Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Ειδικευμένες Αρχιτεκτονικές για ενεργειακά αποδοτική και έξυπνη εκχώρηση φυσικής μνήμης

Μελέτη και υλοποίηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Σπύρου Γαλανόπουλου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
Αθήνα, Οκτώβριος 2025

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

Ειδικευμένες Αρχιτεκτονικές για ενεργειακά αποδοτική και έξυπνη εκχώρηση φυσικής μνήμης

Μελέτη και υλοποίηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Σπύρου Γαλανόπουλου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29η Οκτωβρίου, 2025.

.....	.....	.....
Δημήτριος Σούντρης	Σωτήριος Ξύδης	Γεώργιος Λεντάρης
Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής ΠΑ.Δ.Α.

Αθήνα, Οκτώβριος 2025

.....

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

στην οικογένεια μου

Περίληψη

Η δέσμευση μνήμης έχει καταστεί καίριο σημείο συμφόρησης αναφορικά με την επίδοση και την ενέργεια σε σύγχρονα συστήματα. Κάθε δέσμευση μνήμης κατά κανόνα απαιτεί χειρισμό ενός σφάλματος σελίδας στο software, που συνεπάγεται ακριβά context switches, pipeline flushes και εκτέλεση σε ενεργοβόρους out-of-order πυρήνες. Παρότι αυτά τα overheads είναι αποδεκτά για μεγάλης διάρκειας εφαρμογές, κυριαρχούν σε μικρής διάρκειας και ευαίσθητα στην καθυστέρηση workloads όπως οι serverless συναρτήσεις, τα microservices και το LLM inference, όπου η δέσμευση μνήμης συχνά αντιπροσωπεύει περισσότερο από το 30% του συνολικού χρόνου εκτέλεσης. Ταυτόχρονα, σύνθετες πολιτικές τοποθέτησης (π.χ. χρωματισμός σελίδων, NUMA/NUCA-aware δέσμευση) απαιτούν όλο και στενότερη σύζευξη μεταξύ δέσμευσης μνήμης και μετάφραση σελίδας, καθώς και αποκρισιμότητα σε συνθήκες χρόνου εκτέλεσης (runtime), δυνατότητες που είναι δύσκολο να επιτευχθούν σε χρονικές κλίμακες software.

Στην παρούσα εργασία προτείνουμε το **Valinor**, ένα νέο hardware-software co-design το οποίο επιταχύνει και εμπλουτίζει τη δέσμευση μνήμης, εισάγοντας έναν προγραμματιζόμενο μηχανισμό στο hardware. Το Valinor επιτρέπει στο λειτουργικό σύστημα να προγραμματίζει επιλεκτικά τον μηχανισμό αυτό, ώστε να χειρίζεται δεσμεύσεις μνήμης κατευθείαν στο hardware, παρακάμπτοντας ακριβά kernel traps. Ο μηχανισμός ενσωματώνεται στην ιεραρχία μνήμης, εκτελεί πολιτικές που ορίζονται από το ΛΣ σε σφάλματα σελίδας, ενημερώνει τις δομές μετάφρασης και προσαρμόζει τις αποφάσεις τοποθέτησης δυναμικά χρησιμοποιώντας ανατροφοδότηση από το υλικό (π.χ. εύρος ζώνης DRAM, πληρότητα κρυφής μνήμης). Έτσι επιτρέπει γρήγορη, translation-aware, και προσαρμοζόμενη στον χρόνο εκτέλεσης δέσμευση μνήμης, διατηρώντας παράλληλα τον έλεγχο του ΛΣ πάνω στην πολιτική.

Κατασκευάζουμε ένα πρωτότυπο του Valinor σε FPGA χρησιμοποιώντας έναν τροποποιημένο RISC-V πυρήνα που τρέχει Linux. Σε διάφορα μικρής διάρκειας workloads το Valinor επιταχύνει τη δέσμευση σελίδων κατά $7-15\times$ και βελτιώνει τη συνολική επίδοση κατά 77%, με αμελητέο κόστος υλικού (1.5%). Το Valinor αποδεικνύει ότι ο συνδυασμός της προγραμματισιμότητας με την επιτάχυνση στο hardware καθιστά εφικτά συστήματα μνήμης που συνδυάζουν υψηλές επιδόσεις με μεγάλη προσαρμοστικότητα σε δυναμικές συνθήκες χρόνου εκτέλεσης.

Λέξεις Κλειδιά — εικονική μνήμη, σφάλμα σελίδας, μετάφραση διεύθυνσης, επαναπρογραμματιζόμενο υλικό, λειτουργικά συστήματα, αρχιτεκτονική CPU

Abstract

Memory allocation has become a critical performance and energy bottleneck in modern systems. Each allocation typically requires handling a page fault in software, which involves expensive context switches, pipeline flushes, and execution on power-hungry out-of- order cores. While acceptable for long-running applications, these overheads dominate short-lived and latency-sensitive workloads such as serverless functions, microservices, and LLM inference, where allocation often accounts for more than 30% of total runtime. At the same time, advanced placement policies (e.g., page coloring, NUMA/NUCA-aware allocation) increasingly demand tight coupling between allocation and address translation, as well as responsiveness to runtime conditions—capabilities that are hard to achieve at software timescales.

We present Valinor, a new hardware–software co-design that accelerates and enriches memory allocation by introducing a programmable hardware allocation engine. Valinor allows the operating system to selectively program this engine to handle allocations directly in hardware, bypassing expensive kernel traps. The engine integrates into the memory hierarchy, executes OS-defined policies on page faults, updates translation structures, and adapts placement decisions dynamically using microarchitectural feedback (e.g., DRAM bandwidth, cache occupancy). This enables fast, translation- aware, and runtime-adaptive memory allocation, while retaining OS control over policy.

We prototype Valinor on an FPGA using a modified RISC-V core that runs Linux. Across a range of short-lived and placement-sensitive workloads, Valinor accelerates page allocation by $7-15\times$, improves end-to-end application performance by 77% on average, with negligible hardware cost (1.5% area). Valinor demonstrates that combining OS programmability with hardware acceleration enables memory systems that are both high-performance and highly adaptable to dynamic runtime conditions.

Keywords — virtual memory, page fault, address translation, reconfigurable hardware, operating systems, CPU architecture

Ευχαριστίες

Πρώτα απ' όλα θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή μου Δημήτριο Σούντρη που όχι μόνο μου έδωσε τη δυνατότητα να εκπονήσω τη διπλωματική μου στο εργαστήριό του, αλλά με εμπιστεύτηκε να αναλάβω και ένα από τα κοινά θέματα συνεργασίας με το SAFARI lab στο ETH. Ευχαριστώ επίσης τον υποψήφιο διδάκτορα Κωνσταντίνο Κανελλόπουλο και τον Δρ. Δημοσθένη Μασούρο για την πολύτιμη καθοδήγησή τους και την επίλυση αποριών στην πορεία της διπλωματικής. Θα ήθελα επίσης να ευχαριστήσω όλα τα μέλη του Microlab για το ευχάριστο κλίμα που διαμορφώνουν στο εργαστήριο. Ιδιαίτερη μνεία αρμόζει στον υποψήφιο διδάκτορα Ηλία Παπαλάμπρου για την τεχνική υποστήριξή του κατά την υλοποίηση του πρωτοτύπου στο FPGA. Επιπλέον, ένα μεγάλο ευχαριστώ οφείλω στον καθηγητή Onur Mutlu που με δέχτηκε ως intern στο εργαστήριό του, καθώς και στα υπόλοιπα μέλη του SAFARI lab, τα οποία διευκόλυναν την εγκατάστασή μου στο περιβάλλον του ETH και της Ζυρίχης γενικότερα. Ευχαριστώ ακόμη τους φίλους μου σε Αθήνα και Ξάνθη για τις ευχάριστες στιγμές που έχουμε περάσει μαζί καθ' όλη τη διάρκεια των σπουδών μου. Τέλος, το μεγαλύτερο ευχαριστώ οφείλω στους γονείς μου Κωνσταντίνο και Αναστασία για τη συμπαράστασή τους όλα αυτά τα χρόνια.

Γαλανόπουλος Σπύρος
Οκτώβριος 2025

Contents

Figure List

Table List

Chapter 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

Η δέσμευση μνήμης είναι μία θεμελιώδης υπηρεσία του Λειτουργικού Συστήματος (ΛΣ), με όλο και μεγαλύτερη επίδραση στην επίδοση και την ενεργειακή αποδοτικότητα σε σύγχρονα υπολογιστικά συστήματα. Ακόμα και όταν τα δεδομένα δε χρειάζεται να ανακτηθούν από τον δίσκο, κάθε δέσμευση μνήμης συνήθως απαιτεί χειρισμό ενός σφάλματος σελίδας (page fault) στο software: ο επεξεργαστής κάνει trap στο ΛΣ, το οποίο έπειτα αντιστοιχίζει μία φυσική σελίδα, ενημερώνει τα page tables, και συνεχίζει την εκτέλεση.

Εντούτοις, η εξ ολοκλήρου υλοποίηση της δέσμευσης φυσικών σελίδων αποκλειστικά σοτ software συνεπάγεται δύο βασικούς περιορισμούς. Πρώτον, καθώς ιστορικά η διαδικασία της δέσμευσης μνήμης μέσω software αντισατθμιζόταν σε workloads μεγάλης διάρκειας, έχει καταστεί *πρώτης τάξης bottleneck* σε σύγχρονα περιβάλλοντα, όπου σύντομης διάρκειας, ευαίσθητες στην καθυστέρηση εφαρμογές κυριαρχούν [1, 2]. Για παράδειγμα, serverless συναρτήσεις, microservices, data analytics, ακόμα και εκφάνσεις inference μεγάλων γλωσσικών μοντέλων (LLMs) (π.χ. σε συστήματα που υποστηρίζουν ενοποιημένη μνήμη μεταξύ CPU και GPU) συχνά εμφανίζουν χρόνους εκτέλεσης μεταξύ microsecond και millisecond, καθιστώντας τις χιλιάδες κύκλους που δαπανώνται στη δέσμευση μνήμης μέσω σφαλμάτων σελίδας ένα σημαντικό ποσοστό του συνολικού χρόνου εκτέλεσης. Οι δραστηριότητες του ΛΣ που αφορούν τη δέσμευση μνήμης μπορεί να αντιπροσωπεύουν περισσότερο από το 30% του χρόνου εκτέλεσης σε τέτοια workloads, και να καταναλώνει μέχρι και 40% της ενέργειας του συστήματος. Δεύτερον, το software του ΛΣ που είναι υπεύθυνο για τη δέσμευση φυσικής μνήμης δεν έχει πρόσβαση σε σημαντική μικροαρχιτεκτονική κατάσταση (π.χ. χρήση εύρους ζώνης της μνήμης, πληρότητα κρυφής μνήμης, συγκρούσεις στο row buffer της DRAM), περιορίζοντας την ικανότητά του να προσαρμόζει τις αποφάσεις τοποθέτησης με βάση τις συνθήκες χρόνου εκτέλεσης.

Προηγούμενες εργασίες έχουν προτείνει επιτάχυνση της δέσμευσης μνήμης, αναθέτοντάς τη σε συγκεκριμένα components στη CPU που χειρίζονται δεσμεύσεις μνήμης απευθείας στο hardware χωρίς να καλούν το ΛΣ. Τέτοιου είδους τεχνικές δέσμευσης αντιμετωπίζουν το overhead λόγω context switches, pipeline flushes, και ακριβής out-of-order εκτέλεσης κατά τη διάρκεια του χειρισμού σφαλμάτων σελίδας, μειώνοντας έτσι την καθυστέρηση για δέσμευση μνήμης και την κατανάλωση ενέργειας.

Εντούτοις, οι υπάρχουσες προτάσεις που αναθέτουν πλήρως στο hardware τη δέσμευση μνήμης έχουν δύο βασικούς περιορισμούς: (i) θέτουν σε προτεραιότητα την ταχύτητα της δέσμευσης σε βάρος της ποιότητας τοποθέτησης, χωρίς να λαμβάνουν υπόψιν τη μικροαρχιτεκτονική κατάσταση ή τους στόχους των ίδιων των εφαρμογών και (ii) υλοποιούν τις πολιτικές δέσμευσης μνήμης σε σταθερά λογικά κυκλώματα, περιορίζοντας την προσαρμοστικότητα σε μεταβαλλόμενα workloads και συνθήκες συστήματος. Πρώτον, παρότι ο χειρισμός της δέσμευσης μνήμης στο hardware μπορεί όντως να μειώσει τα kernel traps και τα context switches, οι υπάρχουσες εργασίες συνήθως προσπαθούν να βελτιστοποιήσουν την ταχύτητα δέσμευσης και όχι την ποιότητα τοποθέτησης. Υπάρχουσε μέθοδοι μέθοδοι βασισμένες στο hardware

ακολουθούν απλές ευριστικές μεθόδους, όπως να επιλέγουν την πρώτη διαθέσιμη ελεύθερη σελίδα, ώστε να ελαχιστοποιούν την καθυστέρηση στο critical path. Εντούτοις, τέτοιου είδους "τυφλή" δέσμευση αγνοεί στόχους τοποθέτησης σε επίπεδο εφαρμογής και συστήματος, οι οποίοι μπορούν να επιβληθούν αν η δέσμευση μνήμης γίνεται σε επίπεδο software και οι οποίοι είναι αναγκαίοι για βελτιστοποίηση της επίδοσης, απομόνωσης, και ενεργειακής αποδοτικότητας. Για παράδειγμα, πολιτικές προσαρμοσμένες σε NUMA αρχιτεκτονικές, χρωματισμός σελίδων, ή tiered memory δεν μπορούν να επιβληθούν αν το hardware πραγματοποιεί τοποθέτηση αυτόνομα, χωρίς να λαμβάνει υπόψιν τη σημασιολογία των προγραμμάτων. Ως αποτέλεσμα, τα οφέλη από τη γρήγορη δέσμευση μνήμης μπορεί να αντισταθμιστούν από υποβέλτιστη τοποθέτηση δεδομένων, η οποία αηξάνει τις συγκρούσεις ή παραβιάζει πολιτικές καταμερισμού (π.χ. καταμερισμό σε επίπεδο bank). Δεύτερον, η υλοποίηση πολιτικών δέσμευσης σε σταθερά κυκλώματα συνεπάγεται ακαμψία. Όταν ένας συγκεκριμένος κανόνας τοποθέτησης ενσωματωθεί στο hardware δυσχεραίνεται σημαντικά η προσαρμογή σε μεταβαλλόμενα workloads, μοτίβα παρεμβολής ή τοπολογίες υλικού. Αυτή η έλλειψη προγραμματισιμότητας εμποδίζει το σύστημα από το να προσαρμόζει τις στρατηγικές δέσμευσης μνήμης σε ποικολόμορφα σενάρια ή εξελισσόμενες ανάγκες εφαρμογών. Συνεπώς, η πλήρης ανάθεση της δέσμευσης φυσικής μνήμης στο hardware με άπληστες πολιτικές θυσιάζει σημαντικές βελτιστοποιήσεις τοποθέτησης και προσαρμοστικότητας, πρειορίζοντας την αποδοτικότητα τέτοιων προσεγγίσεων.

Για να αντιμετωπίσουμε τους περιορισμούς της δέσμευσης μνήμης τόσο αμιγώς στο software όσο και αμιγώς στο hardware, προτείνουμε το Valinor, ένα νέο συνεργατικό σχήμα εκχώρησης μνήμης μεταξύ hardware και ΛΣ που διατηρεί την ευελιξία της δέσμευσης μνήμης στο software, παρέχοντας ταυτόχρονα τη χαμηλή καθυστέρηση (latency) και κατανάλωση ενέργειας των μηχανισμών που βασίζονται στο hardware. Προτείνουμε την προσθήκη ενός hardware μηχανισμού εκχώρησης μνήμης τον οποίο το ΛΣ μπορεί να προγραμματίσει και στον οποίο μπορεί να αναθέτει τα αιτήματα εκχώρησης μνήμης όποτε χρειάζεται, παρακάμπτοντας τον συνήθη χειρισμό του σφάλματος σελίδας στο software. Η σχεδιαστική απόφαση αυτή προσφέρει τρία βασικά οφέλη:

1. **Κέρδη επίδοσης σε επίπεδο hardware.** Ο μηχανισμός που προτείνουμε παρακάμπτει τον αργό χειριστή σφαλμάτων σελίδας στο software και αναθέτει τις δεσμεύσεις μνήμης στο hardware, περιορίζοντας έτσι την καθυστέρηση και κατανάλωση ενέργειας.

2. **Προγραμματισιμότητα και επεκτασιμότητα πολιτικών.** Ο μηχανισμός μας παρέχει μία προγραμματιζόμενη διεπαφή που επιτρέπει στο ΛΣ να ορίζει πολιτικές δέσμευσης μνήμης και μετάφρασης στο software και να τις φορτώνει στο hardware, επιτρέποντας στο hardware να υποστηρίζει διαφορετικές πολιτικές.

3. Πορσαρμοστικότητα στην κατάσταση του hardware. Καθώς ο μηχανισμός στο hardware διαθέτει άμεση ορατότητα σε μικροαρχιτεκτονικές μετρικές, μπορεί να κατευθύνει δυναμικά τις δεσμεύσεις μνήμης ώστε να βελτιστοποιήσει την επίδοση υπό μεταβαλλόμενες συνθήκες.

Αποτελέσματα. Κατασκευάσαμε ένα πρωτότυπο του Valinor σε FPGA, βασιζόμενοι σε έναν πραγματικό RISC-V BOOM πυρήνα που έτρεχε Linux, προκειμένου να αξιολογήσουμε την επίδοσή του. Δοκιμάσαμε διαφορετικές περιπτώσεις χρήσης που επωφελούνται από τις δυνατότητες του Valinor: (i) επιτάχυνση για μικρής διάρκειας workloads, (ii) ενεργειακά αποδοτική δέσμευση μνήμης και τοποθέτηση δεδομένων, (iii) έλεγχοι ακεραιότητας για ασφαλείς μεταφράσεις. Από την αξιολόγησή μας προκύπτουν τα ακόλουθα αποτελέσματα:

Πρώτον, το Valinor επιταχύνει τη δέσμευση σελίδων κατά **7-15**× σε σύγκριση με τον allocator του Linux, περιορίζοντας τα traps του ΛΣ και χειριζόμενο σφάλματα σελίδων αμιγώς στο hardware. Δεύτερον, το Valinor βελτιώνει τον συνολικό χρόνο εκτέλεσης κατά **77%** κατά μέσο όρο. Τρίτον, μέσα από σύνθεση, προκύπτει ότι ο hardware μηχανισμός καταλαμβάνει λίγο χώρο στο chip (1.5% area overhead).

Στην παρούσα διπλωματική, κάνουμε τις ακόλουθες συνεισφορές:

- Προτείνουμε το **Valinor**, ένα νέο συνεργατικό σχήμα εκχώρησης μνήμης μεταξύ hardware και ΛΣ που συνδυάζει την ευελιξία των software allocators με τη χαμηλή καθυστέρηση και ενεργειακή αποδοτικότητα της βασισμένης στο hardware δέσμευσης. Προτείνουμε την προσθήκη ενός hardware μηχανισμού εκχώρησης μνήμης που επιτρέπει στο ΛΣ να αναθέτει επιλεκτικά αιτήματα δέσμευσης

μνήμης στο hardware, περιορίζοντας έτσι ακριβά kernel traps και context switches, διατηρώντας παράλληλα λεπτομερή έλεγχο της τοποθέτησης.

- Σχεδιάζουμε έναν **προγραμματιζόμενο μηχανισμό δέσμευσης μνήμης στο hradware** που βρίσκεται κοντά στην MMU και μπορεί να ρυθμιστεί από το ΛΣ, ώστε να υλοποιεί πολιτικές δέσμευσης και μετάφρασης στο hardware.
- Κατασκευάζουμε και αξιολογούμε ένα FPGA πρωτότυπο βασισμένο σε έναν πυρήνα RISC-V BOOM που τρέχει Linux.

1.2 Σχετική Βιβλιογραφία

1.2.1 Διαχείριση μνήμης στο Hardware

Αρκετά έργα έχουν εξερευνήσει μηχανισμούς για διαχείριση μνήμης στο hardware, εστιάζοντας στη δέσμευση και αποδέσμευση μνήμης [1, 3, 4, 5, 6, 7, 8, 9, 10], ή ακόμα ενσωματώνοντας μηχανισμούς garbage collection [6, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22].

Πολλές από αυτές τις προσεγγίσεις [4, 5, 6, 7] υλοποιούν παραλλαγές του buddy allocator στο hardware, χρησιμοποιώντας συνδυαστικά κυκλώματα για να εντοπίσουν συνεχείς αδέσμευτες περιοχές κατάλληλου μεγέθους. Ο Kanev κ.ά. [3], από την άλλη, σχεδιάζουν το Mallacc, έναν επιταχυντή για χειρισμό μικρών δεσμεύσεων μνήμης μέσω της συνάρτησης `malloc()` στο hardware. Αξιοποιούν το γεγονός ότι η `malloc()` συχνά χρησιμοποιεί free lists για κλάσεις διαφορετικών μεγεθών (size classes), και υλοποιούν μία μονάδα hardware που επιτρέπει γρήγορο εντοπισμό κλάσης μεγέθους και ενημερώσεις στην αντίστοιχη free list στη συνήθη περίπτωση. Ο Wang κ.ά. επεκτείνουν αυτή την ιδέα στο Memento [1], χρησιμοποιώντας διαχειριζόμενες από το hardware arenas για κάθε κλάση μεγέθους, οι οποίες εκχωρούνται από το Λειτουργικό Σύστημα (ΛΣ) σε επίπεδο (granularity) σελίδας. Με τον τρόπο αυτό καθιστούν εφικτή την εκχώρηση μνήμης μέσω hardware τόσο σε επίπεδο χρήστη (userspace) όσο και σε επίπεδο πυρήνα (kernelspace).

Οι εργασίες αυτές ασχολούνται με την εκχώρηση μνήμης σε επίπεδο αντικειμένου (object), και ως εκ τούτου περιορίζονται σε εκχώρηση μικρών μεγεθών μνήμης. Επιπρόσθετα, απαιτούν τροποποιήσεις στο instruction set, προκειμένου να μπορούν να ενημερώνουν το hardware για ένα αίτημα για εκχώρηση ή αποδέσμευση μνήμης. Το πλαίσιο της παρούσας εργασίαςείναι διαφορετικό, καθώς ασχολούμαστε με εκχώρηση μνήμης σε granularity σελίδας, και στηριζόμαστε σε hardware events που συμβαίνουν σε υπάρχοντα τμήματα της CPU (και συγκεκριμένα στη μονάδα διαχείρισης μνήμης (MMU)) για ενεργοποίηση του μηχανισμού που προτείνουμε.

1.2.2 Σελιδοποίηση (Paging) στο Hardware

Πολλαπλά έργα [23, 2, 24] σχεδιάζουν μηχανισμούς συγκεκριμένα για τη διαχείριση σφαλμάτων σελίδας (page faults) στο hardware. Οι προσεγγίσεις αυτές περιορίζουν το κόστος της αντιστοίχισης μνήμης (memory mapping) μετακινώντας τμήματα του χειρισμού του page fault εκτός του critical path του page fault, σε ένα νήμα στο παρασκήνιο, και προσθέτοντας hardware για τη δημιουργία της αντιστοίχισης μεταξύ εικονικής και φυσικής μνήμης. Αυτό απαιτεί τη διατήρηση ενός memory pool από αδέσμευτες σελίδες στη φυσική μνήμη από τις οποίες το hardware μπορεί να δεσμεύσει γρήγορα τη στιγμή του page fault.

Ο Lee κ.ά. [2] προτείνουν τη διαχείριση των major page faults στο hardware. Η κεντρική ιδέα συνίσταται στην προσθήκη ενός hardware μηχανισμού που εκδίδει εντολές I/O για την προσκόμιση των κατάλληλων blocks από τον δίσκο στη μνήμη και αντιστοιχίζει εικονική μνήμη σε φυσική χωρίς τη μεσολάβηση του ΛΣ. Η θέση του επιθυμητού block διατηρείται ως μεταδεδομένα στο Page Table Entry (PTE) που αντιστοιχεί στην εικονική διεύθυνση, το οποίο δημιουργείται από το OS κατά την `mmap()`. Η φυσική σελίδα που χρησιμοποιείται ως προορισμός για τα δεδομένα από τον δίσκο επιλέγεται από μία FIFO ουρά, η οποία συστηματικά ανανεώνεται με νέες αδέσμευτες σελίδες από ένα νήμα του ΛΣ στο παρασκήνιο.

Ο Tirumalasetty κ.ά [23] μετέρχονται παρόμοιες ιδέες, εστιάζοντας όμως στην επιτάχυνση των minor page faults. Προτείνουν απλοποίηση του critical path του page fault μετακινώντας τις περισσότερες λειτουργίες του χειρισμού του page fault σε ένα παρασκηνιακό νήμα, διατηρώντας μόνο την ανάθεση μίας αδέσμευτης φυσικής σελίδας στην επιθυμητή εικονική σελίδα τη στιγμή του page fault, πράγμα το οποίο μπορεί να πραγματοποιηθεί στο hardware. Για τον σκοπό αυτό, διατηρούν ένα FIFO ring buffer με αδέσμευτες φυσικές σελίδε,ς οι οποίες ανανεώνονται με μηδενισμένες σελίδες από ένα παρασκηνιακό task του ΛΣ, και από το οποίο το hardware δεσμεύει μία σελίδα κάθε φορά που συναντά εικονική σελίδα χωρίς αντιστοίχιση. Προκειμένου το hardware να μπορεί να αποφασίσει αν χρειάζεται να διαχειριστεί το ίδιο ένα page fault, κατασκευάζουν όλα τα Page Table Entries για την επιθυμητή περιοχή εικονικής μνήμης κατά την κλήση της `mmap()`, και θέτουν ένα συγκεκριμένο bit στο PTE για να ενημερώνουν τον Page Table Walker ότι ένα page fault πρέπει να το χειριστεί το hardware. Επιπλέον, αναθέτουν την επικαιροποίηση των σχετικών δομών του ΛΣ (π.χ. προσθήκη των σελίδων σε λίστα LRU, δημιουργία αντίστροφων αντιστοιχίσεων) στο ίδιο το ΛΣ αργότερα, εκτός του critical path.

Τέλος, παρόμοιοι μηχανισμοί χρησιμοποιούνται στο [24] και στο [25], εντούτοις τα συγκεκριμένα έργα εστιάζουν σε συστήματα disaggregated μνήμης.

Παρότι οι συγκεκριμένες λύσεις μπορούν να μειώσουν σημαντικά το χρονικό κόστος διαχείρισης page faults, έχουν πολλαπλά μειονεκτήματα. Συγκεκριμένα:

1. Δεσμεύουν πόρους του συστήματος για την εκτέλεση παρασκηνιακών νημάτων
2. Δεν παρέχουν δυνατότητα επιλογής της φυσικής σελίδας στην οποία αντιστοιχίζεται μία εικονική διεύθυνση. Αντίθετα, το hardware πάντα επιλέγει αυθαίρετα την πρώτη διαθέσιμη σελίδα από μία ουρά που χειρίζεται το software
3. Τα page tables για τις περιοχές εικονικής μνήμης τις οποίες διαχειρίζεται το hardware πρέπει να δημιουργούνται εκ των προτέρων (π.χ. κατά την εκτέλεση της `mmap()`), προκειμένου το hardware να αποφασίζει αν πρέπει να χειριστεί το ίδιο το page fault ή να το αναθέσει στον πυρήνα του ΛΣ

Αντίθετα, ο μηχανισμός που προτείνουμε αναθέτει *ολόκληρη* τη διαδικασία εκχώρησης μνήμης στο hardware, παρέχοντας παράλληλα ευελιξία στην αντιστοίχιση. Επιπλέον, το hardware διαπιστώνει ποιες σελίδες πρέπει να χειριστεί μέσω ενός μηχανισμού που διασχίζει μέσα στο hardware τις δομές του ΛΣ για περιοχές εικονικής μνήμης, εξαλείφοντας έτσι την ανάγκη για εκ των προτέρων δημιουργία του page table.

1.2.3 Επαναπρογραμματιζόμενο Υλικό

Αρκετά έργα προτείνουν την προσθήκη επαναπρογραμματιζόμενου υλικού κοντά στον επεξεργαστή. Μία συνήθης προσέγγιση είναι η τοποθέτηση μίας προγραμματιζόμενης μονάδας γενικού σκοπού κοντά στον επεξεργαστή, [26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38], ενώ άλλες εργασίες εστιάζουν σε συγκεκριμένες μονάδες ή λειτουργίες, όπως στον memory controller citebojnordi2012pardis, carter1999impulse, martin2009mmcdataflow, kornaros2003mmoptqueue, firoozshahian2009smartmemories, kunbin2005qualitymc, στις caches και στα directories [39, 40, 41, 42, 43], στο prefetching citeprodigyHPCA2021, yuan2021qei, ainsworth2018event, στην ασφάλεια [44, 45] ή στην επιτάχυνση αλγορίθμων συμπίεσης [46].

Ορισμένες από αυτές τις εργασίες βασίζονται σε events, ενεργοποιώντας τις επαναπρογραμματιζόμενες μονάδες όταν παρατηρούνται συγκεκριμένα μικροαρχιτεκτονικά γεγονότα. Για παράδειγμα, ο Ainsworth κ.ά [47] προτείνουν την ενσωμάτωση μικρών, in-order, προγραμματιζόμενων πυρήνων που λειτουργούν σε χαμηλή συχνότητα για τη δημιουργία αιτημάτων prefetching. Εκτελούν μικρά προγράμματα που ενεργοποιούνται κατά την πρόσβαση σε δεδομένα σε συγκεκριμένες περιοχές μνήμης, ή όταν prefetched δεδομένα φτάνουν στην L1 cache. Ο Schwedock κ.ά επεκτείνουν την ιδέα του κώδικα που ενεργοποιείται κατά τη μετακίνηση δεδομένων στο [43]. Προτείνουν την επαύξηση της διεπαφής hardware-software μέσω συναρτήσεων που ενεργοποιούνται όταν παρατηρούνται συγκεκριμένα events στις caches (π.χ. cache miss, eviction ή writeback), και εκτελούνται σε επαναπρογραμματιζόμενες μονάδες στο hardware. Αυτό επιτρέπει ελαφριά επεξεργασία δεδομένων (π.χ. συμπίεση) κατά την εισαγωγή στην cache ή όταν δεδομένα γράφονται από την cache στην κύρια μνήμη.

Στην παρούσα εργασία υιοθετούμε μία ανάλογη προσέγγιση με τις προαναφερθείσες εργασίες, εκτελώντας συναρτήσεις σε μία επαναπρογραμματιζόμενη μονάδα, με τη διαφορά ότι εστιάζουμε σε events που προκαλούνται από το υποσύστημα εικονικής μνήμης (oage faults και μεταφράσεις διεύθυνσης).

1.2.4 Βελτιστοποιήσεις στη Μετάφραση Διεύθυνσης (Address Translation)

Εναλλακτικά σχήματα αντιστοίχισης εικονικής μνήμης σε φυσική έχουν μελετηθεί σε μεγάλο βαθμό από προηγούμενες εργασίες. Συγκεκριμένα, έχουν αξιοποιηθεί μεγάλες σελίδες [48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63], συνέχεια στο εικονικό address space [64, 65, 66, 67, 68, 69, 70, 71, 72], μετάφραση που βασίζεται σε κατακερματισμό (hashing) της εικονικής διεύθυνσης[73, 74, 75], καθώς και εναλλακτικές μορφές για το page table, με στόχο την επιτάχυνση του addresstranslation [76, 77, 78, 79, 80, 81, 74, 75, 82, 83, 84, 85, 86]. Η κοντινότερη στη δική μας από αυτές τις εργασίες είναι η [86], που προτείνει μία αρχιτκεκτονική που επιτρέπει την προσαρμογή του πίνακα μετάφρασης από εικονικές σε φυσικές διευθύνσεις. Εντούτοις, η συγκεκριμένη εργασία (i) περιορίζεται στη μετάφραση, με το ΛΣ να χειρίζεται τα page faults, (ii) αναθέτει τη μετάφραση σε ένα παρασκηνιακό νήμα. Αντίθετα, ο μηχανισμός που προτείνουμε στην παρούσα εργασία παρέχει ευελιξία τόσο στη μετάφραση όσο και στην αντιστοίχιση μεταξύ εικονικών και φυσικών σελίδων.

1.3 Motivation

Παραδοσιακές εφαρμογές όπως επιστημινικές προσομοιώσεις, βάσεις δεδομένων και μεγάλης κλίμακας ανάλυση σε γράφους είναι συνήθως μεγάλες σε διάρκεια και παρουσιάζουν μεγάλα και σταθερά αποτυπώματα μνήμης και ακανόνιστα μοτίβα προσβάσεων σε μεγάλα dataset. Για τέτοια workloads, τα overheads σε καθυστέρηση και ενέργεια για λειτουργίες όπως τα minor σφάλματα σελίδων και τα context switches αντισταθμίζονται από τον μεγάλο αριθμό εντολών για υπολογισμούς, καθώς και από τον παραλληλισμό σε επίπεδο μνήμης.

Αντίθετα, τα τελευταία χρόνια υπάρχει μία στροφή προς *μικρής διάρκειας, ευαίσθητα στην καθυστέρηση workloads*. Σε αυτά συμπεριλαμβάνονται *serverless συναρτήσεις*, *microservices*, και *inference Μεγάλων Γλωσσικών Μοντέλων (LLM)*, που εκτελούνται σε bursts, πραγματοποιούν ελάχιστους υπολογισμούς ανά δεδομένο, και συχνά τερματίζουν εντός millisecond. Αυτά τα workloads δεσμεύουν και απελευθερώνουν μνήμη συχνά και απρόβλεπτα. Ως αποτέλεσμα, κάθε εφαρμογή υφίσταται σημαντική καθυστέρηση κατά την εκκίνηση εξαιτίας των minor σφαλμάτων σελίδας κια τις ρουτίνες δέσμευσης φυσικής μνήμης του πυρήνα.

Για να ποσοτικοποιήσουμε το μέγεθος του προβλήματος αυτού, πραγματοποιήσαμε λεπτομερείς μετρήσεις καθυστέρησης και ενέργειας σε ένα Intel Xeon σύστημα, χρησιμοποιώντας ένα εργαλείο βασισμένο στο eBPF και σε μετρητές RAPL της Intel. Στα Σχήματα 1.3.1 και 1.3.2 φαίνεται το ποσοστό του συνολικού χρόνου εκτέλεσης και ενέργειας συστήματος που καταναλώνεται σε ρουτίνες δέσμευσης μνήμης για διάφορες εφαρμογές. Καθίσταται σαφές ότι, κατά μέσο όρο, το **32% του συνολικού χρόνου εκτέλεσης** and **21% της ενέργειας συστήματος** καταναλώνονται αποκλειστικά από τις ρουτίνες δέσμευσης φυσικής μνήμης.

Αρκετές εργασίες έχουν ασχοληθεί με διαχείριση των σφαλμάτων σελίδας στο hardware. Οι προτάσεις αυτές εισάγουν σταθερές hardware μηχανές πεπερασμένης κατάστασης που επιταχύνουν ορισμένες ρουτίνες διαχείρισης μνήμης. Παρότι αυτή η ανάθεση στο hardware μπορεί να μειώσει σημαντικά την καθυστέρηση των σφαλμάτων σελίδας, αυτό το επιτυγχάνει αφαιρώντας το ΛΣ από το critical path, θυσιάζοντας έτσι την ευελιξία και προγραμματισιμότητα στην οποία στηρίζονται τα σύγχρονα συστήματα.

Οι παρατηρήσεις αυτές αναδεικνύουν μία ευκαιρία για αναθεώρηση της δέσμευσης φυσικής μνήμης. Τα συστήματα του μέλλοντος απαιτούν συνεργατική μεταξύ hardware και software δέσμευση μνήμης, που συνδυάζει τη γρήγορη εκτέλεση στο hardware με την προσαρμοστικότητα του software.

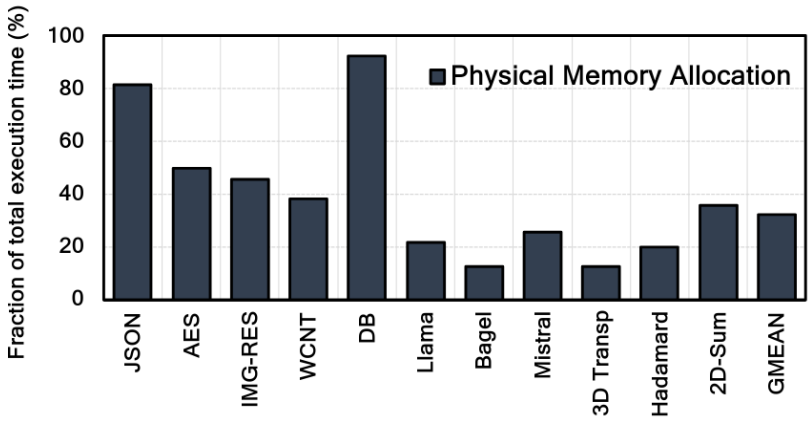


Figure 1.3.1: Ποσοστό του χρόνου εκτέλεσης που δαπανάται σε ρουτίνες δέσμευσης φυσικής μνήμης.

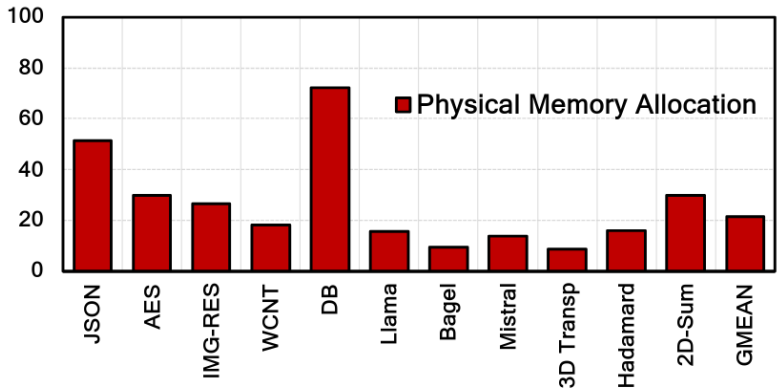


Figure 1.3.2: Ποσοστό της συνολικής ενέργειας συστήματος που δαπανάται σε ρουτίνες δέσμευσης φυσικής μνήμης.

1.4 Μηχανισμός εκχώρησης μνήμης στο Hardware

Στην παρούσα εργασία προτείνουμε το **Valinor**, ένα νέο συνεργατικό σχήμα εκχώρησης μνήμης μεταξύ hardware και ΛΣ που διατηρεί την ευελιξία της δέσμευσης μνήμης στο software, παρέχοντας ταυτόχρονα τη χαμηλή καθυστέρηση (latency) και κατανάλωση ενέργειας των μηχανισμών που βασίζονται στο hardware. Προτείνουμε την προσθήκη ενός hardware μηχανισμού εκχώρησης μνήμης τον οποίο το ΛΣ μπορεί να προγραμματίσει και στον οποίο μπορεί να αναθέτει τα αιτήματα εκχώρησης μνήμης όποτε χρειάζεται, παρακάμπτοντας τον συνήθη χειρισμό του σφάλματος σελίδας στο software. Η σχεδιαστική απόφαση αυτή προσφέρει τρία πλεονεκτήματα σε σχέση με προσεγγίσεις που βασίζονται αμιγώς στο software ή το hardware:

1. **Πλεονεκτήματα της εκχώρησης μνήμης στο hardware.** Σε μικρής διάρκειας ή ευαίσθητες στην καθυστέρηση (latency-critical) εφαρμογές, όπως είναι για παράδειγμα οι serverless συναρτήσεις, τα microservices, ή το LLM inference, ο χειρισμός των σφαλμάτων σελίδας μέσω του ΛΣ προκαλεί ακριβά context switches. Αυτά εισάγουν stalls στο pipeline του επεξεργαστή, περιορίζουν τον παραλληλισμό και σπαταλούν ενέργεια σε Out-of-Order πυρήνες. Το Valinor επιτρέπει στο ΛΣ να παρακάμψει αυτή τη διαδικασία αναθέτοντας εκχωρήσεις μνήμης σε έναν μηχανισμό του hardware, μειώνοντας σημαντικά τόσο την καθυστέρηση όσο και την κατανάλωση ενέργειας

2. **Έξυπνες πολιτικές εκχώρησης μνήμης.** Πολλές προηγμένες βελτιστοποιήσεις της δι-

αχείρισης μνήμης, όπως για παράδειγμα ο χωρματισμός σελίδων, η NUMA-aware τοποθέτηση, η δέσμευση guard rows, ή μικρής διάρκειας memory pools, απαιτούν προσαρμογή της αντιστοίχησης εικονικής μνήμης σε φυσική. Οι υπάρχοντες μηχανισμοί hardware πραγματοποιούν τυφλά αυτές τις αντιστοιχίσεις και δεν μπορούν να εκφράσουν τέτοιες πολιτικές. Το Valinor παρέχει στο ΛΣ τη δυνατότητα να προγραμματίζει κανόνες τοποθέτησης στον μηχανισμό. Με τον τρόπο αυτό επιτρέπει γρήγορη εκχώρηση, διατηρώντας όμως λεπτομερή έλεγχο της συμπεριφοράς της μετάφρασης

3. **Πολιτικές εκχώρησης μνήμης προσαρμοσμένες σε δυναμικές συνθήκες στο hardware.** Καθώς ο μηχανισμός στο hardware έχει άμεση πρόσβαση στην μικροαρχιτεκτονική κατάσταση (π.χ. εύρος ζώνης της DRAM, περιεχόμενα της κρυφής μνήμης, αντιστοιχίσεις διευθύνσεων), το Valinor μπορεί να προσαρμόσει την τοποθέτηση μνήμης με βάση ανατροφοδότηση κατά τη διάρκεια της εκτέλεσης. Μπορεί για παράδειγμα να δεσμεύσει σελίδες μακριά από DRAM banks με υψηλή συμφόρηση, να διαμερίσει δυναμικά την κρυφή μνήμη, ή να εξισορροπήσει τις δεσμεύσεις μνήμης σε διαφορετικούς τομείς NUMA. Αυτό επιτρέπει προσαρμοζόμενες στρατηγικές δέσμευσης μνήμης, των οποίων το κόστος υλοποίησης αμιγώς στο software θα ήταν απαγορευτικά μεγάλο.

Στο Σχήμα 1.4.1 παρουσιάζεται η λειτουργία του Valinor, καθώς και η ενσωμάτωσή του με το ΛΣ. Σε πρώτο στάδιο, η εφαρμογή εκφράζει τις υψηλού επιπέδου σημασιολογικές απαιτήσεις (π.χ. ότι οι περισσότερες δεσμεύσεις μνήμης αφορούν αντικείμενα με μικρή διάρκεια ζωής), χρησιμοποιώντας κάποια προσαρμοσμένη βιβλιοθήκη για δέσμευση μνήμης (π.χ. tcmalloc [87], jemalloc [**jemalloc**], ή mimalloc [**mimalloc**]). Το ΛΣ επιλέγει την κατάλληλη βιβλιοθήκη με βάση τις ανάγκες της εφαρμογής (π.χ. μία βιβλιοθήκη για εκχώρηση μνήμης για latency-critical εφαρμογές). Το ΛΣ ρυθμίζει τον μηχανισμό στο hardware, ώστε να υλοποιεί την πολιτική allocation της βιβλιοθήκης (π.χ. ενημερώνει έναν pointer ώστε ο hardware μηχανισμός να μπορεί να φορτώσει τη βιβλιοθήκη). Στη συνέχεια, όταν η εφαρμογή πραγματοποιεί κάποια πρόσβαση στη μνήμη και η MMU του πυρήνα διαπιστώνει ένα TLB mis, και η μετάφραση από εικονική μνήμη σε φυσική για τη συγκεκριμένη διεύθυνση δεν υπάρχει στο Page Table, ο Page Table Walker (PTW) του πυρήνα προκαλεί εξαίρεση, καθώς η αντιστοίχιση είτε δεν έχει δημιουργηθεί ακόμα ή έχει δημιουργηθεί από τον hardware μηχανισμό (που χρησιμοποιεί δικές του δομές για να διατηρεί). Έπειτα, ο πυρήνας πρώτα ανακαλύπτει τις ιδιότητες της περιοχής μνήμης στο οποίο επιχειρείται πρόσβαση, όπως π.χ. αν αντιστοιχεί σε αρχείο (file-backed) ή είναι ανώνυμη, καθώς και τα δικαιώματα πρόσβασης. Αν το αντικείμενο μνήμης ικανοποιεί τα κριτήρια για εκχώρηση μνήμης στο hardware, ο πυρήνας ενεργοποιεί τον hardware μηχανισμό. Ο μηχανισμός πρώτα ελέγχει αν έχει ήδη εκχωρηθεί η ζητούμενη σελήδα (π.χ. σε προηγούμενη πρόσβαση). Σε αυτή την περίπτωση, επιστρέφει τον αντίστοιχο αριθμό φυσικής σελίδας (Page frame number, PFN) και τα δικαιώματα πρόσβασης στον πυρήνα, ολοκληρώνοντας τη μετάφραση σελίδας. Ειδάλλως, δεσμεύει μια καινούρια φυσική σελίδα σύμφωνα με την πολιτική που έχει προγραμματιστεί να υλοποιεί (π.χ. από μία περιοχή μνήμης συγκεκριμένα για αντικείμενα με μικρή διάρκεια ζωής), ενημερώνει τις εσωτερικές δομές δεδομένων του, και επιστρέφει την καινούρια αντιστοίχιση στον πυρήνα. Ο πυρήνας έπειτα ενημερώνει το TLB με τη νέα αντιστοίχιση και συνεχίζει την εκτέλεση. Αν το αντικείμενο μνήμης δεν ικανοποιεί τα κριτήρια (π.χ. αντιστοιχεί σε αρχείο), ο πυρήνας επιστρέφει στον συνήθη χειριστή σφάλματος σελίδας του ΛΣ. Τέλος, όταν η εφαρμογή αποδεσμεύει μνήμη, το ΛΣ πληροφορεί τον μηχανισμό, ώστε να ενημρεώσει κατάλληλα τις εσωτερικές δομές δεδομένων του.

1.4.1 Προγραμματιζόμενος μηχανισμός στο hardware

Το κύριο τμήμα του Valinor είναι ένας προγραμματιζόμενος μηχανισμός εκχώρησης μνήμης στο hardware, στόχος του οποίου είναι να χειρίζεται τη δέσμευση μνήμης (και, ενδεχομένως, τη μετάφραση). Όπως διακρίνεται στην Εικόνα [], βρίσκεται κοντά στην L2 cache και ελέγχεται από τον PTW.

Προκειμένου να παρέχει την επιθυμητή λειτουργικότητα, ο μηχανισμός εκχώρησης μνήμης πρέπει να μπορεί να υλοποιήσει τις ακόλουθες συναρτήσεις:

1. `page_fault_handler(pid, vpn, prot)`: Δεσμεύει μία φυσική σελίδα για εικονική διεύθυνση με συγκεκριμένο `vpn` και δικαιώματα πρόσβασης `prot` για μία διεργασία με συγκεκριμένο `pid`
2. `address_translation_handler(pid, vpn)`: Ελέγχει αν η εικονική μνήμη με αριθμό εικονικής μνήμης `vpn` έχει αντιστοιχιστεί από τον hardware μηχανισμό (αλλά δεν υπάρχει στο Page Table

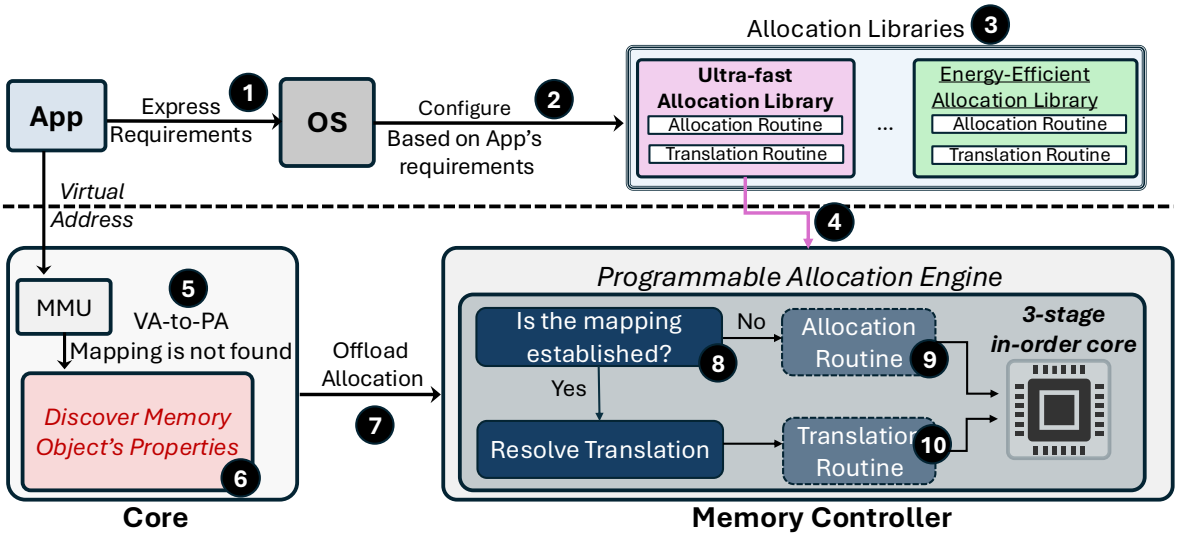


Figure 1.4.1: Η αρχιτεκτονική του Valinor και αλληλεπίδρασή του με το ΛΣ. Το ΛΣ μπορεί να προγραμματίσει τον hardware μηχανισμό ώστε να χειρίζεται απευθείας αιτήματα εκχώρησης μνήμης, προσπερνώντας τον συνήθη χειρισμό στο software

του ΛΣ) για μία διεργασία `pid`. Στην περίπτωση αυτή, επιστρέφει το αντίστοιχο PFN μαζί με τα δικαιώματα πρόσβασης

3. `free(pid, vpn_start, vpn_end)`: Διαγράφει όλες τις αντιστοιχίσεις για σελίδες με `vpm` στη ζητούμενη περιοχή για μία διεργασία `pid`

4. `ioctl(info)`: Χρησιμοποιείται κατά την αρχικοποίηση της βιβλιοθήκης (π.χ. ενημέρωση του hardware μηχανισμού για παραμέτρους της βιβλιοθήκης)

Οι πρώτες δύο λειτουργίες ενεργοποιούνται από συγκεκριμένα γεγονότα στο hardware, ενώ οι `free` και `ioctl` ενεργοποιούνται από το software. Για τον λόγο αυτό, το ISA πρέπει να επεκταθεί με δύο νέες εντολές.

1.4.2 Φίλτρο περιοχών εικονικής μνήμης

Όπως προαναφέρθηκε, ο μηχανισμός εκχώρησης μνήμης χρειάζεται πληροφορία για τα δικαιώματα πρόσβασης της περιοχής εικονικής μνήμης (Virtual Memory Area, VMA) στην οποία ανήκει η εικονική διεύθυνση. Επιπρόσθετα, ανάλογα με την υλοποίηση του μηχανισμού εκχώρησης μνήμης, ενδέχεται να υπάρχει ανάγκη να περιορίζονται τα σφάλματα σελίδας που αυτός χειρίζεται. Για παράδειγμα, προς αποφυγήν περίπλοκων αιτημάτων DMA, ορισμένες υλοποιήσεις του μηχανισμού μπορεί να περιορίζονται σε minor σφάλματα σελίδας. Για τους λόγους αυτούς, προτείνουμε τη χρήση ενός hardware component, του φίλτρου περιοχών εικονικής μνήμης (Virtual Memory Area Filter, VMAF), ο στόχος του οποίου είναι διττός:

1. Να φιλτράρει τις εικονικές διευθύνσεις για τις οποίες πρέπει να ενεργοποιηθεί ο μηχανισμός εκχώρησης μνήμης

2. Για διευθύνσεις τις οποίες πρέπει να χειριστεί ο μηχανισμός μνήμης, να βρίσκει τα δικαιώματα πρόσβασης της αντίστοιχης VMA

Ο ακριβής σχεδιασμός του VMAF εξαρτάται από τις ανάγκες του σχεδιασή του συστήματος, καθώς και παό τον συγκεκριμένο τρόπο με τον οποίο οι πληροφορίες για τις VMA διατηρούνται στο ΛΣ. Για παράδειγμα, σε εκδόσεις του πυρήνα του Linux πριν την 6.1, τα `vm_area_structs` περιέχονται σε ένα red-black δέντρο ανά διεργασία. Συνεπώς, το VMAF πρέπει να μπορεί να πραγματοποιεί διάσχιση δυαδικού δέντρου για να ανακαλύπτει τις επιθυμητές ιδιότητες.

Το VMAF ενεργοποιείται από τον PTW κατά μία αποτυχημένη απόπειρα μετάφρασης διεύθυνσης.

1.4.3 Υποστήριξη σε επίπεδο ΛΣ και Software

Όταν ένα userspace πρόγραμμα χρειάζεται να χρησιμοποιήσει το Valinor με μία συγκεκριμένη βιβλιοθήκη, ο προγραμματιστής πρέπει να ενημερώσει το σύστημα για τη συγκεκριμένη βιβλιοθήκη που χρησιμοποιείται, έτσι ώστε το ΛΣ να μπορεί να προβεί στις κατάλληλες ενέργειες αρχικοποίησης. Για τον σκοπό αυτό, προσθέτουμε τρεις νέες κλήσεις συστήματος (syscalls), και συγκεκριμένα τις `valinor_install_lib(lib)`, `valinor_start()` και `valinor_end()`. Η πρώτη από αυτές ενημερώνει το ΛΣ ότι μία συγκεκριμένη βιβλιοθήκη πρέπει να χρησιμοποιηθεί από τον hardware μηχανισμό, ενώ οι άλλες δύο σηματοδοτούν την αρχή και το τέλος αντίστοιχα μιας περιοχής κώδικα που πρέπει να χειριστεί το Valinor.

Για την υλοποίηση των κλήσεων συστήματος αυτών, το ΛΣ πρέπει να παρέχει στον μηχανισμό hardware δύο πληροφορίες, και συγκεκριμένα:

1. Μεταδεδομένα της βιβλιοθήκης που χρησιμοποιείται. Για παράδειγμα, αν η βιβλιοθήκη δε χρησιμοποιεί τις δικές της δομές δεδομένων για μετάφραση, αλλά αντίθεται βασίζεται στο Page Table του ΛΣ, τότε ο μηχανισμός χρειάζεται έναν δείκτη στη ρίζα του Page Table
2. Δεδομένα ανεξάρτητα της βιβλιοθήκης, και συγκεκριμένα τη θέση της βιβλιοθήκης στη μνήμη και ένα σήμα που να ενεργοποιεί και να απενεργοποιεί τον μηχανισμό

Τα μεταδεδομένα της βιβλιοθήκης παρέχονται κατά την κλήση της `valinor_install_lib()`, μέσω κλήσεων στην εντολή `ioctl` του Valinor. Η θέση της βιβλιοθήκης και το σήμα ενεργοποίησης, από την άλλη, παρέχονται μέσω ειδικού σκοπού memory-mapped καταχωρητών.

1.4.4 Περιπτώσεις Χρήσης

Στον Πίνακα 1.1 συνοψίζονται οι διαφορετικές περιπτώσεις χρήσης που μελετώνται στην παρούσα εργασία.

Βιβλιοθήκη για γρήγορη δέσμευση (Latency-First Allocation Library, LFA)

Ο βασικός στόχος της βιβλιοθήκης LFA είναι να θέσει σε προτεραιότητα την πολύ γρήγορη δέσμευση μνήμης. Προς αυτή την κατεύθυνση, χρησιμοποιούμε ένα σχήμα δέσμευσης μνήμης με βάση τον κατακερματισμό σε ένα συγκεκριμένο συνεχόμενο allocation pool σελίδων, το οποίο χρησιμοποιείται αποκλειστικά από τον hardware μηχανισμό για δέσμευση μνήμης.

Δέσμευση μνήμης: Ο προγραμματιστής της βιβλιοθήκης ορίζει μία συνάρτηση κατακερματισμού (hash function) H που αντιστοιχίζει διευθύνσεις σε θέσεις στο allocation pool. Κατά τη λήψη ενός αιτήματος μετάφρασης μίας εικονικής διεύθυνσης με αριθμό εικονικής διεύθυνσης (Virtual Page Number) VPN, ο hardware μηχανισμός υπολογίζει την τιμή $n = H(VPM)$ προκειμένου να αντιστοιχίσει την εικονική διεύθυνση σε μία συγκεκριμένη σελίδα στο allocation pool. Έπειτα ο allocator ελέγχει την n-ή θέση στο `tag_array` ώστε να διαπιστώσει αν υπάρχει έγκυρο VPN που έχει ήδη αντιστοιχιστεί σε αυτή τη σελίδα. Αν η σελίδα είναι ελεύθερη, τότε ο allocator ενημερώνει την n-ή θέση στο `tag_array` με το VPN και επιστρέφει το PPN που αντιστοιχεί στην n-ή θέση στο allocation pool. Ειδάλλως, η δέσμευση μνήμης θεωρείται ανεπιτυχής και προκαλείται σφάλμα σελίδας.

Μετάφραση Διεύθυνσης: Καθώς το Page Table δεν ενημερώνεται από τον χειριστή σφαλμάτων σελίδας, η διάσχιση του radix tree στον Page Table Walker θα αποτυγχάνει πάντα για σελίδες στο allocation pool. Για τις σελίδες αυτές, λοιπόν, ενεργοποιείται ο χειριστής μετάφρασης σελίδας του ίδιου του hardware μηχανισμού. Για μία σελίδα με αριθμό εικονικής σελίδας VPN, ο hardware μηχανισμός υπολογίζει και πάλι τη συνάρτηση $n = H(VPN)$ και κοιτάζει τη συγκεκριμένη θέση στο `tag_array`. Αν το VPN που περιέχεται στο `tag_array[n]` ταυτίζεται με το VPN, τότε η μετάφραση σελίδας κρίνεται επιτυχής και ο hardware μηχανισμός επιστρέφει το PPN που αντιστοιχεί στην n-ή θέση στο allocation pool. Ειδάλλως, προκύπτει και πάλι σφάλμα σελίδας.

Το allocation pool μπορεί να δεσμευτεί από το ΛΣ κατά την εκκίνηση (π.χ. μέσω `memblock_reserve()`) και ο hardware μηχανισμός μπορεί να ενημερωθεί για αυτό μέσω της συνάρτησης `ioctl` που περιγράφηκε παραπάνω (1.4.1).

Το σχήμα δέσμευσης μνήμης αυτό αποφεύγει τα overheads από α) τις ενημερώσεις του Page Table, β) το περίπλοκο buddy system, με κόστος όμως αυξημένο overhead για μετάφραση σελίδας. Επιπρόσθετα, η επιλογή να χρησιμοποιείται ένα απομονωμένο allocation pool εξαλείφει την ανάγκη για σύνθετο συγχρονισμό μεταξύ του ΛΣ και του hardware μηχανισμού, καθώς και συνάφεια (coherence), σε περίπτωση που ο hardware μηχανισμός διαθέτει cache.

Παρακάτω παρατίθεται ψευδοκώδικας για τη βιβλιοθήκη LFA:

```
PPN translate(VPN v):
  for (int i=0; i< n_hashes; i++):
    Set s = hash_i(v) & (NUM_SETS-1);
    if (seg[s].tag_match(v)) return seg[s].ppn_of(v);
  return allocate(v); // allocate on miss

PPN allocate(VPN v):
  for (int i=0; i< n_hashes; i++):
    Set s = hash_i(v) & (NUM_SETS-1);
    if (seg[s].has_free()) return seg[s].pop_free();
  return FALLBACK; // rare slow path
```

Listing 1.1: LFA: δέσμευση μνήμης και μετάφραση

1.5 Πειραματική αξιολόγηση

1.5.1 Πειράματα

Πρωτότυπο σε RISC-V

Κατασκευάσαμε ένα πρωτότυπο του προτεινόμενου μηχανισμού, για την πειραματική αξιολόγηση του οποίου χρησιμοποιήσαμε ένα ZCU 106 FPGA της Xilinx [88]. Στόχος μας ήταν (i) να αποδείξουμε τη δυνατότητα υλοποίησης του μηχανισμού μας και (ii) να αξιολογήσουμε τα οφέλη σε επίδοση από τη χρήση του μηχανισμού σε ένα πραγματικό σύστημα. Στον Πίνακα 1.2 συνοψίζονται τα τεχνικά χαρακτηριστικά του πρωτοτύπου.

Επιλέξαμε να υλοποιήσουμε τη βιβλιοθήκη γρήγορης εκχώρησης μνήμης (LFA), όπως περιγράφηκε παραπάνω, καθώς η απουσία επικοινωνίας μεταξύ του hardware μηχανισμού και του ΛΣ καθιστούσε την υλοποίηση απλούστερη. Επιπλέον, προς αποφυγή σύνθετων I/O ενεργειών, καθώς και αναζητήσεων στην page cache, περιοριστήκαμε στον χειρισμό ανώνυμων minor σφαλμάτων σελίδας.

Για να μοντελοποιήσουμε την επαναπρογραμματιζόμενη μονάδα, χρησιμοποιήσαμε τον επεξεργαστή RISCV-MINI [89]. Εντάξαμε τον επεξεργαστή αυτό στην MMU του BOOM Out-of-Order πυρήνα [90] για τον χειρισμό εκχωρήσεων μνήμης στο hardware.

Για το VMAF χρησιμοποιήσαμε ένα σταθερό κύκλωμα στο hardware. Καθώς η έκδοση του πυρήνα του Linux που χρησιμοποιήσαμε (5.11.6) χρησιμοποιούσε red-black δέντρα για την αποθήκευση των `vm_area_structs`, το VMAF μας ουσιαστικά πραγματοποιεί διάσχιση δυαδικού δέντρου στο hardware.

Το πρωτότυπο έτρεχε την έκδοση 5.11.6 του πυρήνα του Linux. Προκειμένου να ενσωματώσουμε τον hardware μηχανισμό, χρειάστηκε να τροποποιήσουμε κατάλληλα τον πυρήνα. Για μετρήσεις στο hardware όπως καθυστέρηση σφαλμάτων σελίδας για δεσμεύσεις που χειριζόταν ο RISCV-MINI ή cache misses στον RISCV-MINI, προσθέσαμε κατάλληλους καταχωρήτες CSR στον πυρήνα.

Για να ποσοτικοποιήσουμε το κόστος της χρήσης επαναπρογραμματιζόμενου hardware αντί για σταθερού ASIC, υλοποιήσαμε άλλη μία έκδοση στο FPGA, η οποία περιείχε ένα σταθερό hardware module το οποίο επίσης χειριζόταν δεσμεύσεις μνήμης με βάση τη βιβλιοθήκη LFA. Κατά τη λήψη αιτήματος μετάφρασης μίας εικονικής διεύθυνσης, το hardware module υπολόγιζε τη διεύθυνση του set στην `tag_array` όπου η

διεύθυνση έπρεπε να τοποθετηθεί, και διάβαζε όλους τους ways του set αυτού έναν προς έναν. Αν εντόπιζε τη ζητούμενη διεύθυνση, τότε το module επέστρεφε το αντίστοιχο PTE. Ομοίως, όταν το module δεχόταν αίτημα για allocation, έψαχνε τους ways του αντίστοιχου set για κάποιο κενό. Αν έβρισκε, τότε ενημέρωνε κατάλληλα το αντίστοιχο πεδίο στο `tag_array`. Προς επιτάχυνση της διαδικασίας αυτής, το module επίσης περιλάμβανε μία cache με όλους τους ways από sets στα οποία είχε επιχειρηθεί πρόσφατα πρόσβαση. Η cache ήταν direct-mapped, με χώρο για 64 sets του `tag_array`. Σε αντίθεση με την υλοποίηση με βάση το RISCV-MINI, τα μεταδεδομένα της βιβλιοθήκης (όπως π.χ. το μέγεθος της περιοχής που χρησιμοποιούσε το hardware για δεσμεύσεις μνήμης) διατηρούνταν σε καταχωρητές CSR τους οποίους διάβαζε το hardware module. Χρησιμοποιήσαμε επίσης έναν CSR για τη διεύθυνση της αρχής του `tag_array`.

Πειραματική διάταξη

Δοκιμάσαμε 4 διαφορετικά συστήματα στο FPGA:

1. **Baseline:** Μη τροποποιημένος επεξεργαστής BOOM που έτρεχε μη τροποποιημένη έκδοση του πυρήνα του Linux
2. **SW-only:** Μη τροποποιημένος επεξεργαστής BOOM, αλλά που όμως έτρεχε μία τροποποιημένη έκδοση του πυρήνα του Linux που υλοποιούσε το σχήμα δέσμευσης μνήμης της LFA βιβλιοθήκης, όπως περιγράφεται παραπάνω, αμιγώς στο software. Κατά τον χειρισμό του σφάλματος σελίδας, το ΛΣ προσπαθεί πρώτα να δεσμεύσει μνήμη χρησιμοποιώντας το απλοποιημένο σχήμα δέσμευσης. Αν αυτό αποτύχει, τότε επιστρέφει στο σύνηθες σχήμα δέσμευσης μνήμης του πυρήνα του Linux
3. **Fixed-HW:** Το σταθερό hardware που υλοποιεί τη βιβλιοθήκη LFA, όπως περιγράφηκε παραπάνω
4. **Valinor:** Ο προτεινόμενος μηχανισμός μας, υλοποιημένος με βάση τον RISCV-MINI πυρήνα, όπως περιγράφηκε παραπάνω.

Για την αξιολόγηση του FPGA πρωτοτύπου χρησιμοποιήσαμε 5 microbenchmarks. Επιλέξαμε ορισμένες εφαρμογές με μικρό χρόνο εκτέλεσης, συμπεριλαμβανομένου ενός συνόλου απλών πράξεων με πίνακες και διανύσματα (`spmv`, `vvadd`, `median`, `tramspose`), καθώς και μία serverless συνάρτηση (`cJSON`), όλες εκ των οποίων έχουν μεγάλα αποτυπώματα μνήμης. Εξαιτίας των περιορισμών του FPGA πρωτοτύπου, παραγματοποιήσαμε μία μικρής κλίμακας μελέτη, προσαρμόζοντας τα microbenchmarks ώστε το αποτύπωμα μνήμης τους να κυμαίνεται μεταξύ $14MB$ και $16MB$, και πειραματιστήκαμε με διαφορετικά μεγέθη allocation pool μεταξύ $1MB$ και $16MB$.

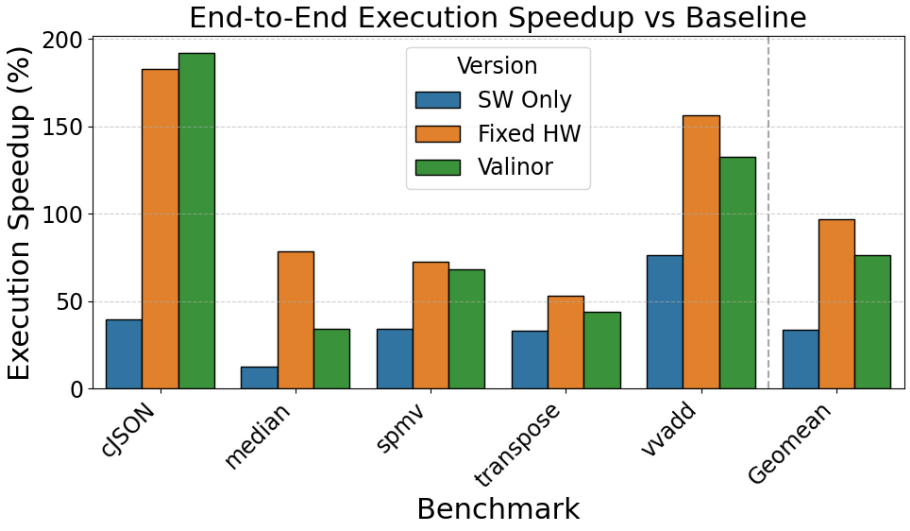


Figure 1.5.1: Επιτάχυνση του συνολικού χρόνου εκτέλεσης για διαφορετικά benchmarks.

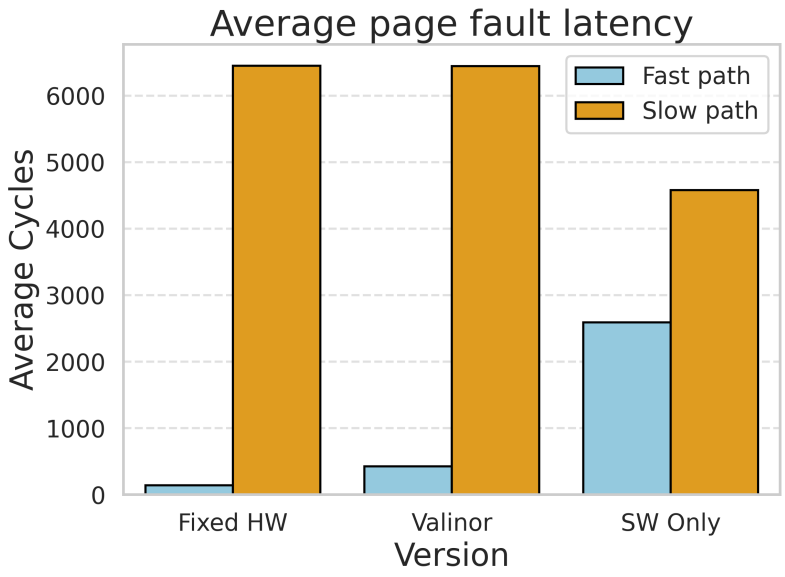


Figure 1.5.2: Μέσος αριθμός κύκλων ανά σφάλμα σελίδας για καθεμία από τις εκδοχές που δοκιμάσαμε. Το "Fast path" αναφέρεται στα σφάλματα που χειρίζεται ο εναλλακτικός μηχανισμός σε κάθε περίπτωση, ενώ το "Slow path" αναφέρεται στη συνήθη διαδικασία, δηλαδή στον χειριστή σφαλμάτων σελίδας του Linux.

1.5.2 Αποτελέσματα

Αξιολόγηση της επίδοσης. Στο Σχήμα 1.5.1 παρατίθεται η επιτάχυνση του συνολικού χρόνου εκτέλεσης για τις διαφορετικές εκδόσεις που δοκιμάστηκαν, όπως μετρήθηκε στο FPGA. Το μέγεθος του allocation pool διατηρήθηκε σταθερό στα $16MB$, ενώ όλα τα benchmarks χρησιμοποιούσαν μνήμη μεταξύ $14MB$ και $16MB$, έτσι ώστε να χωράνε πλήρως στο allocation pool. Από το διάγραμμα προκύπτουν τρεις παρατηρήσεις: (i) Ακόμα και χωρίς τροποποιήσεις στο hardware, η **SW-only** έκσοδη ήδη οδηγεί σε 34% επιτάχυνση, ενδεικτικό της αποδοτικότητας της εναλλακτικής πολιτικής δέσμευσης μνήμης σε σύγκριση με τον buddy allocator, (ii) η **Fixed-HW** έκδοση είναι η γρηγορότερη, επιτυγχάνοντας geomean επιτάχυνση 97%, γεγονός ενδεικτικό των πλεονεκτημάτων της διαχείρισης των σφαλμάτων σελίδας στο hardware, (iii) το **Valinor** συστηματιά παρουσιάζει καλύτερη επίδοση τόσο από το baseline οσο και από τη **SW-only** έκδοση, με geomean επιτάχυνση 77%. Γίνεται λοιπόν αντιληπτό ότι, παρά τη χρήση γενικού σκοπού επαναπρογραμματιζόμενου πυρήνα, η επίδοση του **Valinor** είναι συγκρίσιμη με αυτή του σταθερού hardware module, γεγονός που ποσοτικοποιείται περαιτέρω στην επόμενη παράγραφο.

Καθυστέρηση σφαλμάτων σελίδας. Στο Σχήμα 1.5.2 παρατίθεται ο μέσος αριθμός κύκλων που δαπανάται σε διαφορετικές περιπτώσεις διαχείρισης σφαλμάτων σελίδας. Και στις τρεις εκδόσεις, το "Fast path" αναφέρεται στα σφάλματα σελίδας τα οποία χειρίζεται ο προτεινόμενος μηχανισμός (δηλαδή εντός του hardware για το **Fixed-HW** και το **Valinor**, και χρησιμοποιώντας το LFA σχήμα για το **SW-only**), ενώ το "Slow path" αντιστοιχεί στην εναλλακτική περίπτωση, δηλαδή στον χειριστή σφαλμάτων σελίδας του Linux. Παρατηρούμε ότι η επιτυχής δέσμευση μνήμης με το LFA σχήμα, παρακάμπτοντας τον buddy allocator (όπως συμβαίνει στη **SW-only** έκδοση) ήδη επιταχύνει τα σφάλματα σελίδας κατά 1.77×. Από την άλλη, με το να παρακάμπτουμε πλήρως τον χειριστή σφαλμάτων σελίδας και να αναθέτουμε τη δέσμευση μνήμης στο σταθερό hardware πετυχαίνουμε 45× επιτάχυνση. Τέλος, παρότι ο επαναπρογραμματιζόμενος in-order πυρήνας του **Valinor** χρειάζεται σχεδόν 3× περισσότερους κύκλους σε σχέση με το **Fixed-HW** για να χειριστεί ένα σφάλμα σελίδας, πετυχαίνει επιτάχυνση μίας τάξης μεγέθους (15.2×) σε σχέση με τον χειριστή σφαλμάτων σελίδας. Βλέπουμε λοιπόν ότι, ακόμα και στην περίπτωση περίπλοκου επαναπρογραμματιζόμενου υλικού, η αντιστοίχιση εικονικών διευθύνσεων σε φυσικές μέσω hardware μπορεί να οδηγήσει σε σημαντική επιτάχυνση, απαλλάσσοντας από το επιπλέον κόστος που

επισύρει ο σύνθετος χειριστής σφαλμάτων σελίδας του ΛΣ.

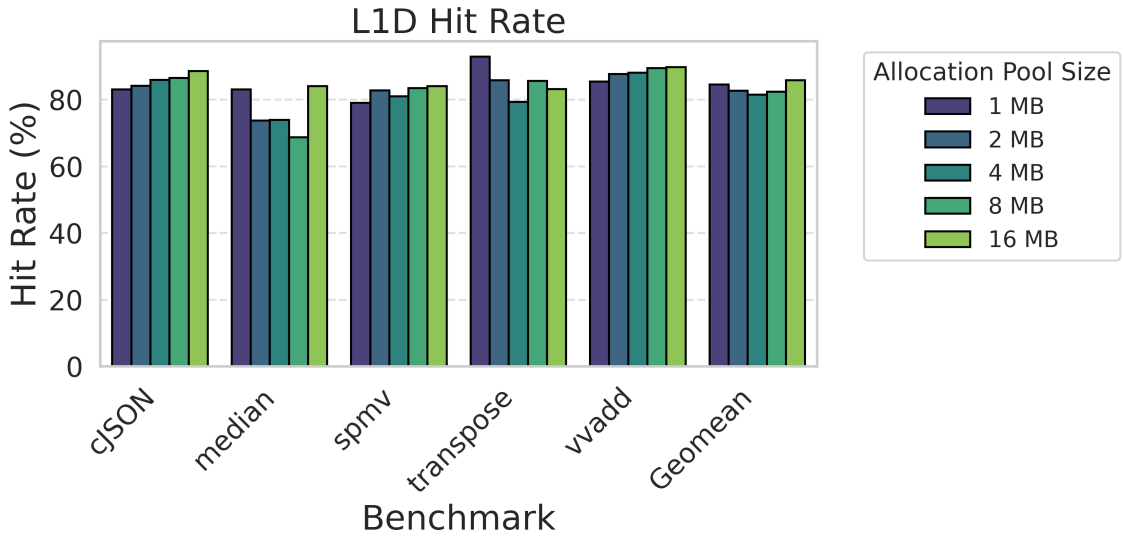


Figure 1.5.3: Ποσοστό ευστοχίας της L1D cache του **Valinor**

Αστοχίες στην L1 cache. Προκειμένου να γίνει καλύτερα αντιληπτό πώς το Valinor επιτυγχάνει τόσο χαμηλή καθυστέρηση, στο Σχήμα 1.5.3 απεικονίζεται το ποσοστό ευστοχίας στην L1D cache του in-order πυρήνα που χρησιμοποιείται στο **Valinor** για διαφορετικά benchmarks. Κάθε benchmarks δοκιμάστηκε για πέντε διαφορετικά μεγέθη του allocation pool. Γίνεται αντιληπτό ότι το ποσοστό ευστοχίας της cache κυμαίνεται συστηματικά γύρω στο 80%. Μετρήσαμε επίσης το ποσοστό της L1I cache, το οποίο όμως ήταν συστηματικά 100% για επανειλημμένες εκτελέσεις του ίδιου benchmark για το ίδιο μέγεθος allocation pool. Αυτά τα δύο γεγονότα είναι ενδεικτικά τωμ πλεονεκτημάτων μίας προγραμματιζόμενης μονάδας στο hardware: Από τη μία, οι βιβλιοθήκες για δέσμευση μνήμης είναι αρκετά μικρές ώστε να χωρέσουν στην κρυφή μνήμη εντολών. Από την άλλη, η επιλογή η επιλογή ενός εναλλακτικού σχήματος δέσμευσης μνήμης με λιγότερα μεταδεδομένα επιτρέπει στα μεταδεδομένα αυτά να χωρέσουν στην κρυφή μνήμη δεδομένων.

Ευαισθησία στο μέγεθος του allocation pool. Τέλος, προκειμένου να αναδείξουμε τα οφέλη του χειρισμού της δέσμευσης μνήμης στο hardware, τρέξαμε δύο από τα benchmarks (τα `spmv` και `vvadd`) για διαφορετικά μεγέθη του allocation pool. Σημειωτέον ότι, όπως και στο Σχήμα 1.5.1, το αποτύπωμα μνήμης των επιλεγμένων benchmarks ήταν μεταξύ $14MB$ και $16MB$, που σημαίνει ότι η απαιτούμενη μνήμη χωρούσε ολόκληρη στο allocation pool μόνο στην περίπτωση του μεγαλύτερου allocation pool. Το Σχήμα 1.5.4 απεικονίζει την επιτάχυνση του συνολικού χρόνου εκτέλεσης, ενώ το Σχήμα 1.5.5 απεικονίζει τη συνολική καθυστέρηση (σε κύκλους) που δαπανάται κατά τη δημιουργία αντιστοιχίσεων εικονικής μνήμης σε φυσική. Κατ' αρχάς, από το Σχήμα 1.5.4 γίνεται αντιληπτό ότι, ενώ για μικρά μεγέθη του allocation pool η επίδραση στον χρόνο εκτέλεσης είναι αμελητέα, καθώς αυξάνεται το μέγεθος του allocation pool η επίδοση βελτιώνεται σημαντικά. Αυτό είναι αναμενόμενο, καθώς το μέγεθος του allocation pool ουσιαστικά καθορίζει πόσες αντιστοιχίσεις εικονικής μνήμης σε φυσική δημιουργούνται στο "fast path". Αυτό επιβεβαιώνεται περαιτέρω από το Σχήμα 1.5.5, που δείχνει ότι, καθώς αυξάνεται το μέγεθος του allocation pool, ο συνολικός χρόνος που δαπανάται για τη δημιουργία αντιστοιχίσεων μειώνεται σημαντικά. Μάλιστα, φαίνεται ότι η συνολική καθυστέρηση για ασφάλματα σελίδας μειώνεται κατά 10.5× (για το `spmv`) και 58.9× (για το `vvadd`) σε σχέση με το baseline στην περίπτωση του `Fixed-HW`, και 7.0× (για το `spmv`) και 15.2× (για το `vvadd`) στην περίπτωση του **Valinor**. Και πάλι λοιπόν βλέπουμε ότι, ενώ το **Valinor** είναι λίγο πιο αργό από το **Fixed-HW**, μπορεί ακόμα να προσφέρει σημαντικά οφέλη από άποψη επίδοσης.

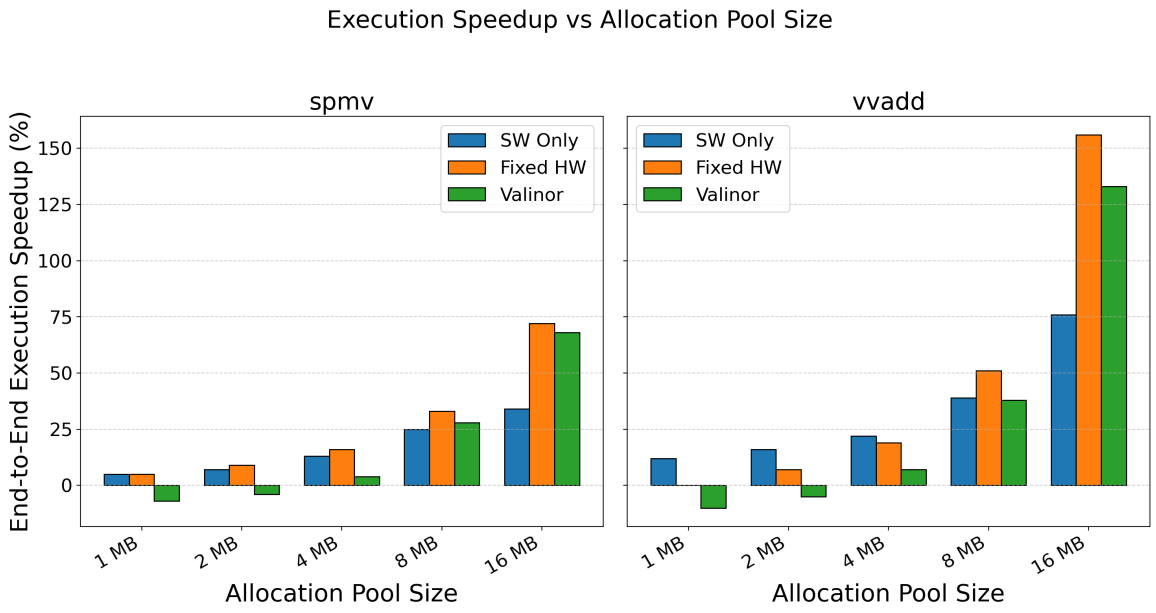


Figure 1.5.4: Επιτάχυνση του συνολικού χρόνου εκτέλεσης για διαφορετικά μεγέθη του allocation pool.

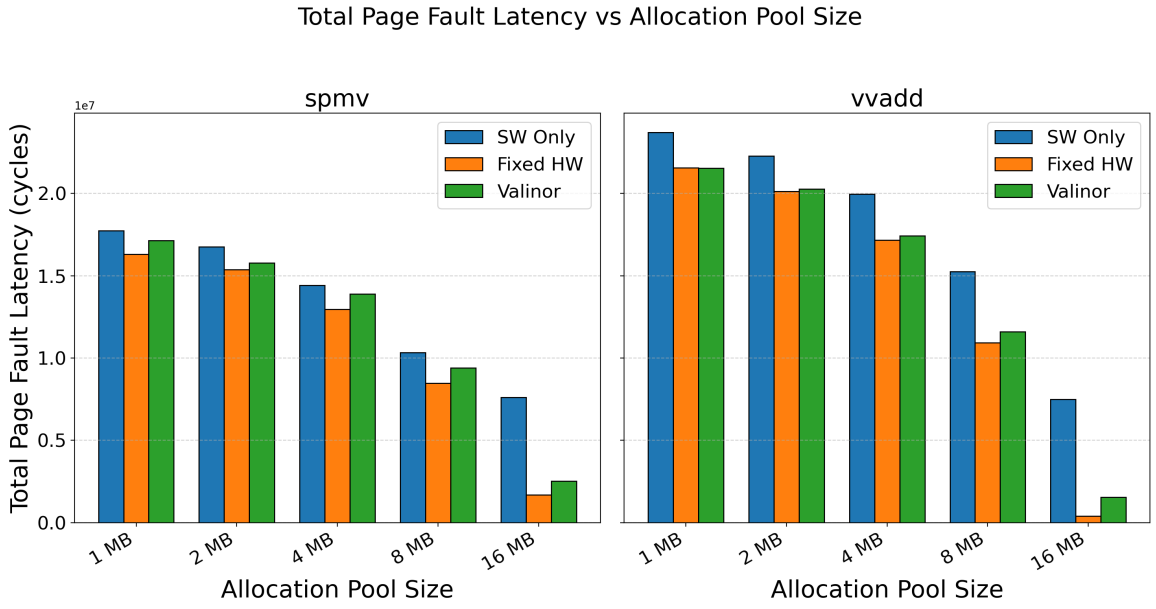


Figure 1.5.5: Συνολική καθυστέρηση για δημιουργία αντιστοιχίσεων για διαφορετικά μεγέθη του allocation pool.

1.6 Συμπεράσματα

Η παρούσα διπλωματική ασχολείται με τις αυξανόμενες προκλήσεις σε επίδοση και ενέργεια της δέσμευσης μνήμης στα σύγχρονα υπολογιστικά συστήματα. Δείχνουμε ότι σε μικρής διάρκειας, ευαίσθητες στην καθυστέρηση εφαρμογές, όπως οι serverless συναρτήσεις, τα microservices και το LLM inference, το κόστος της διαχείρισης της δέσμευσης σελίδων αμιγώς στο software κυριαρχεί στον χρόνο εκτέλεσης και την κατανάλωση ενέργειας.

Παρουσιάζουμε το **Valinor**, έναν προγραμματιζόμενο μηχανισμό στο hardware που συνεργάζεται με το ΛΣ για την επιτάχυνση των σφαλμάτων σελίδας, διατηρώντας λεπτομερή έλεγχο πάνω στις πολιτικές δέσμευσης μνήμης. Σε αντίθεση με προηγούμενους σταθερούς επιταχυντές, το Valinor είναι *προγραμματιζόμενο* και μπορεί να προσαρμόζεται δυναμικά στις συνθήκες χρόνου εκτέλεσης, όπως είναι το εύρος ζώνης της DRAM, η πληρότητα της cache και η τοπικότητα NUMA. Με τον τρόπο αυτό συνδυάζει τη χαμηλή καθυστέρηση της δέσμευσης στο hardware με την ευελιξία των μηχανισμών του software.

Υλοποιήσαμε και αξιολογήσαμε το Valinor σε cycle-accurate προσομοιωτή, καθώς και σε ένα FPGA RISC-V πρωτότυπο που έτρεχε Linux. Τα πειράματά μας αποδεικνύουν ότι το Valinor μειώνει τη συνολική καθυστέρηση σφαλμάτων σελίδας μέχρι και **7–15**×, βελτιώνοντας τη συνολική επίδοση εφαρμογών μέχρι και **77 %** σε σύγκριση με το baseline Linux. Τα αποτελέσματα επίσης δείχνουν ότι το Valinor πετυχαίνει **80% L1D cache hit rate**, κοντά στην αποδοτικότητα ενός σταθερού hardware allocator, διατηρώντας την ευελιξία του software.

Συνοψίζοντας, η παρούσα εργασία αποδεικνύει ότι μία προγραμματιζόμενη, υποβοηθούμενη από το hardware προσέγγιση στη δέσμευση μνήμης μπορεί να γεφυρώσει το χάσμα μεταξύ ευελιξίας και επίδοσης.

Μελλοντική δουλειά

Υπάρχουν πολλαπλές προοπτικές για μελλοντική επέκταση του Valinor:

- **Υποστήριξη για πολλαπλούς πυρήνες και NUMA**, ώστε να συντονίζει τις αποφάσεις για δέσμευση μνήμης μεταξύ διαφορετικών πυρήνων και NUMA περιοχών.
- **Προσαρμογή υποδηγούμενη από τηλεμετρία**, αξιοποιώντας δηλαδή μετρικές του hardware για την καθοδήγηση των στρατηγικών τοποθέτησης και δέσμευσης.
- **Περιβάλλοντα disaggregated memory**, ώστε να επιτρέπει remote δέσμευση μνήμης και μετάφραση.

Όνομα	Σύνοψη	Δέσμευση μνήμης	Μετάφραση	Συγχρονισμός με ΛΣ
Ultra-fast (UF)	Χαμηλής καθυστέρησης δέσμευση μνήμης βασισμένη σε hashing για αντικείμενα μνήμης με μικρή διάρκεια ζωής. Βελτιστοποιεί το fast path και την tail latency.	Αξιοποιεί μία περιοχή μνήμης οργανωμένη με set-associative τρόπο για γρήγορες δεσμεύσεις μνήμης.	Χειρίζεται τη μετάφραση με βάση το hashing στο L2 TLB miss path.	Μία μεγάλη περιοχή μνήμης που δεσμεύεται κατά την αρχικοποίηση.
Runahead	Προ-δεσμεύει σελίδες ασύγχρονα, ώστε να κρύψει μελλοντικά stalls για δέσμευση μνήμης.	Παρασκηνιακό νήμα γεμίζει ένα pool ελεύθερων φυσικ«ών σελίδων, από το οποίο δεσμεύονται σελίδες στο προσκήνιο.	Ξεχωριστό Page Table για προ-δεσμευμένες σελίδες.	Μία μεγάλη περιοχή μνήμης που δεσμεύεται κατά την αρχικοποίηση.
Πραγματικού χρόνου	Αποσκοπεί σε ντετερμινιστικό χρόνο εκτέλεσης με άνω φραγμένη καθυστέρηση στη χειρότερη περίπτωση.	Προ-δεσμευμένες γραμμικές φυσικές σελίδες για αντικείμενα μνήμης πραγματικού χρόνου.	Μετάφραση που βασίζεται στη μετατόπιση (offset) για μία περιοχή πραγματικού χρόνου.	Αίτημα μνήμης για το πρώτο σφάλμα στη συγκεκριμένη περιοχή.
Telemetry-aware	Χρησιμοποιεί μετρητές υλικού (π.χ. εύρους ζώνης, περιεχομένων της cache) για να κατευθύνει την τοποθέτηση δεδομένων.	Επιλέγει banks με βάση την τηλεμετρία χρησιμοποιώντας πολλαπλά hash functions.	Χειρίζεται μετάφραση με βάση το hashing στο L2 TLB miss path.	Μία μεγάλη περιοχή μνήμης που δεσμεύεται κατά την αρχικοποίηση.
Ενεργειακά αποδοτική	Μειώνει την ενέργεια μέσω χαμηλής συχνότητας δέσμευση στο παρασκήνιο.	Ασύγχρονη προ-δέσμευση σε μία κατάσταση χαμηλής ενέργειας. Το προσκήνιο δεσμεύει από αυτό το pool.	Ξεχωριστό Page Table για προ-δεσμευμένες σελίδες.	Σπάνιος (για επεκτάσεις του pool)
Προστασία από Rowhammer	Προστατεύει από το Rowhammer εισάγοντας guard rows μεταξύ ευαίσθητων δεσμεύσεων.	Δεσμεύει φυσικές σελίδες με guard-row spacing μέσω χρωματισμού σελίδων.	Χειρισμός μεταφράσεων με βάση το χρώμα στο L2 TLB miss path.	Μία μεγάλη περιοχή μνήμης που δεσμεύεται κατά την αρχικοποίηση.
Επιβολή ακεραιότητας	Επιβάλλει ακεριότητα ώστε να εξασφαλίσει ότι οι αντιστοιχίσεις δεν παραποιούνται.	Δεσμεύει από ένα set-associative τμήμα και αποθηκεύει επιπρόσθετα μεταδεδομένα για το merkle tree.	Επαληθεύει την ακεραιότητα κατά το L2 TLB miss χρησιμοποιώντας ένα merkle tree αποθηκευμένο σε δεσμευμένες σελίδες.	Μία μεγάλη περιοχή μνήμης που δεσμεύεται κατά την αρχικοποίηση.
Placement-aware	Βελτιστοποιεί τοπικότητα και ανταγωνισμό (π.χ. συγκρούσεις μεταξύ banks)	Free lists σε επίπεδο bank για placement-aware δέσμευση	Χειρίζεται bank-aware μετάφραση στο L2 TLB miss	Μία μεγάλη περιοχή μνήμης που δεσμεύεται κατά την αρχικοποίηση.

Table 1.1: Αντιπροσωπευτικές βιβλιοθήκες δέσμευσης και πολιτικές που μπορούν να προγραμματιστούν στον hardware μηχανισμό του Valinor. Κάθε πολιτική έχει διαφορετικούς στόχους: επίδοση (UF), προ-δέσμευση (RA), ντετερμινισμό σε πραγματικό χρόνο (RT), telemetry-guided προσαρμογή (TA), ενεργειακή αποδοτικότητα (EE), προστασία από Rowhammer (RH), επιβολή ακεραιότητας (INT) και βελτιστοποιήσεις τοποθέτησης (PA).

Table 1.2: Τεχνικά χαρακτηριστικά του RISC-V πρωτοτύπου

Prototype Configuration	
Core	64-bit RISC-V BOOM Out-of-Order core
MMU	L1 I-TLB: 32-entry, direct-mapped, 1-cycle latency
	L1 D-TLB (4 KB): 8-entry, fully assoc, 1-cycle latency
	L2 TLB: 512-entry, direct-mapped, 2-cycle latency
	Page Walk Cache: 8-entry, fully assoc, 1-cycle latency
L1 Cache	L1 I/D-Cache: 2 KB, 4-way assoc, 3-cycle access latency
	LRU replacement policy
Linux Kernel	v5.11.6
Allocation engine	Core: RISCV-MINI 32-bit in-order core [89]
	L1 I/D-Cache: 4KB, direct-mapped
FPGA	Xilinx ZCU 106 [88], 1GB DDR4

Chapter 2

Introduction

Memory allocation is a fundamental OS service that has a growing impact on performance and energy efficiency in modern computing systems. Even when data does not need to be fetched from disk, each allocation typically involves handling a page fault in software: the processor traps into the operating system, which assigns a physical frame, updates the page tables, and resumes execution.

However, performing physical allocation entirely in software comes with two key limitations. First, while the process of solely using software to allocate memory was historically amortized over long-running workloads, it has become a *first-order bottleneck* in modern environments where short-lived, latency-critical tasks dominate [1, 2]. For example, serverless functions, microservices, data analytics and even instances of large language model (LLM) inference (e.g., in systems that support unified memory across CPU and GPU) often execute for microseconds to milliseconds, making the tens of thousands of cycles spent in allocating memory via page faults a significant fraction of total runtime. As we demonstrate in §5 and as recent studies show, allocation-related OS activity can account for more than 30% of execution time in such workloads and consume up to 20% of system energy. Second, the OS software responsible for physical memory allocation is not exposed to critical microarchitectural state (e.g., memory bandwidth utilization, cache occupancy, DRAM row buffer conflicts), limiting its ability to adapt placement decisions based on runtime conditions.

Prior works proposed accelerating page allocation by delegating it to specialized components inside the CPU that handle allocations directly in hardware without invoking the software-level OS stack [23, 2, 24, 25]. For example, Tirumalasetty et al. [91] extend the memory management unit (MMU) with a hardware queue that the software-level OS fills up in the background with free physical pages to serve future allocation requests and avoid trapping to the kernel on the critical path of a page fault while [2] observe that the storage access latency has reduced significantly in modern systems, rendering the overhead of handling page faults in software more pronounced and thus propose offloading major page faults to a dedicated hardware component. Such hardware-based allocation techniques eliminate context switch overheads, pipeline flushes, and expensive out-of-order execution during page faults, thereby reducing allocation latency and energy consumption.

However, existing schemes that completely offload allocation to hardware suffer from two key limitations: (i) they prioritize allocation speed over placement quality without taking into account microarchitectural state or application-level intents, and (ii) they hardwire allocation policies into fixed-function logic, limiting adaptability to changing workloads and system conditions. First, while delegating allocation to hardware can effectively eliminate kernel traps and context-switch overheads, existing designs typically optimize for allocation speed rather than placement quality. Existing hardware-based allocation methods [23, 2, 24, 25, 1] follow simple heuristics—such as selecting the first available free page—to minimize latency on the critical path. However, such "blind" allocation ignores application- and system-level placement intents that otherwise the software-based OS allocation could potentially enforce and are essential for optimizing performance, isolation, and energy efficiency. For instance,

policies that rely on NUMA affinity, page coloring, or tiered memory allocation cannot be enforced if the hardware performs placement autonomously without semantic awareness. As a result, the gains from faster allocation can be offset by suboptimal data placement that increases contention or violates partitioning policies (e.g. bank-level partitioning). Second, hardwiring allocation policies into fixed-function logic introduces rigidity. Once a specific placement rule or target memory region is embedded in hardware, adapting to changing workloads, interference patterns, or hardware topologies becomes impractical. This lack of programmability prevents the system from tailoring allocation strategies to diverse deployment scenarios or evolving application needs. Thus, completely offloading physical memory allocation to hardware with greed-for-speed policies sacrifices important placement optimizations and adaptability, limiting the overall effectiveness of such approaches.

To address the limitations of both purely software- and hardware-based allocation schemes, we propose Valinor, a new hardware–OS cooperative memory allocation substrate that combines the flexibility of software-based allocators with the low latency and efficiency of hardware mechanisms. Valinor introduces a programmable hardware allocation engine that the OS can configure to offload memory allocation requests directly to hardware when beneficial, while still retaining the ability to fall back to the software allocator when needed. This hybrid design allows Valinor to preserve rich allocation semantics—such as NUMA placement, page coloring, or tiered memory policies—while eliminating costly kernel traps and context switches on the allocation path.

At a high level, Valinor provides *three key benefits*:

(1) **Hardware-level performance gains.** In latency-critical workloads (e.g., serverless functions, microservices, or LLM inference), Valinor bypasses the slow software page-fault path by delegating allocations to a dedicated hardware engine. This reduces allocation latency and energy consumption by avoiding kernel involvement, pipeline flushes, and scheduler delays.

(2) **Programmability and policy extensibility.** Unlike fixed-function hardware allocators, Valinor exposes a programmable interface that allows the OS to define allocation and translation policies in software and load them into hardware. This enables the same hardware substrate to support diverse policies—ranging from application-specific pools to NUMA-aware placement—without modifying the MMU or page table walker.

(3) **Adaptivity to hardware state.** Because the allocation engine has direct visibility into microarchitectural metrics (e.g., DRAM bandwidth, cache occupancy, bank conflicts), it can dynamically steer allocations to optimize performance under changing hardware conditions. This allows Valinor to realize adaptive, data-driven allocation decisions that are infeasible in software alone.

Mechanism Overview. The OS programs the hardware allocation engine with the desired allocation library, enabling hardware-level execution of its policy. When a core encounters a page fault, the allocation engine determines whether the access qualifies for hardware allocation and, if so, assigns a physical page according to the installed policy, updates internal data structures, and returns the mapping to the MMU—entirely in hardware. Otherwise, Valinor falls back to the standard OS page fault handler. This cooperative interface maintains compatibility with existing software allocators while enabling hardware acceleration transparently to applications.

Key Results. We prototype Valinor in a real BOOM RISCV soft-core running Linux (i.e., on an FPGA platform) to evaluate its performance and overheads. We evaluate four different use cases that benefit from Valinor's capabilities: (i) accelerating allocations for short-lived, latency-critical workloads, (ii) enabling energy-efficient allocation and data placement policies and, (iii) enforcing integrity checks to secure virtual-to-physical address mappings and, (iv) adapting placement based on runtime microarchitectural conditions. Our evaluation yields the following key results:

First, Valinor accelerates page allocation by **7-15**× compared to the baseline Linux allocator by eliminating OS traps and handling faults entirely in hardware. Second, Valinor improves end-to-end performance by **77%** on average. Fifth, through design synthesis, we find out that the hardware engine incurs minimal area and power overheads (1.5% core area).

In this thesis, we make the following contributions:

- We propose **Valinor**, a hardware–OS cooperative memory allocation substrate that combines the flexibility of software allocators with the low latency and energy efficiency of hardware-based allocation. Valinor introduces a programmable allocation engine that allows the OS to selectively offload allocation requests to hardware, eliminating costly kernel traps and context switches while preserving fine-grained placement control.
- We design a **programmable hardware allocation engine** that resides alongside the MMU and can be configured by the OS to execute allocation and translation policies in hardware. The engine exposes a lightweight ISA-like interface for installing allocator logic and supports policy extensions without modifying the MMU or page table walker.
- We demonstrate **new use cases** enabled by Valinor's hardware–software interface, including adaptive placement for bandwidth-sensitive workloads, energy-efficient allocation for short-lived tasks, and integrity checking of virtual-to-physical mappings—all implemented through the same programmable substrate.
- We implement a Linux prototype of Valinor on a BOOM RISC-V FPGA platform. Across short-lived and placement-sensitive workloads, Valinor achieves **7-15× faster allocation** and **77% end-to-end speedups**, with minimal hardware overhead (1.5% of core area).

Chapter 3

Related Work

To the best of our knowledge, Valinor is the first hardware-software co-design technique that enables programmable virtual-to-physical mapping in hardware, while also maintaining flexibility in the choice of translation scheme. In this section, we qualitatively compare Valinor to prior works that modify the hardware to accommodate changes to the Virtual Memory subsystem.

3.1 Offloading Memory Management to Hardware

A large body of work has explored hardware mechanisms for memory management, focusing on allocation and deallocation [1, 3, 4, 5, 6, 7, 8, 9, 10], or even incorporating garbage collection schemes [6, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22].

Many of these approaches [4, 5, 6, 7] implement variations of the buddy allocator in hardware, using combinatorial circuits to identify free contiguous regions of appropriate size. Kanev et al. [3], on the other hand, design Mallacc, an accelerator for handling small-size `malloc()` allocations in hardware. They rely on the fact that `malloc()` commonly employs free lists for different size classes, and implement a hardware unit that allows quick size class identification and updates to the corresponding free list in the common case. Wang et al. expand on this idea in Memento [1], employing hardware-maintained arenas for each size class which are provided by the OS at page granularity. This way, they enable hardware allocation in both user- and kernelspace.

These works are concerned with allocation at object granularity, and thus restrict themselves to small allocation sizes. They also require ISA modifications to inform hardware of a request for memory or to free allocated memory. Valinor's scope is different, tackling allocation at page granularity, and relying on hardware events of existing CPU components (in the MMU) for its mechanisms to be triggered.

3.2 Hardware Paging

Several works [23, 2, 24, 25] design hardware mechanisms specifically for page fault handling in hardware. These approaches alleviate the memory mapping overhead by moving parts of the page fault handler out of the page fault critical path, to a background thread, and adding hardware to establish a virtual-to-physical mapping. This notably involves maintaining a pool of free physical pages from which the hardware can quickly reserve at fault time.

Lee et al. [2] propose hardware-based major page fault handling. The key idea is to introduce a hardware module that issues I/O commands to fetch appropriate blocks into memory and establishes virtual-to-physical mappings without context-switching to the OS. The location of the desired block is maintained as metadata in the virtual address's PTE, which is created by the OS at `mmap()` time.

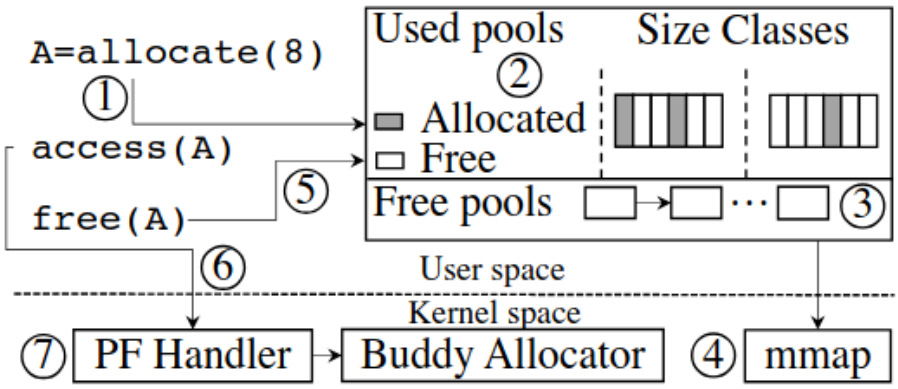
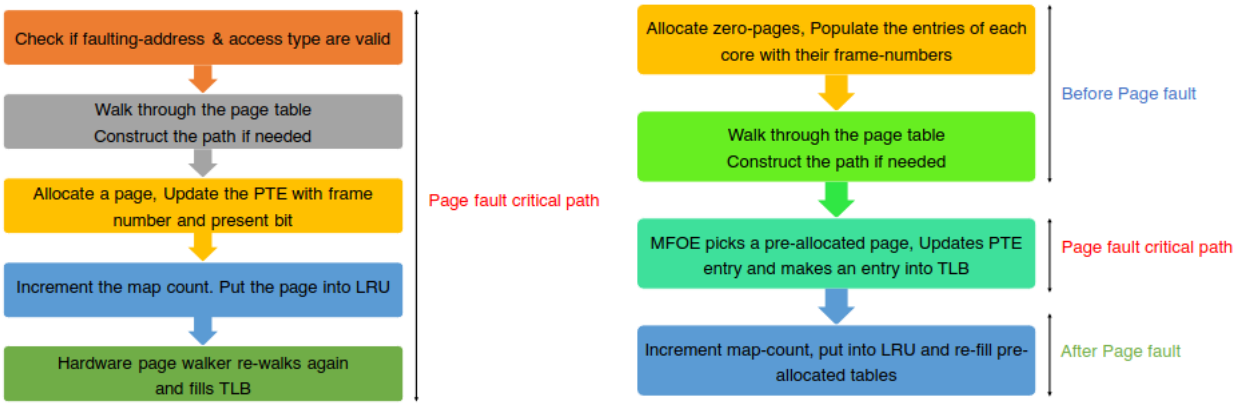


Figure 3.1.1: Memento's [1] arena-based hardware allocation scheme. Figure taken from [1]

The page that is used as a destination for the data of the I/O operation is chosen from a FIFO queue, which is consistently refilled with free pages by a background OS thread.

Tirumalasetty et al. [23] utilize similar ideas, targeting anonymous minor page faults. They propose simplifying the page fault critical path by moving most operations of the page fault handler to a background thread, leaving only assignment of a free physical page to the requested virtual page at fault time, which can be handled in hardware. For this purpose, they maintain a FIFO ring buffer with free physical pages, which is consistently filled with zeroed pages by a background OS task, and from which the hardware consumes an entry whenever it encounters an unmapped virtual page. To determine which mappings can be established in hardware, they construct all Page Table Entries for a target virtual memory area at `mmap()` time, and set an appropriate bit in the PTE to inform the PTW that a page fault should be handled in hardware. Furthermore, they also defer OS maintenance operations (e.g. adding pages to an LRU list for swapping, establishing reverse mappings) to the OS later, outside the critical path.

Finally, similar mechanisms are used in [24] and [25], although their scope is different, targeting disaggregated memory systems.



(a) Default path

(b) Modified path according to [23]

Figure 3.2.1: Default and hardware-based Page Fault critical path. From [23]

Although these solutions can drastically reduce the overheads incurred by page faults, they have several shortcomings. Specifically, they:

1. Compromise system resources to run background tasks, relying on background threads for main-

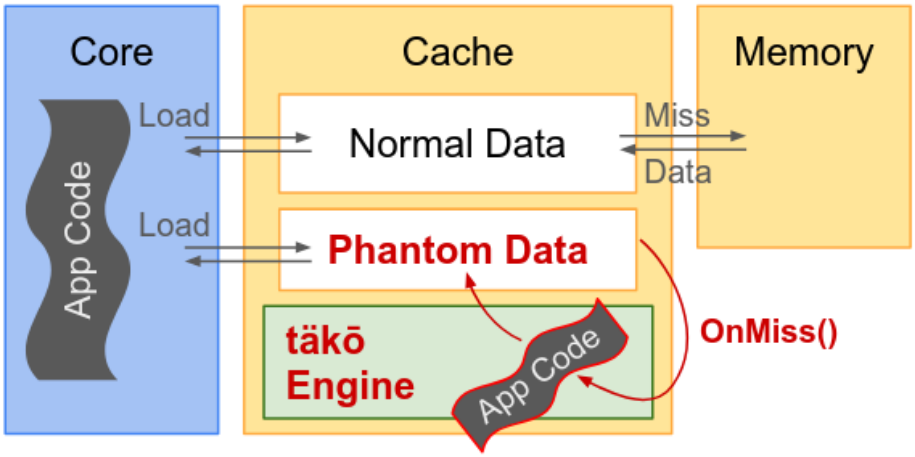


Figure 3.3.1: täkō's [43] event-triggered cache design. Figure taken from [43]

tenance operations

2. Provide no control over the physical page chosen for a virtual mapping. Indeed, the hardware always blindly chooses the first available page from a software-managed queue.

3. The page tables for virtual memory areas eligible for hardware-based page fault handling have to be constructed in advance (i.e. at `mmap()` time), in order for the hardware to decide whether or not to establish the virtual-to-physical mapping itself, or to instead trap to the OS kernel

By contrast, Valinor offloads the *entire* memory allocation path in hardware, while allowing freedom in the choice of mapping. Furthermore, hardware-eligible pages are discovered by a hardware module walking the OS's virtual memory area structures in hardware, thus eliminating the need for page table pre-allocation.

3.3 Reconfigurable fabric

Various works propose introducing reconfigurable hardware near the processor. A common approach is to place a general-purpose programmable unit near the processor [26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38], while other works target specific components or functions, like the memory controller [92, 93, 94, 95, 42, 96], caches and directories [39, 40, 41, 42, 43], prefetching [97, 98, 47], security [44, 45] or compression acceleration [46].

Many of these works [26, 43, 47] are event-driven, activating the reconfigurable units when specific microarchitectural events are observed. For example, Ainsworth et al. [47] propose integrating small, in-order, programmable cores running at low frequency that generate prefetch requests. They run small programs that are triggered upon access to data in specific memory regions, or when prefetched data is inserted in the L1 cache. Schwedock et al. expand on the idea of code triggered by data movement in [43]. They propose augmenting the hardware-software interface by introducing software callbacks that are triggered by caches (namely upon a cache miss, eviction or writeback) and run on reconfigurable fabric. This allows lightweight data processing (e.g. compression) when data is inserted into the cache or when it is written back to main memory.

Valinor adopts a similar approach to the aforementioned works, allowing execution of callbacks on a reprogrammable hardware unit, however our focus is on events triggered by the virtual memory subsystem (i.e. page faults and address translation).

3.4 Address Translation Optimizations

Prior work has extensively explored alternative virtual-to-physical mapping schemes, leveraging large pages [48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63], virtual-to-physical address space contiguity [64, 65, 66, 67, 68, 69, 70, 71, 72], hash-based mappings [73, 74, 75], as well as different Page Table designs [76, 77, 78, 79, 80, 81, 74, 75, 82, 83, 84, 85, 86], with a view to accelerating Address Translation. The closest of these works to our approach is [86], which proposes an architecture that allows for a customizable virtual-to-physical translation table. This work, however, (i) is restricted to translation, with allocation still handled by the OS and (ii) offloads translation to a background thread. Valinor, on the other hand, allows for flexibility in *both* address translation and virtual-to-physical mapping.

Chapter 4

Background

This section summarizes the fundamental principles of the Virtual Memory subsystem, the primary OS component modified by our proposed approach. It also presents some technical details regarding the Chipyard ecosystem, on which our prototype was developed.

4.1 Virtual Memory

4.1.1 Fundamentals of Virtual Memory

Virtual Memory (VM) is a core component of modern operating systems. The Operating System (OS) provides processes with a large, private, contiguous virtual address space (VAS), decoupled from the underlying physical memory (RAM). To achieve this, the OS relies on paging: it splits the virtual address space in pages (by default $4KB$), and maps individual virtual pages to physical pages in memory.

This decoupling provides multiple benefits, namely:

- Process isolation, as every process uses its own address space and is thus unaware of other processes residing in the system.
- Enabling larger-than-RAM programs, as the OS does not have to map the entire virtual address space to RAM, but only the pages being actively used.
- Simplified application memory management, as allocation of physical pages is handled by the OS, meaning the applications themselves do not have to consider which memory regions are occupied by other applications.
- Memory protection, as a process cannot access data belonging to another application e.g. simply by accessing an out-of-bounds address.
- Efficient sharing. Indeed, if two different processes require access to the same data, the OS can simply map virtual pages from both processes to the same physical page, without the need to create different copies for each.

4.1.2 Page Table

By default, the OS can map any virtual page to any physical page without restriction, as shown in Figure 4.1.1. This creates a need for the OS to maintain information about virtual-to-physical mappings. This information is maintained in the **Page Table (PT)**, a per-process data structure [73]. Although the exact form of the PT differs slightly among different architectures, it typically has the form of a Radix Tree, as depicted in Figure 4.1.2 for modern x86-64 processors. Discovering the

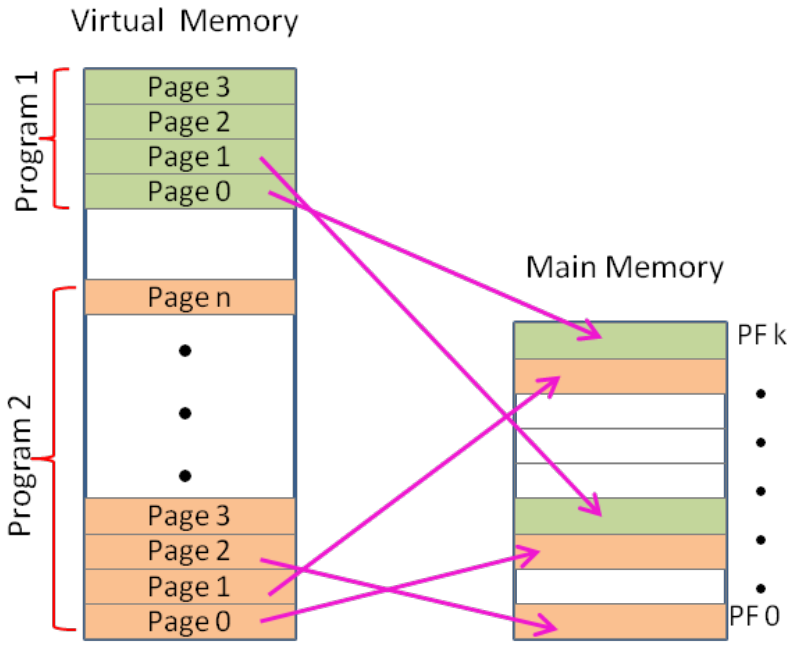


Figure 4.1.1: Main functionality of Virtual Memory. Contiguous virtual address spaces from different processes are split into pages, only some of which are mapped in (not necessarily contiguous) physical addresses. Figure taken from [99]

physical address corresponding to a virtual address requires sequentially traversing the different levels of the PT, a process called **Page Table Walk**. The result is the retrieval of the Page Table Entry (PTE), a 64-bit field which lies at the leaves of the radix tree and contains the corresponding physical address, as well as permission bits for the virtual address.

4.1.3 Address Translation and the Memory Management Unit

Since software issues memory requests using virtual addresses, while hardware operates using physical addresses, the hardware has to be able to convert virtual addresses to the corresponding physical ones. This process is called **Address Translation (AT)**. At its core, AT requires walking the PT for the requested virtual address in hardware and retrieving the PTE containing the physical address. However, given the high latency incurred by page table walks, modern processors employ a specialized

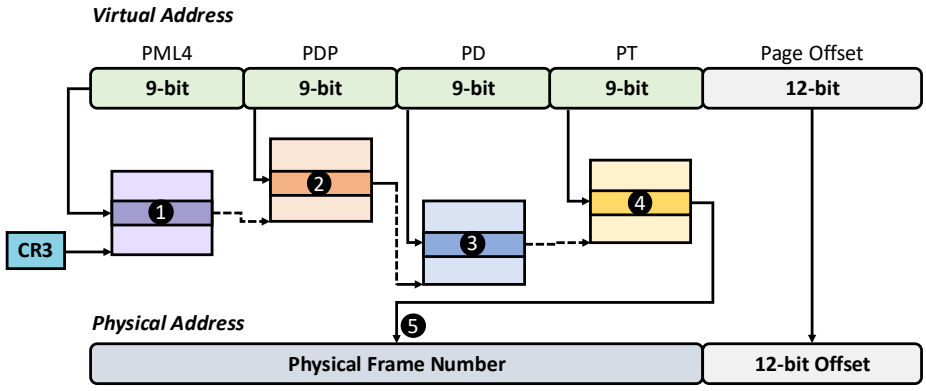


Figure 4.1.2: Four-level radix tree page table walk in x86-64. Figure taken from [73]

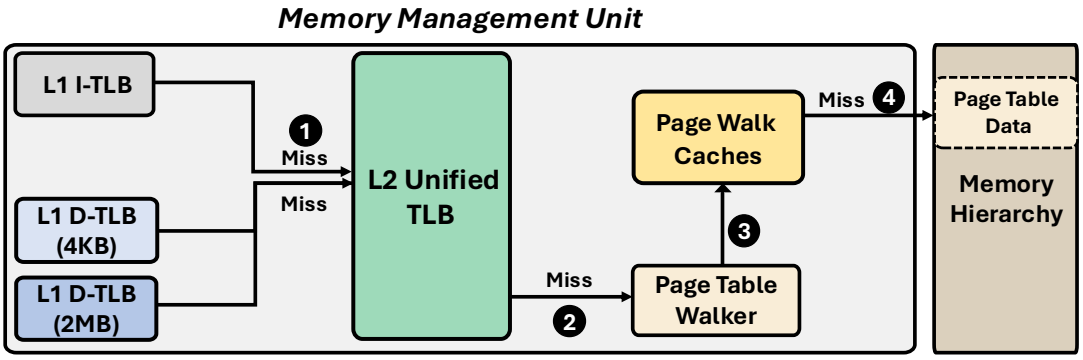


Figure 4.1.3: Structure of the MMU. Figure taken from [73]

hardware component to accelerate AT, the Memory Management Unit (MMU).

Figure 4.1.3 depicts the typical structure of the MMU. It consists of 3 main compoments:

1. A hierarchy of **Translation Lookaside Buffers (TLBs)**
2. A **Page Table Walker (PTW)**
3. A hierarchy of **Page Walk Caches (PWCs)**

TLBs are the first component of the MMU queried during an Address Translation request. They are multi-level caches that are used to store virtual-to-physical mappings for frequently used virtual pages. At the first level of the TLB hierarcy (L1 TLB) lie two distinct TLBs, one for instruction (L1I TLB) and one for data (L1D TLB) pages. On the contrary, the second level (L2 TLB) is unified, and contains both instruction and data pages.

Upon a miss in the L2 TLB, the Page Table Walker (PTW) is activated, a component that automatically performs a Page Table Walk for the requested virtual address. To accelerate the walk, the PTW caches intermediate levels of the PT in the PWCs, and only issues requests to main memory for nodes of the PT not present in the PWCs. Once the PTW has successfully retrieved the physical address, the translation is cached in the TLB.

4.1.4 On-Demand Paging and Page Fault Handling

The OS does not immediately load all virtual pages of a process into physical memory on process startup. Instead, it employs **Demand Paging**, loading each individual page when it is first accessed. The advantages of this strategy are twofold:

1. It enables more efficient physical memory usage, as pages from the VAS of a process that are never accessed will never be allocated in physical memory.
2. It allows for faster process startup compared to loading the entire VAS.

To enable Demand Paging, a hardware exception (**Page Fault**) is triggered by the MMU upon accessing a virtual page whose corresponding PTE is invalid (e.g., Present bit is clear) or violates access permissions. The OS kernel's page fault handler is subsequently invoked to allocate a physical page and create the virtual-to-physical mapping, or terminate the process in the case of access permission violation.

Page faults can be classified into two categories:

1. **Major Page Faults (Hard Faults)**, which are triggered when the required page data is not resident in physical memory and must be retrieved from secondary storage (e.g. disk, SSD). This can occur either for file-backed data not in the page cache or for data that has been swapped out.

2. **Minor Page Faults (Soft Faults)**, which are triggered when the page data is already present in physical memory, but the process's page table lacks a valid mapping for it.

Major faults occur upon access to file-backed data not in the page cache or to data that has been swapped out. On the other hand, there are several cases in which minor page faults can occur, namely:

1. *Initial Allocation (Lazy Allocation):* Such faults are caused upon access to a newly allocated anonymous memory region. This can happen, for example, when `malloc()` needs to allocate heap memory, or when expanding the stack. In this case, the kernel allocates a physical frame (often zero-filled) and creates the PTE mapping.
2. *Copy-on-Write (CoW):* Such faults occur after `fork()`. Initially, parent and child share physical pages mapped read-only. A write attempt by either process triggers a minor fault. The kernel allocates a new physical page, copies the original content, and updates the faulting process's PTE to map the new page with write permissions.
3. *Page Cache Hit:* Such faults occur when accessing a file-backed page whose data is already in the kernel's page cache (due to prior I/O or another process's access) but not yet mapped into the current process's VAS. The kernel simply creates the PTE to map the existing page cache frame.
4. *Shared Memory Attachment:* Such faults occur when mapping an existing shared memory segment whose pages are already in RAM. Similar to Page Cache hits, the kernel simply creates the PTE to point to the existing page.
5. *Reclaiming from Standby List:* Such faults occur when accessing a page that was recently removed from the process working set but remains cached in memory (e.g., on Linux's inactive list). In this case, the kernel also simply creates the PTE to map the existing page.

Major page faults are characterized by high latency due to time-consuming I/O operations. Minor faults, on the other hand, incur much lower latency, as they do not involve expensive disk I/O operations. However, the kernel operations involved, such as the context switch, VMA lookup and PTE modification, incur significant overhead, negatively impacting performance if frequent.

4.1.5 Memory Mapping Types and Management in Linux

Virtual Memory is organized in the Linux Kernel using **Virtual Memory Areas (VMAs)**, which are contiguous regions within a process's VAS sharing common properties (e.g. permissions, backing source). Each VMA is represented by a `vm_area_struct` structure containing the VMA's properties. VMAs are managed per-process and are typically stored using both a linked list and a balanced (red-black) tree, in order to enable efficient searching during fault handling, as well as during `mmap()` and `munmap()` operations.

The information contained in the VMA structures is essential for the page fault handler to identify the nature of the faulting address and determine the correct handling procedure. Specifically, the fault handler needs to distinguish between the following mapping types:

1. **Anonymous Mappings:** Such memory regions are not directly backed by a file (e.g. heap, stack, statically allocated memory or CoW pages). Access to anonymous pages can lead to both minor faults (namely *lazy allocation* and *CoW faults*, see 4.1.4) and major faults (for swapped out anonymous pages).
2. **File-backed mappings:** Such memory regions directly correspond to a file (e.g. `mmap()`ed files, executable code, shared libraries), and are managed by the **Page Cache**, a kernel cache in RAM holding file data blocks. In this case, whether the resulting page fault is major or minor depends the Page Cache: If the page is in the Page Cache, the fault is minor and the handler simply maps the existing page cache frame via the PTE. Otherwise, the fault is major and the handler first has to read the file block from the disk into the Page Cache.

4.1.6 Memory Management in Virtualized Environments

In virtualized environments, there exist three address spaces:

1. Guest Virtual Address (GVA), the address visible to applications
2. Guest Physical Address (GPA), the physical address space visible to the Virtual Machine
3. Host Physical Address (HPA), the physical address space of the actual machine

In such systems, two levels of address translation are involved:

1. GVA to GPA, handled by the guest OS using guest page tables
2. GPA to HPA, handled by the **Hypervisor** or **Virtual Memory Monitor (VMM)**

The hypervisor is responsible for allocating and managing HPA for VMs, mediating access to hardware MMU features (e.g. via Extended/Nested Page Tables or Shadow Page Tables), as well as handling hypervisor-level faults (nested page faults).

As there are two levels of Address Translation, there are also two types of Page Faults:

1. Guest-level page faults, handled by the Guest OS to establish the GVA to GPA mappings
2. Nested Page Faults (EPT/NPT faults), occurring during GPA to HPA translation. They require VMM intervention to establish the GPA to HPA mapping, potentially involving HPA allocation or fetching from hypervisor swap, adding significant overhead. Note that a single guest access can potentially trigger both a guest fault and a nested fault

4.2 RISC-V ISA

For the evaluation of our proposed design, we developed a prototype, which involved modifying a RISC-V core. This section is dedicated to background information regarding the RISC-V ISA and the core our design was based on [90].

RISC-V [100] is an open standard Instruction Set Architecture (ISA) based on Reduced Instruction Set Computer (RISC) principles. Developed by Berkeley, it was primarily conceived with the goal of being open-source, enabling processor design without royalties [101, 102]. It is a versatile ISA, not finetuned to a specific processor type, and maintains a significant degree of extensibility [101]; Indeed, RISC-V consists of a base ISA, along with a number of optional extensions (for e.g. floating-point and atomic operations). Furthermore, it has both 32 and 64 bit variants.

One notable example of RISC-V's extensibility is the `zicsr` extension. It defines a separate address space containing 4096 **Control and Status Registers (CSRs)** and provides instructions for accessing and modifying them. Some CSRs in this address space have already been allocated for special-purpose registers, such as timers and interrupt vectors, while others are reserved for implementation-specific purposes [103]. Although the `zicsr` extension provides instructions for controlling CSRs, their values may also change as side-effects of executed instructions (e.g. the `instret` counter, that counts the number of instructions retired).

4.2.1 Chipyard

Chipyard [104] is an open-source framework enabling development of RISCV systems-on-chip. It provides implementations of several cores [105, 90, 106] and other components, as well as tools that facilitate a number of different use cases, including software simulation of RTL designs using Verilator [107], FPGA-accelerated simulation [108], as well as automated VLSI flows [109].

Designs in Chipyard are implemented using Chisel [110], a Hardware Description Language (HDL) developed by Berkeley. It is built as a library on top of Scala. This means that Chisel code consists of scala programs whose execution produces a hardware graph. The key idea of Chisel is to equip hardware developers with object-oriented features, thus streamlining the RTL design process.

4.2.2 RocketChip

RocketChip [105] is a SoC generator framework integrated in Chipyard. It provides Chisel RTL implementations of a core, as well as various other components required to integrate the core into a fully-fledged SoC. An example of a RocketChip-based SoC is depicted in Figure 4.2.1.

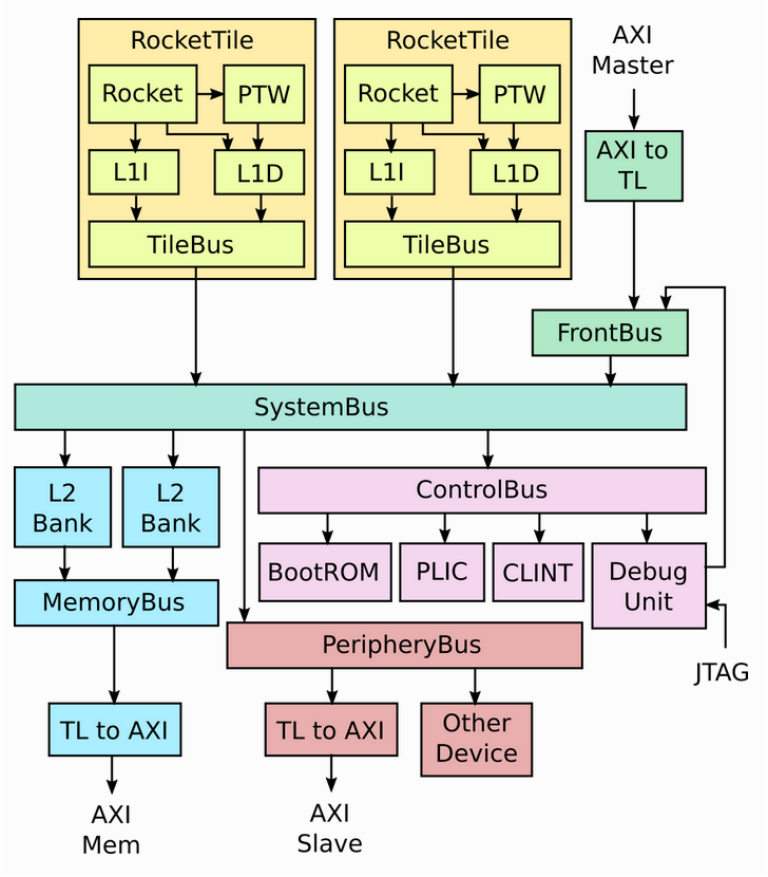


Figure 4.2.1: Diagram of a typical RocketChip system [105]

Components implemented in RocketChip include:

- The **Rocket Core**, a 5-stage in-order core which supports the RV64GC. It includes several components such as a Branch Target Buffer for branch prediction, as well as a TLB for address translation. The core is highly configurable, as several aspects of its design are parametrized
- The **Rocket Tile**, which integrates the RocketCore with L1 caches, as well as a Page Table Walker
- A memory system including L2 cache banks, and a memory bus connecting them to DRAM
- Several MMIO peripherals, including interrupt controllers, a first-stage bootloader, as well as a periphery bus for attaching further devices like NICs and block devices
- A bus for DMA device support

4.2.3 BOOM Core

Another core implemented inside Chipyard is the Berkeley Out-of-Order (BOOM) core [90]. It implements the RISC-V RV64GC ISA, and is designed to be high-performance and parametrizable. It

can replace the RocketCore inside RocketChip, adding features such as Out-of-Order execution. An overview of the BOOM core's pipeline is depicted in Figure 4.2.3.

4.2.4 RISCV-MINI

Apart from the high-performance BOOM core, Berkeley has also released RISCV-MINI [89], a smaller, 3-stage in-order core. It implements the RV32I ISA, meaning it does not include a floating point unit. It is primarily intended to be used for educational purposes, owing to the simplicity of its design.

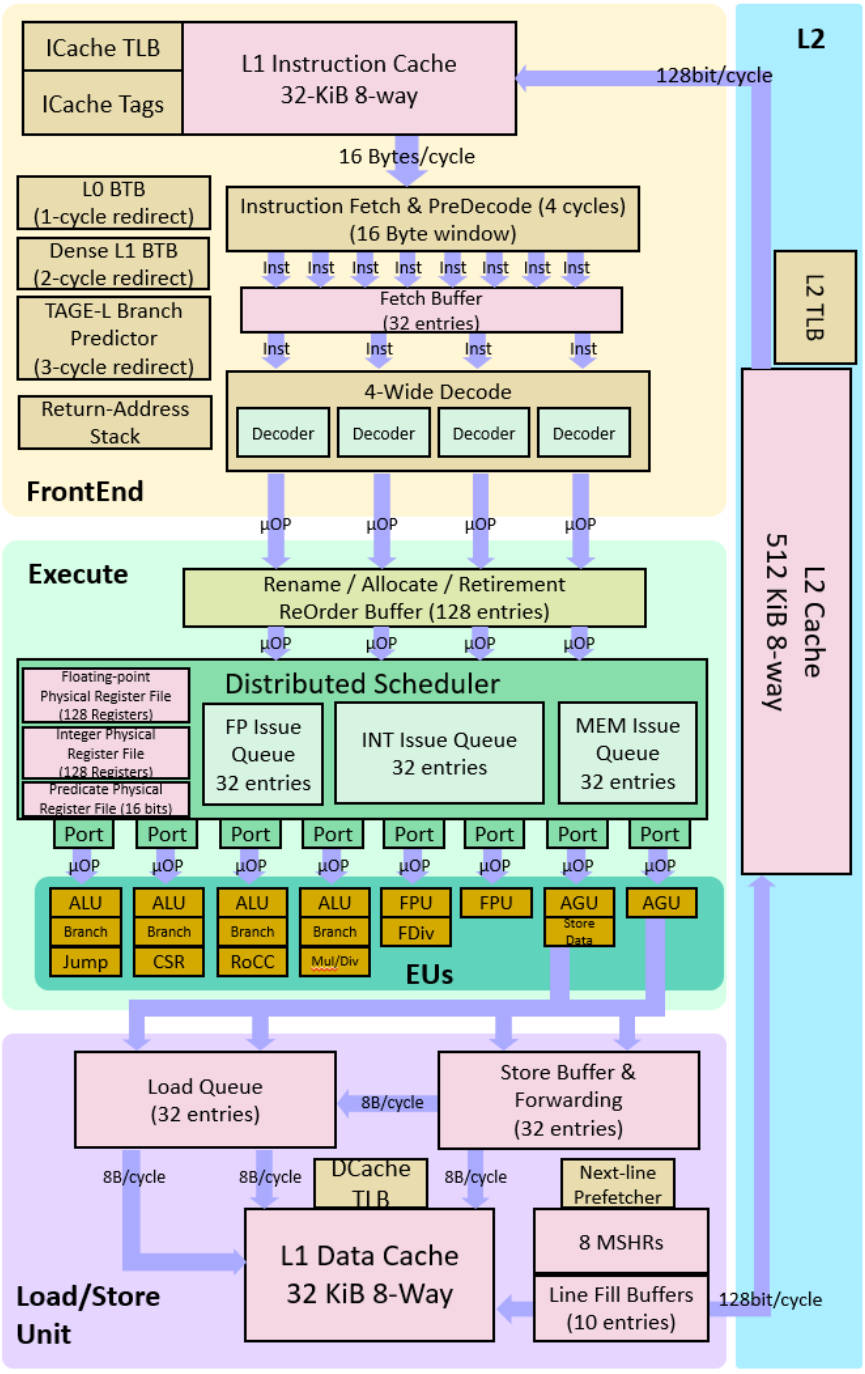


Figure 4.2.2: Overview of the BOOM core architecture [90]

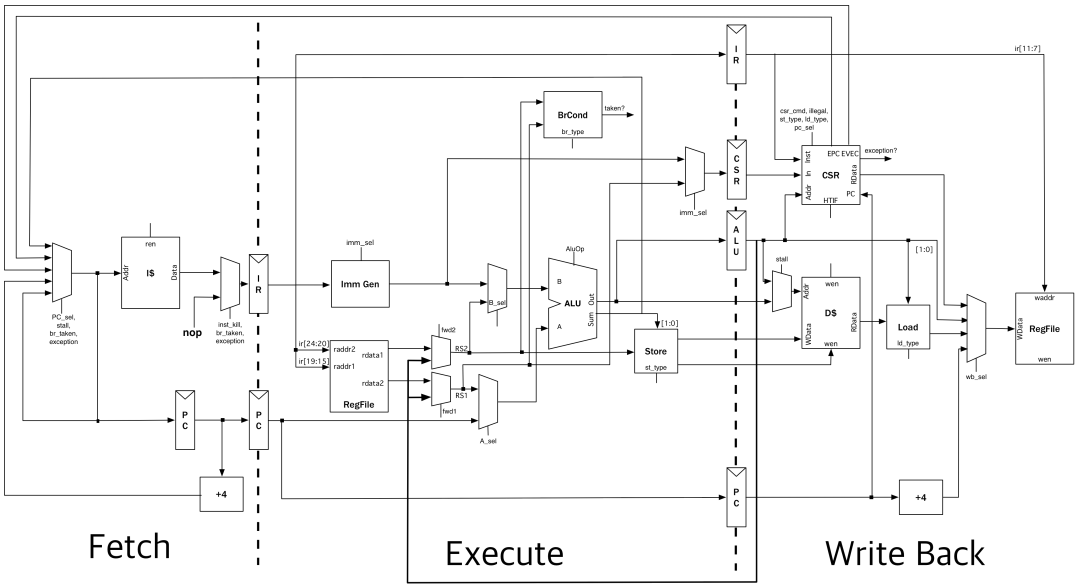


Figure 4.2.3: Overview of the RISCV-MINI architecture [89]

Chapter 5

Motivation

The implementation of virtual memory has remained largely software-centric. While acceptable for long-running, coarse-grained workloads, this reliance on software for handling page faults and managing physical memory introduces growing inefficiencies under emerging computational paradigms. The changing nature of workloads, characterized by short lifetimes, bursty execution, and high invocation frequency, exposes the mismatch between modern usage patterns and the conventional, OS-heavy memory management model.

5.1 The Shifting Nature of Workloads

Traditional applications such as scientific simulations, database servers, and large-scale graph analytics are typically long-running and exhibit large and stable memory footprints. They access large datasets irregularly but amortize the cost of memory management over billions of instructions. For such workloads, the latency and energy overheads of virtual memory operations such as minor page faults, and context switches, are effectively hidden by the abundance of computation and memory-level parallelism.

In contrast, the computing landscape is rapidly evolving towards *short-lived, latency-sensitive workloads*. Examples include *serverless functions*, *microservices*, and *large language model (LLM) inference queries*, which execute in bursts, perform minimal computation per data, and often terminate within milliseconds. These workloads allocate and release memory frequently and unpredictably, leaving little room to amortize the overhead of software-based allocation. As a result, each instance experiences significant startup penalties due to minor page faults and physical page allocation routines within the OS kernel.

5.2 The Cost of Software-Based Memory Allocation

To quantify the magnitude of this problem, we conducted detailed latency and energy profiling on an Intel Xeon system using an eBPF-based tracing tool and Intel RAPL counters. Figures 5.2.1 and 5.2.2 show the fraction of total execution time and system energy consumed by physical memory allocation routines across a variety of workloads. The results reveal that, on average, **32% of total execution time** and **21% of system energy** are consumed solely by physical memory allocation routines.

Each minor page fault invokes a multi-stage software pipeline comprising several kernel functions: `exc_page_fault()`, `do_user_addr_fault()`, `find_vma()`, `__handle_mm_fault()`, `do_anonymous_page()`, and `set_pte_at()`. This path performs permission checks, traverses the red-black VMA tree, allocates and zeroes a new page via the buddy allocator, installs the mapping into the page table, updates the LRU and accounting structures, and finally resumes userspace execution. The

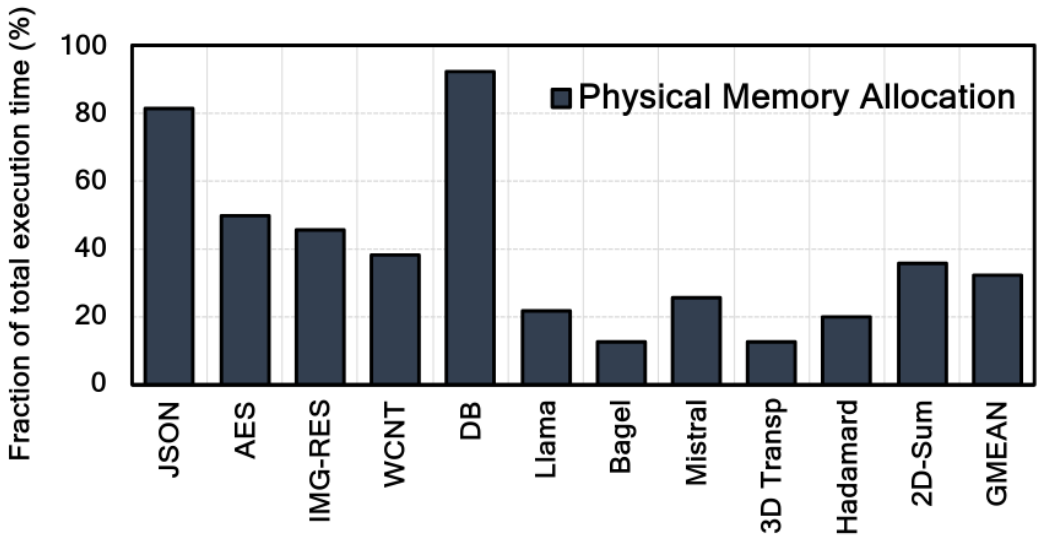


Figure 5.2.1: Fraction of total execution time spent on physical memory allocation routines.

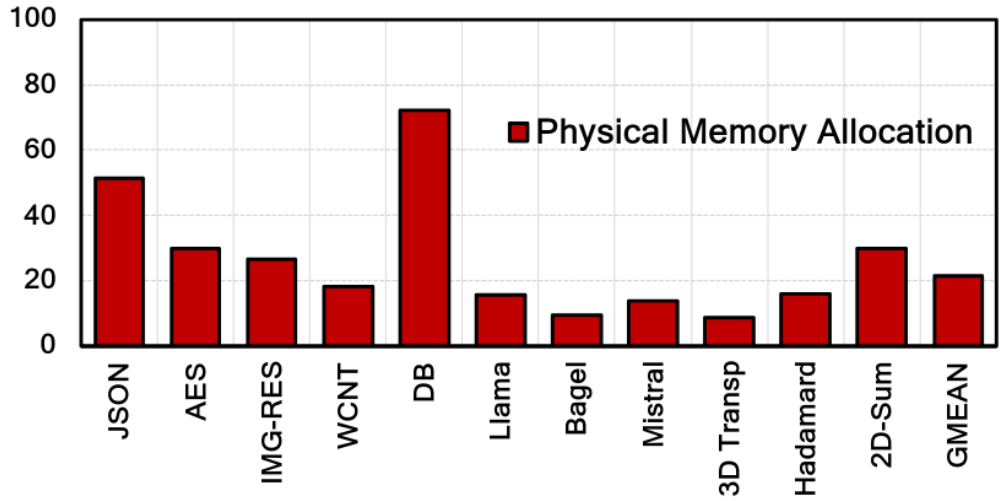


Figure 5.2.2: Fraction of total system energy consumed by physical memory allocation routines.

cumulative result is a lengthy and power-hungry operation that executes thousands of instructions on the out-of-order CPU core.

Figures 5.2.3 and 5.2.5 illustrate the breakdown of latency and energy consumption per minor fault for a microbenchmark that repeatedly allocates and accesses new memory pages. We make three key observations: First, each minor page fault incurs on average **5258 cycles**. Second, each fault retires over **6000 instructions**. Third, on average, a single minor fault consumes approximately **91 µJ** of energy. This means that the cost of initializing a single page can rival the energy of tens of thousands of useful memory accesses. To put this into perspective, to match the energy cost of a single minor page fault, a workload would need to perform almost 18K useful DRAM reads (assuming 5nJ per read). Such overheads are particularly prohibitive for ephemeral workloads, where each function instance may touch only a few pages before termination, making page initialization an *unamortized* and dominant cost.

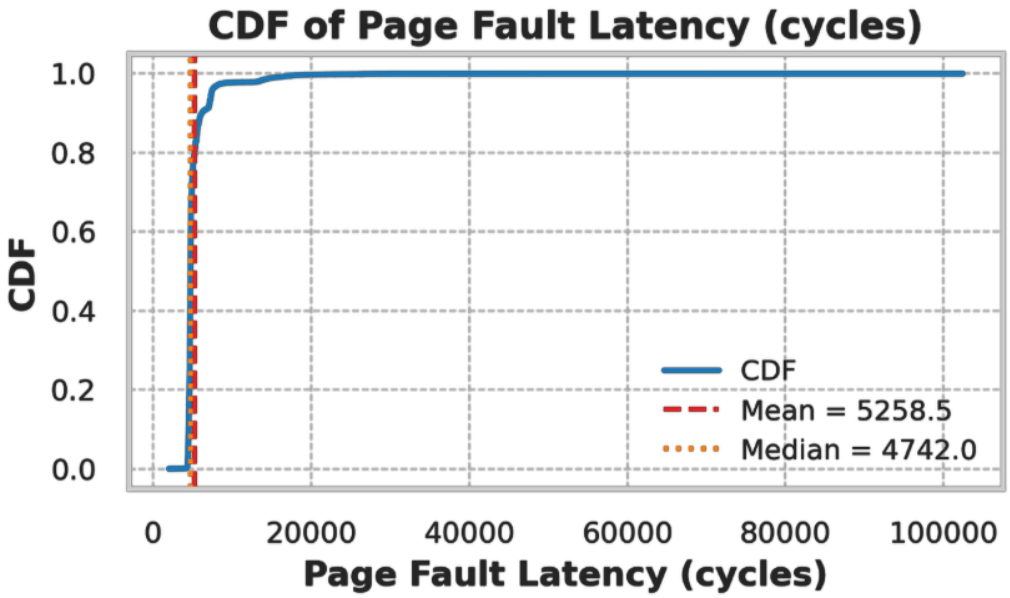


Figure 5.2.3: CDF of latency per minor page fault.

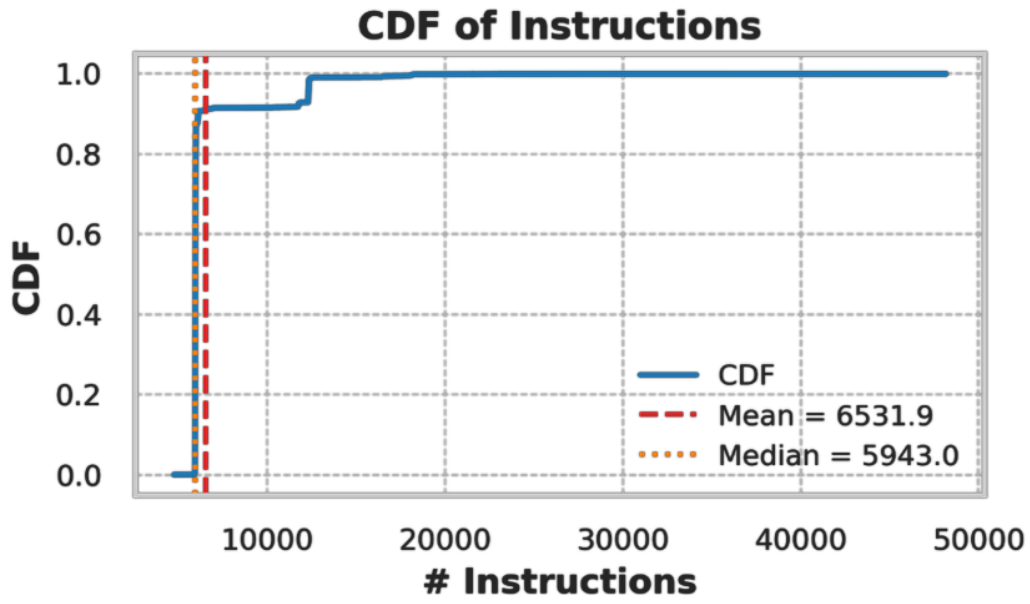


Figure 5.2.4: CDF of number of instructions per minor page fault.

5.3 Limitations of Hardware Delegation

Recent research efforts have explored hardware-assisted page handling and demand paging. These proposals introduce dedicated state machines or fixed-function units to accelerate certain memory management routines. While such hardware delegation can significantly reduce page fault latency, it

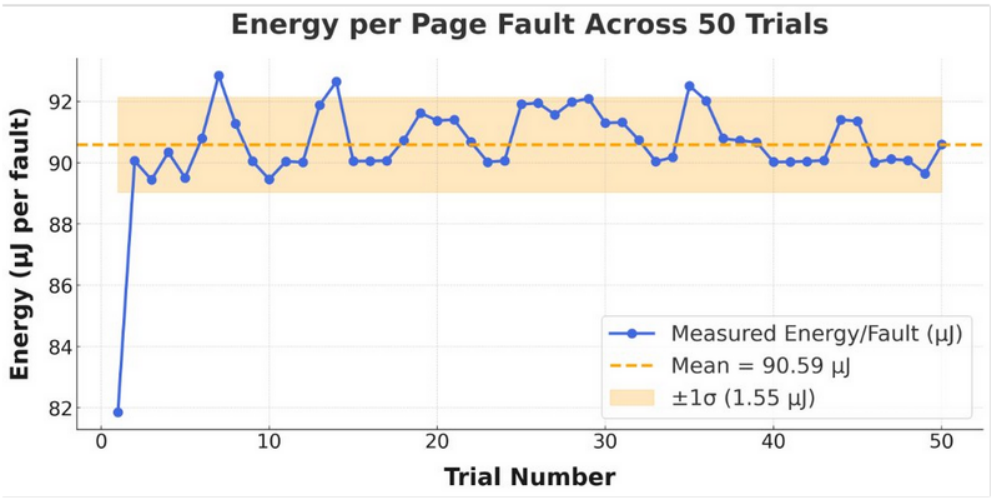


Figure 5.2.5: Energy consumption per minor page fault across different experiment trials.

does so by *removing* the OS from the critical path—thus sacrificing the flexibility and programmability that modern systems rely on.

In practice, hardware-only allocation policies result in 'greed-for-speed' behavior that blindly maps virtual to physical pages without considering higher-level goals such as: NUMA- or NUCA-aware placement, cache partitioning via page coloring, CXL-based disaggregation, rowhammer protection through guard-row allocation, or isolation in virtualized environments. These optimizations are critical for performance, security, and reliability, and they depend on semantic information that only the OS can provide.

5.4 The Opportunity for Programmable Allocation

The observations above expose a key opportunity for rethinking physical memory allocation. We argue that future systems require a *hardware–software co-designed* allocation substrate that combines the low-latency execution of hardware with the intelligence and adaptability of software.

Such a mechanism should:

- Deliver **hardware-class latency and energy efficiency** to reduce the overhead of page allocation and initialization.
- Preserve **software-level flexibility**, enabling the OS to specify policies for security, data placement, or performance isolation.
- Leverage **microarchitectural feedback** (e.g., DRAM bandwidth utilization, LLC occupancy, page access frequency) to guide intelligent placement decisions.

Chapter 6

Proposed Hardware-Based Allocation Engine

6.1 Valinor: Design Overview

In this work, we propose **Valinor**, a new hardware-OS cooperative memory allocation substrate that maintains the flexibility of software-based allocators while providing the low latency and energy efficiency of hardware-based mechanisms. Valinor introduces a programmable hardware allocation engine that the OS can program, using the desired allocation library, to offload memory allocation requests when needed, bypassing the traditional software-based path (and falling back to software only when needed). Such a design decision comprises three key advantages when it comes to memory allocation compared to solely software- or hardware-based approaches:

(1) **Benefits of Hardware-based Allocation.** In short-lived or latency-critical workloads—such as serverless functions, microservices, or LLM inference—page fault handling through the OS incurs expensive context switches. These stall pipelines, disrupt parallelism, and waste energy in out-of-order cores. Valinor allows the OS to bypass this slow path by delegating allocations to a dedicated hardware engine, significantly reducing both latency and energy consumption.

(2) **Custom and Intelligent Allocation Policies.** Many advanced memory management optimizations—such as page coloring, NUMA-aware placement, guard row allocation, or short-lived memory pools—require allocation decisions that directly shape virtual-to-physical mappings. Traditional hardware mechanisms perform "blind" mappings and cannot express such policies. Valinor lets the OS program semantic and placement rules into the engine, enabling fast allocation while retaining fine-grained control over translation behavior.

(3) **Allocation Policies Tailored to Dynamic Hardware Conditions.** Because the allocation engine has direct access to microarchitectural state (e.g., DRAM bandwidth, cache occupancy, address mappings), Valinor can adjust memory placement on the fly based on runtime feedback. For example, it can steer pages away from congested DRAM banks, partition caches dynamically, or balance allocations across NUMA domains. This enables adaptive, data-driven allocation strategies that are prohibitively expensive to realize in software alone.

Figure 6.1.1 illustrates Valinor's end-to-end workflow and integration with the OS. First, the application expresses its high-level semantic requirements (e.g. "I mostly allocate short-lived objects") by linking against a custom memory allocation library (e.g. tcmalloc [87], jemalloc [**jemalloc**], or mimalloc [**mimalloc**]). The OS chooses the appropriate library based on the application's needs (i.e., the ultra-fast allocation library for latency-critical workloads in our example). The OS configures the hardware allocation engine to implement the library's allocation policy (e.g., updates a stack pointer so that the allocation engine can load the library). Next, when the application performs a memory access

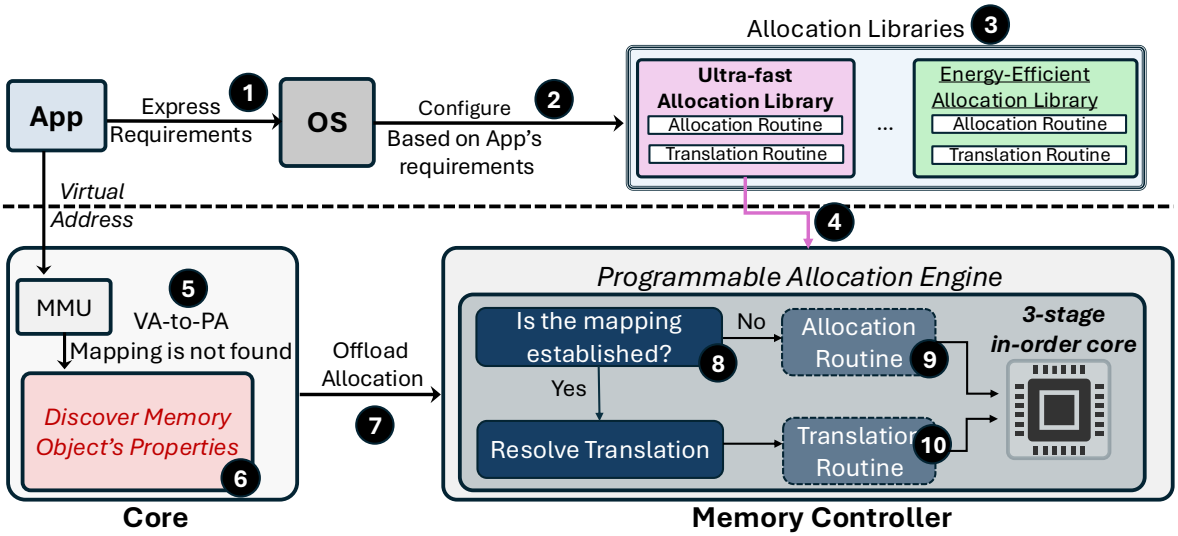


Figure 6.1.1: Valinor's architecture and integration with the OS. The OS can program the hardware allocation engine to handle memory requests directly, bypassing the traditional software path.

and the core's Memory Management Unit (MMU) encounters a TLB miss and the virtual-to-physical translation is not present in the page table, the core's Page Table Walker (PTW) raises an exception since the mapping has not been established yet or it has been previously established by the allocation engine (which uses its own data structure to keep track of it). Next, the core first discovers the properties of the memory object being accessed, e.g., whether it is file-backed or anonymous, and its access permissions (we explain in detail the property discovery process in **??**). If the memory object satisfies the criteria for hardware allocation, the core invokes the hardware allocation engine. The allocation engine first determines if the requested page has already been allocated (e.g., in a previous access). If so, it returns the corresponding physical page number and access permissions to the core, resolving address translation. Otherwise, it allocates a new physical page according to its pre-programmed policy (e.g., from a dedicated memory region for short-lived objects), updates its internal data structures, and returns the new mapping to the core. The core then updates its TLB with the new mapping and resumes execution. If the memory object does not satisfy the criteria (e.g., it is file-backed), the core falls back to the traditional OS page fault handler. Finally, when the application frees memory, the OS informs the allocation engine to update its internal data structures accordingly.

6.2 Valinor: Detailed Design

6.2.1 Programmable Allocation Engine

The primary component of Valinor is a programmable allocation engine, whose goal is to handle allocation (and, possibly, translation). It resides near the L2 cache and is controlled by the PTW.

Key functionality

To provide the desired functionality, the allocation engine needs to be able to execute the following functions:

1. `page_fault_handler(pid, vpn, prot)`: Allocate a physical page for a virtual page with a specific `vpn` and permissions `prot` for a process with a specific `pid`
2. `address_translation_handler(pid, vpn)`: Determine whether page with virtual page number `vpn` has been mapped by the allocation engine (but is not recorded in the OS's Page Table) for process `pid`. If so, return the corresponding physical page number and permissions

3. `free(pid, vpn_start, vpn_end)`: Delete all mappings for pages with `vpn` within the provided range for process `pid`
4. `ioctl(info)`: Reserved for initialization (e.g. informing the allocation engine about library-specific parameters)

The first two operations are triggered by hardware events, as discussed in 6.2.3, whereas `free` `ioctl` are triggered by software. For this reason, the ISA needs to be extended with two new instructions.

Interface with the PTW

Since the set of operations implemented by the allocation engine, as well as their input and output format, are both 1. fixed and 2. limited in complexity, the allocation engine is controlled by the PTW through an interface in hardware, which comprises the following signals:

1. `enable`: 1-bit input signal. The allocation engine can only operate when it is set
2. `mode`: 2-bit input signal. Determines which function out of the ones described in the previous paragraph will be executed
3. `vpn`: 64-bit input signal, the Virtual Page Number of the address to translate or allocate (for functions 1. and 2.), the start of the range of addresses to free (for function 3.), or the information required for ioctl (function 4.)
4. `prot`: 32-bit input signal: The permission bits of the virtual address being translated (for functions 1. and 2.)
5. `pte`: 64-bit output signal. The result of address translation (for functions 1 and 2), or a signal indicating successful completion (for functions 3. and 4.)

6.2.2 Virtual Memory Area Filter

As mentioned in 6.2.1, the allocation engine requires information about the permissions of the virtual memory area the requested virtual address belongs in. Furthermore, depending on the allocation engine's implementation, there may be a need to restrict the page faults being handled. For example, to avoid complex DMA operations, some implementations of the allocation engine may restrict themselves to minor page faults. For these reasons, we introduce a hardware module, the Virtual Memory Area Filter (VMAF), whose goal is twofold:

1. Filter the virtual addresses for which the allocation engine should be activated
2. For addresses that the allocation engine should handle, retrieve the access permissions of the corresponding Virtual Memory Area

The exact design of the VMAF depends on the needs of the system's designer, as well as the particular way information about Virtual Memory Areas is maintained by the OS. For example, in older versions of the Linux Kernel (before 6.1), `vm_area_structs` are contained in a per-process red-black tree. Therefore, the VMAF has to perform a binary tree walk to discover the desired properties.

The VMAF is invoked by the PTW upon an unsuccessful address translation attempt, as described in detail in 6.2.3.

6.2.3 End-to-End Workflow Example

Address Translation

Upon an L1 TLB miss, the TLB sends a request to the Page Table Walker (PTW) to provide a translation for virtual address VA (1). The PTW begins the traditional radix tree walk, trying to find a Page Table Entry (PTE) in the OS's Page Table (2). If a valid PTE is found, it is returned as a response to the L1 TLB (3). Otherwise, the PTW activates the allocation engine in Address Translation mode, passing VA as a parameter (4). The allocation engine then runs its `address_tranlation_handler` to

attempt to translate VA (5). If translation is successful, the allocation engine creates a valid PTE by combining the page's PPN and the corresponding permissions, documented in the engine's metadata, and returns it to the PTW, which subsequently passes it to the L1 TLB as a response (6). In this case, no page fault is triggered. Otherwise, the allocation engine returns a null PTE to the PTW, which then initiates the Page Fault handling procedure described below.

Page Allocation

After a failed Address Translation (as described in the previous paragraph), the PTW activates the VMAF, which subsequently walks the current process's VMA tree with VA as input (7). If the tree walk fails, i.e. the VMAF recognizes that the VA was invalid, the PTW returns a null PTE to the L1 TLB and raises and Access Exception (8). If the VMAF determines that the VA should not be handled by the allocation engine, then the PTW signals a Page Fault, which is then handled by the OS using the traditional handler (9). Otherwise, the VMAF provides the access rights *prot* to the PTW. The PTW then activates the hardware allocation engine again, but this time in Page Fault mode, passing VA and *prot* as parameters (10). The allocation engine then tries to allocate a page based on its pre-programmed scheme. If allocation is successful, the allocation engine updates its internal structures (or the OS's Page Table, depending on the implementation) so as to be able to keep track of the newly-allocated page and its access rights, and combines its PPN with *prot* to create a valid PTE, which is then returned to the PTW and passed to the L1 TLB as a response (11). Therefore, the L1 TLB only sees a successful translation, and is oblivious to the fact that a new page was allocated. If, however, allocation is unsuccessful, the allocation engine returns a null PTE to the PTW, which then signals a Page Fault.

Page Freeing

Pages are returned to the system using `munmap()` for an address range $[VA_{start}, VA_{end}]$. This function is augmented with a call to the new `free` instruction (see 6.2.1), taking VA_{start} and VA_{end} as arguments (12). These arguments are communicated by the core to the PTW, which then activates the PTW in Free mode and passes VA_{start} and $VA_{end}]$ as parameters. The allocation engine then updates its metadata, invalidating any mappings from this virtual address range that have been allocated by the allocation engine itself (13). While the `free()` function executes, the pipeline is stalled. Once the allocation engine finishes operation, it sends a signal to the PTW, which then unstalls the pipeline. Afterwards, `munmap()` continues execution as normal, invalidating all PTEs in the process's Page Table corresponding to the virtual address range $[VA_{start}, VA_{end}]$ (14). Therefore, at the end of `munmap()` both mappings handled by the hardware and the OS have been deleted.

6.2.4 Software- and OS-level support

When a userspace program needs to use Valinor with a particular allocation library, the programmer has to inform the system about the specific library being used, so that the OS can handle initialization operations. For this reason, we introduce three new syscalls, namely `valinor_install_lib(lib)`, `valinor_start()` and `valinor_end()`. The first of these signals to the OS that a particular library should be used in the allocation engine, while the other two indicate the start and end of a region of code handled by Valinor.

To implement the aforementioned syscalls, the OS needs to provide the allocation engine with two different types of information, namely:

1. Library-specific metadata. For example, if the library does not include its own translation structures, but instead uses the OS's Page Table, the allocation engine requires a pointer to the root of the Page Table

2. Library-independent data, namely the location of the library being used and a signal to enable and disable the allocation engine

Name	Summary	Allocation Routine	Translation Routine	OS Sync
Ultra-fast (UF)	Low-latency hash-based allocation for short-lived memory objects; Optimizes fast path and tail latency.	Leverages a memory segment organized in a set-associative manner for quick allocations.	Handles hash-based translation on L2 TLB miss path.	One large segment allocated at library boot.
Runahead	Pre-allocates pages asynchronously to hide future allocation stalls.	Background pre-allocator fills a pool of free physical pages; foreground allocates from this pool.	Separate page-table for pre-allocated pages.	One large segment allocated at library boot.
Real-time	Targets deterministic timing with bounded worst-case latency.	Pre-reserved linear physical pages for real-time memory objects.	Offset-based translation for real-time region.	Requesting memory for the first fault in the region.
Telemetry-aware	Uses hardware counters (e.g., BW, cache occupancy) to steer data placement.	Policy selects banks based on telemetry using a multi-hash-based allocator.	Handles hash-based translation on L2 TLB miss path.	One large segment allocated at library boot.
Energy-efficient	Reduces energy via low-frequency background allocation.	Async pre-allocation in a low-power state; foreground allocates from this pool.	Separate page-table for pre-allocated pages.	Rare (for pool extensions)
Rowhammer Protection	Protects against Rowhammer by inserting guard rows between sensitive allocations.	Allocates physical pages with guard-row spacing via page coloring;	Handles color-based translation on L2 TLB miss path.	One large segment allocated at library boot.
Integrity-Enforcing	Enforces integrity to make sure the virtual-to-physical mapping is not tampered with.	Allocates from a set-associative segment stores additional merkle tree metadata.	Verifies integrity on L2 TLB miss path using merkle tree stored in reserved pages.	One large segment allocated at library boot.
Placement-aware	Optimizes locality and contention (e.g., bank conflicts).	Bank-level free lists for placement-aware allocation & Telemetry awareness.	Handles bank-aware translation on L2 TLB miss path.	One large segment allocated at library boot.

Table 6.1: Representative allocation libraries and policies that can be programmed into Valinor's allocation engine. Each policy targets different goals: performance (UF), pre-allocation (RA), real-time determinism (RT), telemetry-guided adaptation (TA), energy efficiency (EE), Rowhammer protection (RH), integrity enforcement (INT), and placement optimization (PA).

Library-specific metadata is provided during the `valinor_install_lib()` syscall, through repeated calls to Valinor's `ioctl` instruction, described in 6.2.1. Library-independent data, on the other hand, can be profided using special-purpose memory-mapped registers. One register is needed to contain the location of the allocation library, updated during `valinor_install_lib()`, while another one is needed to serve as an enable signal to the allocation engine. The enable signal is set and cleared during `valinor_start()` and `valinor_end()`, respectively.

6.3 Use Cases

Table 6.1 summarizes the different use cases we explore in this work.

To showcase the benefits of Valinor's flexibility, below we present several different allocation schemes, each serving a different objective. To further exhibit the ease of programmability of our proposed mechanism, we provide pseudocode for all allocation schemes presented.

6.3.1 Latency-First Allocation Library (LFA)

The primary objective of LFA is to prioritize very fast allocation. To this end, we employ a hash-based allocation scheme in a dedicated contiguous allocation pool of pages, which only the programmable engine can use for allocation.

Allocation: The allocation engine's programmer defines hash function H mapping addresses to positions in the allocation pool. Upon a request to translate a virtual address with virtual page number

VPN, the allocation engine calculates the value $n = H(VPN)$ to map the virtual address to a specific page within the allocation pool. Then the allocator looks at the n-th position in the `tag_array` to determine whether there is a valid VPN already mapped to this page. If the page is free, then the allocator updates the n-th position in the `tag_array` with VPN and returns the ppn corresponding to the n-th position in the allocation pool. Otherwise, allocation is deemed unsuccessful and a page fault is triggered.

Address Translation: Since the Page Table is not updated by the page fault handler, the radix walk in the Page Table Walker (step 2 in 6.2.3) will always fail for pages in the allocation pool. For these pages, therefore, the allocation engine's own address translation handler will be invoked. For a page with virtual page number VPN, the allocation engine will once again compute the hash $n = H(VPN)$ and look at the corresponding entry in the `tag_array`. If the vpn contained in `tag_array[n]` matches VPN, then address translation is deemed successful the allocation engine returns the ppn corresponding to the n-th position in the allocation pool. Otherwise, a page fault occurs.

The allocation pool can be reserved by the OS at boot time (e.g. using `memblock_reserve()`) and its size can be communicated to the allocation engine via its `ioct` function (described in 6.2.1).

Overall, this scheme avoids the overheads of a) Page Table updates, b) the complicated buddy system, at the cost of increased Address Translation overhead. Furthermore, the choice to use an isolated allocation pool eliminates the need for complex synchronization between the OS and the allocation engine, as well as coherence, if the allocation engine has a cache.

```
PPN translate(VPN v):
  for (int i=0; i< n_hashes; i++):
    Set s = hash_i(v) & (NUM_SETS-1);
    if (seg[s].tag_match(v)) return seg[s].ppn_of(v);
  return allocate(v); // allocate on miss

PPN allocate(VPN v):
  for (int i=0; i< n_hashes; i++):
    Set s = hash_i(v) & (NUM_SETS-1);
    if (seg[s].has_free()) return seg[s].pop_free();
  return FALLBACK; // rare slow path
```

Listing 6.1: UF: allocation and translation

Real-time (RT)

Provides tight WCET. Admission reserves per-task linear regions and installs *region descriptors*. Allocation is $O(1)$ ring-buffer; translation is arithmetic: $PPN = base\_PPN + ((VA - base\_VA) \gg page\_shift)$. No page faults after activation.

```
PPN rt_alloc(Task t) {                    // per-task, per-size ring
  idx = t.ring.head; t.ring.head = (idx+1) & (t.ring.cap-1);
  return t.base_ppn + idx;                // no zero on fast path
}

PPN rt_translate(VA va, Desc d) {         // linear region descriptor
  return d.base_ppn + ((va - d.base_va) >> d.page_shift);
}
```

Listing 6.2: RT: allocation and translation

Telemetry-aware (TA)

Placement decisions adapt to hardware feedback (bandwidth, LLC occupancy). Allocation selects a bank/color and one of multiple hash targets; translation mirrors the selected hash/bank mapping on an L2 TLB miss.

```
PPN ta_alloc(VPN v) {
```

```
  Bank b = policy_pick_bank(counters());       // telemetry-guided
  for (int i=0; i<NHASH; i++) {
    Set s = hash_i(v, b) & (SETS_PER_BANK-1);
    if (bank[b].seg[s].has_free()) return bank[b].seg[s].pop_free();
  }
  return FALLBACK;
}

PPN ta_translate(VPN v) {
  Bank b = policy_bank_hint();                 // mirrored hint
  for (int i=0; i<NHASH; i++) {
    Set s = hash_i(v, b) & (SETS_PER_BANK-1);
    if (bank[b].seg[s].tag_match(v)) return bank[b].seg[s].ppn_of(v);
  }
  return MISS;
}
```

Listing 6.3: TA: allocation and translation

Energy-efficient (EE)

Energy-first via low-frequency, batched preallocation (DVFS-friendly); the foreground pops from the pool. Translation uses a compact pool page-table/tag array to minimize page-table churn.

```
PPN ee_alloc(void) {
  if (pool.free_cnt) return pool.pop();        // no trap, no zero
  if (unlikely(!refill_deferred())) return FALLBACK;
  return pool.pop();
}

PPN ee_translate(VPN v) {
  return pool_tags.lookup(v);                  // tiny tag array / PT
}

// Background: large, sequential batches to cut ACT/PRE energy
void ee_refill_batch() {
  for (int k=0; k<BATCH && bw_ok() && power_ok(); k++) {
    PPN p = buddy_get_page(); zero_seq(p); pool.push(p);
    pool_tags.install(v_hint_for(p), p);
  }
}
```

Listing 6.4: EE: allocation and translation

Integrity-Enforcing (INT)

Ensures translations cannot be tampered with. Allocation occurs in a reserved set-associative segment whose metadata is authenticated (Merkle tree). Translation on L2 TLB miss verifies the Merkle path before returning the PPN; only the engine can install/modify entries.

```
PPN int_alloc(VPN v) {
  Set s = hash(v) & (NUM_SETS-1);
  PPN p = seg[s].pop_free();
  seg[s].install_tag(v, p);
  merkle_update(s, v, p); // commit auth
  return p;
}

PPN int_translate(VPN v) {
  Set s = hash(v) & (NUM_SETS-1);
  if (!seg[s].tag_match(v)) return MISS;
  if (!merkle_verify_path(s, v)) return FAIL_AUTH;
  return seg[s].ppn_of(v);
}
```

Listing 6.5: INT: allocation and translation

Chapter 7

Experimental Evaluation

7.1 Evaluation Methodology

7.1.1 RISC-V Soft-core Prototype

We prototyped our proposed design and tested it on a Xilinx ZCU106 FPGA [88], so as to i) showcase the feasibility of implementation of our key mechanism and ii) evaluate the performance benefits of Valinor on a real system. Table 7.1 summarizes the configuration used for our prototype.

Technical details of the prototype

We decided to implement the LFA library, described in 6.3.1, as the lack of communication between the allocation engine and the OS simplified the implementation significantly. Furthermore, in order to avoid complex I/O operations, as well as page cache lookups, we restricted ourselves to handling anonymous minor faults.

To model our reconfigurable unit, we used the RISCV-MINI processor [89] (see 4.2.4). We integrated this processor within the translation subsystem of the BOOM Out-of-Order core [90] (see 4.2.3) to accommodate allocations in hardware. Specifically, we modified the Finite State Machine in RocketChip's PTW, adding new states to wait for the allocation engine to handle each of the operations described in 6.2.1, as well as a state to wait for the VMAF. In order for the PTW to be able to control the allocation engine, we modified the RISCV-MINI core's fetch stage, so that 1. the program counter would only advance if a PTW-controlled bit was set and 2. upon detecting a rising edge in the enable bit, the program counter would jump to a specific location depending on the type of operation requested. Furthermore, we modified the RISCV-MINI core's cache implementation, so that misses would be handled by RocketChip's L1 Cache.

We modeled the VMAF using a fixed circuit, shown in Figure 7.1.1. Since the version of Linux we used (5.11.6) still uses red-black trees to store `vm_area_structs`, our VMAF is essentially a hardware binary tree walker. After discovering the location of the `vm_area_struct`, the VMAF first checks the `vm_ops` field. A null value in this field corresponds to an anonymous mapping. Therefore, if the `vm_ops` field is not null, the VMAF returns that a page fault should be triggered. Otherwise, it retrieves the `vm_page_prot` field, which contains the permissions for this particular VMA and returns them to the PTW. To accelerate the VMA discovery process, we also included a Content Addressable Memory with enough entries for 16 VMAs. Each entry in the CAM contains the bounds of the `vm_area_struct`, namely the `vm_start` and `vm_end` fields, as well as the aforementioned `vm_page_prot` and `vm_ops` fields. The CAM uses LRU as its eviction policy.

The prototype ran the v.5.11.6 of the Linux kernel. In order to accommodate the allocation engine, we had to modify the kernel as described in 6.2.4. For hardware measurements such as page fault latency for allocations handled by RISCV-MINI or cache misses in RISCV-MINI, we added CSRs to the core,

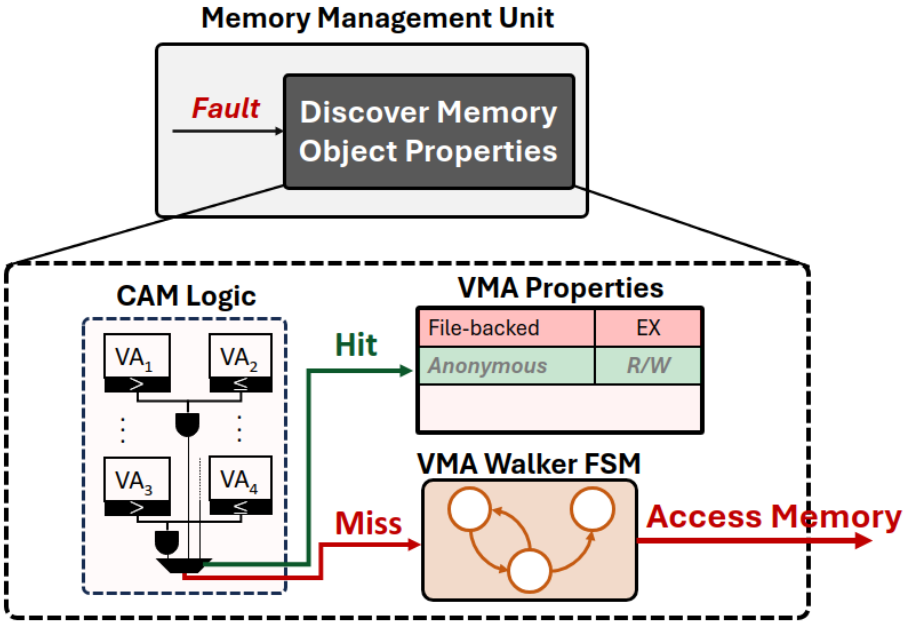


Figure 7.1.1: Overview of our implementation of the VMAF

initialized as zero at the start of every benchmark's execution and updated by appropriate hardware signals.

Table 7.1: Prototype Configuration

Prototype Configuration	
Core	64-bit RISC-V BOOM Out-of-Order core
MMU	L1 I-TLB: 32-entry, direct-mapped, 1-cycle latency
	L1 D-TLB (4 KB): 8-entry, fully assoc, 1-cycle latency
	L2 TLB: 512-entry, direct-mapped, 2-cycle latency
	Page Walk Cache: 8-entry, fully assoc, 1-cycle latency
L1 Cache	L1 I/D-Cache: 2 KB, 4-way assoc, 3-cycle access latency
	LRU replacement policy
Linux Kernel	v5.11.6
Allocation engine	Core: RISCV-MINI 32-bit in-order core [89]
	L1 I/D-Cache: 4KB, direct-mapped
FPGA	Xilinx ZCU 106 [88], 1GB DDR4

Fixed Hardware baseline

In order to quantify the cost of using a reconfigurable unit instead of a fixed ASIC, we implemented another design on the FPGA, containing a fixed hardware module that also handled allocations with the LFA library. The communication with the PTW is virtually identical in this version, and the same holds for the VMAF. When asked to translate a virtual address, the hardware module calculates the address of the set in the `tag_array` where the address should be placed, and retrieves all ways of the corresponding set one by one. If a match with the requested address is found, the module returns the corresponding PTE. Similarly, when the module receives an allocation request, it examines one by one the ways of the set in the `tag_array` corresponding to the requested virtual address. If an empty entry is found, the hardware module updates the field in the `tag_array` to allocate a physical page. To accelerate this process, the module also includes a cache containing all ways of recently accessed sets of the `tag_array`. The cache is direct-mapped, and has space for 64 sets of the `tag_array`. Note

that the associativity of the `tag_array` was fixed at 4 for this implementation.

Unlike the RISCV-MINI-based implementation, in this version there was no need for an `ioctl()` call to inform the hardware of parameters like the size of the allocation pool. Instead, in this implementation we simply used more CSRs to communicate such information to the hardware. We also used a CSR to indicate the starting address of the `tag_array`.

Evaluated systems

We evaluate 4 different systems on the FPGA:

1. **Baseline:** An unmodified BOOM processor running the unmodified Linux Kernel
2. **SW-only:** An unmodified BOOM processor, with a modified Linux kernel that implements the allocation scheme of the LFA Library described in 6.3.1 solely in software. At the start of the page fault, the page fault handler first tries to allocate memory using the simplified allocation scheme; if that fails, then it falls back to the standard allocation path of the Linux kernel
3. **Fixed-HW:** The Fixed Hardware baseline described above
4. **Valinor:** Our proposed design, implemented using RISCV-MINI as described above

Workloads. We evaluated our FPGA prototype against 5 microbenchmarks. We chose several short-running applications, including a set of simple matrix and vector operations (`spmv`, `vvadd`, `median`, `transpose`), as well as a serverless function (`cJSON`), all of which exhibit large memory allocation overheads and short computation times. Due to limitations of the FPGA prototype, we performed a scaled-down study, tuning all microbenchmarks to exhibit a memory footprint between $14MB$ and $16MB$, and experimenting with different allocation pool sizes between $1MB$ and $16MB$.

7.2 Evaluation Results

7.2.1 RISC-V prototype

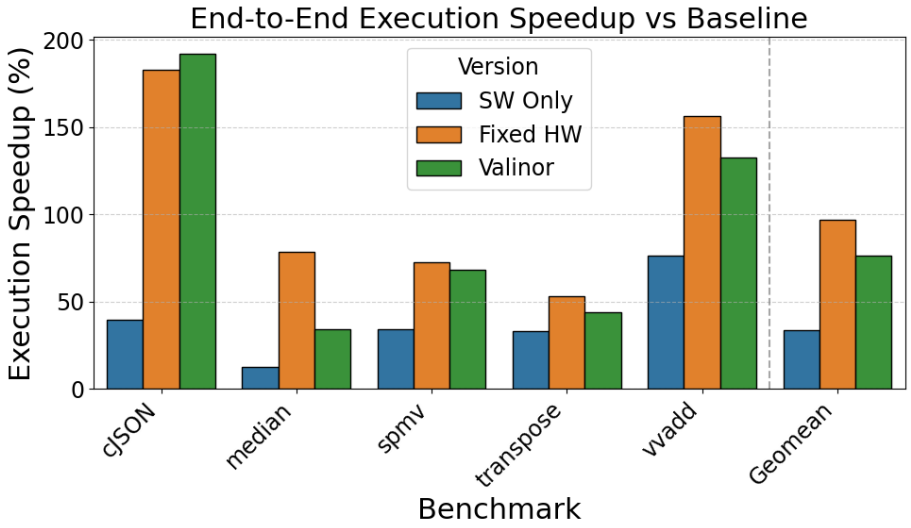


Figure 7.2.1: End-to-End execution speedup for different benchmarks.

Performance evaluation. Figure 7.2.1 shows the end-to-end performance speedup measured on the FPGA prototype for our evaluated schemes across several short-running microbenchmarks. The size of the allocation pool is fixed at $16MB$, while all benchmarks have a memory footprint between $14MB$ and

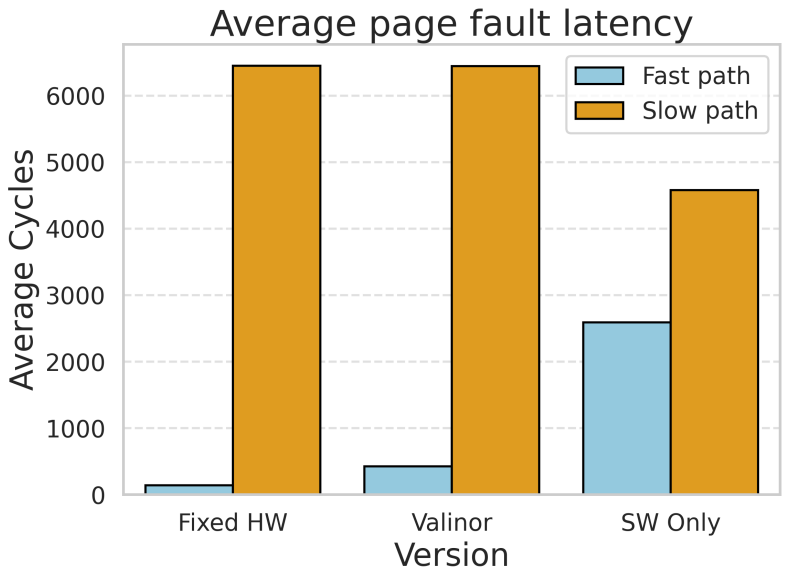


Figure 7.2.2: Average number of cycles taken per page fault for each of our evaluated designs. "Fast path" refers to page faults handled by the custom mechanism in each case, whereas "Slow path" refers to the fallback path, i.e. the default Linux page fault handler.

$16MB$, so that they completely fit into the allocation pool. We make three key observations: (i) Despite making no modifications to the hardware, the **SW-only** version already leads to a geomean speedup of 34%, which is indicative of the allocation scheme's efficiency compared to the buddy allocator, (ii) the **Fixed-HW** version is the fastest, achieving a geomean speedup of 97%, thus showcasing the benefits of offloading page fault handling in hardware, (iii) **Valinor** consistently outperforms both the baseline and the **SW-only** version, with a geomean speedup of 77%. Therefore, it can be seen that, despite employing a general-purpose reconfigurable core, **Valinor**'s performance is still comparable to that of a fixed hardware module, as further quantified in the following paragraph.

Page fault latency. Figure 7.2.2 shows the average number of cycles spent on the different page fault paths. In all three designs, "Fast path" refers to the page faults handled by the proposed mechanism (i.e. in-hardware for **Fixed-HW** and **Valinor** and using the LFA scheme for **SW-only**), whereas "Slow path" corresponds to the fallback path, i.e. the Linux Page Fault handler using the buddy system. We see that merely successfully allocating with the LFA scheme and skipping the buddy allocator (as is the case in **SW-only**) already speeds page fault handling up by 1.77×. On the other hand, bypassing the page fault handler altogether and delegating allocation to fixed hardware is 45× faster than the page fault handler. Finally, although **Valinor**'s in-order reprogrammable core still takes almost 3× longer than **Fixed-HW** to handle a fault, it still achieves an order-of-magnitude speedup (15.2×) compared to the default software page fault handler. We therefore see that, even in the case of the complex reconfigurable fabric, establishing virtual-to-physical mappings in hardware can lead to significant speedup, bypassing the overheads incurred by the convoluted OS page fault handler.

L1 Cache Misses. To better understand how Valinor achieves such low latency, Figure 7.2.3 depicts the L1D hit rate of the in-order core used in **Valinor**, across different benchmarks. Each benchmark was run for five different sizes of the allocation pool. It can be seen that the cache hit rate is consistently around 80%. We also measured the L1I hit rate, which was consistently 100% on repeated executions of the same benchmark with the same allocation pool size. These two facts are indicative of the benefits of a programmable hardware unit: On the one hand, the allocation libraries are small enough to fit in the unit's L1 Instruction cache. On the other hand, the choice of an alternative allocation scheme

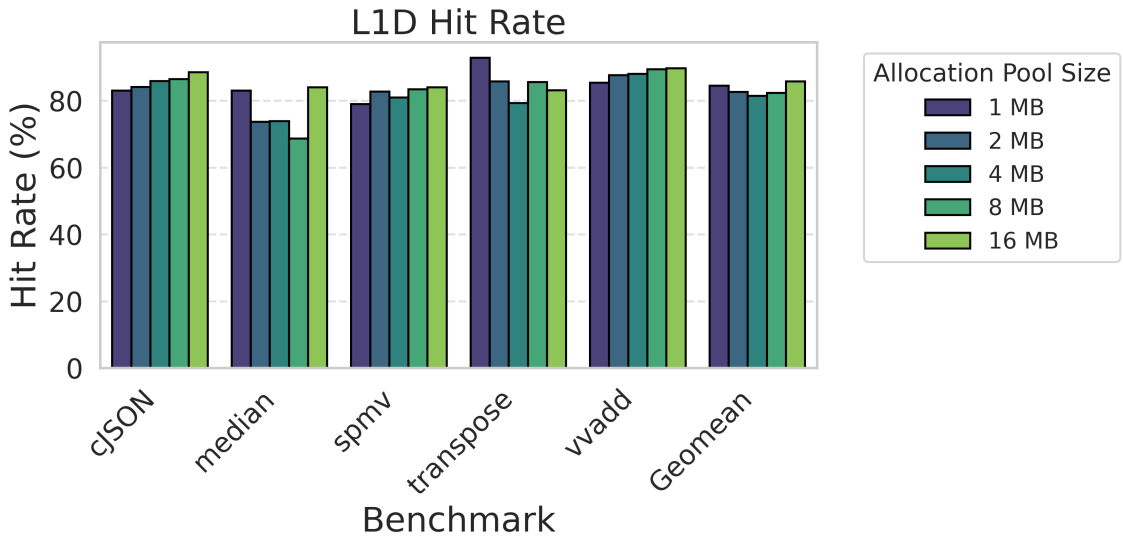


Figure 7.2.3: L1D hit rate in **Valinor**.

with lower metadata permits said metadata to fit in the unit's L1 Data cache.

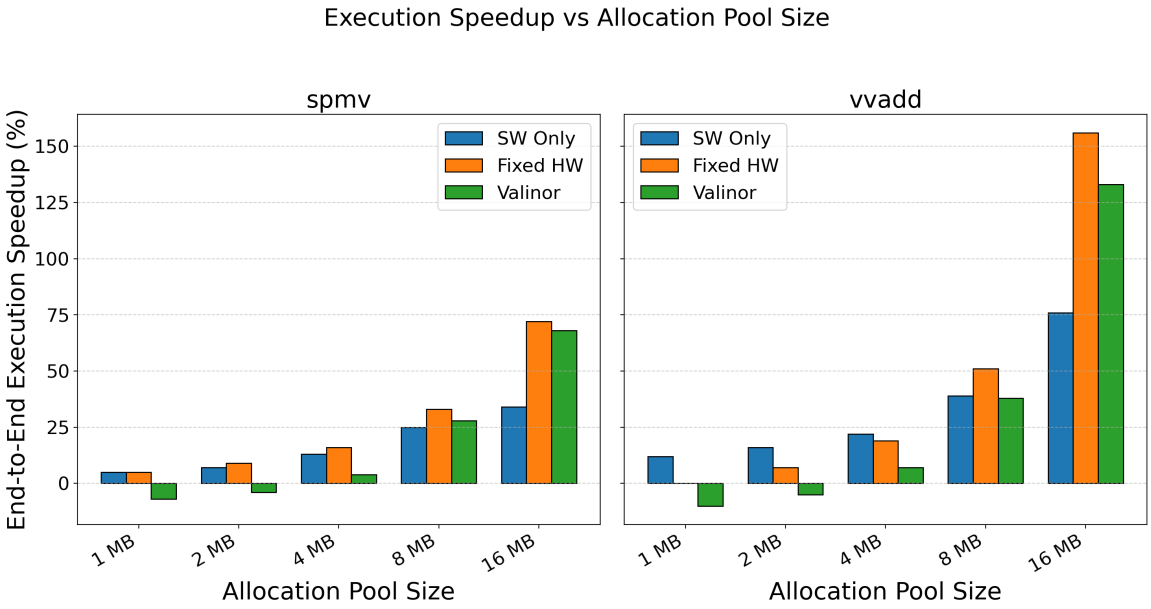


Figure 7.2.4: End-to-End execution speedup for different sizes of the allocation pool

Sensitivity Analysis on Allocation Pool size. To highlight the benefits of handling allocations in hardware, we ran two of the benchmarks (`spmv` and `vvadd`) for different sizes of the allocation pool. Note that, as in figure 7.2.1, the memory footprint of the chosen benchmarks is between $14MB$ and $16MB$, which means that it does not entirely fit in the allocation pool for all but the largest allocation pool size. Figure 7.2.4 shows the end-to-end speedup, whereas Figure 7.2.5 shows the total latency (in cycles) spent establishing virtual-to-physical mappings, whether it be in hardware or via Linux's page fault handler. First, it can be seen from Figure 7.2.4 that, while for small sizes of the allocation pool the effect on execution speed is negligible, as this size increases performance improves significantly.

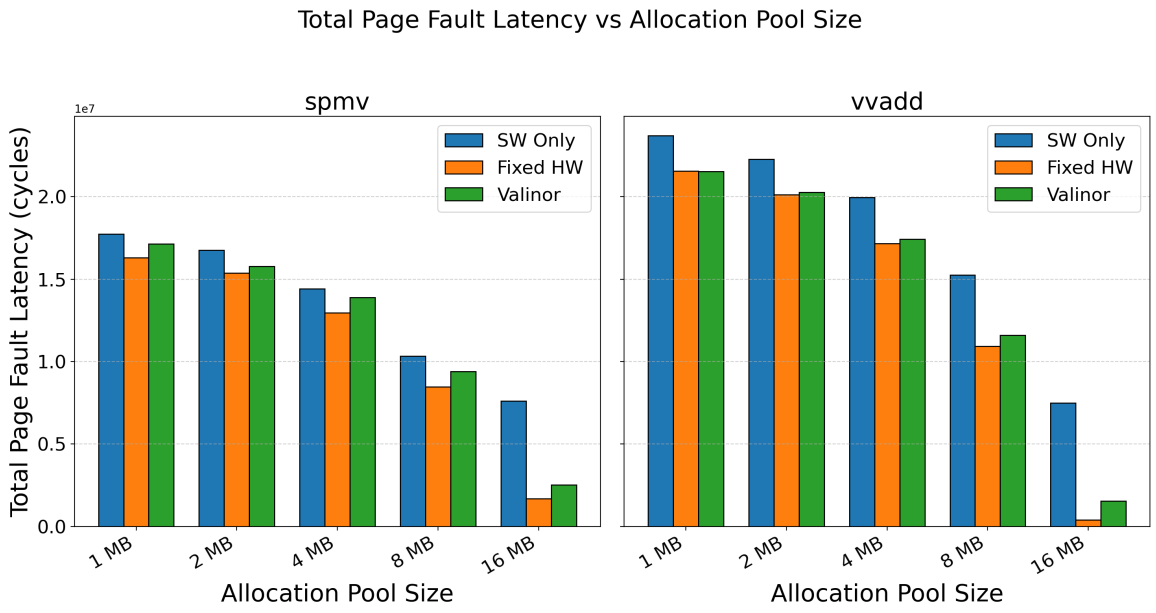


Figure 7.2.5: Total latency for establishing mappings for different sizes of the allocation pool

This is to be expected, as the size of the allocation pool essentially determines how many virtual-to-physical mappings are established in the "fast path". This can be further confirmed by Figure 7.2.5, which shows that, as as the size of the allocation pool increases, the total time spent on establishing virtual-to-physical mappings is significantly reduced. In fact, it can be seen that the total page fault latency is reduced by 10.5× (for `spmv`) and 58.9× `vvadd` compared to the baseline in the **Fixed-HW** version, and by 7.0× (for `spmv`) and 15.2× (for `vvadd`) in the **Valinor** version. Once again, we see that, while slightly slower than **Fixed-HW**, **Valinor** can still provide significant performance benefits.

Area measurements. Finally, to quantify the hardware cost of our proposed mechanism, we used the Yosys Open SYnthesis Suite [111] to measure the area overheads of the modifications for both **Fixed-HW** and **Valinor**. We assumed an 8-core system consisting of (Small) BOOM cores, with a single allocation engine in both cases. We synthesized the design using the nangate45 Open Cell library [112]. According to our measurements, **Valinor** takes up a modest 1.5% chip area, whereas **Fixed-HW** only adds a negligible 0.12%. While the programmable allocation engine takes up more area than the fixed hardware circuit, given our design decision to only have one allocation unit for the entire system and not one per-core, the hardware cost is kept fairly low.

Chapter 8

Conclusion and Future Work

This thesis addressed the increasing performance and energy challenges of memory allocation in modern computing systems. We showed that in short-lived, latency-sensitive workloads, such as serverless functions, microservices, and large language model (LLM) inference, the cost of software-only page allocation dominates execution time and energy consumption. Through real system experiments, we identified the main bottlenecks of conventional software-based allocation.

We introduced **Valinor**, a programmable hardware allocation engine that cooperates with the operating system to accelerate page fault handling while preserving fine-grained control over allocation policies. Valinor allows the operating system to offload memory allocation to hardware when appropriate, bypassing costly kernel traps, while retaining the ability to define and install custom policies. Unlike prior fixed-function accelerators, Valinor is *programmable* and can adapt dynamically to runtime conditions such as DRAM bandwidth utilization, cache occupancy, and NUMA locality. This makes it possible to combine the low latency of hardware-based allocation with the flexibility and semantic expressiveness of software allocators.

We implemented and evaluated Valinor in a cycle-accurate simulator and on an FPGA-based RISC-V prototype running Linux. Our experiments demonstrate that Valinor reduces total page fault latency by up to **7–15**×, improving end-to-end application performance by up to **77%** compared to baseline Linux. The results also show that Valinor achieves an **80% L1D cache hit rate**, close to the efficiency of a fixed-function hardware allocator, while maintaining software-level flexibility. These findings confirm that programmable hardware allocation is both feasible and highly beneficial for modern, latency-critical workloads.

In summary, this work demonstrates that a programmable, hardware-assisted approach to memory allocation can bridge the gap between flexibility and performance. By enabling the operating system to program allocation behavior directly into hardware, Valinor enables a new class of adaptive, high-performance allocation schemes.

Future Work

Several avenues exist for future research and development of Valinor:

- **Multi-core and NUMA Support:** Extending Valinor to coordinate allocation decisions across multiple cores and NUMA domains to improve scalability and fairness.
- **Telemetry-driven Adaptation:** Exploiting real-time hardware metrics (e.g., DRAM congestion, cache pressure) to guide adaptive placement and allocation strategies.
- **Disaggregated Memory Environments:** Integrating Valinor in distributed and disaggregated memory systems to enable efficient remote memory allocation and translation.

By bridging the boundary between OS-managed and hardware-managed memory, we hope that **Valinor** lays the foundation for the next generation of adaptive, efficient, and intelligent memory architectures.

Bibliography

[1] Ziqi Wang et al. "Memento: Architectural Support for Ephemeral Memory Management in Serverless Environments". In: *MICRO*. 2023.

[2] Gyusun Lee et al. "A Case for Hardware-Based Demand Paging". In: *ISCA*. 2020.

[3] Svilen Kanev et al. "Mallacc: Accelerating Memory Allocation". In: *ASPLOS*. 2017.

[4] H. Cam, M. Abd-El-Barr, and S.M. Sait. "A high-performance hardware-efficient memory allocation technique and design". In: *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.99CB37040)*. 1999, pp. 274–276. DOI: 10.1109/ICCD.1999.808436.

[5] J. Morris Chang and Edward F. Gehringer. "A High-Performance Memory Allocator for Object-Oriented Systems". In: *TC*. 1996.

[6] J. Morris Chang, Witawas Srisa-An, and C-TD Lo. "Architectural Support for Dynamic Memory Management". In: *ICCD*. 2000.

[7] Wentong Li, Saraju Mohanty, and Krishna Kavi. "A Page-Based Hybrid (Software-Hardware) Dynamic Memory Allocator". In: *CAL*. 2006.

[8] Wentong Li et al. "Feasibility of Decoupling Memory Management From the Execution Pipeline". In: *J. Syst. Archit.* 2007.

[9] Greg Wright, Matthew L. Seidl, and Mario Wolczko. *An object-aware memory architecture*. Tech. rep. USA, 2005.

[10] J.M. Chang and E.F. Gehringer. "Evaluation of an object-caching coprocessor design for object-oriented systems". In: *Proceedings of 1993 IEEE International Conference on Computer Design ICCD'93*. 1993, pp. 132–139. DOI: 10.1109/ICCD.1993.393393.

[11] José A Joao, Onur Mutlu, and Yale N Patt. "Flexible reference-counting-based hardware acceleration for garbage collection". In: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, pp. 418–428.

[12] Samuel Thomas et al. "Towards Hardware Accelerated Garbage Collection with Near-Memory Processing". In: *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 2022, pp. 1–6. DOI: 10.1109/HPEC55821.2022.9926323.

[13] William J. Schmidt and Kelvin D. Nilsen. "Performance of a hardware-assisted real-time garbage collector". In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VI. San Jose, California, USA: Association for Computing Machinery, 1994, pp. 76–85. ISBN: 0897916603. DOI: 10.1145/195473.195504. URL:

[14] David S. Wise et al. "Research Demonstration of a Hardware Reference-Counting Heap". In: *Lisp Symb. Comput.* 10.2 (July 1997), pp. 159–181. ISSN: 0892-4635. DOI: 10.1023/A:1007715101339. URL:

[15] M. Meyer. "An On-Chip Garbage Collection Coprocessor for Embedded Real-Time Systems". In: *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2005, pp. 517–524.

[16] Andres Amaya Garcia, David May, and Ed Nutting. "Integrated Hardware Garbage Collection". In: *ACM Trans. Embed. Comput. Syst.* 20.5 (July 2021). ISSN: 1539-9087. DOI: 10.1145/3450147. URL:

[17] Matthias Meyer. “A true hardware read barrier”. In: *Proceedings of the 5th International Symposium on Memory Management*. ISMM ’06. Ottawa, Ontario, Canada: Association for Computing Machinery, 2006, pp. 3–16. ISBN: 1595932216. DOI: 10.1145/1133956.1133959. URL:

[18] Jaeyoung Jang et al. “Charon: Specialized Near-Memory Processing Architecture for Clearing Dead Objects in Memory”. In: *MICRO*. 2019.

[19] Matthias Meyer. “A Novel Processor Architecture with Exact Tag-Free Pointers”. In: *IEEE Micro* 24.3 (May 2004), pp. 46–55. ISSN: 0272-1732. DOI: 10.1109/MM.2004.2. URL:

[20] Sylvain Stanchina and Matthias Meyer. “Mark-sweep or copying? a "best of both worlds" algorithm and a hardware-supported real-time implementation”. In: *Proceedings of the 6th International Symposium on Memory Management*. ISMM ’07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pp. 173–182. ISBN: 9781595938930. DOI: 10.1145/1296907.1296928. URL:

[21] Martin Maas, Krste Asanović, and John Kubiatowicz. “A Hardware Accelerator for Tracing Garbage Collection”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 138–151. DOI: 10.1109/ISCA.2018.00022.

[22] W. Srisa-an, C.-T.D. Lo, and J.-M. Chang. “Active memory processor: a hardware garbage collector for real-time Java embedded devices”. In: *IEEE Transactions on Mobile Computing* 2.2 (2003), pp. 89–101. DOI: 10.1109/TMC.2003.1217230.

[23] Chandrahas Tirumalasetty et al. “Reducing Minor Page Fault Overheads through Enhanced Page Walker”. In: *ACM Trans. Archit. Code Optim.* 19.4 (2022), 57:1–57:26. DOI: 10.1145/3547142. URL:

[24] Zhiyuan Guo et al. “Clio: A Hardware-software co-designed Disaggregated Memory system”. In: *ASPLOS*. 2022.

[25] Nathan Pemberton, John D. Kubiatowicz, and Randy H. Katz. *Enabling Efficient and Transparent Remote Memory Access in Disaggregated Datacenters*.

[26] Chanchal Kumar et al. “Post-Fabrication Microarchitecture”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 1270–1281. ISBN: 9781450385572. DOI: 10.1145/3466752.3480119. URL:

[27] Rahul Razdan and Michael D. Smith. “A high-performance microarchitecture with hardware-programmable functional units”. In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: Association for Computing Machinery, 1994, pp. 172–180. ISBN: 0897917073. DOI: 10.1145/192724.192749. URL:

[28] Wittig and Chow. “OneChip: an FPGA processor with reconfigurable logic”. In: *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*. 1996, pp. 126–135. DOI: 10.1109/FPGA.1996.564773.

[29] Zhi Alex Ye et al. “CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit”. In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ISCA ’00. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2000, pp. 225–235. ISBN: 1581132328. DOI: 10.1145/339647.339687. URL:

[30] J. R. Hauser and J. Wawrzynek. “Garp: a MIPS processor with a reconfigurable coprocessor”. In: *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*. FCCM ’97. USA: IEEE Computer Society, 1997, p. 12. ISBN: 0818681594.

[31] Seth Copen Goldstein et al. “PipeRench: a co/processor for streaming multimedia acceleration”. In: *Proceedings of the 26th Annual International Symposium on Computer Architecture*. ISCA ’99. Atlanta, Georgia, USA: IEEE Computer Society, 1999, pp. 28–39. ISBN: 0769501702. DOI: 10.1145/300979.300982. URL:

[32] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. “Dynamically Specialized Datapaths for energy efficient computing”. In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 2011, pp. 503–514. DOI: 10.1109/HPCA.2011.5749755.

[33] A. DeHon. “DPGA-coupled microprocessors: commodity ICs for the early 21st Century”. In: *Proceedings of IEEE Workshop on FPGA’s for Custom Computing Machines*. 1994, pp. 31–39. DOI: 10.1109/FPGA.1994.315596.

[34] P.M. Athanas and H.F. Silverman. "Processor reconfiguration through instruction-set metamorphosis". In: *Computer* 26.3 (1993), pp. 11–18. DOI: 10.1109/2.204677.

[35] Shantanu Gupta et al. "Bundled execution of recurring traces for energy-efficient general purpose processing". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. Porto Alegre, Brazil: Association for Computing Machinery, 2011, pp. 12–23. ISBN: 9781450310536. DOI: 10.1145/2155620.2155623. URL:

[36] Hue-Sung Kim, Arun K. Somani, and Akhilesh Tyagi. "A reconfigurable multi-function computing cache architecture". In: *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*. FPGA '00. Monterey, California, USA: Association for Computing Machinery, 2000, pp. 85–94. ISBN: 1581131933. DOI: 10.1145/329166.329185. URL:

[37] Matthew A. Watkins and David H. Albonesi. "ReMAP: A Reconfigurable Architecture for Chip Multiprocessors". In: *IEEE Micro* 31.1 (2011), pp. 65–77. DOI: 10.1109/MM.2011.14.

[38] Matthew A. Watkins, Mark J. Cianchetti, and David H. Albonesi. "Shared reconfigurable architectures for CMPS". In: *2008 International Conference on Field Programmable Logic and Applications*. 2008, pp. 299–304. DOI: 10.1109/FPL.2008.4629948.

[39] J. Kuskin et al. "The Stanford FLASH multiprocessor". In: *Proceedings of the 21st Annual International Symposium on Computer Architecture*. ISCA '94. Chicago, Illinois, USA: IEEE Computer Society Press, 1994, pp. 302–313. ISBN: 0818655100. DOI: 10.1145/191995.192056. URL:

[40] S. K. Reinhardt, J. R. Larus, and D. A. Wood. "Tempest and Typhoon: User-level Shared Memory". In: *Proceedings of the 21st Annual International Symposium on Computer Architecture*. ISCA '94. 1994.

[41] Anant Agarwal et al. "The MIT Alewife machine: architecture and performance". In: *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. ISCA '98. Barcelona, Spain: Association for Computing Machinery, 1998, pp. 509–520. ISBN: 1581130589. DOI: 10.1145/285930.286009. URL:

[42] Amin Firoozshahian et al. "A memory system design framework: creating smart memories". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. Austin, TX, USA: Association for Computing Machinery, 2009, pp. 406–417. ISBN: 9781605585260. DOI: 10.1145/1555754.1555805. URL:

[43] Brian C. Schwedock et al. "täkō: a polymorphic cache hierarchy for general-purpose optimization of data movement". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 42–58. ISBN: 9781450386104. DOI: 10.1145/3470496.3527379. URL:

[44] Sam Ainsworth and Timothy M. Jones. "The Guardian Council: Parallel Programmable Hardware Security". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 1277–1293. ISBN: 9781450371025. DOI: 10.1145/3373376.3378463. URL:

[45] Daniel Y. Deng et al. "Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric". In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '43. USA: IEEE Computer Society, 2010, pp. 137–148. ISBN: 9780769542997. DOI: 10.1109/MICRO.2010.17. URL:

[46] Yifan Yang, Joel S. Emer, and Daniel Sanchez. "SpZip: architectural support for effective data compression in irregular applications". In: *Proceedings of the 48th Annual International Symposium on Computer Architecture*. ISCA '21. Virtual Event, Spain: IEEE Press, 2021, pp. 1069–1082. ISBN: 9781450390866. DOI: 10.1109/ISCA52012.2021.00087. URL:

[47] Sam Ainsworth and Timothy M Jones. "An event-triggered programmable prefetcher for irregular workloads". In: 2018.

[48] Binh Pham et al. "Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?" In: *MICRO*. 2015.

[49] Chang Hyun Park et al. "Perforated Page: Supporting Fragmented Memory Allocation for Large Pages". In: *ISCA*. 2020.

[50] Faruk Guvenilir and Yale N Patt. "Tailored Page Sizes". In: *ISCA*. 2020.
[51] Youngjin Kwon et al. "Coordinated and Efficient Huge Page Management with Ingens". In: *OSDI*. 2016.
[52] Madhusudhan Talluri et al. "Tradeoffs in Supporting Two Page Sizes". In: *ISCA*. 1992.
[53] Ashish Panwar, Aravinda Prasad, and K Gopinath. "Making Huge Pages Actually Useful". In: *ASPLOS*. 2018.
[54] Ashish Panwar, Sorav Bansal, and K Gopinath. "Hawkeye: Efficient Fine-grained OS Support for Huge Pages". In: *ASPLOS*. 2019.
[55] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. "Trident: Harnessing Architectural Resources for All Page Sizes in X86 Processors". In: *MICRO*. 2021.
[56] Rachata Ausavarungnirun et al. "Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes". In: *MICRO*. 2017.
[57] Zhen Fang et al. "Reevaluating Online Superpage Promotion with Hardware Support". In: *HPCA*. 2001.
[58] Mark Swanson, Leigh Stoller, and John Carter. "Increasing TLB Reach Using Superpages Backed By Shadow Memory". In: *ISCA*. 1998.
[59] Yu Du et al. "Supporting Superpages in Non-Contiguous Physical Memory". In: *HPCA*. 2015.
[60] Madhusudhan Talluri and Mark D. Hill. "Surpassing the TLB Performance of Superpages with Less Operating System Support". In: *ASPLOS*. 1994.
[61] Mel Gorman and Patrick Healy. "Supporting Superpage Allocation Without Additional Hardware Support". In: *ISMM*. 2008.
[62] Mohammad Agbarya et al. "Predicting Execution Times with Partial Simulations in Virtual Memory Research: Why and How". In: *MICRO*. 2020.
[63] Narayanan Ganapathy and Curt Schimmel. "General Purpose Operating System Support for Multiple Page Sizes". In: *ATC*. 1998.
[64] Arkaprava Basu et al. "Efficient Virtual Memory for Big Memory Servers". In: *ISCA*. 2013.
[65] Kaiyang Zhao et al. "Contiguitas: the Pursuit of Physical Memory Contiguity in Datacenters". In: *ISCA*. 2023.
[66] Chloe Alverti et al. "Enhancing and Exploiting Contiguity for Fast Memory Virtualization". In: *ISCA*. 2020.
[67] Zi Yan et al. "Translation Ranger: Operating System Support for Contiguity-Aware TLBs". In: *ISCA*. 2019.
[68] V. Karakostas et al. "Redundant Memory Mappings for Fast Access to Large Memories". In: *ISCA*. 2015.
[69] Chang Hyun Park et al. "Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations". In: *ISCA*. 2017.
[70] Dongwei Chen et al. "FlexPointer: Fast Address TranslatiOn Based On Range TLB and Tagged Pointers". In: *TACO*. 2023.
[71] Binh Pham et al. "CoLT: Coalesced Large-Reach TLBs". In: *MICRO*. 2012.
[72] Jiyuan Zhang et al. "Direct Memory Translation for Virtualized Clouds". In: ASPLOS. 2024.
[73] Konstantinos Kanellopoulos et al. "Utopia: Efficient Address Translation using Hybrid Virtual-to-Physical Address Mapping". In: *MICRO*. 2023. arXiv: 2211.12205 [cs.AR].
[74] Javier Picorel, Djordje Jevdjic, and Babak Falsafi. "Near-Memory Address Translation". In: *PACT*. 2017.
[75] Krishnan Gosakan et al. "Mosaic Pages: Big TLB Reach with Small Pages". In: *ASPLOS*. 2023.
[76] Swapnil Haria, Michael M. Swift, and Mark D. Hill. "Devirtualizing Virtual Memory for Heterogeneous Systems". In: *ASPLOS*. 2018.
[77] Idan Yaniv and Dan Tsafrir. "Hash, Don't Cache (the Page Table)". In: *SIGMETRICS*. 2016.
[78] Dimitrios Skarlatos et al. "Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism". In: *ASPLOS*. 2020.
[79] Sam Ainsworth and Timothy M. Jones. "Compendia: Reducing Virtual-Memory Costs Via Selective Densification". In: *ISMM*. 2021.
[80] Chang Hyun Park et al. "Every Walk's a Hit: Making Page Walks Single-Access Cache Hits". In: *ASPLOS*. 2022.

[81] Osang Kwon et al. “Distributed Page Table: Harnessing Physical Memory as an Unbounded Hashed Page Table”. In: *MICRO*. 2024.
[82] Jovan Stojkovic et al. “Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables”. In: *ASPLOS*. 2022.
[83] Jovan Stojkovic et al. “Memory-Efficient Hashed Page Tables”. In: *HPCA*. 2023.
[84] Reto Achermann et al. “Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines”. In: *ASPLOS*. 2020.
[85] Dimitris Fotakis et al. “Space Efficient Hash Tables with Worst Case Constant Access Time”. In: *STACS*. 2003.
[86] Hanna Alam et al. “Do-It-Yourself Virtual Memory Translation”. In: *ISCA*. 2017.
[87] Sanjay Ghemawat and Paul Menage. *TCMalloc: Thread-Caching Malloc*.
[88] Xilinx ZCU 106 FPGA: https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/zcu106.html.
[89] RISCV-MINI: https://github.com/ucb-bar/riscv-mini.
[90] BOOM Core: https://boom-core.org/.
[91] Chandrahas Tirumalasetty et al. “Reducing Minor Page Fault Overheads through Enhanced Page Walker”. In: *TACO*. 2022.
[92] Mahdi Nazm Bojnordi and Engin Ipek. “PARDIS: a programmable memory controller for the DDRx interfacing standards”. In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA ’12. Portland, Oregon: IEEE Computer Society, 2012, pp. 13–24. ISBN: 9781450316422.
[93] J. Carter et al. “Impulse: Building a Smarter Memory Controller”. In: *Proceedings of the 5th International Symposium on High Performance Computer Architecture*. HPCA ’99. USA: IEEE Computer Society, 1999, p. 70. ISBN: 0769500048.
[94] Jerome Martin et al. “A Microprogrammable Memory Controller for high-performance dataflow applications”. In: *2009 Proceedings of ESSCIRC*. 2009, pp. 348–351. DOI: 10.1109/ESSCIRC.2009.5325981.
[95] G. Kornaros et al. “A fully-programmable memory management system optimizing queue handling at multi gigabit rates”. In: *Proceedings of the 40th Annual Design Automation Conference*. DAC ’03. Anaheim, CA, USA: Association for Computing Machinery, 2003, pp. 54–59. ISBN: 1581136889. DOI: 10.1145/775832.775849. URL:
[96] Kun-Bin Lee, Tzu-Chieh Lin, and Chein-Wei Jen. “An efficient quality-aware memory controller for multimedia platform SoC”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 15.5 (2005), pp. 620–633. DOI: 10.1109/TCSVT.2005.846412.
[97] Nishil Talati et al. “Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design”. In: *HPCA 2021*.
[98] Yifan Yuan et al. “QEI: Query Acceleration Can be Generic and Efficient in the Cloud”. In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2021, pp. 385–398. DOI: 10.1109/HPCA51647.2021.00040.
[99] https://witscad.com/course/computer-architecture/chapter/virtual-memory.
[100] RISC-V architecture: https://riscv.org/specifications/ratified/.
[101] RISC-V ISA unprivileged specifications: https://docs.riscv.org/reference/isa/$_a$*ttachments/riscv-unprivileged.pdf*.
[102] Instruction Sets should be free: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf.
[103] RISC-V ISA privileged specifications: https://docs.riscv.org/reference/isa/$_a$*ttachments/riscv-privileged.pdf*.
[104] Chipyard: https://github.com/ucb-bar/chipyard.
[105] Rocket Core: https://github.com/chipsalliance/rocket-chip.
[106] CVA6 Core: https://github.com/openhwgroup/cva6/.
[107] Verilator: https://www.veripool.org/verilator/.
[108] Sagar Karandikar et al. “FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud”. In: *ISCA*. 2018.
[109] Hammer: https://github.com/ucb-bar/hammer.

[110] Chisel Hardware Description Language: https://www.chisel-lang.org/.
[111] *Yosys Open SYnthesis Suite.*
[112] *NanGate FreePDK45 open cell library.*