtual memory, it was the programmer's responsibility to manage the memory storage (referring to both primary and secondary memory) used by a program. To render it functionable, the programmer had to slice a program into blocks (overlays) and map them into memory. As a result, programmers needed to rewrite programs, and authors needed to rewrite documents, when the content of a program module, the capacity of a local memory, or the configuration of a network changed [22].

The first system implementing virtual memory was Atlas Computer at the University of Manchester in 1959 [37]. Virtual memory, as stated by Bhattacharjee and Lustig [17], is an idealized abstraction of the storage resources that are actually available on a given machine. Concretely, virtual memory is an address space, where a single process maps its data. Each process would ideally want to use abundant memory for its needs, without being concerned about other processes. With virtual memory, this illusion is provided, and a process can use virtual (logical) addresses without any restriction. However, to use a system's real resources, a mapping should be established between the physical memory and each process's virtual memory.

This mapping is called *address translation*, and is managed by the operating system

using both software and hardware means.

VirtualAddressSpace \rightarrow *PhysicalAddressSpace*

Each time a process accesses some data, the program code indicates which virtual address is needed, and the operating system provides the translation of this virtual address. The physical address obtained by this translation shows the location of the requested data in physical memory, enabling read and write operations to be performed to this data on behalf of the process.

All those translations though, consist of data themselves and consequently have to be also stored in physical memory. Hence, questions arise regarding how those translations are stored and accessed. The answers to those questions will be provided in the following sections.

3.2 Paging

In modern systems where many processes with large memory requirements are running simultaneously, having to store a translation for every memory address used by every program would not be affordable. To overcome this, paging is used. More specifically, a program's virtual address space is divided into blocks named *pages*. Now, a virtual page is mapped to a physical page, each consisting of contiguous addresses. Under this mechanism, only one translation per page is required.

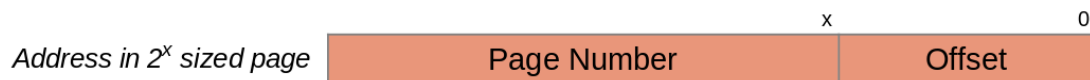


Figure 3.1: Virtual address components

A page containing 2^x addresses is named a 2^x *Byte page*, since each physical address contains one Byte of data. We will be also referring to them as 2^x pages. Pages are aligned in both virtual and physical address space. Hence, all the addresses residing in the same 2^x page differ only in the last x bits. This address part is called *page offset* and the rest is called *page number* or *page frame number* (Figure 3.1). Due to alignment, a physical address is obtained by the concatenation of the physical page number (received by address translation) and the page offset (invariant between virtual and physical space).

Most architectures use 4KB pages. Apart from the 4KB base page size, the x86-64 architecture also supports 2MB and 1GB pages with the mechanism of Transparent Huge

Pages (THP) [10, 8]. Additionally, other architectures support more sizes [44, 12, 46]. Large pages enable the existence of even fewer translations as they can accommodate even more addresses. As it will be shown in the following sections, this induces many benefits. On the other hand, large pages present higher internal fragmentation, meaning that there is a higher probability that a non-negligible part of the page will remain unused. In addition, for a fixed size of physical memory, the page size is inversely proportional to the number of pages frames that can be accommodated into the physical memory. As a result, pages have to compete for fewer entries. On the contrary, smaller pages are more versatile. By versatility we mean that smaller memory parts can be swapped from main memory, whereas in large pages a small and frequently accessed memory section has to be accompanied by the rest of the page it resides in.

3.3 Page Table

Having a larger number of virtual pages than physical pages is the basis for the illusion of an essentially unbounded amount of virtual memory [45]. Specifically in the x86-64 architecture, a virtual address consists of 48 bits, rendering the virtual address space able to map $2^{48}B = 248TB$ of data. Also, x86-64 now offers 57bit virtual addresses [2, 31] resulting in an 128PB virtual memory size. Even in page-level, having to store $248TB/4KB = 2^{36}$ translations would require more than the main memory of many of today's systems. However, processes generally use a tiny fraction of their virtual memory.

The data structure where all translations are stored is the *page table*. Leveraging the above observation, the page table manages to efficiently store the sparse virtual memory useful regions through a radix-tree structure. In this tree structure, the leaves contain the desired translations, and the nodes in the previous levels point to the appropriate subsequent node, finally leading to the translation needed. The traversal of the page table in order to find a translation is called *page walk*.

Pages are used to store any structure in physical memory, and the page table is not an exception. One single page is used to store the entries of the top level, named PGD (page global directory). The next levels are stored via a set of pages also named directories. As mentioned, a small portion of virtual memory has to be mapped, as a result, the only existent directories are the ones ultimately pointing to needed translations. On the contrary, at any level of the page table, the pointer to the next level can be null, indicating that there are no valid virtual addresses in that range [2].

Referring to 4KB pages (address's 12 LSBs used as page offset), the page number

part of a virtual address is divided into 4 9bit parts ($48 - 12 = 4 \cdot 9$). Those parts are used as indices to traverse through the 4 levels of a page table (in the case of 57bit addresses, the page table consists of 5 levels). In a 64bit architecture, each directory can contain up to $\frac{4KB}{64bits} = \frac{2^{12} \cdot 2^3 bits}{2^6 bits} = 2^9$ entries, explaining why 9 bits are used for indexing.

Figure 3.2 summarizes the aforementioned concepts, illustrating a page walk.

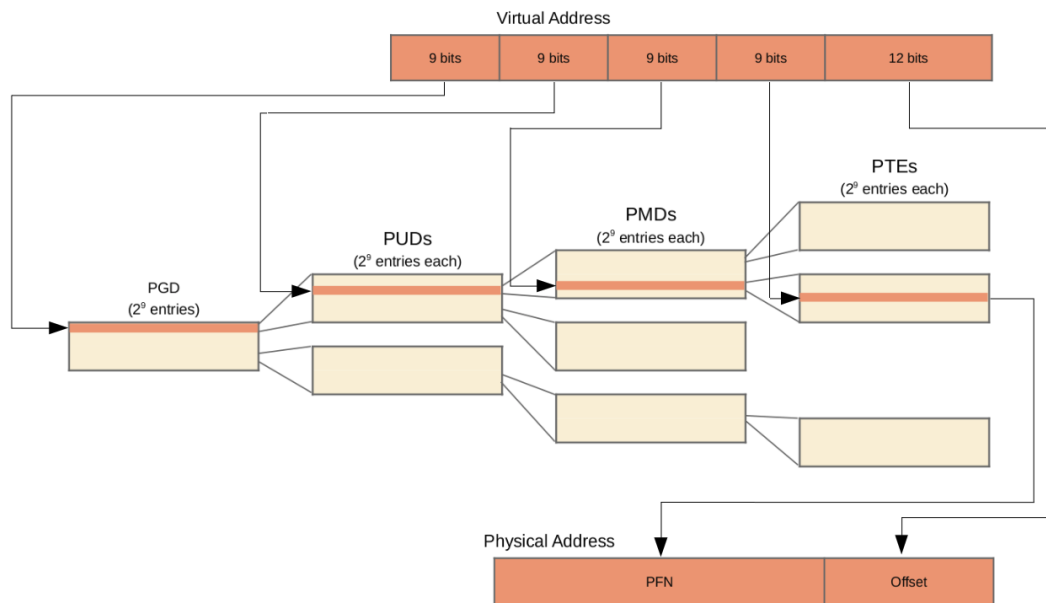


Figure 3.2: Page walk in a 4 level page table

Even if the page table manages to store translations using a little amount of memory, traversing its layers by accessing each time the appropriate directory in memory, may produce high latency cost. Future memory technologies will enable even larger memory capacities [27, 51, 34], where a 5 level page table will be inevitable, provoking up to 5 memory accesses and even more latency. Many hardware features are implemented in order to reduce the cost of a page walk, such as hardware walkers, translation caches and complex TLB structures [42]. Being one of the most fundamental improvisations in address translation, the TLB is analyzed in the next section.

3.4 The TLB Mechanism

The Translation Lookup Buffer (TLB) is an address translation cache. Being a hardware part of a processor core's memory-management unit (MMU) and small in size, the TLB

can access its data in one CPU cycle [45].

As in memory cache, to reduce search time a common TLB operates under index-based search, also supporting associativity. The virtual page number part of an address is now further split into the tag and the index parts, as shown in Figure 3.3. The index part shows the candidate set of the TLB that may contain the desired translation. To establish this, the tags contained in this set are compared to the address's tag.

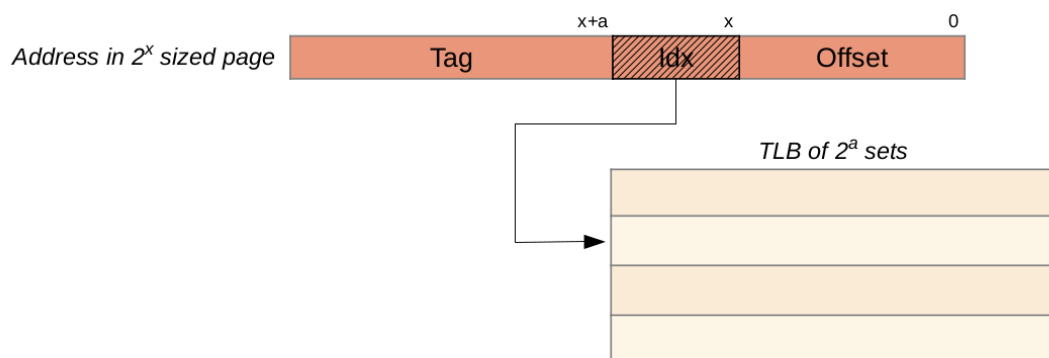


Figure 3.3: Virtual address TLB components

Taking both the TLB and the page table into account, assuming that all translations are stored in main memory, the total cost of address translation is the following:

$$\begin{aligned} \text{cost}(\text{addr_tr}) &= \text{cost}(\text{TLBlookup}) \cdot \mathbb{P}[\text{TLBhit}] + \\ &\quad (\text{cost}(\text{TLBlookup}) + \text{cost}(\text{page_walk})) \cdot \mathbb{P}[\text{TLBmiss}] \\ \Rightarrow \text{cost}(\text{addr_tr}) &= \text{cost}(\text{TLBlookup}) + \text{cost}(\text{page_walk}) \cdot \mathbb{P}[\text{TLBmiss}] \end{aligned}$$

Effort to minimize this cost are pertaining to two orthogonal approaches

- reducing $\text{cost}(\text{page_walk})$
- reducing $\mathbb{P}[\text{TLBmiss}]$ while maintaining $\text{cost}(\text{TLBlookup})$ relatively small

3.5 Virtual Memory Allocation

Each process's virtual memory possesses a number of characteristic memory regions that are mapped either in the top or the bottom of the virtual address space, such as executable code, static data, the heap (controlled by the program break) and the program's

stack. The bounds of all those quantities are stored in the memory descriptor of a process (structure `mm_struct`). The value of those bounds, indicating the growth of each corresponding memory region are illustrated for an example benchmark in Figure 3.4. The scale of the quantities presented in the figure should be considered. Regions in low addresses manage to grow until address `0x3000000`, occupying a total of 48MBs, whereas high addresses regions subsume into less than 8kB. As it can be inferred, the free space in-between is huge.

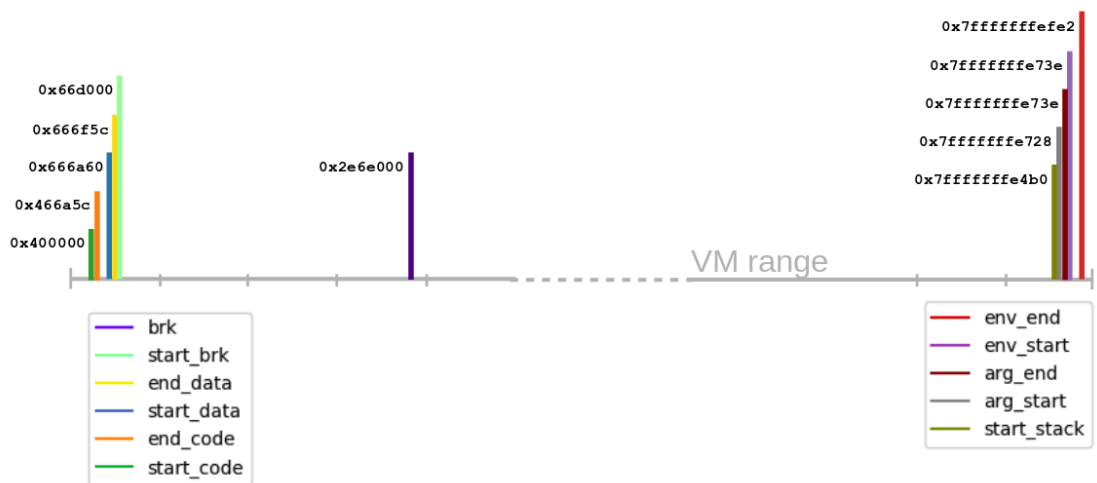


Figure 3.4: Memory regions bounds in GemsFDTD benchmark

What lies in-between is the memory mapping segment. This space contains regions allocated by the `mmap` system call. In general, for allocations greater than or equal to the limit specified (in bytes) by `M_MMAP_THRESHOLD` that can't be satisfied from the free list, the memory-allocation functions employ `mmap` instead of increasing the program break. Allocating memory using `mmap` has the significant advantage that the allocated memory blocks can always be independently released back to the system. (By contrast, the heap can be trimmed only if memory is freed at the top end) [4].

Pertaining to the memory mapping segment, two other interesting fields of a process's memory descriptor are the `mmap` list and the `mm_rb` red-black tree. Those two fields are two distinct data structures containing the same thing; all the memory areas in the memory mapping segment. The former stores them in a linked list, whereas the latter stores them in a red-black tree. Although the kernel would normally avoid the extra baggage of using two data structures to organize the same data, the redundancy comes in handy here. The `mmap` data structure, as a linked list, allows for simple and efficient

traversing of all elements. On the other hand, the `mm_rb` data structure, as a red-black tree, is more suitable to searching for a given element [41].

A red-black tree is the appropriate data structure to use when it is needed to be modified frequently (a thorough examination is presented in Appendix A). In virtual memory areas management, insert operations accompany the commonly used `mmap` system call. Each time a new virtual memory area needs to be mapped, a search takes place in the memory descriptor's red-black tree field, looking for a suitable gap. According to the kernel configuration, this search occurs either by taking left or right nodes first under consideration. The above two design options result in bottom-up and top-down allocation respectively. More specifically, if not explicitly specified by providing an address preference (strict under the `MAP_FIXED` flag), the operating system allocates virtual space starting from lower addresses and growing to higher ones in bottom-up allocation. The contrary holds for top-down. After subsequent space has been found under the above logic, the memory descriptor's `mmap` list and red black tree can be updated. Maintaining one of those allocation policies consistently results in the creation of large contiguous virtual address space regions enhancing the benefits of large pages.

Chapter 4

The DPART Scheme

This chapter describes our proposed mechanism, DPART (**D**eterministically Indexed Address Translation with Multiple Page Sizes via **P**artitioned Address Space). In the beginning (4.1), the characteristics of an ideal hypothetical address translation scheme are presented. Afterwards, after introducing a simplified scheme implementing the idea behind DPART (4.2), we present the details of our technique (4.3). Last but not least, in 4.4 two optimizations are exhibited that complete the implementation of DPART.

4.1 Description of an Ideal Address Translation Scheme

In general, to achieve efficient address translation, a mechanism should ideally possess the following characteristics:

- Multiple page sizes
- Simple, straightforward internal mechanisms
- Index-based TLB lookup
- Proper index bits selection
- Determinism and correctness
- Uniformity

Multiple page sizes are vastly important as they optimize the space needed to encode the virtual memory to physical memory mapping. The utilization of only a small sized page would result in redundant translations in the mapping of a large virtual memory

area, and on the contrary side, using only a large sized page would result in big physical memory space waste when having to translate small memory areas. Consequently, it is substantial that an address translation scheme be able to adapt to the size of the memory amount needed to be used each time. To address this problem, an efficient mechanism should provide a wide page size granularity. The more the sizes, the better translations will adapt to the virtual memory areas length variety.

Much of the related work presents complicated mechanisms, existing either to compute a translation, or to support the implemented mechanism while often running in the background. Some of those mechanisms include prediction logic [11, 19, 42, 43, 49], collection and processing of metadata [50, 52], or need to run searches on multiple different structures simultaneously [53]. It should also be noted that those mechanisms become more and more complicated or even inaccurate when using more page sizes. To combine high performance and energy efficiency, an ideal scheme should only consist of simple, straightforward mechanisms that would neither impose high latencies, nor provoke high energy consumption.

A TLB adhering to a powerful address translation scheme should also support index-based search. Indexing reduces the TLB search domain only to a specific subset of entries. This results in a significant reduction of the search overhead. If index-based search cannot be supported, then we are forced to rely on fully associative TLBs. Fully associative structures' average search time is proportional to their size. Hence they are not scalable, and limited only to few entries that may not be able to support demanding tasks efficiently.

Again pertaining to indexing, it is crucial that the index bits be juxtaposing to the offset bits. If not, then either the index and offset part of a virtual address will share common bits, or extra bits will exist in-between. Both of the preceding induce problems degrading address translation performance. In the case that index and offset parts have bits in common, then the same page will be mapped in multiple TLB sets. This is practically equivalent to using a smaller page size whose offset matches the LSBs remaining before the beginning of the index part, as a single translation in the TLB embodies only mappings that differ in solely those remaining bits. On the contrary, if the index part of an address is selected to be placed towards the address's MSBs, then the tag part shortens. A linear tag size reduction results in an exponential conflict increase. In a contiguous address space, for every bit the index part is shifted rightwards, the number of addresses mapped into the same set doubles, confining the benefits of small sets associativity.

By determinism and correctness, we mean that if an address translation logic deter-

mines that a translation is located in a specific TLB set, then this should be accurate. Speculation mechanisms, where determinism is absent have non-negligible overheads in a misprediction scenario [11, 19, 49]. Even if those cases may occur rarely due to high prediction accuracy, completely distinguishing them is definitely an improvisation. Also, as already denoted, prediction accuracy will heavily deteriorate when having to chose between more classes (i.e. page sizes). As a result, determinism is also essential for scalability. In addition, devastating security leaks may be induced by speculative execution of memory operations as it has already been done in Spectre [38] and Meltdown [40]. In order to be avoided, those dangers should be carefully handled, sometimes by stalling execution or furtherly validating a translation before committing it [43].

At last, the distribution of translations over the TLB entries should approach a uniform distribution. If on the contrary many translations are accumulated in an entries subset, two bad things can happen. Firstly, in the case that many translations are hosted by some few, specific entries, if the number of translations exceeds the number of those entries, some of those translations will be evicted. In an extreme case, lets assume that N translations are mapped in a set of $N - 1$ TLB entries, and the corresponding virtual addresses of those translations are accessed cyclically. This would result in a 0% hit rate. To solve this problem, we should augment the number of candidate entries where those translations can be hosted. Thus we should reduce accumulation by increasing uniformity. Secondly, in a non-uniform distribution, a portion of the TLB structure may be rarely utilized. Since TLB is a costful resource, underusing it is not affordable.

To achieve all of the above six features with little tradeoff, would result in a flawless scheme. Of course, high performance may be achieved by satisfying most but not all of those characteristics and by mitigating the effect of the rest. Is it possible though to create a scheme fulfilling all of them? The answer is affirmative, as DPART does so.

4.2 Two Page Sizes Initiative

Let us first formulate the problem which renders multiple page size translation difficult. For the sake of simplicity, we will be referring to a scheme supporting two page sizes only, 4KB and 2M.

We wish to equip our address translation scheme with both multiple page sizes support and index-based TLB lookup. As shown in Section 3.4, depending on the size of the page that an address belongs to, a different number of bits is used as page offset. Thus, the index part of the address, starting directly after the offset part, is located in different positions for different page sizes. Figure 4.1 depicts this difference.

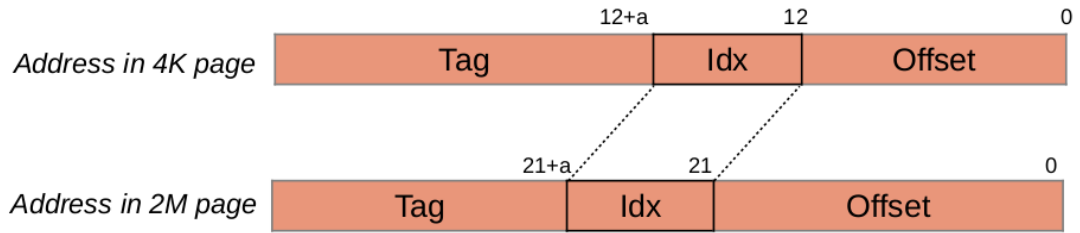


Figure 4.1: Index bits position in different page sizes

It is now cogent that with the absence of a-priori knowledge about the page size that has been used, it is impossible to determine which bits should be used for index. At this point though, one should question how do we determine other quantities characterizing a TLB entry, or general entries in computer architecture structures. The answer is by special bits such as valid, dirty, etc. However, the approach to introduce a new special bit invokes several challenges, some of them concerning when and how this bit's value will be determined.

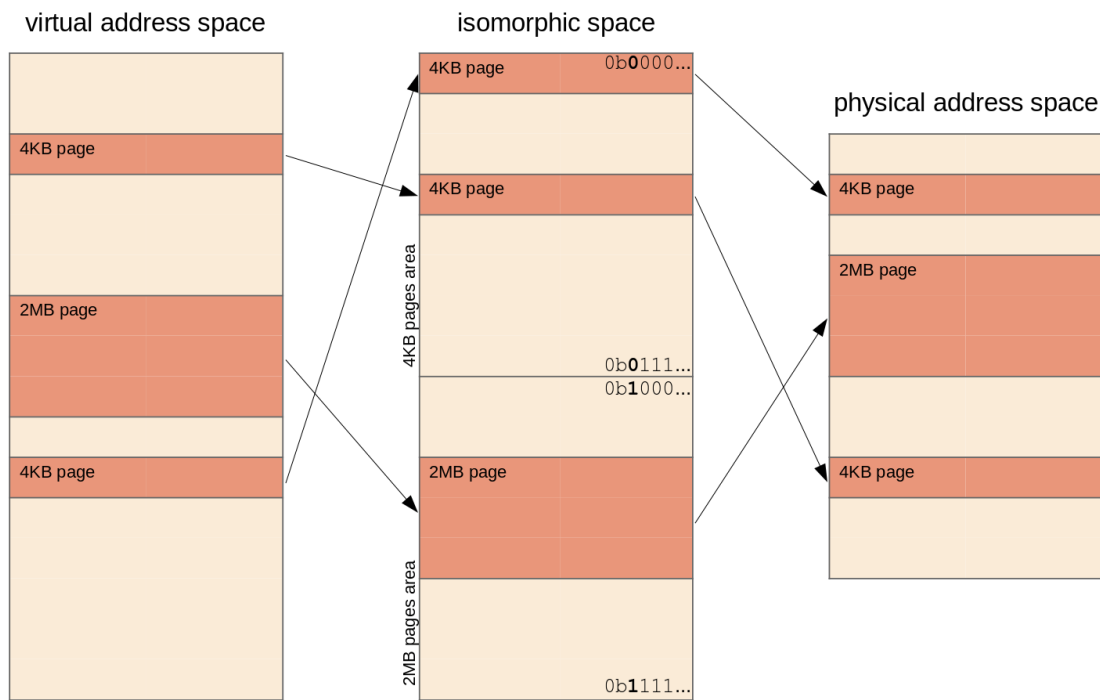


Figure 4.2: Hypothetical isomorphic layer between virtual and physical address space

A much simpler approach is to make this special bit a part of the virtual address.

To imagine how this would work, one could construct an isomorphism of the virtual address space (Figure 4.2). At the isomorphic space, addresses corresponding to 4KB pages will be mapped at the upper space's half and 2MB page addresses will reside in the lower half. Thus, all addresses belonging to 4KB and 2MB pages will start with a 0 or 1 respectively. Under this approach, reading the address's MSB will reveal the page size that was used for its translation.

In Section 4.3.1, we will see that instead of creating a second level of virtual memory, we will apply the aforementioned rule to the virtual address space itself.

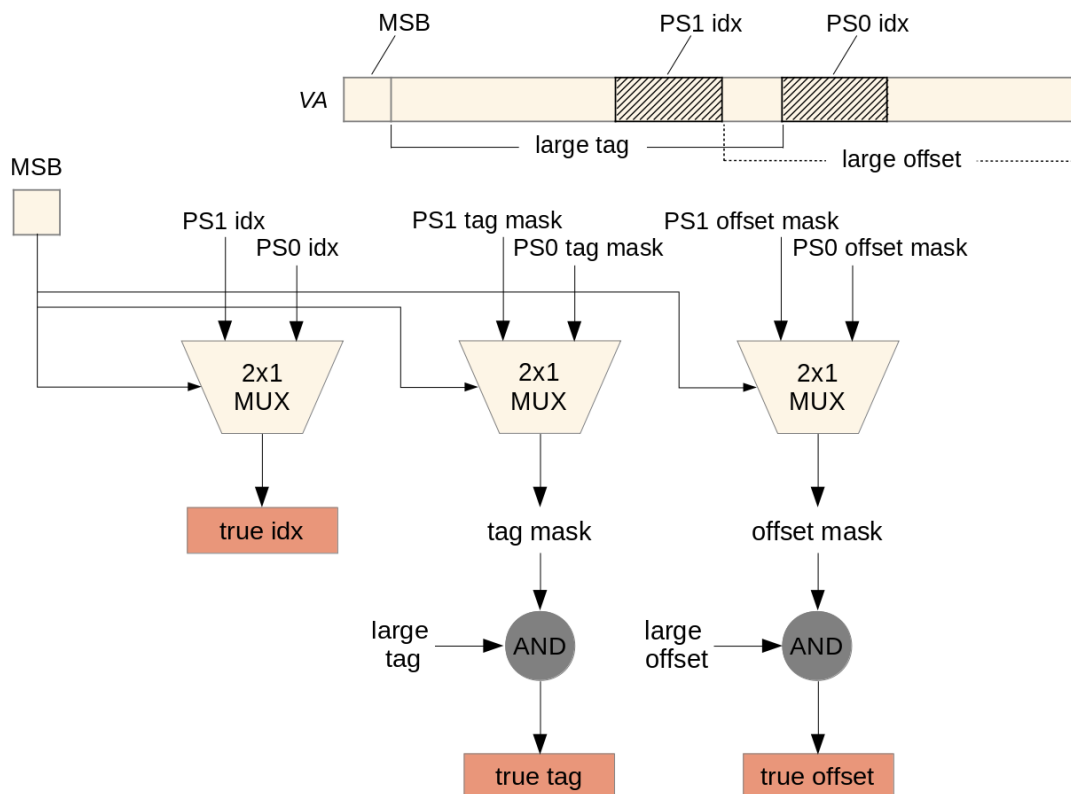


Figure 4.3: Address tag, index and offset identification with 2 supported page sizes

Now that we have a virtual address whose MSB denotes the page size that should be used, the TLB mechanism should be straight-forward. Figure 4.3 shows the detailed hardware mechanisms. The MSB is used as the select line at 3 multiplexers. The one responsible to compute the true index part of the address takes as input the two possible index parts (for 4KB and 2MB pages). Those quantities (illustrated with hatch pattern

and named P*S*i idx on Figure 4.3) are defined deterministically and universally for any address.

As for the other 2 multiplexers, they contribute to the computation of the address true tag and offset. The input lines in each of them are constant masks, which can be constructed at hardware level. The masks for each page size are shown in Figure 4.4. After the multiplexers determine which masks should be used, they are passed to AND gates, alongside with what is shown as large tag and large offset. Those quantities are defined by the largest tag and offset possible for both page sizes. It is clear that the large tag corresponds to the tag that should be used for 4KB pages and contains the 2MB page tag, and the large offset corresponds to the offset that should be used for 2MB pages and contains the 4KB smaller offset. After the application of the AND gates, the true tag and true offset are output.

It may be referred that one instead could use the appropriate bits corresponding to the various page size's tags and offsets directly to the multiplexers' gates. This though would require extravagant hardware support for more page sizes. The presented technique instead, uses constant inputs to the multiplexers, and needs two extra set of bits from the virtual address, the large tag and offset. Of course, as in the case of the index bits, those virtual address sections are deterministic and universal for all addresses.

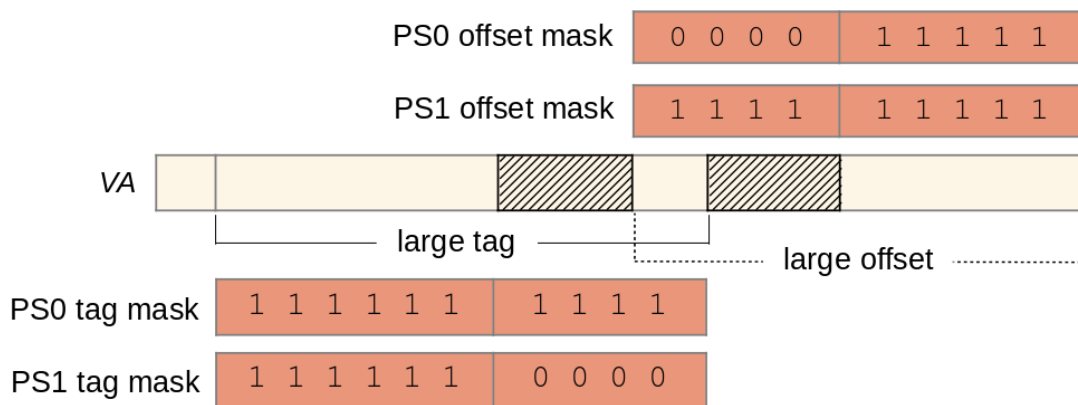


Figure 4.4: Tag and offset masks

Now that the appropriate index, tag, and offset for each page size are determined, a TLB lookup can take place with no further modifications.

4.3 DPART

DPART is the generalization of the previous scheme for arbitrarily many page sizes. Hence, it can be inferred that to identify the appropriate page size, we will be using more MSBs this time. Other than that, the previous logic remains the same; all addresses which belong to a specific page size, will be mapped at a specific virtual address space segment, characteristic of this page size. To elucidate how we manage to establish this MSBs and page size identification connection, we will initially delve into the software part of our technique.

4.3.1 Virtual Address Space Partitioning

As it was stated in Section 4.2, our goal is to force virtual addresses to reside in specific regions of the virtual address space depending on the page they should be mapped with. Those virtual address space regions will be called *partitions*. More specifically, we will be partitioning the virtual address space into 2^n equal sized partitions, where n denotes the number of virtual address MSBs we will be using for our identification. Obviously, in such a scheme, up to 2^n page sizes can be supported. For instance, figure 4.5 illustrates the case of $n = 2$.

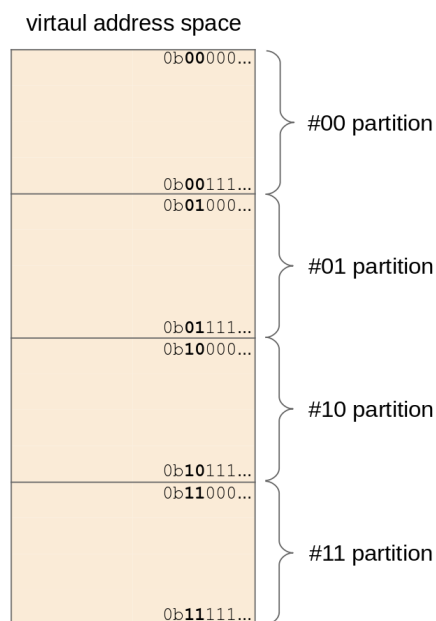


Figure 4.5: Virtual address space partition

We should denote that despite any differences, the idea of partitioning the virtual address space and using an address's MSBs for proper allocation is observed in bibliography by Basu in his PhD thesis [14] and in another instance by Keppel and Pham in a 2019's patent [36]. His approach will later be presented in the related work Chapter 6.

At this point, we will describe how we manage to consistently map and retain memory in the desired partition. As explained in Section 3.5, at the mapping of a new virtual memory area, the Linux kernel uses mechanisms to search virtual memory, and find an appropriate gap to map this area. This procedure takes place for example in `mmap`, `mremap` system calls and shared memory operations. The domain of the predescribed search is limited by the `low_limit`, `high_limit` fields of a `vm_unmapped_area_info` structure. Those limits are architecture specific and in an unmodified Linux kernel their values are often set in the extrema of the allowed virtual space given to a process (i.e. `TASK_UNMAPPED_BASE`, `TASK_SIZE`), depending on the kernel configuration.

Going into the details of this unmapped area search, a gap is defined by the the start of a virtual memory area (`vma`) and the end of its preceding `vma` (`vma->vma_prev`). To make a selected gap compatible for an assignment of a `length`, the following conditions must hold:

$$\begin{aligned} gap_start &\leq info \rightarrow high_limit - length \\ gap_end &> info \rightarrow low_limit + length \\ gap_end - gap_start &\geq length \end{aligned}$$

where `gap_start`, `gap_end` denote the `vma->vma_prev->end` and `vma->start` respectively.

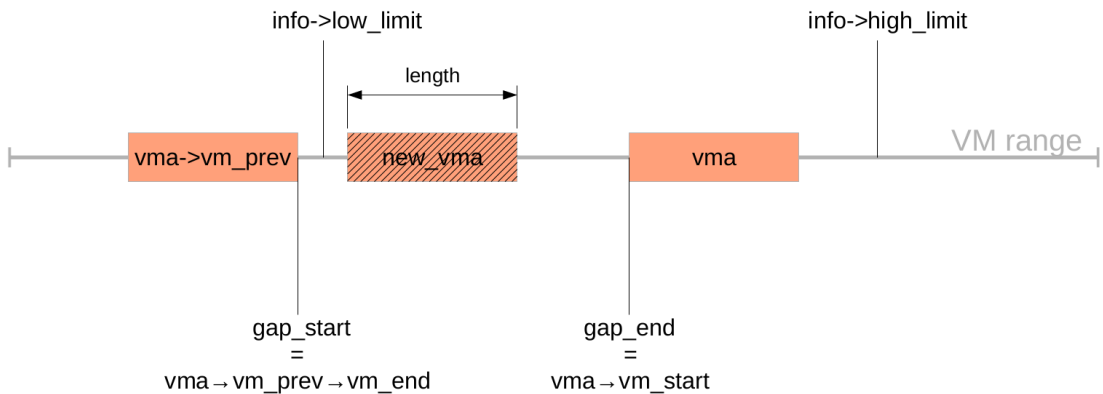


Figure 4.6: Virtual memory area search

At first glance, those inequalities may seem counterintuitive, as the gap's lower bound is compared with the search domain's upper bound of the first inequality and *vica verca*. A thorougher examination though, shows that those inequalities together are sufficient to render a gap able to accommodate a *length* sized new area. The first two inequalities assure that the allowed search limits enclose the desired gap with a margin of *length*, and the third inequality ensures that the area to be mapped fits.

To ensure that a vma will be mapped in the partition of our preference, we first modify the `vm_unmapped_area_info` struct's limits to make them match the partition's bounds. Secondly, we need the mapped virtual memory areas to be aligned based on the page size to be used. To achieve this, we include alignment operations in some parts of the code, and slightly modify the presented inequalities. At this procedure, we neither lose any of the optimizations of the operating system implementation (for example, virtual memory areas are still likely to be mapped continuously), nor cancel any of its policies, such as top-down assignment. We instead use any policy specifically in the partition we are working on, instead of the whole virtual memory.

4.3.2 Usage of Multiple Page Sizes

Now that we have established a mechanism that maps virtual memory areas to the desired partition, we are free to utilize them.

We will be using the first and last partitions for 4KB pages. We are using the basic page size for both of those partitions, because in current systems the low addresses are used for the text segment, data segment, bss segment and heap, and high addresses are used for the stack [47]. The above makes using two partitions equivalent to using no partitions, as they would both refer to 4KB pages. As a result, we are never using a single MSB in our scheme. We should now denote that by virtual address space, we mean the user space, defined by 47 bits (a virtual address consists of 48 bits and the second half of this address space is used by the kernel) [3]. In 5-level paging this becomes 56 bits [2, 31], but this case is beyond the focus of this thesis.

It may be stated that devoting a certain number of bits within the address reduces the virtual memory available for each page size, while before all the address space was available for all sizes. This though, should not be considered an issue. In our approach, we will be using up to 5 bits. From now on, we will be referring to these bits as *partition bits*. The size of each virtual memory partition remains huge, as it is shown in Table 4.1. Zero partition bits pertain to an unmodified system where the whole user space's size is indicated.

At this point, we present which page size will be representing each partition. Table 4.2

shows all the page sizes that will be used and Figure 4.7 depicts how those page sizes are distributed over the partitions. It may be observed that at the case of 5 bits, since going over 32GB pages is extravagant, we repeat the utilization of the same page size in some partitions. This is done in large pages, as those have the highest probability of exceeding their 4TB partition, even if such a scenario is still nearly impossible.

| Partition bits | Partition size (4-level PT) | Partition size (5-level PT) |
|----------------|-----------------------------|-----------------------------|
| 0 | 128TB | 64PB |
| 2 | 32TB | 16PB |
| 3 | 16TB | 8PB |
| 4 | 8TB | 4PB |
| 5 | 4TB | 2PB |

Table 4.1: Partition sizes for 4-level and 5-level page tables

| Partition bits | Page sizes |
|----------------|--|
| 0 | 4K pages only |
| 2 | 4K, 2M, 1G pages |
| 3 | 4K, 32K, 256K, 2M, 16M, 128M, 1G pages |
| 4 | 4K, 8K, 32K, 64K, 256K, 512K, 2M, 4M, 16M, 32M, 128M, 256M, 1G, 2G, 8G pages |
| 5 | all power of 2 page sizes from 4K to 32G |

Table 4.2: Available page sizes under each partitioning scheme

Now, one should question what is the criterion of choosing a page size during the allocation of some memory. We are introducing three policies for this purpose.

- LOWER
- CLOSER
- UPPER

| 2 partition bits | 3 partition bits | 4 partition bits | 5 partition bits |
|------------------|------------------|------------------|------------------|
| 4k | 4k | 4k | 4k |
| | 32k | 8k | 8k |
| | 256k | 32k | 16k |
| | 2M | 64k | 32k |
| 2M | 2M | 256k | 64k |
| | 2M | 512k | 128k |
| | 16M | 2M | 256k |
| | 1G | 4M | 512k |
| 1G | 16M | 16M | 1M |
| | 128M | 32M | 2M |
| | 1G | 128M | 4M |
| | 4k | 256M | 8M |
| 4k | 4k | 1G | 16M |
| | | 2G | 32M |
| | | 8G | 64M |
| | | 4k | 128M |
| | | 4k | 256M |
| | | | 512M |
| | | | 1G |
| | | | 1G |
| | | | 2G |
| | | | 2G |
| | | | 4G |
| | | | 4G |
| | | | 8G |
| | | | 8G |
| | | | 16G |
| | | | 16G |
| | | | 32G |
| | | | 32G |
| | | | 32G |
| | | | 4k |

Figure 4.7: Page size utilized for each partition

For a virtual memory area of size $length$, in those policies we are picking pages from the directly lower, closer and upper page size respectively. Providing an exact definition of them, if $avail_page_sizes$ is the set of the sizes denoted in Table 4.2, then:

$$\begin{aligned}
 page_size_{lower}(length) &= \sup_{ps \in avail_page_sizes} \{ps \mid ps \leq length\} \\
 page_size_{closer}(length) &= \arg \min_{ps \in avail_page_sizes} |\log_2(ps) - \log_2(length)| \\
 page_size_{upper}(length) &= \inf_{ps \in avail_page_sizes} \{ps \mid ps \geq length\}
 \end{aligned}$$

Ostensibly, LOWER is a conservative policy that would use more pages and reduce space waste. UPPER is a more aggressive policy aiming to minimize the number of

pages used paying the tradeoff of redundancy. Obviously, CLOSER tries to incorporate the advantages of both LOWER and UPPER by providing a solution in the middle.

Furthermore, an example of the following policies is provided in Table 4.3 from which advantages of each of them can be inferred.

| Length | Policy | | |
|--------|--------|--------|-------|
| | LOWER | CLOSER | UPPER |
| 150kB | 32kB | 256kB | 256kB |
| 250kB | 32kB | 256kB | 256kB |
| 35kB | 32kB | 256kB | 32kB |

Table 4.3: Page assignment examples for each policy under 3 bits partitioning

One may notice that a mechanism strictly allocating contiguous virtual memory areas to the appropriate partition may include implementation gaps concerning memory areas' size modification through a process's runtime. In detail, the `mremap` system call may expand (or shrink) an existing memory mapping [6]. The behavior of `mremap` is determined by the flag `MREMAP_MAYMOVE`, the existence of which allows the operating system to move the whole mapping in another part of virtual memory. Under this flag, our logic can be implemented without any constraints. However, under its absence the new virtual memory area size may not adhere to its memory partition page size indication.

To confront this issue, we loosen the strictness of our model, allowing the existence of memory regions formed by those circumstances. This avoids costly memory areas movements and simplifies our approach. However, the negative consequences of this selection, is that now more small pages may be used for an expanded memory area, and conversely, larger pages may be used for shrunk memory areas. We observe that implications of those advantages to performance are minor. Even if the remapped area doubles in size, only a few extra pages should be used (at most one in CLOSER and UPPER policies). The shrinking case is very rare, but still it may only present as much page internal fragmentation as the memory area size reduction. To furtherly support our design choice, in large virtual memory areas size modification cases, the `MREMAP_MAYMOVE` is present, preventing any unwanted results.

4.3.3 DPART TLB

At this subsection, details are going to be provided regarding the TLB that accompanies our scheme.

The structure of the TLB entries remain exactly the same as in modern architectures hardware. Specifically, in Figure 4.8 the form of the TLB entries corresponding to a base size page and a larger one, is shown. To achieve this uniformity, in the case of a large page, we add zeroes to the end of its (smaller) tag, until it matches in bit size a 4KB page tag. It is easy to show that this transformation preserves the uniqueness of tags between different pages.

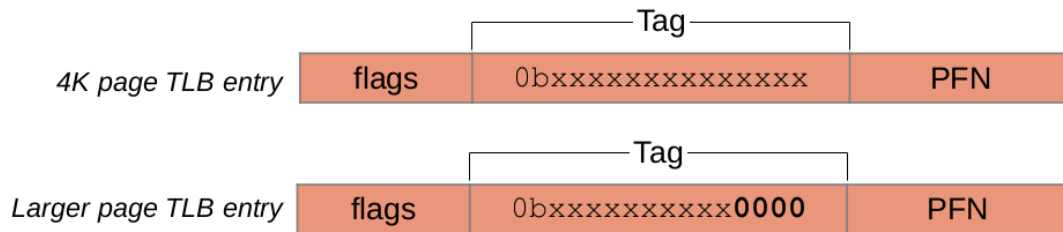


Figure 4.8: TLB entries structure

In detail, let $addr1, addr2$ be two addresses that have been translated via different pages and their translations have been found in the same TLB set. The $nMSBs$ function outputs the MSBs of a portion whose number is equal to the quantity of partition bits. We will prove that having the same tag is impossible, by contradiction. The case where the addresses respective page sizes are equal is omitted as trivial.

Suppose $page_size(addr1) \neq page_size(addr2)$

$$\begin{aligned}
 &tag(addr1) = tag(addr2) \\
 \Rightarrow &nMSBs(tag(addr1)) = nMSBs(tag(addr2)) \\
 &\xrightarrow{n < |tag|} nMSBs(addr1) = nMSBs(addr2) \\
 &\xrightarrow{\text{DPART logic}} page_size(addr1) = page_size(addr2), \text{ a contradiction}
 \end{aligned}$$

As a result, finding a tag in a specific TLB set, deterministically defines its underlying virtual page.

The mechanism that is used to identify the tag, index and offset components of a

virtual address, is the same as the one described in Section 4.2. Its general case for n partition bits is shown in Figure 4.9

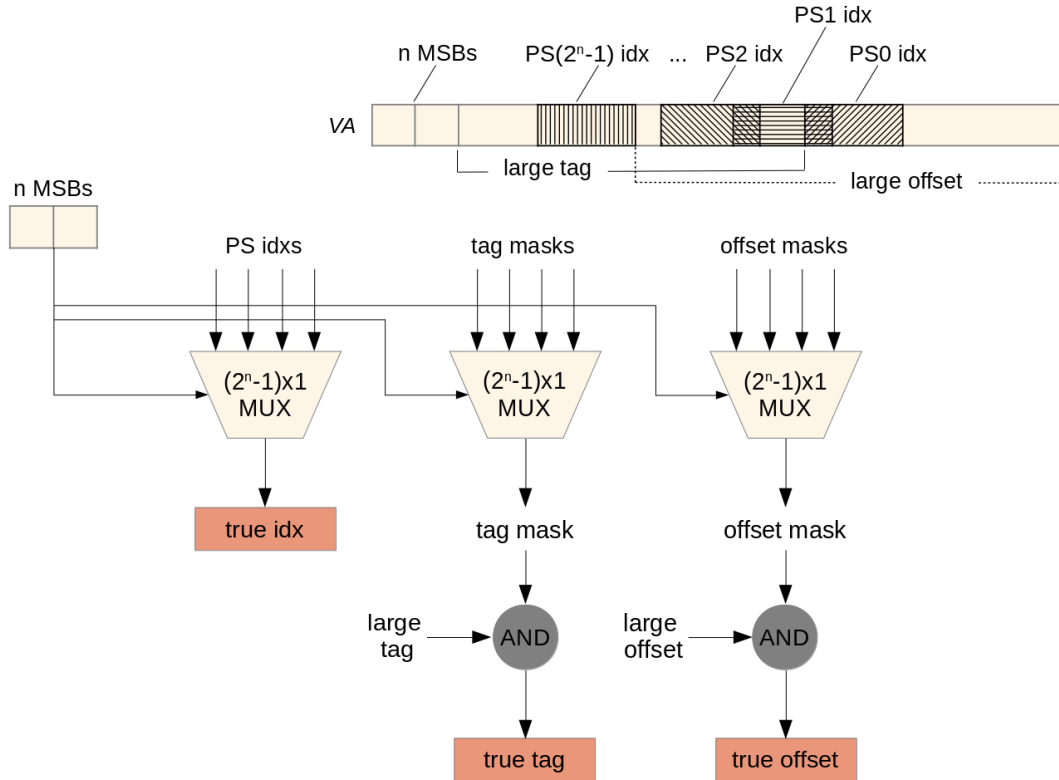


Figure 4.9: Address tag, index and offset identification for $2^n - 1$ supported page sizes

After having identified the true tag and index, a TLB lookup takes place (Figure 4.10). If the required tag is not found in the indexed set, we encounter a TLB miss, exactly like in the conventional TLB. On a hit, we consider two cases on whether the physical address space is aligned at the used page level or not. In the case where the physical memory alignment is in concordance with the virtual memory alignment, to obtain the correct physical address, we only have to concatenate the page frame number from the TLB with the true offset. On the contrary, if physical memory is only aligned at base page size level, then we have to perform an addition operation between the true offset and the first physical address of the desired page. This address is simply found by zero-padding the page frame number stored in the TLB with as many zeroes as the number of bits the smallest offset has.

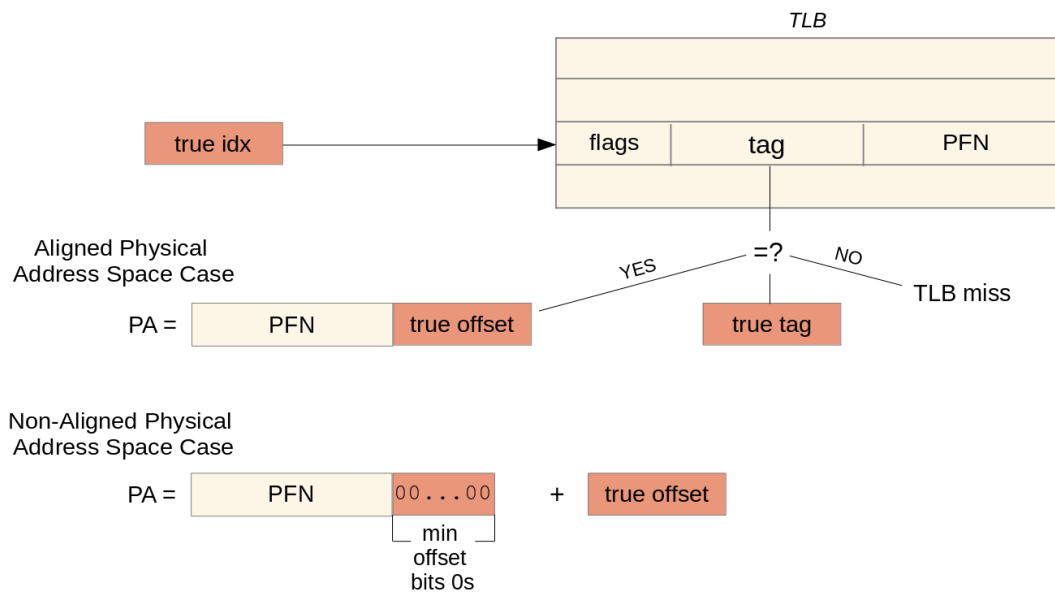


Figure 4.10: TLB mechanism

Physical memory management is not implemented in this thesis, and is included in the future work. However, it is estimated that due to the shortage of physical memory (contrasting with the abundance of virtual memory), alignment requirement at the level of large pages may result in inability to find a suitable contiguous physical address space, and thus smaller pages should be used. On the contrary, if a hardware addition is to be implemented, as described in the non-alignment case, then its overhead should be evaluated.

4.3.4 DPART Page Table

At this point, the modifications on the page table implementation are presented. In Section 3.3 the page table was introduced. In Figure 4.11 we remind its radix tree structure, with each directory consisting of 2^9 entries.

As also stated in the Background chapter, a virtual address is divided into components that are used as indices for the page walk and as offset after a page table entry has been found. Since each directory includes 2^9 entries, 9 bits are used for indexing. Figure 4.12 shows the bits that are used for those purposes, for each page size currently supported in Linux.

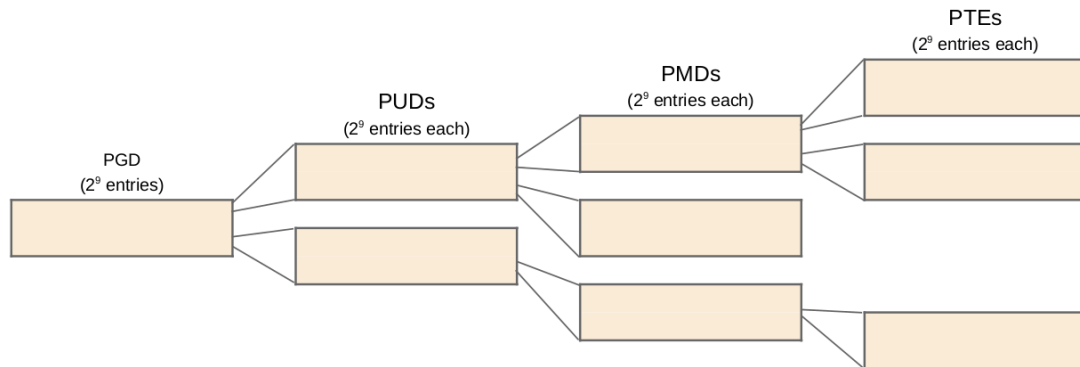


Figure 4.11: Part of page table

Figure 4.12 also shows how 48bit addresses (user and kernel space combined), and the support of 4KB, 2MB and 1GB page sizes fit perfectly together. Removing the offset part from an address (which is determined by the page size), the remaining address bits are always a multiple of 9, same as the number of bits used as indices during a page walk.

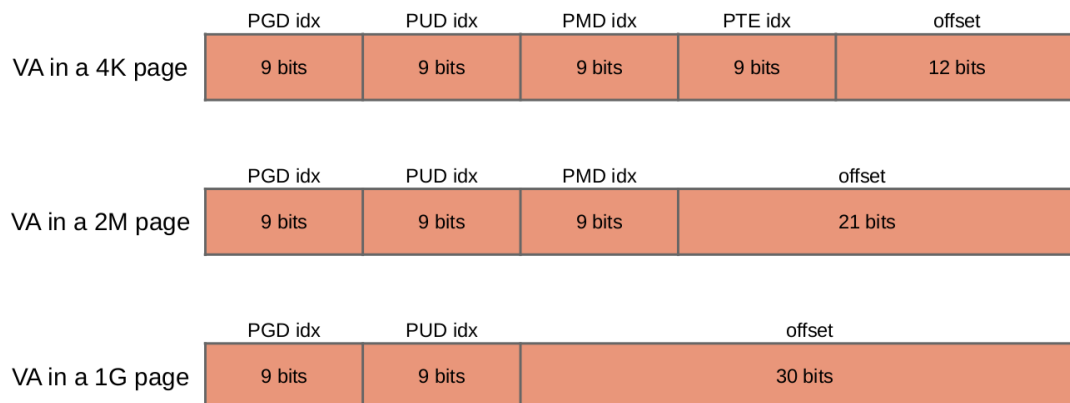


Figure 4.12: Addresses page table components

What happens though with the extra page sizes that DPART supports? There are three different approaches.

The simpler one, requiring no OS or HW modifications, is to treat all addresses as if they were translated by 4KB pages. This is possible, since the page size is not an attribute of an address, but an attribute of the mechanisms that manipulates that address. As a result, the TLB can still exploit all the advantages of our scheme. In a

TLB miss, a page walk should occur (in 4KB page logic), the physical address would be loaded, and then, in possibly larger page logic (according to the DPART TLB), the page frame number should be extracted and stored in the TLB. Even though the page frame number in the TLB differs from the page table entry in their bits quantity, this does not consist of a problem.

Performance-wise, as shown in Section 3.4, for constant TLB lookup time, the overhead of address translation is linear to the cost of a page walk multiplied by the TLB miss rate (assuming that the probability of a page fault is negligible).

$$\text{cost}(\text{addr_tr}) = \text{cost}(\text{TLBlookup}) + \text{cost}(\text{page_walk}) \cdot \mathbb{P}[\text{TLBmiss}]$$

The adoption of 4KB pages forces us in modern systems to suffer 4 memory accesses (or 5 in 5-level paging [2, 31]) in a page table traversal. As a result, to optimize address translation, effort is given in reducing the TLB miss rate, rendering TLB the primary factor virtual memory's performance. Thus, leaving the page table system as it is, would not be considered harmful. Obviously, this does not imply that optimizing it would not be beneficial. Here is where the second approach comes.

So, considering the second approach, it expands the idea from the previous method. What it proposes is to treat addresses as 4KB, 2MB or 1GB. As it was previously explained, treating an address with different page size mechanisms each time is perfectly acceptable. So after identifying the page size determined by DPART using the address's MSBs, what we do is to select from the 4KB, 2MB, 1GB sizes the largest one not exceeding the identified page size. Afterwards, we initialize a page walk as if the address was mapped with that selected size.

This technique takes advantage of the incorporation of larger pages in the page table mechanism. First of all, pages with size of at least 2MB will result in fewer memory accesses, as they would need to access 3 or 2 directories instead of 4. Secondly, less size is needed to store the page table. Take for example a single 1GB page translation. Its representation under 4KB pages would require $2^{30-12} = 2^{18}$ PTEs. The only drawback is that during TLB update, we should provide the appropriate PFN by extracting the correct number of bits from the physical address according to the page size specified by DPART logic, as we also had to do in the previous method.

The steps of this approach are:

- Page size identification using the address's MSBs
- Select the largest size from 4KB, 2MB, 1GB, not exceeding the identified page size
- Extraction of index and offset components according to the selected page size

- Page walk according to indices previously obtained
- Concatenation of PTE with offset to receive physical address
- Computation of PFN according to the identified page size
- TLB update with PFN

The third approach, even being trickier, furtherly improves the previous one. Under this solution, we treat all addresses under the size indicated by DPART. Obviously, this solution fully leverages the advantages of large pages, also resulting in shorter page walks and smaller page table space in memory. The challenge of this method refers to page sizes whose offset is not in the form of $48 - 9k$. Those pages should be examined meticulously. Figure 4.13 shows one of them.

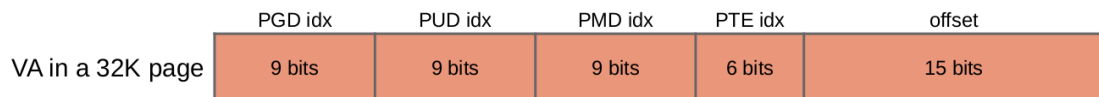


Figure 4.13: 32KB page address page table components

In this example involving an address that belongs to a 32KB page, 6 bits have to be used for the last page table level indexing. As a result, the PTE chunks will accommodate 2^6 instead of 2^9 entries. A page table under this scheme is shown in Figure 4.14. The consequence of this method is leaving some empty space in certain pages used to store the last level of the page table tree structure. At the extreme case, some of these pages may have only 1 entry. This would be caused for example in 1MB pages (with their page offset being equal to 20), as $48 = 9 + 9 + 9 + 1 + 20$.

However, instead of this being a drawback, it gives us space for further optimization. Specifically, the pages storing the last level of the page walk which contain extra space, can be combined into fewer pages. For example, two PTE chunks referring to 8KB pages having 2^8 entries each, can be stored in the same page, as also four chunks of 2^7 entries, etc. To make this possible, let us consider that the contents of the previous page directories are pointers to the next ones. So, the only change appearing is that those pointers will now not always point to the beginning of a page. The above enables us to represent the page table in significantly less space.

Even without this optimization though, this approach is still superior to the previous one. Pages including fewer than 2^9 entries now manage to encode their information in less space, and using instead one of the 4KB, 2MB, 1GB sizes would only lead to redundant

entries. Of course, if needed, due to the abundance of available page sizes in DPART, the exclusion of some page sizes that may be considered space consuming, is easily tolerable. It should be also noted, that a further advantage of this method is the fact that, as now the TLB and the page table function under the same page sizes, the PTE obtained by a page walk is the same as the PFN that has to be filled in a TLB entry.

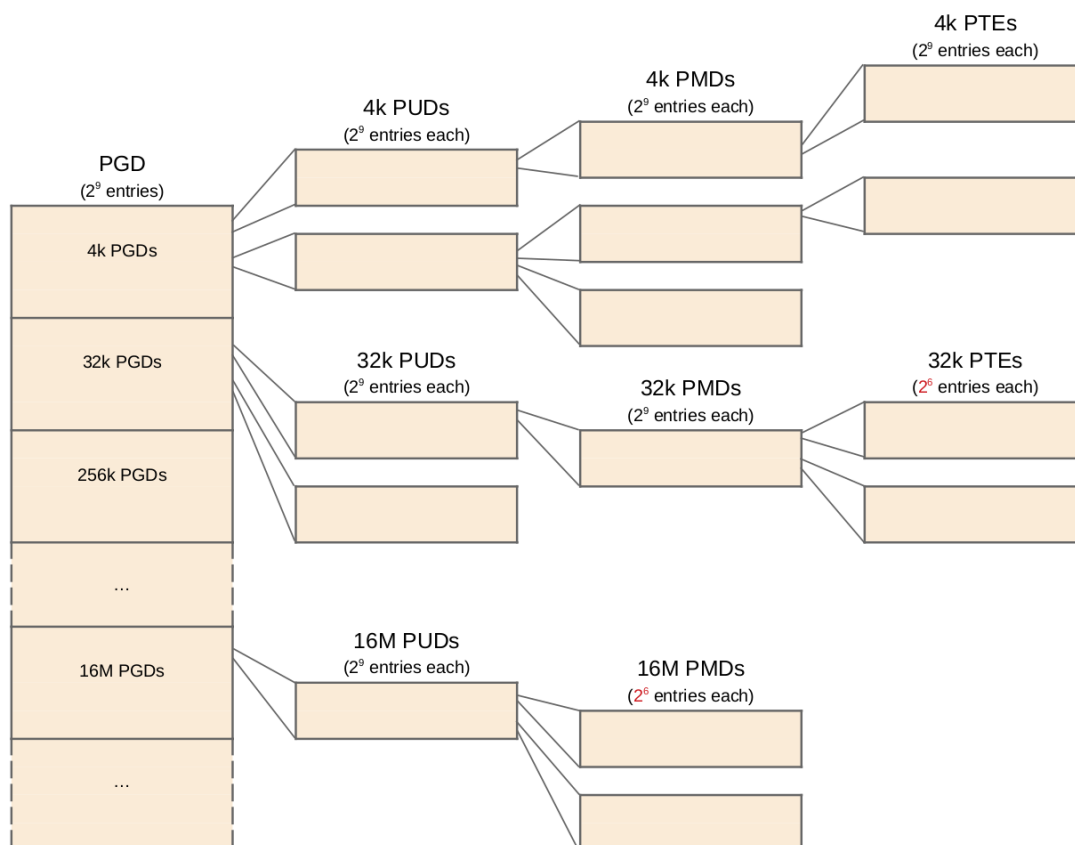


Figure 4.14: Modified page table under the third approach

Concluding this approach, the steps for its implementation would become:

- Page size identification using the address's MSBs
- Extraction of index and offset components according to the identified page size
- Page walk according to indices previously obtained
- Concatenation of PTE with offset to receive physical address
- TLB update with PTE

4.4 Further Optimizations

Up to this point, DPART has been completely established, having discussed the virtual address space partitioning enabling page size identification, the page size selection process, and also the TLB and page table that would support this scheme. This section introduces two further improvements aiming to optimize DPART's TLB hit rates.

4.4.1 Skewing

In the Linux kernel, memory allocation in the virtual address space happens either in top-down (most commonly) or bottom-up logic. More specifically, in top-down logic for example, when asked to find a new empty virtual memory area the Linux kernel begins searching from the highest address allowed and continues to lower addresses as it traverses the memory's red-black tree (see Section 3.5). The opposite happens in bottom-up logic. The above method results in the creation of contiguous memory regions, providing multiple advantages, such as the ability to merge virtual memory areas.

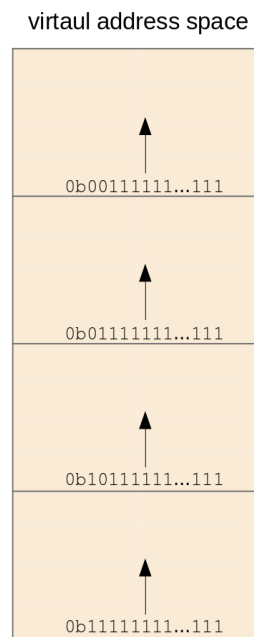


Figure 4.15: Top-down memory allocation in each partition

However, this creates a performance impediment for our scheme. Let's focus on top-down assignment. As it observed in Figure 4.15, in each partition, the majority of bits in addresses initially allocated are all 1s. Obviously, the same holds for the index part of an

address. Consequently, the first pages to be allocated from each partition will all map to the last TLB set. Similarly, the second page to be allocated from each partition, having all but the last index's bits equal to 1, will map to the next-to-last set. Figure 4.16 shows that.

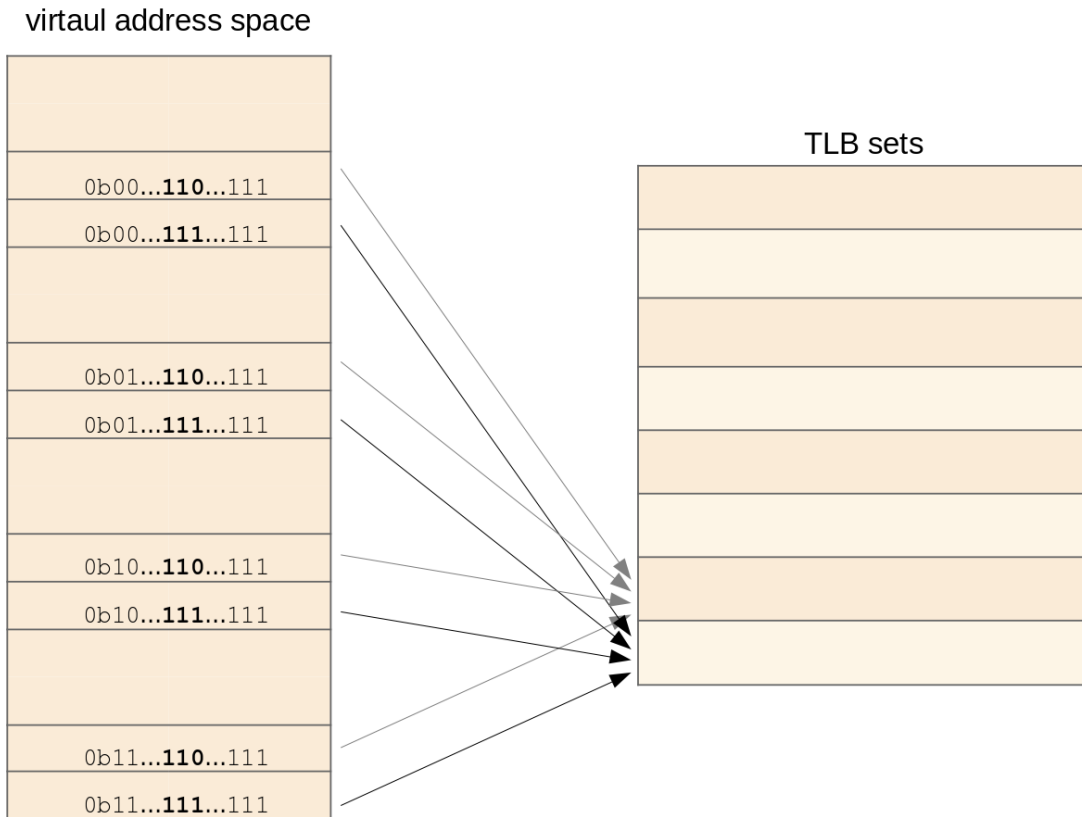


Figure 4.16: Bottom TLB sets accumulation (pages not in scale)

This accumulation in the last TLB sets violates the uniformity characteristic mentioned in 4.1, leading to performance degradation. To overcome this, inspired by Seznec's ideas [55, 56], we are targeting to skew the destination of a page in the TLB. Seznec proposes a scheme where an address can be mapped on different TLB set ways according to its page size. To do so, each TLB way is indexed using a different hash function. Those functions, to increase diffusion over the TLB entries, often apply XOR operations and each of them uses different address's bits.

In DPART, we are keeping Seznec's core idea of spreading translations and define a simpler skewing scheme exploiting DPART's partition bits. Since an address's MSBs differ accordingly to the page size, to get our skewed index we simply apply an XOR

operation between the index and the partition bits. This will result in a constant shift of the set used by pages in each partition, spreading translations deterministically over the TLB sets.

```

1 skew_idx_a (address, idx, idx_bits)
2   skew = CROP_LSB(address, address_bits - partition_bits);
3   skew = KEEP_LSB(skew, idx_bits);
4   return XOR(idx, skew);

```

Listing 4.1: Skew a pseudocode

```

1 skew_idx_b (address, idx, idx_bits)
2   if (partition_bits > idx_bits)
3     skew = CROP_LSB(address, address_bits - idx_bits);
4   else
5     skew = CROP_LSB(address, address_bits - partition_bits);
6     skew = ZERO_PAD(skew, idx_bits - partition_bits);
7   return XOR(idx, skew);

```

Listing 4.2: Skew b pseudocode

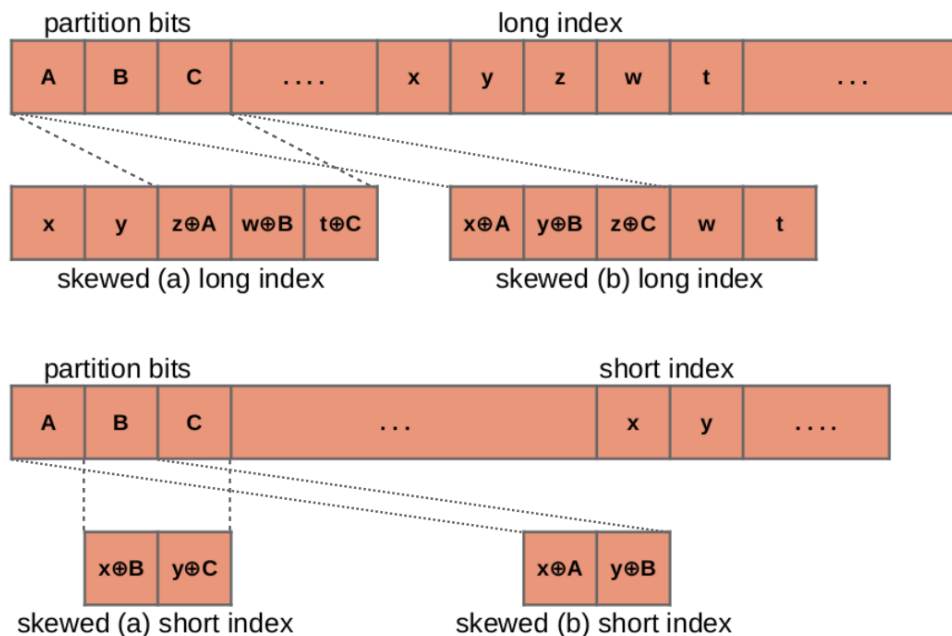


Figure 4.17: Skewing schemes for indices longer or shorter than partition bits

As index bits may differ in number with partition bits and, the XOR operation can be applied in various ways. We propose two different skewing mechanisms (named skew a, b). The pseudocodes of those mechanisms are presented in Listings 4.1 and 4.2. To explain the functionality of those two methods, Figure 4.17 illustrates their application. In the upper/lower part of Figure 4.17, one can see the skewed index in the case where partition bits are less/more than the TLB index bits, respectively.

At this point, we may observe the advantage of this skewing mechanism. To do so, let's recall the problematic behavior presented in Figure 4.16. In Figure 4.18 we show how the skew b method would redistribute page translations over the TLB, and solve the accumulation problem completely.

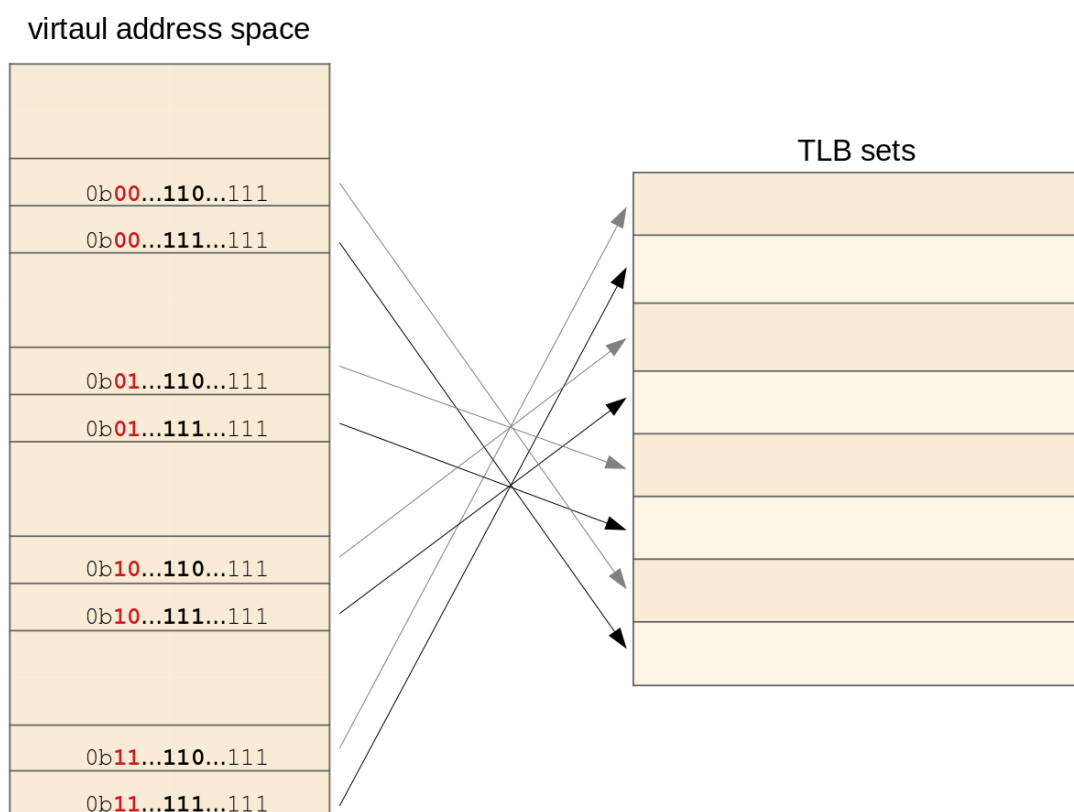


Figure 4.18: Page skewing (pages not in scale)

Another solution to the accumulation problem would suggest to create more TLBs, and use one for every page size, according to the partition bits. This has two severe weaknesses. First of all, the TLBs corresponding to page sizes that are used in low frequency, will be heavily underutilized, resulting in waste. Secondly, this approach is

not scalable, and it would not be affordable to use DPART with many page sizes, as we would need a new TLB structure for each of them. On the contrary, with the skewing schemes, we are fully utilizing the single TLB for all memory translations, potentially filling all its entries.

4.4.2 The Heap Overhead

Another optimization is referring to the mapping of the heap. As shown in Section 3.5, a process's heap is located in the beginning of its virtual address space, controlled by the program break. Since the heap resides in the very first virtual memory partition, 4KB pages are used for its mapping. For the majority of large memory regions allocation, instead of the heap, anonymous mappings are used, directing those regions in the desired partitions. For further optimization though, our intention is to also map the heap using a larger page size.

This requires some OS changes. In a modified `brk` system call, we chose whether to map the requested area directly after the current program break (in the 4KB page area), or to map it using DPART logic. The criterion of this choice is whether a `brk` increment is large enough to require larger pages. Nevertheless, many programs are incrementing their program break gradually with small steps and such an opportunity may never arise. However, using `gperftools`' `TCMalloc` [9], the heap size is reduced, and any gradual program break increments coalesce enabling this optimization. Also, even without `TCMalloc`, at every `brk` call, we could evaluate the heap's length, and if it is large enough, we can remap it to a different partition. Other implementations would suggest modifying `M_MMAP_THRESHOLD` or the `start_brk` portion at the initialization of a process.

At the scope of this thesis, we implement this optimization on simulation level, rather than modifying the operating system. What we do is to identify the boundaries of the heap (defined in `mm->start_brk`, `mm->brk`), and use the appropriate page size in this region, without remapping it.

4.5 Restrictions

In this section we analyze the limitations of our method. First of all, DPART requires an architecture supporting multiple page sizes. In contrary, other schemes may be orthogonal to such a design option such as [15, 35] and be able to function independently.

In addition, to support our scheme we need to use large pages at the first access of a data that is backed by a corresponding large memory allocation. This is implemented

by preallocation on reservation. Other schemes relying to high contiguity also require such an option [35] and notice that on-demand paging is more costful due to the high TLB miss rates that schemes supporting eager paging but limiting those miss ratios. Of course, it may be argued that a program may include big memory allocation calls and end up not using all the requested space, resulting in an unnecessary waste. This can be confronted by adding limitations in our policies, as discussed in the future work section (7.1).

Furthermore, our scheme is sensitive to external fragmentation. More specifically, in order to allocate a page, sufficient physical memory should exist. Otherwise, physical space should be found, or smaller pages should be used. However, this is analogous to any other method trying to leverage the advantages of superpages. The physical memory management part is also discussed in 7.1.

In computer architecture, everything is about tradeoff. We state that despite the above limitations of our method, the immediate page size identification that we offer, unlocking the benefits of a broad variety of page sizes will subsume to a positive result. Experiments showing the advantages of our method follow in the next chapter.

Chapter 5

Experiments

To evaluate our approach, it is necessary to measure a system’s performance running DPART’s logic under heavy memory workloads. At this stage, having implemented DPART’s logic into the Linux kernel, we construct software-running TLB simulators, attempting to output the TLB hit rates that a hardware TLB abiding by DPART’s rules would. However, under this logic, it is not possible to correctly measure the benefits that would arise from the simplicity of DPART. Those benefits, primarily concerning system mechanisms latency and energy consumption, are analyzed in Section 5.4.

In order to make the aforementioned evaluation valuable, we compare DPART with other address translation schemes, including the conventional 4KB pages scheme. To achieve this, we build their respective TLB simulators which are discussed later, in Subsection 5.1.3.

On the whole, in this chapter we present the methodology of our simulations (5.1), analyze the benchmarks being used (5.2), and finally present (5.3) and discuss (5.4) the results extracted.

5.1 Methodology

5.1.1 System

For our measurements we used a virtualized environment, whose host system included the components named in Table 5.1. The virtualization emulator used was QEMU [16].

| microarchitecture | Kaby Lake |
|-------------------|---|
| processor | Intel(R) Core(TM) i5-7600 CPU, 4 cores, 3.50GHz |
| OS | Linux 4.19.50 |
| L1 DTLB | 2M/4M pages, 4-way, 32 entries 1G pages, 4-way, 4 entries 4K pages, 4-way, 64 entries |
| L1 ITLB | 2M/4M pages, fully, 8 entries 4K, 8-way, 128 entries |
| L2 TLB | 4K/2M pages, 6-way, 1536 entries |

Table 5.1: System configuration

For our benchmarks, instead of using glibc’s malloc library function, we used TCMalloc [9]. The effect of TCMalloc in our benchmarks is analyzed in Section 5.2.

5.1.2 Simulating TLB Misses

To evaluate our scheme, we aim to measure our TLB’s hit ratio. To do this we have several options. Performance counters for Linux (PCL or perf) is a kernel-based subsystem that provides a framework for collecting and analyzing performance data [7]. This would allow us to capture a hardware TLB’s metrics while not having to experience simulation time overheads. On the other extrema, Pin [33] and other simulation tools enable a wide variety of actions during an application’s runtime, however significantly increase execution time. Since we neither have a hardware implemented component, nor we want to endure the time overhead of full system simulations, we aim to use an ad-hoc tool, instrumenting specifically TLB misses.

BadgerTrap, created by Gandhi et al. is such a tool, implemented for Linux kernel version 3 [24]. BadgerTrap marks the PTEs of a specified process, and each time a TLB miss occurs, it is converted into a page fault. Afterwards, a handler of our choice is called, and at its completion, a page walk takes place so that at the instruction restart the translation is existent in the TLB and execution proceeds.

One of our contributions in this thesis was writing a version of BadgerTrap compatible for Linux kernel 4. Now, to measure performance, in the TLB miss handler we initiate a lookup of our TLB software structures for the virtual address that caused that miss. As the set of addresses we work on consists of the ones causing TLB misses, this is of only a subset of the total addresses that an application accesses. However, it is a common

ground for all different experiments that we aim to hold, and hence our goal, which is comparison, is effectively achieved.

5.1.3 Software TLB Simulators

This subsection enumerates the experiments that are to be held. These experiments may be categorized under two groups; comparison between all different DPART setups, and comparison between only some selected DPART setups and other address translation schemes. The exact tests of each category are to be specified below.

5.1.3.1 Comparison of Different DPART Configurations

We are going to simulate DPART under 6 configurations. Those configurations are summarized in Table 5.2. As one can observe, those 6 configurations derive from the combinations of the two optimizations added to the basic DPART scheme, as mentioned in 4.4.1 and 4.4.2.

| | no skewing | skewing pattern A | skewing pattern B |
|----------------------|--------------|---------------------|---------------------|
| no heap optimization | dpart nonbrk | dpart nonbrk skew a | dpart nonbrk skew b |
| heap optimization | dpart | dpart skew a | dpart skew b |

Table 5.2: DPART simulations configurations

Furthermore, at OS-level, DPART presents two configurable options:

- partitioning bits
- page size selection policy

And in the process of simulating a TLB following our logic, we can modify the parameters of:

- TLB entries
- TLB associativity

Our initial goal is to investigate how the number of partitioning bits and affect our scheme's efficiency. To do so, we perform simulations based on the parameters of Table 5.3 under all configurations in Table 5.2.

| Partitioning Bits | Policy | TLB size | TLB associativity |
|-------------------|--------|----------|-------------------|
| 2 | CLOSER | 32 | 4 |
| 3 | CLOSER | 32 | 4 |
| 4 | CLOSER | 32 | 4 |
| 5 | CLOSER | 32 | 4 |

Table 5.3: Simulation parameters on varying partition bits

In the above parameter selection, one can observe that a TLB with a small number of entries has been selected. This has been chosen in order to produce lower hit rates and render any differences between the compared parameter combinations distinguishable.

5.1.3.2 Comparison Between DPART and Other Address Translation Schemes

In continuation, we need to observe how TLB entries and associativity affect our proposed scheme, and simultaneously compare it to other address translation schemes. The TLBs we will be simulating derive from the systems that are listed below:

- conventional
- hugepages split
- hugepages merged
- dpart_skew_b
- RMM

All TLBs are using LRU replacement policy.

In detail, the conventional TLB refers to translation based solely on 4KB pages, as in the majority of today's systems with disabled hugepages mechanisms.

Hugepages merged and hugepages split, refer to Transparent Hugepages (THP) [10, 8]. Specifically, modern architectures include a shared 1536 entries L2 TLB supporting both 4KB and 2MB pages and a separate 16 entries TLB for 1G pages only [32, 1]. The split TLB consists of three subTLBs, one for each page size (4KB, 2MB, 1GB). Those subTLBs are considered to be accessed in parallel and LRU policy is implemented separately in each subTLB. In our models, the merged TLB accommodates pages of all 4KB, 2MB, 1GB sizes in its entries and LRU policy is implemented in this structure as a whole, regardless of the page size of the entry to be evicted. Those two TLB patterns

were selected to be simulated since they represent the two extrema functionality cases of a TLB supporting hugepages.

| Partitioning Bits | Policy | TLB size | TLB associativity |
|-------------------|--------|----------|-------------------|
| 5 | CLOSER | 4 | 4 |
| 5 | CLOSER | 8 | 4 |
| 5 | CLOSER | 16 | 4 |
| 5 | CLOSER | 32 | 4 |
| 5 | CLOSER | 64 | 4 |
| 5 | CLOSER | 128 | 4 |
| 5 | CLOSER | 256 | 4 |
| 5 | CLOSER | 512 | 4 |
| 5 | CLOSER | 1024 | 4 |

Table 5.4: Simulation parameters on varying TLB size

| Partitioning Bits | Policy | TLB size | TLB associativity |
|-------------------|--------|----------|-------------------|
| 5 | CLOSER | 64 | 1 |
| 5 | CLOSER | 64 | 2 |
| 5 | CLOSER | 64 | 4 |
| 5 | CLOSER | 64 | 8 |
| 5 | CLOSER | 64 | 16 |
| 5 | CLOSER | 64 | 32 |
| 5 | CLOSER | 64 | 64 |

Table 5.5: Simulation parameters on varying TLB associativity

RMM (Redundant Memory Mappings) is a scheme introduced by Karakostas et al. [35]. In RMM, a region of arbitrary size, defined by its start and end, is representing a virtual memory area. Since those regions are neither aligned, nor their length is necessarily a power of 2, indexing is not possible. As a result, a TLB under RMM logic (named Range TLB) can only be fully associative, and hence, can only be sustainable with a small number of entries. Despite its associativity limitation, since RMM can support any positive integer as page size, it ensures to produce the lowest number of translations, given an arbitrary virtual memory state (exactly one translation for each virtual memory

area). Consequently, RMM is going to be an upper bound in our analysis, representing the perfect TLB hit rate, and we are going to simulate it with arbitrarily many entries.

The DPART TLB that is used in our comparisons incorporates both available optimizations, and as it will be shown, it was selected as it presented the best performance of other DPART TLBs.

To put all models into the test, simulations under the parameter combinations of Tables 5.4 and 5.5 will be held. In addition, to provide a fair comparison, a final diagram will be provided, which will depict the performance of each system under parameters of equivalent cost.

5.2 Benchmark Analysis

For our simulations, we used 6 benchmarks from the SPEC CPU 2006 benchmark suite, that present high memory workloads, and have also been selected in prior related work [35, 50].

In order to understand the functions of those benchmarks, we create a histogram based on the occurrence of their system calls. We isolate the system calls of our interest, referring to memory manipulation and access. To create those histograms, we initially run the benchmarks under the strace call. Afterwards, since many child processes have forked through our benchmark running system, we identify the PID of the desired application by spotting any `execve` system calls. Finally, we are able to collect information concerning those PIDs only. The results are shown in Figure 5.1.

By running this procedure both before and after incorporating TCMalloc into our system, we are able to also evaluate its effect. Our first observation is a vast reduction of `brk` calls under TCMalloc. As promised in 4.4.2, now fewer `brk` calls take place, eliminating the gradual slow increment of the heap area, providing tenable ground for our pertinent optimization.

Another observation refers to the extinction of `mremap` system calls from `soplex`, and in general reduction of `mmap` and `munmap` calls. This comes from TCMalloc's ability to compact those calls and result in not only contiguous areas that will be useful in this work, but also faster execution and space efficient representation of small objects.

Finally, read and write operations remain approximately invariant, while `mprotect`, `open`, and `close` system calls increase.

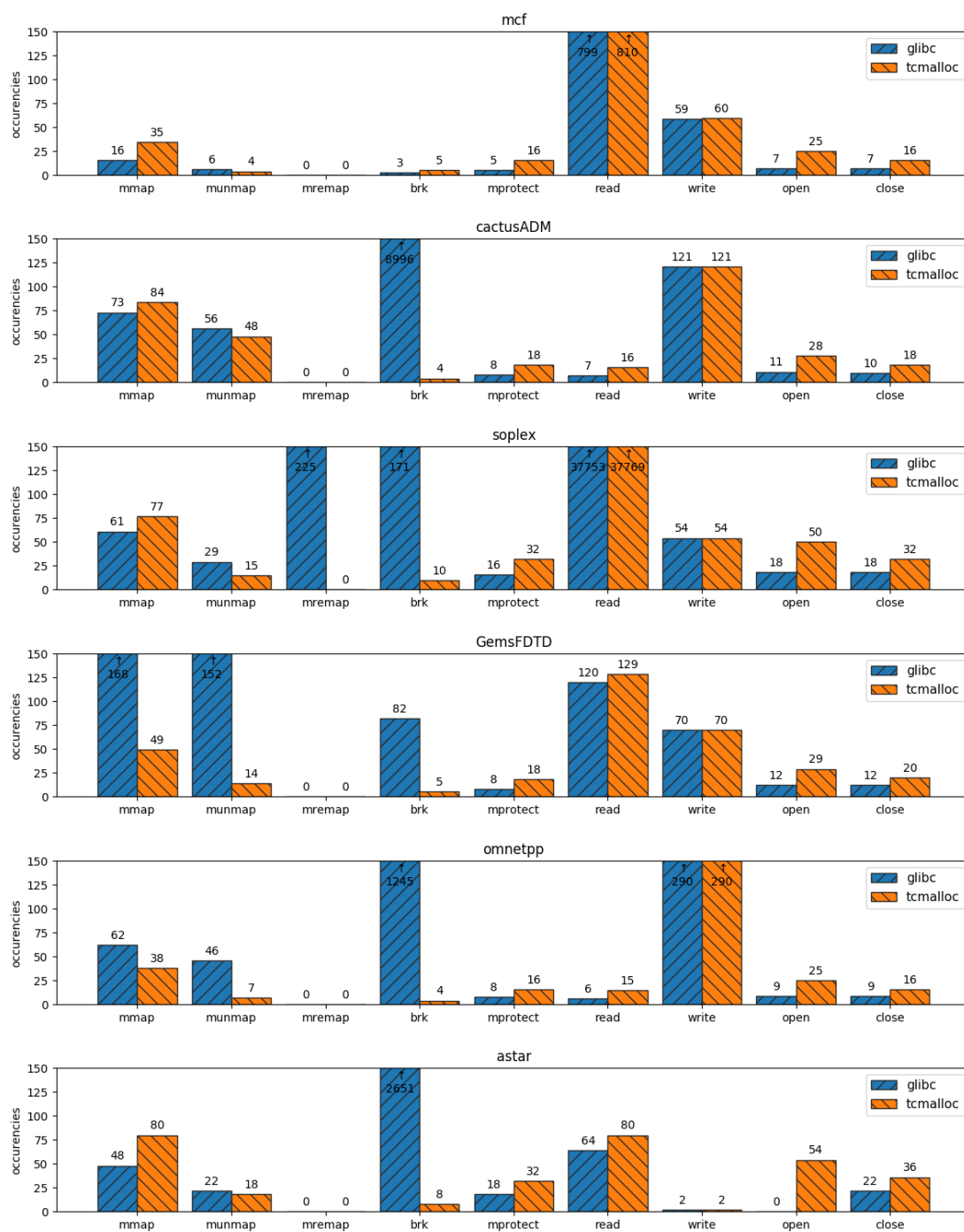


Figure 5.1: System calls frequency with/without TCMalloc

5.3 Results

Having analyzed the environment behind our simulations, we are ready to proceed to their results.

5.3.1 First Results

Before putting DPART into the test with heavy workloads, we perform an initial test validating that our scheme operates properly. To do so, we construct the microbenchmark whose pseudocode is presented in 5.1. What this program essentially does is to allocate a contiguous space of `pages` 4KB pages and cyclically access an element from each page `cycles` times. This would thrash a conventional TLB for values of `pages` larger than the TLB's entries.

```
1 | addr = mmap(NULL, pages*page_size, ...);
2 |
3 | for (i=0; i<pages*page_size; i+=page_size)
4 |     assign(addr[i]);
5 |
6 | for (j=0; j<cycles; j++)
7 |     for (i=0; i<pages*page_size; i+=page_size)
8 |         access(addr[i]);
```

Listing 5.1: Tlb miss microbenchmark pseudocode

We run the microbenchmark for an input of `pages = 100000`, `cycles = 100` accompanied by a 3 partitioning bit 32 entries, 4-way associative non-optimized DPART TLB simulator. Out of the total 10100433 lookups provoked by the instrumented host TLB misses, the DPART TLB manage to experience only 11 misses. This is owing to the fact that our model managed to allocate a page of appropriate size including the whole `addr` array.

5.3.2 OS Changes Execution Time Overhead

Our next analysis aims to ascertain that our operating system modifications do not provoke any significant time overheads, we measure the execution time of each benchmark both at their absence and existence. To get those measurements, for each benchmark we repeated execution three times and received their arithmetic mean. The results are illustrated in Figure 5.2.

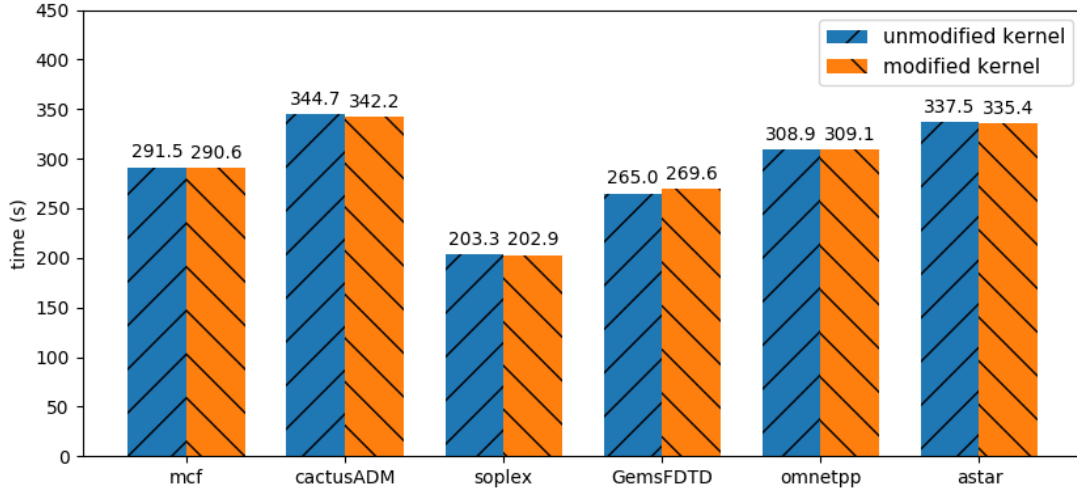


Figure 5.2: Execution time with/without DPART OS modifications

As it can be observed, no time overhead is observed, and any minor differences are due to randomness.

5.3.3 Comparison of Different DPART Configurations Results

Hereby, we present the outputs from simulations run under Table 5.3 for every DPART configuration of Table 5.2. The results being very rich, we provide the outputs for each benchmark in the Appendix and present below the geometric mean of the received portions. The geometric mean is selected as it conveys proportionally changes of the separate benchmarks to the final result. Since we intend to depict TLB miss rates, which can approach zero values, we compute the geometrical mean of the benchmarks' TLB hit rates, and subtract from the 100 percent mark. Equivalently:

$$output_miss_rate = 100 - \left(\prod_{benchmarks} benchmark_hit_rate \right)^{\frac{1}{|benchmarks|}} \%$$

In Figure 5.3 we present those TLB miss metrics for a 32 entries 4-way associative TLB.

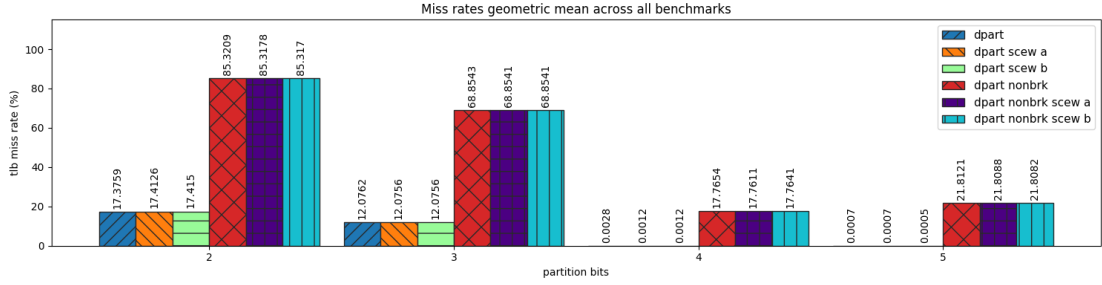


Figure 5.3: TLB miss rates for variant partitioning bits

Miss rates are expected to be high due to the small TLB structure’s size. As previously stated, this selection was made in order to make our models’ differences easily observable. Additionally, the reason why high miss rates are observed in few partition bits (especially in 2) is that the selected policy may not have enough available page sizes to assign a well fitting one to a virtual address space. As a result, many small pages may be used for an area that would be mapped more effectively under a larger size.

The best performing scheme is the skew b heap optimized implementation under 5 bits partitioning. This configuration is challenged to be compared with the other schemes of paragraph 5.1.3.2.

5.3.4 Comparison Between DPART and Other Address Translation Schemes Results

Bar plots including the aforementioned comparisons follow. Figure 5.4 (a) indicates the performance of the 5 different models in a TLB size sensitivity analysis and Figure 5.4 (b) denotes TLB miss rates in a TLB associativity sensitivity analysis.

As we can observe, in all occasions the DPART selected scheme is remarkably close to RMM’s Range TLB, our upper bound. The only cases that present low performance for all schemes are corresponding to TLBs with 8 or less entries. In the case of a 16-entries 4-way set associative TLB, DPART suffers a 1.561% miss rate, being far superior than the conventional and hugepages schemes, whereas the 16 entries fully associative Range TLB presents a 1.1097% miss rate, being slightly better. In all other scenarios, DPART manages to experience a geometric mean TLB miss rate below 0.0005%.

Split and merged huge pages TLBs manage to also offer great performance but only in either a large number of TLB entries, or in highly associative cases. Last, considering the conventional TLB, it fails to present miss rates below 75% in all tested scenarios.

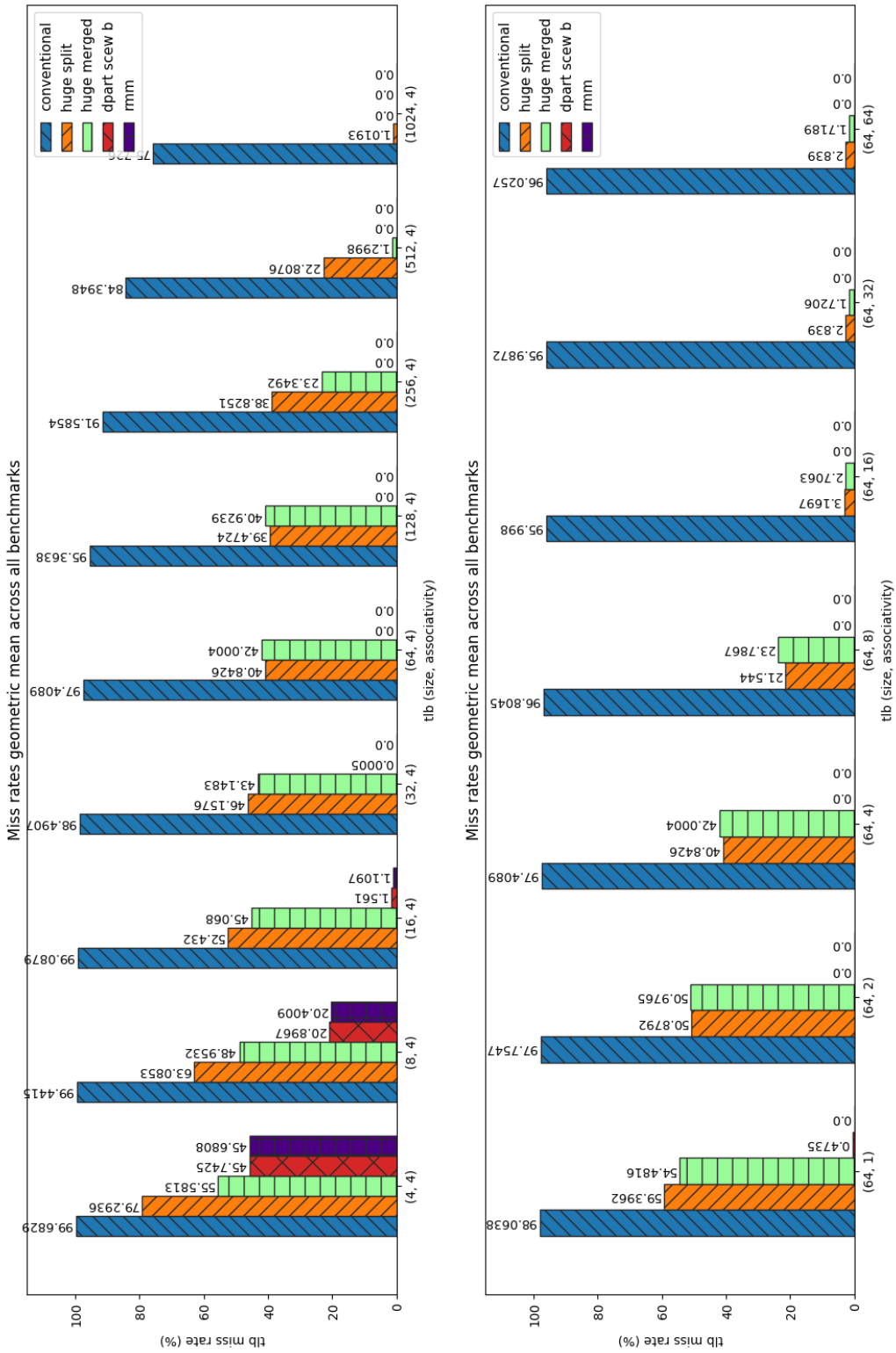


Figure 5.4: TLB miss rates for variant TLB (a) size and (b) associativity

Detailed scores considering all benchmarks separately are included in Appendix C, also demonstrating DPART’s high performance.

Last but not least, Figure 5.5 includes a comparison where RMM Range TLB’s number of entries equals the other schemes’ associativity. This test is provided because Range TLB lacks the ability to be indexed and consequently its average lookup time is proportional to its full size. On the contrary, other TLBs supporting index-based search offer an average lookup time proportional the their associativity.

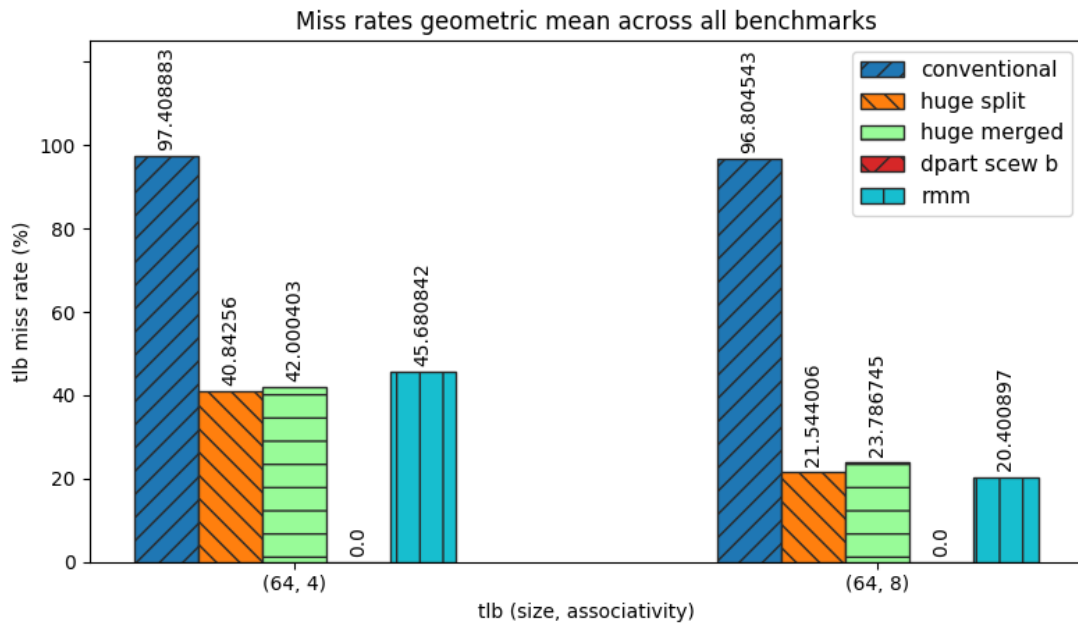


Figure 5.5: Same associativity TLBs comparison

In both cases presented, DPART TLB achieves TLB miss ratios below 10^{-6} (we output values rounded to 6 decimal digits). Other schemes’ resulted miss rates are above 40% and 20% percent in 4-way and 8-way associative cases, respectively.

5.4 Discussion

The previous sections render DPART’s ability to perform well in big-memory workloads irrefutable. Additionally, our scheme manages to achieve those explicitly low TLB miss rates while retaining all ideal characteristics presented in Section 4.1.

DPART manages to support any power of two page size in an index-based search TLB avoiding mirroring or high conflicts that would derive from any compromise regarding

the position of the index bits in a virtual address. Aside from deterministic indexing, our scheme manages to distribute stored translations uniformly across all TLB sets, thus possessing high scalability. As a result, DPART can effectively support TLBs of many entries, contrasting to RMM that also depicted superior performance.

However, perhaps the most vital DPART's advantages don't lie in TLB's reach enhancement. Execution time is not only reduced by TLB hit rate improvement, but also by TLB miss penalty reduction. Our simple mechanism are estimated to present low latencies, thus ameliorating computing performance even more.

Considering required modifications, as for software, only minor operating system changes are required, which, as showed, do not have any impact on execution time. Regarding hardware, both TLB and page table do not need to change their entries' structure. The biggest amendment lies in the addition of 3 multiplexers and 2 and logic gates having deterministically defined and often constant inputs. When inputs are constant, such as the masks we use to obtain the true tag and offset, they can be stored in immutable registers or other hardware units without affecting performance.

Another major advantage that was not evaluated at the scope of this thesis regards power consumption. Circumventing mechanisms running in parallel, verification processes or any other complex procedures, we estimate that DPART's straightforward, deterministic and simple operations will result in low energy costs.

One extra point worth mentioning is that the ability to incorporate all translations for all page sizes in a single TLB structure will contribute in aspects such as chip size reduction and construction cost.

Chapter 6

Related Work

In current bibliography, many efforts have been made targeting to optimize address translation. This chapter presents the majority of the work that is closely related to this thesis.

6.1 Range Based Schemes

An early attempt to reduce the address translation overhead by Basu et al. is the direct segment [15]. Under this scheme, a contiguous range of the process's virtual address space is mapped with a direct segment while the rest of the address space is mapped via conventional page mapping. Due to high contiguity that virtual memory of heavy workloads may present, the direct segment manages to achieve great TLB miss rate reduction. However, this mechanism requires the programmer to explicitly allocate a segment during startup, and cannot support more than one contiguous memory area.

The scheme used in our experiments, based on the idea of direct segments and substantially improving it, is RMM by Karakostas et al. [35]. RMM operates in parallel with standard paging and automatically detects contiguous address space ranges to subsequently map them in the range table structure. Ostensibly, the range table is a fully associative structure accommodating multiple direct segments, while it provides transparency to the application. RMM's most significant disadvantage is its lack of scalability. Since ranges are arbitrarily sized memory areas they cannot be indexed and consequently can reside only in fully associative structures.

DPART manages to overcome range based schemes' limitations by offering indexing TLBs while absence of arbitrary memory areas size representation is not an issue due to the wide variety of page sizes that offers.

6.2 Leveraging Contiguity

Many schemes attempt to benefit from contiguity that mappings may present. Pham et al. propose Coalesced Large-Reach TLBs (CoLT) to coalesce multiple page translations into single TLB entries [52]. Observing that contiguity can actually increase with greater system load, Pham constructs three CoLT schemes depending on associativity, while always having to study translations in parallel with execution to detect contiguity. Pham develops this work and constructs Clustered TLB mapping small set of contiguous virtual pages to clustered sets of physical pages [53]. However, as shown in prior work [35] can only offer limited reductions in overheads and for only small-memory workloads, while outperformed by THP on big-memory workloads.

Another coalescing mechanism was proposed by Park et al. [50]. In their proposed hybrid coalescing technique, the operating system records contiguity status and computes the anchor distance, being the amount of contiguous pages to be stored with a single entry each time. At every context switch the anchor distance is reset, while the underlying software mechanism to determine it runs frequently updating the page table and TLB, costing milliseconds.

Last, MIX TLBs is a recent work by Cox and Bhattacharjee [21]. Working in the same scope with this thesis, they create energy efficient set-associative structures supporting multiple page sizes. However, they use only the set-indexing scheme of base page size resulting in index bits being a part of larger pages offset, violating the proper index bits selection principle of Section 4.1. As a result, mirroring may be present, potentially mapping a large page to all TLB sets. To overcome this, MIX TLBs coalesce contiguous superpages into the same TLB entry. To have a positive tradeoff, as many coalesced pages as potential mirrors are needed. This statement is not guaranteed and no general assumptions may be extracted with certitude about this method.

Apart from other drawbacks stated, coalesced techniques may restrict the maximum size of concatenated translations, and hence limit the TLB reach.

6.3 Prediction Based Mechanisms

Learning techniques have been applied in computer architecture, often with great success. Recently, Hashemi et al. for example have shown that machine learning methods can be applied in architecture components such as the prefetching mechanism, demonstrating superior performance in terms of precision and recall [30].

Speculation has also largely been applied in page size prediction. Pertaining to

architectures supporting many page sizes, to address the page size determination issue, predicting mechanisms are frequently used. Bradford et al. introduce various prediction tables receiving as input the PC, register value, and register names [19]. However, when facing a misprediction, as many TLB lookups as the number of the available page sizes have to occur. Papadopoulou et al. improve this idea by predicting only between base page size and superpage page size, including 64KB, 512KB, 4MB [49]. Having only two classes to choose from, misprediction rates can fall to no more than 1.2%. A disadvantage of this method is that it violates the "Proper index bits selection" desired characteristic of address translation, mentioned in Section 4.1. To overcome this, the same paper leveraged Seznec's skewing scheme [56] in order to compute indices as a function of the appropriate address bits depending on the page size. Even with this methodology though, the misprediction overhead is not entirely eliminated, and still TLB structures for all page sizes have to be accessed sequentially.

In another speculation-based mechanism, Alverti et al. design a scheme where the virtual to physical address offset is extracted, exploiting the concept of memory regions (continuous memory mappings of arbitrary size) [11]. After an address translation is predicted, the CPU enters speculative execution mode, while a verification page walk runs in the background. In the misprediction case, the scheme's pipeline flushes and the instruction replays.

Recent work by Kraska et al. has shown that all existing index structures can be replaced with other types of models such as deep-learning models, resulting in superiority in both speed and space [39]. The above results inspired Margaritov et al. to index the top levels of the page table via learning mechanisms [42]. However, even achieving 99.9% accuracy, this approach presents latency overheads failing to outperform the page table radix tree mechanism's access time. As a result, Margaritov et al. switched to optimizing the page table through prefetching mechanisms [43]. Their successful scheme prefetches the deeper levels of the page table aiming to lower page walk costs. This being an orthogonal approach to *ñame*, can further improve our proposed page table mechanism.

Learning techniques fail to provide 100% correct predictions violating the determinism/correctness address translation desired characteristic. DPART on the other hand, provides deterministically and efficiently computes page size and no speculation is necessary.

6.4 Address Space Partitioning

The core idea of this thesis also appear in prior bibliography. An early work in which virtual address space partitioning takes place is Opportunistic Virtual Caching (OVC) by Basu et al. [13]. Basu divides the virtual space into two partitions (named physical and virtual) according to the highest order bit of the address range. Then, cache lookups with their virtual address in the virtual partition can use the virtual address to cache data.

Later, Basu in his PhD devoted a chapter in his proposed "merged-associative TLB" [14]. This scheme applies a very similar address space partition as this thesis does in order to determine page size. The proposed TLB though consists of multiple sub-TLBs, one for each page size, demolishing the ability to scale and differing from our approach. In addition, Basu states that his model did not enable any reduction in number of TLB misses comparing to a Split TLB supporting two page sizes.

Another work by Keppel et al. submitted as a patent incorporates the same mechanism [36]. In this work, the address's valid bits are used for page size indication through a page size set-probe group association logic. A continuation of this approach is to support multiple page sizes by again including a TLB with several sub-TLBs that each holds a different page size.

The latter two schemes, being the most resembling to our approach fail to use a single TLB for all operations. We manage to do so by ensuring tag uniqueness and provide tenable ground for coexistence of addresses representing multiple page sizes, alongside with all the advantages of this feature.

6.5 Groundbreaking Approaches

Other attempts to improve address translation performance include major modifications in virtual memory. SpaceJMP by Hajj et al. uses multiple address spaces as first-class objects enabling process threads to switch between them [28]. SpaceJMP is developed in the DragonFly BSD and Barrelfish operating systems. A significant advantage of this approach relates to sharing large memory quantities.

In order to overcome address translation limitations presented by conventional mechanisms, Picorel et al, developed near-memory translation mechanisms [54]. Bringing processes closer to memory benefits from the physical data proximity. The Mondrian Data Engine is an example of a system implementing a near-memory processing architecture and demonstrates significant performance boost [23].

Another groundbreaking technique is Devirtualizing Memory (DVM) by Haria et al. [29]. Concept of DVM is the (approximately) identical mapping between virtual and physical address space. In this manner, address translation becomes trivial distinguishing overheads of walking hierarchical page tables, and only a validation mechanism is needed.

Last, Swift proposes file-only memory in order to achieve constant memory operations independent of size[48]. This may be done by leveraging the operating systems ability to manage large quantities of persistent data efficiently through the file system. This entails a complete redesign of memory management for both operating systems and language runtimes.

While those approaches may promise great advantages, their implementation requires redefining both software and hardware features. Thus, the adoption of some of them by market systems may be considered distant. On the other hand, DPART requires minor changes in both software and hardware elements and may retain immutable crucial elements such as TLB entries structure and page table. As a result its incorporation in today's systems may be considered easier.

Chapter 7

Conclusion and Future Work

7.1 Future Work

In this thesis, our address translation scheme, DPART, has been presented. DPART consists of a sufficiently established mechanism whose core functionalities have been fully implemented. Also, our scheme has been evaluated and shown promising TLB performance in our software-based simulations. At this point, future directions considering our mechanism arise, including steps towards the completion of DPART and additional evaluation methods that will unveil even more advantages of our technique.

7.1.1 Further Implementation

Having demonstrated superior performance through our software experiments, a hardware implementation of DPART may be considered as a next step. Hardware components pertaining to the extraction mechanism of the desired TLB offset, index and tag, and a possible optimization considering the page walker, have been thoroughly analyzed in theoretical level.

A software feature of our model that is not fully implemented yet, is the physical memory management section. More specifically, one should consider the case in which a sufficiently large contiguous physical memory area needed to map a requested page does not exist. At this point, we only examine such availability in virtual address space. Thanks to the fact that Linux supports Transparent Huge Pages [8], the baseline of the aforementioned mechanism already exists, and only few adjustments have to be made. At the absence of adequate physical space, our scheme benefits from the abundance of available page sizes, since degradation to a smaller page size may still result in the usage of pages large enough to boost performance. On the contrary, in THP such degradation

options would be equivalent to reducing a page size by the factor of 2^9 .

An addition needed for the completion of DPART refers to implications that a change of memory permissions may have. In detail, the system call `mprotect` may alter the access protections of a process's memory pages [5]. Since `mprotect` functions on base page size level, this may be problematic to our design, as a large page may be divided into segments of different permissions, and a single page should have unique permissions. To overcome this, we may consider two approaches; either page demolition to pages of equal permission bits, or modification of `mprotect` so that it functions on addresses aligned to the page size specified by DPART. The second approach is strongly preferred as the former would require remaps to the appropriate virtual memory partition.

The last software feature that has not been implemented regards the `brk` optimization (Section 4.4.2). The usage of larger pages for the mapping of the heap is now achieved in simulation level, while software implementations have been described but not executed.

Recent research showing that large pages may be harmful to NUMA systems [25] encourage us to include another point worth considering in this future work section. In detail, to improve system's total performance, the selection of the appropriate page size may not be correlated with the application's request only. Many other factors including process information, machine state and the system's current characteristics may play a crucial role in determining the optimal page size. Such a holistic approach, requiring to collect system information and to use them judiciously in order to determine appropriate allocation parameters should be considered.

7.1.2 Further Evaluation

The simulation-based experiments that were held, successfully depicted DPART's ability to experience remarkably low TLB miss rates. However, its simplicity may yield great benefits in other performance factors too. Power consumption ameliorated by simple and low latency mechanisms, reduced chip size offered by the existence of a single TLB structure supporting all page sizes, and total execution time reduction (which remains our primary goal) enabled by the our design, are some such instances. To evaluate the above, we need to use full system simulators such as `pin` [33].

In addition, since our scheme performed miss rates close to zero in the vast majority of the tested configurations under SPEC CPU 2006, we may attempt to use alternative, probably more recent benchmark suites, presenting heavier memory workloads. Last but not least, to furtherly illustrate DPART's advantages or weaknesses and precisely rank it among other implementations, it is crucial that more existing address translation schemes be compared with our logic.

7.2 Concluding Remarks

Because of its big overhead, virtual memory optimization is one of key factors to optimize a computer system's performance. DPART is our proposition to efficient address translation. Our scheme was analyzed in chapter 4 where its mechanisms were initially introduced. Ostensibly, the simplicity straightforwardness and deterministic nature of those mechanisms will result in low latencies and power consumption. A meticulous evaluation of our scheme is provided later-on in chapter 5. As it was observed, DPART's TLB manages to achieve excellent performance, producing constantly approximately equal results with the upper bound of our analysis, the fully associative Range TLB. The importance of address translation optimization results in abundant relevant research. In Chapter 6 we presented some of the most significant and most closely related to our approach work, discussing how their drawbacks may be circumvented by our scheme. Last, future work exists that would contribute in the evolution and thorougher evaluation of our model.

Bibliography

- [1] “Coffee Lake - Microarchitectures - Intel.” https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake.
- [2] “Five-level page tables.” <https://lwn.net/Articles/717293/>.
- [3] “Linux Kernel Documentation” https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt
- [4] “MALLOPT(3) man page.” *Linux Programmer’s Manual*, <http://man7.org/linux/man-pages/man3/mallopt.3.html>
- [5] “MPROTECT(2) man page.” *Linux Programmer’s Manual*, <http://man7.org/linux/man-pages/man2/mprotect.2.html>
- [6] “MREMAP(2) man page.” *Linux Programmer’s Manual*, <http://man7.org/linux/man-pages/man2/mremap.2.html>
- [7] “perf: Linux profiling with performance counters.” https://perf.wiki.kernel.org/index.php/Main_Page.
- [8] “RFC: Transparent Hugepage support.” <https://lwn.net/Articles/358904/>.
- [9] “TCMalloc.” <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>.
- [10] “Transparent hugepages.” <https://lwn.net/Articles/359158/>.
- [11] Alverti, Chloe, Vasileios Karakostas, Georgios Goumas, Nectarios Koziris. “PACT: G: Speculative Offset Address Translation” *ACM Student Research Competition Grand Finals*, 2019.
- [12] ARM Holdings. “ARM® System Memory Management Unit Architecture Specification, SMMU architecture version 3.0 and version 3.1.” *Document number: IHI 0070B*, 2016-2017.

- [13] Basu, Arkaprava, Mark D. Hill, and Michael M. Swift. “Reducing Memory Reference Energy with Opportunistic Virtual Caching.” *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.
- [14] Basu Arkaprava. “Revisiting Virtual Memory.” *Ph.D. Thesis. University of Wisconsin-Madison*, 2013.
- [15] Basu Arkaprava, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. “Efficient Virtual Memory for Big Memory Servers.” *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA 13*, 2013
- [16] Bellard, Fabrice. “QEMU, a Fast and Portable Dynamic Translator.” *USENIX Annual Technical Conference, FREENIX Track*, 2005.
- [17] Bhattacharjee, Abhishek, Lustig, Daniel. “Architectural and Operating System Support for Virtual Memory.” *Morgan & Claypool Publishers. p. 1. ISBN 9781627056021.*, 2017.
- [18] Bovet, Daniel P., and Marco Cesati. “Understanding the Linux Kernel.” *Beijing: OReilly*, 2006.
- [19] J. Bradford, J. Dale, K. Fernsler, T. Heil, and J. Rose, “Multiple page size address translation incorporating page size prediction.” *US Patent 7,739,477*, Jun. 15 2010.
- [20] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. “Introduction to Algorithms.” *Cambridge (Inglaterra): Mit Press*, 2009.
- [21] Cox, Guilherme, and Abhishek Bhattacharjee. “Efficient Address Translation for Architectures with Multiple Page Sizes.” *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS 17*, 2017.
- [22] Denning, Peter. “Before Memory Was Virtual.” *In the Beginning: Recollections of Software Pioneers*, 1997.
- [23] Drumond, Mario, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. “The Mondrian Data Engine.” *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2 (2017): 639–51.
- [24] J. Gandhi, A. Basu, M. Hill and M. Swift, “BadgerTrap.” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 2, pp. 20-23, 2014.

- [25] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. “Large Pages May Be Harmful on NUMA Systems.” In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, 2014.
- [26] Guibas, Leonidas J., and Robert Sedgewick. “A Dichromatic Framework for Balanced Trees.” *19th Annual Symposium on Foundations of Computer Science (Sfcs)*, 1978.
- [27] Hady, Frank T., Annie Foong, Bryan Veal, and Dan Williams. “Platform Storage Performance With 3D XPoint Technology.” *Proceedings of the IEEE 105, no. 9: 1822–33*, 2017.
- [28] Hajj, Izzat El, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-Mei Hwu, Timothy Roscoe, and Karsten Schwan. “SpaceJMP.” *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS 16*, 2016.
- [29] Haria, Swapnil, Mark D. Hill, and Michael M. Swift. “Devirtualizing Memory in Heterogeneous Systems.” *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS 18*, 2018.
- [30] Hashemi, Milad, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, Parthasarathy Ranganathan. “Learning Memory Access Patterns.” *International Conference on Machine Learning - ICML*, 2018.
- [31] Intel Corporation, “5-Level Paging and 5-Level EPT.” <http://kib.kiev.ua/x86docs/SDMs/335252-001.pdf>, 2016.
- [32] Intel Corporation, “Intel 64 and IA-32 Architectures Software Developer’s Manual.” <https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>, 2019.
- [33] Intel Corporation, “Pin - A Dynamic Binary Instrumentation Tool.” <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>, 2012.

- [34] Jun, Hongshin, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. “HBM (High Bandwidth Memory) DRAM Technology and Architecture.” *2017 IEEE International Memory Workshop (IMW)*, 2017.
- [35] Karakostas, Vasileios, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. “Redundant Memory Mappings for Fast Access to Large Memories.” *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA 15*, 2015.
- [36] Keppel, David Pardo, Binh Pham. “Valid Bits of a TLB for checking multiple page sizes in one probe cycle and reconfigurable sub-TLBs.” *US Patent 2019 / 0227947 A1*, Jul. 25 2019
- [37] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. “Readings in computer architecture.” *Morgan Kaufmann Publishers Inc.*, 2000..
- [38] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution.” *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 19–37.
- [39] Kraska, Tim, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. “The Case for Learned Index Structures.” *Proceedings of the 2018 International Conference on Management of Data - SIGMOD 18*, 2018.
- [40] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space.” *Proceedings of the 27th USENIX Security Symposium, MICRO-52*, 2018, pp. 973–990.
- [41] Love, Robert. “Linux Kernel Development a Thorough Guide to the Design and Implementation of the Linux Kernel.” *Upper Saddle River, NJ: Addison-Wesley Publishing Company*, 2010.
- [42] Margaritov, Artemiy, Dmitrii Ustiugov, Edouard Bugnion and Boris Grot. “Virtual Address Translation via Learned Page Table Indexes.” *Conference on Neural Information Processing Systems - NeurIPS*, 2018.
- [43] Margaritov, Artemiy, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. “Prefetched Address Translation.” *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture - MICRO 52*, 2019.

-
- [44] MIPS Technologies, Incorporate. “MIPS® Architecture For Programmers Volume III: MIPS64® / microMIPS64™ Privileged Resource Architecture.” *Document Number: MD00091, Revision 6.03*, 2015.
- [45] Patterson, David A., and John L. Hennessy. “Computer Organization and Design: the Hardware/Software Interface.” *San Francisco: Morgan Kaufmann*, 2005.
- [46] D. Quintero, S. Chabrolles, C. H. Chen, M. Dhandapani, T. Holloway, C. Jadhav, S. K. Kim, S. Kurian, B. Raj, R. Resende, B. Roden, N. Srinivasan, R. Wale, W. Zanatta, and Z. Zhang. “IBM Power Systems Performance Guide Implementing and Optimizing.” *ibm.com/redbooks*, 2013.
- [47] Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne. “Operating Systems Concepts.” *Chichester: Wiley*, 2014.
- [48] Swift, Michael M. “Towards O(1) Memory.” *Proceedings of the 16th Workshop on Hot Topics in Operating Systems - HotOS 17*, 2017.
- [49] Papadopoulou Misel-Myrto, Xin Tong, Andre Sez nec, and Andreas Moshovos. “Prediction-Based Superpage-Friendly TLB Designs.” *IEEE 21st International Symposium on High Performance Computer Architecture - HPCA*, 2015.
- [50] Park, Chang Hyun, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. “Hybrid TLB Coalescing.” *ACM SIGARCH Computer Architecture News* 45, no. 2 (2017): 444–56.
- [51] Pawlowski, J. Thomas. “Hybrid Memory Cube (HMC).” *2011 IEEE Hot Chips 23 Symposium (HCS)*, 2011.
- [52] Pham, Binh, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. “CoLT: Coalesced Large-Reach TLBs.” *45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.
- [53] Pham, Binh, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. “Increasing TLB Reach by Exploiting Clustering in Page Translations.” *IEEE 20th International Symposium on High Performance Computer Architecture - HPCA*, 2014.
- [54] Picorel, Javier, Djordje Jevdjic, and Babak Falsafi. “Near-Memory Address Translation.” *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.
- [55] Sez nec André. “A Case for Two-Way Skewed-Associative Caches.” *Proceedings of the 20th Annual International Symposium on Computer Architecture - ISCA 93*, 1993.

- [56] Seznec André. “Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB.” *IEEE Transactions on Computers* 53, no. 7 (2004): 924–27
- [57] Swift, Michael M. “Towards O(1) Memory.” *Proceedings of the 16th Workshop on Hot Topics in Operating Systems - HotOS 17*, 2017.

Appendices

A. Red-Black Trees

A red-black is a binary tree satisfying the following properties [26, 20]:

- Every node is either red or black.
- The root is black.
- Every leaf (NIL) is black.
- If a node is red, then both its children are black.
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

A red-black tree is approximately balanced. To provide an intuition for this argument, we show that a 3-node chain cannot be a red-black tree. In Figure A.1 all the possible coloring combinations are presented (on a black root). Colorings in A.1 (a), (b), (c) violate the last property and A.1 (d) is not a valid red-black tree since it has two consecutive red nodes.

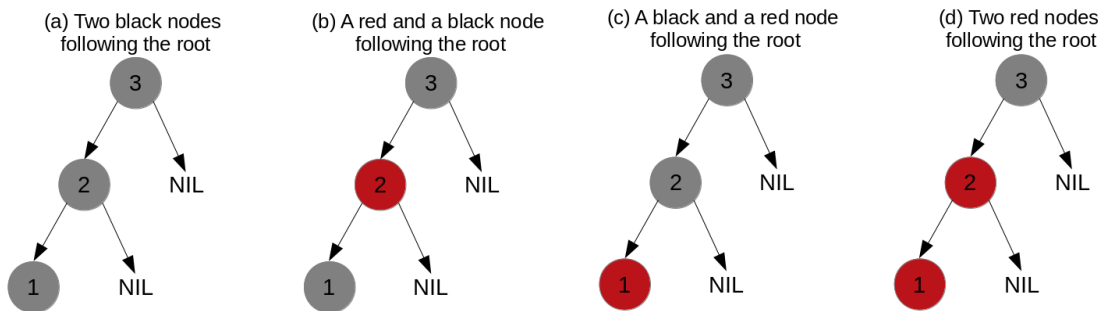


Figure A.1: All possible 3-node chain trees violate the red-black tree properties

It can be proven that a red-black tree with n internal nodes has height of at most $2\log(n + 1)$. The intuition of this argument is that taking only the black nodes into account, we have a perfectly balanced tree, and the red nodes (limited by not being consecutive) can only double the height of the tree.

This tree's height limitation results in $O(\log n)$ tree operations (search, min, max, insertion, deletion). In a tree modification (insertion or deletion), node rearrangements and recolorings have to be performed and due to the red-black tree's properties, those operations can be performed efficiently maintaining the $O(\log n)$ total cost. In comparison to AVL trees for example, AVL are more balanced but may cause more rotations during an insertion or deletion, rendering those operation more costful.

B. TLB Size Sensitivity Test for All Benchmarks

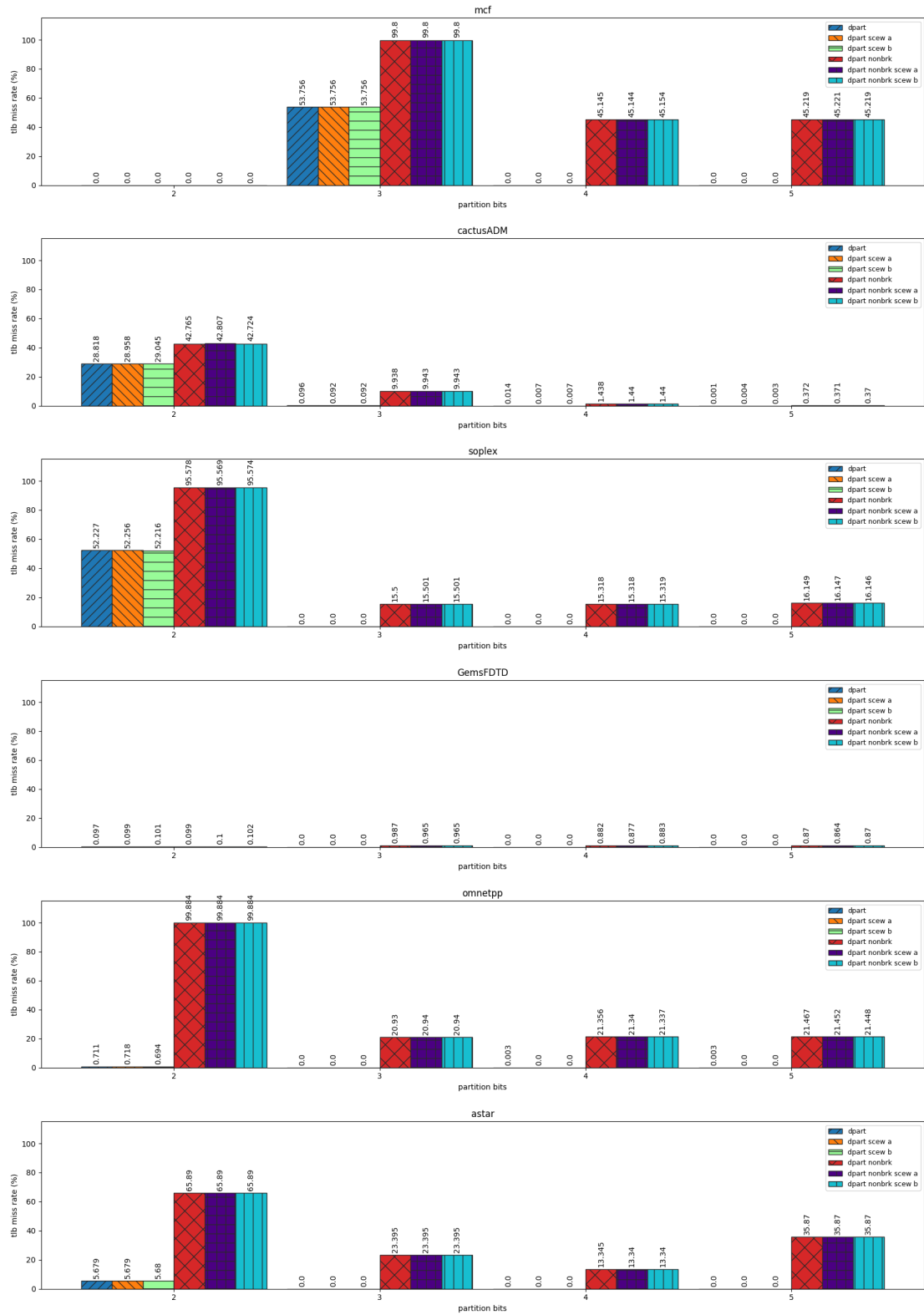


Figure B.1: TLB miss rates for variant partitioning bits

C. TLB Size Sensitivity Test for All Benchmarks

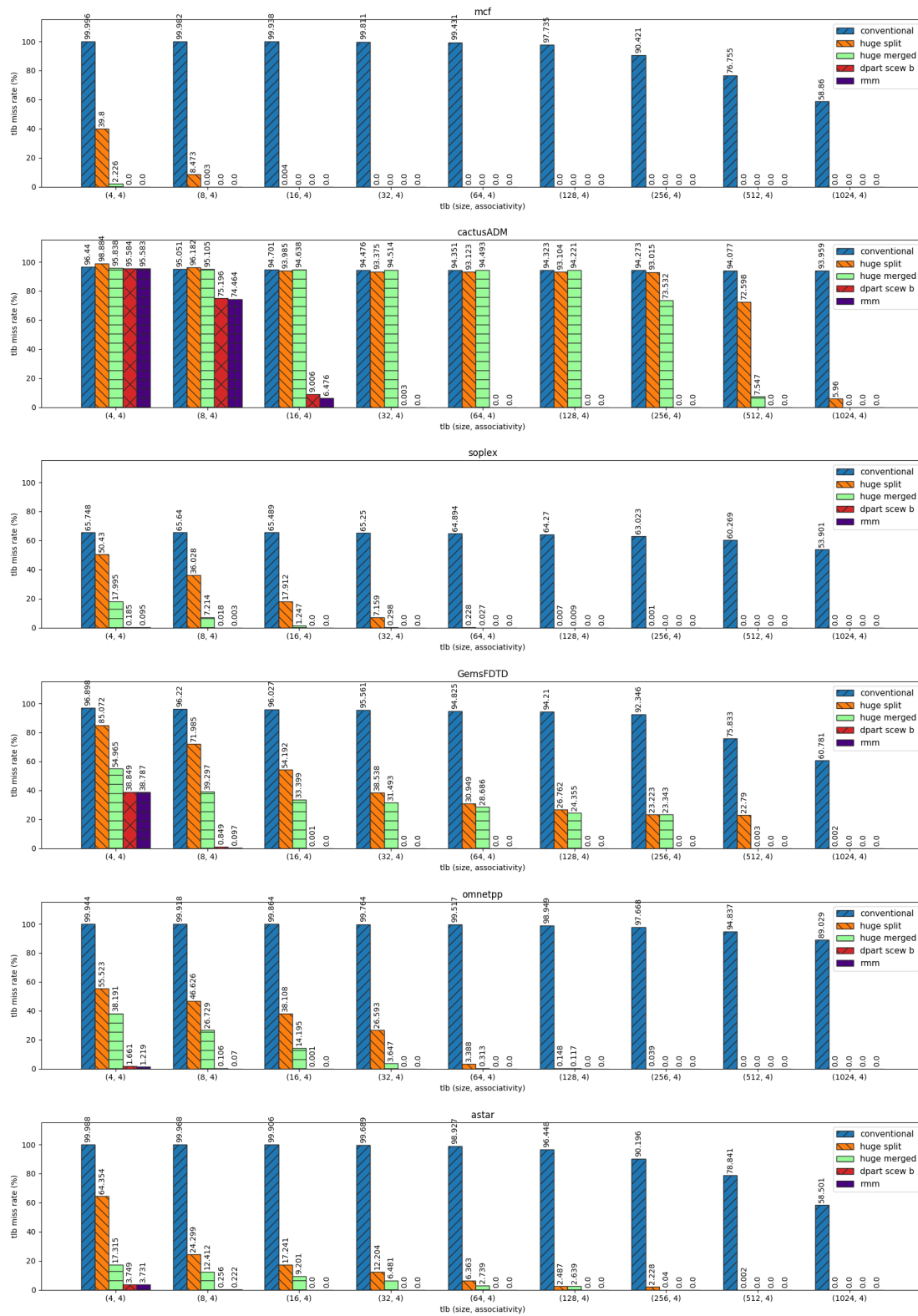


Figure C.1: TLB miss rates for 5 partitioning bits on variant size

D. TLB Associativity Sensitivity Test for All Benchmarks

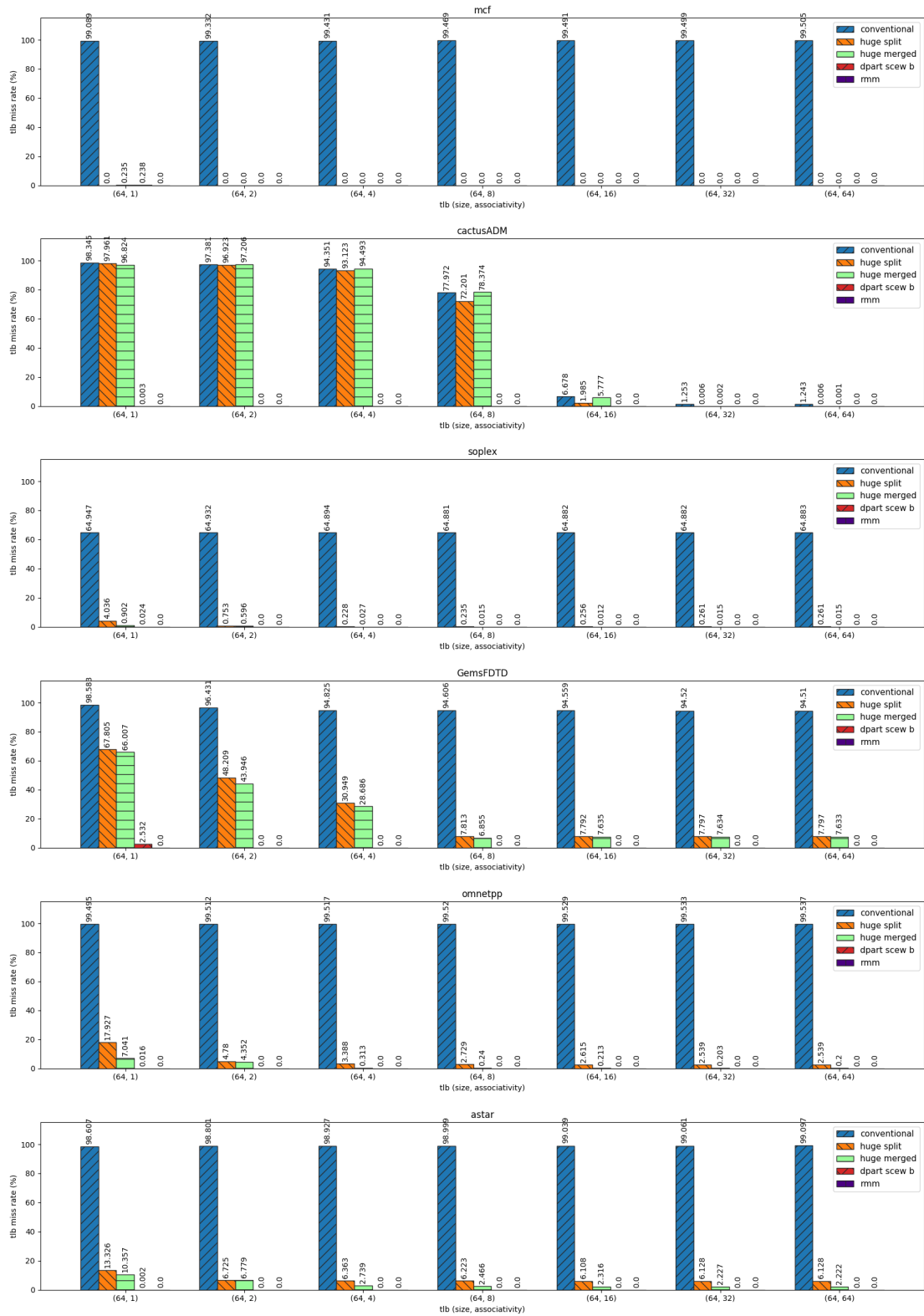


Figure D.1: TLB miss rates for 5 partitioning bits on variant associativity

