National Technical University of Athens

School of Electrical and Computer Engineering

Division of Computer Science

Accelerating Virtual Memory with Hardware-Tailored Memory Management

Doctoral Dissertation

Stratos Psomadakis

Athens, December 2025

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

Accelerating Virtual Memory with Hardware-Tailored Memory Management

Doctoral Dissertation

Stratos Psomadakis

Advisory Committee: Georgios Goumas (Supervisor)
Nectarios Koziris
Nikolaos Papaspyrou

Approved by the seven-member examination committee on December 19th, 2025.

. . . .	. . . .	. . . .
Georgios Goumas Professor National Technical University of Athens	Nectarios Koziris Professor National Technical University of Athens	Nikolaos Papaspyrou Professor National Technical University of Athens
. . . .	. . . .	. . . .
Dionisios N. Pnevmatikatos Professor National Technical University of Athens	Angelos Bilas Professor University of Crete	Vasileios Karakostas Assistant Professor University of Athens
	. . . .	
	Nikos Vasilakis Assistant Professor Brown University	

Athens, December 2025

. . . .

Stratos Psomadakis
Doctor of Electrical and Computer Engineering, NTUA

For the Quest is achieved, and now all is over.
— J. R. R. Tolkien

Abstract

Since its inception in the 1960s, the paged virtual memory abstraction has become ubiquitous in computer systems. Paged virtual memory decouples the process address space from the physical memory of the system and shifts the burden of managing physical memory from userspace to the Operating System (OS). While this greatly simplifies userspace programming and enables the centralized and efficient physical memory management by the OS, it is not without trade-offs. Address Translation (AT), the process of mapping pages of virtual addresses to physical memory frames, incurs significant performance overhead. Hardware virtualization, an abstraction that has, in recent decades, become as ubiquitous as virtual memory, exacerbates this overhead. Additionally, virtualized execution creates a semantic gap between the virtual memory subsystems of the virtualized (guest) and the physical (host) OS instances, which can further affect performance. The contribution of this thesis lies in enabling the OS to take advantage of the virtual memory hardware (HW) via HW-tailored memory management to address both problems.

The first part of the thesis focuses on the AT overhead of paged virtual memory and specifically on the impact of translation granularity on the performance of virtual memory, as well as on the interplay between translation granularity and OS memory management. The thesis surveys the available HW mechanisms that enable multi-grained address translation, namely large pages and TLB coalescing, and the existing OS interfaces that support these mechanisms in order to enable the OS to utilize multiple translation granules. The survey uncovers inherent limitations in these OS interfaces that hamper the effectiveness of the underlying HW mechanisms and the ability of the OS memory manager to effectively and efficiently utilize multiple translation granules. The thesis then proposes *Elastic Translations (ET)*, a novel approach that adopts a markedly different approach regarding the way the OS interfaces with the aforementioned HW mechanisms, and enables the OS to seamlessly and optimally use and select among all HW-supported translation granules. Implementing ET in Linux for the ARMv8-A architecture and evaluating its performance on a real ARMv8-A server shows that ET unlock the full potential of the OS-assisted TLB coalescing feature of the ARMv8-A architecture and enable the OS memory manager to optimally utilize the extended range of translation granules, unlocked by coalescing, even when memory is fragmented, outperforming both state-of-practice and state-of-the-art approaches.

The second part of the thesis focuses on bridging the semantic gap between the virtual memory subsystems of physical (host) and virtualized (guest) OS instances via the virtual memory hardware, i.e., the Memory Management Unit (MMU). This semantic gap becomes particularly apparent and problematic when using Virtual Machine (VM) memory snapshots to accelerate VM boot times, a technique heavily employed in latency-sensitive scenarios such as Function-as-a-Service (FaaS) serverless computing. As VM snapshots are stored as files on disk, the OS on the physical host treats the entire guest memory as a file, being oblivious to the ephemeral anonymous allocations performed by the guest memory subsystem. The thesis proposes a lightweight paravirtualized (PV) mechanism, *AnonPTEs*, to enlighten the host OS regarding the nature of the guest VM memory allocations by taking advantage of the virtual memory hardware. *AnonPTEs* piggyback on the VM AT structures, i.e., the guest VM page tables, to create a mirrored guest physical address space for the ephemeral anonymous allocations of the VM, enabling the host OS to detect such allocations and serve them using anonymous memory instead of unnecessarily fetching memory pages from the VM memory snapshot file on disk. The integration of AnonPTEs with SnapBPF, an eBPF-based state-of-the-art snapshot prefetcher for VM-sandboxed serverless functions, and its subsequent evaluation using representative serverless workloads shows that AnonPTEs eliminate gratuitous major page faults for anonymous VM allocations, thereby improving the tail latency for VM-sandboxed serverless functions.

Keywords: Virtual Memory, Address Translation, Memory Management, Virtualization, Operating Systems, Computer Architecture, Serverless Computing

Περίληψη

Ο μηχανισμός της εικονικής μνήμης με σελιδοποίηση, ο οποίος υποστηρίζεται πρακτικά από όλα τα σύγχρονα υπολογιστικά συστήματα, απεμπλέκει τον προγραμματιστή από τη διαχείριση της φυσικής μνήμης, προσφέροντας ένα απλουστευμένο προγραμματιστικό μοντέλο και επιτρέποντας την κεντρική και αποδοτική διαχείριση της φυσικής μνήμης από το Λειτουργικό Σύστημα (ΛΣ). Ωστόσο, ο μηχανισμός της εικονικής μνήμης εμπεριέχει και συμβιβασμούς, καθώς μπορεί να επηρεάσει αρνητικά την επίδοση του συστήματος. Ένας από τους πιο επιβαρυντικούς παράγοντες είναι η μετάφραση διευθύνσεων, η διαδικασία κατά την οποία εικονικές σελίδες αντιστοιχίζονται σε φυσικές διευθύνσεις. Η εικονικοποίηση του υλικού μέσω εικονικών μηχανών, η οποία τις τελευταίες δεκαετίες έχει γίνει σχεδόν τόσο διαδεδομένη όσο και η εικονική μνήμη, επιβαρύνει περαιτέρω την επίδοση της μετάφρασης διευθύνσεων, συνεπώς και της εικονικής μνήμης. Επιπλέον, η εικονικοποίηση δημιουργεί ένα σημασιολογικό κενό μεταξύ των υποσυστημάτων εικονικής μνήμης του εικονικοποιημένου και του μη εικονικοποιημένου ΛΣ. Το κενό αυτό επιδρά αρνητικά συνολικά στην αποτελεσματικότητα και αποδοτικότητα της εικονικής μνήμης. Η συμβολή αυτής της διατριβής έγκειται στο να επιτρέψει στο ΛΣ να εκμεταλλευτεί το υλικό της εικονικής μνήμης μέσω πολιτικών διαχείρισης μνήμης προσαρμοσμένων στο υλικό για την αντιμετώπιση και των δύο αυτών προβλημάτων.

Το πρώτο μέρος της διατριβής εστιάζει στην αλληλεπίδραση του μεγέθους του κβάντου στο οποίο γίνεται η μετάφραση διευθύνσεων με τους αντίστοιχους μηχανισμούς και τις διεπαφές του ΛΣ. Συγκεκριμένα, γίνεται μελέτη των μηχανισμών υλικού που επιτρέπουν την αύξηση του μεγέθους του κβάντου μετάφρασης διευθύνσεων και αξιολογούνται οι διεπαφές του ΛΣ για την αξιοποίηση των συγκεκριμένων μηχανισμών. Από τη μελέτη αυτή προκύπτουν εγγενείς περιορισμοί των υπαρχουσών διεπαφών, που περιορίζουν την βέλτιστη αξιοποίηση των μηχανισμών αυτών. Για τον λόγο αυτό, η διατριβή προτείνει τον μηχανισμό των Elastic Translations (ET), μια νέα προσέγγιση που επιτρέπει την διαφανή και αποδοτική υποστήριξη των προαναφερθέντων μηχανισμών υλικού από το ΛΣ με σκοπό την βέλτιστη αξιοποίηση όλων των διαθέσιμων κβάντων που οι μηχανισμοί αυτοί υποστηρίζουν. Η υλοποίηση των ET στο Λειτουργικό Σύστημα Linux για την αρχιτεκτονική ARMv8-A και η αξιολόγηση της απόδοσής τους σε ένα πραγματικό ARMv8-A μηχάνημα, δείχνει ότι τα ET επιτρέπουν στο ΛΣ να αξιοποιήσει βέλτιστα το εκτεταμένο εύρος κβάντων μετάφρασης που υποστηρίζεται από το υλικό, βελτιώνοντας σημαντικά την επίδοση της μετάφρασης διευθύνσεων σε σχέση τόσο με καθιερωμένους μηχανισμούς όσο και με τις πλέον σύγχρονες προσεγγίσεις.

Το δεύτερο μέρος της διατριβής επικεντρώνεται στην βελτίωση της επίδοσης της εικονικής μνήμης στην περίπτωση εικονικοποιημένης εκτέλεσης, και συγκεκριμένα στη γεφύρωση του σημασιολογικού κενού μεταξύ των υποσυστημάτων εικονικής μνήμης του εικονικοποιημένου και του μη εικονικοποιημένου ΛΣ, μέσω του υλικού εικονικής μνήμης. Αυτό το σημασιολογικό κενό γίνεται ιδιαίτερα εμφανές και προβληματικό στην περίπτωση όπου χρησιμοποιούνται στιγμιότυπα μνήμης για την επιτάχυνση του χρόνου εκκίνησης των εικονικών μηχανών, μια τεχνική που χρησιμοποιείται ευρέως σε καταστάσεις όπου η καθυστέρηση απόκρισης επηρεάζει σημαντικά την επίδοση, όπως στον υπολογισμό χωρίς διακομιστή. Καθώς τα στιγμιότυπα μνήμης των εικονικών μηχανών αποθηκεύονται ως αρχεία στον δίσκο, το μη εικονικοποιημένο ΛΣ αντιμετωπίζει ολόκληρη τη μνήμη της εικονικής μηχανής ως ένα αρχείο, αγνοώντας τις εφήμερες ανώνυμες εκχωρήσεις μνήμης που πραγματοποιούνται από το υποσύστημα μνήμης της εικονικής μηχανής. Η διατριβή προτείνει έναν μηχανισμό βασισμένο στην παραεικονικοποίηση, τα AnonPTEs, ώστε να γεφυρώσει τα συστήματα εικονικής μνήμης του εικονικοποιημένου και μη εικονικοποιημένου ΛΣ, επιτρέποντας στο μη εικονικοποιημένο ΛΣ να γνωρίζει τη φύση των εκχωρήσεων μνήμης της εικονικής μηχανής. Τα AnonPTEs εκμεταλλεύονται το υλικό εικονικής μνήμης, μέσω των δομών μετάφρασης διευθύνσεων της εικονικής μηχανής, συγκεκριμένα τους πίνακες σελίδων, για να δημιουργήσουν έναν κατοπτρικό φυσικό χώρο διευθύνσεων για τις εφήμερες ανώνυμες εκχωρήσεις μνήμης της εικονικής μηχανής, επιτρέποντας στο μη εικονικοποιημένο ΛΣ να ανιχνεύει τέτοιες εκχωρήσεις και να τις εξυπηρετεί χρησιμοποιώντας ανώνυμη μνήμη, αντί να ανακτά άσκοπα σελίδες από το αρχείο στιγμιότυπου μνήμης της εικονικής μηχανής από τον δίσκο. Η πειραματική αξιολόγηση των AnonPTEs δείχνει ότι, εξαλείφοντας τα μείζονα σφάλματα σελίδας για τις ανώνυμες εκχωρήσεις μνήμης της εικονικής μηχανής, μειώνεται δραστικά η μέγιστη καθυστέρηση απόκρισης όταν χρησιμοποιούνται εικονικές μηχανές με στιγμιότυπα μνήμης, βελτιώνοντας τη συνολική απόδοση του συστήματος.

Λέξεις Κλειδιά: Εικονική Μνήμη, Μετάφραση Διευθύνσεων, Διαχείριση Μνήμης, Εικονικοποίηση, Λειτουργικά Συστήματα, Αρχιτεκτονική Υπολογιστών, Υπολογισμός Χωρίς Διακομιστή

Acknowledgments

I would like to sincerely thank the members of the thesis advisory and examination committees – Professor Georgios Goumas, Professor Nectarios Koziris, Professor Nikolaos Papaspyrou, Professor Dionisios Pnevmatikatos, Professor Angelos Bilas, Assistant Professor Vasileios Karakostas and Assistant Professor Nikos Vasilakis – for their time, feedback and assistance in shaping this doctoral dissertation.

I am especially grateful to my thesis supervisor, Professor Georgios Goumas, for his guidance, Professor Nectarios Koziris, with whom I had the luck to work since the days of my Diploma thesis, and Assistant Professor Vasileios Karakostas, whose enthusiasm for the thesis was always a driving force.

The completion of the dissertation would not have been possible without my colleagues and friends at the Computing Systems Laboratory (CSLab), who made the, at times grueling, PhD journey so much more enjoyable and interesting. I would especially like to thank Dr. Chloe Alverti, whose support, both practical and moral, was indeed invaluable, Dr. Orestis Lagkas Nikolos, Dr. Dimitrios Siakavaras, Dr. Konstantinos Nikas and Christos Katsakioris, with whom I collaborated closely throughout the thesis, as well as Dr. Anastassios (Tasos) Nanos, Konstantinos (Kostis) Papazafeiropoulos and Dr. Stefanos Gerangelos, with whom I worked when I first started my PhD studies. I want to particularly thank Tasos, with whom I had the luck to work since I was an undergraduate, and who was pivotal in my decision to pursue a PhD.

I would also like to thank the undergraduate students, who conducted their Diploma thesis at the lab, with whom I had the chance to work and explore many interesting research directions.

The PhD journey would have been insufferable without the companionship of my friends – Dimitris, Thodoris, Giorgos, and the Tartoufos plenary: Andreas, Chloe, Dimitris, Ioanna, Kostis, Orestis, and the two Panagiotides!

Finally, I would like to thank my family, my sister and especially my parents, whose support has been unwavering and unconditional.

Εκτεταμένη περίληψη στην Ελληνική γλώσσα

Εισαγωγή

Ο μηχανισμός της εικονικής μνήμης με σελιδοποίηση αποτελεί έναν από τους θεμελιώδεις δομικούς λίθους των σύγχρονων υπολογιστικών συστημάτων. Η εικονική μνήμη απεμπλέκει τον προγραμματιστή από τη διαχείριση της φυσικής μνήμης, προσφέροντας ένα απλουστευμένο προγραμματιστικό μοντέλο και επιτρέποντας την κεντρική και αποδοτική διαχείριση της φυσικής μνήμης από το λειτουργικό σύστημα (ΛΣ). Ωστόσο, ο μηχανισμός της εικονικής μνήμης ενδέχεται, κατά περιστάσεις, να επιβαρύνει την επίδοση του συστήματος.

Ένας από τους πιο επιβαρυντικούς παράγοντες του μηχανισμού της εικονικής μνήμης είναι η μετάφραση διευθύνσεων, η διαδικασία κατά την οποία εικονικές σελίδες αντιστοιχίζονται σε φυσικές διευθύνσεις. Καθώς οι απαιτήσεις σε μνήμη των σύγχρονων εφαρμογών συνεχώς αυξάνονται, ξεπερνώντας την εμβέλεια του Translation Lookaside Buffer (TLB), το κόστος μετάφρασης διευθύνσεων, που επιβαρύνει τις εφαρμογές, αυξάνεται σταθερά. Η εικονικοποίηση του υλικού, μέσω εικονικών μηχανών, η οποία τις τελευταίες δεκαετίες έχει γίνει σχεδόν τόσο διαδεδομένη όσο και η εικονική μνήμη, επιβαρύνει περαιτέρω την επίδοση της μετάφρασης διευθύνσεων λόγω της ένθετης μορφής που πλέον έχει ο πινάκας σελίδων (nested page table). Επιπλέον, η εικονικοποίηση δημιουργεί ένα σημασιολογικό κενό μεταξύ των υποσυστημάτων εικονικής μνήμης του εικονικοποιημένου και του μη εικονικοποιημένου ΛΣ, το οποίο έχει ιδιαίτερο αντίκτυπο στην απόδοση του συστήματος, στην περίπτωση όπου χρησιμοποιούνται στιγμιότυπα μνήμης για την επιτάχυνση του χρόνου εκκίνησης των εικονικών μηχανών, μια τεχνική που χρησιμοποιείται ευρέως σε καταστάσεις όπου η καθυστέρηση απόκρισης επηρεάζει σημαντικά την επίδοση, όπως στον υπολογισμό χωρίς διακομιστή.

Η συμβολή αυτής της διατριβής έγκειται στην ανάπτυξη μηχανισμών που επιτρέπουν στο ΛΣ να εκμεταλλευτεί το υλικό της εικονικής μνήμης μέσω πολιτικών διαχείρισης μνήμης προσαρμοσμένων στο υλικό, για την αντιμετώπιση και των δύο αυτών προβλημάτων.

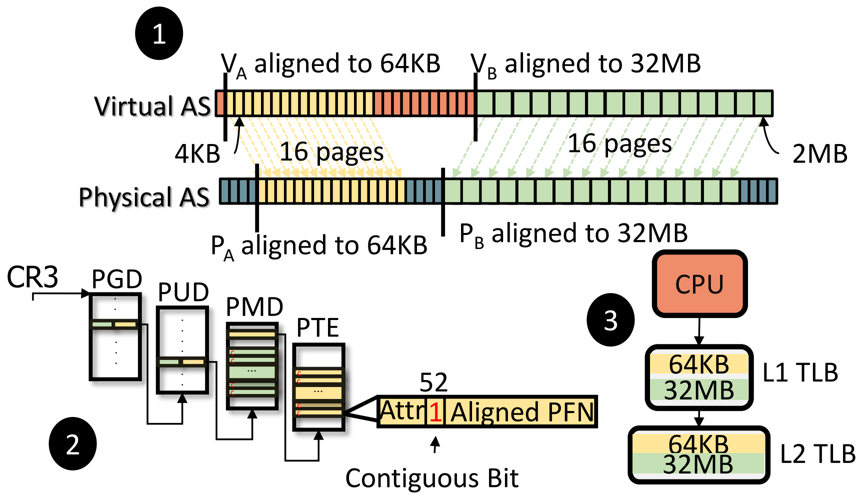
Μετάφραση Διευθύνσεων με Πολλαπλά Μεγέθη

Κατά τον σχεδιασμό συστημάτων εικονικής μνήμης με σελιδοποίηση, μία από τις πλέον κρίσιμες αποφάσεις αφορά το μέγεθος του κβάντου στο οποίο γίνεται η μετάφραση διευθύνσεων, δηλαδή το μέγεθος της σελίδας. Η αύξηση του μεγέθους σελίδας μειώνει τον αριθμό των μεταφράσεων, μετριάζοντας το κόστος της μετάφρασης διευθύνσεων. Ωστόσο, τα μεγαλύτερα μεγέθη σελίδων εντείνουν τον εσωτερικό κατακερματισμό.

Αρχιτεκτονική	**Μεγέθη Μετάφρασης**
x86	4KiB - 2MiB - 1GiB
ARMv8-A	4KiB - **64KiB** - 2MiB - **32MiB** - 1GiB
RISC-V	4KiB - **8KiB .. 1MiB** - 2MiB - **4MiB .. 512MiB** - 1GiB

Πίνακας 1: Υποστηριζόμενα μεγέθη μετάφρασης για σύγχρονες αρχιτεκτονικές επεξεργαστών. Τα ενδιάμεσα μεγέθη (με έντονη γραφή) υποστηρίζονται μέσω OS-assisted TLB coalescing.

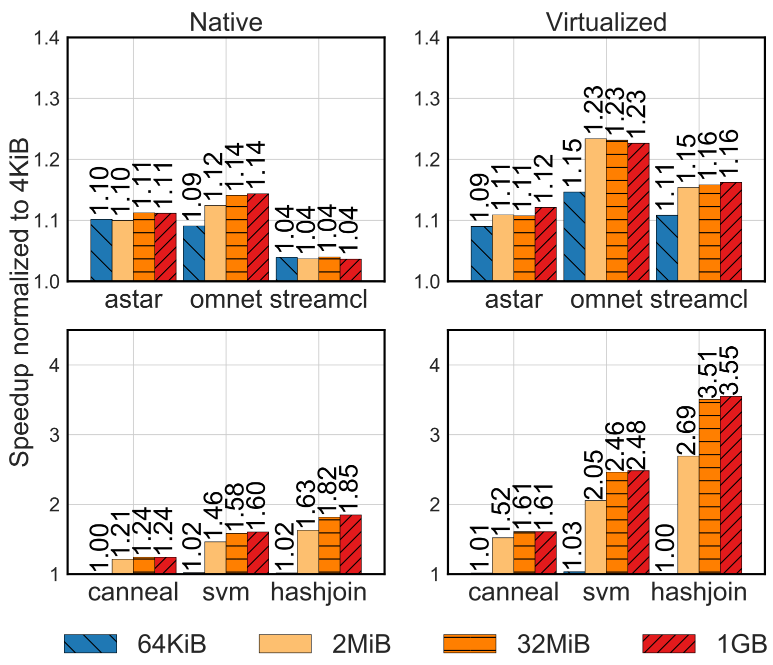
Σύγχρονες αρχιτεκτονικές, όπως η x86, η ARMv8-A και η RISC-V, υποστηρίζουν μεγάλες σελίδες μέσω της αποθήκευσης μεγαλύτερων μεταφράσεων σε υψηλότερα επίπεδα του πολυεπίπεδου πίνακα σελίδων. Ο Πίνακας 1 συνοψίζει τα υποστηριζόμενα μεγέθη μετάφρασης από τις τρεις αυτές αρχιτεκτονικές επεξεργαστών.



Σχήμα 1: OS-assisted TLB coalescing στην αρχιτεκτονική ARMv8-A.

Οι αρχιτεκτονικές ARMv8-A και RISC-V υποστηρίζουν έναν μηχανισμό γνωστό ως OS-assisted TLB coalescing, ο οποίος επιτρέπει ενδιάμεσα μεγέθη μετάφρασης. Το Σχήμα 1 απεικονίζει πως λειτουργεί ο μηχανισμός αυτός στην αρχιτεκτονική ARMv8-A. Το contiguous bit (bit 52) στις εγγραφές του πίνακα σελίδων επιτρέπει στο ΛΣ να σηματοδοτήσει στο υλικό της εικονικής μνήμης ότι 16 διαδοχικές σελίδες είναι συνεχόμενες στην φυσική μνήμη και μπορούν να συνενωθούν σε μία μόνο καταχώρηση στο TLB, δημιουργώντας έτσι μεταφράσεις μεγέθους 64KiB (για PTEs) και 32MiB (για PMDs).

Περιορισμοί Υφιστάμενων Διεπαφών ΛΣ



Σχήμα 2: Επίδοση ενδιάμεσων μεγεθών μετάφρασης (μέσω Linux HugeTLB) σε μη κατακερματισμένο διακομιστή ARMv8-A.

Η πειραματική αξιολόγηση των ενδιάμεσων μεγεθών μετάφρασης σε έναν διακομιστή ARMv8-A (Σχήμα 2) αποκαλύπτει σημαντικές δυνατότητες επίδοσης. Για εφαρμογές με μικρές απαιτήσεις σε μνήμη, οι οποίες ωστόσο λόγω ακανόνιστων προσβάσεων στη μνήμη επιβαρύνονται με αυξημένο κόστος μετάφρασης όταν χρησιμοποιούν 4KiB σελίδες, οι μεταφράσεις 64KiB βελτιώνουν την επίδοσή τους έως και 15%, πλησιάζοντας την επίδοση των σελίδων 2MiB, χωρίς την επιβάρυνση σε κατακερματισμό που επιφέρουν οι μεγαλύτερες 2MiB σελίδες. Για εφαρμογές με μεγαλύτερες απαιτήσεις σε μνήμη, οι μεταφράσεις 32MiB υπερτερούν των σελίδων 2MiB κατά έως και 30% σε εικονικοποιημένη εκτέλεση.

Ωστόσο, τα υφιστάμενα σύγχρονα λειτουργικά συστήματα, όπως το Linux και το FreeBSD, υποστηρίζουν διαφανώς μόνο σελίδες 2MiB, για παράδειγμα μέσω του μηχανισμού των Transparent Huge Pages (THP) στο Linux. Η υποστήριξη σελίδων 1GiB περιορίζεται πίσω από ρητές, μη διαφανείς διεπαφές, όπως το Linux HugeTLB. Η υποστήριξη για ενδιάμεσα μεγέθη μετάφρασης (64KiB, 32MiB) είναι περιορισμένη ή ανύπαρκτη, και οι πολιτικές επιλογής μεγέθους είναι απλοϊκές.

Elastic Translations: Σχεδιασμός

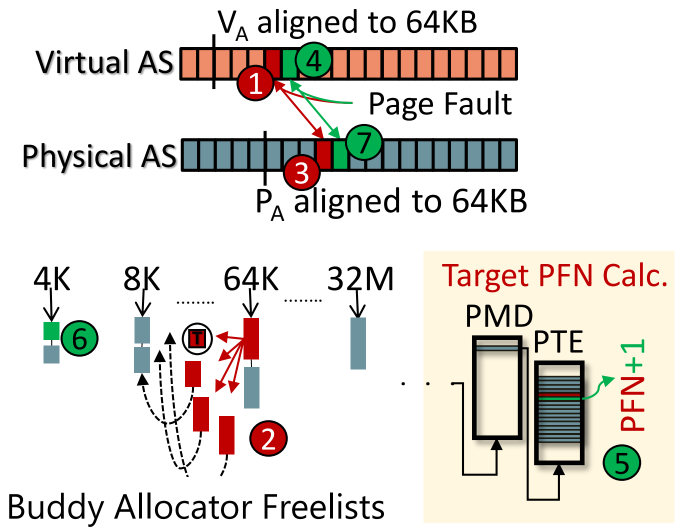
Συστατικό	**Σκοπός**	**Κύρια Συνεισφορά**
Transparent Contig-Bit	Διαφανής δημιουργία μεταφράσεων 64KiB και 32MiB	Υποστήριξη εικονικοποίησης για όλα τα μεγέθη
CoalaPaging	Ευκαιριακή δημιουργία συνεχόμενων αντιστοιχίσεων κατά την εξυπηρέτηση σφαλμάτων σελίδας	Πρακτική και αποτελεσματική πολιτική εκχώρησης 32MiB μεταφράσεων
CoalaKhugepaged	Ασύγχρονη δημιουργία μεταφράσεων 64KiB και 32MiB	Δημιουργία μεταφράσεων υπό κατακερματισμό
Leshy	Επιλογή μεγέθους μετάφρασης μέσω εκτίμησης επιβάρυνσης	Βέλτιστη επιλογή μεγέθους μέσω δειγματοληψίας αστοχιών TLB

Πίνακας 2: Οι συνιστώσες των Elastic Translations.

Το πρώτο μέρος της διατριβής προτείνει τα *Elastic Translations (ET)*, μια νέα προσέγγιση που επιτρέπει στο ΛΣ να αξιοποιεί διαφανώς και βέλτιστα όλα τα υποστηριζόμενα από το υλικό μεγέθη μετάφρασης. Ο Πίνακας 2 συνοψίζει τις τέσσερις συνιστώσες των ET.

Διαφανής Διαχείριση Contiguous Bit

Τα ET επεκτείνουν τον διαχειριστή μνήμης του Linux και το KVM για να ανιχνεύουν αυτόματα πότε ομάδες συνεχόμενων σελίδων ικανοποιούν τις απαιτήσεις ευθυγράμμισης και συνέχειας, που απαιτεί ο μηχανισμός του TLB coalescing. Όποτε δημιουργείται ή τροποποιείται μια εγγραφή στον πίνακα σελίδων κάποιας διεργασίας, τα ET ελέγχουν τις γειτονικές της σελίδες. Εάν κάθε σελίδα, από τις 16 που ανήκουν σε αυτό το coalescing γκρουπ, είναι: (i) κατάλληλα ευθυγραμμισμένη, (ii) συνεχόμενη με την προηγούμενη στον φυσικό χώρο μνήμης, και (iii) έχει συμβατές ιδιότητες και δικαιώματα πρόσβασης με τις γειτονικές της, τα ET θέτουν το contiguous bit, επιτρέποντας στο υλικό εικονικής μνήμης, δηλαδή στο TLB, να συμπτύξει τις 16 αυτές σελίδες σε μία μετάφραση στο υλικό. Για εικονικοποιημένη εκτέλεση, τα ET επεκτείνουν το KVM ώστε το contiguous bit να αντικατοπτρίζεται αυτόματα και στους ένθετους πίνακες σελίδων.



Σχήμα 3: Coalescing-aware δέσμευση μνήμης μέσω του CoalaPaging.

CoalaPaging: Coalescing-aware Δέσμευση Μνήμης

Για τη δημιουργία της συνέχειας που απαιτείται για τις ενδιάμεσες μεταφράσεις, η διατριβή σχεδιάζει το *CoalaPaging*, μια coalescing-aware πολιτική εκχώρησης μνήμης. Το Σχήμα 3 απεικονίζει τη διαδικασία.

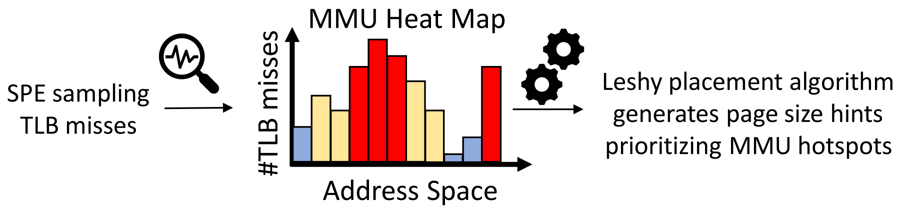
Για το πρώτο σφάλμα εντός μιας περιοχής 64KiB ή 32MiB, το CoalaPaging αναζητά ένα ελεύθερο μπλοκ κατάλληλου μεγέθους, εκχωρεί τη σελίδα που αντιστοιχεί στην διεύθυνση σφάλματος μέσα στην περιοχή αυτή, και επιστρέφει τις υπόλοιπες σελίδες πίσω στον ΛΣ. Για επόμενα σφάλματα, σαρώνει τις εγγραφές του πίνακα σελίδων για αυτή την περιοχή για να βρει μια προηγουμένως εκχωρημένη σελίδα, την οποία χρησιμοποιεί ως anchor, και να υπολογίσει την σελίδα που πρέπει να εκχωρήσει ώστε να εξασφαλίσει την συνέχεια στην φυσική μνήμη.

CoalaKhugepaged: Ασύγχρονη Δημιουργία Συνεχόμενων Μεταφράσεων

Τα ΕΤ γενικεύουν τον μηχανισμό του khugepaged του Linux, έτσι ώστε να συνενώνει μερικώς συνεχόμενες περιοχές σε πλήρως συνενωμένες αντιστοιχίσεις 64KiB και 32MiB. Ο CoalaKhugepaged λειτουργεί συνεργατικά με το CoalaPaging, εκμεταλλευόμενος τη μερική συνέχεια για να μειώσει σημαντικά τις σελίδες που θα πρέπει να μετακινηθούν (αντιγραφούν).

Leshy: Επιλογή Μεγέθους με Καθοδήγηση Υλικού

Τα ΕΤ εισάγουν το *Leshy*, έναν profiler που αξιοποιεί την επέκταση στατιστικής δειγματοληψίας (SPE) της αρχιτεκτονικής ARMv8-A για να συλλέξει δειγματοληπτικά τις αστοχίες TLB των εκτελούμενων εφαρμογών. Το Σχήμα 4 απεικονίζει τη διαδικασία.


SPE sampling TLB misses
MMU Heat Map
#TLB misses
Address Space
Leshy placement algorithm generates page size hints prioritizing MMU hotspots

Σχήμα 4: Η λειτουργία του Leshy profiler.

Το Leshy αναλύει τις αστοχίες TLB και δημιουργεί έναν χάρτη κόστους μετάφρασης του χώρου διευθύνσεων κάθε εφαρμογής. Ταξινομεί τις περιοχές του χώρου διευθύνσεων με βάσει την πίεση που προκαλούν στο υλικό μετάφρασης και επιχειρεί να αντιστοιχίσει βέλτιστα το σύνολο εργασίας της εφαρμογής στα υποστηριζόμενα από το υλικό μεγέθη μετάφρασης. Η αντιστοίχιση αυτή μεταφέρεται στο ΛΣ μέσω μιας επέκτασης στην κλήση συστήματος madvise().

Elastic Translations: Πειραματική Αξιολόγηση

Τα ET υλοποιούνται στο Linux v5.18 για την αρχιτεκτονική ARMv8-A και αξιολογούνται σε έναν διακομιστή Ampere Altra με επεξεργαστές Neoverse N1 (256GiB μνήμη ανά NUMA κόμβο).

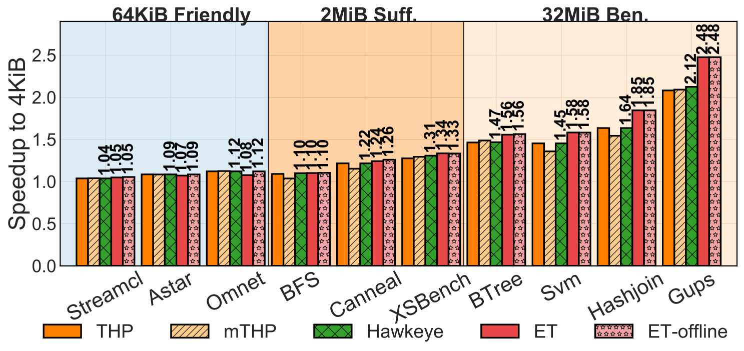
Μη Εικονικοποιημένη Εκτέλεση

Το Σχήμα 5 παρουσιάζει τα αποτελέσματα για μη εικονικοποιημένη εκτέλεση σε έναν διακομιστή χωρίς κατακερματισμό. Για εφαρμογές με μικρές απαιτήσεις σε μνήμη (Astar, Omnetpp, Streamcluster), τα ET χρησιμοποιούν το CoalaPaging για να δημιουργήσουν ευκαιριακά μεταφράσεις 64KiB, μειώνοντας σημαντικά τις αστοχίες TLB.

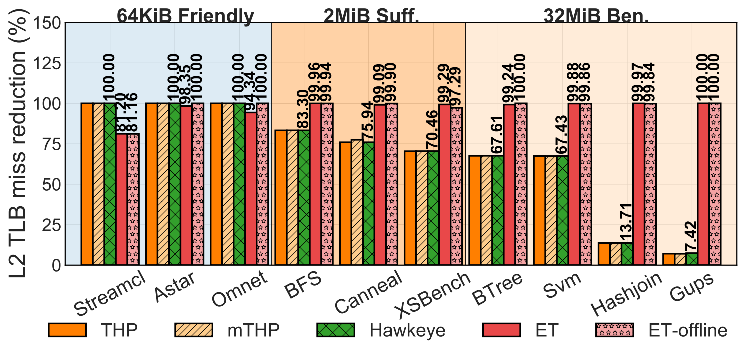
Για εφαρμογές με μεγάλα αποτυπώματα μνήμης και ακανόνιστα μοτίβα προσπέλασης της μνήμης (BTree, SVM, Hashjoin, Gups), τα ET εξαλείφουν τις αστοχίες TLB, χρησιμοποιώντας μεταφράσεις 32MiB για να καλύψουν το 97-99% του αποτυπώματός τους. Αυτό βελτιώνει την επίδοση κατά 19% κατά μέσο όρο και έως 39% έναντι του THP.

Εικονικοποιημένη Εκτέλεση

Το Σχήμα 6 παρουσιάζει τα αποτελέσματα για εικονικοποιημένη εκτέλεση σε διακομιστή χωρίς κατακερματισμό. Παρά την ευκαιριακή του φύση, το CoalaPaging καταφέρνει να δημιουργήσει αποτελεσματικά συνεχόμενες μεταφράσεις 64KiB και 32MiB τόσο στον guest όσο και στον host. Αυτό μεταφράζεται σε σημαντική βελτίωση στην επίδοση για εφαρμογές με μεγάλες απαιτήσεις σε μνήμη: 30% κατά μέσο όρο και έως 150% έναντι του THP.



(α΄) Επιτάχυνση



(β΄) Μείωση αστοχιών TLB

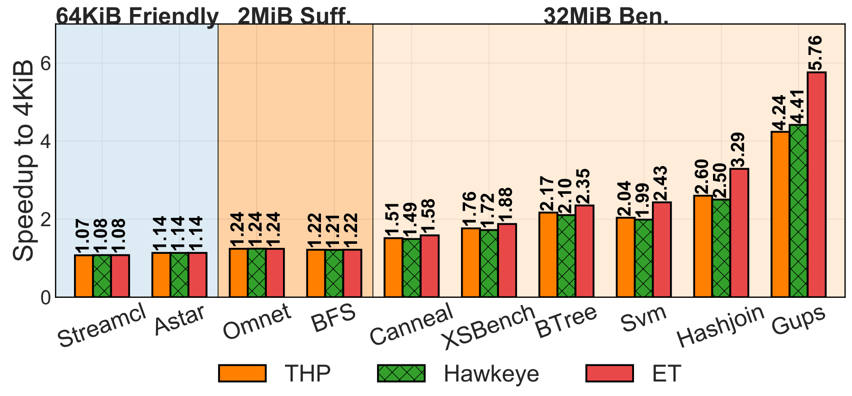
Σχήμα 5: Επίδοση των Elastic Translations σε μη κατακερματισμένο κόμβο για μη εικονικοποιημένη εκτέλεση.

Εξωτερικός Κατακερματισμός

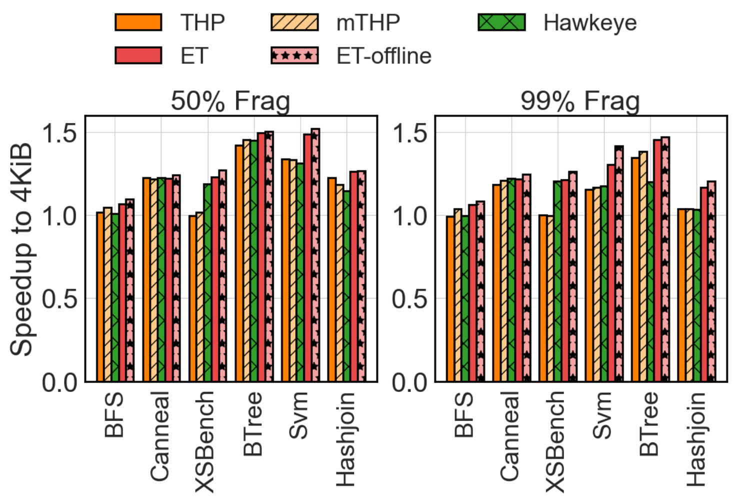
Σε διακομιστή με κατακερματισμένη μνήμη, τα ET υπερτερούν τόσο του THP όσο και του HawkEye (state-of-the-art). Το Σχήμα 7 παρουσιάζει τα αποτελέσματα για δύο σενάρια κατακερματισμού (50% και 99%). Το Leshy εντοπίζει επιτυχώς τις περιοχές μνήμης που απαιτούν μεγαλύτερες μεταφράσεις κατά την εκτέλεση της εφαρμογής και δίνει προτεραιότητα στο να προαχθούν σε 32MiB μεταφράσεις. Η επίδοση βελτιώνεται κατά 12% κατά μέσο όρο και έως 20% έναντι του THP, ενώ μειώνεται η χρήση σελίδων 2MiB κατά 30% κατά μέσο όρο.

Το Σημασιολογικό Κενό της Εικονικής Μνήμης στην Εικονικοποίηση

Το δεύτερο μέρος της διατριβής επικεντρώνεται στη γεφύρωση του σημασιολογικού κενού μεταξύ των υποσυστημάτων εικονικής μνήμης του φυσικού και του εικονικοποιημένου ΛΣ.



Σχήμα 6: Επίδοση των ET σε εικονικοποιημένη εκτέλεση χωρίς κατακερματισμό.



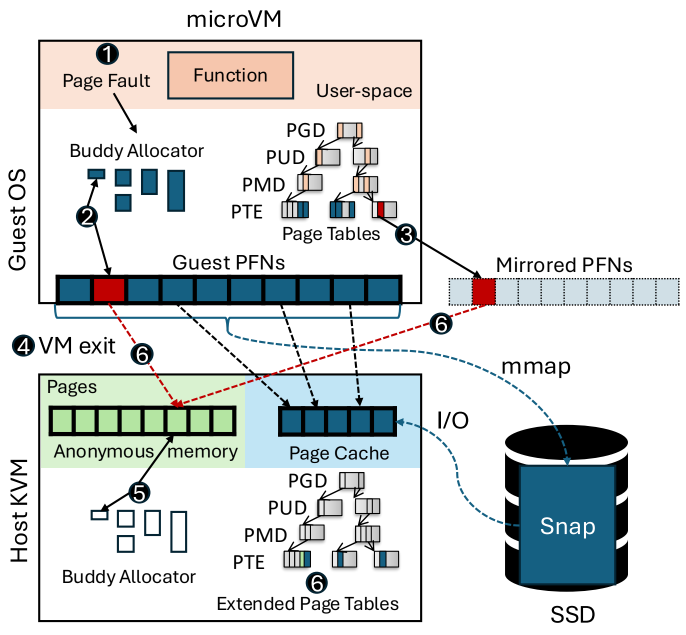
Σχήμα 7: Επίδοση των ET για μη εικονικοποιημένη εκτέλεση υπό κατακερματισμό μνήμης.

Το Πρόβλημα

Η εικονικοποιημένη εκτέλεση δημιουργεί ένα σημασιολογικό κενό μεταξύ του υποσυστήματος εικονικής μνήμης του μη εικονικοποιημένου και του εικονικοποιημένου ΛΣ. Αυτό το κενό γίνεται ιδιαίτερα προβληματικό όταν χρησιμοποιούνται στιγμιότυπα μνήμης εικονικών μηχανών για την επιτάχυνση του χρόνου εκκίνησης εφαρμογών που τρέχουν σε περιβάλλοντα χωρίς διακομιστή (serverless).

Όταν το εικονικοποιημένο ΛΣ εκχωρεί ανώνυμη μνήμη, αυτή αντιστοιχίζεται σε σελίδες του αρχείου στιγμιοτύπου στον δίσκο. Λόγω του σημασιολογικού κενού, το μη εικονικοποιημένο ΛΣ αγνοεί την ανώνυμη φύση αυτών των εκχωρήσεων και ανακτά άσκοπα σελίδες από το στιγμιότυπο στον δίσκο.

AnonPTEs: Σχεδιασμός



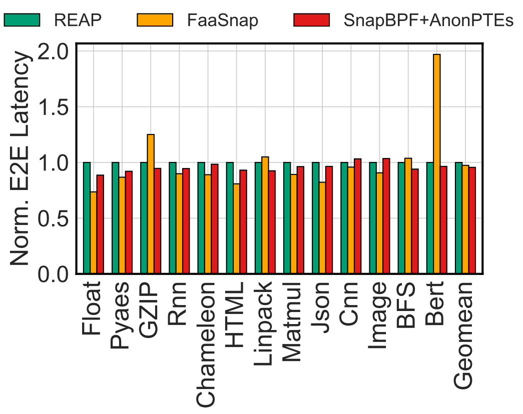
Σχήμα 8: AnonPTEs: Μια διεπαφή παραεικονικοποίησης για εκχωρήσεις μνήμης εικονικών μηχανών που εκκινούν από αρχεία στιγμιοτύπων.

Η διατριβή προτείνει τα *AnonPTEs*, έναν ελαφρύ μηχανισμό παραεικονικοποίησης, για να ενημερώνεται το μη εικονικοποιημένο ΛΣ σχετικά με τη φύση των εκχωρήσεων μνήμης της εικονικής μηχανής. Το Σχήμα 8 απεικονίζει τον μηχανισμό.

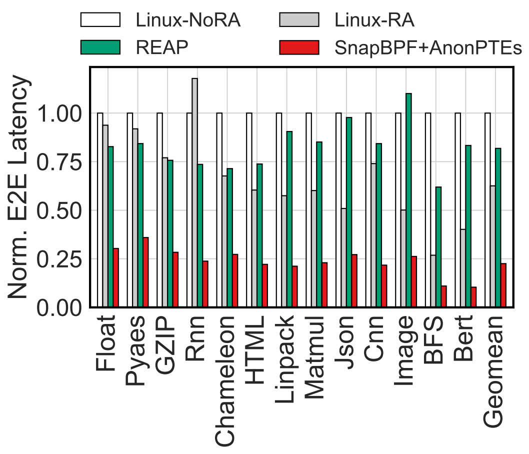
Τα AnonPTEs τροποποιούν τον διαχειριστή μνήμης του εικονικοποιημένου ΛΣ, ώστε όταν επιχειρεί να εκχωρήσει μνήμη, να την μαρκάρει, θέτοντας το πιο σημαντικό bit (MSB) του PFN στον πίνακα σελίδων, αντικατοπτρίζοντας τρόπον τινά την σελίδα αυτή σε ένα υψηλότερο χώρο φυσικών διευθύνσεων. Το μη εικονικοποιημένο ΛΣ (KVM), όταν χειρίζεται ένθετα σφάλματα σελίδας, ανιχνεύει σφάλματα για τέτοιους αντικατοπτρισμένες αντιστοιχίσεις, και χρησιμοποιεί ανώνυμη μνήμη αντί να ανακτήσει σελίδες από το στιγμιότυπο στον δίσκο.

AnonPTEs: Πειραματική Αξιολόγηση

Τα AnonPTEs υλοποιούνται στο Linux v6.3 και ενσωματώνονται με το SnapBPF, έναν prefetching μηχανισμό βασισμένο στην τεχνολογία eBPF. Η αξιολόγηση χρησιμοποιεί εφαρμογές από την σουίτα μετροπρογραμμάτων FunctionBench, καθώς και πραγματικές συναρτήσεις, που χρησιμοποιήθηκαν από πρόσφατη δουλειά (FaaSMem) πάνω στον υπολογισμό χωρίς διακομιστή. Το SnapBPF και τα AnonPTEs συγκρίνονται τόσο με την συμπεριφορά του vanilla Linux, όσο και με state-of-the-art προσεγγίσεις για την εκκίνηση συναρτήσεων από αρχεία στιγμιοτύπων, όπως το REAP και το FaaSnap.



(α′) Καθυστέρηση απόκρισης για μία συνάρτηση.



(β′) Καθυστέρηση απόκρισης για 10 ταυτόχρονες συναρτήσεις.

Σχήμα 9: Επίδοση του SnapBPF με AnonPTEs έναντι state-of-the-art.

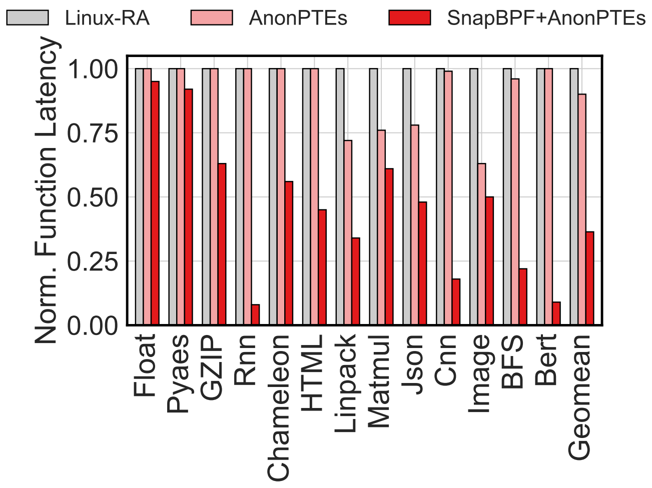
Το Σχήμα 9 παρουσιάζει τα αποτελέσματα για την καθυστέρηση απόκρισης και την συνολική μνήμη που χρησιμοποιούν οι συναρτήσεις όταν εκκινούν από στιγμιότυπα μνήμης. Το SnapBPF, μαζί με τον μηχανισμό των AnonPTEs, φτάνει και ξεπερνάει την απόδοση του FaaSnap, μια state-of-the-art προσέγγιση για εκκίνηση συναρτήσεων από στιγμιότυπα μνήμης, αποφεύγοντας έτσι την περιττή αντιγραφή του working set.

Όταν εκτελούνται 10 ταυτόχρονες instances της ίδιας συνάρτησης, το SnapBPF υπερτερεί σημαντικά του REAP επειδή επιτρέπει τον διαμοιρασμό των σελίδων του στιγμιοτύπου στην page cache του μη εικονικοποιημένου ΛΣ.

Το Σχήμα 10 παρουσιάζει την ανάλυση της επίδρασης των AnonPTEs. Συναρτήσεις με μεγάλο αποτύπωμα ανώνυμης μνήμης κατά την εκτέλεσή τους βλέπουν σημαντικές βελτιώσεις, καθώς τα AnonPTEs ανακατευθύνουν τα ένθετα σφάλματα σελίδας για τέτοιες εκχωρήσεις σε ανώνυμη μνήμη. Ενδεικτικά, η συνολική καθυστέρηση απόκρισης για τη συνάρτηση επεξεργασίας εικόνας (Image) βελτιώνεται περισσότερο από 2x.

Επίλογος

Καθοδηγούμενη από τις αναποτελεσματικότητες της εικονικής μνήμης που προκύπτουν από την αλληλεπίδραση του λογισμικού συστήματος με το υλικό, η διατριβή υποστηρίζει την ανάγκη ενίσχυσης του λογισμικού συστήματος με πολιτικές διαχείρισης μνήμης προσαρμοσμένες στο υλικό.



Σχήμα 10: Ανάλυση της επίδρασης των AnonPTEs στο SnapBPF.

Η διατριβή προτείνει τα Elastic Translations, μια επέκταση για τον διαχειριστή μνήμης του Linux που αξιοποιεί αποτελεσματικά τις δυνατότητες μετάφρασης πολλαπλών μεγεθών του σύγχρονου υλικού. Τα ET βελτιώνουν την επίδοση έως 39% για μη εικονικοποιημένη εκτέλεση και έως 150% για εικονικοποιημένη εκτέλεση.

Η διατριβή προτείνει επίσης τα AnonPTEs, έναν ελαφρύ μηχανισμό παραεικονικοποίησης που επικοινωνεί τη φύση των εκχωρήσεων μνήμης από το εικονικοποιημένο στο μη εικονικοποιημένο ΛΣ. Τα AnonPTEs βελτιώνουν σημαντικά την καθυστέρηση απόκρισης για serverless συναρτήσεις που εκκινούν από στιγμιότυπα εικονικών μηχανών.

Μελλοντικές Κατευθύνσεις

Τα αποτελέσματα επιβεβαιώνουν την ανάγκη για προσαρμογή των μηχανισμών διαχείρισης μνήμης του ΛΣ στα διαθέσιμα χαρακτηριστικά υλικού. Πιθανές μελλοντικές κατευθύνσεις περιλαμβάνουν την επέκταση του εύρους των υποστηριζόμενων μεγεθών μετάφρασης μέσω των ET (RISC-V Svnapot, HW-assisted TLB coalescing, σελίδες 1GiB), επέκταση της page cache του ΛΣ για να υποστηρίζει πολλαπλά μεγέθη μετάφρασης, καθώς και η επέκταση του διαχειριστή μνήμης μέσω της τεχνολογίας eBPF ώστε να μπορεί να γίνει προγραμματιστική επιλογή μεγεθών μετάφρασης από τον χώρο χρήστη.

Γλωσσάριο Τεχνικών Όρων

Το παρόν γλωσσάριο περιλαμβάνει τεχνικούς αγγλικούς όρους που χρησιμοποιούνται στην διατριβή, μαζί με τις αντίστοιχες μεταφράσεις τους στην Ελληνική γλώσσα. Ορισμένοι όροι, ιδιαίτερα ακρωνύμια και ονόματα τεχνολογιών (π.χ. eBPF, KVM), διατηρούνται στην αγγλική τους μορφή καθώς έχουν καθιερωθεί διεθνώς στην βιβλιογραφία.

Τεχνικοί Όροι

English Term	**Ελληνικά**
Translation Lookaside Buffer (TLB)	Προσωρινή Μνήμη Μετάφρασης Διευθύνσεων
TLB Coalescing	Συνένωση Καταχωρήσεων TLB
OS-assisted TLB Coalescing	Συνένωση Καταχωρήσεων TLB με Υποβοήθηση από το ΛΣ
HW-assisted TLB Coalescing	Συνένωση Καταχωρήσεων TLB με Υποβοήθηση από το Υλικό
Coalescing-aware	Έχοντας υπόψιν τον Μηχανισμό Συνένωσης
Contiguous (Contig) Bit	Σημαία Συνέχειας
Anchor Page	Σελίδα Αναφοράς
Multi-grained Address Translation	Μετάφραση Διευθύνσεων με Πολλαπλά Μεγέθη
Granularity Hints	Υποδείξεις Μεγέθους
Memory Bloat	Σπατάλη Μνήμης
Opportunistic Allocation	Ευκαιριακή Εκχώρηση (Μνήμης)
Working Set	Σύνολο Εργασίας
Profiling / Profiler	Σκιαγράφηση Επίδοσης
State-of-the-art (SOTA)	Τεχνολογία Αιχμής
State-of-practice	Τρέχουσα Πρακτική
Page Cache	Κρυφή Μνήμη Σελίδων
Guest (OS)	Φιλοξενούμενος / Εικονικοποιημένο ΛΣ
Host (OS)	Οικοδεσπότης / Μη Εικονικοποιημένο ΛΣ
Serverless Computing	Υπολογισμός χωρίς Διακομιστή
Prefetching	Προφόρτωση
Vanilla	Χωρίς Τροποποιήσεις
(Memory) Snapshot	Στιγμιότυπο (Μνήμης)

English Term	Ελληνικός Όρος
(Page) Deduplication	Αφαίρεση Κοινών Σελίδων

Συντομογραφίες & Ακρωνύμια

English Term	Ελληνικά
PMD (Page Middle Directory)	Μεσαίος Κατάλογος Σελίδων
PUD (Page Upper Directory)	Ανώτερος Κατάλογος Σελίδων
PGD (Page Global Directory)	Καθολικός Κατάλογος Σελίδων
PML4 (Page Map Level 4)	Χάρτης Σελίδων Επιπέδου 4
PCID (Process Context Identifier)	Αναγνωριστικό Πλαισίου Διεργασίας
ASID (Address Space Identifier)	Αναγνωριστικό Χώρου Διευθύνσεων
CR3 (Control Register 3)	Καταχωρητής Ελέγχου
TTBR (Translation Table Base Register)	Καταχωρητής Βάσης Πίνακα Μετάφρασης
Sv39/Sv48/Sv57 (RISC-V VA Modes)	Λειτουργίες Εικονικής Διεύθυνσης RISC-V
cPTE (Contiguous PTE)	Συνεχόμενο PTE
cPMD (Contiguous PMD)	Συνεχόμενο PMD
GVA (Guest Virtual Address)	Εικονική Διεύθυνση Φιλοξενούμενου
GPA (Guest Physical Address)	Φυσική Διεύθυνση Φιλοξενούμενου
HPA (Host Physical Address)	Φυσική Διεύθυνση Οικοδεσπότη
EPT (Extended Page Tables)	Επεκτάμενοι Πίνακες Σελίδων
NPT (Nested Page Tables)	Ένθετοι Πίνακες Σελίδων
AT (Address Translation)	Μετάφραση Διευθύνσεων
THP (Transparent Huge Pages)	Διαφανείς Τεράστιες Σελίδες
mTHP (Multi-sized THP)	THP Πολλαπλών Μεγεθών
KVM (Kernel Virtual Machine)	Εικονική Μηχανή Πυρήνα
FaaS (Function-as-a-Service)	Συνάρτηση ως Υπηρεσία
VM (Virtual Machine)	Εικονική Μηχανή
VMM (Virtual Machine Monitor)	Επόπτης Εικονικής Μηχανής
OS (Operating System)	Λειτουργικό Σύστημα (ΛΣ)
MMU (Memory Management Unit)	Μονάδα Διαχείρισης Μνήμης
TLB (Translation Lookaside Buffer)	Προσωρινή Μνήμη Μετάφρασης
ISA (Instruction Set Architecture)	Αρχιτεκτονική Συνόλου Εντολών
NUMA (Non-Uniform Memory Access)	Μη-Ομοιόμορφη Πρόσβαση Μνήμης
CoW (Copy-on-Write)	Αντιγραφή κατά την Εγγραφή

English Term	**Ελληνικά**
LRU (Least Recently Used)	Λιγότερο Πρόσφατα Χρησιμοποιημένη
SSD (Solid State Drive)	Δίσκος Στερεάς Κατάστασης
HDD (Hard Disk Drive)	Σκληρός Δίσκος
I/O (Input/Output)	Είσοδος/Έξοδος
SPE (Statistical Profiling Extension)	Επέκταση Στατιστικής Δειγματοληψίας
E2E (End-to-End)	Από την αρχή μέχρι το τέλος
WS (Working Set)	Σύνολο Εργασίας
PV (Paravirtualization)	Παρα-εικονικοποίηση

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

It is a truism in computer-system design that all good ideas become bad ideas, and nearly all bad ideas become good ideas: the nature of the field is such that all ideas must be reevaluated in the light of any shift in technology.

Bruce Jacob [1]

1.1 A Trip Down Virtual Memory Lane

1.1.1 The Virtual Memory Abstraction

The abstraction of virtual memory [2] has become one of the foundational building blocks of modern general-purpose computer systems. The need to abstract away and virtualize the system's physical memory emerged early in the evolution of computing. Random access volatile memory, commonly referred to as the main memory of a computer system, was and remains to this day an expensive and scarce resource. During the 1950s, as computer programs started to outgrow the available main memory of contemporary computer systems, programmers were forced to carefully orchestrate the loading and unloading of parts of the code and data of the programs, stored on secondary non-volatile storage, to and from main memory. Multi-programmed execution complicated physical memory management even further. Multiple computer programs now needed to coordinate in order to jointly manage physical memory.

Computer system designers solved the problem at hand by de-privileging the computer programs running on a system and shifting the burden of physical memory management to a privileged system component that eventually became known as the Operating System (OS) [3]. The OS relies on hardware (HW) support in order to give programs the illusion of being the single exclusive owners of the entire memory address space, by decoupling the (virtual) memory address space seen by programs from the physical memory address space of the system [4]. The burden of managing the mapping between the virtual address spaces and the actual physical memory of the system, as well as the placement of computer program memory, i.e., which parts of the program memory will reside on main memory and which on secondary storage, shifted to the Operating System (OS). This paradigm shift freed application programmers from having to implement their own memory management policies and allowed for the efficient centralized management of physical memory by the OS.

1.1.2 Segmentation and Paging

One of the key decisions system designers have to make, when designing and implementing virtual memory systems, concerns whether the mapping of virtual addresses to physical ones occurs in arbitrary or fixed granularity [2]. The first approach, known as *segmentation*, opts for arbitrarily, variably sized segments to map from the virtual to the physical address space [5]. This coarse mapping granularity reduces the number of total mappings required. Moreover, as segments can be arbitrarily sized to match the size of the mapping, they don't suffer from internal fragmentation. However, this size variability leads to external fragmentation. As segments are allocated and freed in physical memory, arbitrarily-sized holes of free memory are created in the physical address space. Subsequent segment allocation requests are not guaranteed to be able to find a large enough contiguous chunk of free physical memory, even though the total free physical memory of the system would be enough to satisfy the allocation request.

The other approach, known as *paging*, divides the virtual address space in fixed-sized *pages*, which are then mapped to equally fixed-sized *frames* of physical memory. Paging inverses the fragmentation trade-offs of segmentation. It eliminates the external fragmentation that plagues segmented virtual memory. Conversely, paged virtual memory suffers from internal fragmentation, due to the fixed size of the mapping. However, the internal fragmentation induced by paging can be controlled via the size of the mapping, i.e., the *page size*.

While segmented virtual memory was adopted by commercial systems in the early days of virtual memory [2], eventually paged virtual memory prevailed as the dominant virtual memory design. Consequently, this influenced the design of modern Operating Systems and their memory management subsystems, which almost exclusively focus on and are designed for paged virtual memory hardware. Nonetheless, in recent years researchers have revisited segments, integrated with traditional paged virtual memory, in an attempt to tame the address translation overheads of paging [6, 7].

1.2 Address Translation for Paged Virtual Memory

When designing paged virtual memory systems, system designers are faced with several key design decisions, the most impactful of which is, as mentioned briefly in the previous section, the granularity of the virtual to physical mappings (translations), i.e., the *page size*. However, the selection of a page size is intertwined with other design choices, that concern how to index and cache these mappings.

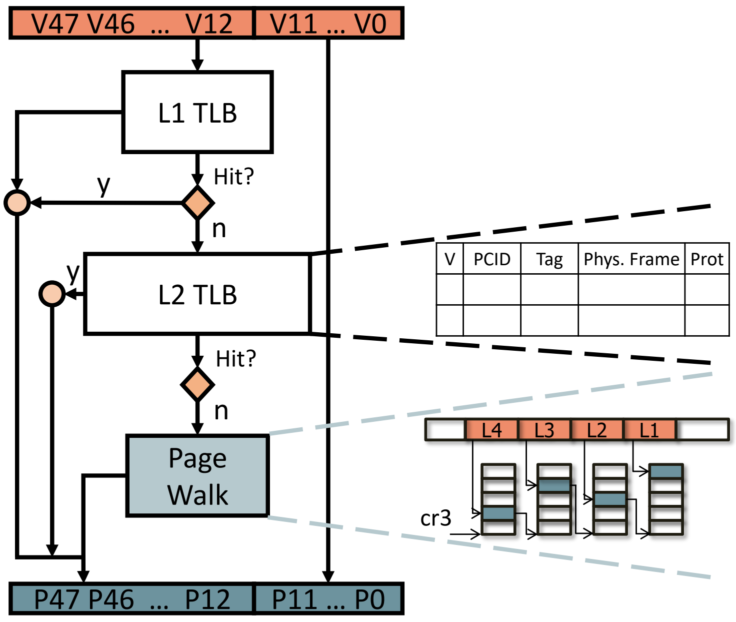


Figure 1.1: Caching virtual-to-physical mappings in the Translation Lookaside Buffer

Even from the earliest virtual memory systems, it was apparent that address translation, i.e., the process of mapping virtual addresses to physical ones, would pose a significant challenge performance wise. As discussed in the previous section, compared to segmentation, the increased number of mappings of paged virtual memory systems made address translation even more challenging. To that end, virtual memory systems employed from the get go specialized per-CPU caches, commonly called Translation Lookaside Buffers (TLBs), that cache the results of these translations, so that the address translation overhead is not imposed on each memory access (Figure 1.1).

The way that TLB misses and TLB refills are handled, i.e., what happens when, upon a memory access, a translation for a particular virtual address is not found in the TLB cache, has a direct impact in determining both the page size and the indexing scheme employed, and consequently on the overhead of address translation and virtual memory as a whole. During the evolution of paged virtual memory systems, two approaches have emerged and been adopted by different CPU architectures, each with its own trade-offs.

SW-managed TLBs, were commonly adopted by RISC (Reduced Instruction Set Computer) CPUs, such as MIPS [8, 9], which favored hardware simplicity, opting for offloading the decisions regarding the indexing and the granularity of the virtual to physical mappings to the software. As such, architectures supporting SW-managed TLBs don't typically include in the specification of the ISA (Instruction Set Architecture) an indexing scheme for virtual-to-physical translations and only provide in the hardware a cache (TLB) for these translations. Importantly, TLB misses and refills are handled in software. Whenever a translation misses in the TLB, the CPU will trap into the OS, which will then load the translation into the TLB. This provides the OS with a lot of flexibility in deciding how to organize and index the virtual-to-physical translations. However, this flexibility comes at a cost. Handling TLB refills by trapping to the OS incurs a significant performance overhead, which quickly outweighed the benefits of flexibility [10].

The second approach, *HW-managed TLBs*, lets the hardware, i.e., the Memory Management Unit (MMU), handle TLB refills. This has the implication that the MMU must know how to retrieve the missing translation from memory. This further implies that the architecture has to specify an exact indexing scheme for the virtual-to-physical translations, commonly called the *page table*. The MMU then implements *hardware page table walkers*, that are able to traverse the page table and retrieve the missing translation. HW-managed TLBs have become the dominant approach for modern, high-performance, general-purpose CPUs, being supported by architectures like x86, ARMv8 and RISC-V [8, 11, 12].

1.2.1 Page Table Structure

As discussed above, depending on the architecture, the indexing scheme for virtual-to-physical translations, i.e., the *page table* structure, might be mandated by the ISA, in the case of HW TLB refillers, or the OS might have the freedom to implement any structure it sees fit to store and index these translations, when TLB refills are offloaded to software.

The vast majority of CPUs and OSes implement one of two indexing structures, namely a sparse, forward-mapped, multi-level radix tree, or a hash-indexed, typically inverted, page table [8, 9], each with different trade-offs [13, 14]. While both structures have been adopted by several successful systems in the past and have also been extensively studied by academia [15–18], radix trees have emerged as the prevailing page table structure, employed by the x86, ARMv8 and RISC-V architectures [11, 12] (Figure 1.2). The POWER ISA [19] is the notable exception, still supporting hashed page tables.

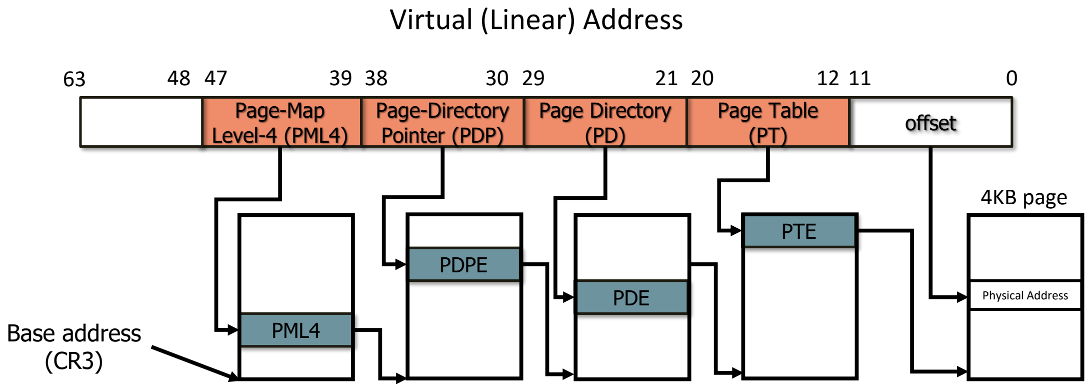


Figure 1.2: x86 Radix Page Table

The whole point really is that the page table tree as far as Linux is concerned is nothing but an _abstraction_ of the VM mapping hardware. It so happens that a tree format is the only sane format to keep full VM information that works well with real loads.

Linus Torvalds [20]

1.2.2 Multi-grained Address Translation

Regardless of the page table structure and the handling of TLB refills, there is a common, crucial parameter that system designers need to define when designing paged virtual memory systems, the granularity of the virtual-to-physical mappings (*translations*). Selecting an optimal page size has been studied since the early days of paged virtual memory [2, 21]. Increasing the page size reduces the number of translations, alleviating the overhead of address translation. On the other hand, larger page sizes face increased internal fragmentation [22]. Moreover, as virtual memory is intertwined with the file system layer of the OS, there is a natural pressure to align the page size with the optimal granularity of transfer of block devices [23]. Early paged virtual memory systems favored smaller page sizes, typically 512 bytes. As the virtual and physical address spaces grew over time, with the transitions to 32-bit and then 64-bit address spaces, and the ever-increasing memory capacity of computer systems, the optimal page size followed, eventually settling on 4KiB, which is the page size most commonly used by modern computer systems, and has essentially become the de facto page size.

However, computer architects soon realized that supporting only one translation granule, i.e., page size, implied certain trade-offs that might not suit the needs of all applications and systems [22, 24]. They hence started to embrace flexibility in the translation granularity, i.e., multi-grained address translation. The most direct way to support multi-grained address translation is by extending the architecture and the OS to support multiple page sizes. As it will become apparent later, this has far-reaching implications both for the micro-architecture and the OS memory manager [25].

While there were proposals that worked entirely in hardware and required limited or no modifications to the architecture specification, i.e., the page table structure [26–28], all approaches on multi-grained address translation relied on some form of cooperation between the OS, the architecture and the micro-architecture [22, 25, 26, 29]. An over-arching requirement on the micro-architecture side, is that the TLB must be modified to be able to cache variably-sized translations [24, 26–28, 30–32].

Earlier approaches. The first attempts to retrofit multi-grained address translation to paged virtual memory systems concerned older generation, typically RISC, architectures and CPUs, which commonly employed non-radix page table structures and SW-refilled TLBs [8–12]. The translation granularity for each PTE was tracked and managed by the OS, usually via a hashed inverted page table. Upon a miss, the OS TLB miss handler would load an appropriately-sized translation into the TLB. On the OS side, a lot of research of that era focused on proprietary UNIX OSes, running on those systems, adapting them so that they could efficiently handle multiple page sizes [8, 9, 33–36]. Three main common issues were raised by these works, concerning i) how should the OS manage and allocate physical memory when dealing with multiple page sizes, ii) who and how should decide about the optimal page size to use, and iii) whether large pages should be created synchronously at fault time and / or asynchronously via page migrations [37, 38].

Radix Trees with HW-refilled TLBs. As these earlier generation architectures were gradually superseded by the x86 architecture, the OSes similarly adapted to the HW-refilled radix tree page tables for the x86 ISA. The x86 radix page table tree supports multiple page sizes by storing larger virtual-to-physical translation higher in the radix tree, effectively short-circuiting the page walk process, and supporting 4KiB base pages at the lowest level, and 2MiB and 1GiB *large pages* in the higher levels of the tree. While the x86 ISA stops at 1GiB large pages, they could theoretically extend to even higher levels of the radix tree (as is specified by RISC-V [39]). x86, ARMv8-A and RISC-V all support large pages via this mechanism.

OS-assisted TLB coalescing. The DEC Alpha architecture [40] supported multiple page sizes via *granularity hints*, embedded within the page table entry (PTE). This design, also referred to as PTE replication [13], was partially adopted by the ARM architecture, in the form of the contig bit [41], working orthogonally to its x86-inspired HW-refilled multi-level radix page tables (Figure 1.3). This was later also adopted by RISC-V in the form of the NAPOT extension [32, 39]. Researchers working on HW-refilled radix trees, i.e., on the x86 architecture, also independently proposed similar, albeit more flexible approaches [42, 43]. As these architectures support large pages via their radix tree levels, this form of multi-grained address translation was known as OS-assisted TLB coalescing. Instead of relying on HW to detect and coalesce virtually and physically

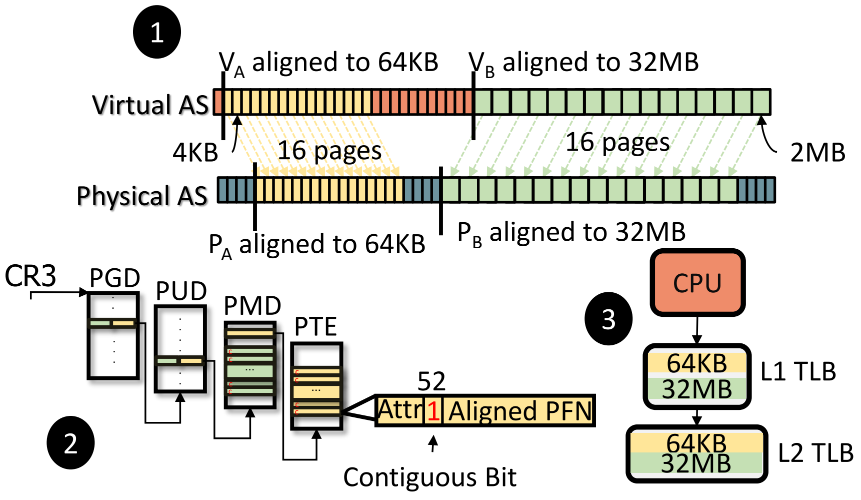


Figure 1.3: ARMv8-A OS-assisted TLB coalescing

Architecture	**Translation Granules** (PTE - cPTE - PMD - cPMD - PUD)
x86	4KiB - 2MiB - 1GiB
ARMv8-A	4KiB - 64KiB - 2MiB - 32MiB - 1GiB 16KiB - 2MiB - 32MiB - 1GiB 64KiB - 2MiB - 512MiB - 16GiB
RISC-V	4KiB - 8KiB .. 1MiB - 2MiB, 4MiB .. 512MiB - 1GiB

Table 1.1: Translation Granules for modern CPU Architectures.

contiguous and properly aligned groups of pages [27, 28], the burden is shifted to the OS, which signals to the TLB, via the page tables, that certain groups of pages are contiguous and can thus be coalesced into a single translation entry.

Table 1.1 summarizes the supported translation granules for x86 [44], ARMv8-A [41] and RISC-V [39]. *PTE*, *PMD* and *PUD* granules are supported via the ”x86” large page mechanism, storing each translation in the respective level of the radix page table tree. Note that the table omits the larger page sizes (512GiB and 256TiB) supported by RISC-V when using 48-bit (Sv47) or 57-bit (Sv57) virtual address spaces. *cPTE* and *cPMD* granules are supported via architectural OS-assisted TLB coalescing mechanisms, the *contig bit* on ARMv8-A and NAPOT on RISC-V. Finally, also note that ARMv8-A allows for three different sets of translation granules, as the architecture supports three different base granules (4KiB, 16KiB and 64KiB).

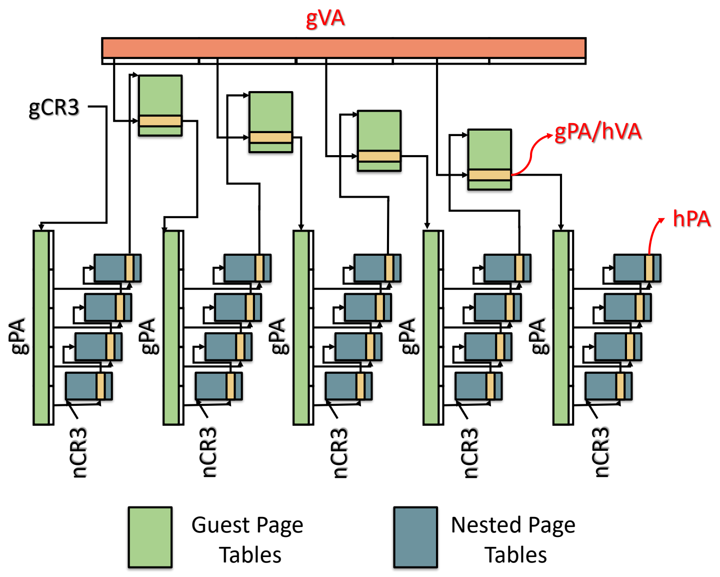


Figure 1.4: Nested Page Tables

1.2.3 Nested Paging

Virtualized execution presents additional challenges for paged virtual memory systems [45]. Full-system virtualization, introduced by the IBM mainframes in the 1970s [46, 47], saw a resurgence in the 1990s [48], targeting cache-coherent Non-Uniform Memory Access (ccNUMA) multiprocessor systems [49, 50], and finally came to dominate the computing landscape, as the foundation of x86-based cloud computing, in the 2010s [51, 52].

As x86 was not inherently able to support virtualization efficiently [46], early approaches opted for software techniques, i.e., paravirtualization [52] and dynamic binary rewriting [53, 54]. Eventually, x86 hardware vendors, Intel and AMD, added architectural extension to their processors, that enabled the efficient virtualization of x86 systems [44, 55]. This approach was later also adopted by the ARMv8-A [41] and RISC-V [39] architectures.

For virtual memory, the software approach, commonly called shadow paging [7, 45, 56] requires the OS to maintain shadow page tables. The hypervisor write-protects the memory where the page table of the virtual machine (VM) resides, so that attempts by the VM to modify its page tables will trap to the hypervisor. The hypervisor then replicates the intended changes to the shadow page tables, which are the ones that are actually used by the virtual memory hardware. Shadow paging introduces considerable complexity to system software and for systems and workloads with a high frequency of page table modifications, it also incurs high overhead due to trapping.

To that end, architectural hardware extensions support a form of virtualized paging, commonly referred to as nested paging (Figure 1.4). The virtual machine (guest) maintains its own page table, which it is able to modify without trapping to the hypervisor. The MMU is modified so that the guest physical memory is then mapped to hypervisor (host) memory via the hypervisor (host) page tables, hence the name, nested paging [45].

While nested paging obviates the trapping overhead for page table modifications, and greatly simplifies system software handling, it adds an extra dimension to page table walks, i.e., a single virtualized translation required walking two sets of page tables, greatly increasing the overhead of page walks, hence TLB misses.

1.3 Problem Statement

1.3.1 Multi-grained Address Translation in modern Operating Systems

As discussed earlier, the first attempts to support multi-grained address translation in Operating Systems was done mostly by commercial UNIXes, such as HP-UX [34, 57], Solaris [33], Irix [35], and AIX [58], which typically ran on RISC CPUs, such as PA-RISC, SPARC, DEC Alpha and POWER, commonly employing SW-refilled TLBs and non-radix page tables. As x86 emerged as the prevailing architecture for general-purpose, high performance CPUs, both on the desktop and for the datacenter, the focus shifted to OSes that ran on the x86 ISA, e.g., Linux and FreeBSD. Consequently, the HW-refilled radix page table structure of the x86 architecture, and its simplified, relative to earlier CPUs, large page model, became the de facto standard, which modern Operating Systems build upon [11, 12].

Both Linux [61, 64] and FreeBSD [65] transparently support 2MiB large pages, while support for 1GiB pages is restricted behind explicit, non-transparent interfaces [59, 66]. As the industry focus shifted to the x86 large model, so did the focus of academia. Most works in the x86 era hence focus on optimizing Linux [62, 67–70] and FreeBSD [71] 2MiB transparent large pages. Linux has been recently extended to support multi-sized transparent large pages, smaller than 2MiB, via the multi-sized THP (*mTHP*) mechanism [60].

Table 1.2 provides a summary of the state-of-practice and state-of-the-art approaches for multi-grained address translation on Linux and FreeBSD. Transparent support for intermediate-sized translations (i.e., *64KiB* and *32MiB*) is limited, both for native and virtualized execution. Most designs focus on the x86 4KiB / 2MiB / 1GiB model, with most fault-time and promotion policies assuming a single available large page size (i.e., 2MiB) [29, 61, 62, 72]. When multiple sizes are considered [60, 63] simplistic fallback policies are used.

	Transparent	Faults		Translation			Promotions	
		Supported Sizes	Policy	Supported Sizes	Policy	Virtualization Support	Supported Sizes	Policy
HugeTLB [59]	✗	4KiB, **64KiB** 2MiB, **32MiB** 1GiB	Pre-allocation Single user-defined size per VMA	4KiB, **64KiB** 2MiB, **32MiB** 1GiB	Defined by fault size	4KiB 2MiB 1GiB	✗	✗
mTHP [60, 61]	✓	4KiB, **64KiB** 2MiB	Eager allocation of the largest possible size Fallback on failure	4KiB, **64KiB** 2MiB	Defined by fault or promotion size	4KiB 2MiB 1GiB	2MiB	Migrate to 2MiB Region selection: Linear scan
FreeBSD [29]	✓	4KiB	2MiB reservation at first 4KiB fault Use reservation to serve the rest	4KiB 2MiB	Defined by fault or promotion size	4KiB 2MiB	2MiB	In-place promotion to 2MiB when every 4KiB page is faulted-in
HawkEye [62]	✓	4KiB 2MiB	Same as mTHP Asynchronous pre-zeroing	4KiB 2MiB	Defined by fault or promotion size	4KiB 2MiB	2MiB	Selectively migrate to 2MiB Region selection: Access frequency based on page table scanning
Trident [63]	✓	4KiB 2MiB 1GiB	Same as HawkEye	4KiB 2MiB 1GiB	Defined by fault or promotion size	4KiB 2MiB 1GiB	2MiB 1GiB	Migrate to largest size possible Fallback on failure Selection same as mTHP
Elastic Translations	✓	4KiB 2MiB	4KiB / 2MiB eager allocation based on VMA size **Opportunistic coalescing-aware allocations across faults**	4KiB, **64KiB** 2MiB, **32MiB**	4KiB or 2MiB based on fault or promotion size **Opportunistic promotion to 64KiB or 32MiB**	4KiB, **64KiB** 2MiB, **32MiB**	**64KiB** 2MiB, **32MiB**	Selectively migrate to 64KiB, 2MiB, 32MiB Region Selection: Size hints based on HW TLB miss sampling

Table 1.2: State-of-practice and state-of-the-art OS interfaces and mechanisms for multi-grained address translation.

1.3.2 The Virtual Memory semantic gap in Virtualized Execution

Virtualized execution has become in the last couple of decades almost as ubiquitous as virtual memory, serving as the cornerstone of cloud computing, driven by both the need of compute infrastructure providers to efficiently manage and consolidate user workloads and the desire of users for resource elasticity and demand pricing.

Besides the address translation overhead incurred by nested paging during virtualized execution, virtualization also creates a semantic gap between the virtual memory subsystems of the host (physical) and guest (virtualized) OS instances [73–75]. Each virtual memory subsystem acts mostly independently, unaware of the decisions taken by the other. This leads to inefficiencies in terms of performance and memory waste.

This semantic gap becomes especially apparent and problematic, when using virtual machine (VM) memory snapshots to accelerate the start-up time of VM-sandboxed serverless functions. Serverless computing pushes the resource elasticity and pay-as-you-go model of cloud computing to the limits. Users encapsulate their business logic as functions, which the serverless providers then deploy and scale independently based on incoming load. These serverless function are typically short-lived, with execution runtimes in the second and millisecond regime [76, 77]. To sandbox and isolate serverless functions between tenants, serverless providers can choose between OS-level virtualization mechanisms, i.e., containers, or virtual machines. The providers typically favor the stronger isolation guarantees of virtualization, hence forgoing the more lightweight OS-level virtualization. This however comes in contrast with the short-lived nature of serverless functions. To that end, providers and academia have worked towards lightening the overhead of virtualization sandboxing, first by creating specialized lightweight virtual machine monitors (VMMs), which sandbox user functions inside minimal *microVMs* [78], and then by utilizing checkpoint-and-restore mechanisms [79], whereby the virtual machine memory is checkpointed to a snapshot file on disk. When a function needs to execute again, instead of paying the overhead of virtual machine creation and booting, the VMM is able to restore the VM and function state from the memory snapshot, hence accelerating start-up time (*cold start*). Researchers [80–82] have pushed this even further, by capturing per-function working sets from the snapshot file and prefetching said working set in advance, thus avoiding the nested page faults that would otherwise happen, as the function would attempt to access its working set pages from the snapshot file.

While this alleviates the faulting overhead, one issue remains. When the guest (virtualized) OS allocates anonymous memory, to be used by the function, this memory is mapped to page on the snapshot file on disk. Due to the virtualization-induced semantic gap between the host and the guest virtual memory subsystems, the host OS (VMM) is unaware about the anonymous, ephemeral nature of these allocations. When the function inside the VM will attempt to access this memory, the host OS will unnecessarily fetch the corresponding pages from the snapshot file on disk, as these pages are anonymous memory inside the VM, they don't contain any valid state.

Prior Art Limitations. Previous approaches to this problem, such as Faast [82], and FaaSnap [81], tackle the issue by resorting to scanning and pre-processing the snapshot file. FaaSnap patches the VM kernel to zero pages when they are freed, then scans the snapshot file for zero pages and maps those zero regions to anonymous memory. Faast relies on the allocator metadata of the VM kernel to identify pages that are not actively used in the snapshot and routes faults for these pages to anonymous memory. Regardless of the mechanism employed, both approaches rely on preemptive snapshot scanning and pre-processing to optimize the handling of VM memory allocations, adding complexity and overhead.

1.4 Thesis Contribution

1.4.1 Elastic Translations

The first part of this thesis focuses on the address translation overhead of paged virtual memory, for both native and virtualized execution. It specifically targets the impact of translation granularity on virtual memory performance, as well as on the interplay between translation granularity and OS memory management.

As discussed in Section 1.3.1, the thesis initially surveys the available HW mechanisms that

Component	Purpose	Main Contribution
Transparent Contig-Bit (kernelspace)	Transparent, opportunistic creation of 64KiB and 32MiB translations via OS-assisted TLB coalescing	Native support for 32MiB translations Virtualization support for all sizes
CoalaPaging (kernelspace)	Transparent, opportunistic creation of contiguous 64KiB and 32MiB mappings across page faults	Practical and scalable allocation policy for 32MiB mappings, with minimal impact on fault latency and memory bloat
CoalaKhugepaged (kernelspace)	Asynchronous creation of 64KiB and 32MiB mappings via migrations	Enables the creation of 64KiB and 32MiB translations under memory pressure (fragmentation)
Leshy Profiler (userspace)	Runtime translation size selection guidance via MMU overhead profiling	Optimal translation size selection from an extended range of sizes via lightweight HW-assisted TLB miss sampling

Table 1.3: The kernelspace and userspace components of Elastic Translations.

enable multi-grained address translation, namely large pages and TLB coalescing, and the existing interfaces that modern OS, such as Linux and FreeBSD provide to support these mechanisms in order to enable the OS to utilize multiple translation granules. The survey uncovers inherent limitations in these OS interfaces that hamper the effectiveness of the underlying HW mechanisms and the ability of the OS memory manager to efficiently and effectively utilize multiple translation granules.

The thesis then proposes *Elastic Translations (ET)*, a novel approach that adopts a markedly different approach regarding the way the OS interfaces with the aforementioned HW mechanisms and enables the OS to seamlessly and optimally use and select among all HW-supported translation granules.

ET contribute four tightly integrated mechanisms, summarized in Table 1.3.

Transparent contiguous-bit management. ET extend the Linux memory manager and KVM to transparently and opportunistically coalesce contiguous memory mappings into larger translations via the OS-assisted TLB coalescing feature of ARMv8-A (contig-bit), for both 64KiB and 32MiB translations, for both native and virtualized execution. Whenever a page table entry is created or modified, ET checks whether the neighboring entries in the coalescing range are suitably aligned, physically contiguous, and have compatible page flags and access permissions. If all conditions are met, ET promote the range to an intermediate-sized translation by setting the contiguous bit in each page table entry. Conversely, when any page entry modification invalidates these conditions, ET demote the range by clearing the contiguous bit and performing the necessary TLB invalidations.

CoalaPaging: Coalescing-aware demand paging. ET augment demand paging with a lightweight, coalescing-aware allocation policy, CoalaPaging. CoalaPaging extends the Linux page fault handler to opportunistically allocate suitable 4KiB and 2MiB pages across consecutive faults in order to lazily generate intermediate-sized contiguity that matches the coalescing sizes supported by the hardware. Rather than eagerly allocating an entire 32MiB region at once, which would increase fault latency and memory bloat, CoalaPaging tracks partial contiguous allocations across faults and completes them opportunistically. This approach provides a practical and scalable allocation policy for 32MiB mappings with minimal impact on fault latency, while avoiding the memory waste associated with eager large-page allocations. CoalaPaging does not reserve memory or require explicit metadata tracking.

CoalaKhugepaged: Asynchronous coalescing-aware promotions. ET generalize Linux's khugepaged to migrate and coalesce partially contiguous regions into fully coalesced 64KiB and 32MiB mappings. CoalaKhugepaged leverages partial contiguity created by CoalaPaging to significantly reduce migration volume, enabling large translation formation even on fragmented systems. CoalaPaging and CoalaKhugepaged work synergistically: fault-time mechanisms establish partial contiguity opportunistically, while background promotion completes the coalescing asynchronously.

Leshy: HW-assisted MMU overhead profiling and translation size selection. ET introduce a profiling-driven translation-size selection policy, guided by lightweight hardware-assisted TLB-miss sampling. Leshy is a userspace profiler that implements the ET policies for translation size selection at runtime, leveraging the ARMv8-A Statistical Profiling Extension (SPE) to sample the TLB misses of workloads and track the virtual address and page walk latency for each sampled miss. From this data, Leshy generates a translation-overhead heatmap of the address space, identifying MMU hotspots, i.e., narrow regions responsible for the majority of TLB misses. Unlike prior approaches that rely on page-based access frequency sampling, HW-assisted TLB miss sampling enables Leshy to identify MMU hotspots at a higher resolution, as not every frequently accessed page contributes equally to translation overhead. Leshy then maps address space regions to translation sizes with the goal of minimizing translation costs, and drives the ET in-kernel mechanisms by loading the generated translation-size profiles into the kernel.

Contiguity Redistribution via Demotions. ET also enable the redistribution of reserved contiguity at runtime via asynchronous demotions. Due to the transient nature of the execution profile of many applications, certain address space regions might go cold, wasting physical memory contiguity for processes and regions that no longer benefit from it. Leshy is extended to dynamically switch between TLB miss sampling and memory access sampling based on the MMU intensity of the profiled application at runtime. When TLB misses drop below a threshold, Leshy switches to memory access sampling to detect cold address space regions mapped by larger translation granules. For these cold regions, it issues demotion hints to a complementary in-kernel ET component, which asynchronously migrates the demoted regions to non-contiguous parts of the physical address space, relinquishing the unused contiguity back to the system for use by other applications experiencing MMU pressure.

Implementing ET in Linux for the ARMv8-A architecture and evaluating its performance on a real ARMv8-A server shows that ET unlock the full potential of the OS-assisted TLB coalescing feature of the ARMv8-A architecture and enable the OS memory manager to optimally utilize the extended range of translation granules, enabled by coalescing, outperforming both state-of-practice and state-of-the-art approaches. Transparent 64KiB translations perform closely to 2MiB pages for memory-intensive workloads with small footprints, for both native and virtualized execution. For larger workloads, transparent 32MiB translations improve performance by 10% on average and up to 39% over THP for native execution, and by 30% on average and up to 150% for virtualized execution. Finally, Leshy's microarchitectural-aware policies guide ET to map the footprint of workloads by utilizing an optimal mix of all available translation sizes, minimizing address translation overhead. This improves overall performance under fragmentation by 12% on average and up to 20% over THP and state-of-the-art approaches, while consistently reducing the number of 2MiB pages used. Collectively, ET provide the first fully transparent, coalescing-aware, and profiler-guided support for multi-granular translations in Linux for both native and virtualized execution.

1.4.2 AnonPTEs

The second part of the thesis focuses on bridging the semantic gap between the virtual memory subsystems of physical and virtualized OS instances via the virtual memory hardware, i.e., the Memory Management Unit (MMU).

As discussed previously in Section 1.3.2, this semantic gap becomes particularly apparent and problematic when using Virtual Machine (VM) memory snapshots to accelerate VM boot times, a technique heavily employed in latency-sensitive scenarios such as Function-as-a-Service (FaaS) serverless computing. As VM snapshots are stored as files on disk, the OS on the physical host treats the entire VM memory as a file, being oblivious to the ephemeral anonymous allocations performed by the VM memory subsystem.

The thesis proposes a lightweight paravirtualized (PV) mechanism, *AnonPTEs*, to enlighten the host OS regarding the nature of the VM memory allocations by taking advantage of the virtual memory hardware. *AnonPTEs* piggyback on the VM AT structures, i.e., the VM page tables, to create a mirrored guest physical address space for the ephemeral anonymous allocations of the VM, enabling the host OS to detect such allocations and serve them using anonymous memory instead of unnecessarily fetching memory pages from the VM memory snapshot file.

An overview of the AnonPTEs mechanism is depicted in Figure 1.5. The VM (guest) kernel is modified so that when it attempts to allocate memory, it marks it in a way that it can be detected by the VMM on the host. When the guest kernel maps freshly-allocated anonymous memory in its page tables, instead of using the actual guest page frame number (gPFN), it sets the most

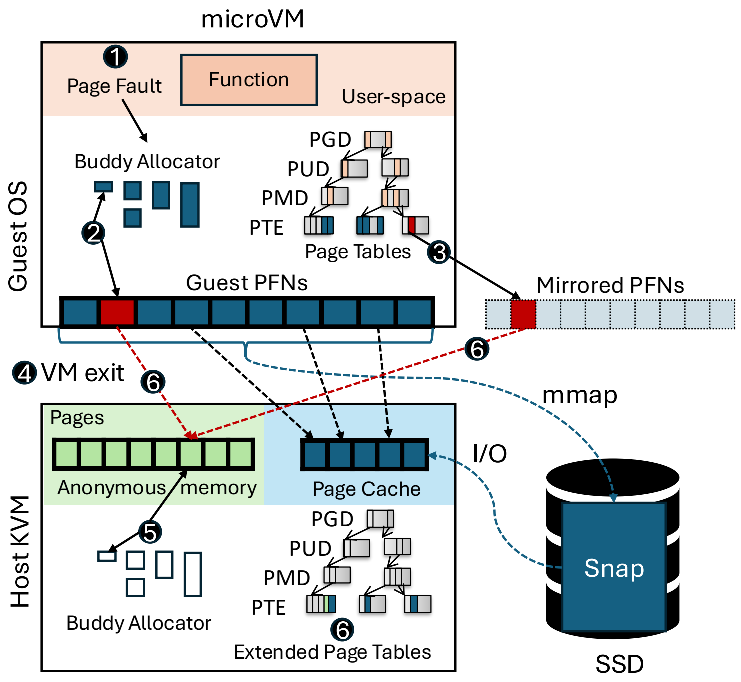


Figure 1.5: AnonPTEs: A lightweight PV interface for VM memory allocations to avoid unnecessary IO.

significant bit (MSB) of the PFN, effectively mirror-mapping this page to a higher PFN space. The host kernel, specifically the Linux Kernel Virtual Machine (KVM), when handling nested page faults for the VM, detects faults for such mirrored PFNs. In that case, it uses anonymous memory to serve the page fault instead of fetching pages from the on-disk snapshot. It then maps this anonymous page to both the mirrored and the original gPFN in the VM's nested page tables, so that when the VM subsequently reuses this memory, it also points to the anonymous page allocated by the host.

AnonPTEs enable the host OS to handle VM memory allocations without redundant I/O from the snapshot file and without requiring any preemptive snapshot scanning or pre-processing. Moreover, the design is hypervisor and OS agnostic, although the thesis implements and evaluates it for Linux and KVM using the Firecracker microVM monitor.

AnonPTEs are implemented in Linux v6.3, and are integrated with the firecracker microVM manager v1.11. The thesis evaluates AnonPTEs for the FaaS use-case, by extending SnapBPF [83], a state-of-the-art eBPF-based serverless function snapshot prefetcher and using representative serverless workloads from the FunctionBench suite [84] and real-world functions [85]. The evaluation results show that AnonPTEs match and improve upon state-of-the-art serverless snapshot prefetching approaches. Functions that allocate large amounts of memory during invocation see significant improvements from the PV PTE marking mechanism, as it redirects nested page faults for such allocations to anonymous memory and eliminates unnecessary fetching of pages from the snapshot file. By eliminating major page faults for anonymous VM allocations, AnonPTEs significantly reduce tail latency for VM-sandboxed serverless functions that employ snapshotting.

1.5 Thesis Organization

Chapter 2 provides an overview of Linux memory management and the eBPF framework, that permeate both proposals.

Chapter 3 provides an in-depth description and evaluation of *Elastic Translations*, which enable the OS to efficiently handle multi-grained address translation, both at fault time and asynchronously. Chapter 3 is based on the paper published in the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO 2024) [86], which was first presented as a poster at the 18th ACM SIGOPS European Conference on Computer Systems (EuroSys'23) [87]. It also presents an extension to the published work that pushes translation elasticity even further by designing a demotion mechanism, allowing the OS to downgrade translations to reclaim contiguity, based on userspace hints, during memory pressure. Finally, it includes a discussion, based on work presented at the ACM Student Research Competition (ACM SRC) at the 57th IEEE/ACM International Symposium on Microarchitecture [88], on the design of OS interfaces to enable userspace to drive OS memory management decisions and a vision for future work, which intends to make contiguity a first-class OS metric.

Chapter 4 presents AnonPTEs, the paravirtualized mechanism that bridges the semantic gap of the virtual memory subsystems of host and guest OS, and its integration in SnapBPF, a state-of-the-art framework, based on eBPF [89], that accelerates snapshot start times for microVM-sandboxed serverless functions via working set prefetching. Chapter 4 is based on the work published at the 17th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage 2025) [83], which earned the best paper award. The first part of the paper, regarding eBPF prefetching, was the research artifact of a diploma thesis conducted earlier at the Computing Systems lab [90] and is not a direct result of this dissertation.

Chapter 5 concludes the thesis and summarizes future directions for hardware-tailored memory management, and Chapter 6 provides a list of publications the author contributed to during their thesis.

CHAPTER 2

Background

This chapter provides a brief background on OS memory management concepts, that permeate both Elastic Translations and AnonPTEs. It specifically focuses on the memory manager of the Linux kernel [91, 92]; how it manages physical memory (Section 2.1), how it handles memory allocations for the filesystem page cache (Section 2.2) as well as for anonymous memory (Section 2.3), and a description of how paged virtual memory works in virtualized execution (Section 2.4). It concludes with a primer on the eBPF technology that is referenced both in the context of Elastic Translations, when discussing OS interfaces for translation elasticity, and AnonPTEs, specifically its integration in SnapBPF, a state-of-the-art prefetcher for serverless functions that employs eBPF.

2.1 The Buddy Allocator

Linux uses a buddy allocator [93–95] to manage the physical memory of the system. The buddy allocator (Figure 2.1) organizes physical memory in contiguous and aligned groups of power-of-two (2^n) 4KiB pages. n, the binary logarithm of the group size in 4KiB pages, is called in Linux term the *order* of the group. A single 4KiB page has an order of 0, a 2MiB group of contiguous and aligned pages has an order if 9, as it consists of 512 (2^9) consecutive 4KiB base pages etc.

For each order, the buddy allocator maintains a linked list of contiguous and aligned pages of that order. The maximum order for which the allocator maintains such a list is called in Linux terms the MAX_ORDER. It defaults to 10, i.e., 4MiB, but it can be and indeed is configured to

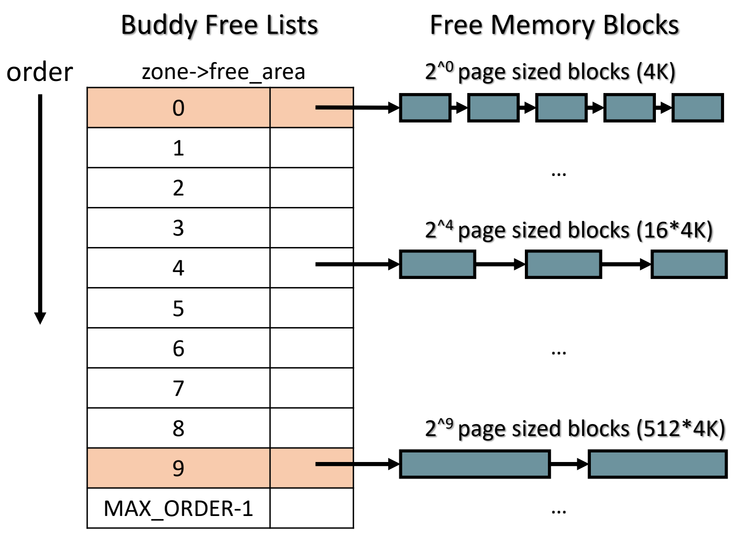

Buddy Free Lists
Free Memory Blocks
order
zone->free_area
0
1
2
3
4
5
6
7
8
9
MAX_ORDER-1
2^0 page sized blocks (4K)
...
2^4 page sized blocks (16*4K)
...
2^9 page sized blocks (512*4K)
...

Figure 2.1: The Linux buddy physical memory allocator.

different larger values for specific kernel configurations. Collectively, these lists are referred to as the buddy allocator *free lists*.

The kernel separates the available physical memory of the system into different *nodes*, based on the system's NUMA (Non-Uniform Memory Access) configuration [96], and further separates each node in memory *zones*, based on specific characteristics of certain parts of the physical address space. The bulk of the available physical memory of each NUMA node typically comprises a single zone, called ZONE_NORMAL. The kernel maintains separate free lists, i.e., separate buddy allocator instances, for each memory zone in the system.

A simplified description of the allocation path of the buddy allocator follows. To serve an allocation request for a specific order, the buddy allocator will first search the free list for the requested order for free blocks of physical memory. In case there are no available free physical blocks of the requested order, the allocator will move up to higher order blocks. When it finds a higher-order free block, it will recursively split it, into lower-order *buddy* blocks, putting the one half (*buddy*) of the split blocks back to the appropriate free list. It will continue splitting the other half, until it arrives at the requested order. At that point, it will allocate the requested order block.

Conversely, when a block is freed, the kernel will attempt to put the free block back to the appropriate free list, based on its order. When doing so, it will attempt to find whether the *buddy* of that block, is also in the free list. If it is, it will coalesce them into a higher order block, and repeat the process for the higher order list.

2.1.1 Per-CPU lists

Serving allocations from the buddy allocator free lists incurs overhead due to locking contention, as for each allocation the kernel must take and hold the lock (spinlock) of the appropriate memory zone. To avoid this overhead for extremely common allocation sizes, namely orders 0 (4KiB) and 9 (2MiB), the Linux buddy allocator includes per-CPU lists (*pcplists*) of free pages, which can be allocated independently by each CPU without the need for synchronization. When an appropriate-sized block is freed, i.e., a 4KiB or 2MiB block, the kernel checks whether it needs to re-fill the per-CPU lists. If so, instead of releasing the free block back to the allocator free lists, it adds it to the per-CPU free list. The kernel also checks whether the pcplists need refill, when an allocation is served from a pcplist. In that case, it will take the spinlock and do a bulk allocation in order to refill said pcplist once.

2.2 The Page Cache

The Linux kernel maintains an in-memory cache of accessed file pages, called the *page cache*. Whenever a file is accessed, the kernel allocates a physical 4KiB frame, which it populates with the contents of the file. It then keeps this page cached in memory, to avoid the I/O in case the same file page is accessed in the future. Writes to file pages in the page cache are also buffered. Dirty pages in the page cache are periodically written back to the filesystem on disk, a process commonly known as *writeback*.

2.2.1 The File-backed Fault Path

File-backed memory in Linux is typically accessed through the mmap() system call or through standard file I/O operations (read(), write()). When a process accesses a file-backed virtual address whose contents are not present in the page cache, the MMU raises a page fault. The Linux page fault handler identifies that the fault occurred in a file-backed VMA and invokes the filesystem's fault() handler, which is responsible for populating the page cache with the requested file contents.

The handler allocates a physical page from the buddy allocator (Section 2.1), reads the file data from the underlying block device, and adds the page to the page cache via add_to_page_cache_lru(). This function inserts the page into the address space radix tree (or XArray [97] in recent kernels), associates it with the file inode, and adds it to the LRU (Least Recently Used) list for memory reclamation purposes. Once the page is populated and added to the page cache, the kernel updates the process's page tables to map the faulting virtual address to the newly allocated physical page.

This demand-paging approach ensures that file data is loaded into memory only when actually accessed, conserving physical memory. However, loading individual pages on-demand can be inefficient, particularly for sequential access patterns where disk I/O latency dominates. To address this, Linux employs a *readahead* mechanism to prefetch file data speculatively.

2.2.2 The Linux Readahead Mechanism

Readahead is a critical I/O optimization technique that prefetches file pages into the page cache before they are explicitly requested by applications [98, 99]. By anticipating future data needs based on access patterns, readahead serves three primary purposes: (i) it hides disk I/O latency from applications by ensuring that data is already present in memory when accessed, (ii) it improves disk utilization by issuing larger, batched I/O requests rather than individual page reads, and (iii) it amortizes the per-request overhead across multiple pages.

The Linux kernel implements an *on-demand readahead* algorithm that dynamically adjusts the prefetch window based on observed access patterns. The key insight is that file access patterns can generally be classified as either sequential or random. For sequential access, aggressive prefetching yields significant benefits, whereas for random access, readahead is largely counterproductive and wastes memory and I/O bandwidth.

Filesystems can also provide hints to influence readahead behavior through the fadvise () system call. Applications can use POSIX_FADV_SEQUENTIAL to indicate sequential access patterns, encouraging aggressive readahead, or POSIX_FADV_RANDOM to disable readahead entirely for random access workloads. The POSIX_FADV_WILLNEED hint explicitly requests prefetching of a specified file region into the page cache, similarly to the readahead() system call [100].

Readahead in Virtualized Environments. In virtualized environments, particularly when VM memory is backed by memory-mapped snapshot files (as discussed in Chapter 4), the host OS readahead mechanism operates on the snapshot file contents without any semantic understanding of the guest memory layout. While readahead can hide I/O latency for sequential snapshot accesses, it may also prefetch pages that correspond to guest free memory or ephemeral allocations, resulting in wasted I/O bandwidth. This semantic gap between host readahead and guest memory semantics motivates the eBPF-based prefetching mechanisms and paravirtualized approaches presented in Chapter 4, which leverage guest knowledge to guide prefetching decisions more effectively than generic readahead heuristics.

2.3 Anonymous Memory

Anonymous memory refers to allocated memory that is not backed by a file. Heap memory allocations by userspace processes, MAP_ANONYMOUS mapping created by the mmap() system call, as well as memory allocations by kernel subsystems and device drivers, are examples of anonymous memory allocations.

2.3.1 The Anonymous Fault Path

Anonymous memory is typically allocated lazily. When a process requests memory (e.g., via malloc), the kernel updates the process's Virtual Memory Areas (VMAs) but does not immediately allocate physical RAM. The page table entries (PTEs) for the new range are left empty.

When the process attempts to access this memory, the MMU raises a page fault. The Linux page fault handler, do_page_fault, identifies that the fault occurred in a valid anonymous VMA and invokes do_anonymous_page().

1. **Read Faults:** If the fault is triggered by a read access, the kernel maps the virtual address to a global, read-only physical page filled with zeros, known as the **Zero Page**. This optimization avoids allocating physical RAM for uninitialized memory that is only being read.

2. **Write Faults:** If the fault is a write, or if the process attempts to write to a CoW (Copy-on-Write) Zero Page, the kernel allocates a fresh 4KiB physical page from the buddy allocator (Section 2.1). The page is cleared (zeroed) for security reasons to prevent data leakage from previous users of the frame, and the process's page tables are updated to point to this new private page.

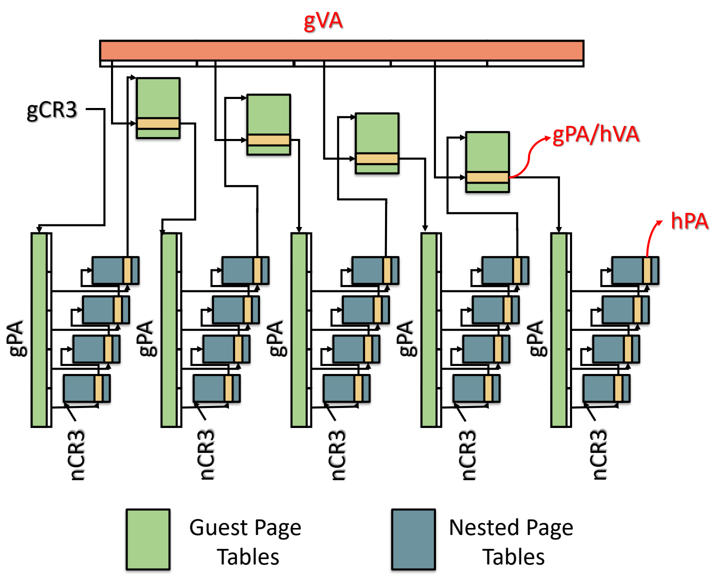


Figure 2.2: The two-dimensional page walk of nested paging.

2.4 Nested Paging

Modern processors support hardware-assisted virtualization (Intel VT-x with EPT, AMD-V with NPT, and ARMv8-A with Stage-2 Translation). This allows the CPU to handle two layers of translation without software intervention for every access (Figure 2.2):

1. **Guest Virtual to Guest Physical (GVA → GPA):** Managed by the Guest OS using standard page tables (CR3 on x86, TTBR on ARM).
2. **Guest Physical to Host Physical (GPA → HPA):** Managed by the Hypervisor (KVM) using Nested Page Tables (EPT/NPT/Stage-2).

When a VM runs, the hardware walks the Guest Page Tables to translate a GVA to a GPA. It then treats this GPA as a virtual address for the second stage, walking the Nested Page Tables to find the final HPA. A miss in the Guest tables triggers a fault injected into the Guest OS. A miss in the Nested tables (e.g., because the host has not yet allocated the memory) triggers a "VM Exit," trapping to KVM on the host.

2.5 eBPF in the Linux Kernel

Originally introduced for high-performance packet filtering, the extended Berkeley Packet Filter (eBPF) [89] has evolved into a framework within the Linux kernel that allows userspace to safely load, sandbox and execute code within the Linux kernel. eBPF enables user processes to attach custom bytecode to a wide range of kernel hooks. These programs execute in kernel context with strict safety guarantees enforced by a static verifier, enabling kernel programmability without requiring kernel modules, rebooting, or changes to core kernel code.

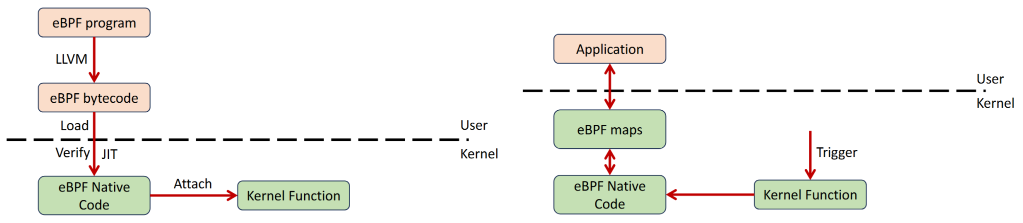


Figure 2.3: The eBPF execution model.

2.5.1 Programming and Execution Model

An overview of the eBPF execution model is shown in Figure 2.3.

An eBPF program consists of LLVM-compiled bytecode that is loaded into the kernel via the bpf() system call. At load time, the in-kernel verifier performs structural and semantic analysis to guarantee memory safety, bounded loop execution, and safe interaction with kernel helper functions. eBPF programs are then JIT-compiled into native instructions and attached to hook points, which may include static kernel tracepoints, dynamic kprobes, scheduler events, LSM hooks, or custom attachment sites added by subsystem developers.

eBPF programs operate in a constrained execution model: they are stateless across invocations, with persistent state stored exclusively in eBPF maps. Maps are generic key-value data structures residing in kernel memory, shared between eBPF programs and userspace.

2.5.2 Relevance to OS Memory Management

Although originally designed for packet processing, the architectural properties of eBPF, namely low-overhead invocation, in-kernel execution, and safe extensibility—make it particularly relevant for memory management specialization. eBPF could allow memory management policies to be selectively overridden, enriched, or parameterized on a per-application basis, without invasive kernel modifications.

CHAPTER 3

Elastic Translations: Fast Virtual Memory with Multiple Translation Sizes

3.1 Overview

This chapter presents Elastic Translations (ET), a holistic memory management solution, that seamlessly and efficiently supports multiple translation granules, extending beyond 2MiB, while remaining resilient to external memory fragmentation. ET currently target the OS-assisted TLB coalescing feature of the ARMv8-A architecture [41] and the translation granules enabled by it, i.e., 64KiB and 32MiB, but the design is amenable to other architectures, e.g., RISC-V [39], or micro-architectures, e.g., AMD Zen [101], which support different ranges of translation granules, via either OS-assisted [32] or HW-assisted [102] TLB coalescing. Table 3.1 provides a brief description of the constituent kernelspace and userspace components of ET, while Table 3.2 outlines how ET compares with existing state-of-the-art and state-of-practice OS interfaces for multi-grained address translation.

Section 3.2 of the chapter provides the necessary background, regarding large pages and translation granules, as well as their interplay with the OS memory manager (Section 3.2.1 and 3.2.2). It presents a comprehensive evaluation of the translation granules enabled by the ARMv8-A OS-assisted TLB coalescing feature, namely 64KiB and 32MiB, for both native and virtualized execution (Section 3.2.3). The results showcase the performance potential of both 64KiB and 32MiB translation granules, with the section concluding with a discussion of the limitations of

Component	Purpose	Main Contribution
Transparent Contig-Bit (kernelspace)	Transparent, opportunistic creation of 64KiB and 32MiB translations via OS-assisted TLB coalescing	Native support for 32MiB translations Virtualization support for all sizes
CoalaPaging (kernelspace)	Transparent, opportunistic creation of contiguous 64KiB and 32MiB mappings across page faults	Practical and scalable allocation policy for 32MiB mappings, with minimal impact on fault latency and memory bloat
CoalaKhugepaged (kernelspace)	Asynchronous creation of 64KiB and 32MiB mappings via migrations	Enables the creation of 64KiB and 32MiB translations under memory pressure (fragmentation)
Leshy Profiler (userspace)	Runtime translation size selection guidance via MMU overhead profiling	Optimal translation size selection from an extended range of sizes via lightweight HW-assisted TLB miss sampling

Table 3.1: The kernelspace and userspace components of Elastic Translations.

existing OS interfaces for multi-grained AT, which render the OS unable to fully and optimally harness this potential (Section 3.2.4), motivating the development of Elastic Translations.

Section 3.3 provides an in-depth description of the design and implementation of ET. ET enable Linux to transparently and opportunistically coalesce contiguous memory mappings (regions) into larger translations, via the OS-assisted TLB coalescing feature of the ARMv8-A architecture (*contig-bit*), for both 64KiB and 32MiB translations, for both native and virtualized execution (Section 3.3.1). The *CoalaPaging* (Section 3.3.2) and *CoalaKhugepaged* (Section 3.3.3) coalescing-aware extensions to the Linux memory manager are responsible for generating the required memory contiguity to drive this transparent *contig-bit* mechanism. *CoalaPaging*, based on contiguity-aware paging [103], *opportunistically* allocates suitable 4KiB and 2MiB pages *across page faults* in order to lazily generate intermediate-sized contiguity that matches the coalescing size supported by the hardware. *CoalaKhugepaged* extends Linux khugepaged [61], enabling the asynchronous creation of 64KiB and 32MiB mappings via migrations. The two mechanisms work synergistically, i.e., when *CoalaPaging* fails to allocate all the contiguous pages required for a 64KiB or 32MiB translation at fault time, e.g., due to external fragmentation, *CoalaKhugepaged* will exploit the partial contiguity to migrate fewer pages. *CoalaPaging* and *CoalaKhugepaged* along with the transparent contig-bit management comprise the *Elastic Translations in-kernel mechanisms*. The section then describes how ET utilize HW-assisted sampling to periodically record the TLB misses of workloads at runtime, and design a userspace profiler, *Leshy*[1], which implements the *ET policies* for translation size selection (Section 3.3.4). Leshy tracks the virtual address and page walk latency for each sampled TLB miss and generates an MMU (address translation) overhead heatmap of the address space. It then maps address space regions to trans-

[1]Leshy is a mythological guardian spirit that can change in size.

lation granules with the goal of minimizing address translation costs based on the aforementioned heatmap. Finally, Leshy drives the ET in-kernel mechanisms, by loading the generated translation-size mappings (hints) into the kernel.

Section 3.3.5 presents an extension to the original ET design, to enable the redistribution of reserved contiguity at runtime via asynchronous demotions. Leshy is extended to dynamically switch between TLB miss sampling and memory access sampling, based on the MMU intensity of the application at runtime. When TLB misses drop below a certain threshold, Leshy switches from TLB miss sampling to memory access sampling, in order to detect regions of the address space, mapped by larger translation granules, that have potentially gone cold. For these cold address space regions, it then issues demotion hints to a complementary in-kernel ET component. The kernel will asynchronously migrate the demoted regions to non-contiguous parts of the physical address space, relinquishing the unused contiguity back to the system. CoalaPaging and CoalaKhugepaged will then be able to harness this released contiguity for other applications that are experiencing MMU pressure.

ET is designed and implemented for Linux v5.18 for the ARMv8-A architecture. The experimental evaluation of ET (Section 3.4) on a real ARMv8-A server (Section 3.4.1) shows that transparent 64KiB translations perform closely to 2MiB pages for memory-intensive workloads with small footprints for both native and virtualized execution. For larger workloads, transparent 32MiB translations improve performance by 10% on average and up to 39% over THP for native execution and by 30% and up to 150% for virtualized execution. Finally, Leshy's microarchitectural-aware policies guide ET to map the footprint of workloads by utilizing an optimal mix of all of the available translation sizes, in order to minimize address translation overhead. This improves overall performance under fragmentation by 12% on average and up to 20% over THP and state-of-the-art while consistently reducing the number of 2MiB pages used.

Section 3.5.1 discusses how userspace can provide policies and hints to the OS to drive translation size selection, both at fault time and asynchronously. First, it presents the original design of ET, where the userspace profiler, Leshy, employs an *madvise()*-based interface, to drive the in-kernel ET mechanisms, CoalaPaging and CoalaKhugepaged. It then proposes a novel approach, which uses the eBPF Linux kernel framework [89] to programmatically control the translation size selection mechanisms of the OS memory manager from userspace.

Finally, the chapter examines the implications and trade-offs of the design decisions made while developing ET, considers future directions for translation size elasticity in the OS and, concludes with a review of related work on address translation, large pages and translation granules, positioning ET contributions within existing literature.

3.2 Background and Motivation

As the working sets of modern server workloads have begun to outgrow TLB reach, i.e., the maximum memory that the TLB is able to cache the translations for, the address translation overhead experienced by applications, stemming mostly from a corresponding significant increase in TLB miss rates, has been steadily rising [6, 7, 17, 18, 103–110]. Compounding the issue, page walks are expected to get costlier as i) paged virtual memory transitions from 4 to 5-level radix page tables [111, 112], and ii) HW-assisted virtualization has become ubiquitous. TLB misses in HW-assisted virtualization are notoriously more expensive [7, 56, 103, 104], as they require the nested traversal of the guest and hypervisor page tables [55]. Industry's response to the problem at hand has been two-pronged.

On the one hand, CPU cores have steadily moved towards larger TLB capacities, first with the introduction of a multi-level TLB hierarchy, and then with the continued increase of both L1 and L2 TLB capacities. However, TLB capacity has been unable to keep up with the ever increasing memory footprints of modern workloads – a phenomenon dubbed address translation (AT) wall [110]. This is due to a combination of factors. The end of Dennard scaling [113] and the slowing down of Moore's law [114] has rendered silicon real estate precious, making the power and silicon costs of increased TLB capacities more pronounced. This in turn has led CPU architects to trade off AT performance for other chip functions, e.g., memory caches. Moreover, as TLBs sit in a critical path both at the frontend and at the backend of the CPU pipeline, they come with stringent latency restrictions, which effectively limit their maximum attainable capacity.

Complementing sheer TLB capacity, modern CPUs also add support for large page sizes, i.e., sizes larger than the base page size, typically 4KiB, to expand the TLB reach and minimize page walk overhead [9]. In particular, the x86 architecture, which comprised the vast majority of server systems in the past couple of decades, supports two different large page sizes, 2MiB and 1GiB, which are implemented by storing larger virtual-to-physical translations higher up the page table radix tree. However, as discussed in the following paragraphs, this is not without trade-offs. As a result, OS designers typically eschew the page sizes at the high end of the supported spectrum, i.e., 1GiB, favoring a middle ground with more balanced trade-offs, i.e., 2MiB, for userspace memory allocations [29, 61, 65]. This, combined with the limited rate of increase of TLB capacities, compared to that of application memory footprints, in recent years, exacerbates the AT wall problem. Figure 3.1 showcases the issue, by plotting the TLB reach of modern CPU cores, when using different page sizes. The TLB reach attainable with 2MiB large pages, the most popular and widely supported large page size [29, 61, 65], has started to fall behind, when compared to the typical available physical memory of modern server systems.

This section first surveys the evolution of HW and OS support for large pages and multi-grained address translation. It then focuses on the intermediate translation granules, unlocked by the OS-assisted TLB coalescing feature of the ARMv8-A architecture, i.e., 64KiB and 32MiB. A comprehensive evaluation of these translation granules, both for native and virtualized execution, showcases their performance potential. The section concludes with an examination of the existing OS mechanisms and interfaces for multi-grained address translation, which uncovers inherent limitations in these mechanisms and interfaces, that prevent the OS from fully harnessing the performance potential of multiple translation granules, motivating the development of Elastic Translations.

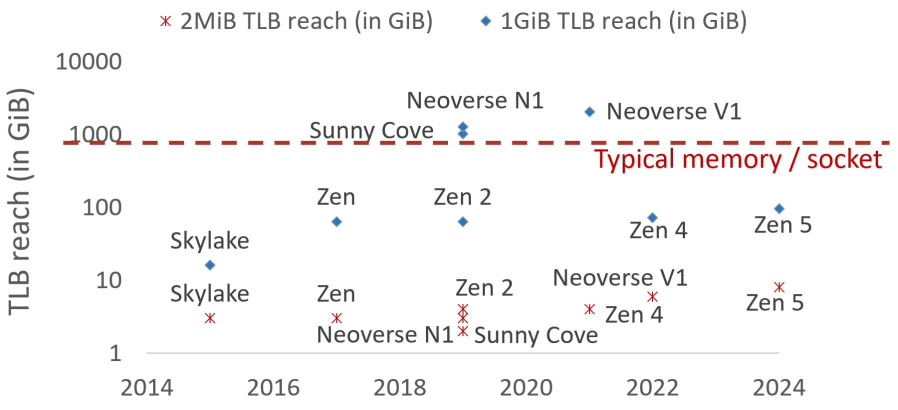


Figure 3.1: Evolution of the TLB reach of modern CPU cores.

3.2.1 OS support for Large Pages

OS large page interfaces can be broadly classified into two categories, non-transparent and transparent. Non-transparent interfaces support all available large page sizes, i.e., for x86, 2MiB and 1GiB, but require applications to explicitly request which specific size to use for which specific region of the address space. Additionally, the large pages need to be allocated in advance and are generally unavailable to the OS memory manager, e.g., for reclaim under memory pressure. This approach is adopted by Linux HugeTLB [59]. By contrast, transparent large page interfaces obviate the need for explicit opt-in by userspace applications and are tightly coupled with the core OS memory management subsystem. However, they typically only support 2MiB pages in modern OSes [29, 61, 65] (Table 3.2) and task the OS with the responsibility of size selection.

Transparent large pages are formed either synchronously or asynchronously. The synchronous path is implemented via demand paging, when a page is first accessed (written to). When a page is accessed for the first time, a page fault is triggered, which the OS handles by allocating physical memory for the faulting page. With transparent large pages, the OS must decide i) whether a large page will be allocated to serve the fault, and ii) which large page size to use, when multiple large page sizes are supported transparently (*fault policy*). Page migrations can also be leveraged

	Transparent	Faults		Translation			Promotions	
		Supported Sizes	Policy	Supported Sizes	Policy	Virtualization Support	Supported Sizes	Policy
HugeTLB [59]	✗	4KiB, **64KiB** 2MiB, **32MiB** 1GiB	Pre-allocation Single user-defined size per VMA	4KiB, **64KiB** 2MiB, **32MiB** 1GiB	Defined by fault size	4KiB 2MiB 1GiB	✗	✗
mTHP [60, 61]	✓	4KiB, **64KiB** 2MiB	Eager allocation of the largest possible size Fallback on failure	4KiB, **64KiB** 2MiB	Defined by fault or promotion size	4KiB 2MiB 1GiB	2MiB	Migrate to 2MiB Region selection: Linear scan
FreeBSD [29]	✓	4KiB	2MiB reservation at first 4KiB fault Use reservation to serve the rest	4KiB 2MiB	Defined by fault or promotion size	4KiB 2MiB	2MiB	In-place promotion to 2MiB when every 4KiB page is faulted-in
HawkEye [62]	✓	4KiB 2MiB	Same as mTHP Asynchronous pre-zeroing	4KiB 2MiB	Defined by fault or promotion size	4KiB 2MiB	2MiB	Selectively migrate to 2MiB Region selection: Access frequency based on page table scanning
Trident [63]	✓	4KiB 2MiB 1GiB	Same as HawkEye	4KiB 2MiB 1GiB	Defined by fault or promotion size	4KiB 2MiB 1GiB	2MiB 1GiB	Migrate to largest size possible Fallback on failure Selection same as mTHP
Elastic Translations	✓	4KiB 2MiB	4KiB / 2MiB eager allocation based on VMA size **Opportunistic coalescing-aware allocations across faults**	4KiB, **64KiB** 2MiB, **32MiB**	4KiB or 2MiB based on fault or promotion size **Opportunistic promotion to 64KiB or 32MiB**	4KiB, **64KiB** 2MiB, **32MiB**	**64KiB** 2MiB, **32MiB**	Selectively migrate to 64KiB, 2MiB, 32MiB Region Selection: Size hints based on HW TLB miss sampling

Table 3.2: State-of-practice and state-of-the-art large page interfaces.

to create large pages asynchronously, off the fault path. The OS periodically scans the memory of running processes and finds discontinuous groups of pages suitable for promotion to a large page. It then allocates a large page in physical memory and migrates to it the aforementioned discontinuous pages. In that case, the OS must decide i) which virtual regions are worth promoting to larger pages and ii) what will be the target size, if multiple sizes are supported (*promotion policy*). For both cases, the allocated memory is mapped to userspace by updating the process page tables. At that point, the OS must select, from the list of available MMU-supported translation sizes, an appropriate size with which to map the allocated memory (*translation policy*). Table 3.2 shows the different policies adopted by state-of-practice and state-of-the-art.

Linux THP [61] opts for a greedy approach, that always allocates entire 2MiB pages at fault time. This has the advantage of backing the workload's address space with large pages as early as possible but performs poorly under memory pressure [62, 67, 69, 71, 72]. The unsolicited use of 2MiB pages leads to their sub-optimal distribution among processes and address space regions, when they run low in the system due to external fragmentation. Linux THP also includes a kernel thread, *khugepaged*, that asynchronously scans and promotes (to 2MiB) suitably-aligned 2MiB regions which are fully or partially backed by 4KiB pages, by migrating the constituent base pages to a contiguous 2MiB block of memory. State-of-the-art improves upon THP by using i) base page utilization [67–69], ii) access frequency sampling [62, 67], iii) coarse-grained MMU overhead profiling [62] and iv) user-provided profiles [72] to select which pages to promote to 2MiB, either at fault time [72] or asynchronously [62, 67–69].

Linux recently added support for multi-sized THP (mTHP) [60]. mTHP introduces a fault-time policy which enables the allocation and mapping of 64KiB blocks of memory. At fault time, Linux will attempt to allocate a 2MiB page. If the 2MiB allocation fails, it will then fall back to 64KiB instead of 4KiB. At the moment, mTHP works solely at fault-time, as there's no support for asynchronous mTHP promotions, and only for native execution. Additionally, it does not support 32MiB translations.

FreeBSD transparently supports 2MiB pages using a reservation based fault-time policy [29, 65]. It reserves a 2MiB block of memory at first fault, but faults the pages in at a 4KiB granularity, promoting them to a large page in place during the last such fault. This strategy keeps fault latency bounded but delays the formation and mapping of large pages [71]. FreeBSD does not employ any kind of asynchronous promotions via migrations.

Neither OS supports 1GiB pages transparently, as this would require the OS to track 1GiB-aligned free blocks and reserve them at fault time, potentially penalizing fault performance and increasing internal fragmentation. Moreover, 1GiB pages quickly become scarce [105, 115, 116], due to external memory fragmentation. Consequently, prior art for transparent 1GiB support [63] mainly relies on asynchronous promotions and aggressive compaction to generate the required contiguity. Neither OS nor state-of-the-art design supports 32MiB pages.

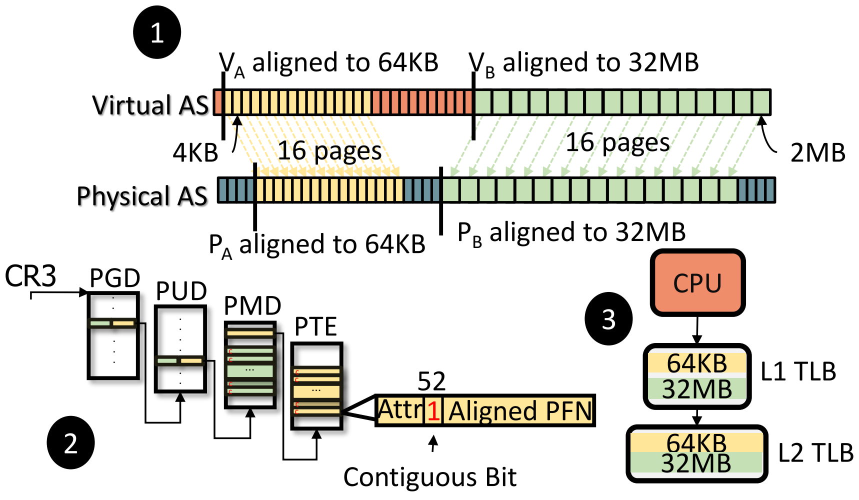


Figure 3.2: ARMv8-A OS-assisted TLB coalescing.

3.2.2 OS-assisted TLB coalescing

TLB coalescing [27, 28, 42] is a technique that caches the translation of N contiguously mapped pages using a single TLB entry. While TLB coalescing can be implemented entirely in HW, the coalescing factor N is typically limited [101, 102] leading to diminishing results [117].

The ARMv8-A architecture supports OS-assisted TLB coalescing instead. Figure 3.2 shows how ARMv8-A enables a coalescing factor of $N = 16$ with OS assistance. Its page table entries include a *contiguous bit* (bit 52) which, if set by the OS in $N = 16$ consecutive page table entries, it indicates to the translation hardware that these N pages are contiguous and *properly aligned according to the coalescing factor*. In Fig. 3.2 the [VA..VA$_{+15}$] virtual 4KiB pages are contiguously mapped to [PA..PA$_{+15}$] physical 4KiB pages (1). VA and PA are also aligned to 64KiB (16 * 4KiB). Since every page in the 64KiB range meets the above criteria, the OS can set the contiguous bit in the 16 consecutive (yellow) PTE entries (2) mapping these 16 virtual pages to their physical frame numbers (PFNs). This allows the MMU to coalesce them into a single TLB entry and cache them as such in the TLB (3). Coalescing increases the TLB reach, effectively forming an intermediate translation size. Similarly, [VB..VB$_{+15}$] contiguously mapped 2MiB pages (green) are coalesced to a 32MiB intermediate translation via setting the contiguous bit in their 16 consecutive (green) PMD entries. Finally, the contiguous bit is also supported in the PMD and PTE levels of nested page tables, which allows the MMU to coalesce contiguously mapped pages for virtualized workloads as well. RISC-V supports a similar OS-assisted TLB coalescing scheme with the Svnapot extension [39].

The performance potential of these intermediate translation sizes remains largely unexplored, as robust OS support for coalesced translations is mostly missing (Section 3.2.1). Preliminary transparent support in Linux exists only for 64KiB translations and only for native execution via mTHP [60, 118]. The cumbersome HugeTLB interface supports both 64KiB and 32MiB translations, albeit *only for native execution* (Table 3.2).

State-of-practice and start-of-the-art systems mainly focus on the 4KiB / 2MiB / 1GiB page sizes supported by x86 (Table 3.2). Most designs target 2MiB pages, attempting to maximize performance by deciding when, how, and which 4KiB base pages to promote to a 2MiB large page. ARMv8 and RISC-V architectures support more translation sizes via OS-assisted TLB coalescing. Their transparent support remains limited and their performance largely unexplored.

3.2.3 OS-assisted coalescing: Performance potential

To assess the performance potential of 64KiB and 32MiB translations, the thesis uses an ARMv8-A server to evaluate them, via the HugeTLB interface, versus 4KiB, 2MiB and 1GiB pages, for both native and virtualized scenarios. For virtualized execution, the thesis extends Linux and KVM to support contiguous-bit intermediate translations. Figure 3.3 summarizes the evaluation results for two sets of workloads (discussed in Section 3.4.1): i) memory intensive workloads that operate on small objects (top) and ii) big-memory workloads (bottom).

For the first set of workloads, 64KiB translations provide up to 10% (native) and 15% (virtualized) performance benefit over 4KiB pages; almost matching the performance of 2MiB pages in some cases. *64KiB translations could be leveraged to improve address translation performance*

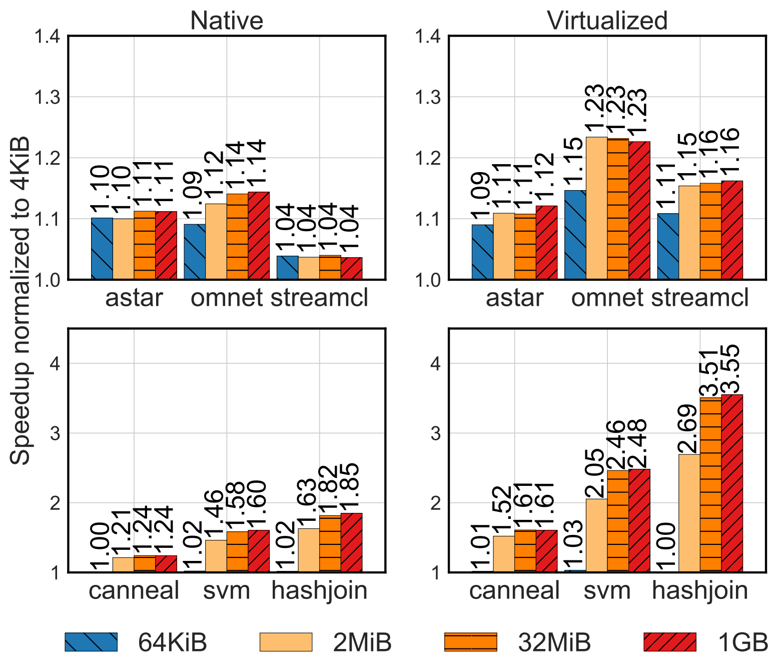


Figure 3.3: Performance of HugeTLB intermediate-sized translations on a non-fragmented ARMv8-A machine.

while obviating the need for larger 2MiB allocations, especially under memory pressure or fragmentation [116].

For the second set of workloads, 32MiB translations often outperform 2MiB large pages, by up to 30% in virtualized execution. Notably, they provide performance close to that of 1GiB pages. *32MiB translations can effectively mitigate translation costs that 2MiB pages are unable to cover, while relaxing the contiguity requirements of 1GiB pages, that are extremely hard to meet on a long-running system [105, 115, 116].*

64KiB and 32MiB translation sizes provide significant performance gains and can be exploited to address limitations exhibited by the 2MiB / 1GiB large page model.

3.2.4 The conundrum of translation size selection

Support for a single transparent large page size, as implemented in Linux and FreeBSD, reduces translation size selection to a binary decision per 2MiB region of the virtual address space: whether to back each region by a 2MiB page or not. Intermediate-sized translations complicate size selection. A methodology is needed to estimate the performance impact of different translation sizes on the different regions of a workload's address space.

The thesis employs MMU overhead as a proxy for estimating the performance impact of trans-

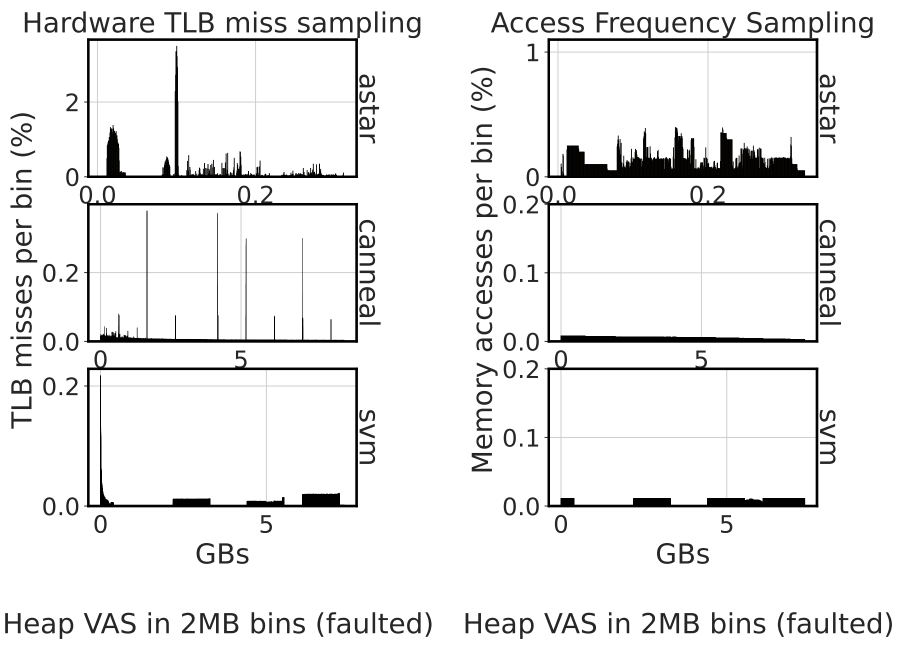


Figure 3.4: Narrow address space regions account for most of the translation overhead (MMU hotspots). HW-assisted sampling is able to detect them at a higher resolution than page access frequency sampling.

lation size [62, 72]. To that end, *fine-grained HW-based TLB miss sampling is used to identify the TLB-miss heavy regions of a process address space.* It leverages the ARMv8-A Statistical Profiling Extension (SPE) [41] to sample the TLB misses of workloads and track the virtual address and page walk latency for each sampled miss.

Figure 3.4 (left) shows the distribution of misses for three MMU-intensive workloads, divided in 2MiB bins. There are wide regions which exhibit minimal overheads, and narrow spikes that are responsible for the majority of the TLB misses. Notably, a single 2MiB region is responsible for ∼*5%* of the total TLB misses for astar. Such *fine-grained translation-overhead information can be leveraged to optimally assign different translation sizes to different virtual regions based on the MMU pressure they generate.*

Prior art relies on page-based access frequency sampling to estimate MMU overhead [62, 63, 67] and guide 2MiB promotions [62, 67]. Figure 3.4 (right) also presents the access frequency heatmaps for the same workloads, generated by periodically sampling the access bit of each populated page of the address space [62]. HW-assisted sampling is able to identify MMU hotspots at a higher resolution. Not every frequently accessed page contributes equally to translation overhead. The evaluation section (Section 3.4) further elaborates and quantifies this difference.

The address space of memory intensive workloads exhibits translation overhead hotspots. HW-based sampling manages to accurately detect them, unlocking the potential for informed translation size selection.

3.3 Elastic Translations

This section presents the design and implementation of *Elastic Translations (ET)*, synergistic mechanisms and policies (Table 3.1) that i) enable the OS to transparently generate and manage intermediate-sized translations in native and virtualized environments (*Transparent Contig-bit*, *CoalaPaging*, *CoalaKhugepaged*) and ii) optimize translation size selection from the now extended pool of available translation sizes (*Leshy*). It concludes with an extension to ET, and specifically Leshy, that allows the redistribution of reserved contiguity via asynchronous demotions.

3.3.1 Transparent contiguous bit management

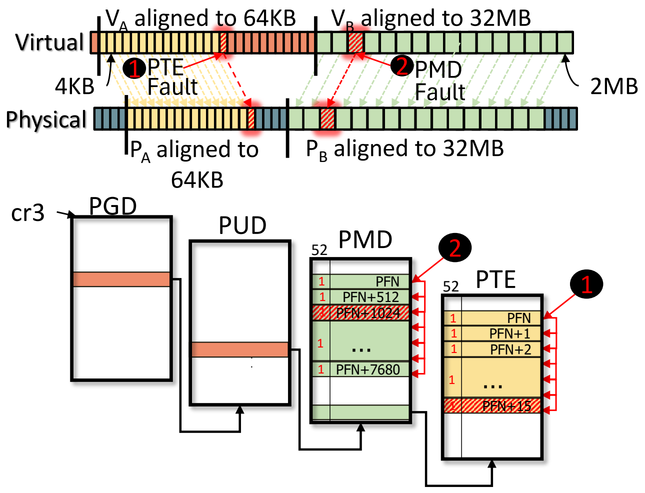


Figure 3.5: ET contiguous bit management during a PMD (2MiB) (2) and PTE (4KiB) (1) fault.

In order to transparently support and opportunistically create intermediate sized translations, ET need to i) detect suitably-aligned contiguously-mapped pages, and ii) transparently promote them to intermediate-sized translations by setting the contiguous bit in each page table entry. Coalesced translations must also be demoted when their constituent pages are no longer contiguous. Figure 3.5 depicts the mechanism. Whenever a page table entry is created or modified, ET check the N (16 in this case) page table entries which belong to the same coalescing range. That is the 16 neighboring entries in a 64KiB range for 4KiB page entries (PTEs) or a 32MiB range for 2MiB page entries (PMDs), starting from the first 64KiB- or 32MiB-aligned entry. If every entry in this range is: i) suitably aligned, both physically and virtually, i.e., for a 4KiB PTE, mapping the virtual page number (VPN) to a physical frame number (PFN), $[PFN mod 16 == VPN mod 16]$, ii) physically contiguous with regard to the other entries in the range, i.e., for a 4KiB PTE $[PFN_{n+1} == PFN_n + 1]$, and iii) has compatible page flags and access permissions as the rest of the range entries, *ET promote the range, by setting the contiguous bit in each PTE or PMD in the range.* Reversely, when a page entry modification invalidates any of the above, ET demote the range by clearing the contiguous bit accordingly and flushing the corresponding TLB entry. When the range is not fully faulted in, ET fall back to the default Linux path for setting the PTE or PMD respectively. The latency overhead of this mechanism is quantified in the evaluation section (Section 3.4).

Whenever the size of a translation entry changes, ARMv8-A mandates invalidating the entry and flushing it from the TLB. This rule is called *break-before-make* in the architecture reference manual [41]. This is always required when demoting an intermediate translation, since leaving stale contiguous entries in the TLBs can enable otherwise invalid memory accesses. However, transparently creating an intermediate translation by setting the contiguous bit does not invalidate its constituent page translations that may be still cached. They still map to the same memory locations with the same permissions. This obviates the need for a TLB flush, thus, as an optimization, *ET opt for lazily flushing newly-promoted page table entries.*

Virtualization support. ET also support virtualized execution under KVM, transparently managing the contiguous bit in the nested page tables. HW-assisted virtualization utilizes nested paging for memory virtualization. The guest OS page tables translate guest virtual addresses (GVA) to guest physical addresses (GPA). The nested page tables, managed by KVM, translate these GPAs to actual host physical addresses (HPAs). The TLB then caches GVA to HPA translations [55, 104].

For ET, the contiguous bit in the guest page tables is managed by the guest OS as described in the previous paragraphs. The contiguous bit in the nested page tables is managed during nested faults by KVM. Allocations triggered by nested faults will eventually need to update the host page tables of the virtual machine monitor (VMM). ET already hook this path, as the VMM is a regular host process, and will thus detect and promote coalesce-able ranges in the VMM host page tables. ET extend KVM so that these promotions are reflected to the nested page tables of the VM, by setting the contiguous bit of the corresponding shadow page table entries (SPTEs). Similarly, whenever the host demotes an intermediate translation, e.g., due to unmapping or migration, KVM is notified and demotes the corresponding SPTEs. By promoting intermediate translations in both host and guest, ET allow the caching of coalesced 2D GVA to HPA translations in the TLB.

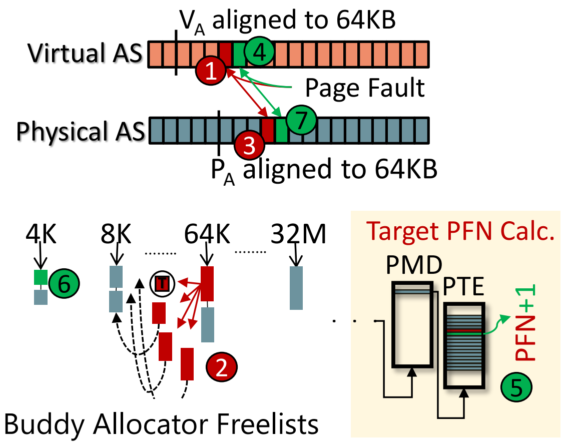


Figure 3.6: CoalaPaging target PFN calculation and allocation. For the first fault in a 64KiB VA range, CoalaPaging gets an order-4 block from the allocator (1), allocate the 4KiB page that matches VA's alignment and return the rest of the block to the allocator (2), and set the PFN of the respective page table entry (3). For subsequent faults in this range (4), ET scan the page table entries of the 64KiB region, calculate a target PFN based on the PFN of the previously faulted entry (5) and attempt to allocate it (6).

3.3.2 Coalescing-aware Paging

To generate the inter-page contiguity required for intermediate-sized translations, the thesis proposes and designs a coalescing-aware allocation policy, *CoalaPaging*, based on contiguity-aware paging (CAPaging) [103]. The goal is to maximize the formation of suitably aligned and contiguous ranges of pages, i.e., 64KiB ranges for PTEs and 32MiB for PMDs. The core idea is to mirror the TLB coalescing logic in the allocation path. On each fault, CoalaPaging attempts to either create or extend a contiguous and aligned 64KiB or 32MiB range of pages, by scanning the page tables and selecting a suitable target page. Figure 3.6 depicts the coalescing-aware allocation process.

First fault. When handling a fault, CoalaPaging scans the page table entries of the 64KiB- or 32MiB-aligned range which the faulting address belongs to. For the first fault within such a range, CoalaPaging attempts to find a suitably sized and aligned free block. CoalaPaging then finds and allocates the page of the block whose PFN alignment with regard to the coalescing factor matches the alignment of the faulting address.

For a 64KiB range of 16 4KiB pages, CoalaPaging finds a free 64KiB block, by searching the order-4 (64KiB) and higher free-lists of the buddy allocator and allocates the 4KiB page whose `[PFN mod 16 == VPN mod 16]`, where PFN is the physical frame number of the page and VPN is the virtual page number of the faulting address. However, CoalaPaging *neither allocates nor reserves the block*. Once the target page is allocated, the remaining pages are added back to the allocator free-lists. To maximize the time window during which these pages remain available, CoalaPaging appends them to the tail of their respective buddy lists. Figure 3.6's steps 1-3 depict the allocation process for the first fault in a coalescing range. CoalaPaging operates in a similar way for THP faults, but now has to find 32MiB (order-13) free blocks. Linux only tracks by default contiguous blocks up to 4MiB (order-10). For CoalaPaging, the kernel is configured it to track up to 32MiB blocks in its allocator free lists.

Subsequent faults. To identify the target physical page for subsequent faults, CoalaPaging scans the page table entries of the 64KiB or 32MiB range that the faulting address belongs to, searching for a populated page table entry. When such a previously faulted PFN is found, CoalaPaging uses it as an *anchor* to calculate the allocation target. Specifically, CoalaPaging first checks that the anchor PFN is properly aligned (as described in the preceding paragraph), and if not, it aborts the CoalaPaging allocation. CoalaPaging then aligns the anchor PFN to 64KiB or 32MiB, depending on fault type, and add the relative index of the faulting VA within the 64KiB or 32MiB range.

CoalaPaging extracts all the necessary information for the target PFN calculation from the page table state, eliminating any additional metadata requirement, in contrast to e.g., CAPaging. Figure 3.6's steps 4-6 depict the process. Section 3.4 quantifies the latency overhead of page table scanning.

Multi-programmed Execution. In a multi-programmed scenario, CoalaPaging coordinates fault-time allocations of different programs, directing them to different parts of the physical address space. As described in 3.3.2, the first CoalaPaging fault in a 32MiB range will allocate a single 2MiB page from a free 32MiB block and release the rest back to the buddy lists. Subsequent faults in this 32MiB VA range will use the page table to compute the anchor PFN and request the correct physical page based on the faulting VPN. Concurrent allocation requests from other programs will follow the same steps, either allocating a 2MiB page from a new 32MiB block or attempting to allocate one from the previously split 32MiB block, based on information found in the page table. As a result, different programs under ET do not compete for the same buddy blocks and are all able to create 32MiB mappings across faults, on a best-effort basis, as long as there is 32MiB-contiguity available in the system. The same applies for 4KiB faults and 64KiB translations. The ET effectiveness in multi-programmed scenarios is evaluated in Section 3.4.2.

Virtualization support. CoalaPaging works without any modifications in virtualized execution. It is independently employed by the guest and the host – generating contiguity independently in the two dimensions. As guest faults trigger nested faults on the host, this simple scheme is sufficient to generate 2D contiguity, similarly to THP [103].

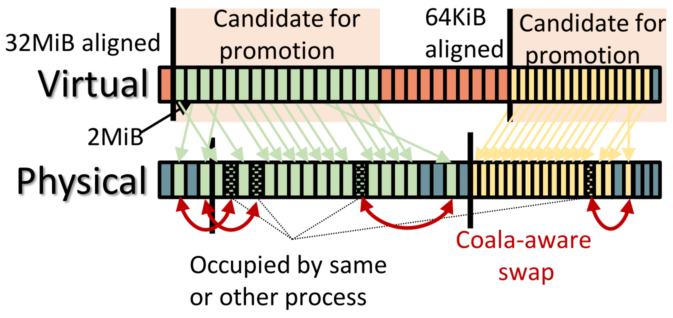


Figure 3.7: Coalescing-aware khugepaged.

3.3.3 Coalescing-aware promotions

ET also support asynchronous promotions via CoalaKhugepaged. For 2MiB pages, Linux khugepaged periodically selects an active process, in a round-robin fashion, and performs a linear scan of its address space, promoting any suitable properly-aligned region, not yet backed by a large page, to 2MiB. In order to promote a region to 2MiB, khugepaged allocates a new 2MiB page and copies the constituent base 4KiB pages to the allocated large page. Khugepaged also includes knobs to control the allocation aggressiveness and CPU overhead of scanning and migrations, allowing the user to control how many pages to scan or collapse per second and including a back-off policy when large page allocations fail due to external fragmentation. CoalaKhugepaged augments khugepaged for optimized coalescing-aware promotions to intermediate-sized translations (64KiB and 32MiB). CoalaKhugepaged works synergistically with CoalaPaging by taking advantage of partially contiguous groups of pages to reduce the number of migrations required for promotion. When CoalaPaging is able to create only a partially contiguous range at fault time, CoalaKhugepaged will attempt to utilize in-place promotions, migrating only misplaced pages to their target PFN if possible (Figure 3.7). If any of the target PFN cannot be replaced (e.g., due to unmovable pages [115]), CoalaKhugepaged will fallback to the default khugepaged behavior, migrating the whole range to freshly allocated memory. To that end, Linux compaction logic is also tuned to work for intermediate-sized blocks (i.e., 32MiB). Asynchronous promotions to intermediate-sized translations, apart from improving resilience to external fragmentation, enable ET to take advantage of informed runtime promotion policies as discussed in 3.3.4.

Fairness. CoalaKhugepaged prioritizes ET-enabled processes, instead of iterating over all running processes in the system (same as [62]), and substitutes linear address-space scan with priority-address-range scanning, guided by TLB miss profiling (as described in 3.3.4). When multiple ET-enabled processes run in the system, CoalaKhugepaged will distribute contiguity among them in a round-robin manner, similar to [62].

3.3.4 Translation size selection policies

With the ET in-kernel mechanisms in-place, the thesis must now devise selection policies to harness the performance potential of the expanded range of supported translation sizes.

Fault-time allocation. At fault time, CoalaPaging uses the size of the faulting virtual memory area (VMA) as an estimator to guide translation size selection. Specifically, when 64KiB translations are able to cover the entire faulting VMA while staying within TLB reach, CoalaPaging attempts to opportunistically create 64KiB translations via base 4KiB fault-time allocations. For larger VMAs, CoalaPaging aims for opportunistic 32MiB translations via THP faults (Section 3.3.2). Similarly to mTHP [60], CoalaPaging employs an incremental fallback policy to smaller translation sizes in case of allocation failure.

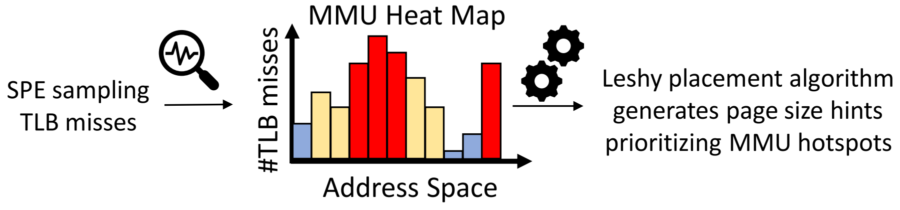

SPE sampling
TLB misses
MMU Heat Map
#TLB misses
Address Space
Leshy placement algorithm
generates page size hints
prioritizing MMU hotspots

Figure 3.8: Leshy tracks the MMU pressure per virtual page and uses this translation overhead heatmap of the address space to calculate translation size hints.

Asynchronous promotions. For asynchronous promotions, in contrast to khugepaged and similarly to prior art [62, 67], ET attempt to estimate which memory regions to scan, migrate and promote to larger translations. ET must also decide which translation size to use for each region. To that end, the thesis designs *Leshy*, a profiler that leverages ARMv8-A Statistical Profiling Extensions (SPE) to sample the TLB misses of running workloads. The decision to sample TLB misses instead of the per-page access frequency, as prior work does [62, 67], is based on the analysis in Section 3.2. Section 3.4 quantifies the accuracy and overhead of both methods. Leshy analyzes the TLB misses and generates a translation overhead heat-map of the address space, aggregating the misses per virtual page (Figure 3.8). Leshy then sorts regions by MMU hotness and attempts to optimally map the working set to translation sizes based on translation overhead.

ET use Leshy to periodically profile workloads *at runtime* and compute optimal translation size hints for each region of the process address space *online*. ET then load the computed hints into the kernel at runtime via an extended *madvise()* interface and use them to drive the in-kernel ET mechanisms. The hints are sorted by MMU overhead and are loaded and stored in the kernel in that order. As discussed in Section 3.3.3, for asynchronous promotions, CoalaKhugepaged will traverse the hints in sorted order, prioritizing promotions for the MMU hotspots of the address space. When offline profiling is an option [72], the hints can be computed and loaded in advance, enabling CoalaPaging to utilize them at fault time, improving upon the greedy fault-time allocation policy. ET retain the fault-time fallback policy in case of allocation failure.

Optimal size selection. In order to optimize translation size selection and generate translation size hints, Leshy needs to find a non-overlapping mapping of address space regions to translation sizes. This mapping should contain a limited number of translations, N and cover a target percentage of the sampled TLB misses. Leshy uses the TLB size (entries) for N and set the coverage target to 99.99% of the total sampled TLB misses. Additionally, the mapping should use the least contiguity-taxing combination of translation sizes that satisfy the above constraints.

To that end, Leshy aggregates the sampled addresses in bins of different sizes and for each bin i of $size_i$ Leshy calculates the total sampled TLB misses, $misses_i$, for all the addresses, $addresses_i$ belonging to it. The optimization problem is the formulated as follows:

$$\begin{aligned} \min \quad \sum_i size_i x_i \quad \text{s.t.} \quad & \sum_i misses_i x_i \geq target \\ & \sum_i x_i \leq N,\ x_i \in \{0,1\} \\ & \bigcap_i addresses_i = \emptyset \end{aligned} \tag{3.1}$$

Algorithm 1: Calculating size hints from TLB misses

```
TlbMisses = Sample(Workload, Duration)

for each Size
    AggregateMisses(Align(VA, Size), Bin[Size])
        for VA in TlbMisses
    Sort(Bin[Size])

for each Size:
    Entries = Take Entry from Bin[Size]
        while CoveredMisses(Entries) < Target
    if CoveredMisses(Entries) >= Target
        Selection = Entries
        InitialSize = Size

while CoveredMisses(Selection) >= Target
    for each Size < InitialSize
        Selection = Substitute(Selection,
            InitialSize, Size)

Sort(Selection)
Return Selection
```

To compute the translation size hints, Leshy first sorts the bins based on the total number of misses caused by each aggregated address (*entry*). Leshy then follows a best-fit approach, whereby it first calculate the minimum translation size (*initial size*) that is able to cover the target TLB misses with N or less entries. Starting from this initial selection of M entries, it retains the

$M-1$ entries with the most TLB misses and recursively attempt to substitute the discarded Mth entry with a sub-selection of smaller-sized entries that are able to match the target misses while not exceeding the configured TLB capacity N.

3.3.5 Contiguity Redistribution

Thus far, Elastic Translations only support asynchronously promoting address space regions, from smaller to larger translation granules, via CoalaKhugepaged. However, due to the transient nature of the execution profile of many applications, certain address space regions might go cold, wasting physical memory contiguity for processes and regions that no longer benefit from it.

This section presents an extension to ET and Leshy, that enables demoting cold address space regions to smaller translation granules, to ensure the fair allocation of the available physical memory contiguity in the system to processes and address space regions at all times. By supporting asynchronous demotions, ET are able to employ greedy fault-time allocations, without sacrificing memory contiguity. Instead of being frugal at fault-time and relying on delayed asynchronous promotions, ET fully utilize the available contiguity at fault-time, allowing workloads to use larger granules as early as possible and alleviating the AT overhead faster (§ 3.4). Underutilized contiguity from cold regions is then gradually released back to the system, driven by fragmentation pressure.

An overview of the proposed extended design is shown in Figure 3.9. The next subsections describe in detail how this extended design i) detects and then ii) demotes cold address space regions that are backed by larger translation granules.

3.3.5.1 Detecting cold regions

Similarly to the original ET design, Leshy starts by sampling the TLB misses of the profiled application, using the ARMv8-A Statistical Profiling Extension (SPE). As described in Section 3.3.4, Leshy controls the sampling frequency to bound the CPU overhead of sampling. The profiler then aggregates the sampled TLB misses at a configurable aggregation interval N and computes an exponential moving average (EMA) over an also configurable number of epochs E. If the sampled TLB misses EMA exceeds a pre-configured high threshold T_{high}, Leshy remains in TLB sampling mode and proceeds as previously described in Section 3.3.4 to compute promotion hints for address space regions based on the sampled TLB misses.

Conversely, when the EMA falls below a low threshold T_{low}, Leshy switches to contiguity redistribution. As the profiled application no longer generates MMU pressure, instead of profiling TLB misses to guide promotions, Leshy now attempts to detect cold address space regions that use larger translation granules, in order to mark them as demotion candidates and make their reserved contiguity available to the system. The process of releasing contiguity back to the

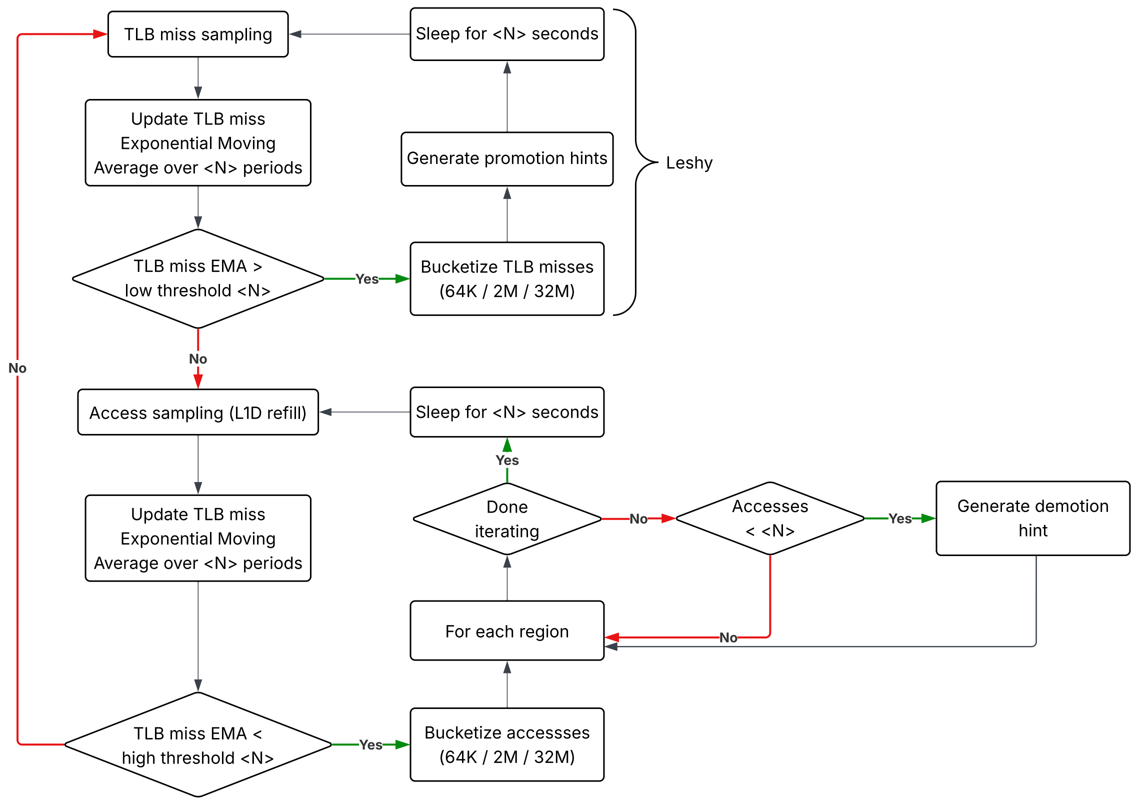

TLB miss sampling
Sleep for <N> seconds
Update TLB miss Exponential Moving Average over <N> periods
Generate promotion hints
Leshy
TLB miss EMA > low threshold <N>
Yes
Bucketize TLB misses (64K / 2M / 32M)
No
No
Access sampling (L1D refill)
Sleep for <N> seconds
Yes
Update TLB miss Exponential Moving Average over <N> periods
Done iterating
No
Accesses < <N>
Yes
Generate demotion hint
For each region
No
TLB miss EMA < high threshold <N>
Yes
Bucketize accessses (64K / 2M / 32M)

Figure 3.9: Extending ET to support demotions.

system is described in detail in the next subsection.

In order to detect cold regions, Leshy can no longer use TLB miss sampling (§ 3.3.4), as it does not allow for distinguishing between MMU-hot regions, which no longer generate TLB pressure due to the larger translation granule, and cold regions that are only rarely or never accessed. Leshy instead samples memory accesses to derive access hotness for address space regions and drive demotions. To reduce the sampling load, instead of sampling all memory accesses for the profiled application, ET use the capability of SPE to filter for *L1D refill* events, i.e., memory accesses that miss in the L1 cache, to derive an estimation of access hotness for address space regions. Newer versions of the ARMv8-A architecture (ARMv8.8+), also support filtering for accesses that miss higher up the cache hierarchy [119, 120], e.g., in the Last Level Cache (LLC), similarly to Memtis [121]. Incorporating and evaluating LLC miss sampling for access hotness tracking with Leshy is considered for future work.

Note that while Leshy samples memory accesses, it also tracks TLB misses generated by those accesses. It is thus able to detect changes in the execution profile of running applications and switch back to TLB miss sampling and promotions, if it detects a spike in the sampled TLB misses.

3.3.5.2 Demoting cold regions

Similarly to the original Leshy design, on each aggregation interval, the sampled accesses are grouped into per-granule buckets, i.e., they get grouped in to 64KiB, 2MiB and 32MiB bins. To generate demotion hints, Leshy starts from the largest bin, i.e., 32MiB, and selects the regions whose sampled accesses fall below a configurable threshold. For these cold regions, it then checks whether that particular region is actually backed by an equally-sized granule. If both conditions are true, Leshy marks the region for demotion.

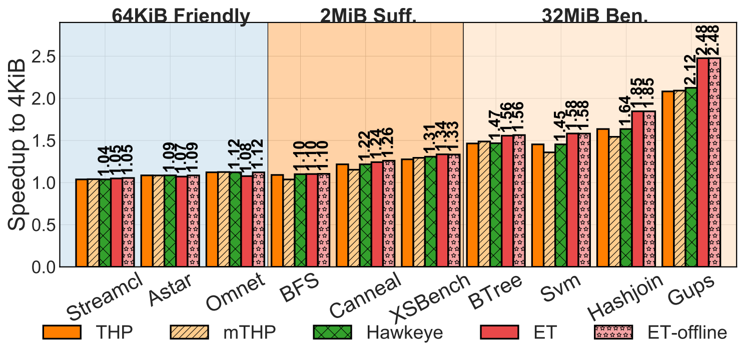
To mark the region for demotion, ET extend the madvise()-based interface, described in Section 3.3.4, by adding an additional advise hint, MADV_DEMOTE. In the kernel, ET handle the MADV_DEMOTE hint by marking the corresponding regions as candidates for demotion. The initial implementation, that is used and evaluated in the thesis, employs synchronous and greedy demotions to release the reserved contiguity of the cold regions back to the system. Each region is demoted synchronously during the madvise() system call, and the demotion involves breaking down the region to the smallest available translation granule, i.e., 4KiB. However, as demotions necessarily involve page migrations, this incurs CPU overhead, even when there is no fragmentation pressure in the system. Additionally, to that end, the thesis envisions, for future work, an asynchronous, tiered design, whereby regions are marked for demotions via madvise(), but are not immediately demoted. Instead, demotions are driven by the fragmentation pressure in the system. Similarly to CoalaKhugepaged, another kernel thread, kdemoted, periodically scans the demotion candidates and selectively demotes regions until the contiguity in the system meets a configured threshold. To avoid overshooting and also bound the CPU overhead in the case of synchronous demotions, Leshy limits the number of regions marked for demotion in each aggregation interval to a configurable number.

3.4 Evaluation

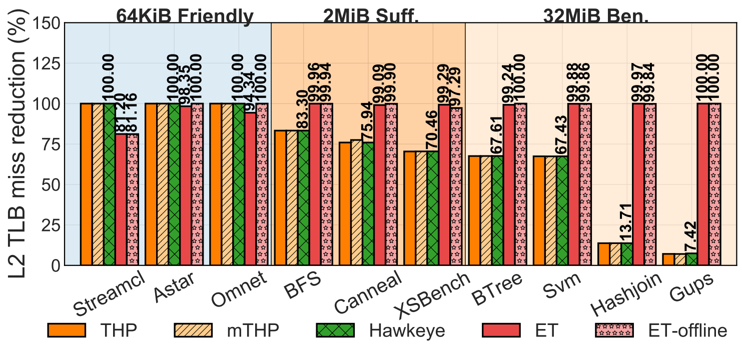
3.4.1 Methodology

Workload	Description	Footprint
astar	A* pathfinding algorithm [122]	400MiB
omnetpp	Network Simulator [122]	150MiB
streamcluster	Online Clustering [123]	100MiB
BFS	GAPBS [124] BFS on the Friendster [125] graph	88GiB
canneal	Chip Routing [123]	14GiB
XSBench	Monte Carlo Cross Section Lookup [126]	122GiB
SVM	Support Vector Machine library [127, 128]	39GiB
BTree	Lookups in a BTree [63]	33 GiB
hashjoin	Hashjoin microbenchmark	70GiB
GUPS	HPCC random updates benchmark [129]	32 GiB

Table 3.3: Evaluation Workloads.



(a) Speedup



(b) TLB miss reduction

Figure 3.10: Elastic Translations (ET) performance on a non-fragmented node for native execution

Experimental Setup. The thesis implements ET for Linux v5.18 and evaluate it on Ubuntu 22.04 for both native and virtualized execution. For virtualized execution, it uses KVM and Qemu v7. For the evaluation, it uses an Ampere Altra server [130, 131], with 2 nodes of 80 ARMv8-2A+ Neoverse N1 cores [132], each with 256GiB of memory. The MMU includes separate data and instruction fully-associative L1 TLBs of 48 entries each, and a unified 5-way set-associative L2 TLB of 1280 entries of any size. L1 misses cost ∼3 cycles and L2 misses over 15 cycles. To minimize jitter, a single NUMA node is used, each thread is pinned on a single core and the core frequency is set to 2.7GHz. GNU libc's malloc is replaced with gperftools tcmalloc [133], similar to [6, 56, 103, 108].

Performance Metrics. The evaluation uses end-to-end execution cycles and L2 TLB misses, reported by the HW performance counters of ARMv8 PMUv3 [41] as the main evaluation metrics. To quantify the ET effect on fault latency, it uses the Linux tracing subsystem to instrument the kernel fault handling path.

Fragmentation. The fragmentation scenarios allocate all of the node memory and then release small chunks at the start of each 2MiB page, similarly to [62, 67, 71, 72]. For each workload, they release memory until i) the free memory in the system equals the footprint of the workload and

ii) the Free Memory Fragmentation Index (FMFI) [134] for 2MiB (order-9) pages equals a defined threshold, reported as a percentage $X\%$. Without asynchronous promotions, the workload would run with $X\%$ of its footprint backed by 2MiB pages.

Workloads. The evaluation uses applications that exhibit varying TLB sensitivity to evaluate the behavior and effectiveness of ET. It includes workloads with large footprints and varying degrees of access irregularity. These workloads are typically backed by 2MiB pages and some can push 2MiB pages to their limit in terms of effectiveness. The thesis also evaluates workloads with smaller footprints but highly irregular access patterns. Table 3.3 provides a description of the evaluation workloads.

Evaluation scenarios. The evaluation uses the 4KiB performance of Linux as the baseline. It compares ET with Linux THP and mTHP. As discussed in Section 3.2 (Table 3.2), mTHP enables 64KiB translations through faults, as a fallback to 2MiB allocations. The thesis also ports Hawk-Eye [62] to Linux v5.18 and ARMv8-A and compare it with ET. For mTHP, it uses Linux v6.8 and also reports the 4KiB performance of Linux v6.8 for reference. To understand the effect of run-time sampling and hint generation versus offline profiling, it uses Leshy to sample workloads and generate translation-size profiles in advance, which it then load into the kernel when the workload is spawned (*ET-offline*). Finally, it compare the ET fault latency to 4KiB, 64KiB, 2MiB and 32MiB synchronous faults. As 32MiB faults are not transparently supported by (m)THP, the evaluation uses a kernel built with a 16KiB base page size (granule) [41], which increases the THP large page size to 32MiB.

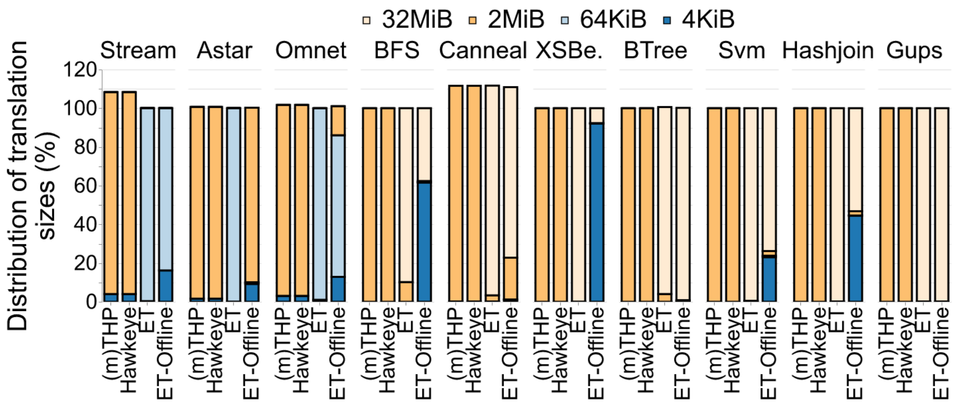


Figure 3.11: Distribution of translation sizes

3.4.2 Native Execution

The workloads are first executed and evaluated natively on a freshly booted machine. Figure 3.10 summarizes the evaluation results for (a) execution time speedup and (b) TLB miss reduction. Figure 3.11 shows the corresponding distribution of translation sizes for each method. It presents a single bar for both THP and mTHP as their distributions are almost identical. Since the memory is not fragmented, asynchronous promotions are rare, which allows isolating the effect of fault-time *allocation policies*. ET use the size of the faulting VMA to guide translation size selection, while ET-offline uses the Leshy generated hints (Section 3.3.4). Based on the performance and translation size distribution, three groups of workloads can be discerned:

64KiB-friendly workloads. For workloads with small footprints, i.e., Astar, Omnetpp, Streamcluster, ET use CoalaPaging to opportunistically map them with 64KiB translations, via coalescing-aware 4KiB allocations at fault time (Figure 3.11). This significantly reduces TLB misses (Figure 3.10b) and the overall performance is close to (m)THP (Figure 3.10a). The results are in line with the initial motivational analysis (Section 3.2.3) and show that CoalaPaging is able to successfully generate 64KiB translations across 4KiB faults. mTHP does not leverage 64KiB faults as 2MiB allocations always succeed.

2MiB-sufficient workloads. For Canneal, XSBench and BFS, ET utilize coalescing-aware 2MiB faults to eventually map their footprint with 32MiB translations (Figure 3.11). This results in a 16-30% reduction in TLB misses compared to (m)THP, but translates to only minor execution speedups up to 3-4%. 2MiB translations are sufficient for these workloads.

32MiB-beneficiary workloads. For the highly irregular workloads, BTree, SVM, Hashjoin and Gups, ET eliminate TLB misses, using 32MiB translations to cover 97-99% of their footprint. This boosts performance by 19% on average and up to 39% versus THP. These results match the motivational analysis (Section 3.2.3) and demonstrate that ET effectively and transparently support all translations sizes. No other design supports 32MiB translations.

For the larger workloads, mTHP appears to perform slightly worse than THP. This is only due to a slightly worse baseline performance (4KiB) of Linux v6.8 (2-3%) and not due to reduced address translation performance (Figure3.10b). HawkEye has identical performance to (m)THP as it always uses 2MiB faults [72] and its async prezeroing has negligible impact.

MMU hotspots. To further study the performance potential of multiple translation sizes, Leshy is run *offline* for all workloads and load the computed translation size hints into the kernel when each workload is spawned. This way CoalaPaging allocations are no longer eager; they are instead guided (Section 3.3.4). Figure 3.11 shows that TLB misses are frequently localized to specific address space regions (Section 3.2.4). ET-Offline is able to detect these hotspots and map only them with larger translation sizes. For example, for XSBench, Svm, BFS and Hashjoin, it uses 4KiB pages for 93%, 34%, 64% and 45% of their address space, mapping the rest with a combination of 2MiB and 32MiB translations. This significantly reduces the usage of larger translations while sustaining performance (Figures 3.10a, 3.10b). For Canneal and BTree, Leshy uses a combination of 2MiB and 32MiB translations for their entire footprint. For Omnetpp and Astar, it uses a combination of 64KiB and 2MiB translation sizes to minimize MMU overheads, while for Streamcluster it exclusively uses 64KiB. For Astar and Omnetpp, Leshy overestimates the impor-

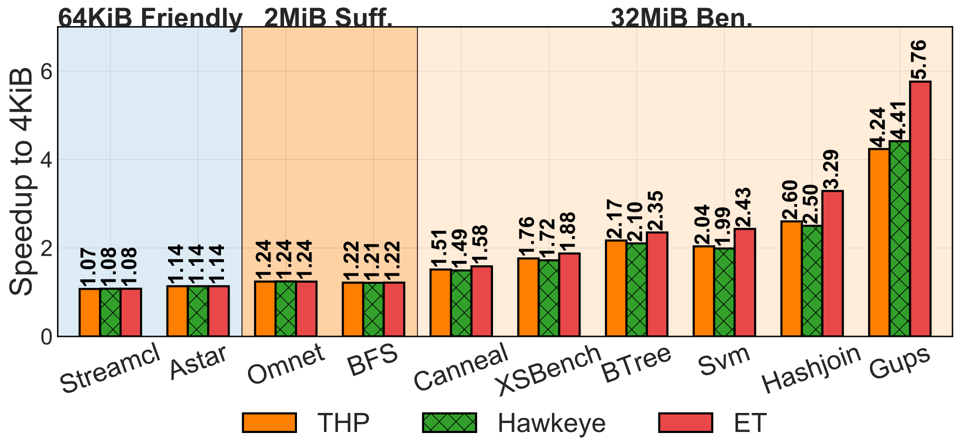


Figure 3.12: ET performance in virtualized execution

tance of some TLB misses, which results in ET-Offline using larger translations compared to ET, with only minor improvements in TLB miss reduction and overall performance. These results lay the ground for the online guided asynchronous promotions discussed later.

Takeaway 1: *One size does not fit all.* ET successfully generate 64KiB and 32MiB translations across faults, relaxing the need for 2MiB pages and improving performance by up to 39% over THP.

3.4.3 Virtualized Execution

The thesis also evaluates ET in virtualized execution. Figure 3.12 presents the results without fragmentation. It omits the results for mTHP as it doesn't support virtualized execution. The costly nested page walks magnify AT overheads, necessitating larger translation sizes. Omnetpp, which was covered by 64KiB translations in native execution, requires some 2MiB pages to sustain performance in virtualized environment. Similarly, 2MiB pages are no longer sufficient for Canneal and XSBench, which now require 32MiB translations. Despite its opportunistic nature (Section 3.3), CoalaPaging manages to effectively generate contiguous 64KiB and 32MiB translations in both guest and host. This translates to significant speedups for big-memory workloads, 30% on average and up to 150% over THP. HawkEye performs slightly worse than THP as there is no fragmentation, thus both systems eagerly allocate 2MiB pages at fault time [72], while HawkEye scanning and pre-zeroing are costlier in a 2D set-up.

Takeaway 2: ET successfully enable intermediate translation sizes in virtualized execution. The costlier page walks magnify ET benefits, speeding-up execution by 30% on average and up to 150% over THP for large workloads.

3.4.4 External fragmentation

Figure 3.13 presents the results for two fragmentation scenarios, 50% and 99% (Section 3.4.1) for native execution. For smaller workloads it was challenging to consistently generate fragmentation, due to their small footprints, so their results are omitted. As the fragmentation increases, all methods increasingly rely on asynchronous migrations to generate large translations. This allows us to evaluate the effect of *asynchronous promotion policies*. ET asynchronous promotions are guided by Leshy translation size hints, which are generated *online* by periodically sampling the TLB misses of each running workload. The figure also shows results for ET-offline, where ET asynchronous promotions are guided by optimal hints pre-calculated by Leshy *offline*. The fault allocation policy remains unchanged in both cases, unlike the previous section where offline hints were also used by ET during faults. As expected, increased fragmentation negatively impacts performance for all methods. However, ET outperform or at least matches state-of-practice and state-of-the-art.

2MiB-Sufficient. For Canneal, all methods perform almost equally, as the workload runs long enough for all methods to promote the entire address space to large pages. For BFS, ET improve performance by 6% over both THP and HawkEye and for XSBench by 20% and 4% respectively. The reason is two-fold; a) Leshy successfully identifies at runtime the MMU hotspots and prioritizes their promotion and b) ET leverage 64KiB and 32MiB translations, which albeit unnecessary without fragmentation, are beneficial for the workloads when memory is fragmented. Consequently, ET manage to sustain higher performance while reducing large page usage by 50% on average compared to THP. HawkEye effectively detects the MMU hotspots for XSBench, but ET's higher resolution achieves slightly better performance while reducing 2MiB usage by 20%. mTHP falls back to 64KiB translations at fault time, which improves performance by $\sim$2% for some workloads. However, mTHP-khugepaged always promotes the formed 64KiB translations to 2MiB, without considering performance impact. These results underline that, while 64KiB contiguity can be utilized when 2MiB pages become scarce due to external fragmentation, efficiently taking advantage of them requires informed promotion policies.

32MiB-beneficiary. For BTree, SVM and Hashjoin, ET outperform both state-of-practice and state-of-the-art; speeding-up performance by 12% over (m)THP and 17% over HawkEye on average when memory is 99% fragmented. Hashjoin and SVM have MMU hotspots at the tail of their address spaces, rendering THP linear promotion scanning ineffective. By contrast, Leshy successfully detects these hotspots at runtime and prioritizes their promotion to 32MiB, improving performance by 16% and 14%. HawkEye is unable to detect these hotspots as accurately and only promotes few regions to 2MiB. At 99% fragmentation, Hawkeye performs worse than (m)THP for the BTree workload, likely due to contention in its internal data structures, identified also by related work [63].

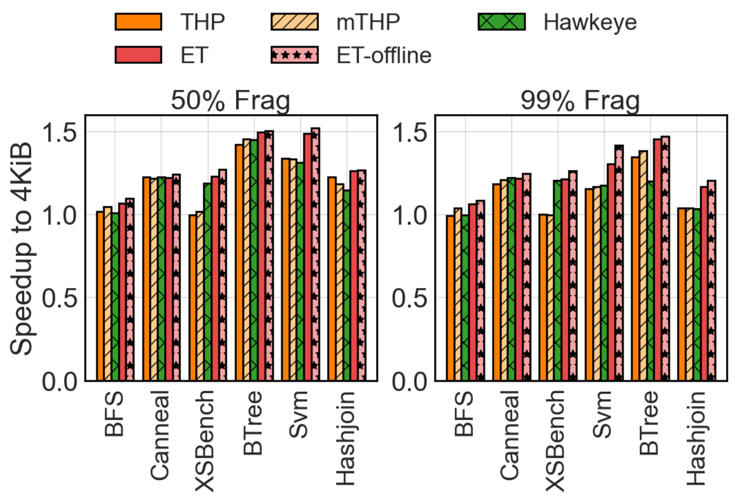


Figure 3.13: ET native performance under fragmentation.

Online vs Offline. Figure 3.13 reveals that HW-assisted TLB miss sampling is able to guide asynchronous promotions at runtime accurately. In most cases, online profiling and hint generation (*ET*) is able to achieve comparable results to hints computed offline (*ET-offline*), resulting in similar translation size distributions. For SVM, the gap between offline and online performance under 99% fragmentation is attributed to the differences between a pro-active (offline) and a re-active (online) method. SVM exhibits a long initialization period with negligible MMU overheads and abruptly switches to the MMU intensive part of its execution. With pre-computed hints, ET-offline is able to start the migrations earlier and by the time SVM enters its second compute-intensive phase, a large part of its address space is already optimally mapped. ET's online profiling, on the other hand, triggers promotions only after SVM starts experiencing MMU overheads, which the profiler detects at runtime. Although longer running workloads might be able to amortize this cost, this spool-up effect also underlines the usefulness of offline profiling when possible.

Takeaway 3: ET accurately detect MMU hotspots at runtime and prioritizes their optimal mapping to an educated mix of translation sizes, when running under fragmentation. This improves performance by 12% on average and up to 20% while reducing 2MiB occupancy by 30% on average.

3.4.5 Performance analysis

Figure 3.14 presents a breakdown of the impact of the various ET components (Table 3.1) for native execution and increasing fragmentation levels. The speedup provided by each component,

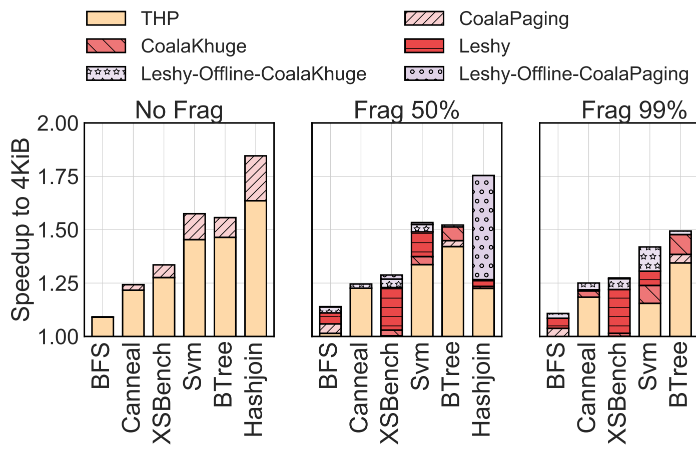


Figure 3.14: Performance breakdown of ET components

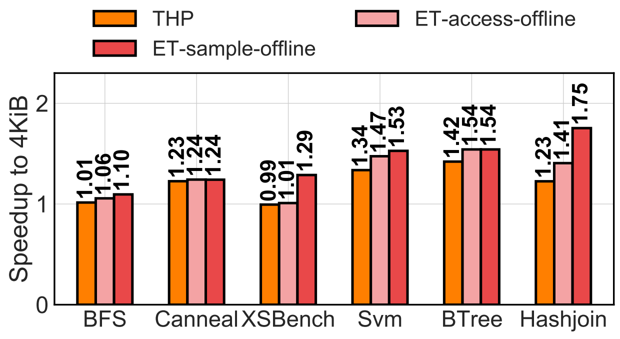


Figure 3.15: TLB miss sampling vs access-bit monitoring accuracy

relative to 4KiB, is stacked on top of each other, starting with vanilla THP. ET comprise a) CoalaPaging that transparently generates 64KiB and 32MiB translations across faults, b) CoalaKhugepaged, that asynchronously promotes regions to 32MiB translations, and c) Leshy that detects MMU hotspots at runtime via TLB miss sampling, computes translation size hints and drives CoalaKhugepaged. The figure also presents the benefit provided by pre-computed (offline) Leshy profiles, when they drive a) CoalaKhugepaged promotions from the beginning of a workload's execution and b) CoalaPaging fault-time allocations.

The impact of each component depends on fragmentation level and workload behavior. Under low fragmentation pressure, ET benefits are mostly driven by CoalaPaging. As fragmentation increases, CoalaPaging impact diminishes, with the exception of BFS. BFS exhibits a small MMU-intensive region at the beginning of its address space. CoalaPaging is able to map it to 64KiB translations and alleviate translation overheads, even when 2MiB pages are scarce. For the rest, CoalaKhugepaged and Leshy dominate performance gains as fragmentation increases. For BTree, where the distribution of TLB misses is relatively uniform throughout its address

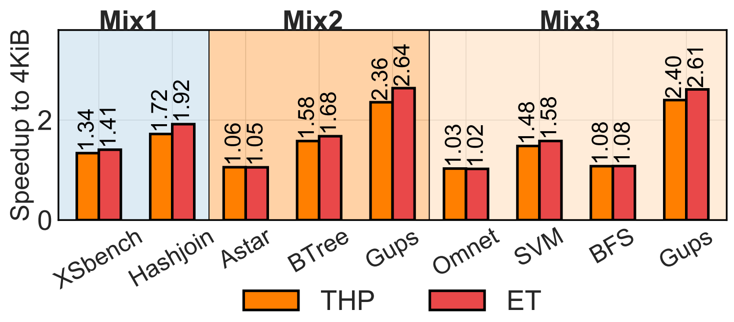


Figure 3.16: ET performance for multi-workload mixes.

space, CoalaKhugepaged's aggressive promotions to 32MiB translations, via linear scanning the workload's address space, are sufficient to alleviate the AT overhead. By contrast, TLB misses for XSBench, Hashjoin and SVM are clustered in small regions at the tail of their address space. For these workloads, ET performance gains stem from Leshy, as it is able to accurately detect these TLB-heavy clusters at runtime and guide CoalaKhugepaged promotions. Pre-computed (offline) Leshy profiles are mainly beneficial to Hashjoin and SVM, albeit for slightly different reasons. Hashjoin benefits from informed CoalaPaging faults when fragmentation is mild as, due to its short runtime, CoalaKhugepaged is unable to cover the MMU-intensive parts of its footprint in time. SVM on the other hand benefits from the fact that with pre-computed translation hints, CoalaKhugepaged asynchronous promotions are able to begin early, before the workload enters its MMU-intensive phase.

TLB miss vs access-bit sampling. The thesis also evaluates the use of access-bit sampling to generate offline translation size hints via Leshy. It uses hints to guide both CoalaPaging fault-time allocations and CoalaKhugepaged asynchronous promotions (similarly to ET-offline in Figures 3.14 and 3.10). Figure 3.15 shows that for 50% fragmentation translation size hints generated by Leshy based on sampled TLB misses exhibit higher accuracy. This corroborates the findings from Section 3.2.4 regarding the relative effectiveness of TLB miss sampling and to some extent explain why ET outperform HawkEye even for 2MiB-sufficient workloads (Section 3.4.4).

3.4.6 Multi-workload experiments

Figure 3.16 presents the results for THP and ET when natively running mixes of workloads concurrently without fragmentation. Three different mixes of workloads are run and the figure plots the speedup achieved for each workload by THP and ET relative to 4KiB. ET are able to sustain its performance benefit over THP (cf. Figure 3.10) in multi-programmed execution due to the way CoalaPaging coordinates concurrent allocation requests from different programs (Section 3.3.2).

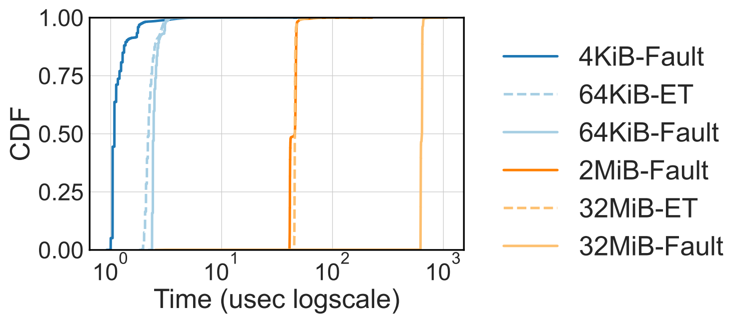


Figure 3.17: Fault latency CDF

3.4.7 Overhead analysis

Fault latency. Figure 3.17 reports the cumulative distribution function (CDF) for the latency of CoalaPaging faults (64KiB and 32MiB), as well as 4KiB, 64KiB (mTHP), 2MiB (THP) and 32MiB (THP-16KiB granule) synchronous faults. To generate the CDF, a micro-benchmark that triggers 100K random anonymous faults is run and the latency of the fault handler is collected for each run. 64KiB ET faults exhibit increased fault latency compared to 4KiB. Linux has an extremely fast path for allocating 4KiB pages (∼1us), utilizing lockless per-CPU page lists (Section 2.1.1). 64KiB ET faults are slightly faster than mTHP's 64KiB faults, as the increased fault size incurs overhead, e.g., synchronous zeroing. On the other hand, 32MiB ET faults perform closely to THP and are an order of magnitude faster than synchronous 32MiB faults, since synchronous 32MiB faults have to zero large blocks of memory, while ET relies on smaller fault-time allocations (2MiB). These results support the design choice to opportunistically allocate contiguous pages across faults and underline its benefits versus an alternative design which relies on larger fault-time allocations [135].

Memory Bloat. In Figure 3.11, the normalized page distribution for canneal and streamcluster exceeds 100% for THP and ET. The reason is that 2MiB pages can increase the effective memory footprint of workloads [62, 67, 71] compared to 4KiB. 32MiB ET translations do not induce extra memory bloat compared to THP, owing to the opportunistic coalescing-aware allocation policy. For streamcluster ET favor 64KiB translations over 2MiB, which reduces memory bloat.

Takeaway 4: The opportunistic design of CoalaPaging keeps fault latency and memory bloat bounded while supporting translation sizes beyond 2MiB.

3.4.8 Contiguity Redistribution

To showcase the effectiveness of the extended ET design to detect and demote cold regions, the evaluation focuses on the hashjoin benchmark, one of the most MMU intensive benchmarks used

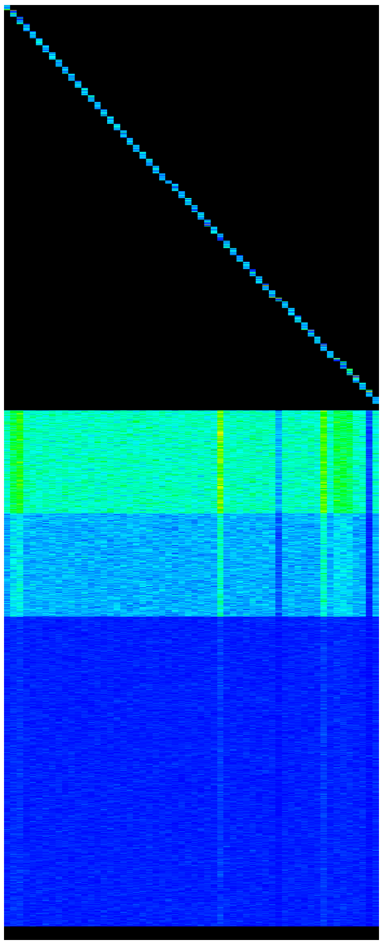


Figure 3.18: Access heatmap for the hashjoin benchmark.

in the original evaluation of ET (see Sections 3.4.1 and 3.4).

The memory accesses performed by hashjoin using the ARM SPE are first sampled, similarly to Leshy. A heatmap of the memory accesses is then generated, shown in Figure 3.18. The heatmap plots the memory access intensity per 32MiB region of the workload's heap (y axis) over time (x axis). The memory access intensity per 32MiB region is assigned a color, ranging from blue (low access intensity) to red (high access intensity).

From the heatmap, it becomes evident that while the middle part of the heap is actively and consistently accessed throughout the workload's lifetime, the beginning of the heap exhibits a more fragmented access pattern. Few gigabytes of memory are accessed for a period of time and then go cold. As discussed previously, a greedy fault time policy would allow these accesses to benefit from larger translation granules and page sizes. However, the contiguity would quickly become wasted, as these regions of memory are never touched again.

Two hashjoin benchmarks are then run, one after the other, in quick succession. Before

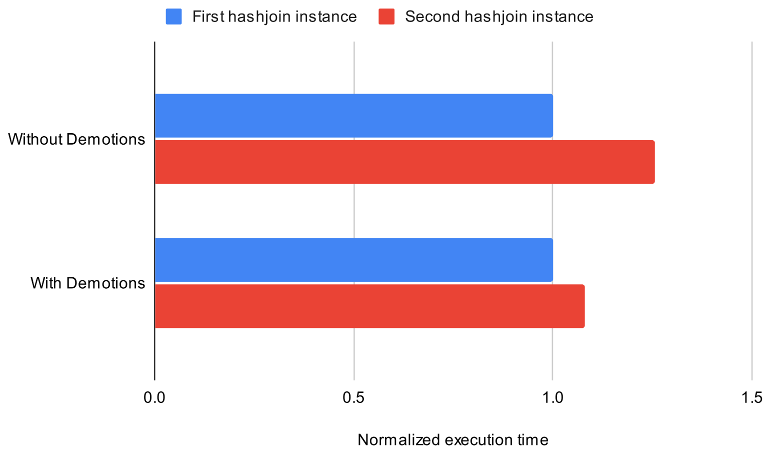


Figure 3.19: Normalized execution runtime, when running two hashjoin instances, on a fragmented system. The extended ET design, with demotions support, is able to successfully detect and demote cold regions, redistributing the limited memory contiguity in the system to the second workload, thereby sustaining performance.

running the benchmarks, the fragmentation tool described in Section 3.4.1 is used to reduce the available contiguity in the system, so that there is enough contiguity only for the first hashjoin instance to be backed by 32MiB translations, but not for the second.

The first hashjoin instance is spawned, let run for a period of time post initialization, and then the second hashjoin instance is spawned. For both instances, ET employ a greedy fault time policy, using ET and CoalaPaging to opportunistically map their memory with 32MiB translations. As described above, the first instance will be able to fully back its memory with 32MiB translations, as there is enough contiguity in the system. However, when using the vanilla ET (Section 3.3), the second benchmark faces fragmentation pressure and is unable to fully back its memory with 32MiB translation, which has significant performance impact, as shown in Figure 3.19.

This is not the case when using the extended ET design. While the first instance is running, Leshy is able to detect the low parts of the workload's heap that are accessed only briefly and then go cold. Leshy then proceeds with relinquishing the reserved contiguity by these regions back to the system, as shown in Figure 3.20. As a result, when the second hashjoin instance runs, it is able to find the required contiguity and back its memory with 32MiB translations. Figure 3.21 shows the number for 32MiB translations used by each benchmark, measured while the second instance has finished initialization and started its compute phase. Leshy is able to successfully detect the cold regions of the first hashjoin instance and redistribute the contiguity reserved by these cold regions to the second hashjoin instance. Figure 3.19 show the performance impact of this redistribution. The extended ET design, with demotions support, sustains performance, close to

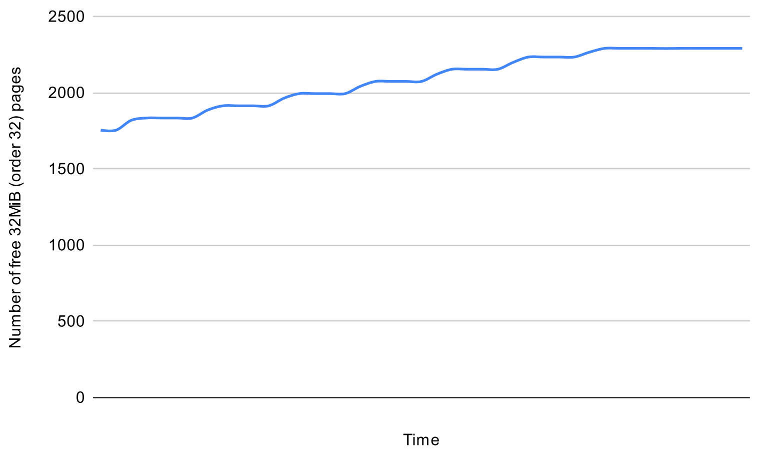


Figure 3.20: 32MiB (order 13) pages in the buddy allocator free lists over time as cold regions get demoted

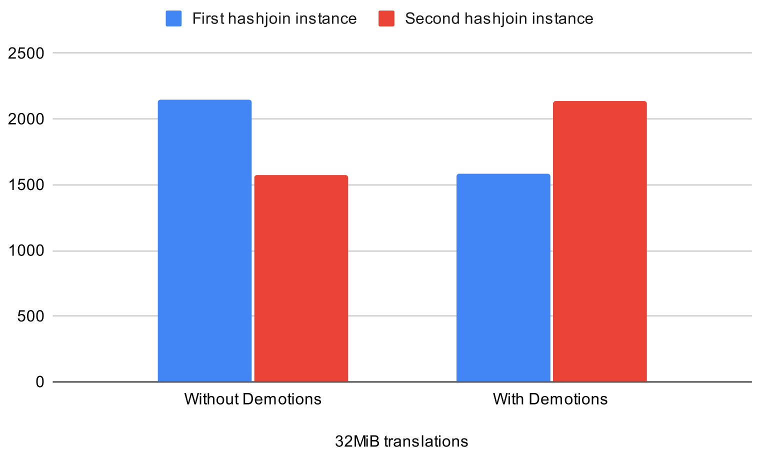


Figure 3.21: 32MiB translations allocated by the first and the second hashjoin benchmark. Leshy enables the redistributing of contiguity between the two hashjoin instances.

the original hashjoin instance. The evaluation also measures the performance of the first hashjoin instance, when demotions are enabled, to quantify the performance impact of demotions on the running application. Our results show that the demotions have minimal impact on performance, as expected, as the demoted regions are no longer accessed. The thesis leaves for future work the exploration of the overall system impact of demotions (e.g., due to memory copies), as well as the performance impact of demotion when workloads exhibit access patterns, where regions go cold and then become hot again.

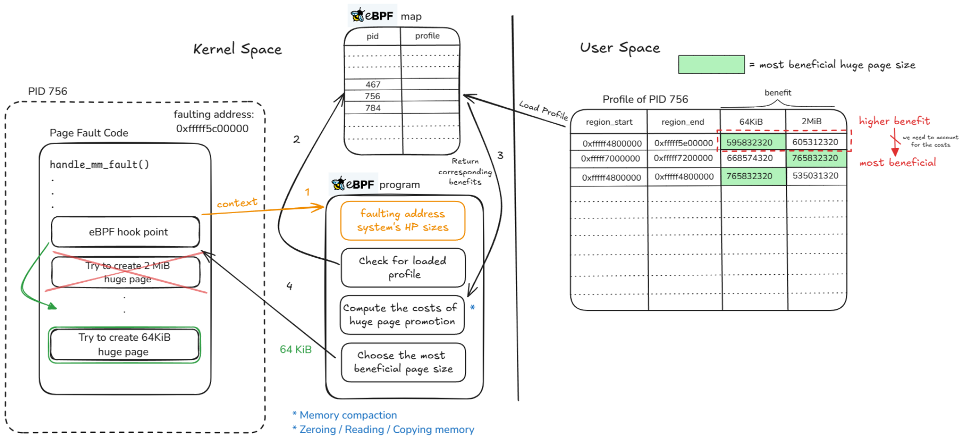
3.5 Discussion

3.5.1 OS interfaces for controlling translation granularity

madvise() interface. One question that arose when designing Elastic Translations concerned the interfaces that the OS should expose to userspace, to allow userspace applications fine-grained control over translation granularity.

As discussed in Sections 3.3.4 and 3.3.5, Elastic Translations employ an interface based on the madvise() system call [136], and specifically the process_madvise() variant [137], that was added to the Linux kernel starting with version v5.10. madvise() is already used by the Linux kernel to allow userspace to control THP policy per VMA, via the MADV_HUGE advice. madvise() applies the advice to the calling process, which makes it less useful for profilers, such as Leshy, which need to control and issue hints on behalf of the profiled processes. By contrast, process_madvise() enables the issuing of madvise() hints for other processes, and is hence used by Leshy to issue translation granularity hints to the kernel and drive the ET kernel components, namely CoalaPaging and CoalaKhugepaged.

The ET madvise() interface defines translation size advice hints, one for each size above 4KiB, as well as advices that control the in-kernel ET mechanisms, CoalaPaging and CoalaKhugepaged, per VMA. Finally, as described in Section 3.3.5, ET define one more advice for controlling demotions.


Kernel Space
eBPF map
pid
profile
467
756
784
User Space
= most beneficial huge page size
PID 756
faulting address: 0xfffff5c00000
Page Fault Code
handle_mm_fault()
eBPF hook point
~~Try to create 2 MiB huge page~~
Try to create 64KiB huge page
context
1
2
3
4
Load Profile
Return corresponding benefits
eBPF program
faulting address system's HP sizes
Check for loaded profile
Compute the costs of huge page promotion
*
Choose the most beneficial page size
64 KiB
* Memory compaction
* Zeroing / Reading / Copying memory
Profile of PID 756
benefit
higher benefit
we need to account for the costs
most beneficial

region_start	region_end	64KiB	2MiB
0xfffff4800000	0xfffff5e00000	595832320	605312320
0xfffff7000000	0xfffff7200000	668574320	765832320
0xfffff4800000	0xfffff4800000	765832320	535031320

Figure 3.22: Controlling translation granularity from userspace with eBPF.

eBPF interface. Another way to implement such an OS interface for controlling translation granularity from userspace could involve eBPF [89] (Section 2.5).

eBPF-mm [88] uses eBPF to implement a similar interface for controlling mTHP (Section 3.2.1) fault-time size selection from userspace. Figure 3.22 shows an overview of the proposed design.

eBPF-mm employs offline profiling, based on either Leshy or DAMON [138], to generate an offline cost-benefit profile per VMA for each supported mTHP translation granule. It then loads this profile, via an eBPF map, in the kernel and uses eBPF to hook the mTHP fault path, specifically the point where the kernel decides on which mTHP size to use to serve the fault. Instead of following the simplistic fallback-based approach of mTHP size selection (Section 3.2.4, Table 3.2), eBPF-mm checks the loaded profile for the VMA that the faulting address belongs to, to find the estimated benefit for each translation size. Inspired by CBMM [72], it then generates a cost-benefit profile for each size, and finally decides on the most beneficial mTHP size that should be used to serve the fault.

A similar approach can be adopted for Elastic Translations, to replace the madvise() interface. eBPF will enable Elastic Translations to more easily support more translation granules and policies, without having to heavily modify the in-kernel components. The decision on which size to use, either for synchronous fault-time allocations with CoalaPaging, or asynchronous promotions or demotions via migrations with CoalaKhugepaged, could be computed via a flexible, userspace-programmable cost-benefit approach, implemented via eBPF, instead of the static hint-based approach of the madvise() interface.

3.5.2 Memory Management

3.5.2.1 Allocation policies

Another approach to generate the contiguity required for intermediate-sized translations is to eagerly allocate 64KiB (order-4) and 32MiB (order-13) pages during faults. As discussed in Section 3.2.1, Linux recently added support for sub-2MiB faults [60] (mTHP). The mTHP evaluation in Section 3.4 shows that, for 64KiB faults, the fault latency remains bounded. However, it also shows that extending this design to 32MiB faults results in inflated fault latency. By contrast, CoalaPaging can seamlessly and efficiently support both 64KiB and 32MiB translations. The thesis considers integrating mTHP to ET, as an alternative mechanism for generating *sub-2MiB* contiguity at fault time, for future work. Async pre-zeroing [62, 63, 72] can also be used to reduce fault latency for larger fault-time allocations; however, it comes with non-negligible CPU overhead. CoalaPaging can be nonetheless seamlessly integrated and take advantage of async pre-zeroing for faster 2MiB faults.

Reservation-based schemes [29, 65, 71, 139] could be used instead of eager allocations in order to reserve larger blocks of memory at fault-time without penalizing fault latency. Similarly to opportunistic designs [103], such as CoalaPaging, reservations trade-off the reduced fault latency with delayed creation of larger translations [71]. Compared to opportunistic designs, reservations opt for stronger guarantees for across-fault contiguity, which however incurs book-keeping overhead and increases memory bloat [71].

3.5.2.2 Transparent 1GiB support

ET focuses on the transparent support for intermediate translation sizes supported by OS-assisted TLB coalescing. The thesis considers extending i) CoalaPaging to opportunistically create 1GiB mappings and ii) Leshy to take into account 1GiB translations and emit 1GiB hints as future work. That said, as shown in Section 3.2, for a range of applications the ET-enabled 32MiB translations are sufficient to alleviate MMU overheads without resorting to the harder to allocate and manage 1GiB pages.

3.5.2.3 Hints in virtualized execution

Using TLB miss sampling to generate hints for virtualized workloads is challenging [140], as sampled VAs are not readily usable by the hypervisor. ET uses it only in the guest and fallback to access bit tracking in the host as a proxy for the MMU overhead of the VM ([62, 67]). The thesis considers exploring a paravirtualized interface [141] to allow virtualized workloads to take full advantage of the Leshy-generated hints as future work.

3.5.3 Architectural considerations

3.5.3.1 TLB micro-architecture.

The micro-architecture of the N1 ARMv8-A core features unified TLBs with regard to translation size. Every TLB entry can be use to store translations of any of the supported sizes. For split TLBs, translation size selection will need to take the different capacities into consideration [142]. Moreover, as discussed in Section 3.3, demoting coalesced translations requires invalidating and flushing the constituent pages. ARMv8-A supports HW-based invalidations for maintaining TLB coherence. Additionally, ARMv8-A has recently added support for range-based HW TLB flushes and invalidations, which should further accelerate TLB coherence. This is in contrast to x86, which handles TLB invalidations in SW with costly interprocessor interrupts. For the latter case, the cost and frequency of TLB shootdowns should be factored in, potentially forgoing promotions if their benefit would not amortize the aforementioned costs [72].

3.5.3.2 Portability

While ET focus on ARMv8-A, ET can be extended to different architectures and translation sizes. The Svnapot extension [39] adds support for OS-assisted TLB coalescing to RISC-V. RISC-V allocates more bits in the page table entries to encode the coalescing factor, hence extending the range of the supported translation sizes. Porting ET to RISC-V and evaluate ET with Svnapot is left for future work.

3.5.3.3 Access and Dirty Bits

ARMv8-A supports HW-based tracking for page accesses (*access (A) bit*) and modifications (*dirty (D) bit*). When a page of an intermediate translation is accessed or modified, the architecture allows the MMU to set the AD bit of any of the constituent pages of the intermediate translation. This has the side-effect that the OS must now check the AD bit status of all the constituent pages of an intermediate translation in order to determine the AD status of a constituent page. For anonymous mappings, that ET currently target, this can affect the performance anonymous memory reclaim (swapping). Studying this effect is left for future work.

3.6 Related Work

3.6.1 Translation sizes and large pages

[43] propose HW and OS modifications to support a wider range of translation sizes. [63] study the effectiveness of 1GiB page sizes and design mechanisms to make their transparent support practical. [143] uses a mix of 2MiB and 1GiB pages to improve translation overhead modeling. ET focus on harnessing the performance potential of the intermediate translation sizes enabled by TLB coalescing on real HW. Transparent OS large page management for the x86 architecture has been excessively studied for both Linux and FreeBSD [62, 67–72]. ET are orthogonal and complementary to these works. ET alleviates fragmentation pressure by reducing 2MiB page usage (Section 3.4), Additionally, ET 32MiB translations build upon THP fault-time allocations, and are thus able to harness the improved THP performance of prior art.

3.6.2 Memory contiguity

Previous research focuses on generating physical memory contiguity, which can be exploited by novel HW components [103, 105] or used to improve THP performance [107, 115, 141]. This thesis builds upon opportunistic allocation policies in the context of TLB coalescing.

3.6.3 Sampling-based profiling

[140] highlight the importance of TLB misses for guiding translation size selection and propose architectural extensions to accelerate scanning and promotion and assist the OS in page size selection. Per-core caches on the L2 TLB miss path track the number of misses per recently accessed 2MiB and 1GiB region. The contents of the caches are dumped to OS accessible memory at fixed intervals. Besides requiring bespoke HW, this solution is difficult to generalize for multiple translation sizes, requiring one cache per-size per-core. [121, 144–147] use sampling-based

profiling for memory deduplication and tiering. ET follow a similar approach targeting translation performance, and corroborate their findings regarding the practicality and accuracy of this approach compared to page-based access frequency sampling.

3.6.4 Address Translation Hardware

Prior works improve translation performance via HW modifications [16, 30, 106, 148–157]. SpecTLB [31] and SpOT [103] exploit predictable contiguous mappings to speedup address translation. [27, 28, 42, 158] propose and improve upon HW TLB coalescing. [117] evaluate the effectiveness of HW TLB coalescing on recent AMD processors [101, 102]. HW coalescing can be used together with OS-assisted coalescing to collectively reduce MMU pressure. The page table structure has also been extensively studied [55, 56, 104, 159]. Previous works have proposed new hashing-based schemes [15, 17, 18, 160], range tables [108, 161] as well as more radical changes [6, 7, 109, 162, 163] to the virtual memory hardware. ET retain the radix tree structure and improves performance by enabling intermediate translation sizes.

3.7 Conclusion

This chapter argues that we need to rethink how the OS interfaces with the HW mechanisms that enable different AT granules, i.e., large pages and TLB coalescing, and how it utilizes these AT granules within its memory management subsystem. It then presents Elastic Translations, an optimized, policy-driven OS-interface, to efficiently support multi-grained AT, concurrently supporting AT granules from disjoint underlying HW mechanisms, such as large pages and OS-assisted TLB coalescing.

ET extend the OS memory manager to enable the transparent and opportunistic creation of intermediate-sized translations, both at fault time with CoalaPaging, and asynchronously with CoalaKhugepaged, for both native and virtualized execution. Leshy, a HW-assisted profiler, samples the TLB misses of applications at runtime, to estimate address translation overhead and implements the ET policies for translation size selection and drives the ET in-kernel mechanisms to optimally map the application footprint to the multiple available translation sizes. By leveraging multiple translation sizes and runtime profiling, ET is able to significantly speed-up execution for memory intensive workloads when compared to state-of-practice and state-of-the-art, for both native and virtualized execution, under varying levels of fragmentation.

CHAPTER 4

AnonPTEs: Bridging the virtualization semantic gap via the MMU

4.1 Overview

In the previous chapters, we saw that hardware virtualization comes potentially at the cost of elevated address translation overhead, as the state-of-practice mechanism for supporting paged virtual memory under hardware virtualization, nested paging (Section 2.4), amplifies the memory accesses needed per translation. Additionally, the decoupling of the virtual memory subsystems between virtualized (guest) and physical (host) Operating Systems creates a semantic gap, that induces performance inefficiencies [73, 74], which will also become apparent later.

However, by trading off performance, hardware virtualization is able to provide strong isolation guarantees, which are not attainable with OS virtualization techniques [164]. Moreover, it neatly encapsulates the entire guest system state (hardware, OS, userspace) in a small number of structures, namely the virtualized CPU architectural state, the state of the virtual devices of the VM, and importantly the VM (guest) physical memory image. This facilitates VM live migration, allowing providers to seamlessly move VMs across physical hosts, as well as VM *snapshotting*, which will be the focus of this chapter, i.e., persisting (*checkpointing*) the entire VM state to storage, as a VM *snapshot*. The VM can then be *restored* from said snapshot and continue execution.

While VM snapshotting was initially employed mostly for disaster recovery and offline VM migration, it has obtained renewed significance in the context of Function-as-a-Service (FaaS) serverless computing [165–170]. FaaS users upload their code encapsulated as a function to the FaaS platform. The FaaS providers then sandbox, deploy and scale the user code on their infrastructure. One of the dominant overheads of the FaaS model stems from cold starts [76, 77], i.e., when new function sandboxes need to be spawned to handle incoming requests. As FaaS providers typically resort to hardware virtualization, to sandbox user code, for stronger tenant isolation guarantees [78], this exacerbates the cold-start overhead. To alleviate this overhead, function snapshotting, based on VM snapshotting, described above, has been proposed by academia [80, 171] and adopted by the industry [79, 172]. The snapshotted function memory, i.e., the memory of the VM sandbox after the function has been initialized and pre-warmed [173, 174], is serialized to storage as a file. This snapshot file is then memory-mapped to act as the memory of newly-spawned pre-warmed VM sandboxes, which will serve incoming function invocations.

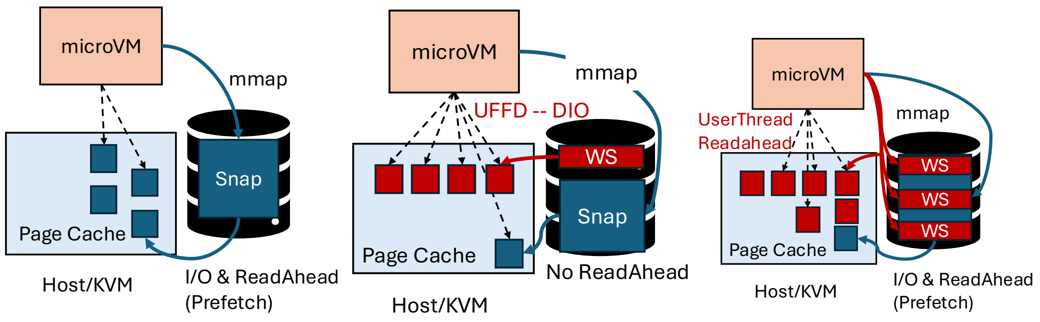
While function snapshotting obviates the need for booting a fresh VM for each cold function invocation, it still suffers from the latency of faulting-in the snapshotted VM memory from storage. Previous works [80–82] have proposed capturing and prefetching the function working set from the snapshot file in userspace to minimize this overhead. SnapBPF [83] improves upon these works by employing eBPF to move the prefetching logic in kernelspace, removing assorted redundancies.

Another issue with function snapshotting stems from the virtualization semantic gap between the virtual memory subsystems of guest and host. As the VM memory snapshot is stored on disk as a file and then memory-mapped by the Virtual Machine Monitor (VMM) as the memory of the VM, the host OS handles accesses to the entire guest physical memory by fetching pages from the file on disk, as it would for a normal file. However, not all pages persisted to the VM snapshot file contain initialized useful state. When the guest OS memory manager allocates anonymous ephemeral memory, there is no persisted state that needs to be fetched from the snapshot file on disk. To that end, FaaSnap [81] and Faast [82] keep track of these pages, when persisting the snapshot, and take care to filter them out when restoring the function.

This chapter presents AnonPTEs, a lightweight paravirtualized interface, designed to bridge this semantic gap by efficiently conveying the nature of the allocations of the VM sandbox to the host OS. AnonPTEs builds upon and extends SnapBPF, with the goal of accelerating the cold starts of microVM-sandboxed functions, when using snapshots. The guest OS uses the nested page tables of the VM to mark anonymous memory allocations via their page table entries. When the host OS handles the nested faults for these page table entries, it serves them by allocating anonymous memory instead of unnecessarily fetching the page from the snapshot. In contrast to prior art that tackles the problem by pre-scanning the snapshot, based on either page contents (zero pages) [81] or the VM kernel allocator metadata [82], in order to detect and filter such pages, AnonPTEs performs this filtering online, requiring no snapshot preparation or scanning.

Section 4.2 first provides necessary background on working set existing prefetching approaches to accelerate snapshotted function starts. Section 4.3 then discusses SnapBPF, a state-of-the-art working set prefetching approach, that employes eBPF to move prefetching to kerenlspace. Section 4.4.1 lays out the issues stemming from the virtualization semantic gap between guest and host OS virtual memory subsystems, when executing serverless functions from VM sandbox snapshots. Section 4.4 present the design of AnonPTEs and its integration with SnapBPF, while Section 4.5 show the evaluation results when using SnapBPF with AnonPTEs to accelerate snapshotted function execution. Finally, Section 4.6 discusses related work and Section 4.7 concludes the chapters and provides future directions.

4.2 Working Set Prefetching



(a) Vanilla Linux Readahead. (b) Record-and-prefetch. (c) FaaSnap.

Figure 4.1: Loading Functions from VM sandbox snapshots on Linux. State-of-practice and state-of-the-art.

Existing snapshot prefetching approaches can be broadly classified into two categories based on the mechanism they employ to capture and prefetch the function working set. Regardless of the mechanism used, the capture and loading of the working set is essentially implemented in userspace. An overview of snapshot prefetching mechanisms is shown in Figure 4.1.

Userfaultfd. REAP [80] and Faast [82] use Linux userspace page fault handling (userfaultfd) [175]. When spawning a new VM sandbox, they register a userspace page fault handler, which gets triggered on VM memory page faults. When handling such a fault, the OS allocates anonymous memory to serve the fault and then hands over the fault to userspace. The userspace fault handler subsequently fetches the faulting page from the snapshot, stored on disk, and copies (*installs*) its contents to the page allocated by the OS.

Both techniques that use userspace faults first identify the function's working set, and then serialize it to storage (**record phase**). For subsequent VM sandbox creations (**invocation phase**), they both prefetch the function working set from storage and preemptively install it in the VMM via userfaultfd. Both REAP and Faast use direct IO when fetching the snapshot from storage, to bypass the page cache and avoid the overhead of intermediate memory copies. As they both rely on userfaultfd, they fail to deduplicate the function working set across different VM sandboxes. The reason for this is that userfaultfd uses anonymous memory which is not shared between VM sandboxes of the same function, making it impossible to deduplicate the working set across different sandboxes in memory.

mincore / mmap. FaaSnap [81] on the other hand relies on the mincore() and mmap() system calls and the OS page cache for both capturing and prefetching the function's working set. The mincore system call returns a byte array which indicates whether each corresponding page of the calling process's virtual memory is resident in RAM [176]. FaaSnap uses the mincore system call to identify which snapshot pages have been fetched from storage into the OS page cache. Similarly to REAP and Faast, it serializes these pages to a separate working set file. In the **invocation phase**, FaaSnap memory-maps the working set file on top of the snapshot file. Instead of using userfaultfd, it relies on OS page cache prefetching (*readahead*), using a userspace thread to issue buffered reads to fetch the working set to memory. This enables FaaSnap to deduplicate the working set across different VM sandboxes via the OS page cache. While this allows in-memory deduplication of the working set between different sandboxes, FaaSnap has to mmap each working set region separately. To reduce the number of mmap'ed regions, FaaSnap coalesces working-set regions with few non-working set pages between them into larger regions. While this reduces the mmap'ed regions to a manageable number, it also inflates the working set file, which can affect performance by amplifying IO, which is confirmed by instrumenting the kernel using eBPF.

4.3 SnapBPF

In contrast to prior work, SnapBPF [83] employs eBPF [89] to hook the readahead mechanism of the OS page cache and both capture and prefetch the function working set in kernel-space.

Capturing the working set. To capture the function's working set, SnapBPF uses kprobes [177] to hook the Linux kernel path that adds pages to the OS page cache. Specifically, it hooks the function add_to_page_cache_lru(). kprobes allow for users to dynamically create hooks associated with kernel functions, where user-provided eBPF programs can be attached to. eBPF programs attached to such hooks are triggered whenever the associated function is executed, and are provided with an execution context, e.g., for function kprobes, the associated function arguments. For SnapBPF, the function arguments passed to the SnapBPF eBPF program include the file offset of the page that is about to be added to the page cache. In this way, once the SnapBPF eBPF program is attached to this hook, it tracks the file offsets of the pages that are fetched into the page cache from the function snapshot.

The capture phase is then as follows. First a new VM sandbox is spawned using the function snapshot. Before actually booting the VM sandbox, the VMM creates the kprobe, as described above, and attaches to it the SnapBPF eBPF capture program. Finally, it invokes the function to capture its working set. The SnapBPF eBPF capture program will be triggered for every page that is added to the system's page cache. Consequently, it has to filter out any pages that do not belong to the function snapshot file, i.e. the pages that are not fetched by the VMM. SnapBPF stores the filtered page offsets, which comprise the working set, in an eBPF map [178]. Additionally, Linux by default uses readahead to prefetch pages from disk and hide storage latency. Hence, SnapBPF disables readahead in order to only fetch and capture the working set pages in this phase. Once the function invocation finishes, the VMM reads the offsets from the eBPF map and stores them to disk. Note, that SnapBPF stores only the page offsets and not the pages themselves, as prior art does.

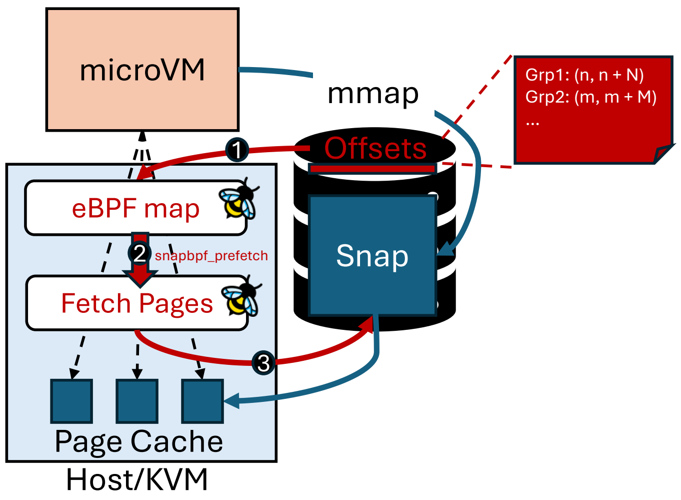


Figure 4.2: SnapBPF Prefetching. Note that SnapBPF captures the offsets of the working set pages, not the pages themselves. SnapBPF will fetch the pages directly from the snapshot file.

Loading the working set. Once the file offsets for the pages comprising the working set have been captured, SnapBPF first groups them into contiguous ranges of offsets and sort them based on the earliest access time of any of the pages in each group. SnapBPF triggers the prefetching of the pages based on this sorted group order, ensuring that read requests for the pages needed the earliest are issued first.

Figure 4.2 shows the steps that load the working set from a function snapshot. When a new VM sandbox is spawned from the function snapshot, in order to handle an incoming function invocation, the VMM first reads the grouped file offsets of the function working set from disk and loads them into the kernel via an eBPF map ①. It then attaches the SnapBPF prefetch eBPF program to the same kprobe used earlier, and triggers the prefetching by accessing the first page of the snapshot ②. The SnapBPF prefetch eBPF program will then read the grouped offsets from the eBPF map and will start issuing consecutive read requests for each contiguous range of offsets from the snapshot file, in the afore-mentioned sorted order, to fetch them into the OS page cache ③.

As the Linux kernel sandboxes eBPF programs, which prevents them from, for example, issuing block requests to storage or manipulating the OS page cache, SnapBPF implements an eBPF helper function, more specifically a kfunc [179] (snapbpf_prefetch ()) ②, which wraps around the Linux page cache readahead routine that prefetches pages from storage (page_cache_ra_unbounded()). Once it issues the read request for the last group of offsets, the eBPF program will disable itself.

By issuing read requests directly to the snapshot file, SnapBPF obviates the need to separately serialize the working set to disk and instead only uses metadata to drive the prefetching. As shown in Section 4.5, this does not penalize performance, as, in contrast to spindle HDDs, modern SSDs don't have the same limitations with regard to high-IOPS, non-sequential I/O. Nonetheless, SnapBPF does minimize the number for block requests the kernel issues to storage by grouping the pages into contiguous ranges, to reduce SW overhead. Finally, since the pages are loaded directly into the page cache, they are shared between multiple concurrent VM sandboxes for the same function, minimizing memory usage. Since SnapBPF employs eBPF and essentially works in kernel-space, there is no need for redundant userspace copies of data from the page cache, which eliminates most of the page cache overhead, that forces prior art, such as REAP and Faast, to opt for direct IO instead.

4.4 Efficient anonymous allocation filtering with AnonPTEs

This section presents the design and implementation of AnonPTEs, a lightweight paravirtualized mechanism, that enables the online filtering of anonymous allocations of the microVM sandbox, and its integration within SnapBPF, for accelerating snapshot starts for microVM-sandboxed serverless functions.

4.4.1 The problem of anonymous allocations.

Due to the semantic gap between the VM and the host memory allocator, not all pages that will be used during the invocation of the function are captured by the working set. For ephemeral memory allocations inside the VM sandbox, i.e., for memory that is allocated during the invocation and freed afterwards, the working set pages will differ between invocations. As prior art points out [81, 82], fetching these pages from snapshot is unnecessary. The host kernel can instead provide the VMM with anonymous memory.

Faast and FaaSnap both tackle this issue by resorting to scanning and pre-processing the snapshot file. FaaSnap patches the VM kernel to zero pages when they are freed. It then scans the snapshot file for zero pages and maps those zero regions of the snapshot file to anonymous memory. Faast relies on the allocator metadata of the VM kernel to identify pages that are not actively used in the snapshot and routes faults for these pages to anonymous memory.

Regardless of the mechanism employed by each approach, both rely on preemptive snapshot scanning and pre-processing to optimize the handling of VM memory allocations.

	Mechanism	On-disk WS serialization	In-memory WS deduplication	Stateless VM Allocation Filtering
REAP [80] / Faast [82]	Userfaultfd (User-space)	Yes	✗	✗
FaaSnap [81]	mincore / mmap (User-space)	Yes	✓	✗
SnapBPF	eBPF (Kernel-space)	No	✓	✗
SnapBPF \w AnonPTEs	eBPF (Kernel-space)	No	✓	✓

Table 4.1: Comparison of snapshot prefetching techniques.

4.4.2 AnonPTEs

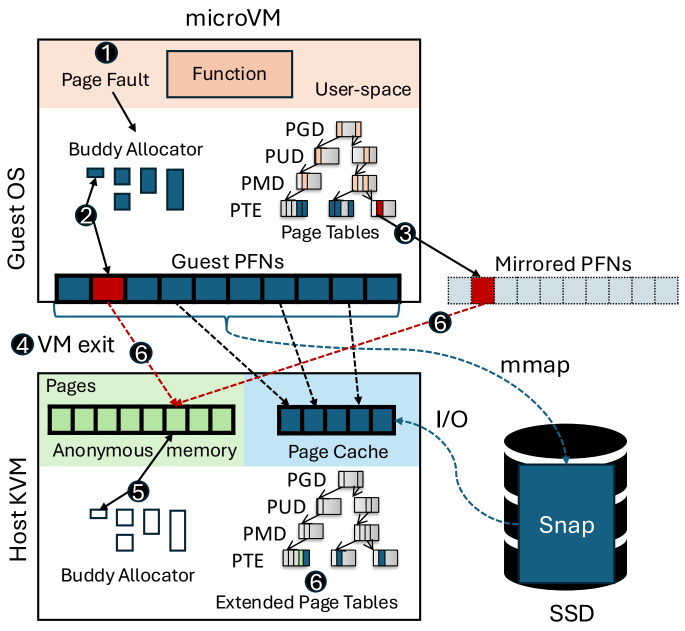


Figure 4.3: AnonPTEs: A PV interface for VM memory allocations to avoid unnecessary IO.

When booting a VM sandbox from a snapshot file, the allocation of new pages by the VM guest OS memory manager will end up fetching pages from the on-disk snapshot, which will eventually be zeroed or overwritten. As mentioned in Section 4.4.1, prior art addresses this issue by preemptively scanning the snapshot, based on either page contents (FaaSnap [81]) or allocator metadata (Faast [82]). This thesis instead adopts a different approach, that works online, without relying on snapshot scanning or pre-processing. It proposes AnonPTEs, a lightweight paravirtualized (PV) PTE marking mechanism.

Figure 4.3 shows an overview of the proposed mechanism. AnonPTEs modify the VM (guest) kernel, so that when it attempts to allocate guest memory ① ②, it marks it in a way that the host (VMM) can detect it and use anonymous memory instead of fetching data from the snapshot file to back it up. Specifically, when the VM kernel attempts to map this freshly-allocated memory in its page tables, instead of using the actual guest page frame number (gPFN) of the page, AnonPTEs set the most significant bit (MSB) of the PFN, effectively mirror-mapping this page to a higher PFN space ③.

The host kernel, specifically the Linux Kernel Virtual Machine (KVM), when handling nested page faults for the VM ④, will be able to detect faults for such mirrored PFNs. In that case, it will use anonymous memory to serve the page fault, instead of fetching pages from the on-disk snapshot ⑤. It will then map this anonymous page to both the mirrored and the original gPFN, in the VM's nested page tables, so that when the VM subsequently reuses this memory, it also points to the anonymous page allocated by the host ⑥.

In this way, AnonPTEs are able to handle the memory allocations of the VM sandbox without redundant I/O from the snapshot file or scanning the snapshot for pages that should be filtered or skipped before-hand. Note that while AnonPTEs is implemented for Linux and KVM the design is essentially hypervisor and OS agnostic.

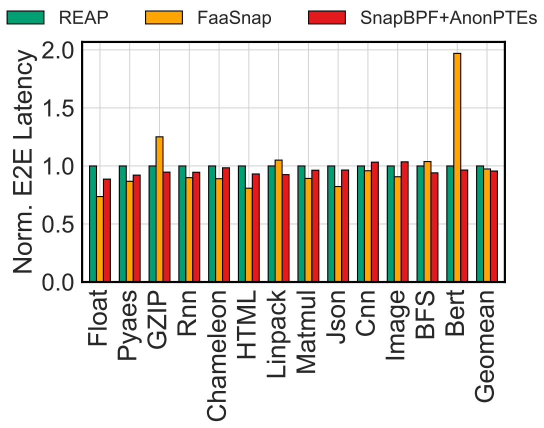
4.5 Evaluation

4.5.1 Methodology

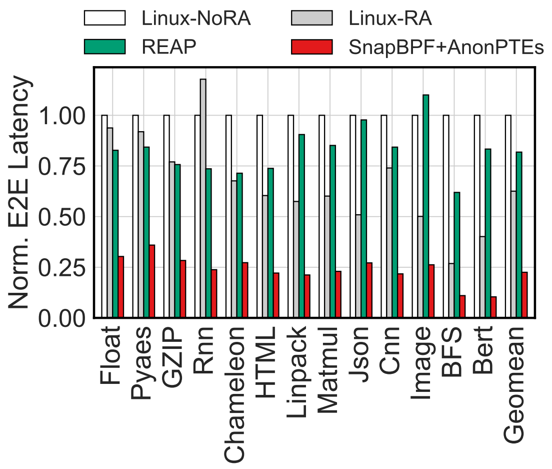
AnonPTEs are integrated in SnapBPF and the enhanced SnapBPF version (*SnapBPF \w AnonPTEs* in Table 4.1) is then evaluated against existing state-of-the-art prefetching approaches, namely REAP and FaaSnap.

AnonPTEs are implemented in Linux v6.3 and the firecracker VMM v1.11 [78] along SnapBPF. The evaluation uses functions representative of common FaaS workloads from the FunctionBench [84] suite, as well as three real-world workloads from FaaSMem [85] (html_serving, graph_bfs, bert). The firecracker VMM is instrumented to track the end-to-end latency for function invocations. The evaluation focuses both on single function execution as well as with executing 10 concurrent function instances, invoking them with identical inputs. For the Linux readahead baseline, the readahead window is set to the default Linux kernel value of 128KiB, i.e., 32 4KiB pages.

Hardware Setup. AnonPTEs and SnapBPF are evaluated on a 2-socket AMD EPYC 7402 CPU [180], with 24 hyperthreaded cores per socket. Each socket has access to 128GiB of DDR4 memory. The function memory snapshots, as well as the function working sets for REAP and FaaSnap, are stored on a 480GiB Micro 5300 TLC NAND flash SATA SSD [181]. To minimize noise, the VMM threads are pinned on specific cores of the first socket. Hyperthreading is disabled and the CPU core frequency is set to 2.5GHz.



(a) E2E function latency for a single function instance.



(b) E2E function latency for 10 concurrent function instances.

Figure 4.4: SnapBPF with AnonPTEs matches and outperforms user-space solutions without requiring separate working set files or offline snapshot scanning.

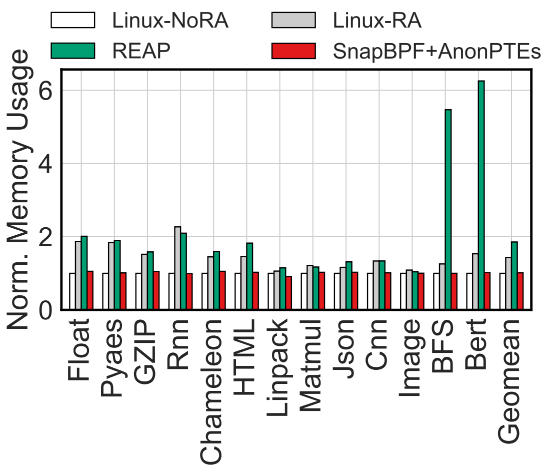


Figure 4.5: Memory consumption for 10 concurrent function instances.

4.5.2 Evaluation Results

Latency. Figure 4.4a shows the end-to-end latency of REAP, FaaSnap and SnapBPF, when executing a single function instance. SnapBPF outperforms REAP, as it doesn't have to copy pages from userspace to kernel-space via userfaultfd. It also matches and in some cases outperforms FaaSnap in terms of E2E function invocation latency, by avoiding the redundant copying of the working set to userspace in the prefetching phase, and by maintaining leaner working sets, similar to REAP.

The shortcomings of userfaultfd-based approaches are more pronounced when executing 10 concurrent instances of the same function, where deduplication and sharing of snapshot pages comes into play. Figure 4.4b shows the E2E latency for this scenario, comparing SnapBPF with vanilla firecracker (no snapshot page prefetching) with Linux readahead both disabled (*Linux-NoRA*) and enabled (*Linux-RA*), as well as with REAP. SnapBPF outperforms vanilla firecracker as it efficiently prefetches the offsets representing the invocation working set. Moreover it outperforms REAP because it enables deduplication of the snapshot pages and sharing them among all 10 function instances. Notably, for functions with large working sets, such as Bert, SnapBPF is able to achieve 8x lower E2E latency than REAP.

Memory. Figure 4.5 shows the system-wide memory usage when running 10 concurrent VM sandboxes of the same function. Userfaultfd-based approaches are unable to deduplicate the working set between different sandboxes, leading to increased memory usage with concurrent function invocations. In this scenario, SnapBPF reduces memory usage by up to 6x for functions with large working set, such as BFS and Bert.

During the experiment, Linux KVM would sometimes result in excessive Copy-on-Write allocations, when handling nested page faults, which would diminish the deduplication benefits. This was due to the fact that KVM would under certain circumstances forcibly handle *read* nested page faults as *write*. This in turn forced the host kernel to CoW the page cache pages to anonymous memory. Consequently, KVM was patched to only opportunistically write-map read nested page faults, i.e., doing it only for faulted-in and already writable pages.

SnapBPF Overhead. The latency of loading the offsets into the kernel, via the eBPF map, was measured to be less than 1% of E2E latency on average (∼1-2ms). A comprehensive analysis of the computational and memory costs of SnapBPF is left for future work.

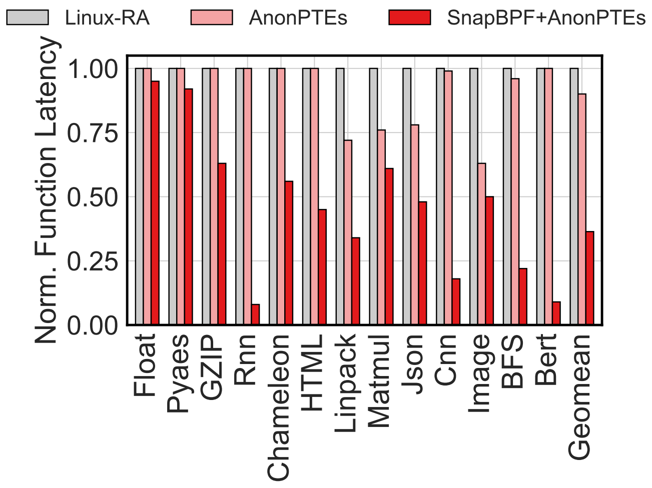


Figure 4.6: Breakdown analysis of the AnonPTEs effect on SnapBPF .

Breakdown Analysis. Figure 4.6 breaks down the effect of AnonPTEs, in terms of function execution latency, when using firecracker to restore and invoke functions from a snapshot. The default Linux readahead behavior (*Linux-RA*) is used as the baseline and show the speedup achievable when using i) only AnonPTEs (pink bar) and ii) AnonPTEs combined with eBPF prefetching (red bar).

Functions that during their invocation allocate large amounts of memory, see significant improvements from AnonPTEs , as they are able to redirect the nested page faults for such allocations to anonymous memory and eliminate the unnecessary fetching of pages from the snapshot file. The E2E latency for the image processing function (Image), for example, is improved by more than 2x. Note that SnapBPF is able to achieve this without having to keep track of and scan for stale or unused pages in the snapshot as prior art does. On the other hand, functions that rely on initialized state, e.g., models, like RNN and Bert, benefit only minimally, if at all, from the optimization of anonymous memory allocations. For these functions, the optimized working set prefetching is the dominant factor.

4.6 Related Work

Overcoming the virtualization semantic gap. There have been multiple works [73, 74] that attempt to tackle the inefficiencies induced by the semantic gap created by virtualization, with regards to OS memory management, with a focus on memory elasticity, oversubscription and overcommitment. [182] also focuses on the semantic gap created by hardware virtualization and, in the context of memory management, proposes ways to overcome the semantic gap to optimize the management of the OS page cache. Other works [183–186] employ eBPF to create para-virtualized interfaces to bridge the virtualization semantic gap for various purposes, including memory management and I/O acceleration. By contrast, AnonPTEs rely on the virtual memory hardware, i.e., the MMU and the guest page tables, to create a mirrored guest physical address space, which is used to convey the nature of guest memory allocations, thereby enabling the host OS (hypervisor) to accelerate the loading of VM memory snapshots.

eBPF. Similarly to SnapBPF, other works [187–189] have proposed using eBPF to make the OS page cache programmable, albeit targeting different use cases. eBPF has also been explored for filesystems [190], storage functions [191], and extending the OS memory manager [88].

Page Cache. Previous research has also focused on optimizing page cache performance and prefetching [192–195], albeit without targeting userspace extensibility and programmability or the FaaS use case.

FaaS Snapshotting. Optimizing function snapshotting has also been studied outside the context of working set prefetching, taking advantage of HW acceleration to improve performance with snapshot compression [196, 197] or evaluating the performance of FaaS snapshotting on storage devices with different performance profiles [198]. For container-based sandboxing, disaggregated memory has also been explored to optimize FaaS snapshotting [199–201].

4.7 Conclusion

This chapter presents AnonPTEs, a para-virtualized solution aimed at addressing the inefficiencies of executing microVM-sandboxed serverless functions from previously checkpointed memory snapshot files, arising from the semantic gap between the virtual memory subsystems of the physical host (VMM) and the microVM guest. By integrating AnonPTEs with SnapBPF, the thesis shows that it is possible to deduplicate function working sets in memory via the OS page cache, without redundant userspace copies, while also enabling the online filtering of VM-sandbox memory allocations via a lightweight paravirtualized PTE marking mechanism.

CHAPTER 5

Conclusion

Driven by inefficiencies of virtual memory, arising from the interaction of system software, i.e., the OS, with the hardware, the thesis argues the need to augment system software with hardware-tailored memory management policies, which will optimally exploit the underlying hardware to tackle these inefficiencies.

The thesis first targets the address translation overhead of paged virtual memory and proposes Elastic Translations, a hardware-tailored extension to the Linux memory manager that can efficiently harness the multi-grained translation capabilities of modern hardware. Elastic Translations are designed and implemented for the OS-assisted TLB coalescing feature, available on modern ARMv8-A and RISC-V processors, enabling the OS to efficiently use a larger range of translation granularities to map the address space of applications. Elastic Translations also employ hardware-assisted event sampling, to generate an MMU-pressure heatmap of the address space, and guide the use of these larger translation granularities, sustaining performance even under heavy fragmentation.

The thesis then shifts focus to microVM-sandboxed serverless functions, running from checkpointed memory snapshot files. The semantic gap, induced by virtualization, between the virtual memory subsystems of the physical host (VMM) and the microVM guest, result in unnecessary I/O, performance overhead, and memory bloat. To bridge that gap, the thesis proposes AnonPTEs, a lightweight paravirtualized mechanism, that piggybacks on the translation structures of the guest VM, i.e., the nested guest page tables, to communicate the nature of memory allocations performed by the guest OS to the host OS memory manager. This alleviates the aforementioned inefficiency, significantly improving the performance and tail latency of microVM serverless functions.

5.1 Future Directions for HW-tailored memory management

The results of both proposals corroborate the need for tailoring the memory management mechanisms and policies of the OS to the available hardware feature, in order to maximize performance gains. Beyond multi-grained address translation and microVM snapshotting, HW-tailored policies can be extended to encompass other aspects of the OS memory manager.

5.1.1 Extending the range of ET-supported AT granules

As mentioned previously (Section 3.5), the initial implementation of ET targets the OS-assisted TLB coalescing feature of the ARMv8-A architecture, via the contig-bit mechanism. Additionally, the experimental evaluation of ET (Section 3.4) focuses on the AT granules enabled by a 4KiB base page size, specifically 2MiB large pages, and 64KiB and 32MiB coalesced translations. ET could be extended to support different architectures and micro-architectures, that unlock different AT granules. ET could also take advantage of the ability of ARMv8-A cores to utilize different base page sizes, larger than 4KiB, which also alters the list of supported AT granules, both those enabled by large pages, and those enabled by the contig-bit mechanism. Finally, Leshy could be extended to also consider Trident 1GiB pages [63], as discussed in Section 3.5.

RISC-V Svnapot. A first step towards extending ET to support more AT granules would be to port ET to the RISC-V Svnapot extension [39]. As described in Section 3.2.2, the RISC-V architecture provisions more bits in its page table entries, which encode the contiguity status of neighboring entries. Consequently, it enables a larger range of AT granules, via OS-assisted TLB coalescing. ET would first need to transparently support the Svnapot HW mechanism, similarly to the ARMv8-A contig-bit feature (Section 3.3.1). This will also require porting ET to a newer Linux kernel version, e.g., at least 6.3, as v5.18 support for the RISC-V architecture was generally not very robust. The synchronous and asynchronous allocation components of ET will then need to be made aware of the additional available AT granules. Finally, Leshy will need to be extended to take into account the additional AT granules, when generating hints for CoalaPaging and CoalaKhugepaged.

HW-assisted TLB coalescing. Recent versions of the AMD Zen micro-architecture [101, 102] as well as the ARM Neoverse micro-architecture [132], also support HW-assisted TLB coalescing (Section 3.2.2). ET needs to be made aware of this HW feature and incorporate the additional AT granules, for the HW-coalesced translations, into its decision making and policies. Due to HW constraints, this form of coalescing typically works at a sub-cacheline granularity, enabling smaller additional AT granules. For example, the AMD Zen cores are able to coalesce only 4 neighboring page table entries to a single translation. I intend to evaluate the performance impact of these smaller AT granules, especially under fragmentation (Section 3.4.4).

1GiB Elastic Translations. As discussed both earlier and in the ET discussion section (§ 3.5), Trident [63] attempts to make the transparent support of 1GiB pages within Linux practical, mainly via asynchronous allocations and aggressive memory compaction. ET opportunistic allocations could be extended to take advantage of this extreme AT granule, while CoalaPaging could integrate the Trident compaction enhancements for asynchronous promotion. In contrast to Trident, which employs greedy allocation and promotion policies, with simplistic fallback mechanisms, ET, in conjunction with guidance from Leshy, will be able to judiciously use 1GiB pages, while also utilizing every other supported AT granule, thus minimizing fragmentation pressure.

5.1.2 Multi-grained AT for the OS Page Cache

The page cache (or disk cache) is a caching structure, maintained by the OS, which keeps the contents of files read from disk in memory, in order to minimize disk accesses and accelerate I/O operations. Until recently, Linux only used 4KiB pages for its page cache, which could potential apply significant fragmentation pressure on the host [103, 116, 202]. In order to address various issues with 4KiB pages in the Linux memory management subsystem, including the reliance of the page cache subsystem on exclusively 4KiB pages, the folio framework [203] was proposed and gradually incorporated within the Linux kernel. Regarding the page cache, this unlocks support for 2MiB THP pages, and potentially sub-2MiB mTHP pages, but faces the same limitations as THP and mTHP, discussed earlier.

One issue with current approaches is the greedy allocation policy, which for page cache also means I/O amplification, as the OS will need to fetch from disk a larger block of data to fill the allocated page. With ET, larger AT granules could potentially be supported, as they can be generated opportunistically, on-demand, balancing I/O amplification and AT performance. To that end, ET could be extended to also support page cache allocations and Leshy could be augmented for page-cache-tailored decision making and hint generation.

5.1.3 AT Tiering

With the ability of the OS memory manager to dynamically promote and demote translations, based on performance monitoring, an Elastic Translations extension could introduce the notion of translation size tiers in the OS memory manager, inspired by traditional memory tiering approaches for hybrid and heterogeneous memory systems [121, 145, 146]. The translation tiers can be defined according to the available HW-supported translation sizes. The OS memory manager would be then extended to track the available free, cold and hot memory for each tier based on

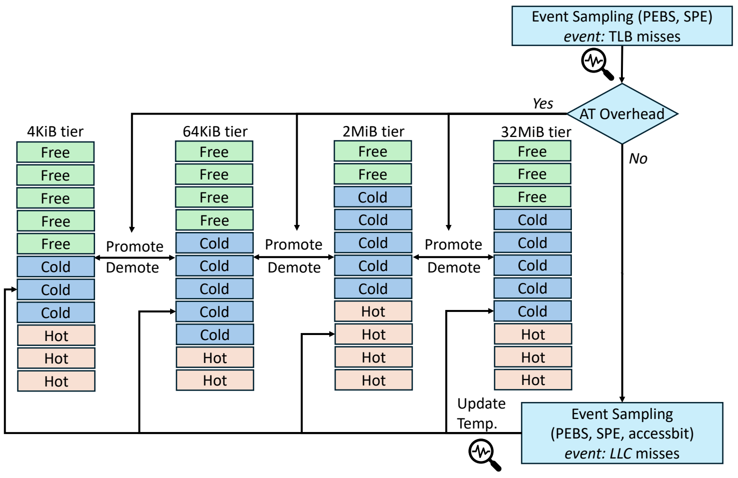


Figure 5.1: A proposal for translation size tiering.

the available free memory contiguity (Figure 5.1).

Classical memory tiering policies for proactive demotion could then be adapted to translation size tiering, to allow the OS to asynchronously optimize translation size placement and harness contiguity for fault time allocations.

By dynamically promoting and demoting memory regions to different translation sizes, based on their tracked TLB and access hotness, translation size tiering would allow the OS to maximally exploit the available contiguity in the system and ensures fairness in the way this contiguity is distributed between applications, especially in heavily fragmented systems, enabling true, bidirectional translation size elasticity.

5.1.4 TLB shootdowns

One such direction would be to more closely align the OS memory manager to the hardware features that accelerate TLB shootdowns [204, 205]. The ARMv8-A architecture supports TLB shootdown in the hardware, by broadcasting the shootdown request from the initiator core to the other cores of the system. While this obviates the need for costly inter-processor interrupt (IPI) synchronization between the initiator and the target cores, the broadcast is oblivious to the cores that might actually have the translations that need to be invalidated cached. There has been a long line of work, both academic and in the industry, to accelerate IPI-based TLB shootdowns, by carefully filtering target cores for the invalidations, and by delaying or coalescing invalidations until absolutely necessary. A hybrid mechanism that combines and chooses between both approaches might provide better performance. The OS memory management could also factor in the performance of TLB shootdowns, when for example deciding whether or not to demote the translation granularity for a region of memory, or whether or not to move a region of memory from or to another memory tier.

5.1.5 Memory tiering

OS-assisted TLB coalescing opens up an interesting direction with regard to memory tiering and its interplay with larger translation granules. Typically, when utilizing large pages, it becomes more difficult for memory tiering systems to track the skewness of larger pages. With coalesced translations, it might be possible to track memory accesses within a coalesced translation. Additionally, for systems supporting more translation granules, such as RISC-V Svnapot, would allow the memory tiering system to navigate more flexibly the trade-offs between memory performance and address translation performance.

5.1.6 Harnessing on-chip hardware accelerators for memory management

x86 processors have recently introduced on-chip accelerators that provide hardware-accelerated memory compression and migration [196, 206, 207] (Intel QAT, Intel DSA). A possible future research direction would be to assess whether these accelerators can be more tightly integrated within the OS memory manager, especially in conjunction with other techniques. For example, the CoalaKhugepaged component of Elastic Translations could use these accelerators to improve asynchronous migration performance, as it might need to migrate and compact larger amounts of memory than the vanilla Linux kernel. Hardware memory compression is already utilized as a more performance swapping backend within the Linux kernel (zswap) [208].

CHAPTER 6

List of Publications

This section provides a list of the publications the author contributed to during their thesis along with a short summary for each publication.

2025

- **SnapBPF: Exploiting eBPF for Serverless Snapshot Prefetching** [83] **[Best Paper Award]**

  Summary: Designs an eBPF-based snapshot prefetching mechanism targeting VM-sandboxed serverless functions. SnapBPF enables the efficient capture and prefetching of function working sets in kernel-space, deduplicates function working sets in memory, and includes a lightweight paravirtualized interface to handle VM-sandbox memory allocations without requiring snapshot pre-processing.

- **CXLfork: Fast Remote Fork over CXL Fabrics** [201] **[Best Paper Award]**

  Summary: Presents CXLfork, a remote fork interface that realizes close to zero-serialization, zero-copy process cloning across nodes over CXL fabrics. CXLfork utilizes globally-shared CXL memory for cluster-wide deduplication of process states and enables fine-grained control of state tiering between local and CXL memory. Used to develop CXLporter, an efficient horizontal autoscaler for serverless functions deployed on CXL fabrics.

- **Scaling Serverless Functions: Horizontal or Vertical? Both!** [209]

Summary: Investigates the trade-offs between horizontal and vertical scaling for VM-sandboxed serverless functions, proposing mechanisms for efficiently combining both approaches.

2024

- **Elastic Translations: Fast Virtual Memory with Multiple Translation Sizes** [86]

  Summary: Proposes Elastic Translations (ET), a holistic memory management solution to explore and exploit intermediate translation sizes (64KiB and 32MiB) supported by ARMv8-A and RISC-V via OS-assisted TLB coalescing. ET implements coalescing-aware OS memory management and employs policies using lightweight HW-assisted TLB miss sampling via the ARMv8-A Statistical Profiling Extension (SPE).

- **FaaSRail: Employing Real Workloads to Generate Representative Load for Serverless Research** [210]

  Summary: Presents a methodology and tool for generating realistic serverless workload traces that better represent production FaaS environments, enabling more meaningful serverless systems research.

- **eBPF-mm: Userspace-guided Memory Management in Linux with eBPF** [88] **[2nd Place]**

  Summary: Explores the use of eBPF to implement userspace-guided memory management policies in the Linux kernel, allowing applications to customize memory management behavior without kernel modifications.

- **Design, Implementation and Evaluation of the SVNAPOT Extension on a RISC-V Processor** [32]

  Summary: Presents the first hardware implementation and evaluation of the RISC-V Svnapot extension for naturally aligned power-of-two (NAPOT) page sizes, enabling TLB coalescing on RISC-V systems.

- **Fast and Efficient Memory Reclamation for Serverless MicroVMs** [75]

  Summary: Identifies the obliviousness of the Linux memory manager to virtually hot-plugged memory as the key issue hindering hot-unplug performance, and designs HotMem, a novel approach for fast and efficient VM memory hot(un)plug targeting VM-sandboxed serverless functions.

2023

- **DAPHNE Runtime: Harnessing Parallelism for Integrated Data Analysis Pipelines** [211]

  Summary: Describes the runtime system for DAPHNE, an open integrated data analysis pipeline infrastructure, focusing on parallel execution capabilities for data science workloads.

- **FaaSCell: A Case for Intra-node Resource Management** [212]

  Summary: Explores intra-node resource management strategies for Function-as-a-Service platforms, investigating how to efficiently share resources among co-located serverless functions.

2022

- **DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines** [213]

  Summary: Presents DAPHNE, an open and extensible system infrastructure designed to unify data management and machine learning workflows in a single integrated data analysis pipeline.

2020

- **Enhancing and Exploiting Contiguity for Fast Memory Virtualization** [103]

  Summary: Proposes synergistic software and hardware mechanisms that alleviate address translation overhead, focusing particularly on virtualized execution. On the software side, introduces contiguity-aware (CA) paging, a novel physical memory allocation technique that creates larger-than-a-page contiguous mappings while preserving demand paging flexibility. On the hardware side, proposes SpOT, a micro-architectural mechanism to hide TLB miss latency by exploiting the regularity of large contiguous mappings to predict address translations.

2019

- **ACTiManager: An End-to-End Interference-Aware Cloud Resource Manager** [214]

  Summary: Presents an interference-aware cloud resource manager for OpenStack.

- **Extending Storage Support for Unikernel Containers** [215]

Summary: Extends unikernel container platforms with improved storage support, enabling more practical deployment of unikernels in serverless and containerized environments.

2018

- **Efficient Resource Management for Data Centers: The ACTiCLOUD Approach** [216]

  Summary: Describes the ACTiCLOUD project's approach to efficient cloud resource management through interference-aware scheduling and placement strategies.

- **utmem: Towards Memory Elasticity in Cloud Workloads** [74]

  Summary: Proposes mechanisms for achieving memory elasticity in cloud environments by allowing userspace applications to directly interface with transcendent memory.

Bibliography & References

[1] Bruce L. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*. Synthesis Lectures on Computer Architecture. 2009. URL https://doi.org/10.2200/S00201ED1V01Y200907CAC007.

[2] Peter J Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[3] Tom Kilburn, R Bruce Payne, and David J Howarth. The atlas supervisor. In *Proceedings of the December 12-14, 1961, eastern joint computer conference: computers-key to total systems control*, pages 279–294, 1961.

[4] John Fotheringham. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, 1961.

[5] Jack B Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM (JACM)*, 12(4):589–602, 1965.

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the ACM/IEEE 40th Annual International Symposium on Computer Architecture*, 2013. URL https://doi.org/10.1145/2485922.2485943.

[7] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th IEEE/ACM Annual International Symposium on Microarchitecture*, 2014. URL https://doi.org/10.1109/MICRO.2014.37.

[8] Andrea Oliveri and Davide Balzarotti. In the land of mmus: Multiarchitecture os-agnostic virtual memory forensics. *ACM Trans. Priv. Secur.*, 25(4), July 2022. URL https://doi.org/10.1145/3528102.

[9] Ian Wienand. A survey of large-page support. *University of New South Wales*, 2006.

[10] Bruce L Jacob and Trevor N Mudge. A look at several memory management units, tlb-refill mechanisms, and page table organizations. *ACM SIGPLAN Notices*, 33(11):295–306, 1998.

[11] Bruce Jacob and Trevor Mudge. Virtual memory: Issues of implementation. *Computer*, 31 (6):33–43, 2002.

[12] Bruce Jacob and Trevor Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, 2002.

[13] Madhusudhan Talluri, Mark D Hill, and Yousef A Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 184–200, 1995.

[14] Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 39–50, 1993.

[15] Idan Yaniv and Dan Tsafrir. Hash, Don'T Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, 2016. URL https://doi.org/10.1145/2896377.2901456.

[16] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the ACM/IEEE 37th Annual International Symposium on Computer Architecture*, 2010. URL https://doi.org/10.1145/1815961.1815970.

[17] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020. URL http://doi.org/10.1145/3373376.3378493.

[18] Jovan Stojkovic, Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 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. URL https://doi.org/10.1145/3503222.3507720.

[19] OpenPOWER Foundation. *Power ISA*, v3.1 edition, May 2020. https://ibm.box.com/s/hhjfw0x0lrbtyzmiaffnbxh2fuo0fog0.

[20] Linux Kernel Mailing List - Linus Torvalds on Page Tables. https://yarchive.net/comp/linux/page_tables.html.

[21] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[22] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th ACM/IEEE Annual International Symposium on Computer Architecture*, 1992. URL https://doi.org/10.1145/139669.140406.

[23] Jeffrey C Mogul. Big memories on the desktop. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*, pages 110–115. IEEE, 1993.

[24] J Bradley Chen, Anita Borg, and Norman P Jouppi. A simulation based study of tlb performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 114–123, 1992.

[25] Yousef A Khalidi, Madhusudhan Talluri, Michael N Nelson, and Dock Williams. Virtual memory support for multiple page sizes. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*, pages 104–109. IEEE, 1993.

[26] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the 6th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994. URL https://doi.org/10.1145/195473.195531.

[27] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012. URL https://doi.org/10.1109/MICRO.2012.32.

[28] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *Proceedings of the 20th IEEE In-*

ternational Symposium on High Performance Computer Architecture, 2014. URL https://doi.org/10.1109/HPCA.2014.6835964.

[29] Juan Navarro, Sitaram Iyer, and Alan Cox. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th ACM SIGOPS Symposium on Operating Systems Design and Implementation*, 2002. URL https://doi.org/10.1145/844128.844138.

[30] Guilherme Cox and Abhishek Bhattacharjee. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017. URL https://doi.org/10.1145/3037697.3037704.

[31] Thomas W. Barr, Alan L. Cox, and Scott Rixner. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the ACM/IEEE 38th Annual International Symposium on Computer Architecture*, 2011. URL https://doi.org/10.1145/2000064.2000101.

[32] Nikolaos-Charalampos Papadopoulos Papadopoulos, Stratos Psomadakis, Vasileios Karakostas, Nectarios Koziris, and Dionisios N. Pnevmatikatos. Design, implementation and evaluation of the SVNAPOT extension on a RISC-V processor. *CoRR*, abs/2406.17802, 2024. doi: 10.48550/ARXIV.2406.17802. URL https://doi.org/10.48550/arXiv.2406.17802.

[33] Richard McDougall. Supporting mulitple page sizes in the solaris operating system, 2004.

[34] HP. Tunable Base Page Size. URL https://support.hpe.com/hpesc/public/docDisplay?cc=us&docId=emr_na-c01916157&lang=en-us.

[35] Narayanan Ganapathy and Curt Schimmel. General purpose operating system support for multiple page sizes. In *1998 USENIX Annual Technical Conference (USENIX ATC 98)*, New Orleans, LA, June 1998. USENIX Association. URL https://www.usenix.org/conference/1998-usenix-annual-technical-conference/general-purpose-operating-system-support-multiple.

[36] C. Cascaval, E. Duesterwald, P.F. Sweeney, and R.W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005. URL https://doi.org/10.1109/PACT.2005.32.

[37] Theodore H Romer, Wayne H Ohlrich, Anna R Karlin, and Brian N Bershad. Reducing tlb and memory overhead using online superpage promotion. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 176–187, 1995.

[38] Zhen Fang, Lixin Zhang, John B Carter, Wilson C Hsieh, and Sally A McKee. Reevaluating online superpage promotion with hardware support. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 63–72. IEEE, 2001.

[39] *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. RISC-V Foundation, 2021. https://wiki.riscv.org/display/HOME/RISC-V+Technical+Specifications.

[40] Richard L. Sites and Richard T. Witek. *Alpha AXP architecture reference manual*, 1995. http://www.bitsavers.org/pdf/dec/alpha/Sites_AlphaAXPArchitectureReferenceManual_2ed_1995.pdf.

[41] *Arm Architecture Reference Manual for A-profile architecture, Rev. J.a.* ARM Corporation, 2023. https://developer.arm.com/documentation/ddi0487/latest/.

[42] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations. In *Proceedings of the 44th ACM/IEEE Annual International Symposium on Computer Architecture*, 2017. URL https://doi.org/10.1145/3079856.3080217.

[43] Faruk Guvenilir and Yale N. Patt. Tailored Page Sizes. In *Proceedings of the 47th ACM/IEEE International Symposium on Computer Architecture*, 2020. URL https://doi.org/10.1109/ISCA45697.2020.00078.

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

[45] Abhishek Bhattacharjee and Daniel Lustig. *Architectural and Operating System Support for Virtual Memory*. Synthesis Lectures on Computer Architecture. 2017. URL https://doi.org/10.2200/S00795ED1V01Y201708CAC042.

[46] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

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

[48] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[49] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, et al. The stanford flash multiprocessor. In *Proceedings of the 21ST annual international symposium on Computer architecture*, pages 302–313, 1994.

[50] James Laudon and Daniel Lenoski. The sgi origin: A ccnuma highly scalable server. *ACM SIGARCH Computer Architecture News*, 25(2):241–251, 1997.

[51] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'07)*, 2007. URL https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf.

[52] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[53] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11):2–13, 2006.

[54] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005. URL https://doi.org10.5555/1247360.1247401.

[55] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008. URL https://doi.org/10.1145/1346281.1346286.

[56] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture*, 2016. URL https://doi.org/10.1109/ISCA.2016.67.

[57] Indira Subramanian, Cliff Mather, Kurt Peterson, and Balakrishna Raghunath. Implementation of multiple pagesize support in HP-UX. In *1998 USENIX Annual Technical Conference (USENIX ATC 98)*, New Orleans, LA,

June 1998. USENIX Association. URL https://www.usenix.org/conference/1998-usenix-annual-technical-conference/implementation-multiple-pagesize-support-hp-ux.

[58] IBM. Dynamic variable page size support, 2019. URL https://www.ibm.com/docs/en/aix/7.2?topic=support-dynamic-variable-page-size.

[59] HugeTLB Pages. https://docs.kernel.org/arch/arm64/hugetlbpage.html.

[60] Ryan Roberts. Multi-size THP for anonymous memory. https://lwn.net/Articles/954094/, .

[61] Transparent Hugepage Support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[62] Ashish Panwar, Sorav Bansal, and K. Gopinath. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the 24th ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019. URL https://doi.org/10.1145/3297858.3304064.

[63] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. Trident: Harnessing Architectural Resources for All Page Sizes in X86 Processors. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021. URL https://doi.org/10.1145/3466752.3480062.

[64] Simon Winwood, Yefim Shuf, and Hubertus Franke. Multiple page size support in the linux kernel. In *Proceedings of the Ottawa Linux Symposium*, OLS '02, pages 573–593, 2002.

[65] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The design and implementation of the FreeBSD operating system.* Pearson Education, 2014.

[66] Mel Gorman and Patrick Healy. Performance characteristics of explicit superpage support. In *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10. Springer-Verlag, 2010. URL https://doi.org/10.1007/978-3-642-24322-6_24.

[67] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016. URL https://doi.org/10.5555/3026877.3026931.

[68] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making Huge Pages Actually Useful. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018. URL https://doi.org/10.1145/3173162.3173203.

[69] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. MEGA: Overcoming Traditional Problems with OS Huge Page Management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, 2019. URL https://doi.org/10.1145/3319647.3325839.

[70] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014. URL https://doi.org/10.5555/2643634.2643659.

[71] Weixi Zhu, Alan L. Cox, and Scott Rixner. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020. URL https://doi.org/10.5555/3489146.3489203.

[72] Mark Mansi, Bijan Tabatabai, and Michael M. Swift. CBMM: Financial Advice for Kernel Memory Managers. In *Proceedings of the 2022 USENIX Annual Technical Conference*, 2022. URL https://www.usenix.org/conference/atc22/presentation/mansi.

[73] Jinho Hwang, Ahsen Uppal, Timothy Wood, and Howie Huang. Mortar: Filling the gaps in data center memory. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 53–64, 2014.

[74] Aimilios Tsalapatis, Stefanos Gerangelos, Stratos Psomadakis, Konstantinos Papazafeiropoulos, and Nectarios Koziris. utmem: Towards memory elasticity in cloud workloads. In *International Conference on High Performance Computing*, pages 173–183. Springer, 2018.

[75] Orestis Lagkas Nikolos, Chloe Alverti, Stratos Psomadakis, Georgios Goumas, and Nectarios Koziris. Fast and efficient memory reclamation for serverless microvms. *arXiv preprint arXiv:2411.12893*, 2024.

[76] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. Serverless cold starts and where to find them. *arXiv preprint arXiv:2410.06145*, 2024.

[77] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX annual technical conference (USENIX ATC)*, 2018. URL https://www.usenix.org/conference/atc18/presentation/wang-liang.

[78] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020. URL https://www.usenix.org/conference/nsdi20/presentation/agache.

[79] Firecracker Development Team. Firecracker Snapshotting, 2025. URL https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md.

[80] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 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. URL https://doi.org/10.1145/3445814.3446714.

[81] Lixiang Ao, George Porter, and Geoffrey M Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022. URL https://doi.org/10.1145/3492321.3524270.

[82] Yongshu Bai, Zhihui Yang, and Feng Gao. Faast: An efficient serverless framework made snapshot-based function response fast. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 174–185, 2024.

[83] Stratos Psomadakis, Dimitrios Siakavaras, Chloe Alverti, Symeon Porgiotis, Orestis Lagkas Nikolos, Christos Katsakioris, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Snapbpf: Exploiting ebpf for serverless snapshot prefetching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '25, 2025. URL https://doi.org/10.1145/3736548.3737823.

[84] Jeongchul Kim and Kyungyong Lee. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019. URL https://doi.org/10.1109/CLOUD.2019.00091.

[85] Chuhao Xu, Yiyu Liu, Zijun Li, Quan Chen, Han Zhao, Deze Zeng, Qian Peng, Xueqi Wu, Haifeng Zhao, Senbo Fu, and Minyi Guo. FaaSMem: Improving Memory Efficiency of Serverless Computing with Memory Pool Architecture. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024. URL https://doi.org/10.1145/3620666.3651355.

[86] Stratos Psomadakis, Chloe Alverti, Vasileios Karakostas, Christos Katsakioris, Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Elastic translations: Fast virtual memory with multiple translation sizes. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024. URL https://doi.org/10.1109/MICRO61859.2024.00012.

[87] Stratos Psomadakis and Georgios Goumas. Transparent os support for variable translation sizes. https://2023.eurosys.org/docs/posters/eurosys23posters-final42.pdf, 2023. EuroSys'23 Posters.

[88] Konstantinos Mores, Stratos Psomadakis, and Georgios Goumas. ebpf-mm: Userspace-guided memory management in linux with ebpf. *arXiv preprint arXiv:2409.11220*, 2024.

[89] Linux Kernel Documentation. eBPF, 2025. URL https://docs.kernel.org/bpf/.

[90] Symeon Porgiotis. Βελτιστοποίηση των λειτουργιών Εισόδου/Εξόδου στο λειτουργικό σύστημα linux με χρήση της τεχνολογίας eBPF, 2024. URL http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/19022.

[91] Lorenzo Stoakes. *The Linux Memory Manager*. No Starch Press, 2025. ISBN 9781718504462. URL https://nostarch.com/linux-memory-manager.

[92] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly Media, 2005. ISBN 0596005652. URL https://www.oreilly.com/library/view/understanding-the-linux/0596005652/.

[93] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, October 1965. ISSN 0001-0782. doi: 10.1145/365628.365655. URL https://doi.org/10.1145/365628.365655.

[94] Donald Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd Edition*. Addison-Wesley Professional, 1997. ISBN 978-0201896831. URL https://www.amazon.com/Art-Computer-Programming-Vol-Fundamental/dp/0201896834.

[95] Uresh Vahalia. *UNIX Internals: The New Frontiers 1st Edition.* Pearson, 1996. ISBN 978-0131019089. URL https://www.amazon.com/UNIX-Internals-Frontiers-Uresh-Vahalia/dp/0131019082.

[96] What is NUMA? https://www.kernel.org/doc/html/v5.0/vm/numa.html.

[97] Matthew Wilcox. XArray. https://docs.kernel.org/core-api/xarray.html.

[98] Fengguang Wu, Hongsheng Xi, Jun Li, and Nanhai Zou. Linux readahead: less tricks for more. In *Proceedings of the Linux Symposium*, volume 2, pages 273–284, 2007. URL https://www.kernel.org/doc/ols/2007/ols2007v2-pages-273-284.pdf.

[99] Neil Brown. Readahead: the documentation I wanted to read. Linux Weekly News (LWN), 2022. URL https://lwn.net/Articles/888715/.

[100] readahead(2) – Linux manual page. Linux man-pages project. URL https://man7.org/linux/man-pages/man2/readahead.2.html.

[101] Mike Clark. A new ×86 core architecture for the next generation of computing. In *Proceedings of the 2016 IEEE Hot Chips 28 Symposium*, 2016. URL https://doi.org/10.1109/HOTCHIPS.2016.7936224.

[102] *Software Optimization Guide for AMD EPYC™ 7003 Processors, Rev 3.00.* AMD, 2020. https://developer.amd.com/resources/developer-guides-manuals/.

[103] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*, 2020. URL https://doi.org/10.1109/ISCA45697.2020.00050.

[104] Timothy Merrifield and H. Reza Taheri. Performance Implications of Extended Page Tables on Virtualized X86 Processors. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2016. URL https://doi.org/10.1145/2892242.2892258.

[105] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation Ranger: Operating System Support for Contiguity-aware TLBs. In *Proceedings of the 46th ACM/IEEE*

International Symposium on Computer Architecture, 2019. URL https://doi.org/10.1145/3307650.3322223.

[106] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 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*, 2022. URL https://doi.org/10.1145/3503222.3507718.

[107] Artemiy Margaritov, Dmitrii Ustiugov, Amna Shahab, and Boris Grot. PTEMagnet: Fine-Grained Physical Memory Reservation for Faster Page Walks in Public Clouds. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021. URL https://doi.org/10.1145/3445814.3446704.

[108] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant Memory Mappings for fast access to large memories. In *Proceedings of the 42nd ACM/IEEE Annual International Symposium on Computer Architecture*, 2015. URL https://doi.org/10.1145/2749469.2749471.

[109] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. Rebooting Virtual Memory with Midgard. In *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture*, 2021. URL https://doi.org/10.1109/ISCA52012.2021.00047.

[110] Abhishek Bhattacharjee. Preserving Virtual Memory by Mitigating the Address Translation Wall. *IEEE Micro*, 2017. URL https://doi.org/10.1109/MM.2017.3711640.

[111] Intel Corporation. 5-Level Paging and 5-Level EPT White Paper, 2017. URL https://cdrdv2-public.intel.com/671442/5-level-paging-white-paper.pdf.

[112] CXL Consortium. Compute Express Link Specification Revision 2.0. https://www.computeexpresslink.org/download-the-specification, 2023.

[113] Dennard scaling. https://en.wikipedia.org/wiki/Dennard_scaling.

[114] Moore's Law. https://en.wikipedia.org/wiki/Moore%27s_law.

[115] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos.

Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters. In *Proceedings of the 50th ACM/IEEE Annual International Symposium on Computer Architecture*, 2023. URL https://doi.org/10.1145/3579371.3589079.

[116] Mark Mansi and Michael M. Swift. Characterizing physical memory fragmentation. https://arxiv.org/abs/2401.03523, 2024.

[117] Eliot H. Solomon, Yufeng Zhou, and Alan L. Cox. An Empirical Evaluation of PTE Coalescing. In *Proceedings of the 2023 IEEE International Symposium on Memory Systems*, 2023. URL https://doi.org/10.1145/3631882.3631902.

[118] Ryan Roberts. Transparent contiguous PTEs for User mappings". https://lore.kernel.org/linux-arm-kernel/87fs0xxd5g.fsf@nvdebian.thelocal/T/, .

[119] *The Armv8.9 architecture extension.* ARM Corporation, 2025. https://developer.arm.com/documentation/109697/2025_09/Feature-descriptions/The-Armv8-9-architecture-extension?lang=en#md454-the-armv89-architecture-extension__feat_FEAT_SPEv1p4.

[120] James Clark - LKML. [PATCH v5 12/12] perf docs: arm-spe: Document new SPE filtering features. URL https://lore.kernel.org/kvmarm/20250721-james-perf-feat_spe_eft-v5-12-a7bc533485a1@linaro.org/.

[121] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023. URL https://doi.org/10.1145/3600006.3613167.

[122] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 2006. URL https://doi.org/10.1145/1186736.1186737.

[123] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008. URL https://doi.org/10.1145/1454115.1454128.

[124] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite, 2017.

[125] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, 2012. URL http://arxiv.org/abs/1205.6233.

[126] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014. URL https://www.mcs.anl.gov/papers/P5064-0114.pdf.

[127] LibSVM. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html.

[128] KDD'12 dataset. https://www.kaggle.com/c/kddcup2012-track1, 2012.

[129] GUPS: HPCC RandomAccess benchmark. https://github.com/alexandermerritt/gups.

[130] WiWynn Mt.Jade. https://www.wiwynn.com/products/19-inch/sv328r.

[131] *Ampere® Altra® Rev A1 64-Bit Multi-Core Processor Datasheet, Rev 1.40.* Ampere Computing, 2023. https://amperecomputing.com/customer-connect/products/altra-family-device-documentation.

[132] *Arm® Neoverse™ N1 Core, Rev r4p1.* ARM Corporation, 2023. https://developer.arm.com/documentation/100616/0401/.

[133] gperftools. https://github.com/gperftools/gperftools.

[134] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Proceedings of the 2006 Ottawa Linux Symposium*, 2006. URL https://www.kernel.org/doc/ols/2006/ols2006v1-pages-369-384.pdf.

[135] Jonathan Corbet. Large folios for anonymous memory. https://lwn.net/Articles/937239/.

[136] *madvise(2) — Linux manual page.* Linux man-pages, 2025. https://man7.org/linux/man-pages/man2/madvise.2.html.

[137] *process_madvise(2) — Linux manual page.* Linux man-pages, 2025. https://man7.org/linux/man-pages/man2/process_madvise.2.html.

[138] SeongJae Park, Yunjae Lee, and Heon Y Yeom. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*, pages 1–7, 2019.

[139] Alan L. Cox. Medium-sized superpages on arm64 and beyond. https://www.freebsd.org/status/report-2022-04-2022-06/superpages/, 2022.

[140] Aninda Manocha, Zi Yan, Tureci Esin, Juan Luis Aragón, Nellans David, and Margaret Martonosi. Architectural Support for Optimizing Huge Page Selection Within the OS. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023. URL https://webs.um.es/jlaragon/papers/manocha_MICRO23.pdf.

[141] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, and Xiaoning Ding. Making Dynamic Page Coalescing Effective on Virtualized Clouds. In *Proceedings of the 18th ACM SIGOPS European Conference on Computer Systems*, 2023. URL https://doi.org/10.1145/3552326.3567487.

[142] Yufeng Zhou, Alan L. Cox, Sandhya Dwarkadas, and Xiaowan Dong. The Impact of Page Size and Microarchitecture on Instruction Address Translation Overhead. *ACM Trans. Archit. Code Optim.*, 2023. URL https://doi.org/10.1145/3600089.

[143] Mohammad Agbarya, Idan Yaniv, Jayneel Gandhi, and Dan Tsafrir. Predicting Execution Times With Partial Simulations in Virtual Memory Research: Why and How. In *Processors of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2020. URL https://doi.org/10.1109/MICRO50266.2020.00046.

[144] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John C. S. Lui. SmartMD: A High Performance Deduplication Engine with Mixed Pages. In *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017. URL https://doi.org/10.5555/3154690.3154759.

[145] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023. URL https://doi.org/10.1145/3582016.3582063.

[146] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021. URL https://doi.org/10.1145/3477132.3483550.

[147] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, 2023.

[148] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. Perforated Page: Supporting Fragmented Memory Allocation for Large Pages. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*, 2020. URL https://doi.org/10.1109/ISCA45697.2020.00079.

[149] Sam Ainsworth and Timothy M. Jones. Compendia: Reducing Virtual-Memory Costs via Selective Densification. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, 2021. URL https://doi.org/10.1145/3459898.3463902.

[150] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017. URL https://doi.org/10.1145/3079856.3080210.

[151] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019. URL https://doi.org/10.1145/3352460.3358294.

[152] Yu Du, Miao Zhou, Bruce R. Childers, Daniel Mossé, and Rami Melhem. Supporting superpages in non-contiguous physical memory. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*, 2015. URL https://doi.org/10.1109/HPCA.2015.7056035.

[153] Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martín Farach-Colton, Rob Johnson, Sudarsun Kannan, William Kuszmaul, Nirjhar Mukherjee, Don Porter, Guido Tagliavini, Janet Vorobyeva, and Evan West. Paging and the Address-Translation Problem. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2021. URL https://doi.org/10.1145/3409964.3461814.

[154] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. BabelFish: Fusing Address Translations for Containers. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2020. URL https://doi.org/10.1109/ISCA45697.2020.00049.

[155] Misel-Myrto Papadopoulou, Xin Tong, André Seznec, and Andreas Moshovos. Prediction-based superpage-friendly TLB designs. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*, 2015. URL https://doi.org/10.1109/HPCA.2015.7056034.

[156] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. CSALT: Context Switch Aware Large TLB. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017. URL https://doi.org/10.1145/3123939.3124549.

[157] Shai Bergman, Mark Silberstein, Takahiro Shinagawa, Peter Pietzuch, and Lluís Vilanova. Translation Pass-Through for Near-Native Paging Performance in VMs. In *Proceedings of the 2023 USENIX Annual Technical Conference*, 2023. URL https://www.usenix.org/conference/atc23/presentation/bergman.

[158] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways? In *Proceedings of the IEEE/ACM 48th International Symposium on Microarchitecture*, 2015. URL https://doi.org/10.1145/2830772.2830773.

[159] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014. ISBN 978-1-4799-6998-2.

[160] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N. Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliavini, Evan West, Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martin Farach-Colton, Jayneel Gandhi, Rob Johnson, Sudarsun Kannan, and Donald E. Porter. Mosaic Pages: Big TLB Reach with Small Pages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023. URL https://doi.org/10.1145/3582016.3582021.

[161] Dongwei Chen, Dong Tong, Chun Yang, Jiangfang Yi, and Xu Cheng. FlexPointer: Fast Address Translation Based on Range TLB and Tagged Pointers. *ACM Trans. Archit. Code Optim.*, 2023. URL https://doi.org/10.1145/3579854.

[162] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018. URL https://doi.org/10.1145/3173162.3173194.

[163] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. CARAT: A Case for Virtual Memory through Compiler- and Runtime-Based Address Translation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020. URL https://doi.org/10.1145/3385412.3385987.

[164] OS-level Virtualization. https://en.wikipedia.org/wiki/OS-level_virtualization.

[165] Amazon Web Services. AWS Lambda, 2025. URL https://aws.amazon.com/lambda.

[166] Microsoft Azure. Azure functions, 2025. URL https://azure.microsoft.com/en-us/products/functions.

[167] Huawei Corporation. Huawei Cloud Functions, 2025. URL https://developer.huawei.com/consumer/en/agconnect/cloud-function/.

[168] Google Corporation. Google Serverless Computing, 2025. URL https://cloud.google.com/serverless.

[169] Alibaba Corporation. Alibaba Serverless Application Engine, 2025. URL https://www.aliyun.com/product/aliware/sae.

[170] Cloudflare Corporation. Cloudflare Workers, 2025. URL https://workers.cloudflare.com/.

[171] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020. URL https://doi.org/10.1145/3373376.3378512.

[172] Amazon Web Services. Improving startup performance with Lambda SnapStart , 2025. URL https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html.

[173] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. Pronghorn: Effective checkpoint orchestration for serverless hot-starts. In *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024.

[174] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022.

[175] Linux Kernel Documentation. Userfaultfd, 2025. URL `https://docs.kernel.org/admin-guide/mm/userfaultfd.html"`.

[176] Linux man pages. mincore(2), 2025. URL `https://man7.org/linux/man-pages/man2/mincore.2.html`.

[177] Linux Kernel Documentation. Kernel probes (kprobes), 2025. URL `https://docs.kernel.org/trace/kprobes.html`.

[178] Linux Kernel Documentation. Bpf maps, 2025. URL `https://docs.kernel.org/bpf/maps.html`.

[179] Linux Kernel Documentation. Bpf kernel functions (kfuncs), 2025. URL `https://docs.kernel.org/bpf/kfuncs.html`.

[180] Inc. Advanced Micro Devices. Amd epyc 7402, 2019. URL `https://www.amd.com/en/support/downloads/drivers.html/processors/epyc/epyc-7002-series/amd-epyc-7402.html`.

[181] Micron Technology Inc. 5300 series sata nand flash ssd, 2019. URL `https://advdownload.advantech.com/productfile/PIS/96FD25-ST1.9T-M53P/file/96FD25-ST19T-M53P_Datasheet20200120180650.pdf`.

[182] Stephen Todd Jones. *Implicit operating system awareness in a virtual machine monitor*. PhD thesis, University of Wisconsin–Madison, 2007.

[183] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 97–112, 2018.

[184] Michael Wei and Nadav Amit. Leveraging hyperupcalls to bridge the semantic gap: An application perspective. *IEEE Data Eng. Bull.*, 42(1):22–35, 2019.

[185] Ori Ben Zur, Jakob Krebs, Shai Aviram Bergman, and Mark Silberstein. Accelerating nested virtualization with {HyperTurtle}. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 987–1002, 2025.

[186] Li Wang, Shi Qiu, Jianqin Yan, Zhirong Shen, Qingbo Wu, Xin Yao, Meiling Wang, Renhai Chen, and Yiming Zhang. A tale of two paths: Optimizing paravirtualized storage i/o with ebpf. *ACM Transactions on Storage*, 2025.

[187] Dusol Lee, Inhyuk Choi, Chanyoung Lee, Sungjin Lee, and Jihong Kim. P2cache: An application-directed page cache for improving performance of data-intensive applications. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, 2023. URL https://doi.org/10.1145/3599691.3603408.

[188] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. {FetchBPF}: Customizable prefetching policies in linux with {eBPF}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024. URL https://www.usenix.org/conference/atc24/presentation/cao.

[189] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. Cache is king: Smart page eviction with ebpf. *arXiv preprint arXiv:2502.02750*, 2025.

[190] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[191] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. {XRP}:{In-Kernel} storage functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[192] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped i/o on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021. URL https://doi.org/10.1145/3447786.3456242.

[193] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[194] Pei Cao, Edward W Felten, Anna R Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems (TOCS)*, 14(4), 1996.

[195] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, 1994.

[196] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. Sabre:{Hardware-Accelerated} snapshot compression for serverless {MicroVMs}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024. URL https://www.usenix.org/conference/osdi24/presentation/lazarev.

[197] Yuqiao Lan, Xiaohui Peng, and Yifan Wang. Snapipeline: Accelerating snapshot startup for faas containers. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, 2024. URL https://doi.org/10.1145/3698038.3698513.

[198] Christos Katsakioris, Chloe Alverti, Vasileios Karakostas, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Faas in the age of (sub-) μs i/o: a performance analysis of snapshotting. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, 2022. URL https://doi.org/10.1145/3534056.3534938.

[199] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast {RDMA-codesigned} remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.

[200] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, et al. Trenv: Transparently share serverless execution environments across different functions and nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024.

[201] Chloe Alverti, Stratos Psomadakis, Burak Ocalan, Shashwat Jaiswal, Tianyin Xu, and Josep Torrellas. Cxlfork: Fast remote fork over cxl fabrics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025.

[202] Alexander Halbuer, Illia Ostapyshyn, Lukas Steiner, Lars Wrenger, Matthias Jung, Christian Dietrich, and Daniel Lohmann. The new costs of physical memory fragmentation. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems*, 2024. URL https://doi.org/10.1145/3698783.3699378.

[203] OpenAnolis. New features of linux memory management - memory folios. https://www.alibabacloud.com/blog/

new-features-of-linux-memory-management---memory-folios_600565.

[204] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down tlb shootdowns! In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[205] John Henry Deppe and Guy Lemieux. Improving risc-v tlb shootdown performance. https://riscv-europe.org/summit/2025/media/proceedings/2025-05-13-RISC-V-Summit-Europe-P3.2.02-DEPPE-abstract.pdf, 2025. RISC-V Summit Europe '25.

[206] Intel® quickassist technology (intel® qat). https://www.intel.com/content/www/us/en/developer/topic-technology/open/quick-assist-technology/overview.html.

[207] Intel® data streaming accelerator (intel® dsa). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/data-streaming-accelerator.html.

[208] zswap. https://docs.kernel.org/admin-guide/mm/zswap.html.

[209] Orestis Lagkas Nikolos, Chloe Alverti, Stratos Psomadakis, Georgios Goumas, and Nectarios Koziris. Scaling serverless functions: Horizontal or vertical? both! In *Proceedings of the 3rd Workshop on SErverless Systems, Applications and MEthodologies*, pages 30–32, 2025.

[210] Christos Katsakioris, Chloe Alverti, Konstantinos Nikas, Dimitrios Siakavaras, Stratos Psomadakis, and Nectarios Koziris. Faasrail: Employing real workloads to generate representative load for serverless research. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, 2024. URL https://doi.org/10.1145/3625549.3658684.

[211] Aristotelis Vontzalidis, Stratos Psomadakis, Constantinos Bitsakos, Mark Dokter, Kevin Innerebner, Patrick Damme, Matthias Boehm, Florina Ciorba, Ahmed Eleliemy, Vasileios Karakostas, Aleš Zamuda, and Dimitrios Tsoumakos. Daphne runtime: Harnessing parallelism for integrated data analysis pipelines. In *Euro-Par 2023: Parallel Processing Workshops*, pages 242–246. Springer Nature Switzerland, 2024. ISBN 978-3-031-48803-0.

[212] Christos Katsakioris, Chloe Alverti, Konstantinos Nikas, Stratos Psomadakis, Vasileios Karakostas, and Nectarios Koziris. Faascell: A case for intra-node resource management: Work-in-progress. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*, pages 57–60, 2023.

[213] Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina M. Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Faerber, Georgios I. Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaz Kosar, Alexander Krause, Daniel Krems, Andreas Laber, Wolfgang Lehner, Eric Mier, Marcus Paradies, Bernhard Peischl, Gabrielle Poerwawinata, Stratos Psomadakis, Tilmann Rabl, Piotr Ratuszniak, Pedro Silva, Nikolai Skuppin, Andreas Starzacher, Benjamin Steinwender, Ilin Tolovski, Pinar Tözün, Wojciech Ulatowski, Yuanyuan Wang, Izajasz P. Wrosz, Ales Zamuda, Ce Zhang, and Xiaoxiang Zhu. DAPHNE: an open and extensible system infrastructure for integrated data analysis pipelines. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL https://www.cidrdb.org/cidr2022/papers/p4-damme.pdf.

[214] Stratos Psomadakis, Stefanos Gerangelos, Dimitrios Siakavaras, Ioannis Papadakis, Marina Vemmou, Aspa Skalidi, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, and Georgios Goumas. Actimanager: An end-to-end interference-aware cloud resource manager. In *Proceedings of the 20th International Middleware Conference Demos and Posters*, pages 27–28, 2019.

[215] Orestis Lagkas Nikolos, Konstantinos Papazafeiropoulos, Stratos Psomadakis, Anastassios Nanos, and Nectarios Koziris. Extending storage support for unikernel containers. In *Proceedings of the 5th International Workshop on Serverless Computing*, pages 31–36, 2019.

[216] Vasileios Karakostas, Georgios Goumas, Ewnetu Bayuh Lakew, Erik Elmroth, Stefanos Gerangelos, Simon Kolberg, Konstantinos Nikas, Stratos Psomadakis, Dimitrios Siakavaras, Petter Svärd, et al. Efficient resource management for data centers: the acticloud approach. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 244–246, 2018.