
**National Technical University of Athens**

School of Electrical and Computer Engineering

Division of Computer Science


# Hardware and Software Co-Design for Efficient Memory Access

Doctoral Dissertation


**CHLOE ALVERTI**



Athens, 5 December 2022

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

# Hardware and Software Co-Design for Efficient Memory Access

Doctoral Dissertation

**CHLOE ALVERTI**

**Advisors:**

Georgios Goumas
Nectarios Koziris
Vasileios Karakostas

| | | |
|---|---|---|
| . . . . . | . . . . . | . . . . . |
| Georgios Goumas<br>Associate Professor<br>National Technical<br>University of Athens | Nectarios Koziris<br>Professor<br>National Technical<br>University of Athens | Vasileios Karakostas<br>Assistant Professor<br>University of<br>Athens |
| . . . . . | . . . . . | . . . . . |
| Dionisios N. Pnevmatikatos<br>Professor<br>National Technical<br>University of Athens | Dimitrios Tsoumakos<br>Professor<br>National Technical<br>University of Athens | Angelos Bilas<br>Professor<br>University of<br>Crete |
| | . . . . . | |
| | Michael Swift<br>Professor<br>University of Wisconsin-<br>Madison | |

Athens, 5 December 2022

. . . . . .

**Chloe Alverti**
**Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.**

Copyright © Chloe Alverti
Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π., 2022
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

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

*To my parents, Lena and Nikos, who have and always will be a core part of myself,*
*my role models and an endless source of love and support.*
*I have been truly lucky.*

*Στους γονείς μου, Λένα και Νίκο, που είναι και θα είναι πάντα κομμάτι μου,*
*μια τρομερή πηγή αγάπης και υποστήριξης.*
*Έχω υπάρξει πολύ τυχερή.*

# Περίληψη

Η εικονική μνήμη είναι μια κρίσιμη υπολογιστική αφαίρεση που έχει αντέξει στη δοκιμασία του χρόνου. Διευκολύνει τον προγραμματισμό δημιουργώντας την ψευδαίσθηση ότι η φυσική μνήμη είναι τεράστια, γραμμική και ιδιωτική ανά διεργασία, επιτρέπει την πρόσβαση σε συσκευες Ε/Ε στο χώρο της μνήμης και βοηθά στην ευέλικτη διαχείριση πόρων. Ωστόσο, αυτές οι θεμελιώδεις ιδιότητες δεν παρέχονται δωρεάν. Η εικονική μνήμη προϋποθέτει ότι το Λειτουργικό Σύστημα (ΛΣ) δημιουργεί και διαχειρίζεται αυτήν την αφαίρεση της φυσικής μνήμης που αναγνωρίζει κάθε διεργασία, τον λεγόμενο εικονικό χώρο διευθύνσεων, και τον αντιστοιχεί σε πραγματικούς φυσικούς πόρους. Η εικονική μνήμη επίσης επιβάλλει ότι κάθε λειτουργία πρόσβασης στη μνήμη (του επεξεργαστή) περνάει από ένα βήμα μετάφρασης. Κανένας από τους παραπάνω μηχανισμούς δεν είναι φθηνός και, για την ακρίβεια, το κόστος τους συνεχώς αυξάνεται.

Υπάρχουν τέσσερις τάσεις που στρεσάρουν την απόδοση της εικονικής μνήμης σήμερα, (i) η μετεωρική άνοδος των απαιτήσεων χωρητικότητας μνήμης, (ii) η σεισμική μετατόπιση των χρηστών στη χρήση του υπολογιστικού νέφους (cloud), (iii) η ταχεία εξέλιξη των συσκευών αποθήκευσης με όρους απόδοσης και (iv) η αυξανόμενη ετερογένεια των συσκευών υπολογισμού και αποθήκευσης στα συστήματα μεγάλων δεδομένων. Οι δύο πρώτες ανεβάζουν σημαντικά τον πήχη απόδοσης της διαδικασίας μετάφρασης εικονικών διευθύνσεων σε φυσικές και οι δύο επόμενες μας προτρέπουν να ξανασκεφτούμε τη σημασιολογία και την υλοποίηση των διεπαφών της εικονικής μνήμης. Η παρούσα διατριβή συμβάλλει και προς τις δύο κατευθύνσεις.

Η εκθετική αύξηση του όγκου των παγκόσμιων δεδομένων και η αντίστοιχη αύξηση των απαιτήσεων μνήμης των εφαρμογών, οδήγησαν την κυρίαρχη υλοποίηση της εικονικής μνήμης −τη σελιδοποίηση− στο λεγόμενο Τείχος Μετάφρασης (Address Translation Wall) [53] περίπου μια δεκαετία πριν. Στην παρούσα διατριβή δείχνουμε ότι παρά το γεγονός ότι το υλικό μετάφρασης ανα επεξεργαστή τριαπλασσιάστηκε από τότε, π.χ. με την ενσωμάτωση μεγαλύτερων κρυφών μνημών αναζήτησης μετάφρασης (TLBs) η με την καλύτερη υποστήριξη των μεγάλων σελίδων, οι εφαρμογές μεγάλης έντασης δεδομένων (big data) μπορεί ακόμα να ξοδεύουν έως και το 30% του χρόνου εκτέλεσής τους στη μετάφραση

διευθύνσεων – ειδικά όταν εκτελούνται σε εικονικά περιβάλλοντα. Για την αντιμετώπιση της κακής αυτής κλιμάκωσης απόδοσης της σελιδοποίησης, η παρούσα διατριβή προτείνει συνεργιστικούς μηχανισμούς λογισμικού και υλικού που δημιουργούν και εκμεταλλεύονται την ύπαρξη γραμμικότητας στις αντιστοιχίσεις εικονικών σελίδων σε φυσικές. Στο λογισμικό, προτείνουμε την *Σελιδοποίηση με επίγνωση γειτνίασης (CA paging)*, μια νέα τεχνική διαχείρισης μνήμης που βελτιώνει τον χειριστή σφαλμάτων σελίδας του ΛΣ με υποδείξεις για την δέσμευση κατάλληλων σελίδων για τη δημιουργία μεγάλων συνεχόμενων αντιστοιχίσεων εικονικών διευθύνσεων σε φυσικές ανά διεργασία. Η σελιδοποίηση CA εφαρμόζεται τόσο σε φυσικές όσο και σε εικονικές μηχανές και διατηρεί τις ευέλικτες τεχνικές διαχείρισης μνήμης ενός σύγχρονου ΛΣ, π.χ. τη σελιδοποίηση κατ'απαίτηση και την αντιγραφή κατά την εγγραφή (CoW), ενώ αποφεύγει κάθε είδους εκ των προτέρων δέσμευση μνήμης. Υλοποιήσαμε την σελιδοποίηση CA στο ΛΣ Linux και τη διαθέτουμε ως λογισμικό ανοικτού κώδικα. Στο υλικό, για την αξιοποίηση της παραγόμενης γραμμικότητας στις απεικονίσεις, προτείνουμε την κερδοσκοπική μετάφραση διευθύνσεων με βάση τη μετατόπιση (SpOT). Το SpOT είναι μια μικρο-αρχιτεκτονική επέκταση που εκμεταλλεύεται την υποκείμενη γραμμικότητα στις αντιστοιχίσεις για να προβλέψει τη μετάφραση διευθύνσεων σε περίπτωση αστοχίας στην ιεραρχία των κρυφών μνημών TLB. Το SpOT μπορεί να εφαρμοστεί άμεσα και με διαφάνεια τόσο σε φυσικά όσο και σε εικονικά περιβάλλοντα – επειδή λειτουργεί εξ ολοκλήρου σε επίπεδο μικροαρχιτεκτονικής. Σε συνδυασμό με τη σελιδοποίηση CA, το SpOT μειώνει το κόστος μετάφρασης από ∼16,5% σε ∼0,9% κατά μέσο όρο για εφαρμογές μεγάλων δεδομένων που εκτελούνται σε εικονικές μηχανές, ανταλλάσοντας τις ισχυρές εγγυήσεις ασφάλειας (security) με εναν απλό αρχιτεκτονικό σχεδιασμό.

Η εικονική μνήμη, εκτός από προγραμματιστική αφαίρεση για τη φυσική μνήμη, είναι επίσης μια σημαντική διεπαφή για την πρόσβαση στις συσκευές εισόδου-εξόδου (Ε/Ε). Οι αντιστοιχίσεις αρχείων επιτρέπουν στις εφαρμογές να έχουν πρόσβαση σε μόνιμα δεδομένα μέσω αναφορών στη μνήμη. Ωστόσο, οι συσκευές αποθήκευσης υψηλής απόδοσης έχουν εξελιχθεί σημαντικά την τελευταία δεκαετία και στις μέρες μας προσφέρουν χρόνους απόκρισης μονοψήφιους ή ακόμα και μικρότερους του δευτερολέπτου, εκθέτοντας το λογισμικό συστήματος Ε/Ε του ΛΣ ως απαγορευτικά ακριβό. Στη παρούσα διατριβή, μελετάμε την περίπτωση συσκευών μη πτητικής μνήμης (PMem) και της διεπαφής αρχείων άμεσης πρόσβασης (DAX). Με το PMem και το DAX, η εικονική μνήμη μπορεί να απεικονίσει φυσικές διευθύνσεις αποθήκευσης μόνιμων δεδομένων απευθείας στο χώρο του χρήστη, επιτρέποντας τη πρόσβαση σε μόνιμα δεδομένα μέσω εντολών

load/store του επεξεργαστή. Ωστόσο, στη μελέτη μας διαπιστώνουμε ότι οι λειτουργίες της εικονικής μνήμης συχνά μειώνουν την απόδοση της άμεσης πρόσβασης, αποτυγχάνοντας να προσφέρουν αυτό που μπορεί να προσφέρει το υποκείμενο υλικό. Στη παρούσα διατριβή αναλύουμε όλες τις πηγές κόστους στη χρήση της μνήμης ως διεπαφής αρχείων και μελετάμε πως επηρεάζονται οι δαπανηροί μηχανισμοί της εικονικής μνήμης από νέες τεχνολογίες αποθήκευσης ή και εαν ακόμα γίνονται απαρχαιωμένοι. Με βάση την ανάλυσή μας, προτείνουμε μια νέα διεπαφή για γρήγορη και κλιμακώσιμη άμεση πρόσβαση σε μόνιμα δεδομένα (DaxVM). Το DaxVM είναι μια διεπαφή απεικόνισης αρχείων αποθηκευμένων σε μη-πτητικες μνήμες, που χαλαρώνει τη σημασιολογία POSIX, και υλοποιείται με επανασχεδιασμό των λειτουργιών της εικονικής μνήμης και με επέκταση των συστημάτων αρχείων για PMem – με όλες τις αλλαγές να καθοδηγούνται από τα μοναδικά χαρακτηριστικά της άμεσης πρόσβασης (dax). Το DaxVM υποστηρίζει (i) γρήγορες λειτουργίες αντιστοίχισης μνήμης (O(1)) μέσω μόνιμων πινάκων σελίδων ενσωματωμένων στα μεταδεδομένα του συστήματος αρχείων, (ii) τη νωχελική ακύρωση των TLB, (iii) την κλιμακώσιμη (σε πολλούς πυρήνες) διαχείριση του εικονικού χώρου διευθύνσεων για εφήμερες αντιστοιχίσεις, (iv) την εξάλειψη του κόστους δυνατότητας διαχείρισης της ανθεκτικότητας των μόνιμων δεδομένων από τον χώρο του πυρήνα όταν είναι υπεύθυνος ο χώρος χρήστη και (v) τον ασύγχρονο μηδενισμό των μπλοκ αποθήκευσης από το σύστημα αρχείων. Υλοποιήσαμε το DaxVM στο ΛΣ Linux και στα συστήματα αρχείων ext4-DAX και NOVA και το διαθέτουμε ως λογισμικό ανοιχτού κώδικα. Το αξιολογούμε σε ένα πραγματικό σύστημα εξοπλισμένο με Intel Optane. Για εφαρμογές πολλαπλών νημάτων που επεξεργάζονται πολλά μικρά αρχεία για μικρά διαστήματα, π.χ. Apache, το DaxVM βελτιώνει την απόδοση της κλήσης συστήματος mmap έως και 4,9x. Αντιστρέφει επίσης την τάση που ευνοεί τη χρήση της κλήσης συστήματος read για τέτοιες εφαρμογές, ξεπερνώντας την έως και 1,5x. Το DaxVM αυξάνει επίσης τη διαθεσιμότητα του συστήματος, παρέχοντας γρήγορους χρόνους εκκίνησης για βάσεις δεδομένων και διατηρεί υψηλή απόδοση ακόμα και όταν οι συσκευές αποθήκευσης υποφέρουν απο εξωτερικό κατακερματισμό.

Συνοψίζοντας, η παρούσα διατριβή επανεξετάζει τη σχεδίαση και την υλοποίηση του μηχανισμού της εικονικής μνήμης στο σήμερα και προτείνει τεχνικές στο υλικό και στο λογισμικό που την επεκτείνουν ωστε να i) κλιμακώνει καλύτερα με τις συνεχώς αυξανόμενες χωρητικότητες της κύριας μνήμης μέσω ενός αποδοτικού μηχανισμού μετάφρασης διευθύνεσων και ii) προσφέρει μια ειδική διεπαφή αντιστοίχισης αρχείων που φέρνει την απόδοση στα όρια αυτού που μπορεί να παρέχει το υποκείμενο υλικό για άμεση πρόσβαση σε μόνιμα δεδομένα.

**Λέξεις κλειδιά:** εικονική μνήμη, διαχείριση φυσικής μνήμης, μη-πτητική μνήμη, συστήματα αρχείων, λειτουργικά συστήματα, αρχιτεκτονική υπολογιστών

# Abstract

Virtual memory is a crucial computing abstraction that has stood the test of time. The level of indirection that it introduces, facilitates programming, i.e. creates the illusion that physical memory is vast, linear and private per application or enables the access of I/O devices in memory space, and assists agile resource management. However these fundamental properties do not come for free. Virtual memory assumes that the Operating System (OS) must maintain the abstraction of memory that each process acknowledges, the virtual address space indirection, and map it to actual physical resources. It also assumes that each CPU memory access operation must go through a translation step. None of the above mechanisms is cheap and, if anything, their costs are getting more and more profound.

There are four trends that stress the performance of virtual memory today, (i) the meteoric rise in memory demands and capacities, (ii) the seismic shift of users from enterprise data centers to the cloud, (iii) the rapid evolution of high-performance storage devices and (iv) the increasing heterogeneity in both the compute and the store landscape of data-center systems. The first two considerably raise the efficiency bar for address translation and the second two urge us to re-visit the legacy virtual memory interfaces semantics and consecutively their design. This thesis contributes in both directions.

The exponential growth of global data and the corresponding increase in the memory demands of workloads led virtual memory's dominant implementation – paging– hit the Address Translation Wall [53] almost a decade ago. In this thesis we show that despite the fact that vendors tripled translation hardware budget since then, e.g. by incorporating larger TLBs and MMU caches or better huge page support, memory-intensive workloads can still spend up to 30% of their execution time in address translation – especially when they run in virtualized environments. To deal with paging's poor performance scaling, this thesis proposes synergistic software and hardware mechanisms that create and exploit linearity in mappings. We propose *Contiguity-Aware (CA) paging*, a novel memory management technique that enhances the Operating System's page fault handler with hints to allocate target pages and create vast contiguous virtual-to-physical mappings per process. CA paging is applicable to both native and nested paging and it maintains all lightweight

memory management techniques of a modern OS, i.e demand paging, Copy-On-Write etc, while avoiding any memory reservation or pre-allocation. We implement our proposal in stock Linux and make it publicly available. On the hardware side, to harvest the generated contiguity, we propose Speculative Offset Address Translation (SpOT). SpOT is a micro-architecture engine that exploits the underlying linearity in mappings to predict address translation on the TLB miss path. While most state-of-the-art hardware proposals fail to support virtualization, due to the architectural complexity of tracking and caching arbitrarily sized mappings in two-dimensional execution, SpOT is directly and transparently applicable to both native and virtualized environments – because it works entirely on the micro-architecture level. Combined with CA paging, SpOT reduces the translation overheads of nested paging from ∼16.5% to ∼0.9% on average for memory-intensive workloads, in a design that trades architectural complexity with strong security guarantees.

Apart from a physical memory abstraction, virtual memory is also an important interface towards IO devices; file mappings allow applications to access persistent data via memory dereference. However, high-performance storage has evolved significantly the past decade and nowadays devices offer single digit or even sub- microsecond latencies, exposing the kernel software IO stack as a prohibitively expensive data path. In this thesis, we study the case of persistent memory (PMem) and the direct access file interface (DAX). With PMem and DAX, virtual memory can map storage locations directly to user-space, enabling persistent data access via CPU load/store instructions; forming the shortest existing path to storage. Yet in our study we find that virtual memory operations often throttle direct access performance, failing to deliver what the underlying hardware can provide. In this thesis we break down all sources of overhead in using memory as a file interface. We study how the expensive mechanisms of virtual memory are affected by the new fast storage technology or if they even become obsolete. Based on our analysis, we propose a new interface for fast and scalable direct access to persistent data. DaxVM is a POSIX-relaxed file mapping interface for persistent memory, implemented by redesigning virtual memory operations and extending PMem-aware file systems – all changes driven by direct access unique characteristics. DaxVM supports (i) O(1) memory mapping operations via persistent page tables integrated in file system's inode metadata, (ii) lazy invalidation of the TLBs, (iii) scalable address space management for ephemeral mappings, (iv) elimination of kernel-space durability management support when user-space is in charge and (v) asynchronous storage block pre-zeroing by the file system to accelerate DAX append operations. We implement

DaxVM in stock Linux and the ext4-DAX and NOVA file systems and make it publicly available. We evaluate it on a real system equipped with Intel Optane. For multi-threaded workloads that process multiple small files for short intervals, e.g., Apache, DaxVM improves standard mmap performance up to 4.9x. It also reverses the trend that favors read for such setups, outperforming it by up to 1.5x. DaxVM also increases system availability, providing fast boot times for PMem databases, and sustains high throughput even when they run on fragmented file system images.

Overall, this thesis revisits today's virtual memory design and proposes hardware and software techniques that extend it to (i) scale better with the ever increasing memory capacities, through efficient address translation, and (ii) form a dedicated file mapping interface to push performance to the limits of what the underlying hardware can provide for direct access to persistent data.

**Keywords:** virtual memory, memory management, address translation, persistent memory, file systems, operating systems, computer architecture

# Ευχαριστίες

Η παρούσα εργασία είναι το απτό, τελικό αποτέλεσμα των διδακτορικών σπουδών μου των τελευταίων 5μιση ετών στο Εργαστήριο Υπολογιστικών Συστημάτων (CSLab) της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου. Στην ουσία όμως ειναι το αποτέλεσμα μιας πολύ μεγαλύτερης περιόδου η οποία ξεκινάει από το ίδιο ακριβώς εργαστήριο το 2014. Ολοκληρώνοντας αυτό το −τόσο σημαντικό για μένα− ταξίδι θα ήθελα να επιχειρήσω τη μερική καταγραφή του και κυρίως να ευχαριστήσω όλους τους ανθρώπους που συνάντησα στην πορεία του. Ο καθένας με το διακριτό του ρόλο με ενέπνευσε, με στήριξε και τελικά καθόρισε την πορεία της εργασίας μου. Δημιουργήθηκαν έτσι ισχυροί δεσμοί και σχέσεις ζωής που αποτελούν τελικά το σημαντικότερο επίτευγμα και όλη την ουσία. *Ευχαριστώ λοιπόν από τα βάθη της καρδιάς μου:*

- **..τους επιβλέποντες και τα μέλη της επιτροπής μου**

  – *Γεώργιο Γκούμα (επιβλέποντα καθηγητή)*

  Τον γνώρισα στο τέλος των προπτυχιακών μου σπουδών, το 2014, όταν εργάστηκα μαζί του στη διπλωματική μου εργασία στο CSLab. Του χρωστώ λοιπόν, πριν από όλα, την πρώτη μου επαφή με την ερευνητική διαδικασία και τον πρώτο ενθουσιασμό για το καινούργιο αυτό για μένα −τότε− σύμπαν. Μου έδειξε εμπιστοσύνη και έκτοτε η υποστήριξή του ήταν συνεχής σε όλα τα επίπεδα. Με το πέρας των προπτυχιακών μου σπουδών, με ενεθάρρυνε να ακολουθήσω διδακτορικές σπουδές, όπως ήθελα, και γράφτηκα ως ΥΔ στο εργαστήριο μας. Με παρότρυνε στην εμβόλιμη επιθυμία και επιλογή μου να φύγω για μερικά χρόνια στη Σουηδία και να εργαστώ ως ερευνητικό προσωπικό. Το 2017, ενίσχυσε την απόφασή μου να επιστρέψω και να ξεκινήσω τη διατριβή μου. Τα τελευταία 5μιση χρόνια έχει σταθεί

στο πλάι μου σε κάθε μου ερευνητική προσπάθεια και μου έχει χαρίσει απλόχερη ελευθερία, δίνοντάς μου μεταξύ άλλων και το χώρο να εξερευνήσω ερευνητικά μονοπάτια τα οποία δεν ήταν απαραίτητα εντός των άμεσων ερευνητικών του ενδιαφερόντων. Πέρα όμως από την συνεργασία και επίβλεψη, είμαι ευγνώμων, γιατί βρέθηκε κοντά μου και ως άνθρωπος. Δεν με πίεσε ποτέ, προσπάθησε πάντα να με καταλάβει και να με γνωρίσει, στάθηκε πάντα με διάθεση φροντίδας απέναντι σε κάθε δυσκολία. Έτσι μέσα στα χρόνια, έχει δημιουργηθεί ένας ισχυρός δεσμός –με όλη την ιδιαιτερότητα και μαγεία που έχει πάντα η σχέση δάσκαλου και μαθητή– ο οποίος εύχομαι να κρατήσει για πολύ, πολύ καιρό ακόμα.

– *Βασίλειο Καρακώστα (συν-επιβλέποντα),*

μεταδιδακτορικό ερευνητή στο εργαστήριο μας κατά τη διάρκεια των διδακτορικών σπουδών μου και καθηγητή σήμερα στο ΕΚΠΑ. Ο Βασίλης είναι από τους πιο σημαντικούς ανθρώπους αυτής της περιόδου. Συν-επέβλεψε το διδακτορικό μου και συνεργαστήκαμε πολύ στενά από τη πρώτη κιόλας μέρα, δουλεύοντας σε μια ερευνητική κατεύθυνση που προέκυψε από τη δική του ερευνητική εμπειρία. Κάθε εργασία, κάθε βήμα του διδακτορικού μου το κάναμε μαζί και αυτό είχε δύο εξίσου σημαντικά αποτελέσματα. Πρώτον, χάρη στο Βασίλη και την καθοδήγησή του έμαθα πάρα πολλά και απέκτησα δεξιότητες τόσο τεχνικές όσο και μεθοδολογικές για τη σωστή διεξαγωγή της έρευνας – γνώσεις πολύτιμες που θα με συντροφεύουν για πάντα. Δεύτερον, χάρη (και) στο Βασίλη ,το διδακτορικό μου έγινε μια φοβερά διασκεδαστική και ενθουσιώδης εμπειρία, καθώς μαζί του μοιράστηκα κάθε στιγμή της. Πάντα με θετική στάση απέναντι στα πράγματα, ακόμα και στις αναποδιές, με έναν πηγαίο ενθουσιασμό για την έρευνα, με μόνιμο ενδιαφέρον ο συνεργάτης του να τα καταφέρει, χωρίς να πιέζει και χωρίς να απαιτεί, με τεράστια υπομονή και επιμονή, με μια μόνιμη παρακίνηση και προτροπή για νέους και υψηλότερους στόχους, ήταν ο καλύτερος συνοδοιπόρος που θα μπορούσε κανείς να έχει. Η σχέση μας έγινε με τα χρόνια προσωπική. Με έχει ζήσει στις θετικές και στις αρνητικές διαθέσεις μου, έχει υπομείνει το άγχος και την ένταση μου, με έχει στηρίξει στις αναποδιές και στις προσωπικές ανηφόρες, έχει ξενυχτήσει μαζί μου σε κάθε προθεσμία.... Έτσι, πέρα από την εξαιρετική συνεργασία ,που ελπίζω να συνεχιστεί, νιώθω ότι έκανα και έναν φίλο ζωής. Ανυπομονώ, λοιπόν, για τα επόμενα.

– *Νεκτάριο Κοζύρη (καθηγητή ΕΜΠ),*

ο οποίος με την αγάπη του και το ενδιαφέρον του για την επιστήμη των υπολογιστών και τη σύγχρονη έρευνα φρόντισε να έχουμε πάντα στη διάθεση μας εξοπλισμό αιχμής – κάτι που έδρασε ως καταλύτης και για τη δική μου ερευνητική εργασία. Στις πολύτιμες συμβουλές του έχω ανατρέξει πολλές φορές όλα αυτά τα χρόνια, τόσο για τη δουλειά μου, όσο και για κάθε σημαντική επαγγελματική απόφαση που χρειάστηκε να πάρω.

– *Διονύσιο Πνευματικάτο (καθηγητή ΕΜΠ),*

τόσο για τα πολύτιμα σχόλιά του για τη διατριβή μου όσο και για τις επαγγελματικές του συμβουλές τις οποίες έχω επίσης πολλάκις ζητήσει.

– *Δημήτριο Τσουμάκο και Άγγελο Μπίλα (καθηγητές ΕΜΠ και Πανεπιστημίου Κρήτης),*

για το χρόνο τους και τα πολύτιμα σχόλιά τους ως μέλη της επταμελούς επιτροπής μου.

– *τον Michael Swift (καθηγητής UWM),*

από το University of Wisconsin-Madison (UWM) και τελευταίο μέλος της επιτροπής μου. Ο κύριος Swift εξελίχθηκε σε έναν άνθρωπο εξαιρετικής σημασίας για τη διατριβή μου και όχι μόνο. Το 2019 επισκέφθηκα για 3 μήνες το UWM για να δουλέψω μαζί του, μέσω μιας ευρωπαϊκής υποτροφίας (HiPEAC). Αυτό το ταξίδι στην Αμερική είναι μια από τις πιο ζωντανές και έντονες εμπειρίες του διδακτορικού μου. Έμαθα πολλά, ήρθα σε επαφή με πολλούς ερευνητές, διεύρυνα τους ερευνητικούς μου ορίζοντες και ξεκίνησα κάτι τελείως καινούργιο (μια νέα εργασία) στο πλαίσιο της διατριβής μου. Τότε ξεκίνησε μια μακρά συνεργασία, η οποία κράτησε για πάνω από δύο χρόνια και συνεχίζεται μέχρι και σήμερα. Υπό την καθοδήγηση του κ. Swift έμαθα πολλά για τη μέθοδο, την επιμονή, την υπομονή, τη λεπτομέρεια και την εμβάθυνση που απαιτεί η ερευνητική διαδικασία και αυτή η εμπειρία είναι ένα από τα πολυτιμότερα «εργαλεία» που απέκτησα τα χρόνια του διδακτορικού μου. Πέρα από την τεχνική και μεθοδολογική του καθοδήγηση, δεν θα ξεχάσω ποτέ την ευγένεια και την καλοσύνη του. Μέσα στις αντιξοότητες της πανδημίας και παρά την εξ αποστάσεως συνεργασία, η υποστήριξη του ήταν μεγάλη σε όλα τα επίπεδα. Με συγκίνησε και με τίμησε, ιδιαίτερα, που παρευρέθηκε δια ζώσης στην στην υποστήριξη της διατριβής μου.

- **...τους πολύτιμους στενούς συνεργάτες**

  – *Στράτο Ψωμαδάκη,*

  ΥΔ στο εργαστήριο μας, κυρίως όμως ξεχωριστό και πολύ αγαπημένο άνθρωπο. Συνεργαστήκαμε πολύ στενά όλα τα χρόνια του διδακτορικού μου και είναι κι αυτός ένας από τους πολυτιμότερους ανθρώπους αυτής της περιόδου. Του οφείλω ειλικρινά πάρα πολλά. Πέρα από τις κοινές μας δουλειές, κάθε ιδέα μου τη συζήτησα μαζί του, σε κάθε τεχνική απορία/δυσκολία μου έτρεχα να με σώσει. Αλλά δεν είναι μόνο η συνεργασία και η πρακτική βοήθεια στην ερευνητική δουλειά, ο Στράτος υπήρξε πάνω και πέρα από όλα ένα τεράστιο ψυχολογικό στήριγμα. Σε κάθε απογοήτευση, κάθε ματαίωση, κάθε αναποδιά, που μοιραία ειναι μέρος μιας διατριβής, ο Στράτος με βοήθησε πολύ να ξανασταθώ όρθια. Επίσης ήταν και είναι έμπνευση. Η οξυδέρκειά του και η βαθειά γνώση του για το αντικείμενο μας, με κάνει πάντα να ενθουσιάζομαι και να κινητοποιούμαι. Τον ευχαριστώ απο καρδιάς για όλα.

– *Κωσταντίνο Νίκα,*

μεταδιδακτορικό ερευνητή στο εργαστήριο μας, με τον οποίο συνεργάστηκα στενά τόσο στο πλαίσιο της διατριβής, όσο και στα χρόνια των προπτυχιακών μου σπουδών για τη διπλωματική μου εργασία.

– *Jayneel Gandhi,*

ερευνητή στη Meta και στενό συνεργάτη των τελευταίων χρόνων. Πέρα από τη συνεργασία, θα ήθελα να τον ευχαριστήσω θερμά και για την ξεχωριστή ευκαιρία για πρακτική εργασία στην ομάδα του, τότε, στη Vmware Research, η οποία δυστυχώς δεν πραγματοποιήθηκε ποτέ λόγω της πανδημίας...

– *Χρήστο Κατσακιώρη,*

ΥΔ στο εργαστήριο μας, με τον οποίο συνεργαστήκαμε στενά στο τέλος του διδακτορικού μου. Η εμπιστοσύνη που μου έδειξε και δείχνει με τιμά και με κινητοποιεί. Έμαθα πολλά από τον ίδιο, τεχνικά και μεθοδολογικά για το αντικείμενο μας, και ελπίζω να συνεχίσουμε να δουλεύουμε μαζι!

– *τον Sujay Yadalam,,*

ΥΔ στο University of Wisconsin-Madison, με τον οποίο επίσης συνεργαστήκαμε τον τελευταίο χρόνο και είναι το προσωπικό μου βιωματικό παράδειγμα ότι στην έρευνα η απόσταση εκμηδενίζεται! Με έμαθε πολλά για τη πειραματική διαδικασία και η όρεξή του για τη δουλειά μου υπενθύμιζε πάντα ότι υπάρχει χαρά σε ό,τι κάνουμε. Του εύχομαι τα καλύτερα.

– *τους προπτυχιακούς φοιτητές που έκαναν τη διπλωματική τους με την ομάδα μας,*

τον Παναγιώτη Μπάκο, τον Θωμά Τσαπράλη, τον Ιάσονα Μαρμάνη, την Ελένη Μιχαλάκη, τον Βασίλη Πολίτη, τον Χρήστο Λούρα και τον Φίλιππο Τόφαλο. Η συνεργασία με τον καθένα ήταν εξαιρετική και έμαθα κι εγώ πολλά –τεχνικά και μεθοδολογικά– που ήταν πολύτιμα για το διδακτορικό μου!

- **...τους αγαπημένους συνάδελφους του εργαστηρίου μας**

  Τη Νικέλα Παπαδοπούλου με την οποία καταλήξαμε να έχουμε γειτονικά γραφεία στο εργαστήριο μας, αλλά η φιλία μας ξεκινά από πολύ πιο πριν. Είναι από τους σημαντικότερους ανθρώπους στη ζωή μου και ανεκτίμητη συνάδελφος και συνεργάτης. Τον Δημήτρη Σιακαβάρα, τον οποίο γνώρισα στο εργαστήριο και εξελίχθηκε σε πολύτιμο φίλο – μια από τις μεγαλύτερες χαρές αυτών των χρόνων. Από τους ευγενέστερους και δοτικότερους ανθρώπους που έχω τη τύχη να ξέρω – μόνιμο στήριγμα για καθε ΥΔ στο εργαστήριο μας ως admin. Τον Κωστή Παπαζαφειρόπουλο, ο οποίος είναι πρώτα από όλα παλιός φίλος και δευτερευόντως συνάδελφος. Στα 15 χρόνια που τον γνωρίζω έχω μοιραστεί κάθε χαρά και κάθε λύπη μου μαζί του και σκοπεύω να συνεχίσω. Τον Ορέστη Λάγκα Νικολό, τον οποίο γνωρίζω από παλιά και στα

χρόνια του διδακτορικού μας μοιραστήκαμε σκέψεις και ανησυχίες –επαγγελματικές και μη– τόσο που τον θεωρώ φίλο ζωής. Τον Παναγιώτη Μπάκο, ο οποίος εκτός από εξαιρετικός και υποστηρικτικός συνεργάτης, είναι και απίστευτος άνθρωπος. Νιώθω τυχερή που τον γνώρισα και ανυπομονώ για τα επόμενα. Τον Ανδρέα Μπινίσκο, ο οποίος είναι αγαπημένος συνάδελφος, φίλος και σημείο αναφοράς για όλους μας. Τον Γιάννη Παπαδάκη, όπου εκτός των άλλων του χρωστάω –μαζί με τον Χρήστο και τον Παναγιώτη– ότι πέρασα τα περισσότερα μαθήματα του διδακτορικού! Την Ιωάννα Αλιφιεράκη, με την οποία γνωριστήκαμε κάνοντας τη διπλωματική μας εργασία και είναι κι αυτή μέλος αυτής της ισχυρής παρέας/οικογένειας. Τον Στέφανο Γεράγγελο και την Αθηνά Ελαφρού, που μοιραστήκαμε για χρόνια το ίδιο γραφείο και με βοήθησαν στη διατριβή μου. Τέλος, θέλω να ευχαριστήσω απο καρδιάς και όλους τους υπόλοιπους ΥΔ και συνάδελφους του εργαστηρίου μας, χάρη στους οποίους τα χρόνια της διατριβής ήταν τόσο όμορφα.

- **...τους συναδέλφους από τη Σουηδία**

  Τον καθηγητή Per Stenstrom, ο οποίος με μεγάλη φροντίδα επέβλεψε και καθοδήγησε την ερευνητική μου εργασία το διάστημα 2015-2017 στο Chalmers University of Technology της Σουηδίας. Αυτή η εμπειρία ήταν καταλύτης για την μετέπειτα επαγγελματική και όχι μόνο ζωή μου. Με γέμισε ενθουσιασμό για την έρευνα, με δίδαξε και επέκτεινε τους ακαδημαικούς μου ορίζοντες. Έμμεσα είναι κι αυτός κομμάτι αυτής της διατριβής, καθώς εκείνα τα χρόνια έθεσε πολύ ισχυρά θεμέλια.

  Τον Άγγελο Αρελάκη και τον Γιάννη Νικολακόπουλο με τους οποίους συνεργάστηκα στενά και απέκτησα γνώσεις πολύτιμες για τη παρούσα διατριβή. Τον Πέτρο Βουδούρη, με τον οποίο μοιράστηκα αυτή την μοναδική εμπειρία των χρόνων στη Σουηδία (επαγγελματική και μη) και ήταν, είναι και θα είναι για μένα κάτι σαν οικογένεια (ακόμα και αν χανόμαστε). Τον Paul Renaud Goud, τον Aras Atalar, τον Ivan Walulya καθώς και κάθε άλλο φίλο που έκανα εκεί και θα θυμάμαι για πάντα.

- **...τους φίλους και την οικογένεια μου**

  τον Νίκο, τη Βάλια, την Έλλη, τον Λούζα, τον Παναγιώτη, τον Δημήτρη, τον Γιάννη, την Ελεάννα, την Κατερίνα, τον Άρη, την Κατερίνα, τη Σεμέλη, τον Λάμπρο, τη Χριστίνα, την Ιωάννα και όλους τους ανθρώπους που με άντεξαν (και) τα χρόνια του διδακτορικού και το πέρασαν όλο αυτό μαζί μου. Είμαι πολύ τυχερή που τους έχω στη ζωή μου και είναι για μένα η πιο ισχυρή πηγή χαράς και το κίνητρο μου να συνεχίζω. Τα παιδιά που δουλεύουν στον Ταρτούφο, το συνοικιακό μαγαζί των Ιλισίων, όπου κυριολεκτικά περάσαμε αμέτρητες ώρες με συνάδελφους και μη, φιλοσοφώντας επί παντώς επιστητού. Τον ξάδερφο μου Πέτρο και την Ευαγγελία, που είναι για μένα πηγή έμπνευσης και τεράστο στήριγμα. Το θείο μου που είναι πάντα εκεί για μένα. Τους γονείς μου στους οποίους χρωστάω τα πάντα.

Μέχρι τώρα στη ζωή οι άνθρωποι, με τους οποίους έχω δουλέψει στενά, έχουν αποδειχθεί υπέροχοι. Αυτό ήταν και είναι η μεγαλύτερη μου τύχη.

# Contents

# List of Figures

# List of Tables

CHAPTER 1

# Εκτεταμένη περίληψη στην ελληνική γλώσσα

Τα σύγχρονα υπολογιστικά συστήματα εξαρτώνται από τον μηχανισμό της εικονικής μνήμης – αφαίρεση που έχει αποδειχθεί κρίσιμη για την επιτυχία των υπολογιστών και έχει αντέξει στη δοκιμασία του χρόνου. Επιτρέπει στους προγραμματιστές να γράψουν κώδικα θεωρώντας τη διαθέσιμη *μνήμη πάντα επαρκή, τεράστια, γραμμική και ιδιωτική* για την εφαρμογή τους. Τους επιτρέπει επίσης να *πρόσπελαύνουν τις συσκευές Ε/Ε στο χώρο μνήμης*, χρησιμοποιώντας εντολές γλώσσας μηχανής (assembly) σαν να διαβάζουν και να γράφουν στη μνήμη (π.χ. load/store). Αυτό απλοποιεί περαιτέρω τον προγραμματισμό και δυνητικά επιτρέπει την *εξάλειψη του πυρήνα του λειτουργικού συστήματος* από τη στοίβα λογισμικού Ε/Ε, μια τεχνική που χρησιμοποιείται για καλύτερη απόδοση.

Ωστόσο, αυτή η ευελιξία έχει κόστος. Δεδομένου ότι το λογισμικό χρησιμοποιεί εικονικές διευθύνσεις για την πρόσβαση στα διάφορα επίπεδα της ιεραρχίας αποθήκευσης δεδομένων (κύρια μνήμη, συσκευές PCI-e, κ.λπ.), επικρατούν τρεις πηγές κόστους: (i) η διαχείριση των χώρων εικονικών διευθύνσεων των διεργασιών ενός συστήματος, (ii) η δημιουργία και η διαχείριση των αντιστοιχίσεων της εικονικής μνήμης στο υλικό και (iii) η μετάφραση που απαιτείται για κάθε εντολή load/store του επεξεργαστή. Τα δύο πρώτα περιορίζουν σημαντικά την κλιμακωσιμότητα της εικονικής μνήμης στην εποχή των πολλών πυρήνων [48, 72] ενώ πριν από μια δεκαετία οι υπολογιστές χτύπησαν στο Address Translation Wall [53] και δεν το ξεπέρασαν ποτέ πλήρως.

Οι σύγχρονες τάσεις στη σχεδίαση υπολογιστικών συστημάτων συνεχίζουν να πιέζουν την απόδοση της εικονικής μνήμης, δοκιμάζοντας και τους τρεις προαναφερθέντες μηχανισμούς.

Η μετεωρική αύξηση των απαιτήσεων σε μνήμη, η μαζική στροφή στο υπολογιστικό νέφος (cloud computing), η αυξανόμενη ετερογένεια στις στοίβες υπολογιστών και αποθήκευσης και η κυριαρχία των πολλών πυρήνων και του παράλληλου προγραμματισμού, επιβάλλουν να επανεξετάσουμε τόσο το λογισμικό συστήματος όσο και το υλικό του μηχανισμού της εικονικής μνήμης, μαζί με τη συνέργεια τους.

Στη παρούσα διατριβή εστιάζουμε στο κόστος μετάφρασης διευθύνσεων για εφαρμογές με μεγάλη ένταση σε δεδομένα και στην απόδοση της εικονικής μνήμης για εφαρμογές με έντονη δραστηριότητα Ε/Ε. Σε αυτό το κεφάλαιο εξετάζουμε αρχικά και εν συντομία τους βασικούς μηχανισμούς της εικονικής μνήμης, π.χ. τη σελιδοποίηση, όταν αυτή λειτουργεί σαν αφαίρεση μνήμης άλλα και σα διεπαφή για την πρόσβαση σε αρχεία. Στη συνέχεια, παρουσιάζουμε τα κίνητρα με τα οποία διαλέξαμε να εξετάσουμε συγκεκριμένα σημεία συμφόρησης της απόδοσης και της κλιμακωσιμότητας της εικονικής μνήμης και, τέλος, παρουσιάζουμε τις συνεισφορές της παρούσας διατριβής για την αντιμετώπισή τους.

## 1.1 Σελιδοποιημένη εικονική μνήμη

Η εικονική μνήμη είναι ένας έντονα συ-σχεδιασμένος μηχανισμός (στο υλικό και στο λογισμικό) όπου i) το ΛΣ διαχειρίζεται τους εικονικούς χώρους διευθύνσεων των διεργασιών και τους αντιστοιχίζει σε φυσικά μέσα αποθήκευσης (π.χ. μνήμη), ενώ ii) προσαρμοσμένη αρχιτεκτονική υποστήριξη στο υλικό επιταχύνει το απαραίτητο βήμα μετάφρασης διευθύνσεων κατά την εκτέλεση κάθε εντολής load/store του επεξεργαστή. Εκτός αφαίρεση της φυσικής μνήμης, η εικονική μνήμη είναι επίσης μια πολύτιμη διεπαφή για την αποθήκευση μόνιμων δεδομένων (π.χ. μέσω αντιστοιχίσεων αρχείων).

### 1.1.1 Λειτουργικό Σύστημα

**Διαχείριση του χώρου εικονικών διευθύνσεων.** Κάθε διεργασία έχει έναν μεγάλο γραμμικό χώρο εικονικών διευθύνσεων που σχηματίζει την αφαίρεση μνήμης που εκτίθεται στον προγραμματιστή. Τον διαχειρίζεται το ΛΣ σε μεγέθη σελίδας. Τα διαθέσιμα μεγέθη σελίδας είναι συγκεκριμένα (κβαντισμένα) και εξαρτώνται από την αρχιτεκτονική, για την x86 αρχιτεκτονική μπορεί να είναι 4KB (μικρές σελίδες) ή 2MB/1GB (μεγάλες σελίδες). Το ΛΣ δεσμεύει εύρη εικονικών διευθύνσεων από αυτόν τον χώρο όταν οι διεργασίες ζητούν να δεσμεύσουν φυσική μνήμη ή να αποκτήσουν πρόσβαση σε αρχεία μέσω απεικονίσεων μνήμης.

**Απεικονίσεις (αντιστοιχίσεις) μνήμης.** Το λειτουργικό σύστημα αντιστοιχίζει/απεικονίζει τις δεσμευμένες εικονικές διευθύνσεις κάθε διεργασίας σε ένα μέσο αποθήκευσης, συνήθως στη φυσική μνήμη, για να επιτρέψει την πρόσβαση σε δεδομένα από το χώρο χρήστη μέσω αναφορών (memory references). Η διαχείριση της φυσικής μνήμης γίνεται επίσης σε μεγέθη

σελίδας (4KB) από το ΛΣ και το λειτουργικό σύστημα αντιστοιχίζει εικονικές σελίδες σε φυσικά πλαίσια ανά διεργασία. Με αυτόν τον τρόπο, οι διεργασίες έχουν μια απομονωμένη και προστατευμένη απεικόνιση της μνήμης, αλλά μπορούν ακόμα να μοιράζονται σελίδες μεταξύ τους (π.χ. αντιστοι-χίσεις σε κοινά φυσικά πλαίσια). Το λειτουργικό σύστημα αποθηκεύει τις αντιστοιχίσεις σελίδων εικονικής σε φυσική μνήμη σε μια ειδική δομή δεδομένων ανά διεργασία, συνήθως ένα δέντρο, το οποίο αποτελεί μέρος του περιβάλλοντος εκτέλεσης κάθε διεργασίας και ονομάζεται πίνακας σελίδων. Κάθε καταχώρηση του πίνακα σελίδων (PTE) περιέχει μια μετάφραση από εικονική σε φυσική σελίδα. Οι πίνακες σελίδων εξαρτώνται από την αρχιτεκτονική, όπως και τα μεγέθη σελίδων, καθώς χρησιμοποιούνται απευθείας από το υλικό (HW) της εικονικής μνήμης (θα συζητηθεί αργότερα). Το λειτουργικό σύστημα, στη κοινή περίπτωση, δημιουργεί τις αντιστοι-χίσεις/απεικονίσης νωχελικά ανά σελίδα – μέσω μιας διακοπής ΛΣ (trap) γνωστής ως σφάλμα σελίδας.

**Διαχείριση φυσικής μνήμης.** Αν και δεν συνδέεται αποκλειστικά με την εικονική μνήμη, η διαχείριση της φυσικής μνήμης διασταυρώνεται με πολλές λειτουργίες της. Στην κοινή περίπτωση, τα περισσότερα ΛΣ δεσμεύουν φυσική μνήμη κατ' απαίτηση –δηλαδή κατά τη διάρκεια των σφαλμάτων σελίδας– και σε μεγέθη σελίδας (ή αλλιώς πλαισίου). Τα σφάλματα σελίδας πυροδο-τούνται από κρύες (πρώτες/αρχικές) προσβάσεις σε εικονικές σελίδες (διευθύνσεις) από την εκάστοτε εφαρμογή. Το μέγεθος δέσμευσης (π.χ. μέγεθος σελίδας) επηρεάζει την απόδοση μετάφρασης διευθύνσεων της εικονικής μνήμης (υλικό) και θα συζητηθεί αργότερα.

**Απεικονίσεις αρχείων.** Η εικονική μνήμη μπορεί επίσης να χρησιμοποιηθεί ως διεπαφή για την πρόσβαση σε αρχεία. Η Είσοδος/'Εξοδος μέσω απεικόνισης στη μνήμη (memory-mapped IO) είναι η τεχνική που αντιστοιχίζει εικονικές διευθύνσεις σε σελίδες αρχείων και επιτρέπει την πρόσβαση αρχείων μέσω αναφοράς μνήμης. Το λειτουργικό σύστημα δεσμεύει ένα εύρος εικονικών διευθύνσεων για κάθε απεικόνιση αρχείου και, για μέσα αποθήκευσης που υποστηρίζουν μόνο *πρόσβαση σε επίπεδο μπλόκ, αποθηκεύει τα αρχεία προσωρινά στη φυσική μνήμη* για να τα αντιστοιχίσει έπειτα στο χώρο χρήστη. Μια προσωρινή μνήμη σε επίπεδο ΛΣ, διαχειρίσιμη από το λογισμικό συστήματος και γνωστή ως *page cache*, συσχετίζει τα μπλοκ αποθήκευσης των αρχείων με φυσικά πλαίσια μνήμης και, σε ενορχήστρωση με το σύστημα αρχείων, δημιουργεί αντίγραφα μεταξύ τους. Μόλις αντιγραφούν τα δεδομένα των αρχείων στη φυσική μνήμη, οι δεσμευμένες εικονικές σελίδες αντιστοιχίζονται στα αντίστοιχα πλαίσια φυσικής μνήμης χρησιμοποιώντας τους ίδιους μηχανισμούς σελιδοποίησης που συζητήθηκαν στην προηγούμενη παράγραφο. *Με το memory-mapped IO, τα αρχεία αντιγράφονται από την συσκευή αποθήκευσης στη φυσική μνήμη για να προσπελαστούν.*

Ο χώρος εικονικών διευθύνσεων είναι ένας εγγενώς κοινόχρηστος πόρος μεταξύ των νημάτων μιας διεργασίας και, ως εκ τούτου, καταλήγει να είναι σημείο έντονης συμφόρησης στην εποχή των πολλών πυρήνων. Δυστυχώς, τα περισσότερα λειτουργικά συστήματα σειριοποιούν τις

λειτουργίες του εικονικού χώρου διευθύνσεων [59,60,70,71,140], περιορίζοντας την κλιμακωσιμότητα της εικονικής μνήμης. Το κόστος της σελιδοποίησης (π.χ. καθυστέρηση λόγω σφαλμάτων σελίδας) είναι επίσης σημαντικό. Ιδιαίτερα για εφαρμογές με έντονη δραστηριότητα Ε/Ε και χρήση της τεχνικής απεικόνισης αρχείων στη μνήμη, τα σφάλματα σελίδας μπορούν να μειώσουν σημαντικά τη συνολική απόδοση [124, 125, 168, 169, 220]. Όλα τα παραπάνω κόστη οδήγησαν στην αμφισβήτηση της εφαρμογής της εικονική μνήμης ως διεπαφής αρχείων, ακόμη και για εφαρμογές με μεγάλη ένταση ΙΟ όπως βάσεις δεδομένων [84], παρά την ευκολία χρήσης της.

### 1.1.2 Υποστήριξη στην αρχιτεκτονική για τη μετάφραση διευθύνσεων

Η εικονική μνήμη υποδηλώνει ότι κάθε λειτουργία αναφοράς μνήμης (load/store) απαιτεί ένα βήμα μετάφρασης για να εκτελεστεί. Ο επεξεργαστής μεταφράζει την εικονική διεύθυνση κάθε εντολής load/store σε φυσική διεύθυνση για να στείλει ένα αίτημα προσπέλασης της φυσικής μνήμης που θα ανακτήσει τα αντίστοιχα δεδομένα. Καθώς η μετάφραση βρίσκεται στο κρίσιμο (από πλευράς απόδοσης) τμήμα πρόσβασης στη μνήμη, οι περισσότερες αρχιτεκτονικές χρησιμοποιούν ειδικές κρυφές μνήμες υλικού ανά πυρήνα επεξεργαστή, γνωστές ως Translation Lookaside Buffers (TLBs), για να αποθηκεύσουν τις πιο πρόσφατα χρησιμοποιημένες μεταφράσεις (εγγραφές του πίνακα σελίδων) (PTE) και να επιταχύνουν τη διαδικασία. Εάν η μετάφραση που απαιτείται βρίσκεται στην ιεραρχία του TLB (ευστοχία), η αναζήτηση κοστίζει λιγότερο από 10 κύκλους [117]. Σε περίπτωση που η απαιτούμενη μετάφραση λείπει (αστοχία), οι πίνακες σελίδων της διαδικασίας πρέπει να διασχιστούν για να ανακτηθεί η μετάφραση. Οι περισσότεροι επεξεργαστές σήμερα χειρίζονται τις αστοχίες TLB εξ ολοκλήρου στο υλικό. Σε μια αστοχία TLB, το υλικό ενεργοποιεί μια μηχανή κατάστασης που ονομάζεται περιηγητής πίνακα σελίδων (page table walker) που διασχίζει τον πίνακα σελίδων ανά διεργασία και φορτώνει (αποθηκεύει) το αντίστοιχο PTE στο TLB. Οι διασχίσεις του πίνακα σελίδων είναι ακριβές, καθώς μπορεί να απαιτήσουν πολλαπλές προσβάσεις στην κύρια μνήμη και έτσι να κοστίσουν έως και εκατοντάδες κύκλους. Αυτό μπορεί να προκαλέσει σημαντικές επιβραδύνσεις σε εφαρμογές έντασης δεδομένων [47,56,130,171,172,176,177] και έτσι η χωρητικότητα του TLB –γνωστή ως το εύρος TLB– και το κόστος αστοχίας TLB είναι σημαντικοί παράγοντες απόδοσης στα σύγχρονα συστήματα.

Μια ακόμα σημαντική πηγή κόστους είναι η διατήρηση της συνάφειας της μετάφρασης διευθύνσεων. Η διατήρηση της συνάφειας των κρυφών μνημών TLB με τις αντιστοιχίσεις του εικονικού χώρου διευθύνσεων μιας διεργασίας (με τον πίνακα σελίδων) είναι δαπανηρή. Η διαχείριση του TLB είναι μέρος των λειτουργιών της εικονικής μνήμης και αντιπροσωπευτικό παράδειγμα είναι η ακύρωση αποθηκευμένων αντιστοιχίσεων, π.χ. κατά τη διάρκεια αιτημάτων αποδέσμευσης μνήμης. Οι εγγραφές TLB που αποθηκεύουν τις παλιές καταχωρήσεις του πίνακα σελίδων πρέπει να ακυρωθούν, για ασφάλεια και ορθότητα, και στην αρχιτεκτονική x86 αυτό

γίνεται στο λογισμικό. Το λειτουργικό σύστημα ενεργοποιεί τις ακυρώσεις TLB (shootdowns), που μεταδίδονται μέσω διακοπών μεταξύ των επεξεργαστών. Οι διακοπές αυτές σειριοποιούν όλες τις λειτουργίες της εικονικής μνήμης και μπορεί να κοστίσουν έως και χιλιάδες κύκλους [34, 35, 141]. Η συνάφεια TLB είναι λοιπόν ένα ακόμα σημαντικό εμπόδιο κλιμακωσιμότητας της συ-σχεδιασμένης διεπαφής της εικονική μνήμης σε πολλούς πυρήνες.

## 1.2 Κίνητρο εργασίας

Η υψηλή επίδοση στη μετάφραση διευθύνσεων, η κλιμακώσιμότητα του χώρου εικονικών διευθύνσεων (σε πολλούς πυρήνες) και η σελιδοποίηση με χαμηλό κόστος είναι σημαντικοί παράγοντες για την αποδοτικότητα της εικονικής μνήμης. Σε αυτήν την ενότητα συζητάμε ορισμένες υπολογιστικές τάσεις που στρεσάρουν τη σημερινή σχεδίαση του μηχανισμού της εικονικής μνήμης και, κατά τη γνώμη μας, αμφισβητούν την κλιμακωσιμότητα της διεπαφής τόσο σε πολλούς πυρήνες όσο και σε αυξημένες χωρητικότητες αποθήκευσης.

### 1.2.1 Μετεωρική αύξηση της ζήτησης μνήμης

Η παγκόσμια κυκλοφορία δεδομένων αυξάνεται με σχεδόν εκθετικό ρυθμό την τελευταία δεκαετίες. Η τεχνητή νοημοσύνη/μηχανική μάθηση (AI/ML) είναι το χαρακτηριστικό παράδειγμα εφαρμο-γών με αυξανόμενη ζήτηση σε δεδομένα – 10x ετησίως [49, 81]. Άλλα παραδείγματα είναι η ανάλυση μεγάλων δεδομένων (big data analytics) [43, 82] και ο υπολογισμός κοντά στη μνήμη (in-memory computing) [97, 213]. Ο αντίκτυπος αυτής της έντασης σε δεδομένα γίνεται ιδιαίτερα αισθητός στα κέντρα δεδομένων (data centers).

Ο σύγχρονος μηχανισμός της εικονικής μνήμης πιέζεται να υποστηρίξει επιτυχώς την εκθετική αύξηση των αποτυπωμάτων μνήμης των εφαρμογών (memory footprints) και γίνεται ο ίδιος σημείο συμφόρησης της απόδοσης του συστήματος. Το πρωταρχικό πρόβλημα είναι η περιορισμένη κάλυψη που προσφέρει η κρυφή μνήμη διευθύνσεων TLB; τα υψηλά ποσοστά αστοχίας της οποίας εμφανίστηκαν ως εμπόδιο υπολογιστικής επίδοσης πριν από περίπου μια δεκαετία. Η ιεραρχία TLB αποτυγχάνει να καλύψει τα ενεργά σύνολα εργασίας των εφαρμογών μεγάλων δεδομένων, ειδικά όταν έχουν ακανόνιστα μοτίβα πρόσβασης, π.χ. επεξεργασία γράφων. Το πρόβλημα επισημάνθηκε από τον ακαδημαϊκό κόσμο [47, 130, 176, 177] και αναγνωρίστηκε από τη βιομηχανία με τους προμηθευτές επεξεργαστών να αυξάνουν την εμβέλεια (κάλυψη – reach) των TLBs έκτοτε. Την τελευταία δεκαετία υπάρχει σταθερή αύξηση μέσω μιας διπλής προσέγγισης: i) μεγαλύτερα TLBs (με όρους χωρητικότητας) και ii) καλύτερη υποστήριξη για μεγάλες σελίδες (2MB και 1GB). Ωστόσο η κάλυψη χώρου διευθύνσεων της τάξης των TBs, ίδιου μεγέθους δλδ με τις σύγχρονες χωρητικότητες της μνήμης, είναι δυνατή μόνο όταν χρησιμοποιούνται

100% σελίδες μεγέθους 1 GB στους πιο πρόσφατους επεξεργαστές (π.χ. IceLake Intel). Διαφορετικά, η κάλυψη που προσφέρουν τα TLB είναι της τάξης των GBs. Δυστυχώς, οι σελίδες 1 GB δεν υποστηρίζονται αυτόματα (transparently) από κανένα λειτουργικό σύστημα και μια τέτοια υπο-στήριξη δεν είναι απλή, όπως συζητείται περαιτέρω στην Ενότητα 5.2. Επιπλέον, αυτή η αυξημένη υποστήριξη των TLBs στο υλικό είναι ακριβή όσον αφορά την ισχύ – τα TLB μπορούν να καταναλώσουν 15-20% της ενέργειας του chip [53, 131].

Εκτός από την αύξηση της εμβέλειας του TLB, οι προμηθευτές επεξεργαστών έχουν επενδύσει και σε ένα μάλλον περίπλοκο μηχανισμό στο υλικό για τον περιορισμό της καθυστέρησης στην εκτέλεση σε περίπτωση αστοχίας στο TLB. Οι κρυφές μνήμες MMU αποθηκεύουν τα ανώτερα επίπεδα του πίνακα σελίδων για να μειώσουν τον αριθμό των προσβάσεων στη μνήμη που απαιτούνται κατά τη διάρκεια μιας διάσχισης του πίνακα σε περίπτωση αστοχίας στο TLB. Επιπλέον, οι περιπατητές σελίδων (στο υλικό – page table walkers) έχουν πολλαπλά νήματα για να εξυπηρετούν περισσότερες από μία αστοχίες TLB ταυτόχρονα (π.χ. το Skylake της Intel έχει 2-way walkers).

Ωστόσο, σε αυτή τη διατριβή δείχνουμε ότι εφαρμογές με μεγάλη ένταση δεδομένων και ακανόνιστη πρόσβαση στη μνήμη όταν εκτελούνται εγγενώς σε έναν σύγχρονο επεξεργαστή και χρησιμοποιούν μεγάλες σελίδες που υποστηρίζονται με διαφάνεια (αυτόματα –2MB), εξακολουθούν να υφίστανται έως και 30% επιβαρύνσεις στο χρόνο εκτέλεσης τους λόγω της μετάφρασης διευθύνσεων.

#### 1.2.1.1 Επέκταση μνήμης μέσω του διαύλου/πρωτοκόλλου CXL

Ως απάντηση στην εκθετική αύξηση του όγκου δεδομένων, η βιομηχανία υπολογιστών βρίσκεται στο κατώφλι μιας αλλαγής στην αρχιτεκτονική των υπολογιστικών συστημάτων. Τα κέντρα δεδομένων μετακινούνται από ένα μοντέλο όπου κάθε διακομιστής έχει τη δική του αποκλειστική μνήμη, σε ένα μοντέλο που υπάρχουν κοινόχρηστες συστάδες πόρων μνήμης. Ενώ η ιδεα τέτοιων αποκεντρωμένων διαμοιραζομένων μνημών και των αντίστοιχων καθολικών διεπαφών μεταξύ των μονάδων επεξεργασίας υπάρχει εδώ και πολλά χρόνια, η πρόσφατη μαζική υιοθέτηση του διαύλου Compute Express Link (CXL [85]) –ως συναφούς διαύλου για επεξεργαστές, μνήμες και επιταχυντές– θεωρείται καταλύτης για να γίνει αυτή η ιδέα πραγματικότητα [148,156]. Με τους σύντομα διαθέσιμους επεξεργαστές και συσκευές DDR συμβατούς με το CXL, εισερχόμαστε σε μια νέα εποχή πολλαπλών "επιπέδων" μνήμης, όπου το κάθε επίπεδο θα έχει διαφορετικά χαρακτηριστικά σε χρόνους απόκρισης και σε εύρος ζώνης, αλλά οι πιθανές συγκεντρωτικές χωρητικότητες θα είναι της τάξης των peta-bytes [**?**]. Η πρόσβαση στη μνήμη CXL εξακολουθεί να γίνεται μέσω της αφαίρεσης της εικονικής μνήμης [187], με την αποκεντρωμένη μνήμη πιθανότατα να εκτίθεται στο λειτουργικό σύστημα ως κόμβος NUMA χωρίς CPU, πιέζοντας περαιτέρω τις απαιτήσεις εμβέλειας των κρυφών μνημών μετάφρασης TLB.

#### 1.2.1.2 Σελιδοποίηση 5 επιπέδων

Ένας άλλος περιορισμός που αναδείχθηκε από την αύξηση των χωρητικοτήτων μνήμης, είναι ότι η σελιδοποίηση 4 επιπέδων x86 δεν μπορεί να καλύψει περισσότερα από 64 TB φυσικής μνήμης [11]. Η Intel για να αντιμετωπίσει το πρόβλημα προσέθεσε ένα επιπλέον επίπεδο στο δέντρο του πίνακα σελίδων [22, 77] που επιτρέπει την κάλυψη έως και 4 PiB φυσικού χώρου διευθύνσεων. Η αρχιτεκτονική υποστήριξη για τη σελιδοποίηση 5 επιπέδων έχει εισαχθεί στον πρόσφατο επεξεργαστή Ice Lake της Intel, και το Linux το υποστηρίζει επίσης. Η προσθήκη του επιπλέον επιπέδου στη σελιδοποίηση προσθέτει κόστος στη διάσχιση του πίνακα σελίδων σε περίπτωση αστοχίας στην ιεραρχία TLB, καθώς απαιτείται πρόσθετη πρόσβαση στη μνήμη. Όπως συζητάμε στην επόμενη ενότητα, αυτή η επιβάρυνση μεγεθύνεται στην εικονική εκτέλεση.

Παρά την αυξημένη αρχιτεκτονική υποστήριξη της σελιδοποιημένης εικονικής μνήμης που υπάρχει στους σύγχρονους επεξεργαστές, η εμβέλεια TLB εξακολουθεί να είναι περιορισμένη και αποτυγχάνει να κλιμακώσει με τις συνεχώς αυξανόμενες απαιτήσεις μνήμης. Η αποκεντρωμένη μνήμη μπορεί να οδηγήσει σε συστήματα με χωρητικότητες μνήμης της τάξης των petabyte, στρεσάροντας περαιτέρω της απαιτήσεις εμβέλειας TLB και προσθέτοντας επίπεδα σελιδοποίησης κάτι που καθιστά εγγενώς πιο δαπανηρό τον σημερινό μηχανισμό μετάφρασης διευθύνσεων.

### 1.2.2 Κυριαρχία της εικονικοποίησης

Η εικονικοποίηση είναι μια παλιά ιδέα [105] που αναβίωσε από το Disco [61] στα τέλη της δεκαετίας του ’90 και εξελίχθηκε σήμερα ως η κύρια τεχνική ανάπτυξης υπολογισμών. Η εικονικοποίηση προσθέτει ένα επιπλέον επίπεδο αφαίρεσης μεταξύ των Λειτουργικών Συστημάτων (OS) και του υλικού, γνωστό και ως Virtual Machine Monitor – π.χ. KVM [138]. Η εικονικοποίηση επιτρέπει καλύτερη διαχείριση και απομόνωση των υπολογιστικών πόρων. Ειδικά με τη μορφή Εικονικών Μηχανών (VM), επιτρέπει την κατ’ απαίτηση κατανομή και επομένως καλύτερη συνολική χρήση του υλικού ενός διακομιστή, κυριαρχώντας έτσι στις υποδομές του υπολογι-στικού νέφους.

Η εικονικοποίηση υλικού –CPU, IO, μνήμη – ωστόσο, συνήθως έχει κόστος. Ιδιαίτερα η εικονικοποίηση του πόρου της μνήμης έχει αποδειχθεί πολύ κοστοβόρα με όρους επίδοσης και δύσκολη να επιταχυνθεί [30, 101, 102, 193, 198]. Αυτό που βλέπει η εικονική μηχανή ως φυσική μνήμη είναι μια αφαίρεση και αντιστοιχίζεται ανεξάρτητα στο υλικό (φυσική μνήμη του διακομιστή) όπου βρίσκονται πραγματικά τα δεδομένα. Επομένως, απαιτείται ένα επιπλέον επίπεδο μετάφρασης για κάθε εντολή προσπέλασης μνήμης των εφαρμογών που εκτελούνται μέσα στο VM. Με τεχνική εικονικοποίησης υποβοηθούμενη από το υλικό (nested paging), η εικονική διεύθυνση (gVA) κάθε εντολής φόρτωσης/αποθήκευσης CPU μεταφράζεται πρώτα

στη φυσική διεύθυνση της εικονικής μηχανής (gPA) και στη συνέχεια στη φυσική διεύθυνση του διακομιστή (hPA) όπου αποθηκεύονται τα δεδομένα (Ενότητα 5.2). Κατά τη διάρκεια μιας αστοχίας TLB, η δισδιάστατη μετάφραση εκτελείται στο υλικό με εμφωλευμένη διάσχιση των πινάκων σελίδας της εικονικής μηχανής και του διακομιστή (hypervisor). Αυτή η εμφώλευση καθιστά τις αστοχίες TLB πολύ ακριβές καθώς ο αριθμός των προσβάσεων στη μνήμη πολλαπλασιάζεται με τα επίπεδα σελιδοποίησης, π.χ. απαιτούνται έως και 24 προσβάσεις στη μνήμη κατά την εκτέλεση σε εικονική μηχανή, ενώ μόνο 4 σε φυσική εκτέλεση όταν εξετάζουμε τη σελιδοποίηση 4 επιπέδων.

Στην παρούσα διατριβή διαπιστώνουμε ότι, παρά την όλη αρχιτεκτονική υποστήριξη –π.χ. τις κρυφές μνήμες MMU – η εικονικοποίηση της μνήμης μπορεί να επιφέρει 2 φορές πιο ακριβή διαδικασία μετάφρασης διευθύνσεων σε σύγκριση με την εκτέλεση σε φυσικό μηχάνημα σε έναν σύγχρονο επεξεργαστή.

Η συνεχιζόμενη μετατόπιση των χρηστών στο υπολογιστικό νέφος [81], π.χ. μέσω του ανερχόμενο μοντέλου ανάπτυξης υπολογισμών Serverless [27, 189], καθιστούν την αποτελεσματική εικονικοποίηση του πόρου της μνήμης έναν σχεδιαστικό στόχο υψηλής προτεραιότητας. Η μετάφραση διευθύνσεων σε τέτοια δισδιάστατα περιβάλλοντα επιβαρύνεται με αυξημένο κόστος που μπορεί να μειώσει την απόδοση.

### 1.2.3 Η σύγχρονη ιεραρχία συσκευών αποθήκευσης και η άμεση πρόσβαση (direct access) σε μόνιμα δεδομένα

Οι συσκευές Ε/Ε έχουν εξελιχθεί σημαντικά την τελευταία δεκαετία και μπορούν να προσφέρουν μονοψήφιους χρόνους απόκρισης μs (π.χ. SSD χαμηλής καθυστέρησης [132,218]) ή ακόμη χαμηλότερους (π.χ. μη-πτητική μνήμη [119]), μετατοπίζοντας πλήρως τους κανόνες σχεδιασμού αποτελεσματικής υποστήριξης λογισμικού συστήματος σε κέντρα δεδομένων για εφαρμογές Ε/Ε. Η τρέχουσα στοίβα λειτουργικού συστήματος έχει προσαρμοστεί σε μεγάλο βαθμό για να αντιμετωπίζει καθυστερήσεις της τάξης των ms (π.χ. Ε/Ε μονάδας σκληρού δίσκου) και πολλοί από τους μηχανισμούς που χρησιμοποιεί (π.χ. context-switch) έχουν ξαφνικά σήμερα κόστος συγκρίσιμο με τις νέες συσκευές χαμηλής καθυστέρησης [46]. Αυτή η αναντιστοιχία έχει εντοπιστεί πολλές φορές στη βιβλιογραφία (π.χ. [46, 175]) και πολυάριθμες εργασίες δείχνουν πώς η επίδοση δεν επηρεάζεται πλέον μόνο ή κυρίαρχα από την απόδοση υλικού, καθώς οι διακομιστές συχνά ξοδεύουν μεγάλο μέρος του χρόνου τους στην εκτέλεση κώδικα λειτουργικού συστήματος [64, 146]. Το λειτουργικό σύστημα έγινε ξαφνικά το σημείο συμφόρησης και το λογισμικό του συστήματος πρέπει να επανασχεδιαστεί ώστε να λειτουργεί στη κλίμακα του δευτερολέπτου ή νανοδευτερόλεπτου για να υποστηρίξει με επιτυχία τις λειτουργίες και την απόδοση μπορεί να προσφέρει το υποκείμενο υλικό [236]. Αυτή η διατριβή εστιάζει στην

εικονική μνήμη ως τη διεπαφή του λειτουργικού συστήματος με τις μη-πτητικές μνήμες ως συσκευές αποθήκευσης, ένα μάλλον μοναδικό μη πτητικό επίπεδο που συνδυάζει εξαιρετικά χαμηλό χρόνο απόκρισης με διευθυνσιοδότηση σε επίπεδο byte.

Η μη-πτητική μνήμη είναι μια νέα τεχνολογία μόνιμης αποθήκευσης [119,186] που συνδέεται με τους επεξεργαστές μέσω του διαύλου μνήμης ή του συνδέσμου CXL [85], όμοια με τη DRAM, και επομένως είναι άμεσα προσβάσιμη μέσω των εντολών load/store του επεξεργαστή. Η τεχνολογία συνδυάζει μοναδικά τη μη πτητικότητα και τη δυνατότητα διευθυνσιοδότησης σε επίπεδο byte με χρόνους απόκρισης και εύρος ζώνης πιο κοντά σε αυτό της DRAM παρά σε άλλες συσκευές μόνιμης αποθήκευσης. Αυτό αμβλύνει τη διάκριση πολλών δεκαετιών μεταξύ αργής αλλά μόνιμης αποθήκευσης και γρήγορης αλλά πτητικής μνήμης. Η υψηλή απόδοση E/E – στην κλίμακα των νανοδευτερόλεπτων- κάνει τις προσβάσεις στη συσκευή αποθήκευσης πολύ φθηνό-τερες από τις κλήσεις συστήματος του ΛΣ και η μείωση του κόστους εκτέλεσης του κώδικα του ΛΣ αναδύεται σε ισχυρή σχεδιαστική απαίτηση [15, 67, 68, 74, 90, 125, 142, 149, 162, 211, 216, 219, 220, 221, 222, 230]. Η εικονική μνήμη έχει πρωταγωνιστικό ρόλο σε αυτό το πεδίο, καθώς επιτρέπει την *άμεση πρόσβαση* στον χώρο αποθήκευσης. Η απεικόνιση στη μνήμη των αρχείων PMem (μέσω κλήσεων συστήματος mmap) μπορεί να αντιστοιχίσει εικονικές σελίδες από το χώρο διευθύνσεων των διεργασιών απευθείας σε φυσικές διευθύνσεις της μη-πτητικής μνήμης, *παρακάμπτοντας εξ ολοκλήρου οποιαδήποτε προσωρινή αποθήκευση στη DRAM (π.χ. page cache)* (Section 1.1.1) και ουσιαστικά απεικονίζοντας το χώρο αποθήκευσης μόνιμων δεδομένων απευθείας στο χώρο χρήστη. Αυτό δημιουργεί το *πρώτο μονοπάτι πρόσβασης σε συσκευή αποθήκευσης με μηδενικές αντιγραφές (χωρίς προσωρινά αντίγραφα των μόνιμων δεδομένων στη DRAM)* καθώς οι εφαρμογές έχουν άμεση πρόσβαση σε μόνιμα δεδομένα μέσω παραπομπών στη μνήμη (εντολές load/store του επεξεργαστή).

Συνεπώς, η άμεση πρόσβαση στην μη-πτητική μνήμη (persistent memory -PMem) μέσω της mmap() είναι μια «τέλεια διεπαφή» για την PMem, καθώς παρέχει στις εφαρμογές τη συντομότερη δυνατή πρόσβαση στα αποθηκευμένα μόνιμα δεδομένα [222]. Ωστόσο, το κόστος (σε επίδοση) των λειτουργιών/της διεπαφής της εικονικής μνήμης μπορεί συχνά να μειώσει τη συνολική απόδοση [124, 125, 149, 220] μιας εφαρμογής. Στα πλαίσια της παρούσας διατριβής μετρήσαμε τη μέση καθυστέρηση ανάγνωσης αρχείων 256 KB χρησιμοποιώντας είτε την κλήση συστήματος mmap() είτε τη κλήση συστήματος read(). Η read αντιγράφει εγγενώς τα δεδομένα αρχείων από τη μόνιμη μνήμη σε μια ιδιωτική προσωρινή μνήμη στη DRAM για τη πρόσβαση σε αυτά, ενώ η mmap παρέχει την άμεση πρόσβαση στα δεδομένα (όπως αναφέρθηκε πάρα πάνω). Παρατηρούμε ότι η άμεση πρόσβαση αποδίδει σημαντικά χειρότερα από τη read παρά το πλεονέκτημα της μηδενικής αντιγραφής. Το ∼30% του συνολικού χρόνου (ένα νήμα) ξοδεύεται στην εκτέλεση του κώδικα του λειτουργικού συστήματος που i) διαχειρίζεται την αντιστοίχιση των εικονικών διευθύνσεων της διεργασίας (mmap), ii) συμπληρώνει τους πίνακες σελίδων της (σφάλματα σελίδας) και iii) τους καταστρέφει ενώ ακυρώνει τα TLB ( munmap). Αυτό το κόστος

εκτέλεσης πυρήνα αυξάνεται σε ∼60% όταν πολλαπλά νήματα επεξεργάζονται πολλά αρχεία ταυτόχρονα, λόγω συμφόρησης στα κλειδώματα της εικονικής μνήμης [70,71,72,140,141]. Αυτά τα ευρήματα επιβεβαιώνουν προηγούμενες μελέτες που υπογραμμίζουν τη σελιδοποίηση ως σημαντικό παράγοντα απόδοσης για την άμεση πρόσβαση στη PMem (π.χ. [124, 149, 220]).

Οι επιπτώσεις της εικονικής μνήμης στην επίδοση των εφαρμογών όταν χρησιμοποιείται ως διεπαφή αρχείων στο Linux είναι γνωστές από το 2000 [205] και παραμένουν εντυπωσιακά ίδιες για πάνω από είκοσι χρόνια. Ωστόσο, με τις συσκευές αποθήκευσης που επιτρέπουν μόνο διευθυνσιοδότηση σε επίπεδο μπλοκ (π.χ. SSD) τα δεδομένα αρχείων πρέπει να αντιγράφονται υποχρεωτικά από τη συσκευή αποθήκευσης στην πτητική μνήμη πριν απεικονιστούν (page cache, Section 1.1.1). Είναι η ανάδειξη των συσκευών με χαμηλούς χρόνους απόκρισης και *η δυνατότητα άμεσης πρόσβασης (χωρίς πτητικά αντίγραφα) σε μόνιμα δεδομένα* που πιστεύουμε ότι μας προτρέπουν να ξανασκεφτούμε σήμερα τη σχεδίαση της διεπαφής αυτής.

Οι σημερινές συσκευές Ε/Ε μπορούν να προσφέρουν μικρούς χρόνους απόκρισης (στην τάξη του δευτερολέπτου και λιγότερο) εκθέτοντας το λογισμικό συστήματος ως το σημείο συμφόρησης. Η απόδοση της εικονικής μνήμης, η οποία λειτουργεί ως διεπαφή προς τα αρχεία, είναι κρίσιμη για την *άμεση πρόσβαση* σε μόνιμα δεδομένα, π.χ. σε συσκευές μη πτητικής μνήμης. Ο σημερινός σχεδιασμός εισάγει υψηλές καθυστερήσεις και κλιμακώνει ελάχιστα σε πολλούς πυρήνες, αποτυγχάνοντας να αποδώσει αυτό που μπορεί να προσφέρει το υποκείμενο υλικό.

### 1.2.4 Μια κοινή διεπαφή σε έναν ταχέως αναπτυσσόμενο και ετερογενή υπολογιστικό χάρτη

Η εικονική μνήμη σχεδιάστηκε σε μια περίοδο που τα συστήματα μνήμης ήταν σε μεγάλο βαθμό ομοιογενή, με ένα μόνο επίπεδο τοπικής μνήμης και μια τοπική συσκευή αποθήκευσης με διευθυνσιοδότηση σε επίπεδο μπλοκ. Σήμερα, ολόκληρη η στοίβα αποθήκευσης έρχεται ως ιεραρχία και κάθε επίπεδο έχει ποικίλα χαρακτηριστικά όσον αφορά τον χρόνο απόκρισης, το εύρος ζώνης και την πρόσβαση (π.χ. σε προσωρινή μνήμη ή άμεση). Επίσης, πολλαπλές μονάδες επεξεργασίας και οι τοπικές τους μνήμες συνδέονται με συνάφεια. Η εικονική μνήμη πρέπει να συμβαδίζει με αυτήν την ταχέως αναπτυσσόμενη αρχιτεκτονική συστήματος ως κοινή διεπαφή για σχεδόν όλα τα στοιχεία. Αυτό αναδεικνύει περαιτέρω τις απαιτήσεις για αποδοτική μετάφραση διευ-θύνσεων, που συζητήθηκε στην Ενότητα 1.2.1, καθώς η μετάφραση πρέπει να έχει εμβέλεια πολύ μεγαλύτερη από το μέγεθος της τοπικής μνήμης. Επίσης, κλονίζει τις βασικές υποθέσεις της παλαιάς σχεδίασης της διεπαφής της εικονικής μνήμης. Η σημασιολογία της διεπαφής της εικονικής μνήμης -π.χ. POSIX– παραμένει σε μεγάλο βαθμό ανέπαφη με τα χρόνια και ομοιογενής για όλους τους τύπους αντιστοιχίσεων/απεικονίσεων μνήμης, ανεξάρτητα από

το μέσο αποθήκευσης. Πιστεύουμε ότι σε αυτή την αδράνεια λανθάνουν/ελλοχεύουν σημαντικές δυνατότητες βελτίωσης απόδοσης.

Ο αντίκτυπος της σημασιολογίας των διεπαφών στην κλιμακωσιμότητα και την απόδοση του λογισμικού συστήματος είναι ένα επίσημα μελετημένο θέμα από τους Clements et.al. [72]. Όσον αφορά στην εικονική μνήμη, διάφορες ερευνητικές εργασίες προτείνουν ριζικές αλλαγές στη διαχείριση του χώρου διευθύνσεων [59, 72, 91] ή των TLBs [72, 141] για τη βελτίωση της απόδοσης και συχνά χαλαρώνουν τις απαιτήσεις της σημασιολογίας POSIX στη διαδικασία. Σε αυτή τη διατριβή -και σε συνέχεια της προηγούμενης υποενότητας– εστιάζουμε στην εικονική μνήμη ως διεπαφή *άμεσης πρόσβασης* σε μόνιμα δεδομένα αποθηκευμένα σε συσκευές με δυνατότητα διευθυνσιοδότησης σε επίπεδο byte (π.χ. μη πτητικές μνήμες). Δείχνουμε (όπως και παλαιότερες εργασίες) ότι η κλιμακωσιμότητα της άμεσης πρόσβασης σε πολλούς πυρήνες υποφέρει από τη ταυτόχρονη απελευθέρωση πόρων κατά την αίτηση του χρήστη (π.χ. καταστροφή αντιστοιχί-σεων/απεικονίσεων κατά τη διάρκεια των κλήσεων συτήματος munmap) και από τη δημιουργία αντιστοιχί-σεων/απεικονίσεων σε πολλαπλάσια του μεγέθους σελίδας (σφάλματα σελίδας). Η πρώτη προέρχεται από μια ισχυρή και γενική απαίτηση του πρότυπου POSIX ενώ η δεύτερη αφορά στην υλοποίηση του ΛΣ Linux. Αναρωτιόμαστε αν και πώς αυτές οι προδιαγραφές/απαιτή-σεις εξακολουθούν να είναι σχετικές όταν οι εικονικές διευθύνσεις αντιστοιχούνται απευθείας στον χώρο αποθήκευσης, έναν πολύ πιο πλεονάζων/άφθονο πόρο σε σχέση με τη φυσική μνήμη ο οποίος ανακτάται κιόλας αργά. Ένα άλλο παράδειγμα διερώτησης είναι, χρειαζόμαστε ακόμα έναν κοινό χώρο διευθύνσεων για αντιστοιχίσεις/απεικονίσεις σωρού και μόνιμων δεδομένων όταν χρησιμοποιούμε διεπαφή άμεσης πρόσβασης για τα δεύτερα; Η επανεξέταση, η επέκταση ή η αλλαγή διεπαφών για την αποτελεσματική πρόσβαση σε μη-πτητικές μνήμες είναι μια καθιερωμένη στρατηγική, αλλά η υπάρχουσα βιβλιογραφία εξετάζει κυρίως τα APIs και τη σχεδίαση των συστημάτων αρχείων (π.χ. [211], [220]) και όχι της εικονικής μνήμης.

Οι ιεραρχίες μνήμης πολλών ταχυτήτων και οι συσκευές αποθήκευσης υψηλής επίδοσης μεταβάλλουν γρήγορα τη χρήση της εικονικής μνήμης ως κοινής διεπαφής σε πολλά μέσα με διαφορετικά χαρακτηριστικά (π.χ. πτητικότητα, απόδοση κ.λπ.). Αυτό επιβάλλει νέες προκλήσεις στο σχεδιασμό των μηχανισμών και των κανόνων της διεπαφής της εικονικής μνήμης αμφισβητώντας την τωρινή τους μορφή.

## 1.3 Ορισμός του ερευνητικού προβλήματος

Αυτή η διατριβή εστιάζει στη βελτίωση της απόδοσης της εικονικής μνήμης όσον αφορά i) τη μετάφραση διευθύνσεων και ii) την άμεση πρόσβαση σε συσκευές αποθήκευσης με δυνατότητα διευθυνσιοδότησης σε επίπεδο byte (π.χ. μη-πτητική μνήμη). Για να διαμορφώσουμε και να

αξιολογήσουμε τις ερευνητικές μας προτάσεις σε περιβάλλοντα αντιπροσωπευτικά των πραγματικών συστημάτων, υλοποιούμε το μεγαλύτερο μέρος της εργασίας μας χρησιμοποιώντας πραγματικές συσκευές και το ΛΣ Linux. Για τις επεκτάσεις υλικού που προτείνουμε, χρησιμοποιούμε μοντέλα απόδοσης για να προβλέψουμε τον αντίκτυπο τους.

**Υψηλό κόστος μετάφρασης διευθύνσεων.** Η απόδοση/επίδοση του μηχανισμού μετάφρασης εικονικών διευθύνσεων αποτυγχάνει να κλιμακώσει με τη μετεωρική αύξηση των απαιτήσεων σε μνήμη, κυρίως λόγω της περιορισμένης εμβέλειας της ιεραρχίας TLB. Τόσο η βιβλιογραφία όσο και η παρούσα διατριβή δείχνουν ότι η διάσχιση των πινάκων σελίδων των διεργασιών (κατά την αστοχία στο TLB) μπορεί ακόμα να αποτελέσει κυρίαρχο κόστος εκτέλεσης. Αυτό μεγεθύνεται σε εικονικά περιβάλλοντα λόγω της εμφωλευμένης σελιδοποίησης και παρά την αυξημένη υποστήριξη σε υλικό μετάφρασης στους πρόσφατους επεξεργαστές (π.χ. [47, 101, 130, 171, 172, 194, 199]).

Ο πρώτος στόχος αυτής της διατριβής είναι να ελαχιστοποιήσει το κόστος μετάφρασης για την εκτέλεση τόσο σε φυσικά όσο και σε εικονικά μηχανήματα, δουλεύοντας στο όριο/τομή των επιπέδων υλικού και λογισμικού (OS) της εικονικής μνήμης. Στοχεύουμε σε έναν (επανα)-σχεδιασμό που: i) προσφέρει σχεδόν μηδενικό κόστος μετάφρασης διευθύνσεων, ii) κλιμακώνει με τις συνεχώς αυξανόμενες χωρητικότητες μνήμης, iii) διατηρεί την ευέλικτη διαχείριση φυσικής μνήμης και iv) είναι διαφανής στις εφαρμογές.

**Περιοριστική διεπαφή για άμεση πρόσβαση σε μόνιμα δεδομένα.** Η μη-πτητική μνήμη (Persistent Memory - PMem) είναι μια μοναδική τεχνολογία αποθή-κευσης που συνδυάζει εξαιρετικά χαμηλούς χρόνους απόκρισης, διευθυνσιοδότηση σε επίπεδο byte και διασύνδεση με το δίαυλο μνήμης-επεξεργαστή. Η απεικόνιση αρχείων PMem μέσω της εικονικής μνήμης μπορεί να παρέχει *άμεση πρόσβαση* σε μόνιμα δεδομένα μέσω εντολών load/store του επεξεργαστή, σχηματίζοντας το συντομότερο διαθέσιμο μονοπάτι προς τη συσκευή αποθήκευσης. Ωστόσο, τόσο προηγούμενες εργασίες όσο και αυτή η διατριβή, δείχνουν ότι οι λειτουργίες/ μηχανισμοί της εικονικής μνήμης εισάγουν υψηλές καθυστερήσεις και δεν κλιμακώνουν επιτυχώς σε πολλούς πυρήνες, περιορίζοντας την απόδοση της άμεσης πρόσβασης (π.χ. [124, 125, 149, 220]).

Ο δεύτερος στόχος αυτής της διπλωματικής εργασίας είναι η μελέτη της εικονικής μνήμης ως διεπαφής αρχείων και ο εντοπισμός όλων των πηγών επιβάρυνσης/κόστους επίδοσης στον τρέχοντα σχεδιασμό. Με βάση τα ευρήματα, στοχεύουμε να επανασχεδιάσουμε τη διεπαφή της εικονικής μνήμης – μαζί με τη σημασιολογία της – για άμεση πρόσβαση σε μόνιμα δεδομένα, στοχεύοντας να πλησιάσουμε αυτό που μπορεί να προσφέρει το υποκείμενο υλικό όσον αφορά την επίδοση.

## 1.4 Ερευνητικές προτάσεις

Η παρούσα διατριβή κάνει τις ακόλουθες συνεισφορές:

- Για να μειώσουμε το κόστος μετάφρασης διευθύνσεων, προτείνουμε τη *σελιδοποίηση με επίγνωση γειτνίασης (Contiguity-Aware paging)* και τη *κερδοσκοπική μετάφραση διευθύνσεων με βάση τη μετατόπιση (Speculative Offset Address Translation – SpOT)*, μια συ-σχεδιασμένη υλοποίηση της εικονικής μνήμης που κρύβει το κόστος μετάφρασης με υποθετική εκτέλεση στον επεξεργαστή. Η τεχνική είναι εφαρμόσιμη τόσο σε φυσικά όσο και σε εικονικά μηχανήματα.

- Για να βελτιώσουμε την επίδοση και την κλιμακωσιμότητα της *άμεσης πρόσβασης* σε μόνιμα δεδομένα, προτείνουμε μια νέα διεπαφή που ονομάζουμε *DaxVM*. Το DaxVM απεικονίζει αρχεία αποθηκευμένα σε μη-πτητικές μνήμες χαλαρώνοντας τις απαιτήσεις του προτύπου POSIX και επανασχεδιάζοντας μηχανισμούς της εικονικής μνήμης για να μειώσει σημαντικά το κόστος εκτέλεσης του λογισμικού συστήματος.

Στη συνέχεια συζητάμε τα πιο σημαντικά σημεία κάθε συνεισφοράς.

### 1.4.1 Αποτελεσματική εικονικοποίηση της μνήμης μέσω συνεχόμενων αντιστοιχίσεων/απεικονίσεων

Πολλές ερευνητικές εργασίες [47, 101, 130, 172, 177, 227] εκμεταλλεύονται τη γειτνίαση στις απεικονίσεις μνήμης, δηλαδή σελίδες που αντιστοιχίζονται συνεχόμενα στον εικονικό και το φυσικό χώρο διευθύνσεων ανά διεργασία, για να επιταχύνουν τη μετάφραση διευθύνσεων. Ωστόσο συνήθως δεν υποστηρίζουν την ευέλικτη διαχείριση της φυσικής μνήμης; αντιθέτως για να δημιουργήσουν την επιθυμητή γειτνίαση παραβιάζουν θεμελιώδεις αρχές του λειτουργικού συστή-ματος. Επίσης, οι περισσότερες προτεινόμενες επεκτάσεις υλικού αφορούν σε φυσικά μηχανήματα και δεν υποστηρίζουν εικονικοποίηση. Για την επίλυση αυτών των προβλημάτων σε αυτή τη διατριβή προτείνουμε i) τη *σελιδοποίηση με επίγνωση γειτνίασης (Contiguity-Aware Paging – CA)* στο επίπεδο του λογισμικού και ii) την *κερδοσκοπική μετάφραση διευθύνσεων με βάση τη μετατόπιση (Speculative Offset Address Translation – SpOT)* στο επίπεδο του υλικού.

Η σελιδοποίηση CA επεκτείνει τη σελιδοποίηση κατ'απαίτηση, την πάγια τεχνική διαχείρισης της φυσικής μνήμης των σύγχρονων Λειτουργικών Συστημάτων. Δημιουργεί συνεχόμενες αντιστοιχίσεις νωχελικά και χωρίς ισχυρές εγγυήσεις, δεσμεύοντας στοχευμένες φυσικές σελίδες κατά τα σφάλματα σελίδας. Έτσι διατηρεί όλες τις βασικές τεχνικές διαχείρισης της φυσικής μνήμης ενός ΛΣ, όπως η σελιδοποίηση, η δέσμευση κατ'απαίτηση, η αντιγραφή κατά την εγγραφή κ.α. Η τεχνική είναι εγγενώς συμβατή με την εμφωλευμένη σελιδοποίηση (εικονικοποίηση) και εφαρμόζεται ανεξάρτητα από το ΛΣ της εικονικής μηχανής και το ΛΣ του φυσικού μηχανήματος (διακομιστή) για τη δημιουργία συνεχόμενων αντιστοιχίσεων σε κάθε διάσταση. Διατηρεί ελάχιστα μεταδεδομένα ανά αντιστοίχιση για τον εντοπισμό στοχευμένων φυσικών σελίδων για δέσμευση κατά τη διάρκεια των σφαλμάτων σελίδας. Διατηρεί επίσης ένα χάρτη των

διαθέσιμων ελεύθερων μπλοκ μνήμης του συστήματος και εφαρμόζει πολιτικές τοποθέτησης για την αντι-μετώπιση του εξωτερικού κατακερματισμού, αλλά χωρίς δέσμευση ή εκ των προτέρων εκχώρηση φυσικής μνήμης. Υποστηρίζει απεικονίσεις σωρού και αρχείων, εφαρμογές πολλαπλών νημάτων και ταυτόχρονη εκτέλεση πολλών εφαρμογών. Έχουμε υλοποιήσει την τεχνική CA paging στο Linux και την έχουμε διαθέσει δημόσια[1].

Το SpOT είναι μια μικρο-αρχιτεκτονική επέκταση που προβλέπει τη μετάφραση (φυσική διεύθυνση) σε περίπτωση αστοχίας στο TLB. Τροφοδοτεί την πρόβλεψη στον επεξεργαστή για να συνεχίσει με υποθετική εκτέλεση εντολών ενώ η διάσχιση των πινάκων σελίδων για την εύρεση της ντετερμινιστικής μετάφρασης και την επαλήθευση της πρόβλεψης συνεχίζει να συμβαίνει στο παρασκήνιο. Η βασική παρατήρηση είναι ότι μπορούμε να εκμεταλλευτούμε την υποκείμενη γειτνίαση στις απεικονίσεις, που δημιουργείται από το CA paging, για να προβλέψουμε τις μεταφράσεις. Δείχνουμε ότι η ντετερμινιστική καταγραφή των ορίων συνεχόμενων αντιστοιχίσεων αυθαίρετου μεγέθους για την αποθήκευση τους σε κρυφές μνήμες (τύπου TLB) είναι αρχιτεκτονικά περίπλοκη σε εικονικοποιημένα περιβάλλοντα, επεκτείνοντας την πρόταση για φυσικά μηχανήματα RMM [130]). Θεωρούμε ότι αυτή η πολυπλοκότητα είναι ο λόγος για τον οποίο οι περισσότερες προτάσεις στη βιβλιογραφία δεν υποστηρίζουν την εικονικοποίηση. Αντίθετα, ο προτεινόμενος σχεδιασμός SpOT της παρούσας διατριβής επιτυγχάνει υψηλή απόδοση και χαμηλή αρχιτεκτονική πολυπλοκότητα σταματώντας ωστόσο να παρέχει ισχυρές εγγυήσεις ασφαλούς εκτέλεσης.

Η σελιδοποίηση CA είναι μια SW τεχνική που μπορεί να υποστηρίξει οποιαδήποτε μέθοδο HW που εκμεταλλεύεται τη γραμμικότητα στις απεικονίσεις για να επιταχύνει τις μεταφράσεις. Σε συνδυασμό με το SpOT σχηματίζουν μια βελτιστοποίηση άμεσα εφαρμόσιμη σε οποιαδήποτε τεχνική HW που στηρίζεται στην προσωρινή αποθήκευση πρόσφατων μεταφράσεων σε κρυφές μνήμες (π.χ. TLB). Αξιολογούμε τον συνδυασμό τους με τη κλασική σελιδοποίηση (TLB) και έχοντας ενεργοποιημένη τη διαφανή υποστήριξη των μεγάλων σελίδων στο ΛΣ. Το συ-σχεδιασμένο προτεινόμενο σχήμα μειώνει το γενικό κόστος μετάφρασης σε <1% κατά μέσο όρο για εφαρμογές με ένταση δεδομένων που εκτελούνται μέσα σε εικονικές μηχανές.

### 1.4.2 Στρεσάροντας τα όρια της μνήμης ως διεπαφής για την πρόσβαση σε αρχεία

Σε αυτή τη διατριβή μελετάμε τη διεπαφή της εικονικής μνήμης για άμεση πρόσβαση (Direct Access – DAX) [4] σε μόνιμα δεδομένα. Εντοπίζουμε όλες τις πηγές κόστους της εικονικής μνήμης που επιβαρύνουν την εκτέλεση εφαρμογών με ένταση Ε/Ε και προτείνουμε το ***DaxVM***, μια νέα διεπαφή για την πρόσβαση σε αρχεία αποθηκευμένα σε μη πτητικές μνήμες. Το DaxVM φέρνει τη συνολική απόδοση κοντά σε αυτό που μπορεί να προσφέρει η συσκευή αποθήκευσης.

[1] https://github.com/cslab-ntua/contiguity-isca2020

Η άμεση πρόσβαση σε συσκευή αποθήκευσης με δυνατότητα διευθυνσιοδότησης σε επίπεδο byte παραλείπει την ανάγκη για οποιαδήποτε μορφή αποθήκευσης προσωρινών αντιγράφων των αρχείων στη DRAM (π.χ. page cache). Ωστόσο, εντοπίζουμε πολλά σημεία στη σχεδίαση της εικονικής μνήμης που υποθέτουν ότι τα δεδομένα είναι πάντα προσωρινά αποθηκευμένα στη DRAM και δείχνουμε πώς αυτό επηρεάζει την απόδοση κατά την άμεση πρόσβαση (DAX) σε μόνιμα δεδομένα. Για παράδειγμα, οι αντιστοιχίσεις/απεικονίσεις δημιουργούνται πάντα κατ'απαίτηση (νωχελικά) για την εξοικονόμηση του σπάνιου/πολύτιμου πόρου της φυσικής μνήμης και τα TLB ακυρώνονται συγχρονισμένα όταν οι απεικονίσεις καταστρέφονται (π.χ. munmap) για την άμεση αποδέσμευση της φυσικής μνήμης. Το DaxVM διατηρεί ***μόνιμους πίνακες σελίδων*** ανά αρχείο και τους (απο)επισυνάπτει στους χώρους διευθύνσεων των διεργασιών κατά τις λειτουργίες m(un)map για την εξάλειψη του κόστους της σελιδοποίησης και την υποστή-ριξη ***O(1 ) mmap*** [201] κλήσεων συστήματος. Το DaxVM καταγράφει επίσης τις αιτήσεις για τη διάλυση απεικονίσεων (e.g. munmap) και ***ακυρώνει νωχελικά και όχι σύγχρονα τα TLBs***, ενισχύοντας σημαντικά την κλιμακωσιμότητα της εικονικής μνήμης σε πολλούς πυρήνες. Μια άλλη βασική παρατήρηση είναι ότι οι απεικονίσεις αρχείων αποθηκευμένων στη PMem ενδέχεται να "ζήσουν" για λίγο σε σύγκριση με τις απεικονίσεις σωρού ή απεικονίσεις αρχείων που χρησιμοποιούν προσωρινά αντίγραφα στη φυσική μνήμη. Ενα ενδεικτικό παράδειγμα είναι οι εφαρμογές πολλαπλών νημάτων με ένταση σε λειτουργίες E/E σε μικρά αρχεία –π.χ. διακομιστές ιστού, διακομιστές αλληλογραφίας ή διακομιστές αρχείων– όπου η πρόσβαση στα αρχεία είναι μοναδική (το περιεχόμενο διαβάζεται/γράφεται μια φορά) και με την άμεση πρόσβαση (DAX) αποφεύγονται εντελώς τα προσωρινά αντίγραφα των μόνιμων δεδομένων στη DRAM. Το DaxVM παρέχει έναν ξεχωριστό (απο)κατανεμητή εικονικών διευθύνσεων για τέτοιες ***εφήμερες απεικονίσεις*** που κλιμακώνει καλύτερα σε πολλούς πυρήνες. Επίσης, οι απεικονίσεις αρχείων άμεσης πρόσβασης επιτρέπουν τον έλεγχο της ανθεκτικότητας των μόνιμων δεδομένων από τον χώρο χρήστη και αυτό θεωρείται καλή πρακτική προγραμματισμού [220]. Για τέτοιες περιπτώσεις, το DaxVM *καταργεί εξ ολοκλήρου την ανίχνευση/καταγραφή των σελίδων αρχείων που έχουν υποστεί αλλαγές από το χώρο του πυρήνα.* Τέλος, σε αυτή τη διατριβή δείχνουμε ότι ο σύγχρονος μηδενισμός των μπλοκ που δεσμεύονται για ένα αρχείο είναι μια επιβάρυνση που εισάγεται από την άμεση πρόσβαση (DAX) για λόγους ασφαλείας και μπορεί να επηρεάσει την απόδοση των λειτουργιών επέκτασης αρχείων (append). Αντ' αυτού, προτείνουμε ασύγχρονο μηδενισμό των μπλοκ αποθήκευσης από τα συστήματα αρχείων για PMem, ακολουθώντας αντίστοιχες προτάσεις για πτητική μνήμη [179]. Ο σχεδιασμός του DaxVM εκτίθεται ως μια νέα διεπαφή, με σημασιολογία που επιτρέπει στους προγραμματιστές να ενεργοποιούν/απενεργοποιούν τις περισσότερες βελτιστοποιήσεις του με βάση τις απαιτήσεις απόδοσης και ασφάλειας της εφαρμογής.

Το DaxVM επανεξετάζει και χαλαρώνει τις αυστηρές απαιτήσεις POSIX για αντιστοιχίσεις/απεικονίσεις DAX, ακολουθώντας γνωστούς κανόνες [72] για την κλιμακωσιμότητα των διεπαφών. Υλοποιήσαμε το DaxVM σε Linux και δύο συστήματα αρχείων τελευταίας τεχνολογίας και βελτιστοποιημένα για PMem συσκευές – ext4-DAX [216] και NOVA [221] – και το διαθέτουμε σε μορφή ανοιχτού κώδικα [2]. Για εφαρμογές πολλαπλών νημάτων που επεξεργάζονται πολλά μικρά αρχεία για μικρά διαστήματα, π.χ. Apache, το DaxVM βελτιώνει την απόδοση της βασικής υλοποίησης της mmap στο Linux έως και 4,9x. Αντιστρέφει επίσης την τάση που ευνοεί τη χρήση της read για τέτοιες εφαρμογές, ξεπερνώντας την απόδοση της έως και 1,5x. Το DaxVM αυξάνει επίσης τη διαθεσιμότητα του συστήματος, παρέχοντας γρήγορους χρόνους εκκίνησης για βάσεις δεδομένων βελτιστοποιημένες για συσκευές PMem και διατηρεί την υψηλή απόδοση τους ακόμη και όταν εκτελούνται σε κατακερματισμένα συστήματα αρχείων.

Το DaxVM, παρόλο που έχει σχεδιαστεί για συσκευές Intel Optane, σχετίζεται με πολλαπλές τεχνολογίες γρήγορης αποθήκευσης. Εφαρμόζεται άμεσα σε οποιαδήποτε συσκευή με δυνατότητα διευθυνσιοδότησης σε επίπεδο byte, μια διάταξη που υποστηρίζεται από το αναδυόμενο Compute Express Link [85] – π.χ., η νέα συσκευή SSD της Samsung με σημασιολογία μνήμης [186]. Επιπλέον, οι τεχνολογίες μνήμης flash έχουν μειώσει τον χρόνο απόκρισης από το χώρο αποθήκευσης σε δεκάδες μικροδευτερόλεπτα [175] και η γρήγορη εύρεση των μπλοκ όπου είναι αποθηκευμένα τα δεδομένα των αρχείων [162] (από το σύστημα αρχείων) έχει αναδειχθεί σε κρίσιμο παρά-γοντας απόδοσης. Οι μόνιμοι πίνακες σελίδων αρχείων του DaxVM μπορούν να αξιοποιηθούν για αυτόν το σκοπό.

Οι εφήμερες απεικονίσεις και οι ασύγχρονες ακυρώσεις των απεικονίσεων του DaxVM σχετίζονται με οποιαδήποτε πρόσβαση στη μνήμη με εφήμερα χαρακτηριστικά. Αυτό θα μπορούσε να ισχύει τόσο για άμεση ή προσωρινή πρόσβαση σε απεικονίσεις αρχείων με προσωρινά αντίγραφα στη μνήμη (buffered access) ή ακόμα και για απεικονίσεις σωρού. Οι μνήμες πολλών ταχυτήτων και οι συσκευές αποθήκευσης υψηλής απόδοσης αλλάζουν γρήγορα τη χρήση της μνήμης ως κοινής πλέον διεπαφής σε πολλαπλά μέσα με ποικίλους χρόνους απόκρισης. Αυτό επιφέρει νέες προκλήσεις για τη διαχείριση των χώρων διευθύνσεων, αμφισβητώντας ό,τι ισχύει σήμερα.

[2] https://github.com/cslab-ntua/DaxVM-micro2022

CHAPTER 2

# Introduction

Modern systems, from hyperscale servers to accelerators, depend on virtual memory. This abstraction has been proven crucial to the success of computing and has stood the test of time. It allows programmers to develop code as if the available *memory is always sufficient, vast, linear and private* to their application. It also allows them to *access I/O devices in memory space*, using assembly instructions as if they were reading and writing memory. This simplifies further programming and potentially enables the *elimination of the operating system kernel* from the IO path, a technique employed for better performance.

However, this flexibility does not come for free. Since software uses virtual addresses to access the various tiers of store (main memory, PCI-e devices, etc), three sources of overhead prevail: (i) the management of processes virtual address spaces, (ii) the set-up of memory mappings, and (iii) the translation step required for every load and store CPU instruction. The first two significantly limit the scalability of virtual memory in the many-core era [48, 72] while a decade ago computing hit on the Address Translation Wall [53] and never fully overcame it.

Compute trends continue to stress the virtual memory design, testing all the three aforementioned mechanisms. The meteoric rise in memory demands, the shift to cloud computing, the increasing heterogeneity in the compute and store stacks and the dominance of many-cores and parallel programming as a way to extract performance, urge us to revisit the OS and hardware layers of virtual memory, along with their synergy.

In this thesis we focus on the overheads of address translation for memory-intensive workloads and on the performance of virtual memory operations for IO intensive workloads. In this

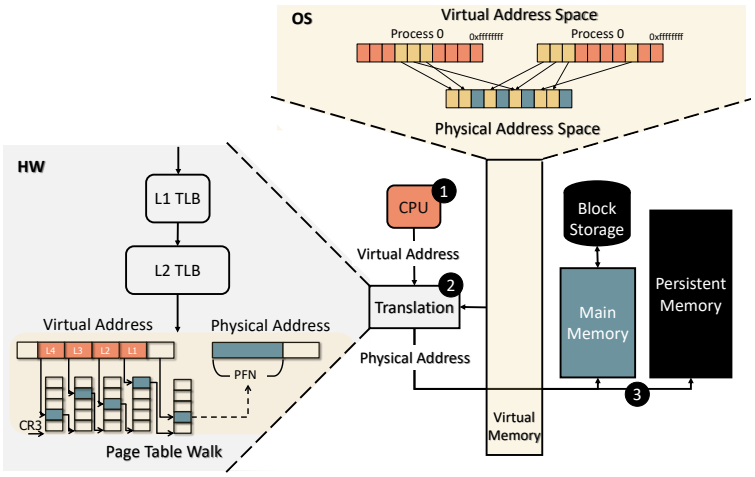



Figure 2.1: Virtual Memory Overview

chapter we first briefly review page-based virtual memory, both as a memory abstraction and as an interface towards files. We then motivate the performance and scalability bottlenecks that this thesis considers and finally introduce our contributions to address them.

## 2.1 Paged Virtual Memory

Virtual memory is a heavily co-designed mechanism where i) the OS manages processes virtual address spaces and maps them to physical store mediums (e.g. memory) while ii) dedicated architectural support accelerates the necessary address translation step on the execution of every load and store CPU instruction. Figure 2.1 gives an overview of the design when paging is used to manage memory. In addition to being the user-space abstraction of physical memory, virtual memory is also a valuable interface to storage (e.g. via file mappings).

### 2.1.1 Operating system

**Virtual Address Space Management.** Each process has a very large linear virtual address space that forms the memory abstraction exposed to the programmer. It is managed by the OS in page granularities with the page size being architecture dependent; for x86 it can be 4KB (small) or 2MB/1GB (huge). The OS allocates virtual address ranges from this space when processes request to allocate physical memory or to access files via memory mapping.

**Memory Mappings.** The OS maps the allocated virtual addresses to a backing medium, commonly the physical memory, to enable user-space data access via dereference. Physical memory is managed in page sizes as well (4KB) and the OS maps virtual pages to physical frames per process. With this indirection, processes have an isolated and protected view of memory but can still share pages (e.g. shared mappings). The OS stores the virtual-to-physical page mappings on

a dedicated data structure, commonly a radix tree, which is part of each process context and is named the page table. Each page table entry (PTE) holds a virtual-to-physical page translation. Page tables are architecture dependent, similar to page sizes, as they are also directly used by the HW layer of virtual memory (discussed later). The OS by default populates mappings per page, setting-up the corresponding page table entry during a trap known as page-fault.

**Physical Memory Management.** Despite not exclusively linked to virtual memory, physical memory management is intersected with virtual memory operations. By default, most operating systems allocate physical memory in page granularities and on demand during page faults – triggered by application cold accesses to address ranges. The allocation size (e.g. page size) affects the address translation performance of virtual memory, discussed in a next paragraph.

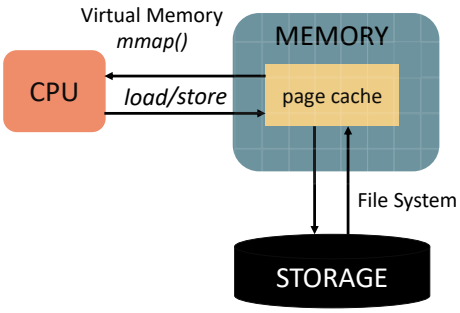



Figure 2.2: Buffered File Mappings

**File Mappings.** Virtual memory can also be used as an interface towards files. *Memory-mapped IO* is the technique that maps virtual addresses to file pages and enables file access via memory dereference. The OS allocates a virtual address range for each file mapping and for storage mediums/interconnects that only support *block-level access*, it *buffers files through physical memory* to map them to user-space (Figure 2.2). An additional OS layer (the *page cache*) associates file storage blocks to physical memory frames and, in orchestration with the file system, triggers copies between them. Once file data is copied to DRAM, the allocated virtual pages are mapped to the corresponding physical memory frames using the same paging mechanisms discussed in the previous paragraph. *With buffered memory-mapped IO, the files are copied from storage to physical memory to be accessed.*

The virtual address space is an inherently shared resource (among threads) and thus heavily contended in the many-core era. Unfortunately, most operating systems serialize address space operations [59,60,70,71,140], limiting virtual memory scalability. Paging software overheads (e.g. page fault latency) are also prevalent. Particularly for IO intensive workloads and file mappings, page faults can throttle performance [124, 125, 168, 169, 220]. All the above overheads have led practitioners to question the applicability of the interface towards storage, even for IO-heavy workloads like databases [84], despite its ease of use.

### 2.1.2 Architectural support for address translation

Virtual memory implies that every memory operation requires a physical translation to be executed. The processor translates the virtual address of every load and store instruction to issue a memory request that will fetch the corresponding data. As translation is on the memory access

critical path, most architectures use dedicated hardware caches per core, known as Translation Lookaside Buffers (TLBs), to store the most recently used translation entries (page table entries) (PTEs) and accelerate the procedure. If the translation required is found in the TLB hierarchy (hit) the look-up costs less than 10 cycles [117]. In case the translation required is missing though, the page tables of the process must be traversed to retrieve it. Most processors nowadays handle TLB misses entirely in hardware. On a TLB miss, the hardware triggers a state machine called the page table walker that walks the per-process page table and loads the corresponding PTE into the TLB. Page table walks are expensive as they can involve multiple main memory accesses and cost up to hundreds of cycles. This can induce significant slowdowns to memory intensive applications [47, 56, 130, 171, 172, 176, 177] and thus the TLB capacity –known as the TLB reach– and the TLB miss penalty are important performance factors.

Another important overhead source is address translation coherence. Keeping TLBs coherent with address space mappings (a.k.a page table entries) can be really expensive. TLB maintenance is part of virtual memory operations and a representative example is the tear down of mappings (during unmap requests). TLB entries that store stale page table entries must get invalidated, for security and correctness, and in x86 this is done in software. The OS triggers TLB invalidations (shootdowns), delivered by inter-processor interrupts, that serialize all virtual memory operations and can cost up to thousands of cycles [34, 35, 141]. TLB coherence is another notorious scalability bottleneck of the virtual memory co-designed interface.

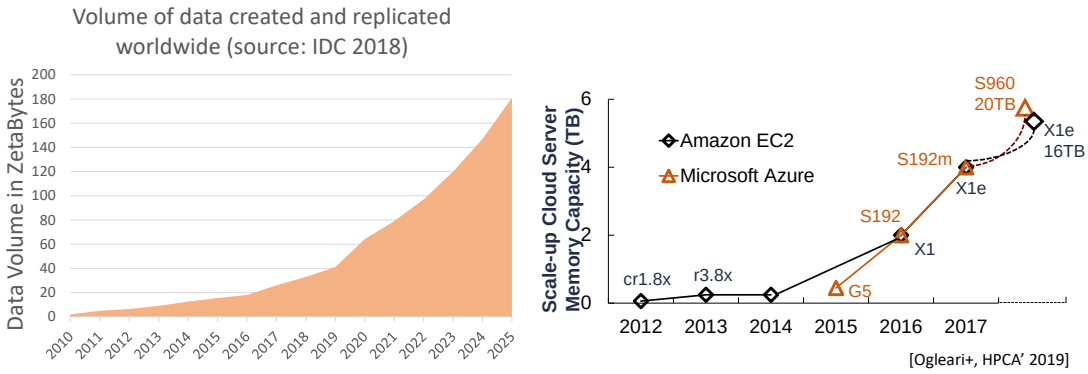
## 2.2 Motivation

High performance address translation, scalable virtual address spaces and low cost paging are important for virtual memory efficiency. In this section we discuss some compute trends that stress today's virtual memory design and, in our opinion, question the interface's scalability both to many cores and to increased store capacities.

### 2.2.1 Meteoric rise in memory demand

Driven by a confluence of mega-trends, global data traffic is increasing at a nearly exponential rate (Figure 2.3a). Artificial intelligence/machine learning (AI/ML) is the marquee example of advanced workloads with increasing demand for data, with skyrocketing data set growth rates of 10x annually [49, 81]. Other examples are big-data analytics [43, 82] and in-memory computing [97, 213]. The impact of all this growth is felt intensely in data centers. Figure 2.3b shows how cloud server memory capacity has been rapidly growing [163].

Virtual memory design is stressed to successfully support the staggering increase in memory footprints, becoming itself a system performance bottleneck. The primary problem is limited TLB




(a) Total data volume

(b) Memory capacities follow the trend

Figure 2.3: Meteoric rise in memory demands

reach that εexposed address translation as a bottleneck a decade ago. The TLB hierarchy fails to cover the active working sets of big-data workloads, especially when they are irregular e.g. graph processing. The problem was highlighted by academia [47, 130, 176, 177] and acknowledged by industry with processor vendors increasing the chip area budgets on TLBs ever since. Figure 2.4 summarizes how the TLB reach of Intel processors has been evolving the past decade, with each entry depicting the coverage achieved when the largest supported page size is used 100%. There is a constant increase via a two-fold approach: i) larger TLBs and ii) better support for huge pages (2MB and 1GB). However, TLB reach in the order of TBs, same to memory capacities, is possible only when 1GB pages are used 100% in the latest processors. Otherwise TLB reach is measured in the order of GBs. Unfortunately 1GB pages are not transparently supported by any operating system, and such a support is not straight-forward as discussed further in Section 5.2. Moreover, this increase is expensive in terms of power, TLBs can consume 15-20% of chip energy [53, 131].

Apart from increasing the TLB reach, processor vendors have invested in a rather complex hardware design to control TLB miss latency penalties. MMU caches store the upper level of

<table>
<tr><th rowspan="2">Year</th><th rowspan="2">Processor</th><th rowspan="2">Paging</th><th colspan="4">L1 TLB</th><th colspan="3">L2 STLB</th></tr>
<tr><th>4KB</th><th>2MB</th><th>4MB</th><th>1GB</th><th>4KB</th><th>2MB</th><th>1GB</th></tr>
<tr><td>2012</td><td>Ivy Bridge</td><td rowspan="8">4-level</td><td rowspan="9">256KB</td><td rowspan="3">64MB</td><td rowspan="3">none</td><td rowspan="9">4GB</td><td>2MB</td><td>none</td><td>none</td></tr>
<tr><td>2013</td><td>Haswell</td><td colspan="2">2GB</td><td>none</td></tr>
<tr><td>2014</td><td>Broadwell</td><td rowspan="7">3GB</td><td rowspan="7">16GB</td></tr>
<tr><td>2015</td><td>SkyLake</td><td rowspan="6">128MB</td></tr>
<tr><td>2016</td><td>KabyLake</td></tr>
<tr><td>2017</td><td>CoffeeLake</td></tr>
<tr><td>2018</td><td>CannonLake</td></tr>
<tr><td>2019</td><td>IceLake</td><td>5-level</td><td>8GB</td><td colspan="2">1TB (1GB) or 2GB (2MB)</td></tr>
</table>

Figure 2.4: Intel processors paging and TLB reach evolution

page table radix trees to decrease the number of memory accesses required during a page table walk. Moreover, multi-threaded hardware page table walkers service more than a single TLB miss concurrently (e.g. Intel's Skylake has a 2-way walker).

Still, in this thesis we show that irregular big-memory workloads, running natively on a modern processor and using only transparently supported huge pages, still suffer up to 30% overheads due to address translation.

#### 2.2.1.1 Memory expansion via CXL

In response to the exponential growth in data, the industry is on the threshold of an architectural shift. Data centers are moving from a model where each server has its own dedicated memory, to a dis-aggregated model that employs pools of shared memory resources. While dis-aggregation and universal interfaces across processing units have been around for many years, the industry's convergence on the Compute Express Link (CXL [85]), as a cache-coherent interconnect for processors, memory and accelerators, is a critical enabler to make these concepts a reality [148, 156]. With the soon to be available CXL-compatible processors and DDR devices, we enter a new era of multiple "tiers" of memory, each with different latency and bandwidth characteristics, but with potential aggregated capacities of peta-bytes [185]. CXL-memory is still accessed via virtual memory [187], most likely exposed to the operating system as a NUMA CPU-less node, stressing further TLB reach requirements.

#### 2.2.1.2 5-level paging

Another hard constraint reached, as memory capacities grow, is that x86 4-level paging cannot address more than 64TB of physical memory [11]. Intel responded to the problem at hand by adding an extra level on the page table radix tree [22, 77], and enable coverage of up to 4 PiB of physical address space. 5-level paging architectural support has been introduced in the recent Ice Lake processor (Figure 3.3), and Linux supports it as well. Adding another level in paging adds extra cost on the TLB miss path, as an extra memory access is required during page table walks. As we discuss in the next section, this extra overhead is amplified in virtualized execution.

Despite the increased architectural support for paged virtual memory that exists in modern processors, the TLB reach is still limited and fails to scale with the ever growing memory demands. Memory dis-aggregation can lead to petabyte-scale memory systems, stressing further TLB reach requirements, and adding paging levels to support the larger scale inherently makes today's address translation mechanism costlier.

### 2.2.2 Virtualization dominance

Virtualization is an old idea [105] revived by Disco [61] in the late 90s and grown today to be the main technique of computing deployment. It adds an extra layer of indirection between the Operating Systems (OS) and the hardware, known also as the Virtual Machine Monitor – e.g. KVM [138]. Virtualization enables better resource management and isolation. Especially in the form of Virtual Machines (VMs), it allows on-demand hardware allocation and thus better server utilization; ruling cloud computing infrastructures.

Virtualizing hardware –CPUs, IO, memory– however, commonly comes at a cost. Particularly memory virtualization has proven to be very expensive and difficult to accelerate [30, 101, 102, 193, 198]. The guest machine's view of physical memory is an abstraction itself and is independently mapped to the host server physical memory where data actually reside. Therefore an extra level of translation is required for every memory operation of applications running inside the VM. With hardware-assisted virtualization, the guest virtual address (gVA) of each CPU load/store instruction is first translated to the guest physical address (gPA) and then to the host server's physical address (hPA) where data is stored (Section 5.2). During a TLB miss, the two-dimensional translation is performed in hardware with a nested walk of the guest and the host OS page tables. This nesting makes TLB misses notoriously expensive as the number of memory accesses is multiplied by the levels of paging, e.g. up to 24 memory accesses are required in virtualized execution while only 4 in native when we consider 4-level paging.

We find that, despite all the architectural support –e.g. MMU caches– memory virtualization results in a 2x penalty compared to native execution on a modern processor.

The ongoing seismic shift of business applications and databases moving from enterprise data centers to the cloud [81], e.g. the uprising serverless paradigm [27, 189], make efficient memory virtualization a high-priority design goal. Address translation in such two-dimensional environments bears amplified costs that can throttle performance.

### 2.2.3 The evolving storage stack and direct access to persistent data

IO devices have significantly evolved the past decade and can offer single-digit µs access latencies (e.g. low-latency SSDs [132, 218]), or even lower (e.g. persistent memory [119]), shifting entirely the design rules for efficient data-center IO system software support. The current OS stack has been largely tailored to hide ms-scale latencies (e.g. hard-disk drive IO) and multiple of the mechanisms it employs (e.g. context-switching) have overheads suddenly comparable to the new low-latency devices [46]. This mismatch has been identified multiple times in literature (e.g. [46,175]) and numerous works show how execution is no longer bound by hardware performance as servers often spend much of their time executing operating system code (e.g. [64,146]). The OS

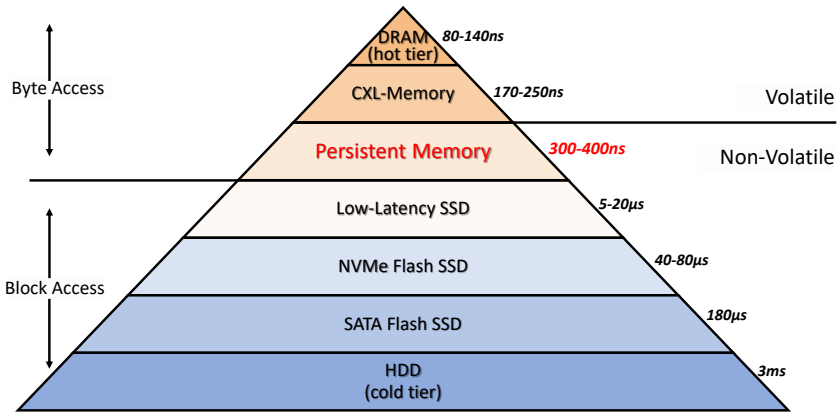



Figure 2.5: Today's multiple tiers of store [132, 156, 218]

has suddenly become the bottleneck and system software must be re-designed to operate at the sub-microsecond or nanosecond scale to successfully deliver to user-space what the underlying hardware can provide [236]. Figure 2.5 gives an overview of a modern store stack. This thesis focuses on virtual memory as the OS interface to persistent memory, a rather unique non-volatile tier that combines ultra low latency with byte-addressability.

Persistent memory (PMem) is a new storage technology [119,186] that is connected to the system via the memory bus or the CXL [85] link, similar to DRAM, and therefore is accessible via CPU load and store instructions. The technology uniquely combines non-volatility and byte-addressability with latency and bandwidth closer to that of DRAM (Figure 2.5). This blurs the decades-old distinction between slow but persistent storage and fast but volatile memory. The high IO performance –at the scale of nanoseconds– makes storage accesses much cheaper than OS invocations and reducing the OS overheads becomes a strong requirement [15, 67, 68, 74, 90, 125, 142, 149, 162, 211, 216, 219, 220, 221, 222, 230]. Virtual memory has a leading role under this scope as it enables *direct access* to storage. Memory mapping PMem files (via mmap system calls) can map processes virtual pages directly to persistent memory physical locations, *bypassing entirely any DRAM buffering (e.g. the page cache)* (Section 2.1.1) and essentially mapping storage directly to user-space (Figure 2.6). This forms the *the first true zero-copy access path to storage (no DRAM data copies)* as applications access directly persistent data via pointer dereference (load/store CPU instructions).

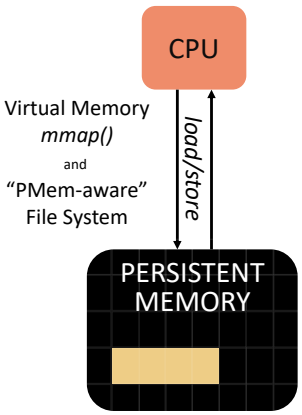



Figure 2.6: Direct Access

Consequently direct access mmap() is a "killer app" for PMem as it gives applications the fastest possible access to stored data [222]. However virtual memory overheads can often throttle performance [124, 125, 149, 220]. Figure 2.7 shows the average latency of reading 256KB files

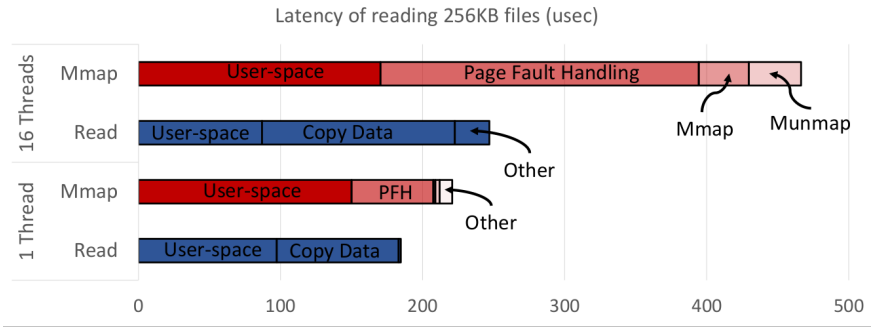



Figure 2.7: Average latency of accessing once 256KB files. The reported latency if for a single file. Direct access performs worse than syscalls (e.g. read) despite avoiding data copies. This is due to expensive virtual memory operations. We measure this on a machine with 384GB Intel Optane 5.5.1 and use bpftrace [108] to track average latency.

(summing up their content in 8-byte word granularities) using either direct access file mappings or the read system calls. Read inherently copies file data from persistent memory to a private DRAM buffer to access them. We observe that direct access performs significantly worse than read despite its zero-copy advantage. It spends ∼30% of the total time (single thread) executing OS code that i) sets up the mapping (mmap), ii) populates its page tables (page faults) and iii) tears it down while invalidating TLBs (munmap). These kernel overheads grow to ∼60% when multiple threads process multiple files concurrently, due to severe contention on virtual memory locks [70, 71, 72, 140, 141]. These findings corroborate past studies that highlight paging as a significant performance factor for PMem direct access (e.g. [124, 149, 220]).

> "Playing games with the virtual memory mapping is very expensive in itself. It has a number of quite real disadvantages (e.g. setup/teardown costs and page faulting) that people tend to ignore because memory copying (e.g. read) is seen as something very slow."
> —Linus Torvalds, 2000 [205]

The implications of virtual memory as a file interface in Linux are known since the year 2000, as Linus Torvalds quote reflects, and have remain surprisingly the same for over twenty years. However, with block-level access storage (e.g. SSDs) file data must still be copied from the device to the volatile memory to be mapped (page cache, Section 2.1.1) It is the era of low latencies and *zero-copy direct access* that we believe urge us to re-think the design of this legacy interface.

Today's IO devices can offer micro and sub-microsecond latencies exposing the system software stack as the bottleneck. Virtual memory performance, operating as an interface towards files, is critical for storage supporting *direct access*, e.g. persistent memory. Today's design introduces high latencies and scales poorly to many cores, failing to deliver what the underlying hardware can provide.

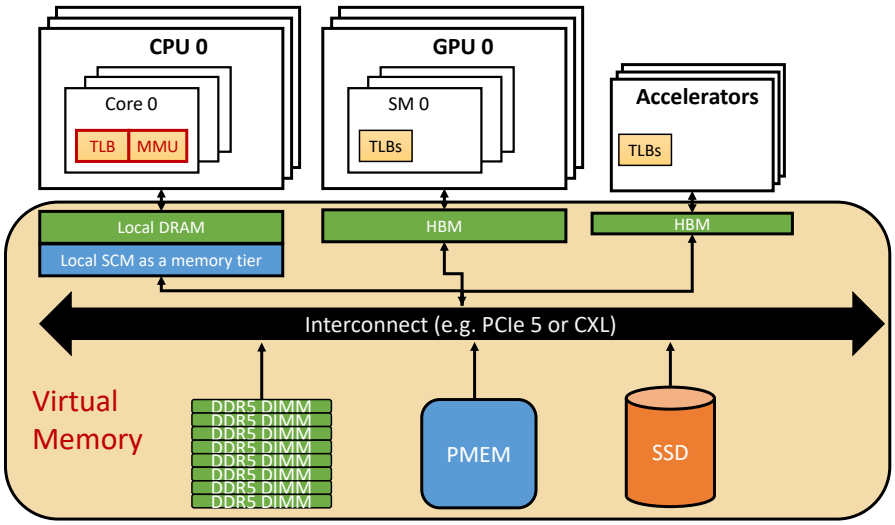



Figure 2.8: A common interface in a heterogeneous coherent world

### 2.2.4 A common interface in a rapidly growing heterogeneous world

"What fundamental opportunities for scalability are latent in interfaces, such as system call APIs? Can they be identified considering interface specifications only?"
—Austin T. Clements [72]

Virtual memory was designed at a period that memory systems were largely homogeneous, with a single local memory tier backed by a local storage device. Nowadays the entire store stack comes as a hierarchy and each tier has varying characteristics in terms of latency, bandwidth and access (e.g. buffered or direct). Also multiple processing units and their local memories are coherently inter-connected. Figure 2.8 shows an example of the new heterogeneous compute and store landscape. Virtual memory must pace with the rapidly expanding system architecture as a common interface for all components. This stresses further the address translation requirements, discussed in Section 2.2.1, as translation must reach capacities way beyond the size of local memory. It also shakes the core assumptions of this legacy interface. Virtual memory semantics –e.g. POSIX– have remained largely intact over the years and homogeneous for all types of memory mappings, irrespective of the backing store medium. We believe that significant performance potential is latent in this inertia.

The impact of interfaces semantics on system software scalability and performance is a formally studied topic by Clements et.al. [72]. With respect to virtual memory, various works propose radical changes to address space [59, 72, 91] or TLB [72, 141] management to improve performance and often relax POSIX requirements in the process. In this thesis –and in continuation to the previous subsection– we focus on virtual memory semantics as a *direct access* interface

to fast byte-addressable storage. Figure 2.7 shows how direct access scalability suffers from synchronous resource release (e.g. destroying mappings synchronously to munmap requests) and from populating mappings in multiples of the page size (faults). The first derives from a strong POSIX requirement while the second is mostly specific to the Linux implementation. We wonder if and how these (and other state-of-practice specifications) are still relevant when virtual addresses map directly storage, a much more abundant resource than physical memory that is reclaimed slowly. Another example question is do we still need a common address space or a unified manager for heap and storage mappings in the presence of direct access? Re-thinking, extending or altering interfaces for efficient PMem access is an established strategy but prior work mostly considers file system APIs and designs (e.g. [211], [220]).

Memory tiering and fast storage rapidly change the usage of memory as a now common interface to multiple mediums with varying characteristics (e.g. volatility, performance etc). This imposes new challenges to the virtual memory design, questioning the state-of-practice.

## 2.3 Problem Statement

This thesis focuses on the improvement of virtual memory performance in terms of i) address translation and ii) direct access to byte-addressable storage (e.g. persistent memory). To shape and evaluate our approaches in system environments representative of real-world deployments, we implement and perform most of our work on real hardware and stock Linux. For hardware proposals we employ performance models of hardware and project improvements.

**High Address Translation Overheads.** Address translation performance fails to scale with the meteoric rise in memory demands, primarily due to limited TLB reach. Prior work and this thesis show that page walks can still dominate execution, especially in virtualized environments due to nested paging, despite the increased budget for translation hardware in recent processors (e.g. [47, 101, 130, 171, 172, 194, 199]).

The first goal of this thesis is to minimize translation costs for both native and virtualized execution working at the boundary/intersection of the hardware and software (OS) layers of virtual memory. We aim at a (re)design that: i) offers near-zero address translation costs, ii) scales with the ever-increasing memory capacities, iii) maintains flexible, lightweight physical memory management and iv) is transparent to applications.

**Limiting interface for direct access to persistent data.** Persistent memory is a unique storage technology that combines ultra low latency, byte-addressability and inter-connection with the memory bus. Mapping PMem files with virtual memory can provide *direct access* to persistent

data via CPU load and store instructions, forming the shortest available path to storage. However, prior works and this thesis show that virtual memory operations introduce high latencies and scale poorly to many cores, limiting direct access performance (e.g. [124, 125, 149, 220]).

The second goal of this thesis is to study virtual memory as a file interface and identify all the sources of overhead in the current design. Based on the findings we aim to re-think and re-design the virtual memory interface – along with its semantics – for direct access to persistent data, targeting to come close to what the underlying hardware can provide in terms of performance.

## 2.4 Proposals

This thesis makes the following contributions:

- To reduce address translation costs we propose *Contiguity-Aware Paging* and *Speculative Offset Address Translation*, a virtual memory co-designed implementation that hides address translation overheads under speculative execution. The technique is applicable to both native and virtualized execution.
- To improve the performance and scalability of *direct access* to persistent memory we propose *DaxVM*. DaxVM is a novel POSIX-relaxed file mapping interface with a virtual memory re-design that significantly reduces system software overheads for PMem access.

Next we highlight the most important concepts of each contribution.

### 2.4.1 Efficient Memory Virtualization via Contiguous Mappings

A long line of research [47, 101, 130, 172, 177, 227] exploits contiguity in mappings, i.e. pages contiguously mapped in the virtual and physical address spaces per process, to accelerate address translation. However, prior work lacks lightweight memory management support to generate the desired contiguity – it commonly breaks fundamental OS principles. Also, most proposed hardware designs do not support virtualization. To solve these problems in this thesis we propose i) *Contiguity-Aware (CA) paging* and ii) *Speculative Offset Address Translation (SpOT)*.

CA paging extends demand paging, the default physical memory management technique of modern Operating Systems. It creates lazily and at a best-effort basis contiguous mappings by allocating target physical pages across page faults, preserving all core lightweight management techniques such as paging, on demand allocation, Copy-On-Write etc. It is inherently compatible to nested paging, applied independently by the guest and host OS to generate contiguous mappings in each dimension. It maintains minimal metadata per memory mapping to identify target physical pages for allocation during faults. It also employs a system contiguity map and

mapping placement policies to deal with external fragmentation, but without reserving or pre-allocating physical memory. It supports heap and file mappings, multi-threaded workloads and multi-program execution. We've implemented it in Linux and made it publicly available[1].

SpOT is a micro-architectural speculation engine, that predicts the missing physical address translation on a TLB miss. It feeds the address to the processor to continue the execution in speculative mode while verification, e.g. (nested) page walk, happens in the background. The key observation is that we can exploit the underlying contiguity, generated by CA paging, to predict translations. We show that tracking and caching the boundaries of arbitrarily sized contiguous mappings in virtualized execution is architecturally complex, by extending the state-of-the-art RMM design [130]. We consider this complexity the reason why most prior work designs do not support virtualization. The proposed SpOT design on the other hand achieves high performance, trading architectural complexity with strong security guarantees.

CA paging is a SW-enabler for any HW-method that exploits linearity in mappings to accelerate translations, and combined with SpOT they form a drop-in optimization for any HW technique that involves translation caching. We evaluate them on the side of paging with transparent huge page support enabled. The co-designed proposed scheme reduces translation overheads to <1% on average for memory-intensive workloads running inside virtual machines.

### 2.4.2 Stressing the Limits of Memory as a File Interface

In this thesis we study virtual memory interface for direct access (DAX) [4] to persistent data. We identify all the overhead sources and propose ***DaxVM***, a new mapping interface that pushes system performance close to what the underlying storage hardware can provide.

Direct access to byte-addressable storage omits the need for any form of DRAM buffering of persistent data (e.g. page cache); yet we identify multiple virtual memory design points that assume data is always buffered through memory to be accessed. We show how this affects virtual memory operations performance under DAX. In example, mappings are always lazily populated to save scarce DRAM resources and TLBs are synchronously invalidated when mappings are teared down to free memory. DaxVM maintains ***pre-populated persistent page tables*** per file and (de)attaches them to process address spaces during m(un)map operations to eliminate paging overheads and provide ***O(1) mmap*** [201]. DaxVM also batches unmap requests and ***flushes TLBs lazily*** for direct access mappings, significantly boosting virtual memory scalability to many cores. Another key observation is that file mappings for PMem storage may live shortly compared to heap or memory-buffered file mappings. In example, common IO-bound multi-threaded applications that operate over small files –e.g. web-servers, mail-servers or file-servers– access files once and with direct access avoid entirely the DRAM copies of persistent

[1]https://github.com/cslab-ntua/contiguity-isca2020

data. DaxVM provides a dedicated virtual address space (de)allocator for such ***ephemeral mappings*** that scales better to many cores. Also, direct access file mappings enable durability control/enforcement from user-space and this is considered as good programming practice [220]. For such cases, DaxVM *eliminates entirely kernel-space tracking of dirty pages*. Finally, in this thesis we show that zeroing newly allocated blocks is an overhead introduced by direct access for security reasons and can throttle append performance. We propose asynchronous zero-out of storage blocks by PMem file systems instead, following proposals for volatile memory [179]. DaxVM design is exposed as a new interface, with semantics that allow developers to enable/disable most optimizations based on the application's performance and security requirements.

DaxVM re-thinks and relaxes POSIX strict requirements for DAX mappings, following known rules [72] for interface scalability. We implemented DaxVM in Linux and two state-of-practice and state-of-the-art PMem-optimized file systems – ext4-DAX [216] and NOVA [221] – and made it publicly available [2]. For multi-threaded workloads that process multiple small files for short intervals, e.g., Apache, DaxVM improves standard mmap performance up to 4.9x. It also reverses the trend that favors read for such setups, outperforming it by up to 1.5x. DaxVM also increases system availability, providing fast boot times for PMem databases, and sustains their high throughput even when running on fragmented file system images.

DaxVM, despite being designed on Intel Optane, is relevant to multiple fast storage technologies. It is directly applicable to any byte-addressable device, a design advocated by the emerging Compute Express Link [85] – e.g., Samsung's memory-semantic SSD [186]. Moreover, state-of-the-art flash memory technologies have reduced storage-access latency to tens of microseconds [175] and fast file data blocks indexing [162] has become a critical performance factor. DaxVM's persistent page tables can be leveraged for this scope.

DaxVM's ephemeral mappings and asynchronous unmappings are relevant to any memory access with ephemeral characteristics. This could apply both to direct or buffered memory-mapped storage access or even heap mappings. Memory tiering and fast storage rapidly change the usage of memory as a now common interface to multiple mediums with varying latencies. This imposes new challenges to address space management, questioning the state-of-practice.

## 2.5 Thesis Organization

Chapter 2 provides additional background on page based virtual memory, physical memory management, address translation in native and virtualized environments and finally on persistent memory and its available interfaces.

Chapter 3 presents our co-designed virtual memory implementation to address high address translation overheads in virtualized execution and scale better translation performance with the

[2] https://github.com/cslab-ntua/DaxVM-micro2022

ever-increasing memory capacities. It describes (i) *Contiguity-aware paging (CA paging)*, (ii) the hardware virtualization extensions for Redundant Memory Mappings to support virtual machines and (iii) the micro-architectural support for *Speculative Offset Address Translation (SpOT)*. This chapter follows mostly from our work published in the 47nd International Symposium on Computer Architecture (ISCA 2020) [33].

Chapter 4 presents our study on virtual memory overheads over persistent memory DAX file mappings and describes DaxVM, the POSIX-relax file mapping interface we propose. It gives details for our real system implementation and evaluation. This chapter follows mostly from our work published in the 55th IEEE/ACM International Symposium on Microarchitecture [32].

Chapter 5 concludes this thesis and points to future research directions.

CHAPTER 3

# Background

This chapter provides background on virtual memory as a physical memory abstraction and as an interface towards files. More specifically, we discuss the functionality it provides and we introduce the basic concepts of virtual memory's most common implementation –paging. We then discuss the necessity for address translation and its state-of-practice and state-of-the-art architectural support in native and virtualized execution. We also go through the OS internals with respect to (i) virtual address space management, (ii) mapping management and address translation and (iii) physical memory management. We finally discuss the file mapping interface and focus on the unique case of direct access (DAX) and persistent memory (PMem). We go through the software and hardware layers of the PMem data path. We comment on how the OS virtual memory internals apply here, and we introduce also PMem-aware file systems. We finally discuss good programming practices for PMem access and some state-of-the-art proposals for efficient PMem interfaces. At the end of each section we have some key take-away messages on how the basic concepts discussed apply on later chapters of this thesis.

## 3.1 The Virtual Memory Abstraction

Virtual memory was invented in the late 50's [136] as a response to the problem of scarce main memory resources and the consequent programming burden of allocating/managing both memory and auxiliary storage, transferring manually the required data from one to another at the various computation phases of workloads. In turn, virtual memory provides the programmer

with a big flat unified view of memory, known as the *virtual address space*, and the OS manages automatically and under the hood the allocation and data movement between the various tiers of store. This allows the automatic deployment of workloads whose memory needs to expand beyond the limits of available physical memory in a machine.

Apart from the catalysing boost in programmability, due to the resource management abstraction, other aspects made virtual memory a dominant computing mechanism as well. For example, the level of indirection that it introduces enables:

- Consolidation of multiple processes running on the same machine, as the OS efficiently manages and shares physical resources between them. One example of agile physical memory management is demand paging, which we cover later on this section.
- Protection enforcement, as access rights can be set per virtual address ranges in a process address space – enforced by the hardware layer of virtual memory discussed next. In example, parts of the address space like text (code binary) are marked as read-only and thus programming bugs cannot overwrite them and corrupt data. Per process address spaces and protection also enable the isolation of processes, as they can not access memory that they do not own.
- Sharing of data. Virtual address ranges, even in multiple processes address spaces, can be mapped to the same physical location of a shared data block (e.g. library code) – saving physical resources.

An impactful virtual memory attribute is the granularity in which the OS manages the virtual address space of a process. Under this scope, paging was introduced in the late 60's [86] and has been the dominant virtual memory implementation since then.

## 3.2 Paging and Address Translation

Figure 3.1 gives an example of page-based virtual memory management. There are four key conceptions (i) the virtual address space, (ii) the physical address space, (iii) the page mappings and (iv) address translation.

**Virtual address space.** This is private per process and constitutes the set of virtual memory addresses that user-space can use. The OS manages and makes this set available to each process. In a 64-bit x86 system with 4-level paging (discussed next) this is commonly the range $[0, 2^{48}]$, meaning that each process can potentially address 256TB of private store.

**Physical address space.** This is the set of addresses used by the underlying hardware. It commonly refers to main memory physical addresses, but it can also reflect locations in other devices (e.g. IO as we will see later in this chapter). The Operating System is in charge of allocating

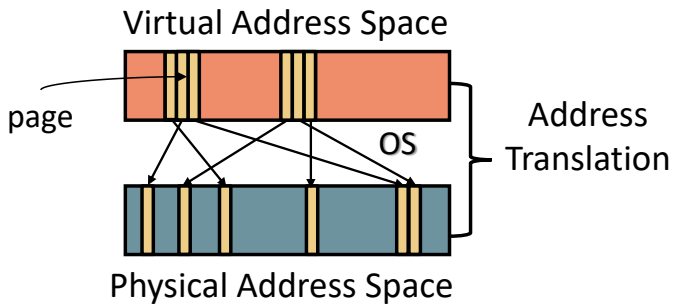



Figure 3.1: Paged Virtual Memory.

physical memory blocks for the various running processes. It enables user-space access to the allocated blocks by mapping them into processes virtual address spaces.

**Pages and the virtual-to-physical page mappings.** Paging is named after the granularity in which the OS manages the virtual and physical address space, the page – an architecture dependent and fixed size. In most architectures the base page size is 4KB and this is also the granularity in which physical memory is managed by the OS. To expose physical memory to user-space, the OS divides each process address space in page chunks, named virtual pages, and maps them to equally sized physical page frames,e.g. 4KB blocks of free memory. There are a few larger page sizes, multiples to base (e.g. 2MB and 1GB), that are supported in mappings – discussed later in this section. These virtual-to-physical page mappings per process are maintained by the OS and are exposed to the Memory Management Unit (MMU) of the CPU.

**Address Translation.** As programmers use virtual addresses to access data, there is a translation step required in every memory operation executed by a running process on the CPU. Every load and store instruction issues an access on a virtual address, thus its physical translation is necessary for the CPU to request and fetch the corresponding data from the memory subsystem. This step is an inherent cost of virtual memory and CPUs are equipped with architectural support to accelerate it.

### 3.2.1 Address Translation Hardware

Virtual Memory and particularly its address translation step, is perhaps the most representative example of hardware and software co-design. In this section we discuss how OS maintained information is accessed and exploited by the Memory Management Unit (MMU) of a modern CPU to accelerate address translation.

**Page Tables.** The Operating System stores the virtual-to-physical page mappings of every process in dedicated data structures, named page tables, which are also accessed by the address translation hardware. Page tables are indexed by virtual page numbers and the resulting page table entry stores the *number of the physical page frame* where the corresponding data is stored.

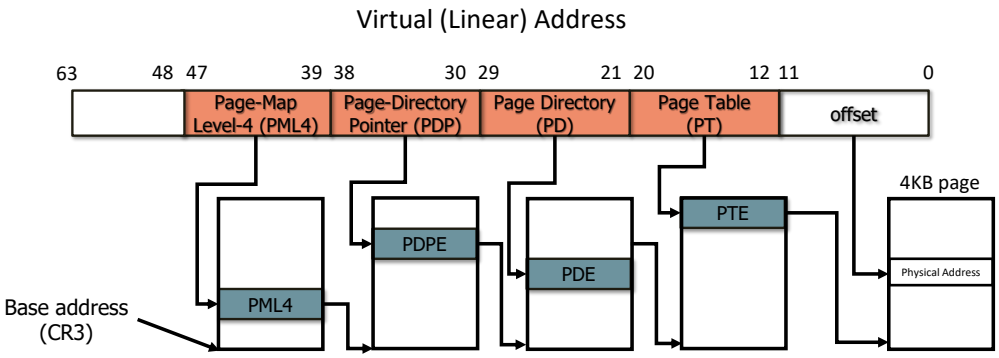



Figure 3.2: The multi-level organization of the page table in x86-64 architecture with 4-level paging. Page walks access all levels of the table requiring an equal number of memory references.

In other words, each process's page tables store the valid virtual-to-physical page translations for its entire address space. Hence, page table traversals, known as page walks, can retrieve the physical translation of any mapped virtual address of a process.

**Page Table Entry (PTE) Metadata.** Apart from the *physical frame number –the address translation of the virtual page–*, a PTE also stores metadata for the virtual-to-physical page mapping that enable/control much of the virtual memory functionality discussed in 3.1 (e.g. protection) and monitor user-space page access. In x86 the metadata bits of each 64-bit PTE entry are [123]:

- The present (or valid) bit: must be set to 1 to map a valid 4KB page.
- The protection bit: if set to 0 writes are not allowed to the 4KB page referenced by this entry.
- The privilege bit (user/supervisor): if set to 0 user-mode access is not allowed to the 4KB page referenced by this entry.
- The cache management bits (write-through bit, cache disable bit, pat bits): a conjunction of them sets the type of caching of memory access to the 4KB page referenced by this entry.
- The access bit: indicates if software has accessed the 4KB page referenced by this entry.
- The dirty bit: indicates if software has written to the 4KB page referenced by this entry.
- The protection key bits: determine the protection key of the page. Memory protection keys (MPK) are an additional mechanism to control access to user-mode addresses, with lower performance overheads compared to the per-page protection bit, offering also intra-process memory isolation (different protection among threads of the same process) [133].
- The no execute bit: if set to 1 instruction fetches are not allowed from the 4KB page controlled by this entry.

In Chapter 5 we discuss how many of the page table entry metadata are tailored for volatile memory monitoring (e.g. access and dirty bits) and are irrelevant for direct access to storage (introduced in the following subsections).

**Multi-level paging.** Paging is nowadays commonly multi-level, meaning that page-tables are organized in a multi-level hierarchy. Figure 3.2 shows the page table organization of x86 [123] with 4-level paging. Each entry of the three intermediate levels (e.g. PML, PDPE, PDE) stores the base address of the next level page table. The last level page table entry (PTE) stores the base address of the physical 4KB frame that holds the corresponding data. The virtual page number (48 bits of the full virtual address minus the bits of the page offset) is divided into 4 regions and each region indexes the corresponding page table level. Figure 3.2 shows the top-down traversal of the page table to retrieve a missing translation of a 4KB virtual page.

Page tables could be implemented as monolithic flat arrays but that would be a significant waste of memory. Processes tend to populate sparsely their address spaces –allocate and use fragments of it– thus multiple entries would always stay "empty" or "invalid", as the corresponding virtual pages would never get mapped to physical memory. With multi-level page table organization we only need the root of the table tree to always be in memory and the rest of the levels can be built (or paged in from disk) on demand as the process populates (or accesses) regions of its address space. Multi-level paging is a memory-efficient page table organization. On the other hand, more levels in paging imply more memory references to get the physical address of the target virtual page – one access for each level required. This is commonly referred to as the page table walk cost. Multiple state-of-the-art studies propose different page table organizations, e.g. hashing [198, 199, 234] or flattening [173], targeting a lower walk cost.

**The page table base register.** The page tables are part of a process *control block* or *context*. As we will discuss in a later section they are maintained by the OS per process and when the process is scheduled-in to a CPU core (context switch), the OS sets the base (root) physical address of the page table tree to the CPU *page table base register (i.e. CR3)*. This enables hardware access to the tables.

**Hardware Page Table Walkers.** Nowadays most processors provide support for hardware traversals of the page tables to retrieve the physical translation for a virtual page during load/store operations. A hardware finite state machine, the page table walker, uses the CR3 register to walk top-down the tables in the same way as depicted in Figure 3.2.

In Chapter 4 we show how page walk costs are an important execution overhead for memory intensive workloads and we propose techniques to hide this cost.

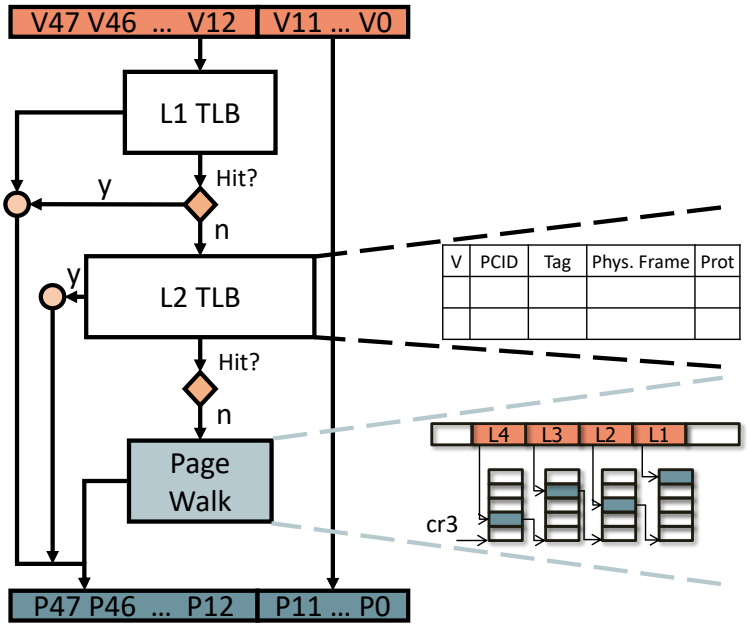



Figure 3.3: Address Translation with hierarchical TLBs is performed entirely in HW.

**Translation Lookaside Buffers.** Walking the tables on every load/store CPU instruction, to retrieve the physical translation of the target virtual address, is prohibitively expensive even when the walk is performed entirely in HW. It would add the extra cost of multiple page walk memory accesses on every user-space memory reference. For that reason, most processors maintain hardware caches close to the cores, the *Translation Lookaside Buffers*, that store the most recently used page table entries. TLBs are commonly organized in a two-level hierarchy (Figure 3.3). The L1 TLB is small, set or fully associative, and its purpose is to support very fast search operations in parallel with the L1 data cache access, as the latter is commonly virtually indexed. The L2 TLB is larger and its purpose is to store more page table entries. On the execution of every memory instruction, the TLB hierarchy is first searched for the missing translation, and if found (hit), the look-up has costed less than 10 cycles [117] – significantly cheaper than accessing main memory. On a TLB miss event, a hardware page walk is triggered to retrieve the missing translation. The *TLB reach* –the amount of memory accessible from the TLB or in other words its hit ratio– is one of the most important performance factors of address translation [47, 101, 102, 171, 172, 176, 177, 178]. Figure 3.3 summarizes the core steps of the address translation hardware path.

**Address Space Identifiers.** As all processes have access to a private set of the same range of virtual addresses –e.g. all have a virtual address space of $[0,2^{48}]$– the problem of TLB maintenance during context-switches arises (when we switch the process that runs on a CPU core, e.g. scheduling). To avoid flushing the entire TLB, most processors nowadays support address space

identifiers. ASIDs or Intel's PCIDs (process context identifiers) uniquely identify each process and TLB entries are also tagged with them. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. This saves the need for TLB flushes whenever the root page table address (e.g CR3) changes on the CPU, i.e during context switching or privilege switching (e.g. system calls), in most cases.

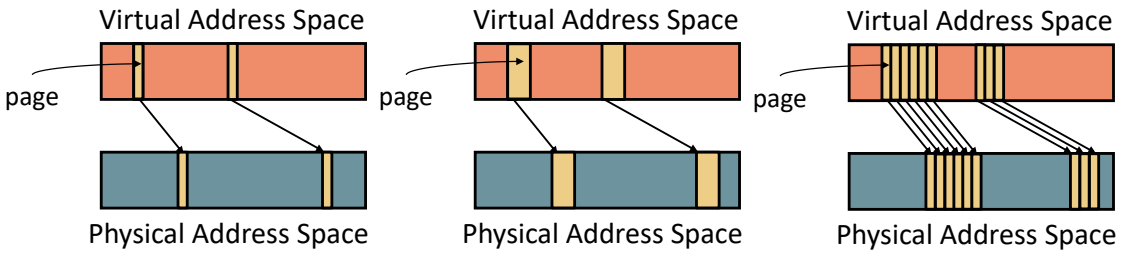
**MMU caches.** To minimize the costs of the page walks that TLB misses trigger, most processors are equipped with extra MMU caches storing intermediate levels of the page table tree, i.e. PML4 and PDP or PD levels [44, 52]. Page walks that hit on the MMU caches or even on the CPU data caches, potentially storing also page table entries fetched by the walks that access main memory, are significantly cheaper. The reference to each level of the tree during a page walk (Figure 3.2) no longer necessarily translates to an expensive access to main memory. Despite this significant acceleration, in Chapter 4 we show that page walk overheads can still be an important percentage of the execution time in memory-intensive workloads.

Note that the memory hierarchy may also cache any level of the page table. For example, Intel processors may cache the page table up to the L1 cache. This accelerates the page walk references made by the hardware page table walker.

### 3.2.2 Contiguity in mappings can be exploited to accelerate address translation

One of paging's most important properties is the non-contiguous physical memory allocation, i.e. contiguous virtual addresses are not necessarily mapped to contiguous physical blocks (Figure 3.1). This property is important for internal and external fragmentation control, and the smaller the page size the most fine-grain the control over all kinds of memory fragmentation. This property has also gained attention the past years for security reasons, in terms of non-predictable physical memory allocations [217]. There is an opposing force, however, to this. The larger the contiguity in mappings, the more opportunity there is for address translation acceleration. In this section we discuss how the latter drives the state-of-practice and state-of-the-art virtual memory designs in the big-memory era, where workloads frequently suffer from costly TLB misses [47, 101, 102, 155, 171, 172, 176, 177, 178, 194, 199] and the 4KB page size has likely aged to be too small.

**Huge Pages.** Huge pages are what they sound like, larger blocks of virtual and physical memory, properly aligned and mapped, with uniform protection (Figure 3.4b). They increase the TLB reach and decrease the page walk latency. For example, with 2MB pages a single TLB entry can translate 2MB of memory and the PD (3d) level of the page table tree becomes the last level of the tree. The PD level now stores the physical translation of the 2MB virtual page and the




(a) Small Pages (e.g. 4KB) (b) Huge Pages (e.g 2MB) (c) Larger-than-a-page contiguous

Figure 3.4: Contiguity in Mappings and Address Translation.

page walk involves only 3 memory references. Their efficiency [157] has made them the state-of-practice mitigation technique for address translation overheads and modern operating systems support them transparently (e.g., THP [19]). However, 2MB pages still fail to eliminate translation overheads for big-memory irregular workloads – as we will show and discuss in Chapter 4.

Huge page sizes larger than 2MB can push the translation performance barrier further away. In x86-64, though, the out-of-the-box eligible sizes are 1GB and 512GB due to the page table layout. However, such huge gaps bear challenges [47]. First, paged translation requires alignment and large aligned free blocks quickly become scarce in long running systems [227]. Moreover, transparent management is not straightforward. For example, intermediate sized mappings will either use larger pages wasting physical resources (internal fragmentation and bloat) or will use multiple smaller ones suffering from translation penalties. In fact, all considerations around 2MB management, including page fault tail latency (which we will discuss in a later section), fairness, and NUMA placement [143, 158, 164, 166, 226], manifest more severely as the page size increases.

**Larger-than-a-page contiguous mappings.** To come around some of paging's increasing page size inefficiencies, primarily its alignment and fixed size restrictions, state-of-the-art proposes breaking the correlation between the mapping and the translation granularity. Multiple proposals [83,130,172,176,177,202] preserve smaller page sizes in mappings and increase the TLB reach by leveraging larger-than-a-page contiguous mappings, i.e., contiguous virtual pages mapped to contiguous physical pages (Figure 3.4c). The most common way to exploit this linearity is to cache such mappings as a single translation entry, and the most important advantage of this strategy is that a single entry can potentially have a varying size and thus scale TLB reach more easily. TLB coalescing [83, 172, 176, 177] compacts such mappings to a single translation entry to slightly modified commodity TLBs. But to comply with current TLB design and indexing, it either supports a limited number of small coalescing factors (e.g. [177]) or caches mappings with certain alignment restrictions (e.g. [172]). In Chapter 4 we show how these properties limit the method's efficacy. The most flexible alternative representation of larger-than-a-page contiguous mappings is that of range translations (Figure 3.5) introduced by Redundant Memory Mappings

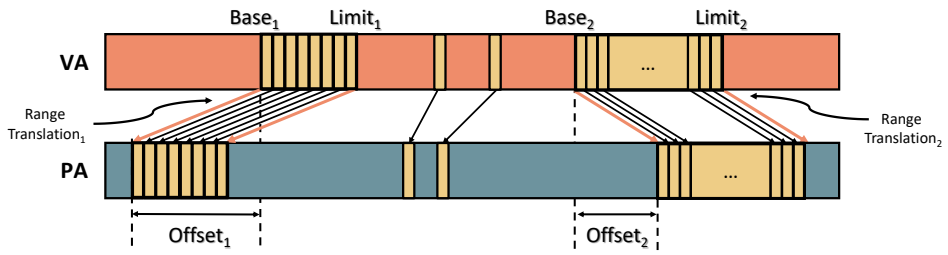
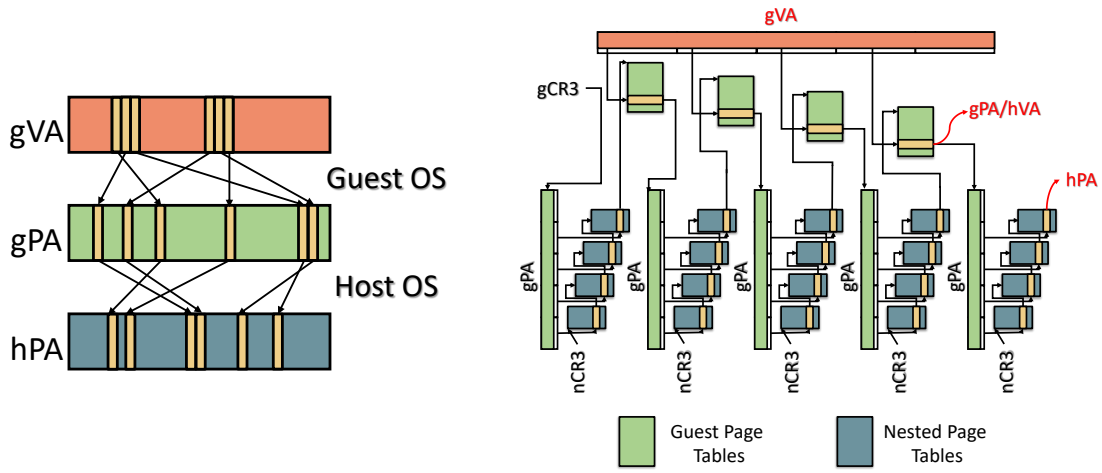



Figure 3.5: Range Translations.

(RMM) [130]. Given that in this thesis we examine extensions of the RMM design to support virtualization, we provide a more detailed overview of the initial proposal.

**Range Translations.** Redundant Memory Mappings (RMM) [130] extend the Direct Segment [47] proposal that initially revived and re-purposed segmentation for a primary region on a process address space. This entire line of research [47, 101, 130] draws on the key observation that most memory-intensive workloads apply unified protection over large virtual ranges in their space and most of their footprint resides always in memory. Thus coarse-grain virtual-to-physical translation representations do not break such applications functionality. To avoid the cumbersome management of a single monolithic physical segment per process, RMM introduces architectural support for *range translations*. In more detail, the authors define a subset of a process's pages that are virtually and physically contiguous as a *range translation* (Figure 3.5). Each process can have multiple range translation entities and each is represented by BASE, LIMIT, and OFFSET values that can translate any address falling into its boundaries. The two key advantages of RMM are that i) range translations can be of unlimited size and of no special virtual or physical alignment and ii) paging is still enabled in their regions. This means that the virtual area of each range translation remains mapped by pages and the entire architectural support for RMM is redundant to that of paging. Later on this chapter we discuss how RMM generates such contiguous mappings in the OS, we now focus on the method's architectural support.

RMM maintains a range table per process, managed by the OS and accessible by HW, similar to page tables. Each range table entry stores the BASE, LIMIT and OFFSET values of a range translation along with its protection (e.g read/write). The tables are implemented as B-trees since no fixed indexing scheme can be applied due to the un-aligned and arbitrary size range properties. A fully-associative Range TLB caches the most recently used range translations. On a L1 page TLB miss the Range TLB is looked-up in parallel with the L2 page TLB. On a Range TLB hit, hardware logic generates the missing physical address translation for the target virtual address falling into the cached range translation boundaries. On a Range TLB miss, a regular page walk is triggered to fetch the missing page translation on the L2 page TLB so that the processor can proceed with execution. In parallel and on the background a more expensive range table walk




(a) Three address spaces (b) Nested Page Walk

Figure 3.6: Address Translation in Virtualized Execution

is also performed in hardware to update the Range TLB with the missing range translation. The method efficacy depends on high Range TLB hit ratios and small range table depths and for that reason it provides optimizations that filter out single page or small contiguous page mappings.

RMM was originally proposed for native execution, and despite its efficiency, in this thesis we show that extending it for virtual machines requires extra complex architectural support.

> In Chapter 4 we discuss architectural extensions to RMM required to support the challenging setup of virtualization, introduced in the next paragraph.

### 3.2.3 Support for virtual machines

A virtual machine is the virtualization of a computing system running on physical, "real-world", hardware usually referred to as the "host machine". The virtual machine –its virtual devices and the OS instance that manages them– are generally referred to as the "guest". To run multiple OS instances (guests) on top of shared physical hardware, virtualization introduces a layer of indirection called the VMM or hypervisor. Under this scope, virtualizing memory introduces an extra layer of indirection between the guest applications virtual address spaces and the underlying physical memory, the guest physical address space (Figure 3.6a). In other words, the guest physical address space is what the guest OS understands as physical memory and it is itself independently mapped to real hardware by the hypervisor. So in virtualized execution we have:

- the guest virtual address space (gVA)
- the guest physical address space (gPA) and
- the host physical (hPA) address space

and thus on every memory operation an extra translation step is required to retrieve the physical location where data reside.

In this thesis we focus on hardware-assisted memory virtualization, a technique also known as nested paging [51]. With this method, the guest OS maintains the guest page tables (gPT) holding gVA→gPA mappings and the hypervisor maintains the nested page tables (nPT) that hold gPA→hPA mappings independently. Extended hardware walkers traverse both tables in a nested fashion to retrieve 2D translations, gVA→hPA, and cache them on the TLBs (Figure 3.6b). The two-dimensional page walk is required because each intermediate guest physical address must be translated through the hypervisor page tables. The nesting multiplies page walk overheads versus native execution, i.e. figure 3.6b shows how memory references grow from a native 4 to a virtualized 24 references with 4-level paging [99].

TLB miss overheads can throttle memory virtualization performance and have proven difficult to eliminate [101,199]. Also the two-dimensionality of the problem imposes challenges in the creation and exploitation of effective 2D larger-than-a-page contiguous mappings, introduced in the previous paragraph, to accelerate translation.

In Chapter 4 we study address translation in virtualized execution and propose hardware and software techniques to exploit larger-than-a-page contiguous mappings.

### 3.2.4 Address Translation Coherence

TLBs must remain coherent to processes page tables throughout their execution. The OS may update the process page tables during various virtual memory operations that change its address space state. In example, during unmap operations the OS destroys the PTEs of the mapping that is torn-down. These changes have to be propagated to the TLBs of the various cores to invalidate any cached copies of the modified PTEs. Otherwise, erroneous or insecure execution may take place if the application access a stale address mapping. Processors based on x86_64 architecture do not support hardware TLB coherence – i.e. automatic invalidation of the TLBs upon PTE modifications. The OS is in charge of the invalidation instead, via a process commonly known as TLB shootdown. Figure 3.7, originally found in [39], shows the steps that lead to and implement a shootdown in a multi-core CPU. First accesses on a mapping lead to the caching of its page tables entry on the private TLBs of the cores that issued the memory accesses (1,2). Then the OS updates the entry, e.g. it clears it due to an unmap operation (3). It sends out an invalidation request to all participating cores via an inter-processor interrupt (IPI) and waits for an acknowledgment that each core has performed its local TLB invalidation (4-5). Cores invalidate their TLBs in an interrupt handling routine and at the end send back the acknowledgment.

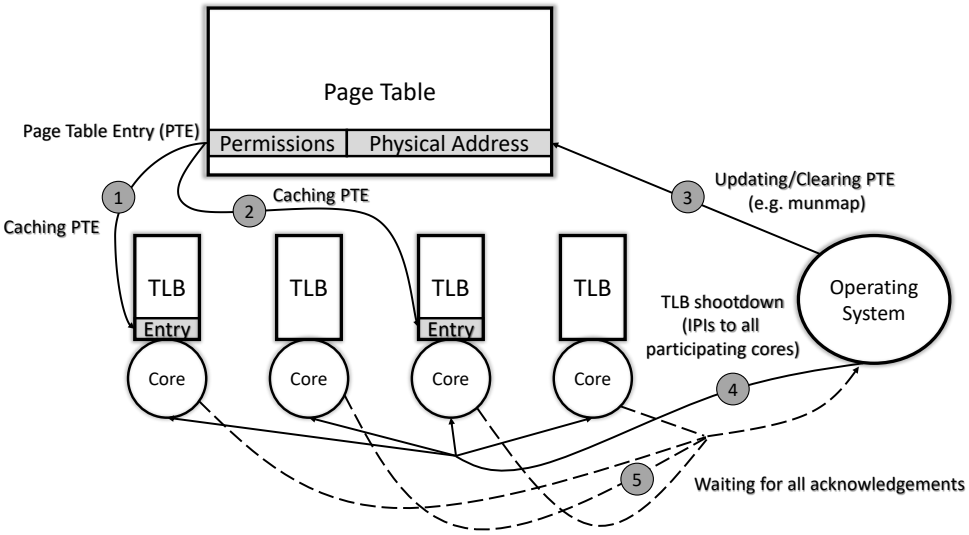



Figure 3.7: TLB shootdowns [39].

TLB shootdowns are notoriously expensive operations, inherently non-scalable as they are based on IPIs, and their cost increases with the number of cores [34, 35, 39, 141, 168, 170, 209].

In Chapter 5 we study translation coherence impact on the scalability of the virtual memory interface for direct access to persistent data.

## 3.3 The role of the Operating System

The OS is in charge of managing processes virtual address spaces, mapping them to physical resources and (de)allocating physical memory as needed. In this section we go briefly through some of its fundamental operations and give some details on the Linux implementation – as it is the OS we extend in our proposals.

### 3.3.1 Virtual Address Space management

As discussed in the previous section, the OS provides every process with a private linear set of virtual addresses, e.g. the range $[0,2^{48}]$ in x86_64 and 4-level paging. This address space contains the process's code, data, and stack. When a process is created (forked) a few virtual memory regions are instantly populated, e.g. the process heap and stack or the region mapping the program binary file. The processes can then use system calls to request the extension of a virtual memory region or the population (in other words allocation) of a new one to store anonymous data (e.g.

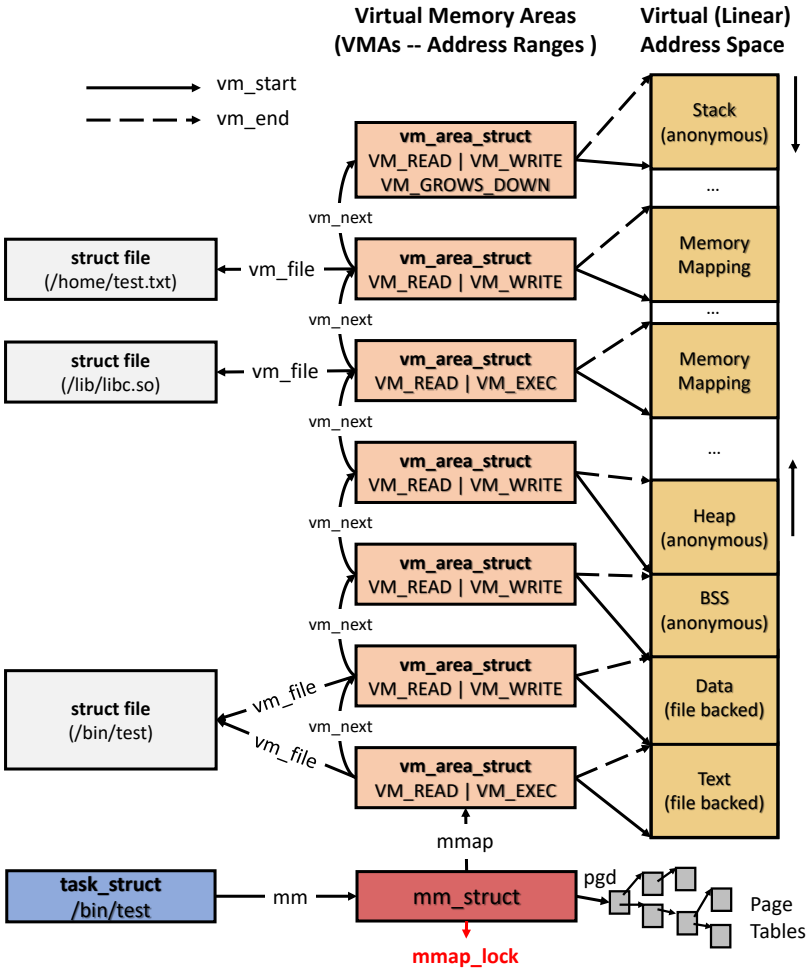
| System Call | Description |
|---|---|
| brk() | Changes the heap size of a process |
| execve() | Loads a new executable file, thus changing the process address space |
| exit() | Terminates the current process and destroys its address space |
| fork() | Creates a new process, and thus a new address space |
| mmap() | Creates a memory mapping (file or anonymous), enlarging the address space |
| mremap() | Expands or shrinks a memory region |
| mprotect() | Changes the access protections of address space range |
| madvise() | Gives advice to the kernel about address space range access (e.g. pattern) |
| msync() | Flushes changes made to a file mapped to memory back to storage |
| mlock() | Locks part or all of the process address space in DRAM |
| mbind() | Sets NUMA memory policy for part or all of the process address space |
| munmap() | Destroys a memory mapping (file or anonymous), contracting the process address space |
| shmat() | Attaches a shared memory region |
| shmdt() | Detached a shared memory region |

Figure 3.8: System calls related to virtual memory region creation, management and deletion [58].

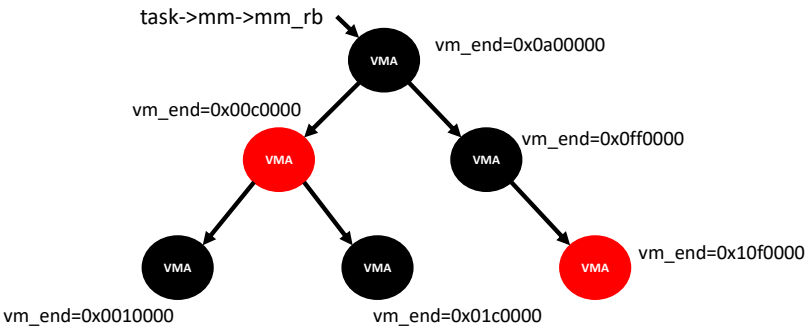
heap) or map and access files (discussed in Section 3.3.4). Figure 3.8 summarizes the most common system calls in Linux related to virtual address space management. In example, the brk() system call extends the process heap and the mmap() system call allocates a new set of virtual addresses to map both files or anonymous memory.

Figure 3.9a gives an example of a process address space layout in Linux and some of the core OS data structures involved in its description. At the bottom of the address space it is the text region, which holds the instructions of the program the process executes. Then it is the BSS and the data segments containing static (global) variables in C and the heap that dynamically expands. At the top of the address space we find the stack that grows down and in between we have various sparsely located mappings of files or anonymous memory. In the past, the initial virtual addresses of the segments had the same value for virtually all processes. Now, randomizing the address space [104] is preferred for security reasons and linux randomizes the stack, the file mapping segment, and the heap, adding offset to their start address. Virtual memory operations that (de)allocate virtual addresses either shrink/expand an existing mapping or destroy/create one.

Each mapping in the process address space is commonly referred to as virtual memory area (VMA) and in linux it is represented by the vm_area_struct that stores among others its protection properties (e.g. read/write/execute) and the file it maps, if the VMA corresponds to a file mapping. All VMAs are recorded under the central struct that represents the entire address space of the process, the mm_struct. VMAs are indexed by multiple data structures (e.g. linked lists) but a red-black tree is also used for fast O(log n) search (Figure 3.9b). The OS uses this tree on every virtual memory operation. The mm_struct stores as well the page table tree of the process, discussed in the previous section.




(a) Virtual Memory Areas represent populated virtual ranges.




(b) The VMA red-black tree.

Figure 3.9: Virtual Address Space in Linux [58, 144]

**Virtual Memory Locks.** Linux protects the entire address space of a process under a central lock, the mmap semaphore, that all virtual memory operations hold as readers or writers. In example, when virtual address ranges are (de)allocated or expanded/shrinked during system calls, when their protection changes or when the process page tables are referred or updated –e.g. during page fault traps discussed next– the OS locks the entire address space of the process to perform maintenance or scan it safely. This central locking is a well-known scalability bottleneck that limits virtual memory performance in the multi-core era [48, 59, 60, 70, 71, 80, 140]

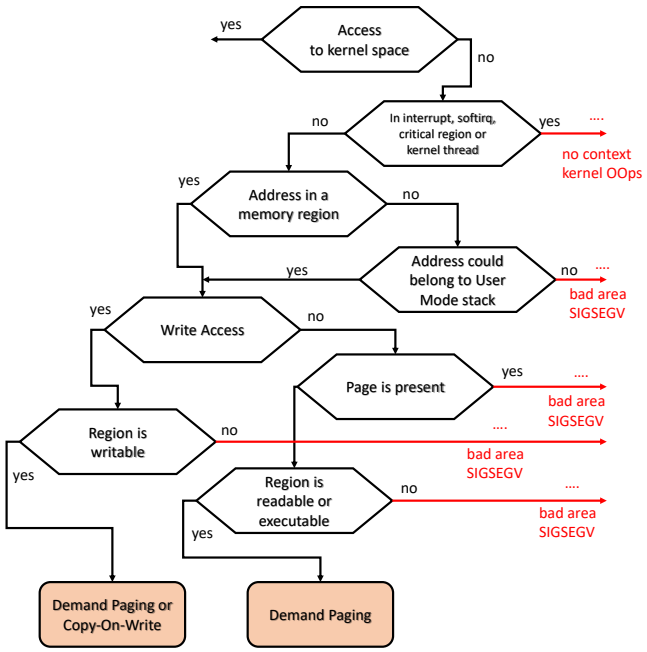
In Chapter 4 we extend the vm_area_struct to store minimal metadata for the page fault handler, discussed next, to perform contiguous allocations during demand paging. Also in Chapter 5 we revisit VMA recording data structures and virtual memory locking under the scope of direct access to persistent data.

### 3.3.2 The Page Fault Handler and Demand Paging I

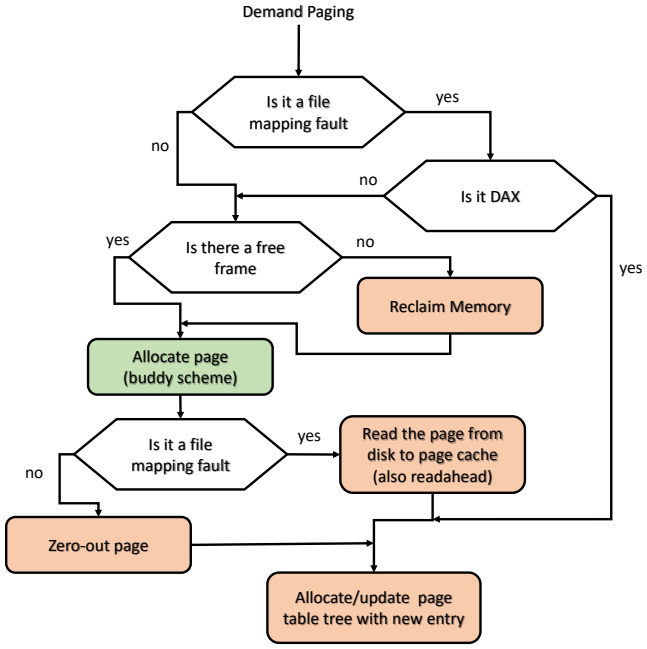
The Operating System, apart from allocating and managing active areas in a process address space, is also in charge of (i) mapping them to physical resources and (ii) enforcing memory protection in orchestration with the MMU hardware. Both fundamental properties are implemented via a special exception – known as the page fault.

**Memory Protection.** As discussed in earlier sections one of virtual memory's important functionality is the enforcement of memory protection, i.e. forbidding write or execute access over sets of virtual addresses or forbidding access altogether to virtual addresses that the process has never allocated. Such illegal accesses are detected by the MMU hardware that upon memory reference fails to find a valid translation entry or an entry with valid permissions for the target virtual address on the TLBs or the page tables. At that point the hardware throws and exception –page fault– and traps into the OS that executes the exception handler. The OS first performs various checks to detect if the access is really illegal. Figure 3.10a shows some of the checks of a user-mode page fault – when the memory access happens from user-space. If the access is indeed illegal, the OS commonly sends a SIGSEGV signal on the process, the known segmentation fault. But there are some scenarios (marked in beiz) that the access is found to be legal and just valid page table entries do not yet exist. We go through the most common scenario that is also tightly coupled with the way a modern OS manages physical memory, demand paging.

**Lazy mapping population a.k.a demand paging.** Modern OS do not allocate any physical resources upon user new mapping requests, at least not by default. In example, when brk() or mmap() system calls are issued –introduced in Figure 3.8– the OS allocates only a virtual address




(a) Part I: checks if it is a valid user-space access [58].




(b) Part II: demand paging.

Figure 3.10: The Page Fault Handler [58].

range for the new mapping –the virtual memory area creation– and does not allocate any physical memory for it. Instead the OS maps processes' virtual addresses to physical resources in a per-page basis and when the process accesses each virtual page for the first time – a technique named demand paging. Demand paging minimizes tail latency as no bulk physical memory allocations are performed, and most importantly enables agile physical memory management. Workloads tend to access parts of their address spaces in bursts and with temporal and spatial locality. With demand paging the OS can micro-manage processes footprints, keeping in-memory only the pages of the active working-set of the workload under its various execution phases.

When page faults are triggered by cold accesses on valid pages, the OS detects that and performs the necessary steps to set up a valid page mapping. Figure 3.10b shows the basic steps of demand paging when the faulting address refers to a valid anonymous (heap) or file mapping.

If the faulting virtual page refers to anonymous memory, the page fault handler allocates a physical page and sets up the page table entry to store the physical frame number – initializing the page mapping. In the next paragraph, we discuss in more detail the OS's mechanism to allocate physical pages. If no available free physical page can be found at the time of the fault the OS may have to reclaim physical memory, i.e. throw cold pages to disk to create free space. During anonymous faults the most common expensive operations are the zero-out of the newly allocated pages and the set-up of the page tables (marked in beiz). Some of these overheads increase with the size of the page and various works try to come around sky-rocketing page fault latencies in modern systems [143, 153, 158, 164, 179].

We discuss page faults for file mappings in the following Section 3.3.4.2.

### 3.3.3 Physical Memory Allocation and Contiguity

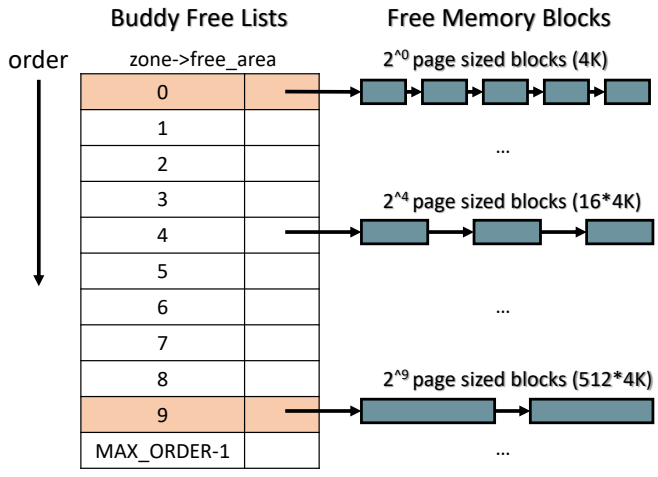


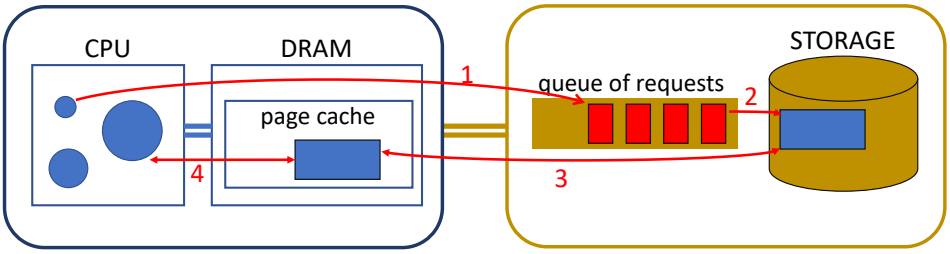

Figure 3.11: The buddy allocator free block lists [58].

During page faults the OS commonly allocates physical pages to map the faulting virtual pages. The buddy allocator is the core mechanism that most OS use to manage physical memory (Figure 3.11). It maintains [0, MAX_ORDER] lists, each populated by free aligned blocks of $2^{order}$ pages. This power-of-two logic supports fast memory coalescing and simplifies the management of free blocks. Memory allocation requests are served by order 0 (4K pages) or order 9 (2M pages [19]). If the lists are empty, larger blocks are split recursively. When memory is freed, the buddy allocator coalesces properly aligned free neighboring blocks (buddies) of the same order recursively, to control external fragmentation.

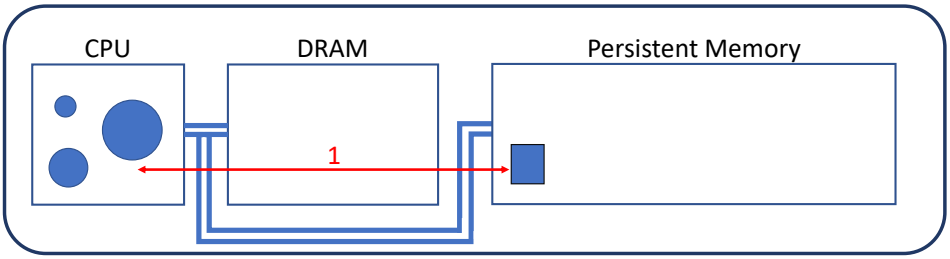
This scheme is tailored for paged virtual memory and does not facilitate the creation of larger-than-a-page contiguous mappings, discussed in Section 3.2.2, for novel address translation hardware. With demand paging and the buddy lists, the maximum contiguity generated under control is within the fixed size boundaries of pages (e.g. small or huge). At the same time the maximum contiguous free memory blocks tracked by the system have sizes of $2^{MAX\_ORDER}$ and any greater contiguity falls under the radar.

**Novel OS support for larger-than-a-page contiguous mappings.** To generate contiguous mappings beyond the page size limit most prior works employ some pre-allocation technique [47, 130] or page migrations [227]. Direct Segments [47] allocate one big physical segment block when the process is spawned while RMM [130] apply a more flexible scheme that pre-allocates memory at user memory allocation request time. In more detail, during mmap() or brk() system calls RMM synchronously allocates physical memory in large contiguous blocks. The authors name this technique *eager paging* as it breaks the on-demand allocation property of modern OS. To track large free memory blocks on the system, RMM increases the MAX_ORDER OS constant parameter, increasing the tail-latency of the coalescing operations of the buddy-allocator and remaining highly sensitive to external fragmentation (as we will show in Chapter 4). Translation Ranger [227] on the other hand does not work at the allocation path at all and employs page migrations to coalesce processes footprints and create virtual-to-physical contiguous page mappings. However, the technique pays migration overheads even when abundant contiguity can be found on the system at allocation time and its mechanisms are cumbersome in terms of multi-program support, i.e. multiple processes running concurrently on the same system. In general, in this thesis we find that there is a lack of lightweight and non-intrusive support for large contiguous mappings creation.

In Chapter 4 we study lightweight-memory management OS support for contiguous mappings and propose a technique compatible to demand paging.




(a) Block-level access [63].




(b) Byte-level access [63].

Figure 3.12: Block-level vs byte-level access.

### 3.3.4 Virtual Memory as a File Interface

As discussed throughout this section, virtual memory can also be used to access files. In example, the mmap() system call (Figure 3.8) can be used to allocate a virtual address range (a virtual memory area–VMA) that will map a file into memory. This means that after a successful call, all process's accesses in the newly allocated virtual range must result in accesses on the file's data at the corresponding offset. The OS implements this functionality, employing both its virtual memory and file system layers, that is commonly referred to as the file mapping interface.

**Block level vs Byte level Storage access, the case of Persistent Memory.** Most traditional IO devices support block-level access or are connected to inter-connection links that support block-level addressing. Thus to enable CPU access to file data the latter must be copied from storage to DRAM, as CPU performs accesses only at byte-granularity. Figure 3.12a shows such a set-up. For most common IO devices, i.e. not ultra low-latency SSDs, the OS maintains a DRAM cache of the most recently used file pages –the page cache– to which it also applies prefetching (readahead) techniques to hide IO latency and utilize better IO device bandwidth. With the file mapping interface, as discussed later, the OS maps page cache physical frames –caching copies of file data– to processes virtual address spaces.

However, recent storage technologies enable direct CPU access to fast byte-addressable storage, such as *persistent memory* [119, 186] and/or the CXL link [85]. Persistent memory (PMem)

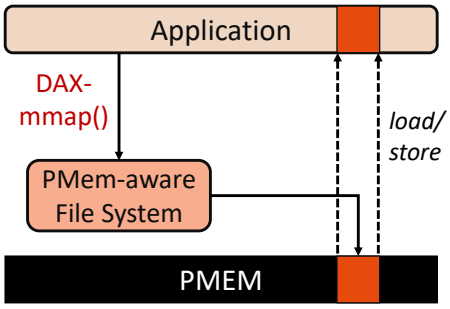



Figure 3.13: DAX-mmap

is connected to the system via the memory bus, like DRAM, and is accessible via CPU load and store instructions. This byte-level access is abstractly depicted in Figure 3.12b. The Direct Access (DAX) file interface [4] and its file mappings can map persistent memory pages directly to processes virtual address spaces, forming the shortest available path to storage.

#### 3.3.4.1 DAX file mappings and direct access to persistent data

DAX code in the Linux kernel bypasses the page cache file data buffering – removes the extra copy by performing reads and writes directly to the storage device. For file mappings, the storage device is mapped directly to user-space [4] and Figure 3.13 shows such a DAX mapping. During DAX-mmap operations the OS allocates a new virtual memory area for the file mapping, similar to default mmap, but the newly created VMA is marked as DAX. During page faults within its range the OS sets-up page table entries that directly store persistent memory physical addresses. The CPU MMU hardware is then in charge of translating virtual addresses into PMem physical addresses, as with DRAM, during load and store CPU instructions. We discuss page faults for file mappings in the next paragraph. File-systems must support DAX to enable such file mapping operations. Currently 3 mainline filesystems support it, ext2, ext4 and xfs, but as we will discuss next, there are multiple novel PMem-aware file systems.

#### 3.3.4.2 The Page Fault Handler and Demand Paging II

Demand paging, introduced in Section 3.3.2, is applied by the OS on file mappings as well. Figure 3.10b shows the path of page faults triggered by cold accesses on virtual pages that map files. The handler differs if the mapping is for direct or DRAM-buffered access to storage, as expected.

For buffered access the OS allocates a physical page, invokes the file system to retrieve file block storage location and triggers IO operations to bring-in the requested data from storage to the DRAM page. It then sets-up the page table entry to store the number of the physical frame that now caches a copy of the persistent data. As discussed, the OS commonly maintains a DRAM cache of the most recently used file pages –the page cache– to which it also applies prefetching

techniques. For page faults that hit on the cache no IO is performed. In any case, the page table entries for file mappings are set-up to map process virtual pages to page cache physical page frames. For buffered mappings, it is the page cache maintenance that has been identified as one of the most important sources of mapping overhead [168, 170].

For direct access the OS does not have to allocate any physical memory at all, as data are already stored *in a byte-addressable medium.* The OS simply invokes the PMem-aware file system to retrieve the storage physical location were data reside and sets the page table entry for the faulting virtual page to store directly the retrieved PMem physical page frame. For direct access, it is the file system indexing [149, 162] and the page table set-up [124, 149, 220] that dominate fault overheads and can commonly throttle performance.

In Chapter 5 we study page fault overheads for direct access to persistent data (DAX).

#### 3.3.4.3 PMem-aware kernel file systems

With PMem, storage accesses can be cheaper than OS invocations, so reducing the OS overheads is a strong requirement. Multiple works attempt to reduce the PMem software stack overheads by extending file systems to support DAX [67,216], and allow direct storage access, or by designing file systems entirely from scratch as PMem-aware (e.g. [74, 90, 124, 221, 222, 230]) and also optimize metadata operation. The latter commonly use the byte-addressability of the medium to provide stronger reliability guarantees and/or deploy high-performance logging and journaling mechanisms. In Chapter 5 we will consider both types of file systems.

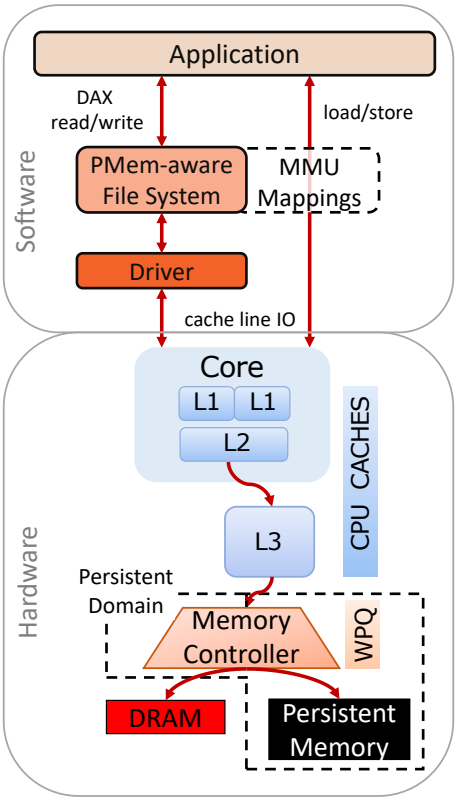



Figure 3.14: PMem Access: Software and Hardware layers [50].

#### 3.3.4.4 Putting it all together

Figure 3.14 puts together both the hardware and software layers of persistent memory access. If we focus on file MMU mappings, the OS maps directly processes virtual pages to persistent memory physical frames via DAX-mmap and page table set-ups. After that point any load/store instruction over the virtual range of the mapping is translated by the MMU hardware to storage locations. The memory subsystem fetches persistent data from storage to the cache-hierarchy during access and at cache line granularities, as with DRAM. In a sense,

with DAX and PMem we have cache line IO operations. One interesting aspect of this is durability management and what we call the persistent domain. As shown in the picture, to persist data with PMem one only needs to evict the corresponding dirty cache lines storing them from the CPU cache hierarchy.

#### 3.3.4.5 User-space durability management

Without DAX, syncing a memory-mapped file's data to storage (e.g. via msync()) writes back data from the page cache to storage in the granularity of pages. With DAX it is necessary to only write back dirty CPU cache lines to persist data. This creates the opportunity for applications to manage data persistency directly from user-space and at fine granularities. The application may use non-temporal stores or cache line flush instructions (e.g., the clwb instruction combined with the sfence instruction) to persist file data in the granularity of bytes [122, 241]. In fact, this approach is recommended [220] as it can significantly outperform kernel msync() operations. To manage durability in user-space safely, the applications needs to use DAX-mmap() with the MAP_SYNC( [78]) interface. The interface guarantees metadata durability and consistency for file system integrity.

**FLEX [220].** One way to harvest PMem benefits is re-designing applications to use persistent memory objects (e.g., data structures [38, 110, 160, 232]) managed by user-space persistent memory programming libraries [20, 73, 212]. However, designing PMem-aware applications with persistent data structures requires substantial programming effort [5, 220]. Xu et al. propose FLEX, a simpler programming approach to attain PMem benefits.FLEX emulates file system operations like read()/write() in user-space by mapping a file during open() and using user-space memcpy() instead of read(), and non-temporal store instructions instead of write().

#### 3.3.4.6 User-space file systems

In the same direction, multiple works [66, 87, 125, 142, 149, 211] exploit PMem direct access via new file system (FS) designs with user-space components. Performing (meta)data operations directly from user-space avoids syscall overheads, but comes with two inherent challenges: (i) (meta)data security and (ii) concurrent file sharing. Mapping parts [66, 125, 142] or the entire FS image [149, 211] to user-space for large time frames opens a window for intentional attacks or unintentional errors (stray writes) that can leak data or corrupt the FS image [87]. Such FS must employ a mechanism to control this that may lead to scalability issues [142, 211]. Moreover, such FS commonly employ user-space data buffering to provide performance, e.g., for writes [125, 142, 211], making concurrent file sharing among processes difficult to support. In addition, many user-level FS do not support memory mapping [142, 211] at all, removing a high performance

interface. Kernel-space FS can be less performant but support seamlessly sharing and secure (meta)data operations. In this paper we focus on such well-tested mature FS targeting to improve the kernel's file mapping interface performance rather than bypass it.

CHAPTER 4

# Enhancing and Exploiting Contiguity for Fast Memory Virtualization

## 4.1 Overview

Page-based address translation overheads are alleviated by caching translations in Translation Look-aside Buffers (TLBs). However, the growing demand for physical memory is limiting the efficacy of TLBs, increasing the rate of costly TLB misses. To make things worse, the adoption of virtualized cloud infrastructure amplifies these overheads. The state-of-practice MMU virtualization technique (hardware-assisted nested paging [51]) requires two-dimensional address translation that increases the TLB miss penalty up to 6× compared to native execution. Looking forward, the continuing growth in physical memory sizes (e.g. via CXL [85] and the tiered-memory future of servers [148]) will soon require 5-level paging [22], further exacerbating the cost of TLB misses. Ideally, software and hardware support for address translation should minimize overheads, maintain memory availability with flexible allocations, avoid memory waste by minimizing fragmentation and preserve resource-saving mechanisms like demand paging and copy-on-write, under both native and virtualized systems.

In response, industry has increased the page size and translation hardware [21]. We show that 2MB huge pages still fail to cover the needs of irregular workloads, and nested paging magnifies the problem. Even though increasing the page size is prominent, larger pages increase internal fragmentation and restrict fine-grained memory management [143, 158, 164, 166, 226].

| | THP | DS [47] | RMM [130] | SpOT | vRMM |
|---|---|---|---|---|---|
| **OS memory manager** | demand paging | static pre-allocation | dynamic pre-allocation | Contiguity-Aware (CA) paging | |
| **translation representation** | pages | segments | ranges | pages | ranges |
| **virtualization support** | nested paging | dual direct segments [101] | – | nested paging | nested ranges |
| **micro-architecture** | – | – | – | speculation | – |

Table 4.1: Overview of our contributions with respect to state-of-practice (THP) and state-of-the-art (DS,RMM) approaches for reducing virtualization overheads.

Prior works [47, 83, 101, 130, 172, 176, 177, 202] have shown the potential of breaking the traditional page-based mapping between hardware translation and OS memory management. They usually exploit larger-than-a-page contiguous mappings [54] but fail to achieve the desired flexibility. For example, previous proposals rely on pre-allocation [101, 130], which suffers from external fragmentation, and is antagonistic to demand paging. Additionally, prior hardware schemes [130, 172] track the exact boundaries of contiguous mappings but do not support virtualized execution. Finally, prior approaches for increasing TLB reach have been limited by indexing and alignment requirements [83, 172, 176, 177, 202] reducing their efficacy.

In this thesis, we aim to reduce the address translation overhead, focusing primarily on virtualized execution; a set-up that magnifies the problem and hardens the solution. We generate and exploit larger-than-a-page contiguous mappings while avoiding pre-allocation to preserve the flexibility of existing paging-based mechanisms. We take a two-fold approach: (i) we introduce *contiguity-aware (CA) paging* which promotes contiguity as a first-class citizen in the OS memory manager, and (ii) we harvest the generated contiguity to accelerate address translation via via two orthogonal hardware designs: *vRMM* that works at the architecture level and *SpOT* that works at the micro-architecture level. Table4.1 summarizes the contributions of this work (in green). Note that, CA paging and SpOT can improve both native and virtualized execution.

*CA paging* enables the OS memory allocator to generate contiguous mappings beyond the page-size limit, using minimal per-process metadata. Specifically, CA paging allocates contiguous physical pages to map contiguous virtual memory regions of processes, working across page faults and on a best-effort basis. In this way, CA paging creates and extends contiguous mappings gradually while preserving the increased memory utilization and low tail latency of demand paging. CA paging can improve the performance of any hardware design that relies on contiguous mappings [83, 130, 172, 176, 177, 202] and can be used both in native and virtualized execution. In virtualized execution, it is used independently by the guest and the host OS. Our results show

that CA paging significantly boosts the creation of vast contiguous mappings, achieves performance similar to pre-allocation, and outperforms it in the presence of external fragmentation. Compared to asynchronous defragmentation [227], CA paging operates on the allocation path and generates contiguity instantly, increasing the opportunity to exploit contiguity and avoiding the cost of the post-allocation page migrations.

To better exploit the large generated contiguous mappings of CA paging, we examine two orthogonal hardware techniques. First we extend Redundant Memory Mappings [130] to support virtualization (*virtualized Redundant Memory Mappings (vRMM)*); focusing particularly on hardware assisted virtualization (nested paging). In vRMM, we define a 2D range translation as a contiguous mapping in both guest and host address spaces (guest_virtual_address(gVA)→guest_physical_address(gPA) and guest_physical_address(gPA)→host_physical_address(hPA)). Assuming RMM support, we introduce: (a) co-designed support for nested range tables that hold gPA →hPA range translations and (b) hardware extensions to traverse the guest and nested range tables to generate a 2D range translation. Unlike the native design [130] that uses pre-allocation on the software layer to generate ranges, we combine vRMM architectural support with CA paging and adjust range table construction. We show that vRMM and CA paging significantly reduce the number of nested page walks due to TLB misses, bringing translation overheads below 1%.

Driven by the increased architectural complexity of vRMM, we propose a second alternate technique that works entirely at the micro-architecture level: *Speculative Offset-based Address Translation (SpOT)*. We base our approach on the key observation that contiguity can be expressed simply through *offsets*, decoupling contiguous mappings from virtual boundary checks.

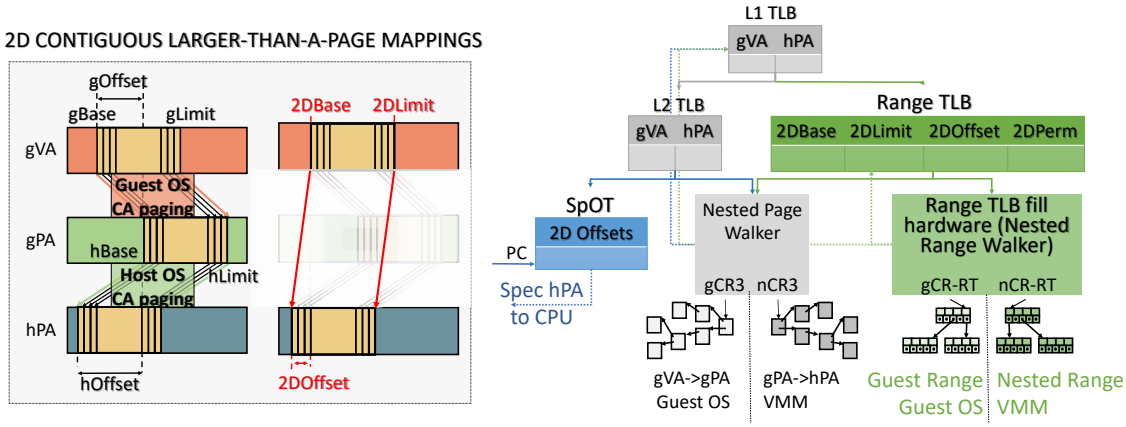
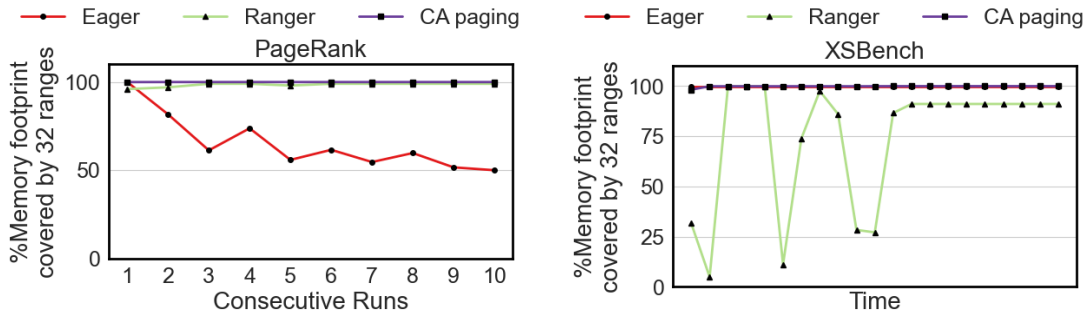



Figure 4.1: General overview of our proposal for virtualized execution. (a) CA paging is used by the guest and the host independently; seamlessly generating contiguity on both dimensions; the intersection forms the desired 2D contiguity. (b) vRMM (green) and SpOT (blue) are two orthogonal techniques to exploit the 2D contiguity and eliminate address translation overheads.




(a) Pre-allocation suffers from external fragmentation (b) Asynchronous defrag delays contiguity generation.

Figure 4.2: Trade-offs between pre-allocation (eager paging), asynchronous defragmentation (ranger), and CA paging. Pre-allocation suffers from external fragmentation and asynchronous defragmentation delays contiguity generation.

SpOT is a simple hardware mechanism on the TLB miss path that speculates the existence of large contiguous mappings to predict translations while performing verification page walks in the background. SpOT exploits the OS provided contiguity at the micro-architectural level, requiring minimal hardware support. In contrast to prior speculation designs [45, 178], SpOT predicts translations far beyond the huge page limit and is completely independent of virtual addressing and alignment. SpOT supports both native and virtualized systems. Our evaluation in a virtualized system shows that SpOT combined with CA paging reduces the address translation overhead of nested paging from ∼16.5% to ∼0.9% on average. SpOT performs close to prior schemes [101, 130] but without pre-allocation and complex virtualization extensions at the architecture level. While speculation introduces security concerns, SpOT uses the same generic mitigation mechanisms proposed for other speculation attacks [135, 224].

## 4.2 Software Technique: Contiguity-aware Paging

### 4.2.1 Key design concepts

All the state-of-the-art techniques that increase TLB reach require contiguity across pages [83, 130, 172, 176, 177, 202]. Prior proposals mostly lie on two extremes: they either rely on randomly generated contiguity by a vanilla OS [83, 172, 176, 177], or on brute-force pre-allocation schemes via reservation [47, 101] and eager paging [130]. The former clearly wastes opportunities for contiguity, while the latter abandons key OS mechanisms for flexible memory management (e.g. on-demand allocations, copy-on-write faults and sharing etc.). Pre-allocation is also sensitive to

external fragmentation. Translation Ranger [227] takes a different approach, and creates contiguity performing asynchronous and iterative memory defragmentation. A system daemon scans periodically process memory and migrates random physical pages to contiguous ones. Ranger is an effective contiguity mechanism, but migrations may delay to coalesce an application's footprint; migrations also penalize memory accesses latency and trigger costly TLB shootdowns [34, 141].

As a response to the limitations of the state-of-the-art, we propose CA paging; an extension to the core OS memory manager that operates at allocation time and creates large contiguous mappings while preserving the flexibility of demand paging. Figure 4.2 depicts the discussed prior work limits and depicts CA paging's impact. Figure 4.2a shows the percentage of the PageRank's memory footprint that is covered by the 32 largest contiguous mappings for 10 consecutive runs of the benchmark (Section 4.5 describes our methodology in detail). We observe that eager paging is sensitive to external fragmentation as the coverage drops progressively. CA paging sustains contiguity, harvesting unaligned physical contiguity in the system. Figure 4.2b shows the percentage of XSBench's memory footprint that is covered by the 32 largest contiguous mappings during the entire execution of the benchmark. We observe that Ranger's migrations delay to coalesce the application's footprint to contiguous memory. CA paging avoids unnecessary post-allocation migrations, harvesting the system's available contiguity at page fault time.

### 4.2.2 CA paging overview

*Contiguity-aware (CA) paging* relies on the existing demand paging mechanism, but instead of allocating physical pages randomly, it steers the allocation of physical pages to create contiguous mappings. We introduce lightweight mechanisms and policies to the core physical OS allocator to lazily create vast contiguous mappings across page faults. CA paging requires minimal metadata, i.e., an *Offset* per virtual memory area (VMA) and a system-wide *contiguity map*. To decide the placement of a VMA's pages, CA paging uses a next-fit policy. CA paging deals with external fragmentation, supports multithreaded applications, and serves all common page fault types. We design and prototype it in Linux for native and KVM for virtualized systems (nested paging).

**Demand Paging.** VMAs are contiguous virtual address ranges in a process's address space, not necessarily backed by physical memory, represented by the `vma struct` in Linux. Physical memory allocation happens on demand, when a virtual page of a VMA is touched for the first time. The OS core physical memory manager is a power-of-two buddy allocator, maintaining [0, MAX_ORDER] lists. Each list is populated by free aligned blocks of $2^{order}$ pages. Allocation requests are served by the first available block of order 0 (4KB) or order 9 (2MB pages [19]) lists. Demand paging enables flexible memory management, but its random page allocations inhibit the creation of large contiguous mappings.

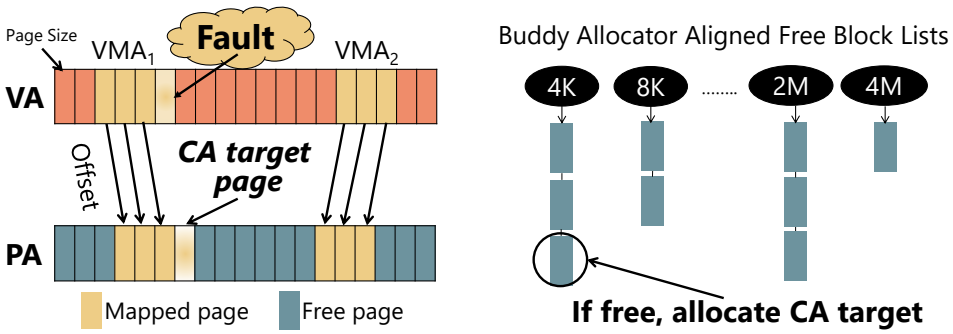



Figure 4.3: Overview of contiguity-aware paging.

**Basic mechanism.** CA paging leverages the unaligned and unlimited *Offset* representation of larger-than-a-page contiguous virtual-to-physical mappings. *Offset* is defined as the common $[virtual\_address - physical\_address]$ identifier for all pages belonging to the same mapping. CA paging tracks the *Offset* of the first page mapping created for each VMA (first page fault) and stores it as minimal metadata to the corresponding `vma struct`. On a future fault in the same VMA, CA paging uses the *Offset* to identify a target physical page for allocation. It examines the occupational status of the target page, and if free, allocates it, extending the current VMA mapping contiguity. Figure 4.3 (left) illustrates how CA paging exploits the notion of *Offset* to perform contiguous allocations.

CA paging examines the availability of the target page relying completely on existing OS metadata. In Linux, it retrieves a handle to the target page's structure using the system memory map (`mem_map`), indexed by page physical address. Dedicated attributes (`_mapcount`, `_count`) indicate if the target page is already in use. If the target page is free, it can be of the requested size or part of a larger block. In the latter case, CA paging splits the block using the default buddy allocator routine. In both cases, it retrieves the target page from buddy's lists (Figure 4.3 (right)). CA paging is independent to the order (size) of the allocation request and serves both 4KB and 2MB page faults.

**Contiguity Map.** The first page allocation for a VMA can greatly affect the later-on generated contiguity. To maximize contiguity, CA paging directs the mapping to a region of the physical memory where there is enough free contiguity. To achieve this, a map of the system's free contiguous space is necessary. In Linux, the maximum size of tracked free memory is limited by the MAX_ORDER attribute. Typically that equals to 11, and the allocator maintains up to 4MB aligned free blocks. Prior research [130] proposes increasing MAX_ORDER but that approach is sensitive to external fragmentation (Section 4.5.1.1).

We introduce the $contiguity\_map$, an indexing structure on top of the buddy allocator's MAX_ORDER list (Figure 4.4) to record unaligned contiguity at scales larger than the buddy heap. Each entry of the map represents a variable length sequence (cluster) of free MAX_ORDER blocks.

It stores the starting physical address and the total size of the cluster. Updates to the map are triggered by all insertions/deletions to the corresponding buddy list. To avoid search operations on every update, all physically indexed base blocks of a cluster point to their corresponding *contiguity_map* entry (re-purposing the `mapping` attribute of the `page struct`, not used when a page is free). We currently implement the map as a linked list sorted by physical address. Even if a tree could yield better performance, our evaluation shows that keeping the map up to date does not affect performance. A separate *contiguity_map* instance is maintained per NUMA node (`struct zone`), as the OS maintains a separate buddy instance per NUMA node.

### 4.2.3 CA paging Mechanism

**Placing the first page.** During the first page fault for a VMA, CA paging searches the *contiguity_map* for a free physical region that could fit it. Using the total VMA size as a key, it applies the next-fit placement policy; it searches for an available free block of the requested size, starting from where it left off the previous time. If no block larger or equal to the requested size is available, next-fit selects the largest found. CA paging allocates the first page of the selected region and sets the *Offset* attribute of the corresponding `vma struct`. Figure 4.5 visualizes the steps following the first fault in a VMA. Note that with CA paging, unlike the traditional segmentation case, a placement decision does not result in the allocation of the entire VMA. Instead, CA paging directs the forthcoming page faults of the same source VMA to the selected free block through the *Offset* attribute (best effort approach).

As CA paging does not allocate memory beyond the page size, competition for the same free blocks can arise when multiple faults by different processes or different VMAs of the same process trigger placement decisions. We opt for next-fit policy because it can defer such racing. The block that is selected to serve a placement request is the last one to be considered for the next request. To implement next-fit, we use a simple rover pointer over the *contiguity_map*.

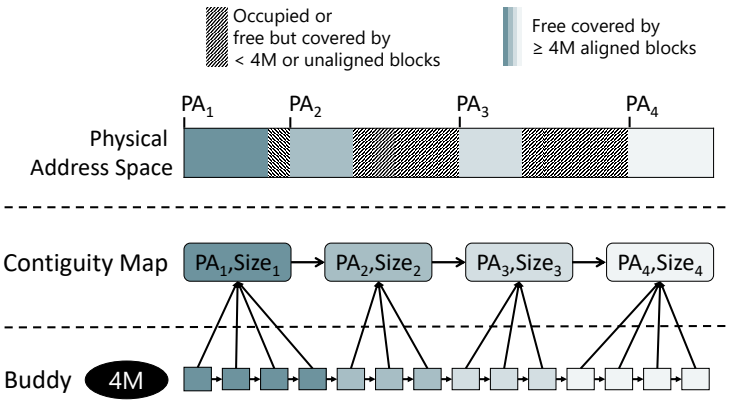



Figure 4.4: The *contiguity_map* of CA paging.

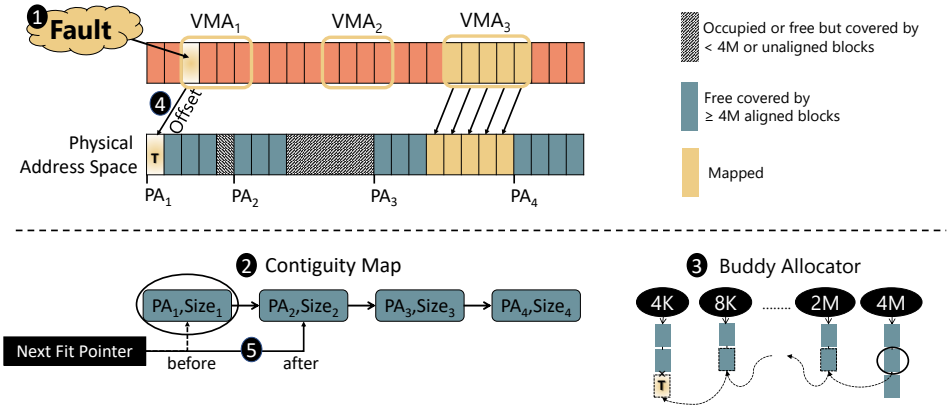



Figure 4.5: Placing the first page: 1) first page fault 2) *contiguity_map* search for free region, 3) allocation, 4) Offset update, 5) next-fit rover pointer update.

**Handling unsuccessful CA allocations.** A CA paging allocation target may be unavailable, either because the end of a free physical block is reached or some other running process has allocated the target page. Upon failure, if the fault is for a huge page, CA paging runs again the placement decision routine using as key the size of the remaining unmapped VMA region (see below sub-VMA placement) and tracks the new *Offset*. If, instead, the fault is for a 4KB page, CA falls back to the default arbitrary allocation mechanism and skips the *Offset* tracking. Making decisions on top of huge pages is more effective when targeting vast contiguous mappings. Also huge allocations amortize placement overhead with their costly large block zeroing operations.

**Dealing with external fragmentation.** If there is no available free block to fit an entire VMA (due to external fragmentation), CA paging makes multiple sub-VMA placement decisions and distributes the VMA to multiple smaller free physical blocks. The sub-placement decisions are triggered by unsuccessful allocation attempts (see above). In such scenarios CA paging performance depends on the fault pattern. To support multiple sub-VMA regions, we track multiple *Offsets* per VMA (instead of a single one) combined with the virtual address of the fault that created them. During a page fault, CA paging picks the *Offset* associated with the virtual address closest to the currently faulting. To control the search latency we track up to 64 *Offsets* per VMA and apply a FIFO policy.

To restrain fragmentation, we also apply a general optimization. We keep the MAX_ORDER buddy list sorted by physical address, using neighbors address computation and recursive logic for fast operation (similar to buddy coalescing). This sorting prevents small random (4KB) page allocations (e.g., the fallback path for CA failures) from using scattered physical pages and fragmenting large free contiguous blocks. As discussed in Section 4.5.1, CA paging acts as a prevention strategy with respect to external fragmentation. Keeping an application's footprint coalesced and isolated from other processes reduces the fragmentation of the physical address space.

**Avoiding multithreading pitfalls.** In multithreaded applications, different threads may fault concurrently triggering parallel allocations for different virtual addresses. We use spin locks to protect CA paging VMA metadata updates. Nevertheless, concurrent allocations inside the same VMA stress the *Offset* selection of CA paging. For example, if two different threads fault for virtual addresses of the same sub-VMA region and both fail (target physical pages occupied), they will both trigger re-placement decisions. Without proper handling, this race results in multiple *Offset* updates for the same region and unnecessarily stresses the next-fit mechanism.

To handle such cases and support concurrent faults, CA paging allows only the first thread that enters the allocation path to trigger a re-placement and *Offset* update in case of a failure (using an atomic flag per VMA). If another thread fails, there are two options: (i) fallback to the default allocation, or (ii) retry until replacement is allowed or the allocation succeeds. We choose the former to not penalize fault latency.

**Supported faults.** CA paging supports all anonymous and copy-on-write (4KB or 2MB) page faults, preserving demand paging. CA paging works also for the readahead allocations of the system's page cache, tracking an *Offset* attribute per file (`struct address_space`). Page cache allocations directly improve the performance of applications that use memory-mapped files. However, they can indirectly affect the translation performance of all processes. E.g. readahead allocations are usually interleaved with anonymous faults, as applications tend to read file data to populate heap structures. If randomly allocated they can penalize the process's contiguity. Moreover, page cache allocations tend to outlive processes, increasing the possibility of the file pages to be reused. If they are scattered, they tend to fragment the physical address space. CA paging allocates them contiguously restraining process interference and fragmentation.

#### 4.2.3.1 Virtualized execution

As CA paging is embedded in the OS memory management, it is applied in each dimension (guest/host) independently, elegantly enabling contiguous allocations to span the virtualization level and complying to nested paging. In the guest OS, it boosts the creation of gVA→gPA contiguous mappings across guest page faults ($1^{st}$ dimension) and in the host, the creation of gPA→hPA across nested faults ($2^{nd}$ dimension) (Figure 4.6 left).

In virtualized execution, a larger-than-a-page mapping is effectively 2D contiguous, only if contiguous in both dimensions (Figure 4.6 right). Thus, using independently CA paging in each dimension creates such mappings on a best-effort basis. On a freshly booted virtual machine (VM), all guest page faults lead to nested faults as the guest physical pages are not mapped to host physical memory. In this early phase, CA paging is triggered in both dimensions consecutively. However, with nested paging the mappings of the second dimension (gPA→hPA) remain

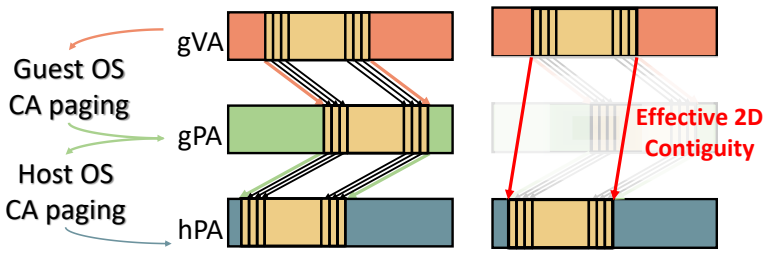



Figure 4.6: CA paging is applied independently in the guest and host OS, generating guest and host contiguous mappings. Their intersection forms the effective 2D contiguity exploited by the proposed hardware designs in Section 4.3

as long as the virtual machine is alive or until the host OS reclaims them. Thus, the $2^{nd}$ dimension contiguity persists as a VM ages, while the guest CA paging creates new $1^{st}$ dimension contiguous mappings for new applications running inside the VM. This leads to a less controlled generation of full 2D contiguous mappings, e.g., $1^{st}$ and $2^{nd}$ dimension mappings can be unaligned, smaller or larger with respect to each other (Figures 4.8,4.9). Our experiments indicate that CA though is still effective and creates significant 2D contiguity.

## 4.3 Hardware Techniques: vRMM and SpOT

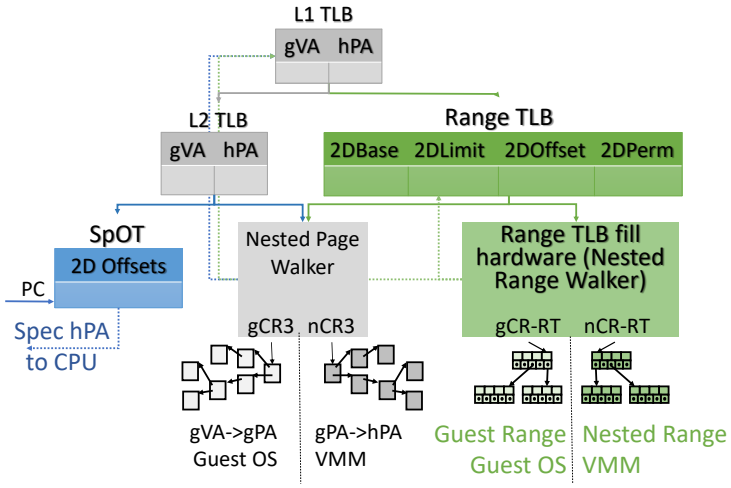



Figure 4.7: vRMM caches effective 2D contiguous mappings in a range TLB, looked-up in parallel with the L2-page TLB. It requires complex architectural support; virtualization extensions over native RMM (nested ranges and a nested range walker). SpOT works entirely on the micro-architecture level, caching only 2DOffsets and using them to predict address translations on the L2-TLB miss path. It feeds the predicted address to the CPU which continues execution on Speculative mode.

Any address translation scheme that leverages contiguous mappings [83, 130, 172, 176, 177, 202] benefits from CA paging. We target the challenging setup of virtualized execution, which most state-of-the-art proposals do not support. To exploit CA paging and reduce translation overheads in virtualized execution, we propose two orthogonal hardware schemes. vRMM is an architectural solution (OS/hypervisor involved); it increases the TLB reach by caching 2D range translations in virtualized execution. SpOT is a micro-architectural solution (no OS/hypervisor support); it hides TLB miss latency with speculative execution by caching only 2D Offsets and predicting the missing TLB entry. vRMM is secure and high-performant, but involves a very complex design. SpoT is comparable in terms of performance and requires minimum micro-architectural support but raises security concerns. We discuss both designs in detail in the next paragraphs. Figure 4.7 gives an overview of the proposed hardware techniques.

### 4.3.1 Virtualized Redundant Memory Mappings

We describe vRMM as a set of virtualization extensions to native RMM [130] for x86-64, assuming nested paging and a type-2 hypervisor (KVM).

**Native RMM overview.** RMM [130] was proposed to increase the TLB reach and reduce the number of page walks in native execution. RMM exploits contiguity through the notion of a ***range translation***: an arbitrarily large range of pages that are contiguously allocated in both virtual and physical address space. Ranges are represented as ***[base, offset, limit, permissions]***, stored in a per-process software (OS) managed range table and cached on a range TLB. One of the most significant advantages of RMM is that it is ***redundant to paging***. The traditional page table and TLB hierarchy remain the same and the range TLB is looked-up in parallel with the L2-page TLB. In case of a page and a range TLB miss, a hardware range walker traverses the range table and fetches range translations to the range TLB. To avoid range walks in miss a reserved bit is set on all page table entries of pages that belong to a range translation. Thus upon a page and a range TLB miss, first the page walker fetches the missing PTE and checks if the range bit is set. Then the range walker updates the range TLB in the background. A register holds the starting physical address of the range table (CR-RT); it is OS-maintained and part of each process's context (similar to cr3). Due to the unlimited, unaligned nature of range translations and of their representation, range table is a B-tree and the range TLB is a fully-associative structure.

#### 4.3.1.1 vRMM overview

To extend RMM and support virtualization we define 2 types (dimensions [101]) of ranges [1]: *guest* and *nested*. We name their intersections *full effective 2D* ranges (Figures 4.8 and 4.9). vRMM is *redundant to nested paging*.

[1]We use range as a short term for range translation.

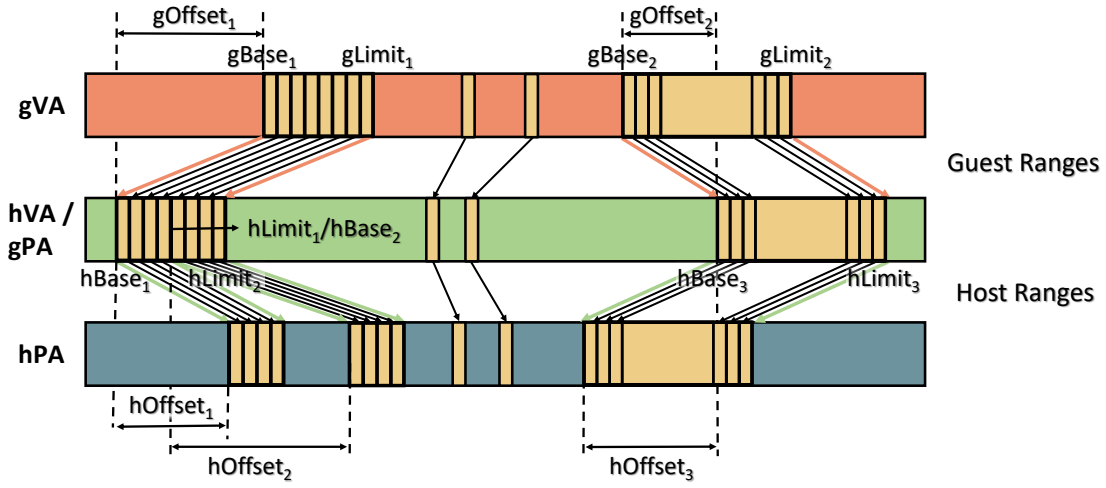



Figure 4.8: Unaligned arbitrarily-sized contiguous mappings (ranges) in virtualized execution.

- **Guest range (gVA→gPA)**: contiguously mapped pages, with uniform protection, per process running on the VM. Represented via [gBase, gOffset, gLimit, gPermissions]. Maintained by the guest OS in the guest range tables. (1st dimension)
- **Nested range (gPA→hPA)**: contiguously mapped pages per VM (same protection is guaranteed). Represented via [hBase, hOffset, hLimit, hPermissions]. Maintained by the hypervisor/VMM in the nested range tables. They are a subset of the host ranges (hVA→hPA) maintained by the host OS in the host range tables per host process [79]. Note that gPA and hVA address spaces are linearly related, in a manner controlled by the VMM. (2nd dimension)
- **Effective 2D range (gVA→hPA)**: contiguously mapped pages, with uniform protection, per process running on the VM. Represented via [2DBase, 2DOffset, 2DLimit, 2DPermissions]. Generated online, during range TLB misses, by the nested range walker. (intersection of the two dimensions)

**Range TLB.** It caches effective 2D gVA→hPA ranges. The structure and access remains intact with respect to native RMM: (i) it is accessed in parallel with the L2-page TLB, (ii) in case of a hit, the hardware generates the corresponding gVA→hPA page translation using the [2DBase, 2DOffset, 2DLimit] attributes of the 2D range translation and inserts it in the L1-page TLB, (iii) in case of a miss, the nested range walker generates the missing 2D range translation and fills it in the range TLB. The range TLB remains (inherently) fully associative.

**Range Tables.** In vRMM, guest range tables store the guest ranges (gVA→gPA) per process running inside the VM and are managed by the guest OS independently. They are redundant to guest page tables. Extended/nested range tables store the nested (host) ranges (gPA→hPA) per

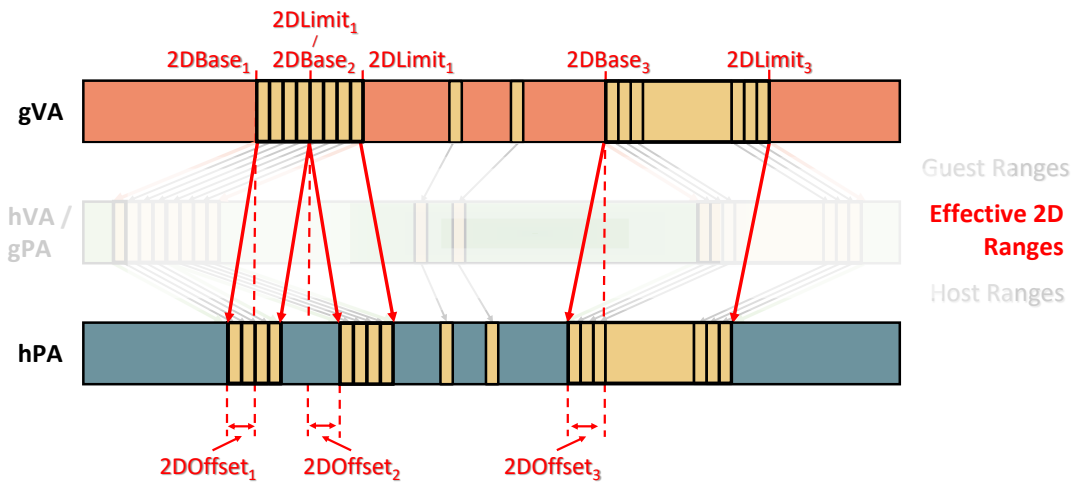



Figure 4.9: The effective full 2D contiguous mappings (ranges). vRMM caches the 2D ranges in a range TLB. SpOT caches only the 2D Offsets to predict translations.

VM and are managed by the hypervisor/VMM – e.g. KVM. A *nested range walker* uses the guest and nested range tables to dynamically generate full 2D gVA→hPA range translations and fetch them in the range TLB (discussed later).

All dimensions' ranges hold mappings of more than a *rmm_threshold* contiguous pages (32 in our experiments). A reserved bit in the last-level (PTE or PMD) guest/nested page table entry is set to indicate if the corresponding page is part of a guest/nested range.

**Range Table setup.** A notable difference in our design, compared to default RMM, is range table management. It derives from the different memory management technique used to generate ranges. The native proposal [130] pre-allocates physical memory blocks synchronously to virtual memory block allocations (malloc(), mmap()) (eager paging). This instant range creation enables the synchronous construction of range tables. Instead, we build on top of lazy and flexible demand paging, using CA paging.

CA paging gradually constructs ranges as the process touches its virtual addresses for the first time (across faults) (Section 4.2). Therefore, range tables must be updated/populated constantly. During a fault, CA paging examines the neighboring page table entries of the faulting page and if they are part of a range translation (reserved bit set), it extends the corresponding entry on the range table. If the neighbors are not part of a range, CA paging inspects the entries in *rmm_threshold* distance to detect a mapping of contiguous pages. If found, it promotes it to a range translation, creates a new range table entry and sets the corresponding page table bits.

In virtualized execution this happens independently in each dimension. The guest OS uses CA paging to generate guest ranges for the processes running inside the VM and setup their guest range tables. The host OS uses CA paging to generate host ranges for each VM and gradually

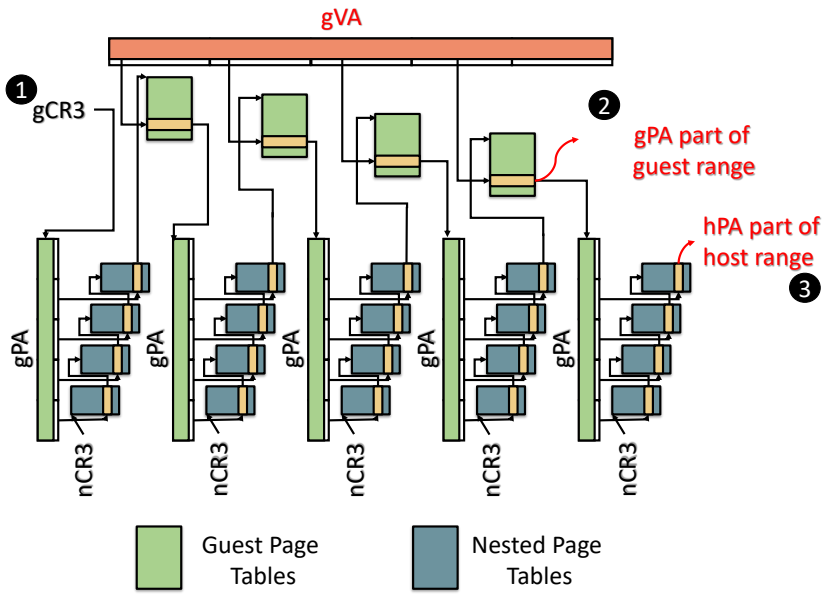



Figure 4.10: Range TLB miss in vRMM. Part1: The nested page walker identifies the missing page translation and if the page is part of a guest and host range.

setup the host range tables. The hypervisor (KVM) uses either nested faults (VM-exits) or the mechanism of MMU notifiers [79], already present for nested paging, to create or update nested range table entries based on the host range tables.

An update of a range translation in any dimension triggers a range TLB flush. However, in our experiments we find that flush operations do not affect range TLB performance as they occur rarely compared to the number of lookups.

#### 4.3.1.2 Nested Range Walk

On a range TLB miss, a nested range walk is required to fetch the missing 2D range translation, if it exists.

**Range TLB miss.** In vRMM, gCR-RT and nCR-RT registers hold the base gPA and hPA of the guest and nested range tables, and are part of a process's context, similar to nested paging's gCR3 and nCR3. Figures 4.10 and 4.11 visualise how a range TLB miss is served. Figure 4.10 shows the first part: (1) a standard nested page walk is triggered and retrieves the page's gVA→hPA translation and stores it in the L1-TLB. (2) During the walk, the walker checks if the reserved bit of the last level entry of the guest page table (gPTE/gPMD) is set - indicating the page is part of a guest range translation. (3) The walker checks also the corresponding entry of the nested page table (nPTE/nPMD) which indicates if the page is also part of a host range translation. If both set, a nested range walk is triggered. This first part filters out unnecessary range walks.

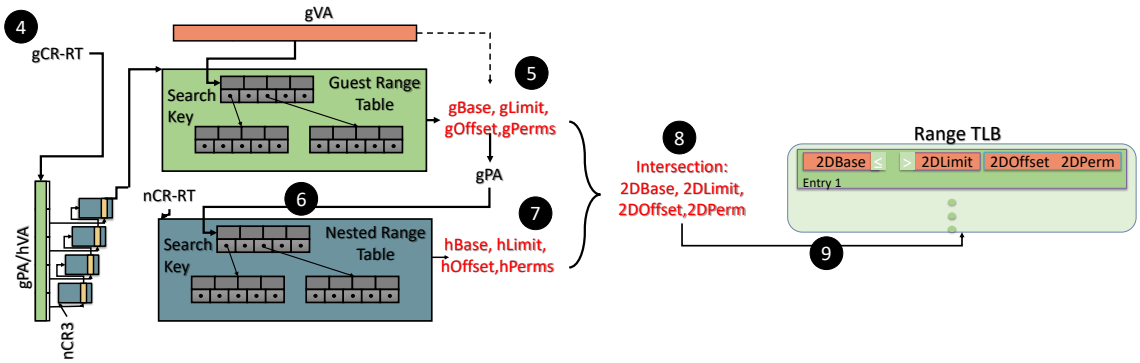



Figure 4.11: Range TLB miss in vRMM. Part2: The nested range walker walks the guest and nested range tables, and generates the 2D range translation.

Figure 4.11 shows the actual nested range walk. The standard nested page walker is used to translate the gCR-RT register to identify the hPA of the page storing the guest range table (4). Then, the range walker walks the guest range table and identifies the guest range translation, similarly to native RMM (5). In case the guest range table does not fit in a single page, the translation step of gPA→hPA occurs for each accessed page of the guest range table using the standard nested page walker. Fortunately, the representation of range translations in the range table is compact; a single page can hold 128 range translations, which is sufficient for mapping the vast majority of a process address space.

Once the nested page/range walker has identified the guest range translation for the gVA of the missing address translation, it calculates the corresponding gPA using the gOffset attribute (gPA=gVA−gOffset) (5). It uses it as a search key for the host range walk. The range walker uses the nCR-RT register to walk the nested range table and identifies the host range translation where the gPA of the missing address translation belongs to (6-7). Note that the walk of the range tables happens in the background off the critical path, so as the latency of TLB miss remains unaffected.

The dynamic generation of the full 2D gVA→hPA range translation requires the intersection of the guest and host (nested) ranges (8), which can be of different size and unaligned. As discussed in Section 4.2, in virtualized execution ranges in each dimension are created in a non-coordinated way. Thus the guest ranges can be backed by multiple host/nested ranges and vice-versa (Figures 4.8 and 4.9), in the same way that a guest huge page can be backed by small pages in the host and the reverse. The intersection is done on the guest physical address space (gPA) as ranges of both dimensions are reflected there. We name gPABase and gPALimit the guest physical addresses of the guest range boundaries.

- gPABase = gBase-gOffset
- gPALimit = gLimit-gOffset

The minimum distance of the gPA of the page that triggered the range TLB miss, from the gPABase/hBase and gPALimit/hLimit –the boundaries of the guest/nested ranges– is used to calculate the 2DBase, 2DLimit and finally the 2DOffset.

- 2DBase = hBase < gPABase ? gBase : (gBase + (hBase-gPABase))
- 2DLimit = gPALimit < hLimit ? gLimit : (gLimit - (gPALimit-hLimit))
- 2DOffset = 2DBase-2DLimit

All the above are generated online by the walker. Figure 4.11 shows an example of generating the direct gVA→hPA range translation. After the effective 2D range is generated online, it is finally fetched on the range TLB (9).

#### 4.3.1.3 vRMM design requires complex and redundant virtualization extensions

vRMM has the potential to exploit all existing contiguity, supporting unlimited and unaligned ranges in all dimensions. But as shown in the previous paragraphs, this comes at a non-negligible cost. Additional support to maintain nested range tables is required, and the nested range walker is a complex hardware design that among others: a) uses the nested page walker to get the host physical address of the pages that store the fragments of the guest's range table B-tree that need to be traversed and b) has additional logic to generate online the intersection of guest and nested ranges. All this additional overhead to an already complex and redundant to nested paging design may make vRMM a less appealing design choice for adoption by processor vendors.

**Other alternatives.** Hybrid coalescing [172] combines contiguous page translations into one translation entry, and augments TLBs to hold both coalesced and regular page translations. The coalesced entries are aligned at variable granularity (anchor distance). The OS stores the coalesced entries in modified page tables, and dynamically adjusts the anchor distance to reflect the process's average contiguity. Virtualizing hybrid coalescing (named as vHC in this work) involves separate anchor distances for the guest and the host OS and therefore would require: (i) the hypervisor to maintain the host coalesced entries in the nested page tables, and (ii) an augmented nested page walker to intersect guest/host entries and calculate the 2D coalesced entry, respecting guest alignment. vHC requires simpler architectural support than vRMM, as it augments the existing radix page table trees, making also the nested walk easier. However, vHC suffers from its alignment restrictions. Table 4.2 shows the number of vRMM ranges and vHC coalesced entries required to cover the 99% of big-memory workload's footprint in virtualized execution. We observe that CA successfully supports both techniques, significantly reducing the total number of entries for both methods compared to default THP. However, we observe that vHC fails to fully exploit the contiguity generated by CA as the anchor entries are 38× compared

| | Mem. (GB) | default THP Ranges | default THP vHC entries | CA paging Ranges | CA paging vHC entries |
|---|---|---|---|---|---|
| **SVM** | 29G | 3759 | 6224 | 10 | 422 |
| **PageRank** | 78G | 37453 | 39355 | 11 | 828 |
| **hashjoin** | 102G | 4152 | 4260 | 7 | 403 |
| **XSBench** | 122G | 4658 | 4968 | 11 | 644 |
| **BT** | | | | 931 | 7061 |
| **geomean** | - | 7223 | 8485 | 23 | 914 |

Table 4.2: Number of ranges (vRMM), and anchor entries (vHC) to map 99% of the footprint of big-memory workloads, using (i) default THP, (ii) CA paging in virtualized execution.

to ranges. This is due to the method's virtual alignment restrictions, confirming the important performance potential of unaligned contiguity.

### 4.3.2 Speculative Offset-based Address Translation

We pose the research question: *Can we have high performance translation leveraging unaligned contiguity of unlimited size but with simpler hardware support?*

**Observation.** We find that the root cause of vRMM's complexity and vHC's low performance potential is the requirement for explicit tracking of the mappings' virtual and physical boundaries. *What if we speculate them?* Figure 4.7 shows the key idea of SpOT; instead of tracking mappings boundaries in guest and host, SpOT tracks only gVA→hPA *2DOffsets* and uses them to predict missing address translations.

#### 4.3.2.1 SpOT Overview

We present SpOT in the context of virtualized execution as its operation in native execution can be inferred in a straightforward manner. SpOT works on the *micro-architectural* level and it primarily consists of a simple *prediction table* that caches *[2D offset, permissions]* translation tuples (Figure 4.12). Each *offset* maps from gVA→hPA and is dynamically calculated and stored in the prediction table by the nested page walker (HW) at the end of the walk for a missing gVA translation. On a last level TLB miss, SpOT uses the *offset* generated by the previous TLB miss of the same memory instruction and predicts a host physical address (hPA). In a sense, SpOT speculates that the specific instruction is accessing a contiguously mapped range of pages and transparently tracks its corresponding offset to perform predictions. SpOT feeds the processor with the predicted hPA to continue execution in a speculative mode and the verification page walk happens in background. Thus, SpOT hides the latency of page walks.

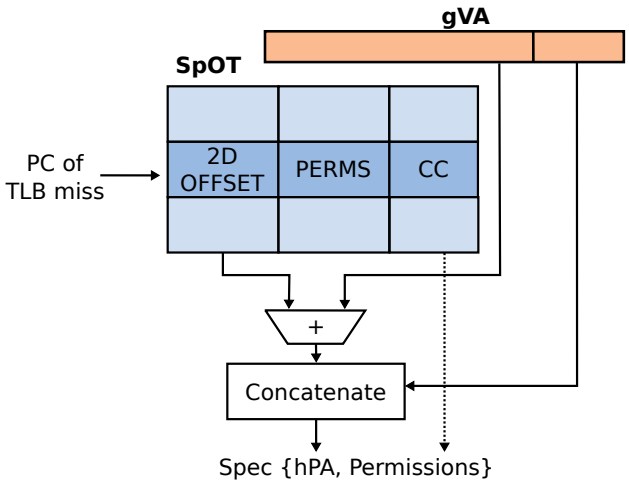


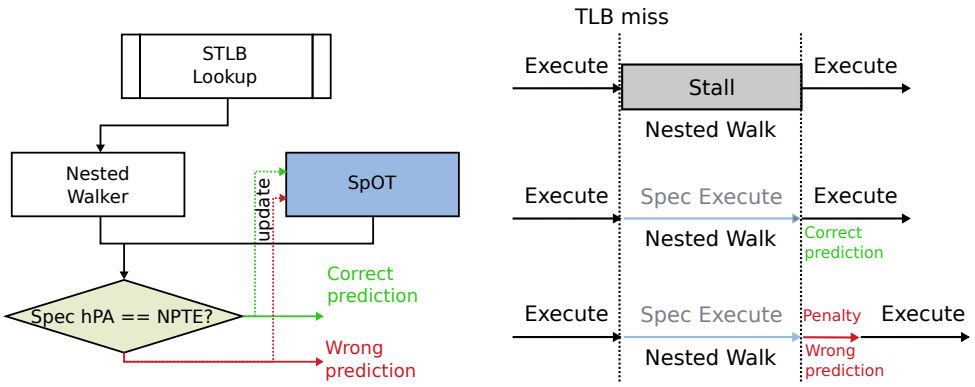

Figure 4.12: SpOT predicts the physical address of missing translations, inferring the offsets of contiguous mappings. It consists of a micro-architectural prediction table tracking the [offset,permissions] of recently missed translations.

#### 4.3.2.2 SpOT Mechanism

**Prediction Table.** To lookup and fill the prediction table with offsets, we use the program counter (PC) for indexing and tag matching. Only a few instructions are typically responsible for most TLB misses and therefore PC-indexing keeps the table size small (64 entries in our experiments). It also serves the core idea of SpOT to correlate instructions with contiguous mappings. In the presence of a contiguity-aware allocator (CA paging) and thanks to locality, each memory instruction usually performs accesses (regular or irregular) inside a contiguously mapped range of pages at different execution phases.

**TLB miss path.** We consider address translation speculation only for the last-level TLB misses. Such misses trigger both the default nested page walk and a parallel lookup in SpOT's separate prediction engine (Figure 4.13a). SpOT uses the [offset,permissions] retrieved by the prediction table to predict a host physical address (hPA) translation for the guest virtual address (gVA) that caused the miss (Figure 4.12). It subtracts the *offset* from the gVA and predicts a *spec* $hPA = gVA - offset$. It also speculates that this memory access will have the same permission rights as the previous access of the same instruction. Then it feeds the processor with the *spec hPA* to continue its execution but in speculative mode. It is worth mentioning that unlike previous designs [178], we do not fetch the predicted translation in the regular TLB hierarchy, leaving the TLB design intact.

As the speculative execution proceeds, the verification nested page walk happens in the background (Figure 4.13a, 4.13b). When the walk completes, the spec hPA is compared to the original




(a) SpOT predicts a physical translation on a TLB miss.

(b) Speculative execution in parallel with nested walk.

Figure 4.13: SpOT is integrated in the L2 TLB miss path and hides nested page walk latency under speculative execution.

hPA retrieved from the nested page table entry (NPTE) and two scenarios exist: (i) the speculation was *correct* and SpOT managed to hide the walk latency with useful speculative execution, (ii) the speculation was *incorrect* and SpOT must flush the pipeline and replay the memory instruction. Flush is necessary because following instructions may have consumed incorrect data. Incorrect predictions (mis-predictions) affect performance, as the cost of flushing is added on top of the regular nested page walk latency.

**Prediction table fills.** The prediction table is updated at the end of a nested page walk. The offset of the missing translation is calculated ($offset = gVA - hPA$) and fetched in the table along with the permission rights of the access. To minimize possible PC conflicts, SpOT's prediction table is a set associative structure and uses an LRU replacement policy.

**Building confidence.** As mis-predictions restricts SpOT's effectiveness, we add a 2-bit saturating counter for each prediction entry. When an offset is firstly fetched into the prediction table the counter is set to 1. Correct predictions increase the counter by 1 and mis-predictions decrease it. Predicted physical addresses are fed to the processor only when the counter is $>1$. When the counter is $<=1$ no speculation is performed. Predictions, though, are still calculated and compared to the original hPA at the end of each nested page walk to update the confidence counters. Finally, an entry is replaced with a new offset only when the counter equals 0.

**Preventing thrashing.** To further boost SpOT accuracy, we involve the OS into filtering offsets with low prediction potential. If the prediction table is updated on every TLB miss, offsets that do not belong to large contiguous mappings may thrash it. Such offsets will never gain confidence to enable predictions and will evict valuable offsets from the table.

We mark translations that belong to larger contiguous mappings using a reserved bit in their corresponding page table entry (PTE), similar to vRMM. In detail, the OS (CA paging) sets this

bit at the end of a successful allocation (page fault) when updating the PTE. The OS checks if the neighboring PTEs have the bit set, i.e., belong to contiguous physical pages. If that bit is not set, the OS examines whether the last allocation extended a contiguous mapping beyond a *threshold* size. If so, the OS sets the bit to all PTEs that belong to that contiguous mapping, establishing its offset as candidate to trigger predictions. Even CA paging could dynamically adjust the threshold based on its contiguity statistics, we currently empirically set it to 32 contiguous pages. With nested paging, the guest and host OS sets the bit in the gPTs and nPTs, and the nested page walker updates the prediction table only if both bits are set. This optimization crosses the border of micro-architecture but we consider it a very simple and cheap mechanism. Note that still the accurate size and boundaries of contiguous mappings are not calculated or tracked.

**Hiding the verification page walk cost.** With SpOT the page walk can be entirely or partially overlapped with useful work in case of correct prediction. This can include prefetching data using the speculative address translation, overlapping the page walk with the data fetch cost [55, 109, 178]. In case the processor allows aggressive speculative execution, it can execute instructions that depend on the missing translation/data, increasing further performance opportunity.

## 4.4 Discussion

#### 4.4.0.1 SpoT Security Considerations

Speculation has been identified as a source of security vulnerabilities through cache side-channel attacks [62, 107]. Transient unsafe loads (USLs [107]) executed at a hardware mis-speculated control/data path can transmit secret data via micro-architectural covert channels before the mis-speculation is resolved. In example, USLs are the loads that are executed after branch predictions (Spectre attacks [139]) or the loads executed after exceptions (Meltdown attacks [150]). SpOT introduces a new unsafe memory instruction, i.e., load missing in the TLB, that an attacker can exploit to read data from unauthorized memory locations. The loads that follow a SpOT prediction are considered USLs until the prediction is verified. Fortunately, proposed mitigation techniques [135, 224, 225] for Spectre/Meltdown-type attacks also mitigate SpOT's vulnerabilities. Specifically, those techniques fetch and keep the data of USLs in a speculative buffer and do not commit their changes to the cache hierarchy until the loads are considered safe. Preventing USLs from changing the cache micro-architectural state effectively blocks all cache side channel attacks. Note that such mitigation techniques are necessary for secure execution regardless of SpOT's presence. Section 4.5.1.2 discusses the performance impact of SpOT when such mitigation techniques are employed. Finally, SpOT does not speculatively change the TLB state, so no additional mitigation for MMU attacks [208] is required.

#### 4.4.0.2 CA paging Considerations

**VMA size.** CA paging targets big-memory applications that suffer from high translation overheads. Such applications typically have a few large VMAs. If an application has multiple small VMAs, CA paging will inherently create multiple contiguous mappings due to the discontinuities in its virtual address space. Such applications may not benefit from the translation schemes that CA paging supports.

**Reservation.** Under severe memory pressure, different processes or VMAs may end up competing for the same scarce contiguous physical blocks. To shield contiguity, CA paging could employ reservation [161, 202]. In this paper we opt for best-effort strategies and consider reservation for future work.

**NUMA placement.** CA paging is currently aligned to the vanilla Linux policy with respect to NUMA placement. Extending demand paging, it allocates pages from the node local to the core that makes the allocation request. We have not examined autoNUMA, a technique that migrates pages across nodes tracking locality via periodic hint page faults. We consider extending CA paging to support autoNUMA faults –migrating pages to contiguous space– for future work.

## 4.5 Evaluation

**OS prototype and server machine.** We prototype CA paging in Linux v4.19 for anonymous/copy-on-write page faults and page cache allocations. We use Qemu/KVM v2.1.2 for virtualized execution. Our code and scripts are publicly available on GitHub[2]. Table 4.3 summarizes the configuration details of our experimentation system. In our study, we focus only on the costly L2 STLB misses that trigger page walks; hence, we refer to those as TLB misses for simplicity.

**Contiguity results.** We collect statistics for contiguous mappings through page table information. We use the standard `pagemap` [14] API for native execution, and we develop an in-house virtual-machine introspection (VMI) tool with similar functionality for virtualized execution. For the latter, the guest OS exposes the application's guest page table to the host (registering its location in the guest physical address space), and then the host reads and combines the guest and the nested page tables info to calculate a full 2D translation.

**Hardware emulation.** We emulate various hardware address translation schemes by instrumenting the TLB misses that trigger page walks in our real system as applications run with BadgerTrap [100] in the guest OS. BadgerTrap uses page table marking to force TLB misses to cause page faults and enables hardware emulation in special fault handlers. For SpOT, we use a 4-way set associative prediction table of 32 entries. For Direct Segments (DS), we use the dual direct mode [101] that allows direct 2D gVA→hPA translation through a single direct segment.

[2]https://github.com/cslab-ntua/contiguity-isca2020.git

| | |
|---|---|
| **Native Environment** | |
| **Processors** | 2-socket Intel Xeon CPU E5-2630 v4 (Broadwell)<br>10 cores/socket, hyperthreading disabled, 2.2GHz |
| **L1 DTLB** | 4K: 64-entry, 4-way set associative<br>2M: 32-entry, 4-way set associative |
| **L2 DTLB** | 4K/2M: 1536-entry, 6-way set associative |
| **Memory** | 256G (128G per socket) |
| **OS** | Debian Linux v4.19 |
| **Fully-Virtualized Environment** | |
| **VMM** | QEMU (KVM) v2.1.2 20vCPUS 2-socket |
| **Memory** | 256G (128G per socket) |
| **Host/Guest OS** | Debian Linux v4.19 |
| **Emulated hardware** | |
| **Direct Segment** | Dual direct mode (single 2D segment) |
| **vRMM** | Range TLB: 32-entry, fully associative |
| **SpOT** | Prediction Table: 32-entry, 4-way set associative |

Table 4.3: System Configuration.

For vRMM, we use a fully-associative range TLB of 32 entries and we implement the guest and nested range tables as flat arrays, rather than B-trees. To identify the boundaries of 2D translations inside the guest OS, we expose the nested range table to the guest at a reserved guest physical address area, using the standard nested page tables.

**Performance model.** We collect statistics from performance counters with *perf* (CPU cycles, TLB misses, page walk cycles) to quantify virtual memory overhead. In more detail, we use PAPI [204] commands injected in the benchmarks code to exclude their initialization phase. To keep a common baseline, we adopt the same methodology of prior works [47,83,101,130,178]. We identify the ideal execution time of zero address translation overhead ($T_{ideal}$) and then compare all measured and simulated overheads to the ideal execution time using a simple linear performance model. For vRMM, we assume that the latency of the nested range table walk is hidden entirely in the background. For SpOT, we assume that: (i) correct speculations hide the entire TLB miss cost, (ii) decisions to not apply speculation expose the entire TLB miss cost, and (iii) mis-speculations add extra 20 cycles for flushing the pipeline [178] on top of the TLB miss cost. For DS, we assume the dual direct mode that provides gVA→hPA address translation. Table 4.4 summarizes how we compute virtual memory overheads for the various configurations.

| Performance Model | |
|---|---|
| **Ideal execution time** | $T_{ideal} = T_{THP} - C_{THP}$ |
| **Native 4K/THP overhead** | $O_{4K/THP} = C_{4K/THP}/T_{ideal}$ |
| **Virtual. 4K/THP overhead** | $O_{v4K/vTHP} = C_{v4K/vTHP}/T_{ideal}$ |
| **vRMM overhead** | $O_{vRMM}$ =$(M_{SIM} * AvgC_{vTHP})/T_{ideal}$ |
| **DS overhead** | $Over_{DS}$ =$(M_{SIM} * AvgC_{v4K})/T_{ideal}$ |
| **SpOT overhead** | $O_{SpOT}$ =$((NP_{SIM} * AvgC_{vTHP})+MP_{SIM}*$ $(AvgC_{vTHP} + MP_{penalty}))/T_{ideal}$ |
| T: Total execution cycles | AvgC: average cost of page walk |
| C: Cycles spent in page walks | $M_{SIM}$: Simulated page walks |
| $MP_{penalty}$: 20 cycles | $MP_{SIM}$: Simulated mispredictions |
| v4K/vTHP: 4K+4K/THP+THP | $NP_{SIM}$: Simulated no predictions |

Table 4.4: Performance model based on hardware performance counters and hardware emulation with BadgerTrap [100].

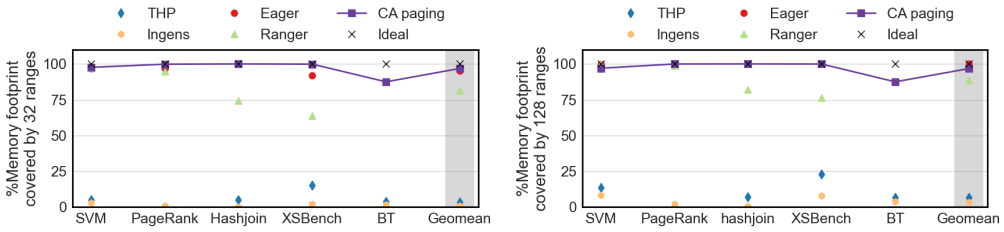
| Workloads | | |
|---|---|---|
| **OpenMP (10 threads)** | hashjoin microbenchmark | 102G |
| | XSBench [206] | 128G |
| **Serial** | Liblinear SVM [94], kdd12 dataset | 29G |
| | Ligra PageRank [192], *friendster* graph [147] | 78G |
| | BT (NPBe [41]) class E | 167G |

Table 4.5: Workloads description and memory footprint.

**Workloads.** We use a set of memory/TLB intensive workloads, single- and multi- threaded, from graph analytics, high performance, and machine learning domains (Table 4.5). Note that we run PageRank with a single thread to enable comparison with Translation Ranger [227] as multi-threaded execution was erroneous. CA paging results remain similar for both the single- and multi-threaded version.

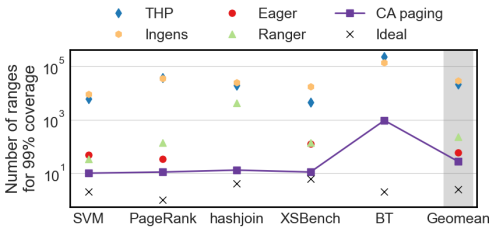
### 4.5.1 Results

We first evaluate the impact of CA paging on the creation of contiguous mappings in both native and virtualized environments. We then evaluate SpOT that exploits the contiguity of CA paging to mitigate the address translation overhead in virtualized execution.




(a) Coverage with 32 largest mappings

(b) Coverage with 128 largest mappings




(c) #mappings to cover 99% of memory

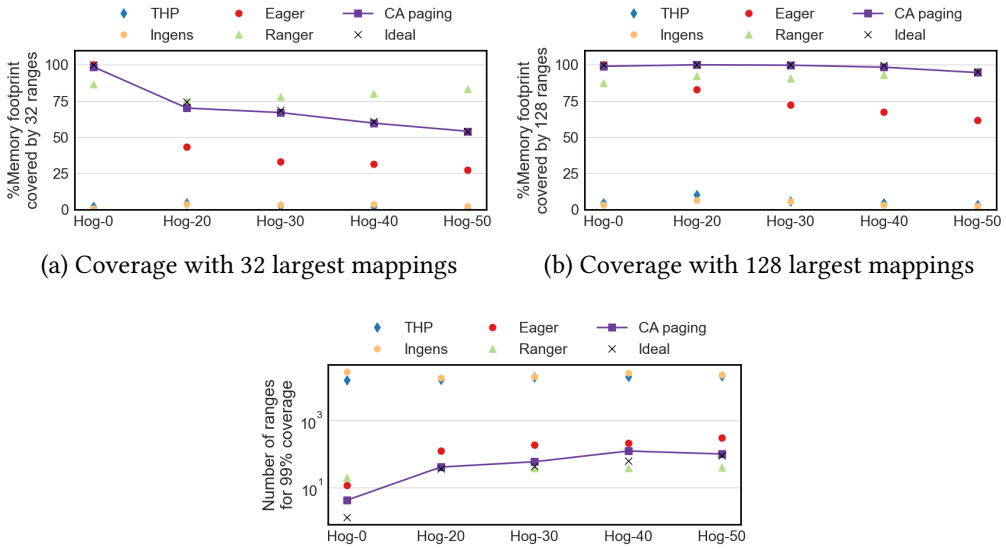
Figure 4.14: Contiguity performance without memory pressure for native execution.

#### 4.5.1.1 Software Results: Contiguity-aware Paging

We compare *CA paging* with: (i) *default paging–THP*, the default OS technique that supports transparent 2M allocations, (ii) *Ingens* [143], a transparent huge page management framework that performs asynchronous huge page promotions, (iii) *eager paging* [130] that increases the kernel MAX_ORDER attribute to allow the buddy allocator to maintain larger blocks and uses them to perform pre-allocations, (iv) *translation ranger* [227] that coalesces application's memory footprint asynchronously using post-allocation page migrations, and (v) *ideal paging* that applies an offline best-fit algorithm to find the maximum contiguity that could be provided based on the *contiguity_map*'s state before execution. As CA paging is applicable to native and virtualized execution (Section 4.2), we first present extensive native results to allow comparison with the other techniques. We summarize virtualized execution performance at the end of this section.

To evaluate the impact on the virtual-to-physical mapping contiguity, we use the memory footprint coverage of the 32 and 128 largest mappings (higher is better) and the average number of mappings required to cover 99% of the total footprint (lower is better), averaged throughout application's execution time [227]. For all configurations, we use a modified TCMalloc [17], that increases maximum allocation as proposed for eager paging [130]. Note that, CA paging and ranger are independent to the user space allocator. We did CA paging experiments with standard libc and the results remain unchanged.

**Contiguity in the absence of memory pressure.** Figure 4.14 summarizes the contiguity results when applications execute natively on a machine without external fragmentation. Both




(a) Coverage with 32 largest mappings

(b) Coverage with 128 largest mappings

(c) #mappings to cover 99% of memory

Figure 4.15: Contiguity performance under memory pressure/external fragmentation. Geomean results for all benchmarks.

THP and Ingens perform similarly, generating thousands of non contiguous mappings to cover applications footprint. This is expected behavior as both techniques control and manage contiguity up to 2MB (huge page). CA paging generates contiguity comparable to that of eager paging and improved compared to translation ranger, avoiding pre-allocations and page migrations. It covers on average 99% of applications footprint with ∼27 mappings, orders of magnitude less than default paging. The effectiveness of translation ranger for XSBench and hashjoin decreases, as their allocation phase is significant compared to total execution, and post-allocation migrations takes time to fully coalesce their footprints (Figure 4.2b). CA paging performance drops for the BT workload as irregular faults compete for the last contiguous free blocks of the first NUMA node, right before it spans to the second. We plan to study this side-effect in the future.

Note that we exclude hashjoin from eager paging results and BT from both eager paging and translation ranger results. The two benchmarks, either due to memory bloat (hashjoin) or own requirements (BT), span over two NUMA nodes and those techniques currently do not support NUMA topologies.

**Fragmentation Impact.** To profile external fragmentation impact we use a "hog" micro- benchmark [143, 176, 177]. Due to the increased memory pressure, all workloads footprint span two NUMA nodes as there is not enough free memory in a single node to cover them. For that reason, we turn NUMA off, via Linux kernel boot parameters, to enable comparison of CA paging with the other techniques. Figure 4.15 summarizes the geometric mean contiguity results for

all benchmarks when memory pressure increases from 0% to 50%. We exclude BT as its 167G footprint does not fit in the "hogged" memory.

Both THP and Ingens perform poorly and similar to the no memory pressure case. This is expected as our hogging micro-benchmark fragments physical memory in coarse granularities (>2MB) and thus, there are plenty of free huge pages to back benchmarks' footprints.

CA paging is fairly robust, outperforming eager paging. It covers ∼94% of the footprints with only 128 mappings under maximum pressure (hog-50) and always follows Ideal paging (with small deviations). Therefore, CA paging manages to fully exploit the available unaligned free contiguity in the system. On the other hand, eager paging is highly sensitive to fragmentation due to alignment restrictions. It relies on buddy allocator's higher order blocks and the allocator tracks only aligned contiguous blocks. Finally, translation ranger remains almost unaffected by the increasing memory pressure, outperforming all allocation techniques in 32 mappings coverage (better than Ideal paging), as it relies on post-allocation migrations. CA paging, however, achieves similar performance with respect to 128 mappings and 99% coverage. Generally, we consider the two approaches orthogonal and mutually assisted; CA paging can generate early-on contiguity, and if required, ranger's migrations can further boost it, similarly to how khugepaged [19] complements THP allocations.

**Fragmentation restraint.** Previously we evaluated contiguity on an already fragmented machine; CA paging, though, can delay fragmentation as a machine ages. Figure 4.16 depicts the distribution of the unaligned free block sizes after a set of benchmarks runs to completion using default and CA paging. We notice that a significantly larger portion of free memory is backed by >1GB blocks. This is attributed to the allocation (and consecutive release) of contiguous pages and to the long-lived contiguous page cache mappings (Section 4.2).

**Multi-programmed case.** Figure 4.17 depicts contiguity results while running two instances of the SVM workload without fragmentation. CA paging provides increased contiguity, avoiding eager pre-allocations, as the next fit placement policy successfully prevents workloads interference over the same free blocks. Translation ranger fails to coalesce the two footprints, migrating

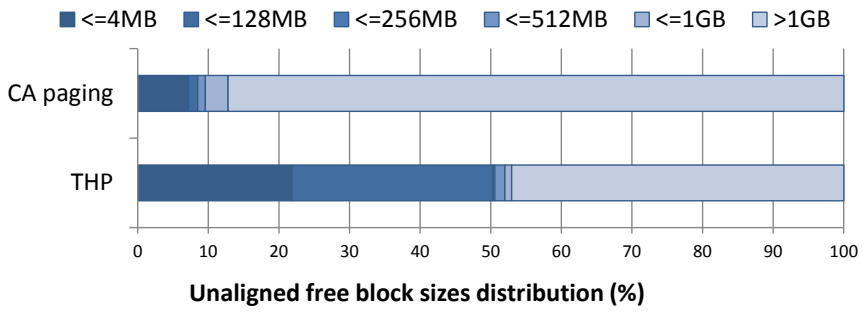



Figure 4.16: Free block size distribution after benchmarks execution.

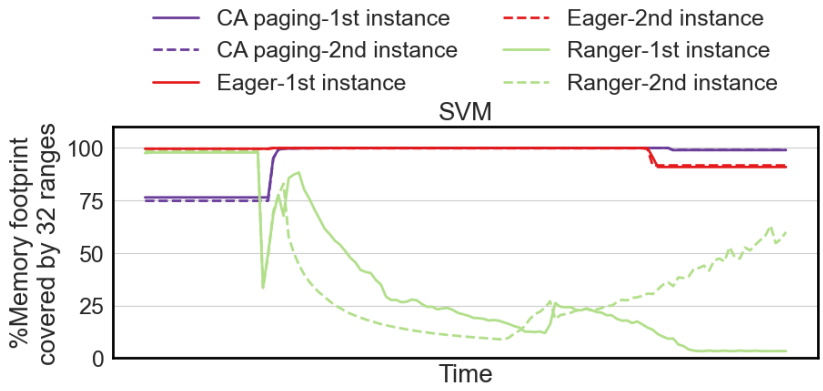



Figure 4.17: 32 largest mappings coverage while running two instances of SVM (straight and dotted line for every method).

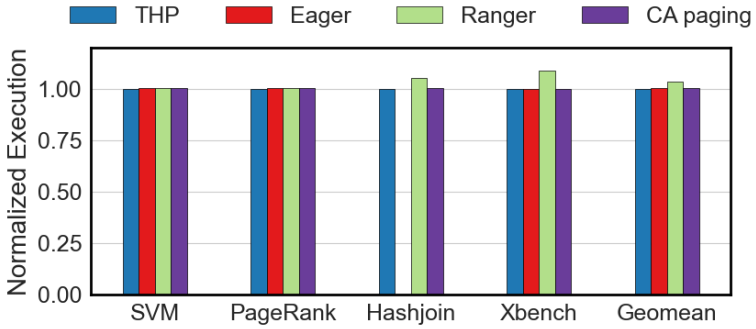



Figure 4.18: Software runtime overheads normalized to THP.

pages between them across the entire execution. Note that the code released for ranger is not optimized to serve multiple processes. Multi-programmed workloads require ranger to scan serially all processes' footprint at every defragmentation epoch, penalizing its response time.

**Software Overhead Analysis.** We evaluate the isolated software overheads of the different mechanisms when there is no gain from novel larger-than-a-page address translation schemes. Figure 4.18 depicts the normalized execution time of benchmarks running on our commodity hardware. Hashjoin does not run with eager paging as the benchmark spans to two NUMA nodes with this method and this is not supported. Translation ranger penalty is ∼3% on average due to page migrations. Eager and CA paging add no overhead. We also run a set of TLB friendly workloads from Spec2017 and find that the execution time is not affected by CA paging. However, the two paging methods behave differently with respect to tail latency and resource utilization. Table 4.6 summarizes the number of page faults and their average latency (us) measured for all benchmarks with ftrace [10]. CA paging does not affect latency while eager paging magnifies it due to zeroing large blocks. The latter though decreases the total number of faults.

Table 4.6: Total number of page faults and 99th latency (us).

| **99th latency (us)** | | | **Total Number of page faults** | | |
|---|---|---|---|---|---|
| THP | CA paging | Eager paging | THP | CA paging | Eager paging |
| 515 | 526 | 80372 | 45148 | 45148 | 67 |

Table 4.7: Bloat [memory (overhead%)] compared to 4KB.

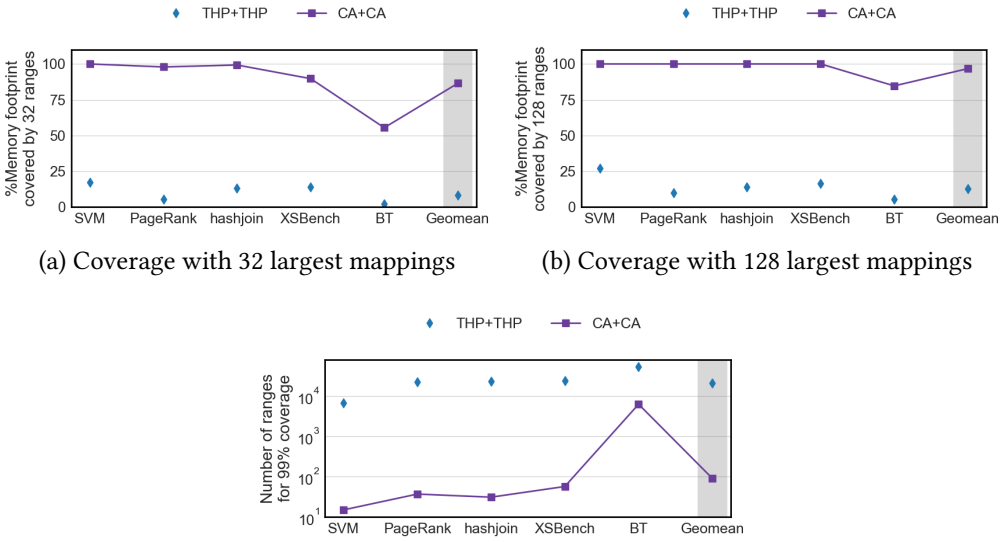
| | **SVM** | **PageRank** | **hashjoin** | **XSBench** | **BT** |
|---|---|---|---|---|---|
| **THP** (MB) | 13.3(0.0%) | 5.2(0.0%) | 3.8(0.0%) | 4.7(0.0%) | 136(0.1%) |
| **Ingens** (MB) | 1.4(0.0%) | 3.3(0.0%) | 0.4(0.0%) | 1.4(0.0%) | 89(0.0%) |
| **CA** (MB) | 13.1(0.0%) | 6.8(0.0%) | 3.3(0.0%) | 6.2(0.0%) | 137(0.1%) |
| **Eager** (GB) | 2.3(8.0%) | 5(6.5%) | 48(47.5%) | 0.5(0.4%) | 0.1(0.1%) |

Finally, Table 4.7 summarizes the extra memory allocated by the different techniques compared to demand paging (bloat) with 4K pages. We observe that CA paging and THP perform the same (bloat up to 136MB), as CA paging builds on top of THP and does not affect the page size decision. Ingens, on the other hand, decreases it as it asynchronously promotes 4K pages to huge based on utilization. Note that CA paging can add mechanisms from Ingens to boost contiguity while preserving the low internal fragmentation that Ingens offers. We plan to study this combination for future work. Finally, eager pre-allocation suffers the most as it leads to occupation of multiple GBs that the application will not eventually use.

**Virtualized execution.** Figure 4.19 summarizes the results for virtualized execution. We employ CA paging in both guest and host OS independently, without any form of coordination, and measure the 2D gVA→hPA mappings contiguity. On average, CA paging decreases the number of mappings required for 99% coverage by an order of magnitude (∼90) compared to default paging and covers ∼86%/∼96% with 32/128 mappings. However, we observe that 32 mappings coverage is slightly worst compared to native execution. This is expected as the contiguous mappings in the guest and host dimensions are created independently and on a best-effort basis. Note also that our applications run consecutively without VM reboots. Therefore, unaligned mismatches between the guest and the host contiguous mappings are more frequent, as the gPA-to-hPA mappings persist across benchmarks runs (Section 4.2).

#### 4.5.1.2 Hardware Results: vRMM and SpOT

We now quantify the execution overhead of address translation and evaluate vRMM and SpOT in virtualized execution. Figure 4.20 summarizes our findings. We run our experiments without extra memory pressure (no ”hogging”). We use performance counters to measure the translation




(a) Coverage with 32 largest mappings

(b) Coverage with 128 largest mappings

(c) #mappings to cover 99% of memory

Figure 4.19: Contiguity performance without memory pressure for virtualized execution.

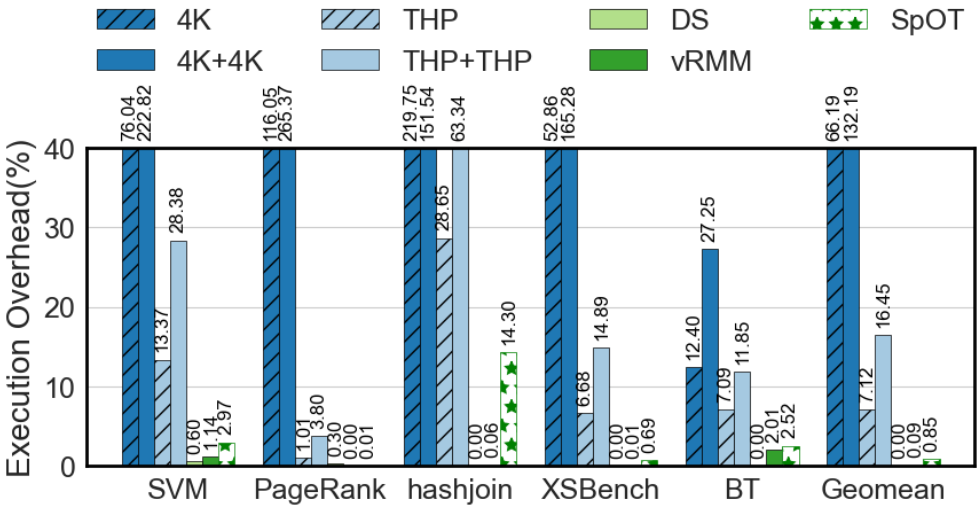



Figure 4.20: Execution time overheads due to data TLB misses that trigger page walks in virtualized execution.

penalty in native (hashed bars) and virtualized (solid bars) execution with base pages (blue) and Transparent Huge Pages (lightblue). We emulate the performance of SpOT, vRMM and DS [101].

**Paging Overheads.** For native execution, our results corroborate all past studies indicating that address translation overheads are exceptionally high with 4K pages. Overheads above 100% are due to overlapped page walk cycles and comparison with the ideal baseline. THP reduces

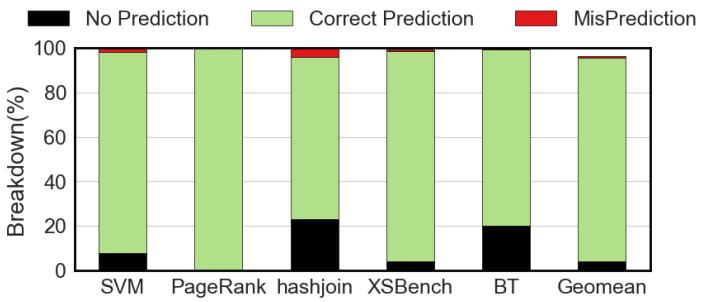



Figure 4.21: Percentage of TLB misses that SpOT made (i) correct predictions, (ii) mispredictions, and (iii) no predictions.

substantially the overhead but fails to eliminate it, bringing it to ∼7% on average and up to ∼13% for SVM. Note >99% of the memory footprint of the workloads is mapped with 2M pages. In virtualized execution, the address translation overhead is magnified due to nested page walks. Even with THP on, it grows to ∼16.5% on average and up to ∼28% for SVM.

**vRMM Performance.** For vRMM we use CA paging in both the guest and host OS. Note that RMM was originally proposed with a pre-allocation technique named eager paging [130]. We verify previous section's observation that CA paging is equally effective, avoiding all drawbacks of pre-allocation, when memory is not fragmented. vRMM with CA paging reduce the translation overheads to less than 0.1% on average. Range TLB hits are always above 95%, reducing the number of nested page walks. The technique performs slightly worse for SVM and BT due to CA paging stressing, discussed also in the previous paragraph.

**SpOT Performance.** For the evaluation of SpOT we apply CA paging in both the guest and host OS as well. SpOT reduces the translation overheads to ∼0.85% on average. The performance improves significantly for all applications, but less for SVM and BT. For BT, CA paging (Figure 4.14a,4.19a)) fails to provide optimal contiguity when the application expands to the second NUMA node as discussed in Section 4.5.1.1. For SVM, CA paging successfully maps 99% of the application's footprint with less than 32 mappings (Figure 4.19a) but a portion of the observed TLB misses (∼4%) are on a few virtual addresses that fall outside these mappings. SVM has also high number of irregular TLB misses triggered by the same instruction. SpOT sensitivity to the access pattern is highly exposed by the hashjoin micro-benchmark as well which makes random accesses. To better understand SpOT performance, Figure 4.21 breaks down the percentage of TLB misses predicted correctly, mis-predicted, and not predicted at all. We observe that correct predictions can be over 99% (PageRank), while mis-predictions never more than 4% (hashjoin).

**Comparison with Direct Segments.** Finally, we compare vRMM and SpOT with direct segments (DS) dual mode [101]. We observe that DS eliminate TLB miss penalty almost entirely.

Despite its prominent efficiency, the method is rigid, reserving the segment when a virtual machine boots and abolishing paging. SpOT and vRMM, combined with CA paging, preserves the benefits of demand paging sustaining high address translation performance comparable to DS.

**Security mitigation techniques discussion.** As discussed in Section 4.3, SpOT can be exploited to leak data from unauthorized memory locations through cache side-channel attacks. Fortunately, proposed Spectre/Meltdown mitigation techniques [135, 224, 225] can also mitigate SpOT vulnerabilities. However, such techniques introduce performance overheads proportionate to the number of Unsafe Loads (USLs) [224], i.e., loads that are executed in speculative state. Studying accurately the impact of SpOT USLs requires full system cycle-accurate simulation that is prohibitively slow for our TLB studies. However, we make some rough estimations for the number of SpOT USLs and their impact on performance, assuming the InvisiSpec design [224, 225]. We use performance counters to measure the number of TLB misses, loads, cycles, and the average latency of page walk, and we calculate the number of SpOT USLs (Equation 2 in Table 4.8). To put our results into perspective, we also measure the number of branches, and we compare the number of SpOT USLs with the number of Spectre USLs, i.e., unsafe loads that are executed due to branch predictions (Equation 1 in Table 4.8). We assume a linear distribution of load instructions over time. Table 4.8 summarizes the results for all our workloads (geometric mean). We observe that the events that trigger speculative execution with SpOT, i.e., TLB misses, are a small fraction (0.25%) compared to Spectre's branch predictions (5%). However, SpOT's transient window of speculative execution is much larger (the average page walk latency is $\sim$81 cycles in our experiments) compared to branch resolution ($\sim$20 cycles [178]). In total, $\sim$3% of total instructions would execute as USLs with SpOT, whereas the percentage of USLs with Spectre would be $\sim$16%. As InvisiSpec for mitigating Spectre USLs has been shown to add $\sim$5% overhead [225], we expect that extending InvisiSpec for SpOT USLs would introduce $< 2\%$ overhead. Hence, SpOT's performance translation benefits would remain still beneficial.

| **Branches/ Instructions(%)** | **DTLB misses/ Instructions(%)** | **Spectre USL/ Instructions(%)** | **SpOT USL/ Instructions(%)** |
|---|---|---|---|
| 5.87 | 0.25 | 16.5 | 2.9 |
| Spectre USL = #Branches * Branch Resolution Cycles * Loads/cycle (1) | | | |
| Spot USL = #DTLB misses * Page Walk Cycles * Loads/cycle (2) | | | |

Table 4.8: Estimation of Unsafe Load Instructions (USL).

## 4.6 Related Work

**Memory Management.** Multiple software proposals [143,158,161,164] improve huge page management, addressing issues like fairness, memory bloat, increased tail latency, and fragmentation. Instead, CA paging targets the reduction of translation overheads that persist in the presence of huge pages, and builds on top of huge page management to create larger-than-a-page contiguous mappings for novel translation hardware. Other proposals control external fragmentation [106, 166] again in the scope of huge pages, focusing on the allocation [166] and the reclamation [106] OS routines. In contrast, we study fragmentation in coarser granularities and show that contiguous allocation beyond the page size can delay fragmentation.

**Address Translation Hardware.** Bhargava et al. [51] analyzed nested paging translation overhead and proposed MMU caching and large page sizes. Our experiments show that such support–that is present in commodity processors–is not sufficient, as the address translation overhead still remains significant. Other works have focused on the implications of huge pages and have proposed specialized hardware to support them better [89, 95, 98, 167, 171, 190, 200, 202]. Still, those designs provide limited TLB reach and suffer from alignment issues. SpOT and vRMM harvest unaligned contiguity to hide the page walk latency.

Multiple works [102, 214, 238] combine shadow and nested paging to minimize the MMU virtualization overhead. Our evaluation focuses on nested paging, the state-of-practice virtualization technique. While vRMM is dependent/complis to hardware-assisted virtualization, CA paging and SpOT are agnostic to the virtualization technology and directly applicable to shadow and hybrid paging. Ahn et al. [29] proposed an inverted shadow page table combined with a flat nested page table, and used speculative execution to relax the synchronization between the tables. That design modified paging subsystem extensively.

DVM [109] introduces regions for which the virtual address equals the physical address (identity mappings) and caches only the translation permissions. An optional enhancement speculates whether a mapping is identity. DVM restricts the flexibility of common OS mechanisms, e.g., copy-on-write and fork. Our approach is compatible with such mechanisms and vRMM and SpOT accelerate translations without any virtual or physical special address requirements.

Speculating address translation has been proposed by SpecTLB [45] and Glue [178]. SpOT, albeit motivated by those designs, differs in some key ways. Their target is to predict the physical addresses of multiple reserved (SpecTLB) or splintered (Glue) base 4KB pages that belong to a huge 2MB page. Because such base pages are aligned with respect to their huge page boundaries, they are promoted to a single speculative huge page TLB translation entry. Hence, SpecTLB and Glue target to sustain huge page performance under different memory management conditions. We are the first, on the other hand, to leverage speculation to exploit unaligned, larger-than-a-(huge)-page contiguous mappings while completely avoiding the complexity of maintaining

them in software (OS) and hardware. SpOT targets to predict translations far beyond the huge page limit and the mechanism is completely independent to virtual addressing and alignment. In a sense, SpOT builds on top of the idea of range translations without tracking them; instead it exploits instructions memory locality combined with inferred mappings contiguity to predict translations. Finally, SpOT's prediction mechanism bears similarities to SIPT [239] as they both use a PC indexed prediction table of offsets, but they target different problems and use different mechanisms. SIPT targets to speculatively index larger L1 data caches, predicts just a few (e.g., 1-3) bits of a physical address, and requires a complex perceptron confidence mechanism to throttle mispredictions. Instead, SpOT targets to predict the entire physical address translation to hide the cost of TLB misses without any complex confidence mechanism.

Several mechanisms reduce the cost of page walks either targeting alternative page table representations [31, 194, 234], enhanced MMU caches [44, 52], direct page table indexing [155], or page table replication [25]. SpOT is orthogonal as it hides page walk latency under speculative execution. TLB prefetching can also reduce TLB misses by predicting the next missing translation [57, 127, 188]. Instead, SpOT predicts the actual address translation itself.

Finally, prior works propose: (i) storing TLB data as part of the memory subsystem [154, 184], (ii) pinning frequently accessed pages with poor temporal locality to reduce the number of TLB misses [92], (iii) modifying TLBs to better accommodate chip multiprocessors [56, 197, 237] and (iv) reducing TLB shootdown overheads through hardware [40, 182, 209, 228] or OS [34, 35, 141] optimizations. Both vRMM and SpOT are orthogonal to those mechanisms, as they are either redundant to the page TLB hierarchy (vRMM) or working on the page TLB miss path (SpOT).

CHAPTER 5

# Stressing the Limits of Memory as a File Interface

## 5.1 Overview

Persistent memory (PMem) is a new storage technology [119,186] that is connected to the system via the memory bus, like DRAM, and is accessible via CPU load and store instructions. The technology uniquely combines four characteristics: (i) scaling capacities, (ii) byte-addressability, (iii) latency/bandwidth close to DRAM, and (iv) non-volatility, blurring the decades-old distinction between slow but persistent storage and fast but volatile memory.

With PMem, storage accesses can be cheaper than OS invocations, so reducing the OS overheads is a strong requirement. The DAX (Direct Access) interface [4] can map persistent memory directly to user-space, enabling applications to access storage via regular load/store instructions. Multiple works [15, 74, 90, 125, 142, 162, 211, 219, 220, 221, 222] attempt to reduce PMem software stack overheads e.g., by optimizing file systems [67, 216], or designing them from scratch as PMem-aware to optimize metadata operations [125, 221, 222, 230]. This work focuses on a different part of the stack: the performance of *DAX memory-mapped file access.* We refer to this as the memory-mapped (MM) file interface.

Prior research focuses on MM overheads deriving from the until-recently necessary DRAM buffering of the data (page cache) [168, 170]. PMem and DAX-mmap remove this necessity, as *files are already stored in byte-addressable memory.* Despite the true zero-copy access that they provide, we find that memory-mapping can still be significantly slower than system call file access. The overheads have two sources. First, fast storage exposes the overheads of Linux

mmap operations. Second, direct access to fast storage enables new use cases for mmap; e.g. replacing read system calls with mmap operations for applications that access numerous small files (e.g., web and mail servers). Such new use-cases, change the traditional mmap workload and expose new overheads not previously important.

In this thesis, we analyze how Linux behaves when it maps files stored in PMem and observe that the interface's generic design often assumes that file mappings refer to DRAM resources. For example, all file mappings are populated lazily via page-faults and deleted synchronously to save scarce volatile memory. Such savings are irrelevant with PMem and DAX-mmap. In addition, hardware-maintained metadata – page access and dirty bits – target and drive efficient volatile memory management. By design they assist the selection of victim pages to reclaim page cache memory. With PMem and DAX, page cache management is no longer necessary. From this analysis, we identify multiple opportunities to remove unnecessary overheads targeting a design that comes close to the limits of what hardware can provide for MM direct access to byte-addressable storage.

We propose *DaxVM*, an efficient interface to byte-addressable storage, that extends the Linux virtual memory and file system layers. To reduce latency, DaxVM maintains shareable *pre-populated page tables* per file (file tables) and (de) attaches them to processes' address spaces during mmap. This eliminates paging costs and enables fast *O(1) operation* [201]. To improve scalability, DaxVM provides support for *ephemeral* file access patterns, e.g. opening multiple small files, reading/processing the data once, and closing them. Such concurrent access usually dictates the use of read/write system calls, as contention over virtual memory locks makes MM access prohibitive. DaxVM introduces a dedicated lightweight virtual address space manager that enables *ephemeral MM access* scaling to many cores. DaxVM also provides batched, *asynchronous unmapping operations* to minimize TLB coherence overheads. It also minimizes and potentially eliminates *kernel-space dirty tracking* overheads for applications that manage durability from user-space. Finally, DaxVM introduces *asynchronous background block zeroing* on PMem file systems to deal with the inherent double-writing costs of MM append operations.

DaxVM combines these techniques under a new *high- performance interface* for persistent memory operations providing new m(un)map calls. Separating the current unified volatile and non-volatile interface supports the observation that usage patterns for persistent and volatile data may differ substantially, enabling distinct optimizations.

Some of DaxVM's mechanisms are inspired by prior works; however, those works targeted different setups (e.g., block mapping for flash storage [112] or lazy unmapping for volatile memory [141]), required hardware extensions [112], or described high-level ideas [201, 223]. Our work combines these into an interface to persistent memory, implements them in a real operating system—enabling us to study the complexity and the details of their realization and synergy—and evaluates them with commodity hardware.

Despite tailoring for byte-addressable storage and PMem, various DaxVM aspects, e.g., file tables and virtual memory scalability optimizations, are relevant for fast access to other high-performance storage mediums (Section 5.6).

We implement DaxVM in Linux 5.1.0 with the ext4-DAX [216] and we make it publicly available[1]. For multi-threaded workloads operating for short time intervals over multiple small files, e.g., Apache [2], DaxVM improves standard mmap performance up to 4.9×. It also reverses the trend that favors read for such setups, outperforming it by up to 1.5×. It comparably boosts the performance of other applications with ephemeral access patterns that do not move data out of PMem (e.g., text search like ag [18]),. DaxVM also increases system availability, providing fast boot times for PMem databases [16]. It can finally provide up to 2.95× better throughput than baseline MM for PMem-optimized key-value stores [118] running on a fragmented ext4 image. This chapter makes the following contributions:

- It details the inherent costs of MM file access and we identify virtual memory features that assume data are always buffered in DRAM. We show how byte-addressable storage attributes can be leveraged to control the costs.
- It integrates file tables with a well-tested kernel-space file system (ext4) and show how they can eliminate paging costs via O(1) mmap. We study the address translation overhead implications of placing page tables on a slower medium (PMem) and show mitigations.
- It exploits the potentially ephemeral lifetime of DAX mappings and their access characteristics for fast address space (de)allocation and lazy unmappings, significantly improving DAX MM scalability to many cores.
- It shows that block pre-zeroing is an inherently different requirement for MM access than write syscalls which undermines MM benefits. We demonstrate how asynchronous pre-zeroing in the file system removes this cost.
- It shows that kernel's dirty page tracking harms performance even for applications that manage durability from user-space, and enable bypassing these costs.
- It shows that a dedicated interface for DAX mappings unleashes optimization opportunities not possible with POSIX strict semantics.
- It combines all the above in DaxVM, an interface close to the limit of what hardware can provide for MM access. We provide an end-to-end implementation in Linux and evaluate it on a real system.

[1]DaxVM is available at `https://github.com/cslab-ntua/DaxVM-micro2022`

## 5.2 Background

**DAX** [4] mechanism enables PMem file access without buffering data through DRAM (e.g., copying files pages in page cache). With DAX-mmap, virtual pages are directly mapped to PMem physical locations. The OS virtual memory subsystem creates virtual-to-pmem translations and persistent data can be accessed via load/store CPU instructions.

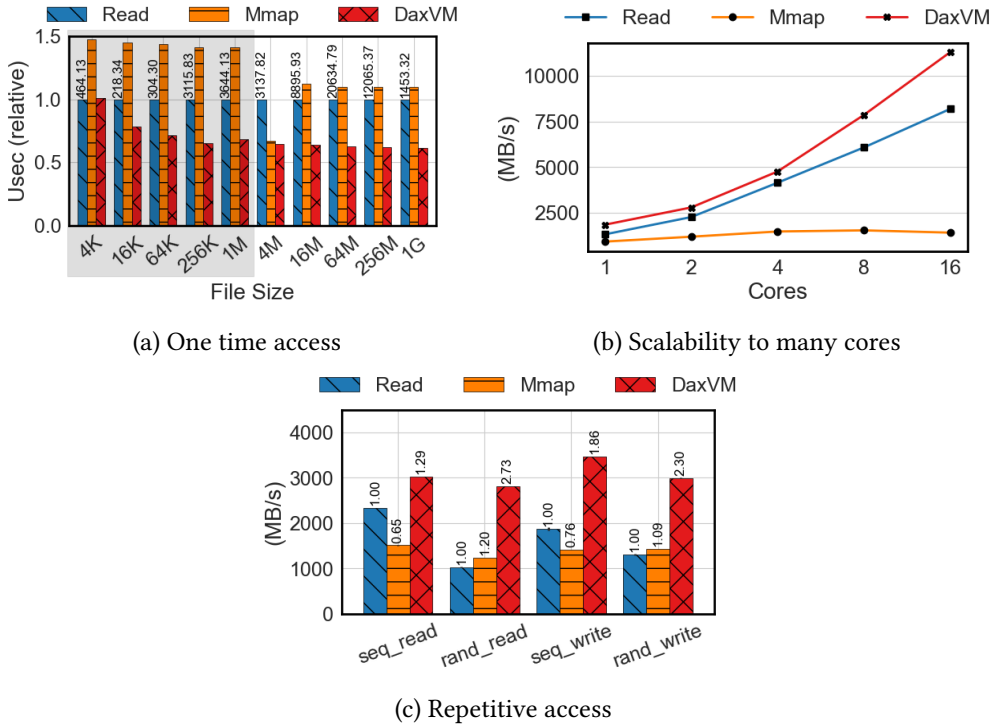
**Memory-mapped and system call file access.** Intuitively, memory mapping files holds important advantages compared to system call access (read/write), even for traditional block storage. Memory mapping files spares crossing the user/kernel boundary on multiple same file access, and always avoids at least one data copy. However, with block devices, file data must still be copied from the device to the volatile page cache to be mapped. In addition, the avoided extra copy (compared to read/write) is relatively cheap; from page cache to private per-process memory locations. Hence, for decades the guidance has been to use memory-mapping over block devices *only* when files are big and accessed randomly or multiple times [205].

PMem and DAX mappings offer true zero-copy storage access for the first time. However, with the faster and direct access storage path, the performance bottleneck moves from the device latencies to the software stack, with the complex and prohibitively expensive virtual memory operations being the primary source of overhead. In this paper we seek to study how this affects the decades-old trade-off between memory-mapped and system call file access.

## 5.3 Memory as File Interface

In this section we study how DAX zero-copy memory-mapped file access performs compared to system call access. We aim to understand the inherent overheads of file mapping. We run experiments on a system equipped with 384GB Intel Optane DCPMM (PMem) in AppDirect mode with an aged ext4-DAX file system using micro-benchmarks (Section 5.5).

**One time access.** First, we examine the latency of accessing multiple files once: opening them, reading their content and closing them. This is a common access pattern in server workloads (e.g., web servers, mail servers). For memory-mapped access we measure the latency of mapping a file (mmap), access all its data in-place at 8-byte granularity (sum them) and then unmap it (munmap). For system call access, we read the file data into a private buffer (read) with one call and consume them similarly. Figure 5.1a shows the latency results for a single thread as a function of the file size. We plot the average of reading up to 50K files or as can fit on 100GB of storage. For small files (shaded), memory-mapped access is significantly slower than read (up to ∼30%) despite the avoided copy. We refer to this as the *small files problem*. For larger files, the performance of memory-mapped access depends heavily on the number of huge persistent pages that back the mapped file. For files close to 2MB, memory-mapped access performs significantly better than




(a) One time access

(b) Scalability to many cores

(c) Repetitive access

Figure 5.1: DAX interfaces: (a) the latency of reading a file once via MM is worse than read system calls, especially for small file sizes (lower is better), (b) MM read-once access does not scale to many cores (higher is better), (c) MM repetitive access on a large file can perform worse than read/write (higher is better). All results are from a system equipped with Intel's Optane DCPMM and an aged ext4-DAX [216] file system image (Section 5.5). DaxVM significantly reduces latency and improves scalability for MM, regardless of the file system fragmentation.

system call access as files are 100% huge page covered. However, as the file size increases the performance drops non-deterministically, depending on the mix of small and huge pages that back the file, due to fragmentation on the aged ext4-DAX system [124]. For example, memory-mapped access performs ∼10% worse than read for 1GB file on this run. We did not consider 1GB huge pages in our experiments.

Next, we focus on throughput, and perform the same access pattern but over 32KB files using multiple threads. Figure 5.1b shows the operational throughput. As thread count increases, memory-mapped access does not scale to many cores, corroborating a known problem [59,71,72].

**Repetitive access.** Figure 5.1c shows the operational throughput of another common file access pattern, that of repetitive operations over the same large file (e.g., databases). We issue sequential and random 4KB reads and writes over a 100GB file. For memory-mapped access, we initially map the entire file and use memcpy() [124,125,220] with AVX512 instructions [149,231] and non-temporal stores [215] to perform reads and writes [124, 125, 149, 220]. We observe that

| | FlashMap [112] | O(1) [201] | Merr [223] | ctFS [149] | **DaxVM** |
|---|---|---|---|---|---|
| PMem storage | | ✓ | ✓ | ✓ | ✓ |
| Real OS implementation | ✓ | | ✓ | ✓ | |
| Commodity hardware | ✓ | | | ✓ | ✓ |
| O(1) mmap | ✓ | ✓ | ✓ | | ✓ |
| Scalable mmap | | | | | ✓ |
| Fast unmap | | | | | ✓ |
| Per-process permissions | ✓ | | ✓ | ✓ | ✓ |
| Dirty-page tracking avoidance | | | | | ✓ |
| Asynchronous block pre-zeroing | | | | | ✓ |

Table 5.1: Comparison of DaxVM with prior works that focus on memory mapping storage.

memory-map performs equally (for random access) or even worse than system calls (for sequential access) [84].

We now discuss inherent memory-mapped access overheads and compare them to system call behavior. We examine how PMem attributes affect some of the current OS design assumptions. We discuss prior work and seek ways to minimize the overheads; we target a design to stress the limits of memory as a file interface.

### 5.3.1 Virtual Memory Overheads

Memory mapping files inherently involves virtual memory operations that are costly.

#### 5.3.1.1 Paging

Memory-mapped file access requires a page table entry (PTE) for each mapped page of the file. Linux populates virtual mappings lazily, adding the cost of a synchronous page fault to create PTEs at page access time.

**DAX impact:** With block devices, file content is page cache buffered so faulting is important for fine-grain volatile memory management. With PMem, moving data between storage and DRAM is unnecessary as files are already in a byte-addressable medium. Also, the entire set of physical translations is known upfront and changes slowly as it reflects storage locations.

**Prior works:** Multiple works [112, 149, 191, 201, 219, 223] leverage the concept of page tables maintained by the file system to translate file offsets to PMem physical locations. O(1) memory [201] suggests the high-level idea of sharing them among processes to eliminate paging, following an older proposal for flash SSDs [112], but the design is only discussed at a conceptual level. MERR [223] emulates O(1) operations for fast address space randomization and requires

special hardware to support per-process permissions. SIMFS [191] implements some key concepts but fails to support file sharing with different permissions and requires a pre-set global maximum file size. All the above also emulate persistent memory so the performance of persistent page tables remains unclear. Finally, ctFS [149] integrates file tables to a FS that maps the entire DAX device to user-space, trading secure (meta)data operations for fast appends. Table 5.1 summarizes the limitations of prior works and the key aspects of the DaxVM approach.

**DaxVM approach:** We provide flexible O(1) memory mappings via *pre-populated file tables* integrated on a well-tested kernel file system, without restrictions and using only existing hardware on a real system. We manage the potential overheads (e.g., TLB miss costs) of shared page tables residing on persistent memory.

#### 5.3.1.2 Virtual address space management

File mapping requires (de)allocating an area in the process address space to (un)map the file. Operating systems serialize address space operations (e.g., virtual address allocation), limiting manycore scalability of virtual memory. For example, Linux protects the entire virtual address space of a process with a semaphore (mm→mmap_sem) [80].

Linux also records all allocated virtual memory areas (VMA) in a centralized data-structure (the VMA red-black tree). This fine-grain recording enables the support of a variety of POSIX memory operations (partial munmap and mprotect, etc.). However, it induces significant lock contention when VMAs live briefly (are quickly unmapped). Applications that access many small files once before closing (e.g., web servers, mail servers) issue frequent (un)map requests but rarely any other memory operations. On the other hand, applications that repetitively access files (e.g., databases) commonly use complex operations (mremap, etc.).

**Prior works:** Clements et. al [70,71] proposed using concurrent data structures to enable simultaneous operations on different address ranges. However, a relevant Linux implementation [80] showed that the transition is not trivial because the contention may be transferred to range locks as memory operations commonly affect multiple ranges [140].

**DaxVM approach:** We focus on file mappings only, and differentiate the address space (de)allocation scheme based on the expected mapping lifetime and the required operation support. We provide lightweight file mappings that trade complex memory operations support (e.g., partial mprotect/mremap) for scalable virtual address space (de)allocation operations.

#### 5.3.1.3 Synchronous resource release

File unmapping releases virtual addresses and requires maintaining virtual memory coherence. POSIX dictates that the release occurs synchronously, i.e., before the operation returns. This requirement for synchrony requires clearing PTEs and invalidating corresponding TLB entries in local and remote cores (shootdowns). If stale translations remain in the TLB, an application could access reclaimed physical memory, raising correctness and security issues. Shootdowns are inherently non-scalable operations, requiring synchronous inter-processor interrupts (IPIs) [48] that cost up to thousands of cycles [35].

**DAX impact:** With block devices, unmap operations also potentially release physical resources (page cache) under memory pressure. PMem mappings no longer occupy volatile memory; thus, unmap operations release only virtual addresses. The mapped persistent memory is independently and exclusively reclaimed by specific file system operations (e.g., when files truncate) whose frequency is relatively low [42].

**Prior works:** To avoid scalability overheads, state-of-the-art PMem user-space filesystems, never unmap the files [125, 149]. For example, SplitFS [125] maps files under the hood and keeps them mapped to user-space until the process dies or the files get truncated. This extreme strategy raises safety concerns [223]. LATR [141] on the other hand, uses a message-passing mechanism in place of IPIs to invalidate TLBs locally and asynchronously [72], but in very short time intervals as it targets volatile memory. This general-purpose mechanism is complex, error prone [35], and still suffers from scalability issues due to its own locking (Section 5.5).

**DaxVM approach:** We focus on file mappings only, and take advantage of the fact that the mapped physical resources (storage) are reclaimed slowly and independently. We opt for a simple design of lazy unmapping; we still unmap files but asynchronously and use existing robust mechanisms to achieve that.

#### 5.3.1.4 Dirty page tracking for file syncing

With block devices, both system call and memory-mapped write access is buffered in volatile memory and the corresponding pages are tagged as dirty in the OS page cache metadata tree, to be flushed to disk during sync (e.g., fsync).

**DAX impact:** Persisting data requires just writing-back dirty CPU cache lines. Thus, DAX write system calls commonly bypass the CPU caches using non-temporal store instructions to copy data to the device (e.g., ext4-DAX, NOVA), omitting the need of any dirty tracking.

With DAX memory-mapped access, however, the kernel still has to record the physical regions that user-space dirtied, to be able to flush the corresponding CPU cache lines on sync operations. Thus, the OS still uses the page cache tree to tag dirty pages when DAX-mmap is

used. While x86 hardware page walkers set the PTE dirty bit when a page is first written, Linux also tracks mapped dirty pages in software. It initially marks pages as read-only and relies on permission faults to detect writes and update the page cache tree [6, 181, 240]. Sync operations write-protect file pages again for all mapping processes after flushing, to restart the mechanism. We measure that performing one msync call every 10 write operations (random access, 1KB each) on a memory-mapped 10G file causes ∼2.8x more faults compared to no sync.

**Prior works:** DAX mapping allows applications to manage data durability from user-space, omitting sync system calls. Prior works actually recommend this approach [125, 149, 220]. The application can use non-temporal stores or cache line flush instructions (e.g., clwb and sfence) over DAX mappings to persist file data at the granularity of bytes. However, the OS in those works still tracks each initial dirty page access and remains oblivious to user-space managed endurance. In this way, the system remains compatible with sync operations but pays all the overheads of dirty-page tracking.

**DaxVM approach:** We drop all kernel-space dirty page tracking activity for applications that manage durability from user-space, to achieve maximum performance.

### 5.3.2 Double writing for secure appends

Append operations are write operations that involve block allocations by the file system. To append a file via MM requires to allocate first new blocks (e.g., via fallocate) and then map them to user-space for write access.

**DAX impact:** Append operations via direct access file mappings can potentially leak information if storage blocks with stale data (e.g. blocks that deleted files previously occupied) get directly mapped to use-space without being cleared. To address this issue and secure MM appends, PMem file systems zero-out blocks during fallocate operations. This necessary block zeroing, introduced by DAX, doubles the written bytes per MM append operation penalizing performance. We measure that 30-40% of DAX MM append operation latency (fallocate()+mmap()+memcpy() sequence) is spent in block zeroing, irrespective to the append size.

Appends through DAX write system calls update directly block content and thus it is not strictly necessary to zero-out the blocks that get allocated in the process to secure operations. In example, NOVA and PMFS do not zero out blocks during append system calls, they just overwrite blocks stale data with non-temporal stores. Ext4-DAX, though, zeroes out blocks on DAX append write system calls as well, to fortify against races between DAX write and DAX mmap calls that could again end up in leaking stale data via DAX mmap.

**Prior works:** The exact same security concerns exist for volatile memory, and multiple works [165, 179] propose asynchronous page pre-zeroing to control allocation overheads. For storage, asynchronous zeroing is not that simple as it can consume the available bandwidth and stall concurrent requests to the device. It is only considered for SSDs (garbage collectors) due to their erase-before-write NAND nature and multiple works attempt to mitigate GC overheads [128].

**DaxVM approach:** Inspired by volatile memory strategies and harnessing the high PMem bandwidth we adopt asynchronous pre-zeroing in PMem file systems.

### 5.3.3 Micro-architectural performance

Memory-mapped and system call file access behave differently at the micro-architectural level. These are fundamental observations about CPU and OS operation that we cannot work around.
**TLB performance.** Linux maps the entire PMem physical space with huge pages. Thus, the internal copy of a read/write call benefits from reduced TLB misses, even when files are $<$2M or fragmented. But, small file MM access always pays small page TLB miss costs and large file performance depends heavily on the file system fragmentation and the ability to use huge pages.
**Cache performance.** System calls copy data, which pre-fetches persistent data into higher layers of the cache hierarchy. User-space code runs faster hitting in the caches. With memory-mapped access the user-space code will pay the cost of fetching data from persistent memory.
**Vectorization.** User-space memory-mapped access can use Advanced Vector Extensions [115], to perform SIMD operations over file data (e.g., memcpy). This can significantly improve performance [96, 149, 231]. Copies inside kernel system calls cannot benefit from AVX instructions as supporting them would introduce register save and restore overheads when crossing the user-kernel space boundary.

## 5.4 DaxVM

Based on the observed opportunities we design and implement DaxVM, a fast and scalable MM interface aiming to come close to the limit of what hardware can provide. DaxVM extends the OS memory-management and file system layers and consists of five key components.
**1. Fast paging operations through pre-populated file tables.** With DaxVM, the file system maintains pre-populated page table fragments that translate file offsets into storage physical addresses. These *file tables* are attached/detached to processes page tables during m(un)map operations, eliminating the paging setup and teardown costs of DAX mappings.
**2. Scalable address space management for ephemeral mappings.** DaxVM maintains a dedicated heap to serve fast (de)allocation requests for *ephemeral mappings*. These mappings are expected to live for short periods and support no other operations (e.g., protection change).

**3. Asynchronous resource release; batching unmap requests.** If application correctness does not rely on synchronous unmapping, DaxVM's virtual memory layer can *optionally* defer munmap operations. It tracks the *zombie* mappings and releases them asynchronously in batches. This eliminates frequent fine-grain TLB shootdowns.
**4. Low durability cost.** DaxVM provides a mode that drops all kernel-space dirty tracking for applications that manage durability in user-space and opt for maximum performance.
**5. Asynchronous storage block pre-zeroing**. DaxVM extends PMem file systems to asynchronously zero out storage blocks when they are freed (e.g., unlink, truncate).
DaxVM comes as a *new interface*, with stripped-down POSIX features and relaxed restrictions targeting performance (Section 5.4.6). It adds two new system calls: daxvm_mmap and daxvm_munmap along with new optional flags.

DaxVM speeds applications that perform frequent m(un)map operations, e.g., briefly process small files, applications sensitive to paging (e.g., databases), and allocation-intensive workloads especially on fragmented FS images.

We implement and evaluate DaxVM in Linux 5.1 and the ext4-DAX [216] and NOVA [221] file systems. We target the x86-64 architecture. DaxVM primarily targets DAX-aware file systems that relax data operation atomicity for performance (e.g., NOVA relaxed [221], xfs-DAX). They allow in-place updates on DAX mappings. Atomic copy-on-write updates, e.g., shadow paging, negate DaxVM benefits with frequent page table updates.

### 5.4.1 O(1) mmap

DaxVM maintains pre-populated page tables per file (file tables), and (de)attaches them to process address spaces during m(un)map operations. This eliminates paging costs and provides instant access to files irrespective to their size (O(1) operation). Prior work [201] discussed O(1) memory conceptually without any implementation, while others simulated its functionality [223]. Here we focus on designing and implementing O(1) performance in a real system.

#### 5.4.1.1 Pre-populated File Tables

They are fragments of an x86-64 page table (radix tree) and translate file offsets to PMem addresses. For example, if a 1MB file is stored in pages P1-P256 of PMem, the file table stores the addresses of P1-P256 starting from index 0.
**Maintenance.** Upon storage allocation (e.g., append), DaxVM populates the file's tables with the physical addresses of the newly allocated PMem pages. Upon storage de-allocation (e.g., ftruncate), DaxVM clears the file table entries and/or frees the corresponding file tables.
**Fragments instead of entire trees.** Files usually shrink/grow sequentially and densely at the end of the file. Unlike the sparse population of virtual address spaces, this characteristic makes

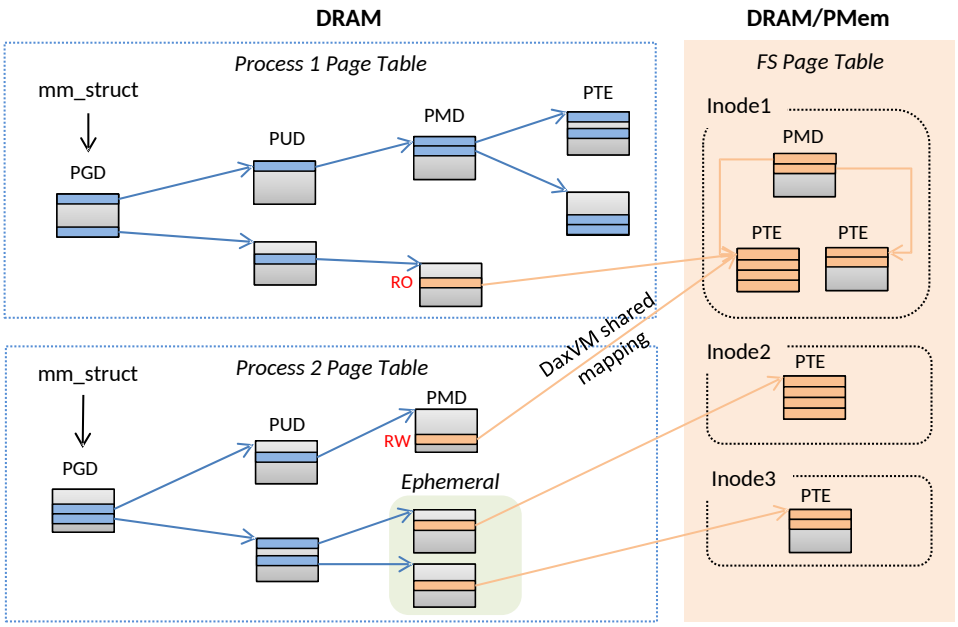



Figure 5.2: DaxVM maintains pre-populated shared file tables and attaches them to processes address spaces for O(1) mappings.

it possible to build file tables in a bottom-up fashion. For small files, we use a single 4KB page of PTEs, and expand in 4KB increments. In Figure 5.2 only the PTE level of the radix tree is needed to hold the translations of inode 2. This bottom-up maintenance controls file tables storage tax.

**Huge Pages**. For aligned huge page blocks in larger files, DaxVM supports huge PMD entry formats. To simplify the description, we consider the general 4KB block condition.

**PTE status bits.** Page table entries also maintain status bits to record per-page process access. Surprisingly, we find that most of these bits track metadata mostly relevant to volatile memory management: the access and dirty bit are mainly used for page cache evictions and volatile memory reclamation. DaxVM drops their maintenance in the file table entries, as reclamation happens explicitly during file delete for DAX mappings. DaxVM sets the PTE permission bits to maximum and supports per-process access at 2MB or coarser granularities. Similarly, it manages durability at coarser granularities. We discuss both in the next section.

**Dynamic File Table Management.** Tables can be stored both in DRAM and PMem.

*Volatile file tables* are re-constructed each time the system accesses an inode for the first time, loading it to the VFS inode cache (cold open). The table's root pointer is stored as metadata in VFS inodes. As long as the inode is cached, the tables are updated. When the inode is evicted from the VFS cache, they are destroyed.

*Persistent file tables* are stored in PMem pages and survive across power cycles/failures. Their root pointer is stored as metadata in the file's permanent inode struct. During table updates, the table entries must be flushed from the CPU caches synchronously to guarantee persistence. To

| Benchmark | DRAM file tables | PMem file tables |
|---|---|---|
| seq_read | 28 | 103 |
| rand_read | 111 | 821 |

Table 5.2: Average page walk cycles measured when file tables are stored in PMem or DRAM. We consider sequential and random 4K access on a 10G memory-mapped file.

| Performance Monitor | |
|---|---|
| AvgPageWalk | Total Page Walk Cycles / Number of TLB misses |
| MMU overhead | Total Page Walk Cycles / Execution Time Cycles |
| Rule | if (AvgPageWalk $>$ 200 c) and (MMU overhead $>$ 5%) migrate |

Table 5.3: DaxVM monitors the average TLB miss costs and MMU overheads to migrate file tables to DRAM if necessary.

control this overhead, DaxVM leverages that multiple PTEs are usually updated sequentially within a single operation (e.g., append). When possible, it batches their flushes at cache-line granularity (8 64-bit PTEs in x86_64).

Persistent tables occupy storage resources but provide good cold-start performance and save DRAM as they substitute parts of multiple processes' page tables. Apart from the storage cost, they introduce higher TLB miss costs, as page table walkers have to access slower memory [26]. Table 5.2 shows the average page walk latency measured with *perf* when we perform sequential and random reads on a file mapped using volatile and persistent file tables. We observe that with random access and persistent page tables, TLB misses can cost up to 800 cycles. On the other hand, keeping all file tables always in DRAM can lead to waste of resources, while aggressively reclaiming them can penalize performance as they will have to be re-constructed frequently.

To keep the best of both worlds, DaxVM maintains volatile tables for files smaller than a threshold (32KB) and persists them for larger. This policy controls the storage tax which is high for small files (e.g., for every 4KB file a 4KB PTE is allocated). DaxVM also monitors the MMU performance of applications via performance counters [165]; it tracks the average page walk latency along with the average time spent in page walks (MMU overhead) (Table 5.3). If the latency is above 200 cycles and 5% of the execution time is spent in walks, DaxVM (i) builds asynchronously volatile tables (copying the persistent ones) and (ii) walks the process tables to detach the persistent fragments and attach the new volatile. After DaxVM migrates file tables to DRAM both volatile and persistent tables are maintained.

**File Tables and Crash Consistency.** For journaling systems, e.g., ext4, file tables are updated within a journal transaction. When the transaction is committed, the tables are guaranteed to be consistent and persistent. Similarly, for a logging FS like NOVA, file table updates happen before

the (meta)data updates are committed (log entry appended). File table PTEs are flushed on write and re-use the fence from the FS log/journal commit. Incomplete PTEs are recovered on reboot when replaying open transactions. The overhead of persisting file tables is included in all our experimental results.

#### 5.4.1.2 Fast table (de)attachment

DaxVM uses file tables to minimize the cost of creating a mapping. When an application maps a file, DaxVM populates all translations for the requested target file offset (i.e., mmap-populate). It attaches parts of the pre-populated file tables to the process's private page table. Figure 5.2 depicts how DaxVM builds a page table in DRAM (blue) up to the PMD level. Then it attaches the pre-populated file PTE (orange) on the PMD. Attachments enable O(1) mmap operations; the latency is near constant with respect to file size.

**Mapping size.** Attaching a file table's fragment updates interior pointers at some level of the process's private page table radix tree. Therefore, the attachment can happen only at certain granularities/levels, i.e., at PMD, PUD, etc. In addition, the mapping's virtual address and the corresponding file offset must be properly aligned, i.e., to 2MB for PMD.

To enable O(1) mmap, DaxVM silently rounds the size and file offsets attributes of the daxvm_mmap system call to the granularity of the next level of the process's page table tree. Up to 1GB, files are mapped using PMD entries at 2MB granularity, and files above 1GB are mapped using PUD entries at 1GB granularity. This leads to the anomaly that mapping files >1GB can be faster than smaller files.

DaxVM returns to the user the virtual address that maps to the requested file offset. A larger portion of the file may be silently mapped (before/after the requested boundaries).

**Permission rights per process.** With DaxVM, the pre-populated file page table fragments are *shared among the processes that map the same file.* The pre-populated PTEs have the maximum access rights pre-set. To enable different access permissions per process, DaxVM manages the permission bits at the attachment level (e.g., the PMD) rather than at the PTEs, as the former belongs to the private part of each process's page tables. Figure 5.2 shows how two different processes have read-only and read/write access rights over the same 2MB file region while still using the shared file tables. The x86 translation hardware (page walker and TLB) applies the minimum access rights found at all the page table levels for an address, enabling this strategy [123].

**Copy-on-Write and Fork.** Private mappings initially share data and mark all PTEs as read only, relying on CoWs. To support private mappings, DaxVM initially attaches shared file tables in the process table tree, On a CoW fault, DaxVM copies the file data and also copies the corresponding file table leaf(s) into volatile private page(s). At process creation DaxVM sets the child process

page tables to point at the file's shared file tables (shared mappings). For private mappings it follows a same mechanism as CoW. This preserves O(1) mmap for the child.

**ASLR.** With DaxVM, the address space layout randomization works seamlessly at 2MB granularity. File tables are attached to randomly allocated virtual addresses aligned to 2MB. The file data will always be located at the same offset within the random 2MB region (alignment restriction).

**FS extensions.** To support O(1) mmap a file system must be extended to (de)construct and update file tables during storage block (de)allocations and to attach them during mmap.

#### 5.4.1.3 Virtualization

In this paragraph we discuss how O(1) mmap integrates with persistent memory virtualization in the current design.

**Persistent Memory Virtualization.** As discussed in Chapter 4, three sets of page tables exist with hardware assisted virtualization: (i) the guest page tables per process running inside the virtual machine, (ii) the host page tables for the VMM host process (e.g. qemu) and (iii) the extended/nested page tables maintained by the hypervisor (kvm) per VM. O(1) mmap can be used by any process running inside the virtual machine to memory-map a file residing in a virtual persistent memory device exposed to the VM. Pre-populated file tables will be attached to the private guest page table tree of the process and most of the guest paging overheads will be eliminated (as in native execution). If the back-end of the virtual device is a file residing in persistent memory or a raw DAX device [23], the VMM can use DaxVM O(1) mmap to map it. The pre-populated tables will be attached to the VMM host process private page table tree, eliminating host paging. However, the current design does not include any DaxVM extensions (e.g. extended pre-populated page tables) that could be leveraged to eliminate nested paging VM-exits and accelerate persistent memory virtualization itself. We plan to explore DaxVM virtualization in the future. Currently the attached tables on the VMM host page table tree just accelerate the VM exit routine, that would otherwise have to execute host page fault code.

### 5.4.2 Ephemeral mappings

As discussed in Section 5.3, the centralized locking of a process's virtual address space prohibits issuing parallel frequent m(un)map operations as they cannot scale to many cores [60]. This almost excludes MM as an interface for a common file access pattern: open a file, quickly process its data and close it. An old study on distributed file systems shows that 75% of files are open for less than a quarter of a second [42]. We refer to this as *ephemeral file access*, and DaxVM provides a dedicated address space manager for *ephemeral mappings* of persistent memory files. It builds its strategy for better scalability on the idea that such mappings do not require support for complex virtual address space operations beyond unmap.

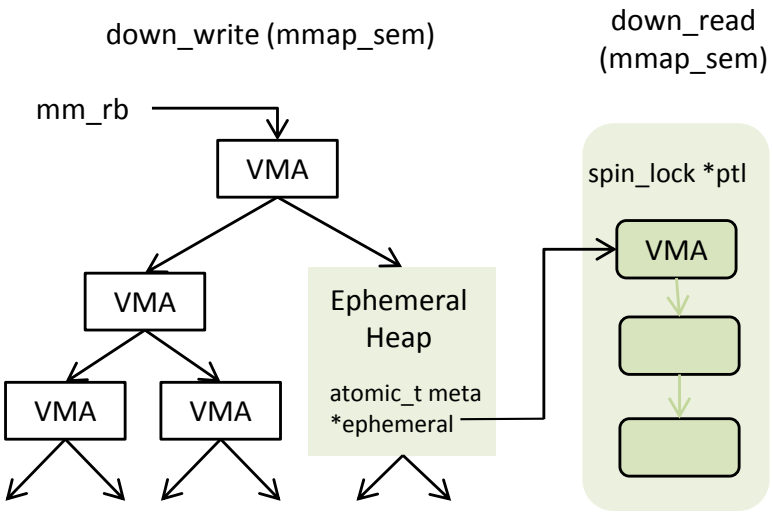



Figure 5.3: DaxVM ephemeral VMAs.

**Ephemeral heap.** DaxVM pre-allocates a virtual address range (*ephemeral heap*) in the process's address space and manages it independently to (de)allocate virtual address regions for ephemeral mappings. The allocator's objectives are similar to a user space heap allocator (e.g., malloc()): to quickly allocate and free address regions. It does not support splitting and merging of mappings.

Our current heap implementation leverages short mapping lifetimes to perform linear allocations. The heap is dynamically extended in virtual regions of 1GB, to avoid exhaustion. Each region's virtual addresses are reclaimed only when all the mappings populating it are destroyed; tracked by a counter. Thus, currently allocations from the ephemeral heap resemble a stack; but other allocation schemes can be applied.

**Ephemeral mapping visibility and tracking.** Only munmap operations are allowed for ephemeral mappings; any other operation (mprotect, mremap, etc.) that falls inside the heap's range returns an error. As complex per-mapping support is omitted, ephemeral VMAs do not need to be recorded by the virtual memory's core data structures, i.e., the VMA red-black tree (`mm_rb`). It is sufficient that the manager records only the aggregate ephemeral heap region. This enables tracking ephemeral VMA's in a dedicated data structure, a list (or a tree) associated only with the heap (Figure 5.3).

The major advantage of this design is that lightweight locking can be used to protect this structure, avoiding contention over the global manager's locks. In Linux, the entire address space of a process is protected by the heavily contended mmap semaphore [80], which mainly protects the VMA tree. Table 5.4 summarizes the main code paths that contend for the semaphore for insight. DAX mappings inherently aren't involved in paths that target volatile memory management (set A). On top of that, DaxVM ephemeral mappings do not fault often (only for dirty page

| | Path | Target | Reader/ Writer | DAX Mappings | Ephemeral Mappings |
|---|---|---|---|---|---|
| A | Khugepaged<br>Ksm<br>Mlock<br>Madvise<br>Mempolicy | Volatile Memory Management | R/W | ✗ | ✗ |
| B | Page Fault | Populate Mapping | R | ✓ | ✗ |
| C | Mremap<br>Mprotect<br>Exec | Resize mapping<br>Change Perm<br>Set up binary | R/W | ✓ | ✗ |
| D | Mmap<br>Munmap<br>Fork<br>Msync | Create Mapping<br>Dissolve Mapping<br>Duplicate mm<br>Flush dirty pages | R/W | ✓ | ✓ |

Table 5.4: Paths acquiring the mmap semaphore and their involvement in DAX and ephemeral mappings management.

tracking) (set B) nor support memory operations (set C). This leaves mainly m(un)map to contend for the semaphore (set D); simplifying the design of a more scalable address space manager.

We use atomic operations to update heap's metadata and a spinlock to protect the ephemeral VMA list. Heap (de)allocations hold the mmap semaphore as readers. The idea of VMA locks instead of a global semaphore has been discussed [75], with the concern that they could lead to contention for one big VMA lock. In our design, ephemeral heap locking scales because the operations that take place under the lock are stripped down and fast.

Ephemeral VMAs are still visible to the file system. They are attached to the address space trees that track the VMAs that map each file (address_space→i_mmap). This enables their management (e.g., unmapping) by the file system.

### 5.4.3 Optimized munmap

Unmapping a virtual region involves three steps: (i) clearing/destroying the page tables, (ii) invalidating the local and remote TLB entries (shootdowns) that cache the region's PTEs, and (iii) releasing resources. DaxVM detaches file tables instead of destroying them.

**Async unmap.** TLB shootdowns are inherently non-scalable as they require IPIs. Linux batches the virtual addresses of a single munmap request to perform a cheaper range TLB invalidation (one IPI) instead of individual page shootdowns. After a certain threshold (33 pages for x86), it

opts for a full TLB flush as the gains of the flush are estimated to outperform the penalty of the TLB misses introduced.

DaxVM builds on this strategy and gives the option to *not perform munmap operations synchronously* at all. It records the VMAs that the user requested to unmap, the now "zombie" VMAs, and defers their unmapping to batch TLB invalidations *across requests*. It tracks the total number of zombie pages and when a threshold is reached, it tears down their corresponding page table entries and performs a single full remote TLB flush on the cores that the application runs. It does so on the munmap request that exceeds the threshold. Apart from the key advantage of replacing frequent TLB invalidations with fewer, cheaper, entire TLB flushes, the virtual memory locks are also held for shorter periods.

**File system races.** While an unmapping is deferred, the size of the mapped file may change if the file gets truncated or even deleted. DaxVM maintains safety by synchronously forcing unmappings if storage blocks are reclaimed.

### 5.4.4 Durability management

DaxVM fully supports msync and fsync calls in the same way as default DAX through permission faults. DaxVM tracks dirty regions at 2MB or coarser granularities, as access permissions are held at the attachment level of the file tables. For example, if a 4KB page is written, DaxVM will mark the entire 2MB region as dirty in the page cache. Note that the same happens if a huge page backs the file. This can potentially penalize fsync calls, but reduces dirty tracking overheads, as fewer permission faults take place (Section 5.5).

DaxVM does not require userspace durability management to work properly. But to further stress performance limits, DaxVM has a *nosync mode* for applications that manage durability from user-space [220]. In this mode, it does not track dirty pages via permission faults and does not record them at all in the page cache metadata tree. In a nutshell, it drops sync operation support (e.g., msync), which becomes a no-op, and data durability becomes entirely *a userspace responsibility*. This creates a race condition if a file is mapped via DaxVM and POSIX simultaneously: data modifications of the DaxVM mapping might not be captured by the msync() operations of the POSIX mapping. To manage this, DaxVM pushes the cost to the POSIX process, which flushes the entire file during its msync().

### 5.4.5 Asynchronous block pre-zeroing

DAX memory-mapped append operations – unlike system calls – inherently require the zero-out of the newly allocated blocks for security reasons, doubling the write activity and penalizing performance by ∼30-40% irrespective to the append size. DaxVM extends the file system to pre-zero blocks asynchronously to avoid this cost.

DaxVM does not interfere with the file system block allocator, to avoid inducing involuntarily external fragmentation to the system. With storage, external fragmentation matters in the granularity of extents – rather than pages – which can grow up to multiple MB. Instead, DaxVM hooks the file system's free operations. Upon a file truncate operation, the blocks to be freed are kept on per-core lists instead of being immediately released to the FS block allocator. A rate-limited kernel-thread periodically scans the lists and zeros-out blocks using non-temporal store instructions for persistence and to minimize bandwidth consumption [231]. Once a whole set of blocks-to-be-freed is zeroed, they are released to the allocator. Per-core lists preserve the scalability of free operations.

With PMem, more so than volatile memory, pre-zeroing consumes precious bandwidth and can potentially penalize other operations. To avoid BW saturation we throttle bandwidth to a configurable amount on an idle core.

### 5.4.6 DaxVM forms a new relaxed interface

Many of DaxVM's mechanisms relax some of POSIX strict requirements and abandon some POSIX functionalities, e.g., advanced memory operations support for all file mappings, to draw performance. Interfaces' impact on scalability and performance is a formally studied topic [72].

DaxVM interface consists of two new system calls (daxvm _mmap and daxvm_munmap). Daxvm_mmap implements O(1) file tables attachment and currently supports shared mappings. From the rest of the POSIX flags, DaxVM currently supports MAP_SYNC and adds three new flags.
**MAP_EPHEMERAL**: the mapping is expected to be brief and does not need any memory operation support. This flag activates the ephemeral address space allocator.
**MAP_UNMAP_ASYNC**: the program does not require access faults right after unmap. Activates asynchronous unmapping.
**MAP_NO_MSYNC**: this flag is combined with MAP_SYNC and means that the program will not rely on msync functionality at all. This flag activates the *no sync* mode, where all dirty page tracking is dropped and msync becomes a no-op.

We now discuss how DaxVM affects other operations.
**Memory protection.** Partial mprotect system calls over DaxVM mappings fail. DaxVM only allows changing the permissions for an entire mapping. Moreover, when the MAP_EPHEMERAL flag is set, any mprotect call fails.
**Mremap.** Similar to mprotect, DaxVM allows only mremap calls on the entire mapping (e.g., to resize) and fails if MAP_ EPHEMERAL is used.
**Madvise.** It is designed for volatile memory management (e.g., page cache), thus DaxVM does not support it.
**Msync.** DaxVM supports msync, unless MAP_ NO_MSYNC is used when it becomes a no op.

**POSIX comparison.** POSIX maps files in multiples of pages and references beyond the mapping's last page results in a segmentation fault. DaxVM guarantees that at least the portion of the user requested is mapped, but a portion before and after may also be silently mapped to the process address space (for proper alignment that enables O(1) mmap). If the file is extended inside this virtual portion, the new pages are automatically mapped to the address space. Moreover, POSIX promises synchronous unmappings. DaxVM relaxes that requirement, but guarantees that mappings are removed before physical and virtual resources are reassigned.

### 5.4.7 Discussion and summary

**Security and correctness.** Daxvm_mmap may map more of the file than requested. If entire file's content must not be visible to the calling process, DaxVM must not be used. Also, with MAP_UNMAP_ASYNC, user accesses to unmapped regions may not trigger an exception for a time window after a daxvm_munmap call. Correctness is not violated as DaxVM
guarantees that the virtual regions will not be re-used before the page table and TLB entries are invalidated. However, if an application depends on traps triggered by accesses to unmapped regions (e.g., userfaultfd() or guard pages), MAP_UNMAP_ASYNC should not be used. With respect to security, if an application expects attacks, e.g., untrusted code injection/execution, DaxVM increases the time that data are vulnerable, keeping them mapped for longer than expected. Note that some DAX user-space file systems (e.g., SplitFS [125]) map files under the hood indefinitely. Applications can limit this behavior omitting the MAP_UNMAP_ASYNC flag.

**Huge pages.** Currently Linux and various file systems try to control DAX paging overheads by backing files with huge (2MB or 1GB) pages. DaxVM supports large pages when present, harnessing their TLB performance advantages. However, huge pages are very sensitive to FS fragmentation [124], due to alignment restrictions, and cannot be used for files smaller than 2MB. For both cases, DaxVM eliminates paging and sustains high performance (more in Section 5.5).

**Programmability.** Applications must change to use DaxVM interface, replacing either a read system call or a POSIX mmap. Simple uses of mmap can be replaced directly, while reads should be replaced with daxvm_mmap and direct access to the data. MAP_EPHEMERAL is meaningful for files accessed briefly (i.e., once) and closed, but functionality does not break if used with mappings of longer lifetime.

**Applicability.** In a nutshell, DaxVM enables performant and scalable concurrent m(un)map requests and minimizes paging costs, unleashing DAX benefits also for applications that perform short-lived accesses to smaller files (a usage previously favoring read/write). DaxVM is still beneficial to applications that access files mapped for long periods (e.g., databases), especially on fragmented FS images, acting complementary to huge pages.

## 5.5 Evaluation

### 5.5.1 Experimental Setup

Our experimental platform is equipped with an Intel Xeon Gold 5812T Cascade Lake CPU with $2 \times 16$ physical cores, with frequency fixed at 2.7 GHz and SMT disabled. Each socket is equipped with 94GB DRAM and 384GB Intel Optane DCPMM (PMem) in AppDirect mode (3 DCPMM DIMMs). We limit our experimentation in one socket. To study interfaces performance under realistic file system conditions, we use the Geriatrix [126] tool to age the fs image. We use the suggested [124] Agrawal profile [28] and apply 100TB of write activity to PMem (70% utilization).

We implement DaxVM in Linux kernel v5.1.0, incorporating it with ext4-DAX [216], NOVA [221], and the core virtual memory manager. We use a set of micro-benchmarks and real-world workloads to evaluate DaxVM in relation to (i) system call file access (read and write) and (ii) the default DAX-mmap interface. Due to space limitations, our evaluation focuses on the commonly used ext4-DAX FS. We discuss where results differentiate significantly with NOVA. We use the *nosync* mode when applications enforce durability from user-space. For Linux mmap we consider both lazy page faulting and pre-faulting (MAP_POPULATE flag – populate). We also provide some comparison with an asynchronous unmapping technique (LATR [141]). We run experiments three times and plot the average.

### 5.5.2 Micro-benchmarks

Because there is no standardized benchmark to profile file memory mapping and compare with read/write file access [159, 203], we construct our own set of micro-benchmarks. We revisit the same experiments as in Figure 5.1; we consider: (a) accessing files once – *ephemeral access* (e.g., webservers) and (b) accessing files repetitively (e.g., databases). We use AVX-512 instructions and non-temporal stores for user-space write access [231].

**Ephemeral access.** We open 50K files (or 100GB/filesize for >2MB files), briefly process their content, and close them. For memory-mapped access, we map each file, access its data in-place at 8-byte granularity, sum them and then unmap the file. For read, we read the entire file with one system call into a private buffer and then process its data similarly.

Figure 5.4 reports throughput (MB processed/second) relative to read for a single thread and as a function of the file size (Figure 5.1 shows latency). For small file sizes (shaded), mmap performs $\sim$ 20% worse than read despite avoiding data copies due to paging. Pre-faulting (Populate) improves performance, as the file size increases, but does not entirely solve the problem; it still pays the cost of (de)constructing page tables. DaxVM improves throughput by up to 50% over read for small files eliminating paging with O(1) mmap operations.

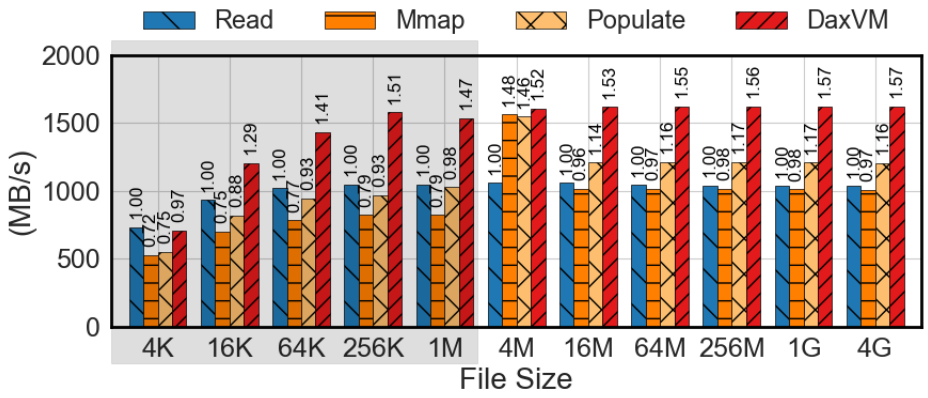



Figure 5.4: Read-once (ephemeral) file access.

For larger files, baseline memory-mapped access is heavily affected by huge page coverage; reporting better performance for file sizes close to 2MB (e.g., 4MB). As the file size increases though, performance drops further and becomes non-deterministic due to the increasing number of small pages involved in the file's mapping from a fragmented FS. DaxVM's file tables provide an almost robust 55% benefit over read and 30-50% over mmap, independent of the FS fragmentation.

**Repetitive access over large files.** We consider the case of memory mapping a 100GB file and use memcpy to perform 1KB and 4KB reads and overwrites in sequential and random order. This microbenchmark [124] mimics database operations [220]; a use-case favoring memory-mapped access as it avoids the significant cost of crossing the user-kernel boundary frequently [46]. Figure 5.5 summarizes our results.

For 1KB access all mmap interfaces outperform read/write access. Notably though, default mmap performs only 11% better than read for sequential access, despite avoiding 100M system calls. This is attributed to paging overheads. Pre-faulting (Populate) improves performance for

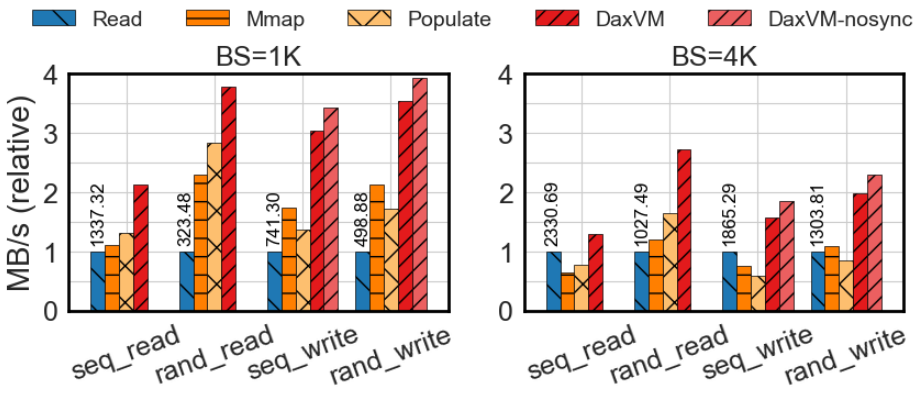



Figure 5.5: Repetitive file access.

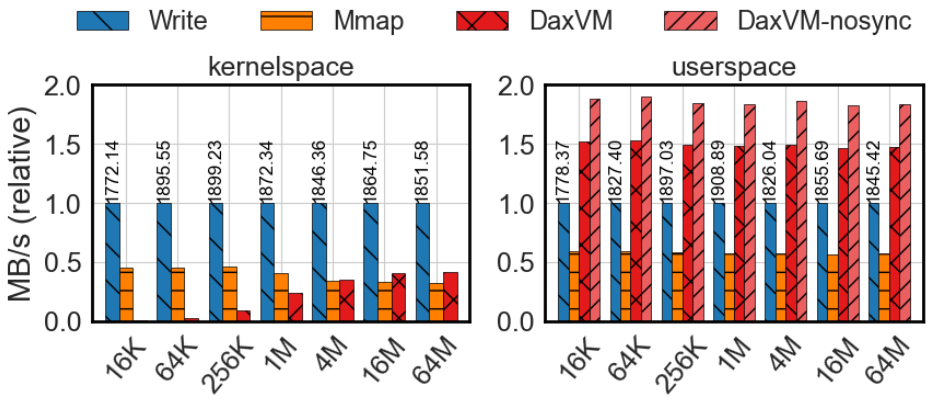



Figure 5.6: Kernel-space and user-space syncing operations.

read access, but penalizes it for write. For the latter, it ends up paying the fault overhead twice for each page of the mapping: (i) pre-population and (ii) dirty page tracking faults (Section 5.3). DaxVM eliminates all costs via O(1) file tables attachment during mmap and managing durability either (i) at 2MB granularity irrespective to fs fragmentation (faults) or (ii) entirely in user-space (nosync). It performs up to 3.9× better than system-call access and 1.9× than default mmap.

For 4KB access, default mmap, even with pre-faults, performs worse than read/write sequential access. The avoided cost of the fewer system calls is not enough to amortize paging overheads. DaxVM outperforms read/write calls from 1.3× up to 2.72×, and mmap from 1.8× to 2.2×.

For the irregular access workloads, DaxVM's performance monitor detects the high TLB miss overheads (Section 5.4) and migrates file tables from PMem to DRAM. We measure that migrating the tables provides a 10% performance improvement, avoiding the costly page walks when table fragments are located in slow PMem.

**Sync.** With PMem, sync operations are needed to ensure modified file data is flushed from processor caches. We consider the same experiment as in Figure 5.5, but with a 10GB file and perform 1000 sync operations after a varying number of sequential write operations. For kernel-space syncing and MM, we use memcpy and perform periodically fsync. For user-space syncing we use non-temporal stores and omit the fsync calls. We turn huge pages off, to stress the comparison with DaxVM, that always performs flushes at 2MB granularities. Figure 5.6 summarizes our results for the variable syncing sizes. We omit pre-faulting results, since as discussed do not benefit write access.

*Kernel-space syncing.* Kernel syncing of a mapped file performs worse than DAX write syscalls (up to 68% slowdown). Writes use non-temporal stores and synchronously persist data, while fsync on a mapped file flushes CPU caches. A prior study [231] shows that non-temporal stores almost double the bandwidth of cacheline flushes. For smaller syncing (<2MB), DaxVM performs up to an order of magnitude worse than default MM because it always handles durability at

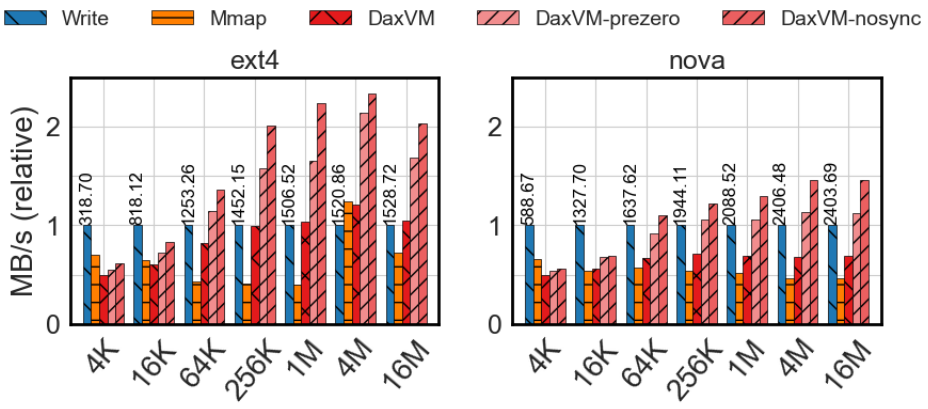



Figure 5.7: Append operations.

2MB granularity. However, in a non-fragmented FS image that uses 2MB pages, the default MM suffers from the same overheads due to huge pages (we measured this). Hence, DaxVM provides the same sync overhead performance trade-off with huge pages irrespective to FS fragmentation.

*User-space syncing.* Despite the kernel bypass, default MM performs worse than writes + kernel syncing (40%). DaxVM performs better and combined with the nosync optimization provides speedup up to 80%.

**Appends.** We now examine append performance via the different interfaces. As discussed in Section 5.3, MM append operations require an fallocate() to allocate new blocks and then map them for user-space write access. For security reasons, the OS must zero all blocks before allowing user-space access. Figure 5.7 shows the relative throughput achieved appending variable sizes as a single operation (one system call) on an empty file from a single thread. We compare against DaxVM (i) without pre-zeroing and with kernel-level page tracking to support sync operations, (ii) with pre-zeroing, and (iii) with both pre-zeroing and nosync, where the application is responsible for data durability. Because the results for ext4-DAX and NOVA differ substantially, we present them separately.

Regarding ext4-DAX, the results show that pre-zeroing can improve MM performance up to 2× for larger file sizes (DaxVM). For ext4-DAX this reflects also as a benefit compared to system call appends, as this FS conservatively zeroes-out blocks also on the system call path unnecessarily. *Nosync* mode boosts further performance up to ∼50%, eliminating entirely durability management faults and their page cache metadata update operations. For 4MB, default MM performance improves significantly due to huge page coverage. For very small files (e.g., 4KB) DaxVM performs worse due to the overheads of page table construction.

On the other hand, NOVA is a PMem-aware file system that does not zero out blocks during write system calls, but it zeroes them out only during fallocate calls for secure user-space DAX access. Figure 5.7 shows how this inherent differentiation in DAX interfaces requirements leads

to more than 2× faster write call performance (compared to MM) even for large append sizes (>1MB). DaxVM's pre-zeroing narrows this gap, and combined with O(1) mmap (file tables) and nosync optimizations, DaxVM outperforms write syscalls by up to 45%. It eliminates paging costs and exposes the user-space benefit of AVX instructions that are unavailable to the kernel. These results underline the necessity of handling block zeroing asynchronously with PMem storage.

**DaxVM storage overheads.** DaxVM occupies at least 4KB for files >32KB and adds an overhead of 4KB per 2MB of data (0.2%). For file tables smaller than 32KB, DaxVM builds volatile files tables. For the 891MB Linux git tree consisting of 68K small files, DaxVM occupies 25MB of PMem, and ephemerally uses up to 216MB of DRAM if all inodes are cached in memory.

**DaxVM latency overheads.** DaxVM benefits come at the cost of (de)constructing page tables during FS operations that involve storage block (de)allocations (e.g., fallocate/append/unlink). We measure the latency of appends with and without DaxVM's file tables. We find that volatile table construction adds almost zero overheads. But, persistent table construction penalizes operations at worst by ∼ 10% for 32KB appends on an empty file, and thereafter the overhead declines and is entirely amortized for 256KB and beyond. Persistent tables are more expensive to (de)construct as cache lines are flushed for durability.

All DaxVM benefits discussed so far are attributed to O(1) mmap, durability management and asynchronous pre-zeroing. We study DaxVM 's scalability optimizations (*ephemeral allocator* and *async unmappings*) on real-world applications in the next section.

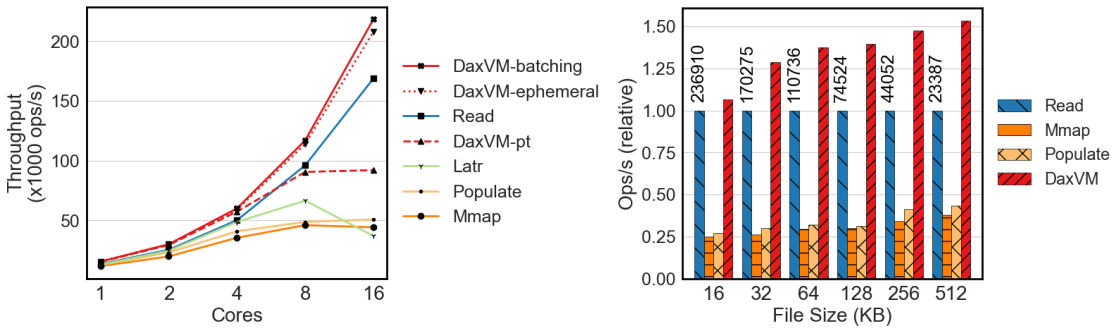
### 5.5.3 Real-world Applications

In this section we measure DaxVM performance with real-world applications operating over small and larger files. We change their source code to use `daxvm_m(un)map`.

#### 5.5.3.1 Small files and ephemeral access

**Apache** [2] webserver uses the `mpm_event` module where threads serve requests via memory-mapped access. They map web pages, copy data into sockets, and unmap them. The scheme stresses virtual memory due to frequent m(un)map requests. We measure Apache's throughput (requests/second) while hosting static 32KB webpages stored on PMem. We use Wrk [12] to generate HTTP requests, configuring it to run with 16 threads and 16 open connections. We run Wrk and Apache on the same machine but on different sockets and scale Apache from 1 to 16 cores (socket limit). We run wrk/Apache with multiple webpages of the same size [3] to avoid the unrealistic scenario of always hitting on the CPU cache when serve a webpage.

Figure 5.8a plots scalability results for Apache for 32KB webpages, and corroborate that its scaling is limited by virtual memory performance [35, 141]. Baseline MM access cannot scale




(a) Apache – scalability

(b) Apache – webpage size

Figure 5.8: DaxVM allows applications that issue many (un)map requests (e.g., web-servers) (b)) to scale to many cores and exposes the zero-copy advantage of MM over system call access on a setup that was previously considered prohibitive.

beyond 4 cores, while read scales almost linearly up to 16. To study DaxVM performance we incrementally add each optimization, starting with pre-populated file tables.

We verify that paging significantly limits MM scalability; DaxVM's O(1) mmap via file tables enables scaling up to 8 cores and improves performance by 80% compared to pre-faulting (Populate). Address space management is the other severe bottleneck. DaxVM's ephemeral address space (de)allocation enables scaling to 16 cores and improves throughput by 100% over file tables alone. The ephemeral heap operations acquire the mmap semaphore only as readers and use independent spinlocks for ephemeral address space management (Section 5.4.2). This enables concurrent m(un)map requests, significantly improving scalability. Finally, for this workload batching unmap requests does not improve substantially performance over ephemeral mappings (5%). The latter is sufficient to release the stress from the mmap semaphore. Overall, DaxVM minimizes VM overheads and outperforms baseline MM by 4× and read by 30%.

Finally, we run experiments with a kernel supporting LATR [141], a mechanism that uses message passing to replace TLB shootdowns with lazy local TLB invalidations on context switches. We run with MAP_POPULATE to control paging costs and find that LATR improves baseline MM performance by 10% at 8 cores and fails to scale beyond that because shootdowns are not the main problem. We find that DaxVM with only asynchronous unmapping (without O(1) mmap) outperforms LATR by 12% because: (i) DaxVM's batching can be more aggressive as it targets only PMem - it flushes the TLBs every 33 batched pages, (ii) DaxVM's batching is very simple, using existing IPIs, while LATR's status tracking mechanisms induce contention on its own locks. Note that DaxVM's asynchronous unmapping efficiency depends on the level of batching (number of pages allowed to be batched). When we increase batching level from 33 to 512, the performance increases by 20%. However, increasing the batching level increases also the DaxVM vulnerability window; the extra time that data remain mapped beyond what the user expected (Section 5.4.7).

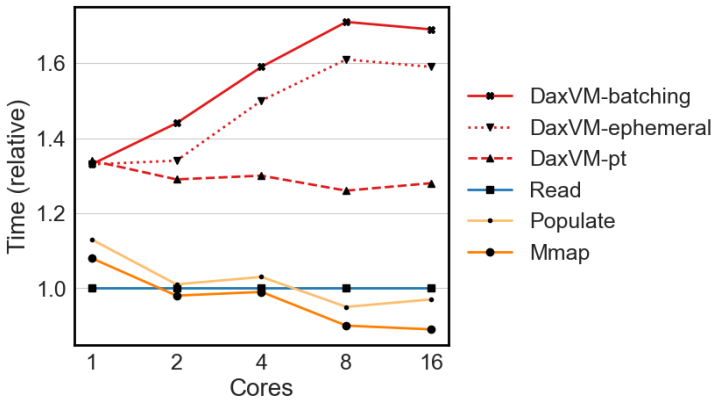



Figure 5.9: Text search performance. DaxVM improves scalability of applications that never move data out of PMem (like text search).

Figure 5.8b shows how webpage size affects performance. It summarizes the relative throughput results (ops/sec with respect to read) when we run Apache at 16 cores and for increasing webpage sizes. With read system call access, Apache copies the webpage content from PMem to DRAM and then from DRAM to a socket, while with MM access it copies it directly from PMem to socket. As the webpage size increases, the added cost of read's extra memory copy becomes more significant. DaxVM eliminates paging overheads and minimizes VM scalability bottlenecks to expose the zero-copy advantage of MM; it provides up to 50% benefit for bigger webpages.

*Multi-threading vs multi-processing:* Using multiple processes to serve requests trades system resource utilization (heavy processes vs. lightweight threads) for scalability to many cores, as there is less contention on the VM locks. Apache can run with a hybrid scheme, spawning a small set of processes with multiple threads each. However we find that even in the extreme case of using single-thread processes, baseline MM performs at best similar to read and only if pre-faulting is applied (populate). DaxVM provides 50% benefit over read both with lightweight threads and on a hybrid configuration, eliminating paging and scalability bottlenecks.

Overall, we note that combining DaxVM's optimizations under a single PMem dedicated interface is essential as the techniques operate synergistically. For example, combining asynchronous unmapping with O(1) mmap (applicable only to PMem) boosts the effect of the former, as shootdowns emerge as a contention bottleneck.

**Text search.** We now examine an application that operates directly over small mapped files (via load/store instructions). We use the ag [18] search engine to search the Linux codebase for a string. The folder contains the source tree (68K files) and a few large files used for git versioning. Using MM access, the search engine maps a file, searches for the requested string and unmaps it, while with read it copies the file into a private buffer. Figure 5.9 shows that DaxVM outperforms baseline mmap interfaces and read by ∼70% at 16 cores. The application does not spend time

copying data and DaxVM eliminates contention on the data access interface. Unlike Apache, asynchronous unmapping further boosts performance by 10%.

#### 5.5.3.2 Large files and long-lived mappings

Our evaluation so far shows that DaxVM is beneficial for applications that issue frequent m(un)map operations to access data. We now examine how DaxVM affects applications already benefiting from MM access, operating over large files and for longer periods. For this set of workloads we only compare against baseline MM access.

**5.5.3.2.1 Increasing availability with fast startup times.** DaxVM's mmap can significantly increase the availability of applications that serve requests from memory-mapped files, as it enables O(1) access to the file data after reboot.

**Redis.** P-Redis [220] is a PMem-aware version of the Redis [16] in-memory key-value store from NVSL [13, 220]. It consists of a key-value cache and an index hash table, both in PMem. When the server is spawned, it maps both structures and uses loads/stores for access. Loading data for P-Redis involves populating the mappings' page tables. With baseline MM access this happens lazily during a warmup period when client requests trigger faults. Figure 5.10 shows throughput for the first 2M random get operations on a 60GB cache that stores 16KB values. Baseline mmap performance increases slowly (warm-up faults) while mmap-populate penalizes server start-up time by 10sec to pre-fault the cache pages and then provides high throughput. DaxVM gets the best of both worlds, achieving instant maximum throughput at no cost.

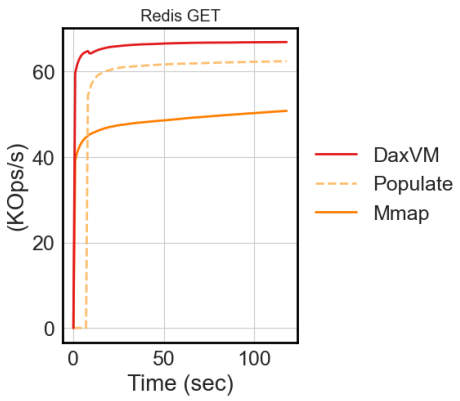



Figure 5.10: Redis boot

**Firecracker snapshots.** Serverless computing benefits from fast function start time to reduce request handling latency. Function instantiation overheads, particularly after long idle periods, are known as the *cold-start latency* problem. Cold starts commonly include boot costs as most providers use virtualization (fat or micro VMs ) to deploy functions for secure and isolated execution. To eliminate boot overheads, the state-of-the-art revisits an old idea: resuming VM execution from snapshots [7, 207]. After a function instance is fully booted, its complete state is captured and serialized in a binary snapshot file. When a new event arrives, the function instance is loaded from the snapshot and can immediately start processing the request, bypassing entirely the boot process. Loading a snapshot from storage implies that the Virtual Machine Monitor (e.g

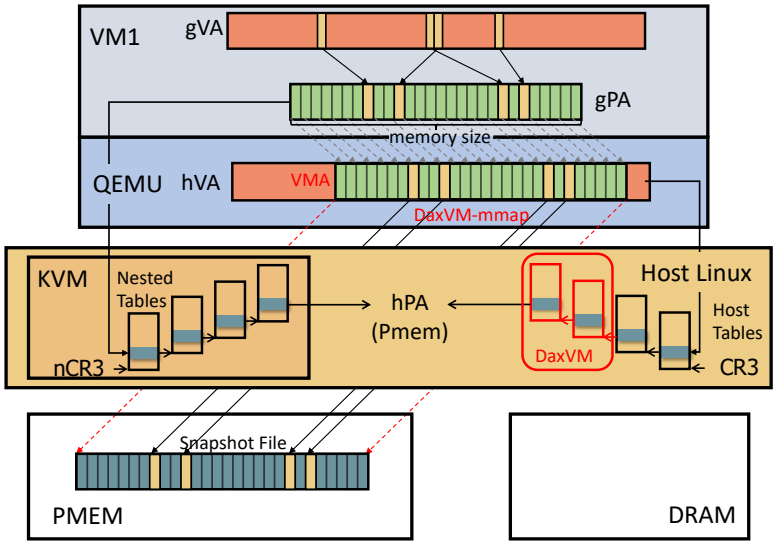



Figure 5.11: DaxVM O(1) mmap accelerates the restoration of a virtual machine state from a snapshot stored in PMem. In this unique set-up snapshot's read-only pages are never copied to DRAM, and thus the VM's physical memory gets essentially backed by PMem [132].

qemu or firecracker) memory maps the snapshot file and exposes it as the instantiated virtual machine's guest physical memory. The mapping is marked as private, and any write to the guest physical address space causes a copy-on-write fault on the host. As the function executes, VM exits are triggered to build both host and extended page tables that map the guest physical address space to snapshot pages. We examine cold start latencies of Firecracker microVMs [27] when the snapshots are stored in PMem and memory mapped using DAX. Figure 5.11 shows such a unique set-up, evaluated recently by Katsakioris et.al [132]. DaxVM O(1) mmap can be used during snapshot loading, to attach the snapshot's pre-populated file tables to the VMM host process private page table tree. This can accelerate the aforementioned VM exits, as only the extended page tables will be built.

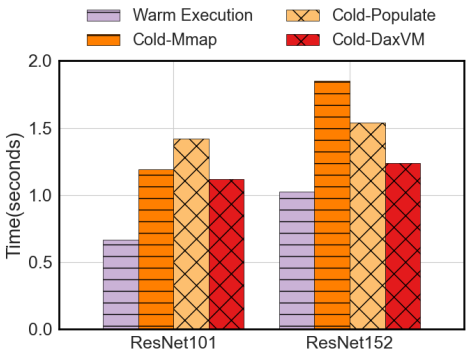



Figure 5.12: Firecracker Cold-starts

Figure 5.12 summarizes our initial results for a CNN function that serves image classification requests using a pre-trained ResNet model [111]. We use 1GB microVMs and we take a snapshot when the function has loaded the model, ready to serve inference requests. We use two different ResNet configurations with increasing depth of layers. We show the time it takes to serve a classification request when the function instance is in-memory (warm) and when it loads from the snapshot (cold). With default MM access, cold execution suffers from expensive VM exits

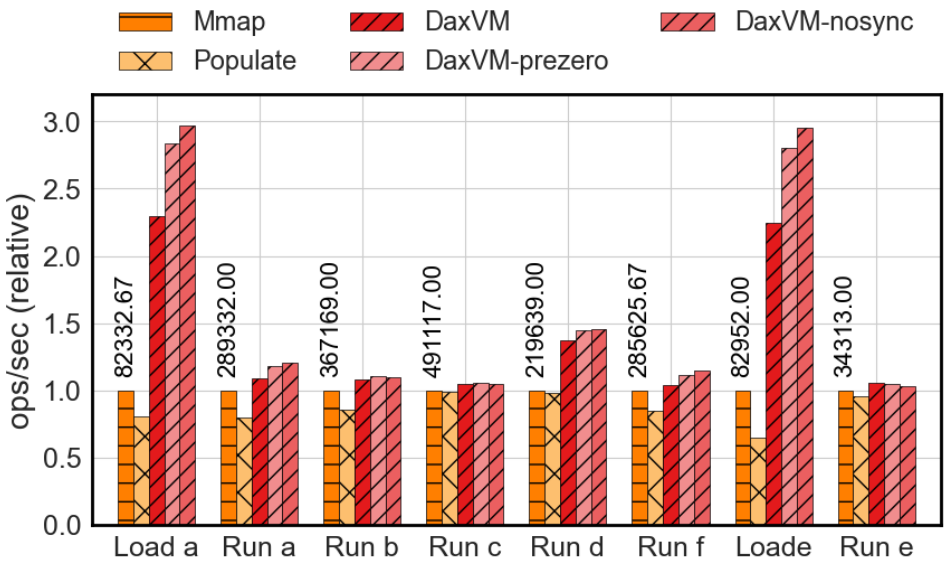



Figure 5.13: YCSB on RocksDB. DaxVM sustains high operational throughput for databases on a fragmented ext4 images.

that build both the host and the extended page tables (∼70% for ResNet152) for the snapshot mapping. Applying pre-faults is beneficial for the large CNN but harmful for the smaller one. Pre-populating the entire mapping of the VM memory sets up host page table entries for regions that are never accessed by the guest/function; paying the cost without any benefit. DaxVM attaches the pre-populated page tables of the snapshot file to the host mapping (host page tables) via O(1) mmap operations. This way i) it accelerates VM-exits as only extended page tables are built and ii) remains still beneficial even if parts of the snapshot are never accessed. Cold execution time with DaxVM drops to only 25% slower than warm. As discussed in Section 5.4.1.3, we consider DaxVM virtualization extensions as future work to eliminate also VM-exits.

**5.5.3.2.2 Databases.** Finally we examine how DaxVM affects the performance of a database optimized to use PMem programming.

**YCSB on Pmem-RocksDB.**. Pmem-RocksDB [118] is Intel's PMem-optimized version of RocksDB [93] that mmaps SSTables/write-ahead logs (WALs) (placed on PMem) and writes directly to PMem using non-temporal stores (e.g., nt-store), omitting kernel sync operations [116]. It also recycles SSTables and WAL files whenever possible to control paging and zeroing overheads. We run YCSB workloads on a 50G dataset [124] and perform ∼12M operations (4KB records).

*Ext4-dax results.* Figure 5.13 summarizes our results. As discussed earlier, pre-faulting (populate) hurts performance of write/append intensive workloads (such as Load a, Load e and Run a). For the rest it performs close to default mmap.

DaxVM significantly improves performance for applications that perform insert operations (e.g., Load a, e). DaxVM dirty page tracking faults happen always at 2MB granularities (Section 5.4), irrespective to the file system's fragmentation. This significantly decreases the number of page-faults (10x less) improving performance by ∼2.3×. When we pre-zero in advance of running the workload (shown), performance is further boosted to ∼2.8x. With concurrent pre-zeroing, a 64MB/sec throttle reduces this by 5-10%. Finally, this version of RocksDB enforces durability from user-space [116] so we apply the *nosync* mode that brings performance to ∼2.95× compared to default mmap, eliminating faults entirely.

The main reason why DaxVM is so effective is that default mmap on an aged ext4-dax suffers from synchronous faults imposed by the MAP_SYNC interface [78] necessary to safely handle durability from user-space. On the first write fault on each mapped page the dirty file metadata (if any) will be synchronously flushed to storage. This triggers journaling transaction commits on ext4 that severely penalize scalability. On an aged FS, 4KB pages are involved on the mapping of a file, and thus such faults are more frequent. With DaxVM this happens always at 2MB granularities (less frequently) irrespective of FS fragmentation, restoring scalability to many cores. Note that on a fresh file system (100% huge page coverage) default mmap performs similarly.

DaxVM improves also performance by 1.46× for workload d (that also performs insertions) and 1.05-1.21× for the rest. All benefits come from fault elimination, as DaxVM's ephemeral allocator and asynchronous unmapping do not affect the long-lived file mappings of the workload.

*NOVA results.* For PMem-aware file systems that update metadata synchronously and in-place (such as NOVA) the MAP_SYNC interface becomes a no-op with zero overheads. We run the same experiments on a NOVA FS image and DaxVM's benefits for Load a and e are ∼35% compared to default mmap. For the rest of the workloads are ∼10%.

*Comparison to default RocksDB [93].* This (Intel) optimized version of the key-value store provides ∼1.1×-2.1× benefit compared to the default version, when we run on a fresh FS image. When run on an aged ext4 image this benefit is penalized (as discussed before). DaxVM sustains up to 2× benefit even on the fragmented FS.

### 5.5.4 Summary

Table 5.5 summarizes our benchmarking observations. DaxVM minimizes paging overheads, for small and large file mappings, and enables virtual memory operation scalability to many cores. This boosts the performance of ephemeral MM file access over small files (e.g. webservers), where baseline MM performs poorly and read system calls are commonly favored. For larger file mappings with long lifetimes and on commonly fragmented FS images, DaxVM sustains high MM performance, reducing the paging overheads attributed to the small pages involved in the

| | Linux MMIO | | DaxVM MMIO | |
|---|---|---|---|---|
| | Fresh (huge pages) | Aged (small pages) | Fresh (huge pages) | Aged (small pages) |
| Low paging overheads for small files | ✗ | ✗ | ✓ | ✓ |
| Low paging overheads for large files | ✓ | ✗ | ✓ | ✓ |
| Operations scalability | ✗ | ✗ | ✓ | ✓ |
| TLB performance | ✓ | ✗ | ✓ | ✗ |

Table 5.5: Summarizing observations.

file's mapping due to FS fragmentation. On a fresh FS image, DaxVM performs equally to all MM interfaces by supporting huge pages in file tables.

## 5.6 Discussion: DaxVM beyond persistent memory

According to Intel's 2022 Q2 earning release [121], the company is winding down its Optane Memory business, which is a significant step back for persistent memory research. We do not consider this as the end of PMem storage design potential and discuss how DaxVM is relevant and beneficial for other fast storage technologies (despite being designed on Optane).

### 5.6.1 O(1) mmap and file tables

*Byte-addressable storage and CXL.* DaxVM is directly applicable to any byte-addressable storage technology; a design advocated by the emerging Compute Express Link (CXL [85]), e.g., Samsung has already announced a memory-semantic SSD that is CXL-compatible [186]. Any such storage solution, even PCIe and byte-addressable Flash NVMe combinations [24], is very close to PMem's philosophy and can benefit from DaxVM.

*Microsecond-scale PCIe SSDs and direct access.* State-of-the-art flash memory technologies have reduced storage-access latency to tens of microseconds [46, 175]. Such performance has exposed system storage stack as an important bottleneck and has questioned DRAM buffering as a necessary layer leading to various proposals for user-space direct access to storage [65, 129, 233]. Such solutions require rethinking and speeding up file system indexing [162] and even accelerating it in hardware [145] for performance and security reasons. DaxVM's FS file tables fall into this scope.

*Memory-mapped buffered access.* DaxVM's O(1) mmap and pre-populated file tables can be integrated as a page cache extension, to speedup traditional buffered storage access.

### 5.6.2 Address Space Scalability

DaxVM's ephemeral mappings and asynchronous unmappings are relevant to any memory access with ephemeral characteristics. This could apply both to direct or buffered memory-mapped storage access or even heap mappings. Memory tiering and fast storage rapidly change the usage of memory as a now common interface to multiple mediums with varying latencies. This imposes new challenges to address space management, questioning the state-of-practice.

## 5.7 Related Work

**User-space file systems.** Multiple works [66, 87, 125, 142, 149, 211, 235] exploit PMem direct access via new file system (FS) designs with user-space components. Performing (meta)data operations directly from user-space avoids syscall overheads, but comes with two inherent challenges: (i) (meta)data security and (ii) concurrent file sharing. Mapping parts [66, 125, 142] or the entire FS image [149, 211] to user-space for large time frames opens a window for intentional attacks or unintentional errors (stray writes) that can leak data or corrupt the FS image [87]. Such FS must employ a mechanism to control this that may lead to scalability issues [142, 211]. In addition, many user-level FS do not support memory mapping [142, 211] at all. Kernel-space FS can be less performant but support seamlessly sharing and secure (meta)data operations. In this paper we focus on such well-tested mature FS targeting to improve the kernel's MM interface performance rather than bypass it.

**File system indexing.** HashFS [162] uses hashing instead of the commonly employed extent trees to accelerate software overheads of file indexing on the read/write system call path. ctFS [149] is a user-space file system that maps the entire DAX device in user-space to (de)allocate files contiguously in the virtual address layers, similar to SCMFS [219]. It then uses page tables to index files quickly. Exposing the entire device to user-space raises significant security concerns acknowledged by the authors. DaxVM (de)attaches file tables directly to address spaces, primarily to eliminate the paging costs of MM access. By doing so, it entirely removes software file indexing from the MM path. We discuss other works that employ file system maintained page tables on Section 5.3.

**Address space scalability.** Past studies [70, 71] of address space scalability target generic solutions (e.g., range locking or concurrent lock-free data structures in VM) that apply to all memory regions. However, the Linux community has been discussing such radical changes for many years [76] and relevant implementations [80, 183] show that the transition is not that easy in

terms of performance [140] or complexity. A key insight of DaxVM is that one can exploit the special lifetime and access characteristics of PMem mappings to provide a much simpler dedicated address space (de)allocator that can scale to many cores (ephemeral mappings).

**Fast unmap.** LATR [141] proposes message passing – a generic radical re-design of the TLB invalidation mechanism to enable lazy invalidations. DaxVM exploits batched un-mapping requests, adopting a dedicated design already present for the IOMMU and traditional storage DMA mappings [174]. The key insight is that opting for a dedicated design for targeted uses can enable higher performance at a much lower complexity. Numerous proposals for faster/simpler delivery of shootdowns in hardware [210,229] and software [48], or for more accurate shootdowns [34,35] would reduce the need for DaxVM's asynchronous unmapping. Boyd-Wickizer et al. [59] examine per-thread address private ranges to avoid synchronization and TLB shootdowns.

**Pre-zeroing.** Hawkey [165] and Trident [179] examine asynchronous pre-zeroing for huge volatile page allocation latency. DaxVM exposes its necessity for PMem file mappings and integrates it in a file system. Our key insight is that block zeroing is an inherently different requirement among DAX interfaces (MM access vs system calls) that if not managed can flip performance trends.

**Faster paging:** Previous works underline the cost of paging and particularly of faults for PMem direct access [125, 220, 223]. They propose huge page usage [125, 220], caching per-process file mappings [69], and O(1) memory [201] on a conceptual or emulated level (more in Section 5.3) DaxVM expands on this work with a real implementation of O(1) mmap in Linux and on unmodified hardware, reduces DRAM consumption by placing file tables in PMem, and avoids dependence on huge pages. WineFS [124] is a new huge-page aware FS for high huge page coverage. DaxVM is complementary to huge pages, supporting them when available, but resilient to fragmentation in terms of paging. Prior work on sharing page tables focused on speeding fork [88] and removing duplicate TLB entries [194].

Song et al. [195, 196] focus on faster page reclamation and batching shootdowns under memory pressure. Papagiannis et al. [170] propose an mmap design that ignores DAX, targeting page cache optimizations. DaxVM focuses on DAX mappings that are neither subject to memory pressure nor use a cache.

**PMem file systems:** Many research projects focus on faster file systems for PMem, and mostly look at (i) avoiding the page cache like DAX [74, 90], (ii) providing faster metadata operations with fine-grained persistence [74, 137, 219, 221, 222], and (iii) moving kernel operations to user-space (discussed before).

CHAPTER 6

# Conclusions

## 6.1 Summary

This thesis analyzes the performance overheads of today's virtual memory design with respect to (i) address translation, focusing on virtualized execution, and (ii) the direct access interface to persistent data. It then proposes hardware/software co-designed techniques to address them.

We introduce complementary software and hardware methods to mitigate the address translation overheads, focusing on the challenging setup of nested paging. On the OS level, we propose CA paging to generate vast contiguous mappings by allocating target pages across page fault traps. CA paging maintains all lightweight memory-management techniques of a modern OS (e.g. on demand allocation, Copy-on-Write etc) and avoids any reservation/pre-allocation of memory, working on a best-effort basis. It can be used to support any hardware scheme that harvests contiguity in mappings, e.g. RMM [130] or AnchorTLB [172]. We have implemented it in stock Linux and made it publicly available. On the hardware side, we propose SpOT, a micro-architectural engine that predicts translations on the TLB miss path. SpOT exploits mappings linearity and hides TLB miss penalties under speculative execution, trading a very simple micro-architectural design –that can easily and transparently support virtualization– with strong security guarantees. Combined with CA paging, SpOT significantly reduces the translation overheads of nested paging from ∼16.5% to ∼0.9%.

We also study all sources of overhead that emerge when we use memory as the interface towards files. We look at how each of the expensive mechanisms of this legacy interface are

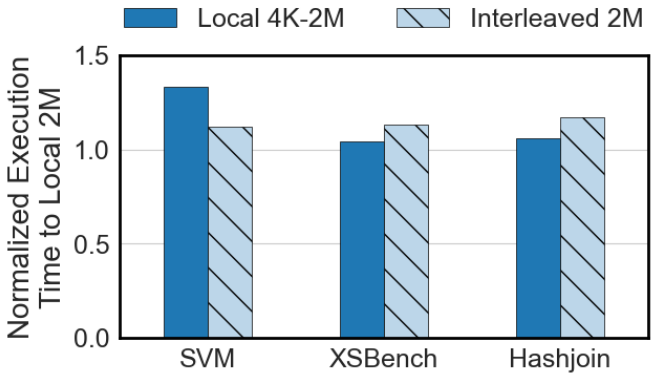



Figure 6.1: Huge Pages and NUMA. Local NUMA placement may not always be the optimal choice in the presence of external fragmentation. We compare the total execution time of a workload when its memory is i) 100% local but 50% covered by huge pages and ii) 100% covered by huge pages but interleaved. We observe that the latter is better for SVM.

affected by new fast storage technologies or become obsolete. Based on our analysis we propose a new interface for fast and scalable direct access to persistent data that aims to come close to what the underlying storage can provide. DaxVM is a POSIX-relaxed file mapping interface for persistent memory, implemented as a re-design of virtual memory operations and a co-designed support in PMem-aware file systems – all driven by direct access unique characteristics. DaxVM offers (i) O(1) memory mapping operations via persistent page tables integrated in file system's inode metadata, (ii) lazy invalidation of TLBs via batching and asynchronous issuing of unmapping requests, (iii) scalable address space management for ephemeral mappings, (iv) elimination of kernel-space durability management support when user-space is in charge and (v) asynchronous storage block pre-zeroing by the file system to accelerate append operations, dealing with added overheads introduced to cover the security implications of direct access. We implement DaxVM in stock Linux and the ext4-DAX and NOVA file systems and make it publicly available. For multi-threaded workloads that process multiple small files for short intervals, e.g., Apache, DaxVM improves standard mmap performance up to 4.9x. It also reverses the trend that favors read for such setups, outperforming it by up to 1.5x. DaxVM also increases system availability, providing fast boot times for PMem databases, and sustains their high throughput even when running on fragmented file system images.

## 6.2 Future Research Directions

In this section we discuss some new research ideas that we shaped during this study.

### 6.2.1 Address translation and Non-Uniform Memory Access performance

The current thesis focused mainly on single-node address translation performance; we did not study the relationship between address translation and Non-Uniform Memory Access (NUMA). In the next paragraphs we discuss briefly how the two interfere, based on literature findings and some initial findings of our own.

NUMA performance may dictate page allocations to span over multiple nodes for better locality and/or less link contention. Address Translation performance may antagonistically require large memory blocks to be contiguously allocated in a single node (e.g. large pages) to minimize TLB miss overheads. In this direction, prior work [103] finds that huge pages can indeed harm NUMA systems performance, especially for heavily threaded workloads that span over multiple sockets. We question if the reverse trend is also valid; if there are scenarios that address translation should be prioritized over NUMA locality for overall better system performance. For initial insight, we fragment a memory node in our server and measure the execution time of workloads running inside a VM with a) 100% of their memory local but 50% covered by 2MB pages (due to external fragmentation) and b) 100% covered by 2MB but interleaved between the local and a remote node. Workloads' threads are always located in a single node. We compare with the state-of-practice optimal setup: 100% locality and 100% huge page coverage. Figure 6.1 summarizes our results. We interestingly observe that SVM performs better when we trade locality for higher huge page coverage (exploiting the available huge pages on a remote node). This is an initial verification that there is research potential in data placement policies that take into account and co-ordinate (i) NUMA locality, (ii) nodes bandwidth contention and (iii) address translation performance. As dis-aggregated memory pools are expected to dominate future data-center system architectures (via CXL [85]), we believe that NUMA placement will play a significant role in future systems performance.

### 6.2.2 Efficient multiple page size support

While the x86 architecture supports only a limited number of page sizes, namely 4K, 2MB and 1GB, other (micro-)architectures and ISAs support multiple page sizes [37,180] or TLB coalescing techniques [1]. Despite the potential in such more flexible and higher reach translation schemes, discussed extensively in this thesis, we find that none of the above is transparently supported by stock Linux. Other page sizes –beyond THP– are supported only by the rigid libhugetlbfs [9] that reserves/pre-allocates memory. Moreover, TLB coalescing exploits only the limited beyond-page-size-limit contiguity that the default Linux buddy allocator randomly generates.

Table 6.1: ARMv8 supported page sizes

| Processor | Base | Large | Intermediate |
|---|---|---|---|
| ARMv8 | 4KB | 2MB, 1GB | 64KB, 32MB |
| | 16KB | 32MB | 2MB, 1GB |
| | 64KB | 512MB | 2MB, 16GB |

#### 6.2.2.1 The case of ARMv8

ARM architecture supports multiple base page sizes (called granules), 4KB, 16KB, 64KB, set at boot time. More interestingly, its paging structures support a 'contig' bit, that promotes contiguously allocated and properly aligned mapped base pages into a single translation entry, cach-able on the TLB, that resembles a restricted version of range translations [130]. We name these new translation granularities, intermediate. Table 6.1 summarizes ARMv8 arch translation support.

We consider an impactful research topic to extend CA paging to transparently support the intermediate page sizes of ARMv8 for the 4KB base page (64KB and 32MB). This would require support to transparently manage the 'contig bit' by the OS and also adjust CA paging's VMA placement decisions to comply with the alignment restrictions of the hardware. Apart from our high interest in extending virtual memory to efficiently support commodity hardware and enable its usage, we also believe that such a study would deal with research questions of broader interest: (i) Are always 2MB pages necessary or 64KB pages are sometimes enough? (ii) Since 64KB/32MB translation entries are created by contiguous allocations of 4KB/2MB base pages, what is the trade-off between address translation performance and internal fragmentation when you compare intermidiate with fixed page sizes? (iii) How do we decide the target page size at page fault time? We generally believe that multiple page sizes –beyond x86 architecture support– can be of great benefit and that CA paging can play a significant role in designing an efficient, transparent OS support for them.

### 6.2.3 Virtual machine snapshotting

As discussed throughout this thesis, cloud infrastructure has dominated the computing deployment landscape and thus virtualization performance has become a leading cost-effectiveness factor. In Chapter 4 we discussed how efficient memory virtualization plays a significant role in this and in Chapter 5 we briefly went through the impact of paging on the performance of a low-latency boot time technique, virtual machine snapshotting. In the realm of serverless computing [189], the latter has concentrated a lot of attention [36, 132, 207]. Efficient execution restoration from virtual machine snapshot files is a set-up that uniquely combines memory management and file access performance, i.e. the state of the guest OS memory is serialized in a file that must be efficiently mapped by the host OS and the hypervisor on a future invocation of

the target application. We consider this area an elaborate intersection of this thesis study and combining Contiguity-Aware paging and the DaxVM file mapping interface (our proposals) an interesting research direction. In example, contiguous guest OS memory allocations, coalesced based on access rights (read-only libraries, read-write heap etc), could increase the linearity in guest memory accesses. In turn, this could increase the effectiveness of host page cache read-ahead linear prefetching over the snapshot file, limiting the need for offline/online working set tracking and specialized prefetching proposed in literature [36, 207]. Also such linearity could boost the usage of larger page sizes in the host, a direction already initially explored for fast snapshot restoration [152]. Moreover, DaxVM O(1) mmap via pre-populated page tables could be leveraged to accelerate further snapshot file mapping. This could involve extending DaxVM O(1) mmap to support block devices and page cache buffered access –pre-populated page tables for snapshot file pages residing on DRAM– and/or virtualize DaxVM by enhancing the hypervisor with extended/nested pre-populated page table caching per VM to eliminate entirely nested paging overheads.

### 6.2.4 Fast user-space access to low-latency SSDs

As already discussed in previous chapters, high performance IO devices evolve rapidly and get widely adopted in the data-center world. In this thesis we have focused on persistent memory technologies, but there is also an increasing number of low-latency PCIe SSDs (e.g. [8]) that operate at single digit microsecond scale and offer bandwidth of multiple GBs/sec. This outstanding device performance has exposed the kernel software IO stack as a very expensive data path, e.g. context switching overheads [113], interrupt processing [46] or storage queue management [114] emerge as bottlenecks. *Kernel-bypass* and user-space access are a way out widely explored both by the state-of-the-art (e.g. [151, 175, 236]) and the state-of-practice (e.g. [233]). To this direction Intel's new ENQCMD instruction and shared virtual addressing [120, 134], initially proposed for accelerator device access, open new opportunities for submitting work to IO devices from user-space using the IOMMU for translation and security reasons. We believe that DaxVM's optimizations, e.g. pre-populate file tables or TLB management, could be adoptedto support the IOMMU and accelerate user-space access for low-latency PCIe devices.

# Bibliography


[1] Amd zen2. https://www.7-cpu.com/cpu/Zen2.html.

[2] Apache HTTP Server project. https://httpd.apache.org/.

[3] Benchmark multiple url paths with wrk. https://gist.github.com/xydinesh/28bd6ac7de1d45b61a9d896e3442248d.

[4] Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[5] Exadata with Persistent Memory:An Epic Journey. https://www.snia.org/educational-library/exadata-persistent-memoryan-epic\-journey-2020.

[6] ext4: Use page_mkwrite vma_operations to get mmap write notification. https://linux-ext4.vger.kernel.narkive.com/kplEwAhG/patch-ext4-use-page-mkwrite-vma-operations-to-get-mmap\-write-notification.

[7] Firecracker Snapshotting. https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md.

[8] Intel® Optane™ DC SSD Series.

[9] libhugetlbfs(7) - Linux man page. https://linux.die.net/man/7/libhugetlbfs.txt.

[10] Linux ftrace utility. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[11] Memory Management. https://github.com/torvalds/linux/blob/master/Documentation/x86/x86_64/mm.rst.

[12] Modern HTTP benchmarking tool. https://github.com/wg/wrk.

[13] Non-volatile systems laboratory. https://nvsl.io/.

[14] Pagemap utility. https://www.kernel.org/doc/Documentation/vm/pagemap.txt.

[15] Persistent Memory File System. https://github.com/linux-pmfs/pmfs.

[16] Redis: an in-memory data structure store. https://redis.io/.

[17] TCMalloc. http://goog-perftools.sourceforge.net/doc/tcma\lloc.html,.

[18] The Silver Searcher: A Code Searching Tool for Programmers. https://www.tecmint.com/the-silver-searcher-\a-code-searching-tool-for-linux/.

[19] Transparent Huge Pages in 2.6.38. http://lwn.net/Articles/423584/.

[20] Persistent Memory Programming. https://lwn.net/Articles/613384/, 2014.

[21] Intel® Xeon® Processor E5-2600 V4 Product Family Technical Overview, 2016.

[22] 5-level paging and 5-level ept white paper. Tech. rep., Intel, 2017.

[23] Persistent memory documentation: Using qemu virtualization. https://docs.pmem.io/persistent-memory/getting-started-guide/creating-development-environments/virtualization/qemu, 2020.

[24] Abulila, A., Mailthody, V. S., Qureshi, Z., Huang, J., Kim, N. S., Xiong, J., and Hwu, W.-m. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), ASPLOS '19.

[25] Achermann, R., Panwar, A., Bhattacharjee, A., Roscoe, T., and Gandhi, J. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems* (2020).

[26] Achermann, R., Panwar, A., Bhattacharjee, A., Roscoe, T., and Gandhi, J. *Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines.* Association for Computing Machinery, New York, NY, USA, 2020, p. 283–300.

[27] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 419–434.

[28] Agrawal, N., Bolosky, W. J., Douceur, J. R., and Lorch, J. R. A Five-Year study of File-System metadata. In *5th USENIX Conference on File and Storage Technologies (FAST 07)* (San Jose, CA, Feb. 2007), USENIX Association.

[29] Ahn, J., Jin, S., and Huh, J. Revisiting Hardware-assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (2012).

[30] Alam, H., Zhang, T., Erez, M., and Etsion, Y. Do-it-yourself virtual memory translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), ISCA '17, p. 457–468.

[31] Alam, H., Zhang, T., Erez, M., and Etsion, Y. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017).

[32] Alverti, C., Karakostas, V., Kunati, N., Goumas, G., and Swift, M. Daxvm: Stressing the limits of memory as a file interface. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2022), pp. 369–387.

[33] Alverti, C., Psomadakis, S., Karakostas, V., Gandhi, J., Nikas, K., Goumas, G., and Koziris, N. Enhancing and exploiting contiguity for fast memory virtualization. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020).

[34] Amit, N. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the USENIX Annual Technical Conference* (2017).

[35] Amit, N., Tai, A., and Wei, M. Don't Shoot down TLB Shootdowns! In *Proceedings of the 15th European Conference on Computer Systems* (2020).

[36] Ao, L., Porter, G., and Voelker, G. M. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), EuroSys '22.

[37] ARM LTD. Arm architecture reference manual for a-profile architecture (d.8.6.1 the contiguous bit), August 2022. https://developer.arm.com/documentation/ddi0487/ia/?lang=en.

[38] Arulraj, J., Levandoski, J., Minhas, U. F., and Larson, P.-A. Bztree: A high-performance latch-free range index for non-volatile memory.

[39] Awad, A., Basu, A., Blagodurov, S., Solihin, Y., and Loh, G. H. Avoiding tlb shootdowns through self-invalidating tlb entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2017), pp. 273–287.

[40] Awad, A., Basu, A., Blagodurov, S., Solihin, Y., and Loh, G. H. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques* (2017).

[41] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. The NAS parallel benchmarks summary and preliminary results. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (1991).

[42] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., and Ousterhout, J. Measurements of a distributed file system. *ACM SIGOPS Operating Systems Review 25* (apr 2000).

[43] Barr, J. Ec2 in-memory processing update: Instances with 4 to 16 tb of memory and scale-out sap hana to 34 tb, 2017.

[44] Barr, T. W., Cox, A. L., and Rixner, S. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010).

[45] Barr, T. W., Cox, A. L., and Rixner, S. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (2011).

[46] Barroso, L., Marty, M., Patterson, D., and Ranganathan, P. Attack of the killer microseconds. *Commun. ACM 60*, 4 (mar 2017), 48–54.

[47] Basu, A., Gandhi, J., Chang, J., Hill, M. D., and Swift, M. M. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013).

[48] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP '09, Association for Computing Machinery, p. 29–44.

[49] Baxter, R. Cxl: Enabling new pliability in the modern data center. `https://memverge.com/cxl-forum-at-fms-2022/`, 2022.

[50] Beeler, B. Intel optane dc persistent memory module (pmm). `https://www.storagereview.com/news/intel-optane-dc-persistent-memory-module-pmm`, 2019.

[51] Bhargava, R., Serebrin, B., Spadini, F., and Manne, S. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008).

[52] Bhattacharjee, A. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013).

[53] Bhattacharjee, A. Preserving virtual memory by mitigating the address translation wall. *IEEE Micro 37*, 5 (2017), 6–10.

[54] Bhattacharjee, A. Preserving Virtual Memory by Mitigating the Address Translation Wall. *IEEE Micro 37*, 5 (Sep. 2017).

[55] Bhattacharjee, A. Translation-Triggered Prefetching. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems* (2017).

[56] Bhattacharjee, A., Lustig, D., and Martonosi, M. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture* (2011).

[57] Bhattacharjee, A., and Martonosi, M. Inter-core Cooperative TLB for Chip Multiprocessors. In *Proceedings of the 15th Annual Conference on Architectural Support for Programming Languages and Operating Systems* (2010).

[58] Bovet, D., and Cesati, M. *Understanding The Linux Kernel.* Oreilly Associates Inc, 2005.

[59] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (USA, 2008), OSDI'08, USENIX Association, p. 43–57.

[60] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (USA, 2010), OSDI'10, USENIX Association, p. 1–16.

[61] Bugnion, E., Devine, S., Govil, K., and Rosenblum, M. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.* (1997), 412–447.

[62] Canella, C., Bulck, J. V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *Proceedings of the 28th USENIX Security Symposium* (2019).

[63] Carellan, E. B. G. D. Intel docs: Speeding up i/o workloads with intel® optane™ persistent memory modules: Block access versus byte access. `https://www.intel.com/content/www/us/en/developer/articles/technical/speeding-up-io-workloads-with-intel-optane-dc-persistent\-memory-modules.html`, 2019.

[64] Caulfield, A. M., De, A., Coburn, J., Mollow, T. I., Gupta, R. K., and Swanson, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (2010), pp. 385–395.

[65] Caulfield, A. M., Mollov, T. I., Eisner, L. A., De, A., Coburn, J., and Swanson, S. Providing safe, user space access to fast, solid state disks. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012* (2012), T. Harris and M. L. Scott, Eds., ACM, pp. 387–400.

[66] Chen, Y., Lu, Y., Zhu, B., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Shu, J. Scalable persistent memory file system with kernel-userspace collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (Feb. 2021), USENIX Association, pp. 81–95.

[67] Chinner, D. xfs: DAX support. `https://lwn.net/Articles/635514/`, 2015.

[68] Choi, J., Hong, J., Kwon, Y., and Han, H. Libnvmmio: Reconstructing software IO path with Failure-Atomic Memory-Mapped interface. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 1–16.

[69] Choi, J., Kim, J., and Han, H. Efficient memory mapped file i/o for in-memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (Santa Clara, CA, July 2017), USENIX Association.

[70] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII, Association for Computing Machinery, p. 199–210.

[71] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, Association for Computing Machinery, p. 211–224.

[72] Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., and Kohler, E. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst. 32*, 4 (Jan. 2015).

[73] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI, Association for Computing Machinery, p. 105–118.

[74] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP '09, Association for Computing Machinery, p. 133–146.

[75] Corbet, J. Memory management locking. `https://lwn.net/Articles/591978/`, 2014.

[76] Corbet, J. Memory-management scalability. `https://lwn.net/Articles/636334/`, 2015.

[77] Corbet, J. Five-level page tables. `https://lwn.net/Articles/717293/`, 2017.

[78] Corbet, J. Two more approaches to persistent-memory writes. `https://lwn.net/Articles/731706/`, 2017.

[79] Corbet, J. Memory management notifiers. https://lwn.net/Articles/266320/, 2018.

[80] Corbet, J. How to get rid of mmap_sem. https://lwn.net/Articles/787629/, 2019.

[81] Corp., R. Data Center Evolution: DDR5 DIMMs Advance Server Performance. Part2: Memory trends in 2021 and beyond. Tech. rep., 2021.

[82] Corp., S. Sap hana: an in-memory, column-oriented, relational database management system. https://www.sap.com/products/technology-platform/hana.html, 2014.

[83] Cox, G., and Bhattacharjee, A. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems* (2017).

[84] Crotty, A., Leis, V., and Pavlo, A. Are you sure you want to use mmap in your database management system? In *CIDR 2022, Conference on Innovative Data Systems Research* (2022).

[85] CXL_Consortium. Compute express link specification revision 2.0. https://www.computeexpresslink.org/download-the-specification.

[86] Daley, R. C., and Dennis, J. B. Virtual memory, processes, and sharing in multics. *Commun. ACM* (1968).

[87] Dong, M., Bu, H., Yi, J., Dong, B., and Chen, H. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 478–493.

[88] Dong, X., Dwarkadas, S., and Cox, A. L. Shared address translation revisited. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys '16, Association for Computing Machinery.

[89] Du, Y., Zhou, M., Childers, B. R., Mossé, D., and Melhem, R. Supporting superpages in non-contiguous physical memory. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture* (2015).

[90] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14, Association for Computing Machinery.

[91] El Hajj, I., Merritt, A., Zellweger, G., Milojicic, D., Achermann, R., Faraboschi, P., Hwu, W.-m., Roscoe, T., and Schwan, K. Spacejmp: Programming with multiple virtual address spaces. ASPLOS '16.

[92] Elnawawy, H., Chowdhury, R. B. R., Awad, A., and Byrd, G. T. Diligent TLBs: A Mechanism for Exploiting Heterogeneity in TLB Miss Behavior. In *Proceedings of the ACM International Conference on Supercomputing* (2019).

[93] Facebook. RocksDB. `http://rocksdb.org.`, 2017.

[94] Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research 9* (2008).

[95] Fang, Z., Zhang, L., Carter, J. B., Hsieh, W. C., and McKee, S. A. Reevaluating Online Superpage Promotion with Hardware Support. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture* (2001).

[96] Fedorova, A. Why mmap is faster than system calls. `https://sasha-f.medium.com/why-mmap-is-faster-than-system-calls-24718e75ab37`.

[97] Foundation, T. Spark. `http://spark.incubator.apache.org/`, 2014.

[98] Ganapathy, N., and Schimmel, C. General Purpose Operating System Support for Multiple Page Sizes. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (1998).

[99] Gandhi, J. *Efficient Memory Virtualization.* PhD thesis, The University of Wisconsin Madison, 2016.

[100] Gandhi, J., Basu, A., Hill, M. D., and Swift, M. M. BadgerTrap: A Tool to Instrument x86-64 TLB Misses. *SIGARCH Comput. Archit. News 42*, 2 (Sept. 2014).

[101] Gandhi, J., Basu, A., Hill, M. D., and Swift, M. M. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014).

[102] Gandhi, J., Hill, M. D., and Swift, M. M. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd International Symposium on Computer Architecture* (2016).

[103] Gaud, F., Lepers, B., Decouchant, J., Funston, J., Fedorova, A., and Quema, V. Large pages may be harmful on NUMA systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 231–242.

[104] Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium* (USA, 2012), Security'12, USENIX Association, p. 40.

[105] Goldberg, R. P. Survey of virtual machine research. *Computer 7*, 6 (1974), 34–45.

[106] Gorman, M., and Healy, P. Supporting Superpage Allocation Without Additional Hardware Support. In *Proceedings of the 7th International Symposium on Memory Management* (2008).

[107] Gras, B., Razavi, K., Bos, H., and Giuffrida, C. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *Proceedings of the 27th USENIX Conference on Security Symposium* (2018).

[108] Gregg, B. *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley Professional, 2019.

[109] Haria, S., Hill, M. D., and Swift, M. M. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems* (2018).

[110] Haria, S., Hill, M. D., and Swift, M. M. MOD: minimally ordered durable datastructures for persistent memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, pp. 775–788.

[111] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[112] Huang, J., Badam, A., Qureshi, M. K., and Schwan, K. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), ISCA '15, Association for Computing Machinery, p. 580–591.

[113] Humphries, J. T., Kaffes, K., Mazières, D., and Kozyrakis, C. A case against (most) context switches. HotOS '21, Association for Computing Machinery.

[114] Hwang, J., Vuppalapati, M., Peter, S., and Agarwal, R. Rearchitecting linux storage stack for µs latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021), USENIX Association, pp. 113–128.

[115] Intel. AVX-512 instructions. https://software.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html.

[116] Intel. How Intel Optimized RocksDB Code for Persistent Memory with PMDK. https://www.intel.com/content/www/us/en/developer/articles/technical/how-intel-optimized-rocksdb-code-for-persistent-memory\-with-pmdk.html.

[117] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. ://cdrdv2-public.intel.com/671488/248966$_{s}oftware_{o}ptimization_{m}anual.pdf$.

[118] Intel. Pmem-RocksDB. https://github.com/pmem/pmem-rocksdb.

[119] Intel. Intel(R) Optane(TM) DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2019.

[120] Intel. Intel® Architecture Instruction Set Extensions and Future Features Programming Reference. https://community.intel.com/legacyfs/online/drupal_files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, 2020.

[121] Intel. Intel reports second-quarter 2022 financial results. https://www.intel.com/content/www/us/en/newsroom/news/intel-reports-q2-2022-financial-results.html, 2022.

[122] Intel Corporation. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[123] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[124] Kadedodi, R., Kadekodi, S., Ponnapalli, S., Shirwadkar, H., Ganger, G., Kolli, A., and Chidambaram, V. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)* (October 2021).

[125] Kadekodi, R., Lee, S. K., Kashyap, S., Kim, T., Kolli, A., and Chidambaram, V. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), SOSP '19, Association for Computing Machinery, p. 494–508.

[126] Kadekodi, S., Nagarajan, V., and Ganger, G. R. Geriatrix: Aging what you see and what you don't see. a file system aging approach for modern storage systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 691–704.

[127] Kandiraju, G. B., and Sivasubramaniam, A. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture* (2002).

[128] Kang, W., Shin, D., and Yoo, S. Reinforcement learning-assisted garbage collection to mitigate long-tail latency in ssd. *ACM Transactions on Embedded Computing Systems 16*, 5s (sep 2017).

[129] Kannan, S., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Wang, Y., Xu, J., and Palani, G. Designing a true Direct-Access file system with DevFS. In *16th USENIX Conference on File and Storage Technologies (FAST 18)* (Oakland, CA, Feb. 2018), USENIX Association, pp. 241–256.

[130] Karakostas, V., Gandhi, J., Ayar, F., Cristal, A., Hill, M. D., McKinley, K. S., Nemirovsky, M., Swift, M. M., and Ünsal, O. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015).

[131] Karakostas, V., Gandhi, J., Cristal, A., Hill, M. D., McKinley, K. S., Nemirovsky, M., Swift, M. M., and Unsal, O. S. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), pp. 631–643.

[132] Katsakioris, C., Alverti, C., Karakostas, V., Nikas, K., Goumas, G., and Koziris, N. Faas in the age of (sub-)us i/o: A performance analysis of snapshotting. In *Proceedings of the 15th ACM International Conference on Systems and Storage* (2022), SYSTOR '22.

[133] kernel documentation, L. Memory protection keys. `https://www.kernel.org/doc/html/latest/core-api/protection-keys.html`.

[134] kernel documentation, L. Shared virtual addressing (sva) with enqcmd. `https://docs.kernel.org/x86/sva.html`, 2017.

[135] Khasawneh, K. N., Koruyeh, E. M., Song, C., Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *Proceedings of the 56th Annual Design Automation Conference* (2019).

[136] Kilburn, T., Edwards, D. B. G., Lanigan, M. J., and Sumner, F. H. One-level storage system. *IRE Transactions on Electronic Computers EC-11*, 2 (1962), 223–235.

[137] Kim, J.-H., Kim, J., Kang, H., Lee, C.-G., Park, S., and Kim, Y. Pnova: Optimizing shared file i/o operations of nvm file system on manycore servers. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems* (2019), APSys '19, Association for Computing Machinery, p. 1–7.

[138] Kivity, Avi. KVM: the Linux Virtual Machine Monitor. In *OLS '07: The 2007 Ottawa Linux Symposium* (2007).

[139] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy* (2019).

[140] Kogan, A., Dice, D., and Issa, S. Scalable range locks for scalable address spaces and beyond. In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[141] Kumar, M. K., Maass, S., Kashyap, S., Veselý, J., Yan, Z., Kim, T., Bhattacharjee, A., and Krishna, T. LATR: Lazy Translation Coherence. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems* (2018).

[142] Kwon, Y., Fingler, H., Hunt, T., Peter, S., Witchel, E., and Anderson, T. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, Association for Computing Machinery, p. 460–477.

[143] Kwon, Y., Yu, H., Peter, S., Rossbach, C. J., and Witchel, E. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016).

[144] Le, T. How the kernel manages your memory. `https://letrungthang.blogspot.com/2010/12/chuyen-tiet-kiem-mua-nha-cua-gioi.html`, 2017.

[145] Lee, G., Jin, W., Song, W., Gong, J., Bae, J., Ham, T. J., Lee, J. W., and Jeong, J. A case for hardware-based demand paging. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020), pp. 1103–1116.

[146] Lee, G., Shin, S., Song, W., Ham, T. J., Lee, J. W., and Jeong, J. Asynchronous I/O stack: A low-latency kernel I/O stack for Ultra-Low latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 603–616.

[147] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[148] Li, H., Berger, D. S., Novakovic, S., Hsu, L., Ernst, D., Zardoshti, P., Shah, M., Agarwal, I., Hill, M. D., Fontoura, M., and Bianchini, R. First-generation memory disaggregation for cloud platforms, 2022.

[149] Li, R., Ren, X., Zhao, X., He, S., Stumm, M., and Yuan, D. ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association, pp. 35–50.

[150] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium* (2018).

[151] Liu, J., Rebello, A., Dai, Y., Ye, C., Kannan, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), SOSP '21.

[152] Malliotakis, I., Papagiannis, A., Marazakis, M., and Bilas, A. Hugemap: Optimizing memory-mapped i/o with huge pages for fast storage. In *Euro-Par 2020: Parallel Processing Workshops* (2021), Springer International Publishing.

[153] Mansi, M., Tabatabai, B., and Swift, M. M. CBMM: Financial advice for kernel memory managers. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 593–608.

[154] Marathe, Y., Gulur, N., Ryoo, J. H., Song, S., and John, L. K. CSALT: Context Switch Aware Large TLB. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (2017).

[155] Margaritov, A., Ustiugov, D., Bugnion, E., and Grot, B. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (2019).

[156] Maruf, H. A., Wang, H., Dhanotia, A., Weiner, J., Agarwal, N., Bhattacharya, P., Petersen, C., Chowdhury, M., Kanaujia, S., and Chauhan, P. Tpp: Transparent page placement for cxl-enabled tiered memory, 2022.

[157] Merrifield, T., and Taheri, H. R. Performance Implications of Extended Page Tables on Virtualized X86 Processors. In *Proceedings of The 12th International Conference on Virtual Execution Environments* (2016).

[158] Michailidis, T., Delis, A., and Roussopoulos, M. MEGA: Overcoming Traditional Problems with OS Huge Page Management. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (2019).

[159] Min, C., Kashyap, S., Maass, S., and Kim, T. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 71–85.

[160] Nalli, S., Haria, S., Hill, M. D., Swift, M. M., Volos, H., and Keeton, K. An analysis of persistent memory use with whisper. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ASPLOS '17, Association for Computing Machinery, p. 135–148.

[161] Navarro, J., Iyer, S., Druschel, P., and Cox, A. L. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating System Design and Implementation* (2002).

[162] Neal, I., Zuo, G., Shiple, E., Khan, T. A., Kwon, Y., Peter, S., and Kasikci, B. Rethinking file mapping for persistent memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (Feb. 2021), USENIX Association, pp. 97–111.

[163] Ogleari, M., Yu, Y., Qian, C., Miller, E., and Zhao, J. String figure: A scalable and elastic memory network architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2019), pp. 647–660.

[164] Panwar, A., Bansal, S., and Gopinath, K. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).

[165] Panwar, A., Bansal, S., and Gopinath, K. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, Association for Computing Machinery.

[166] Panwar, A., Prasad, A., and Gopinath, K. Making Huge Pages Actually Useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems* (2018).

[167] Papadopoulou, M., Tong, X., Seznec, A., and Moshovos, A. Prediction-based superpage-friendly TLB designs. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture* (2015).

[168] Papagiannis, A., Marazakis, M., and Bilas, A. Memory-mapped i/o on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021), EuroSys '21, Association for Computing Machinery, p. 277–293.

[169] Papagiannis, A., Xanthakis, G., Saloustros, G., Marazakis, M., and Bilas, A. Optimizing memory-mapped I/O for fast storage devices. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 813–827.

[170] Papagiannis, A., Xanthakis, G., Saloustros, G., Marazakis, M., and Bilas, A. Optimizing memory-mapped i/o for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 813–827.

[171] Park, C., Cha, S., Kim, B., Kwon, Y., Black-Schaffer, D., and Huh, J. Perforated Page: Supporting Fragmented Memory Allocation for Large Pages. In *Proceedings of the IEEE 47th International Symposium on High Performance Computer Architecture* (2020).

[172] Park, C. H., Heo, T., Jeong, J., and Huh, J. Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017).

[173] Park, C. H., Vougioukas, I., Sandberg, A., and Black-Schaffer, D. Every walk's a hit: Making page walks single-access cache hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2022), ASPLOS '22, Association for Computing Machinery.

[174] Peleg, O., Morrison, A., Serebrin, B., and Tsafrir, D. Utilizing the IOMMU scalably. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, July 2015), USENIX Association, pp. 549–562.

[175] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.* (nov 2015).

[176] Pham, B., Bhattacharjee, A., Eckert, Y., and Loh, G. H. Increasing TLB reach by exploiting clustering in page translations. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture* (2014).

[177] Pham, B., Vaidyanathan, V., Jaleel, A., and Bhattacharjee, A. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012).

[178] Pham, B., Veselý, J., Loh, G. H., and Bhattacharjee, A. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways? In *Proceedings of the 48th International Symposium on Microarchitecture* (2015).

[179] Ram, V. S. S., Panwar, A., and Basu, A. Trident: Harnessing architectural resources for all page sizes in x86 processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2021), MICRO '21, Association for Computing Machinery, p. 1106–1120.

[180] The risc-v instruction set manual volume ii: Privileged architecture (chapter 5 "sv-napot" standard extension for napot translation contiguity, version 1.0, December 2021. https://wiki.riscv.org/display/HOME/RISC-V+Technical+Specifications.

[181] Rodrigues, G. File holes, races, and mmap(). `https://lwn.net/Articles/357767/`, 2009.

[182] Romanescu, B. F., Lebeck, A. R., Sorin, D. J., and Bracy, A. UNified Instruction/Translation/-Data (UNITD) coherence: One protocol to rule them all. In *Proceedings of the the 16th International Symposium on High-Performance Computer Architecture* (2010).

[183] Rybczyńska, M. Introducing maple trees. `https://lwn.net/Articles/845507/`, 2021.

[184] Ryoo, J. H., Gulur, N., Song, S., and John, L. K. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017).

[185] Ryoo, K. K. C., and Fan, G. Meeting petabyte scale memory systems challenges with cxl memory pooling. `https://memverge.com/cxl-forum-at-ocp-summit-2022/`, 2022.

[186] Samsung. Memory-semantic ssd. `https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation\-memory-solutions-at-flash-memory-summit-2022`, 2022.

[187] Samsung. Scalable memory development kit (smdk), 2022.

[188] Saulsbury, A., Dahlgren, F., and Stenström, P. Recency-based TLB Preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (2000).

[189] Savage, N. Going serverless. *Commun. ACM* (2018).

[190] Seznec, A. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. *IEEE Trans. Comput. 53*, 7 (July 2004).

[191] Sha, E. H.-M., Chen, X., Zhuge, Q., Shi, L., and Jiang, W. A new design of in-memory file system based on file virtual address framework. *IEEE Transactions on Computers 65*, 10 (2016), 2959–2972.

[192] Shun, J., and Blelloch, G. E. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2013).

[193] Skarlatos, D., Darbaz, U., Gopireddy, B., Kim, N. S., and Torrellas, J. Babelfish: Fusing address translations for containers. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020), pp. 501–514.

[194] Skarlatos, D., Kokolis, A., Xu, T., and Torrellas, J. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems* (2020).

[195] Song, N. Y., Son, Y., Han, H., and Yeom, H. Y. Efficient memory-mapped i/o on fast storage device. *ACM Trans. Storage 12*, 4 (May 2016).

[196] Song, N. Y., Yu, Y. J., Shin, W., Eom, H., and Yeom, H. Y. Low-latency memory-mapped i/o for data-intensive applications on fast storage devices. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis* (2012), pp. 766–770.

[197] Srikantaiah, S., and Kandemir, M. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *Proceedings of the 43rd Annual International Symposium on Microarchitecture* (2010).

[198] Stojkovic, J., Skarlatos, D., Kokolis, A., Xu, T., and Torrellas, J. Parallel virtualized memory translation with nested elastic cuckoo page tables. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022), ASPLOS '22.

[199] Stojkovic, J., Skarlatos, D., Kokolis, A., Xu, T., and Torrellas, J. Parallel virtualized memory translation with nested elastic cuckoo page tables. ASPLOS '22.

[200] Swanson, M., Stoller, L., and Carter, J. Increasing TLB Reach Using Superpages Backed by Shadow Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (1998).

[201] Swift, M. M. Towards o(1) memory. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (2017), HotOS '17, Association for Computing Machinery, p. 7–11.

[202] Talluri, M., and Hill, M. D. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (1994).

[203] Tarasov, V., Zadok, E., and Shepler, S. Filebench: A flexible framework for file system benchmarking. *login Usenix Mag. 41*, 1 (2016).

[204] Terpstra, D., Jagode, H., You, H., and Dongarra, J. J. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing* (2009).

[205] Torvalds, L. Linux kernel mailing list: mmap/mlock performance versus read. `https://marc.info/?l=linux-kernel&m=95496636207616&w=2`, 2000.

[206] Tramm, J. R., Siegel, A. R., Islam, T., and Schulz, M. XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future* (Kyoto, 2014).

[207] Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., and Grot, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), ASPLOS 2021, Association for Computing Machinery, p. 559–572.

[208] van Schaik, S., Giuffrida, C., Bos, H., and Razavi, K. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *Proceedings of the 27th USENIX Security Symposium* (2018).

[209] Villavieja, C., Karakostas, V., Vilanova, L., Etsion, Y., Ramirez, A., Mendelson, A., Navarro, N., Cristal, A., and Unsal, O. S. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (2011).

[210] Villavieja, C., Karakostas, V., Vilanova, L., Etsion, Y., Ramirez, A., Mendelson, A., Navarro, N., Cristal, A., and Unsal, O. S. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (USA, 2011), PACT '11, IEEE Computer Society, p. 340–349.

[211] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P., and Swift, M. M. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14, Association for Computing Machinery.

[212] Volos, H., Tack, A. J., and Swift, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI, Association for Computing Machinery, p. 91–104.

[213] VoltDB. Voltdb: Smart data fast. `https://www.voltactivedata.com/`, 2014.

[214] Wang, X., Zang, J., Wang, Z., Luo, Y., and Li, X. Selective Hardware/Software Memory Virtualization. In *Proceedings of the 7th ACM International Conference on Virtual Execution Environments* (2011).

[215] Wang, Z., Liu, X., Yang, J., Michailidis, T., Swanson, S., and Zhao, J. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2020), pp. 496–508.

[216] Wilcox, M. Add support for NV-DIMMs to ext4. `https://lwn.net/Articles/613384/`, 2014.

[217] Williams, D. mm: Randomize free memory. `https://lwn.net/Articles/764879/`.

[218] Wu, K., Guo, Z., Hu, G., Tu, K., Alagappan, R., Sen, R., Park, K., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (Feb. 2021), USENIX Association, pp. 307–323.

[219] Wu, X., Qiu, S., and Narasimha Reddy, A. L. Scmfs: A file system for storage class memory and its extensions. *ACM Trans. Storage 9*, 3 (Aug. 2013).

[220] Xu, J., Kim, J., Memaripour, A., and Swanson, S. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), ASPLOS '19, Association for Computing Machinery, p. 427–439.

[221] Xu, J., and Swanson, S. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (USA, 2016), FAST'16, USENIX Association, p. 323–338.

[222] Xu, J., Zhang, L., Memaripour, A., Gangadharaiah, A., Borase, A., Da Silva, T. B., Swanson, S., and Rudoff, A. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, Association for Computing Machinery, p. 478–496.

[223] Xu, Y., Solihin, Y., and Shen, X. Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), ASPLOS '20, Association for Computing Machinery, p. 987–1000.

[224] Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C. W., and Torrellas, J. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (2018).

[225] Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C. W., and Torrellas, J. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (2019).

[226] Yan, Z., Lustig, D., Nellans, D., and Bhattacharjee, A. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).

[227] Yan, Z., Lustig, D., Nellans, D., and Bhattacharjee, A. Translation Ranger: Operating System Support for Contiguity-aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture* (2019).

[228] Yan, Z., Veselý, J., Cox, G., and Bhattacharjee, A. Hardware Translation Coherence for Virtualized Systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017).

[229] Yan, Z., Veselý, J., Cox, G., and Bhattacharjee, A. Hardware translation coherence for virtualized systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), ISCA '17, Association for Computing Machinery, p. 430–443.

[230] Yang, J., Izraelevitz, J., and Swanson, S. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 221–234.

[231] Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J., and Swanson, S. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 169–182.

[232] Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K. L., and He, B. Nv-tree: Reducing consistency cost for nvm-based single level systems. FAST'15, USENIX Association, p. 167–181.

[233] Yang, Z., Harris, J. R., Walker, B., Verkamp, D., Liu, C., Chang, C., Cao, G., Stern, J., Verma, V., and Paul, L. E. Spdk: A development kit to build high performance storage applications.

In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (2017), pp. 154–161.

[234] Yaniv, I., and Tsafrir, D. Hash, Don'T Cache (the Page Table). In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (2016).

[235] Yoshimura, T., Chiba, T., and Horii, H. Evfs: User-level, event-driven file system for non-volatile memory. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems* (USA, 2019), HotStorage'19, USENIX Association, p. 16.

[236] Zhang, I., Raybuck, A., Patel, P., Olynyk, K., Nelson, J., Leija, O. S. N., Martinez, A., Liu, J., Simpson, A. K., Jayakar, S., Penna, P. H., Demoulin, M., Choudhury, P., and Badam, A. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21.

[237] Zhang, L., Speight, E., Rajamony, R., and Lin, J. Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation. In *Proceedings of the 24th ACM International Conference on Supercomputing* (2010).

[238] Zhang, Y., Oertel, R., and Rehm, W. Paging Method Switching for QEMU-KVM Guest Machine. In *Proceedings of the International Conference on Big Data Science and Computing* (2014).

[239] Zheng, T., Zhu, H., and Erez, M. SIPT: Speculatively Indexed, Physically Tagged Caches. In *IEEE International Symposium on High Performance Computer Architecture* (2018).

[240] Zijlstra, P. Tracking shared dirty pages. `https://lwn.net/Articles/185463/`, 2006.

[241] Zwisler, R. Add support for new persistent memory instruction. `https://lwn.net/Articles/619851/`, 2014.