Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

τομεας τεχνολογιας πληροφορικης και υπολογιστων

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Spike-based Dynamic Data Placement over Hybrid DRAM/NVM Memory Systems

Μελέτη και υλοποίηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αριστοτέλη Γρίβα

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2025

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

Spike-based Dynamic Data Placement over Hybrid DRAM/NVM Memory Systems

Μελέτη και υλοποίηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αριστοτέλη Γρίβα

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Ιανουαρίου, 2025.

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

Αθήνα, Ιανουάριος 2025

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

στην οικογένεια μου

Περίληψη

Τα σύγχρονα υπολογιστικά συστήματα απαιτούν ολοένα και περισσότερες λύσεις μνήμης υψηλής χωρητικότητας, οι οποίες προσφέρουν τόσο υψηλή απόδοση όσο και βιωσιμότητα. Για την κάλυψη αυτών των αναγκών, αρχιτεκτονικές ετερογενούς μνήμης—που ενσωματώνουν την παραδοσιακή DRAM με τεχνολογίες Μη Πτητικής Μνήμης (NVM)—αναδύονται ως αποτελεσματικές εναλλακτικές λύσεις. Ωστόσο, αυτά τα συστήματα εισάγουν σύνθετες προκλήσεις στη διαχείριση της δυναμικής τοποθέτησης δεδομένων μεταξύ DRAM και NVM. Σε αυτή τη μελέτη, παρουσιάζουμε το `SPID`, έναν αλγόριθμο τοποθέτησης δεδομένων χαμηλού κόστους, βασισμένο σε αιχμές (spikes) για ετερογενή συστήματα DRAM/NVM. Το `SPID` αξιοποιεί προηγμένους μηχανισμούς ανίχνευσης spikes στο εύρος ζώνης και στα ενεργά αντικείμενα μνήμης, καθώς και προσαρμοστική λήψη αποφάσεων, για την αποτελεσματική βελτιστοποίηση της χρήσης μνήμης. Αξιολογούμε το `SPID` σε σύγκριση με ένα σύνολο βασικών πολιτικών τοποθέτησης και προηγμένων λύσεων, χρησιμοποιώντας πραγματικές εφαρμογές, και δείχνουμε ότι επιτυγχάνουμε 30.82% υψηλότερη απόδοση και 31.61% χαμηλότερη κατανάλωση ενέργειας κατά μέσο όρο. Το `SPID` ενσωματώνεται στη βιβλιοθήκη ανοιχτού κώδικα `SPMalloc`, η οποία αναχαιτίζει τις κλήσεις δυναμικής δέσμευσης μνήμης κατά την εκτέλεση, επιτρέποντας ακριβή παρακολούθηση των μοτίβων δέσμευσης μνήμης με ελάχιστο φόρτο. Μαζί, το `SPID` και η `SPMalloc` βελτιστοποιούν την τοποθέτηση δεδομένων, ενισχύοντας την απόδοση σε περιβάλλοντα ετερογενούς μνήμης.

Όροι Ευρετηρίου — Data Placement, DRAM, NVM, Spike-based, library, heterogeneous, allocation

Abstract

Modern computing systems require more and more high-capacity memory solutions that deliver both high performance and sustainability. To address these needs, heterogeneous memory architectures—integrating traditional DRAM with Non-Volatile Memory (NVM) technologies—are emerging as effective alternatives. However, these systems introduce complex challenges in managing dynamic data placement between DRAM and NVM. In this work, we introduce `SPID`, a lightweight, spike-based data placement algorithm for heterogeneous DRAM/NVM systems. `SPID` leverages advanced bandwidth and active object spike detection mechanisms and adaptive decision-making to optimize memory utilization efficiently. We evaluate `SPID` against a set of baseline placement policies and state-of-the art solutions over real-life applications, showing that we achieve 30.82% higher performance and 31.61% less energy consumption on average. `SPID` is integrated in `SPMalloc`, an open-source library that intercepts dynamic memory allocation calls at runtime, enabling precise monitoring of allocation patterns with minimal overhead. Together, `SPID` and `SPMalloc` optimize data placement, enhancing performance in heterogeneous memory environments.

Index Terms — Data Placement, DRAM, NVM, Spike-based, library, heterogeneous, allocation

Ευχαριστίες

Θα ήθελα να εκφράσω τις ειλικρινείς μου ευχαριστίες στον επιβλέποντα καθηγητή μου, κ. Δημήτριο Σούντρη, για την πολύτιμη καθοδήγησή του καθ' όλη τη διάρκεια αυτής της διπλωματικής εργασίας. Ευχαριστώ θερμά επίσης τον διδάκτορα κ. Μανώλη Κατσαραγάκη, για την άρτια συνεργασία μας, την συνεχή στήριξη και τις εύστοχες και εποικοδομητικές παρατηρήσεις του. Θα ήθελα να ευχαριστήσω ολόψυχα την οικογένειά μου και ιδιαίτερα τους γονείς μου, για την ενθάρρυνση και την ακατάπαυστη υποστήριξή τους, κατά τη διάρκεια αυτής της διαδικασίας και καθ' όλη τη διάρκεια των σπουδών μου. Ευχαριστώ τέλος τους φίλους μου για τη συνεχή συμπαράσταση και τη στήριξή τους, καθώς και για την ενθάρρυνση και τη θετική τους στάση, που συνέβαλαν στο να παραμείνω προσηλωμένος στους στόχους μου.

Αριστοτέλης Γρίβας
Ιανουάριος 2025

Contents

Figure List

Table List

Chapter 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

Τα τελευταία χρόνια, η ανάπτυξη των νέων τεχνολογιών μνήμης που ενσωματώνονται σε σύγχρονα συστήματα Υπολογισμού Υψηλής Απόδοσης (High Performance Computing, HPC) και Cloud έχει εισαγάγει ένα νέο αρχιτεκτονικό παράδειγμα, γνωστό ως ετερογενές σύστημα μνήμης. Για την αντιμετώπιση των περιορισμών που αφορούν στην επεκτασιμότητα και βιωσιμότητα των παραδοσιακών τεχνολογιών DRAM [1], [2], οι Μη Πτητικές Μνήμες (Non-Volatile Memories, NVM), γνωστές και ως Μόνιμες Μνήμες (Persistent Memories, PM), γίνονται όλο και περισσότερο ελκηστικές. Τεχνολογίες όπως οι 3D-XPoint [3], Resistive RAM (ReRAM)[4], και Spin Transfer Torque Magnetic RAM (STT-MRAM)[5] αποτελούν χαρακτηριστικά παραδείγματα NVMs. Αυτές οι τεχνολογίες προσφέρουν διατήρηση των δεδομένων, μειωμένη κατανάλωση ενέργειας, και υψηλότερη πυκνότητα σε σχέση με την DRAM, επιτυγχάνοντας αυξημένες χωρητικότητες μνήμης με χαμηλότερο κόστος [6]. Ωστόσο, οι NVMs παρουσιάζουν επίσης προκλήσεις, όπως υψηλότερο χρόνο πρόσβασης, χαμηλότερο εύρος ζώνης, και περιορισμένη αντοχή στις εγγραφές. Εμφανίζουν επίσης ασυμμετρικές καθυστερήσεις και κατανάλωση ενέργειας μεταξύ λειτουργιών ανάγνωσης και εγγραφής [7], οδηγώντας σε συμβιβασμούς κατά την ενσωμάτωσή τους με την παραδοσιακή DRAM.

Σύγχρονες εμπορικές πλατφόρμες και προμηθευτές, όπως η Google, η Amazon, και η Alibaba [8], [9], [10], συνδυάζουν παραδοσιακά DIMMs DRAM με Intel Optane DC Persistent Memory, που μέχρι σήμερα αποτελεί τη μοναδική εμπορικά διαθέσιμη τεχνολογία NVM. Αυτά τα συστήματα αποσκοπούν στη βιώσιμη και οικονομικά αποδοτική επέκταση της χωρητικότητας μνήμης, παρέχοντας στους χρήστες μια ποικιλία τεχνολογιών μνήμης. Μια σειρά εφαρμογών υποστηρίζεται σε αυτές τις πλατφόρμες, όπως μηχανική μάθηση, βαθιά μάθηση [11], φόρτοι εργασίας βάσεων δεδομένων [12], και επιστημονικές/μεγάλων δεδομένων (big data) εφαρμογές [13]. Αυτοί οι φόρτοι εργασίας απαιτούν υψηλή απόδοση επεξεργασίας με ελάχιστα σημεία συμφόρησης εισόδου/εξόδου (I/O bottlenecks), ασκώντας σημαντική πίεση στο υποσύστημα κύριας μνήμης.

Με τη ραγδαία εξέλιξη των ετερογενών συστημάτων μνήμης και τις ολοένα και πιο ποικίλες απαιτήσεις εφαρμογών, τόσο οι ακαδημαϊκές όσο και οι βιομηχανικές κοινότητες καταβάλλουν προσπάθειες για την ανάπτυξη αποδοτικών αλγορίθμων τοποθέτησης δεδομένων [14], [15], [16]. Πρόσφατες εξελίξεις στις πολιτικές τοποθέτησης υποστηρίζουν την τοποθέτηση δεδομένων σε πολλαπλά επίπεδα ανάλυσης, όπως επίπεδο σελίδας (page-level) [16], [17], [18], [19], επίπεδο αντικειμένου (object-level) [14], [20], [21], [22], και προσαρμοστικό επίπεδο (adaptive-level) [23]. Ωστόσο, αυτές οι προσεγγίσεις απαιτούν εκτεταμένες τεχνικές προφίλ (profiling), συμπεριλαμβανομένης της ενορχήστρωσης(instrumentation) [24], [25] και λεπτομερούς ανάλυσης σε επίπεδο αντικειμένων [21]. Παρόλο που αυτές οι μέθοδοι είναι ακριβείς, συχνά είναι ενεργοβόρες και επιβαρύνουν σημαντικά τον χρόνο εκτέλεσης, περιορίζοντας τη δυνατότητα επεκτασιμότητάς τους σε πραγματικά, μεγάλης κλίμακας συστήματα μνήμης. Για παράδειγμα, έρευνα που παρουσιάστηκε στο [26] υποδεικνύει ότι το πλήρες profiling μνήμης μπορεί να επιφέρει έως και 53%

επιβάρυνση στην απόδοση, ενώ αυτή η επιβάρυνση αυξάνεται σημαντικά όσο αυξάνεται το μέγεθος και η πολυπλοκότητα του φόρτου εργασίας.

Για να ξεπεραστούν αυτοί οι περιορισμοί, οι νέες λύσεις τοποθέτησης δεδομένων πρέπει να βελτιστοποιούν τις διαδικασίες profiling και να ελαχιστοποιούν την επιβάρυνση κατά τον χρόνο εκτέλεσης, διατηρώντας παράλληλα την ακρίβεια της τοποθέτησης. Σε αυτό το πλαίσιο, παρουσιάζουμε το SPID, την πρώτη εργασία που προτείνει ένα ελαφρύ, δυναμικό πλαίσιο τοποθέτησης δεδομένων βασισμένο σε αιχμές (spike-based), σχεδιασμένο ειδικά για ετερογενή συστήματα DRAM/NVM, εξαλείφοντας την εξάρτηση της αποτελεσματικότητας της λύσης από λεπτομερές profiling και άλλες ενεργοβόρες διαδικασίες. Το SPID μειώνει την εξάρτηση από εξαντλητικό profiling σε επίπεδο αντικειμένων και αξιοποιεί την ανίχνευση spikes στις προσβάσεις μνήμης για τον εντοπισμό περιοχών μνήμης με υψηλό εύρος ζώνης (bandwidth spikes) και περιοχών ενεργών αντικειμένων (active object spikes), επιτρέποντας την προσαρμοστική τοποθέτηση δεδομένων σε πραγματικό χρόνο. Αυτή η στοχευμένη προσέγγιση επιτρέπει στο SPID να προσαρμόζεται δυναμικά, διατηρώντας χαμηλό φορτίο profiling, προσφέροντας έτσι μια πρακτική και κλιμακούμενη λύση για εφαρμογές σε πραγματικό χρόνο. **Οι καινοτόμες συνεισφορές αυτής της μελέτης είναι οι εξής:**

- **Παρουσιάζουμε το** SPID, έναν ελαφρύ, κλιμακούμενο αλγόριθμο τοποθέτησης δεδομένων βασισμένο σε spikes για ετερογενή συστήματα DRAM/NVM. Το SPID αξιοποιεί προηγμένους μηχανισμούς ανίχνευσης spikes, συσχετίζοντας spikes στο εύρος ζώνης εγγραφής (Write BW spikes) και σε ενεργά αντικείμενα (Active Object spikes), και παρέχει προσαρμοστική λήψη αποφάσεων για τη βελτιστοποίηση της τοποθέτησης δεδομένων.
- **Αναπτύσσουμε και ενσωματώνουμε την** SPMalloc, μια βιβλιοθήκη ανοιχτού κώδικα που αναχαιτίζει τις κλήσεις δυναμικής δέσμευσης μνήμης, ενσωματώνει τη λειτουργικότητα του SPID και επιτρέπει την απρόσκοπτη τοποθέτηση δεδομένων σε ετερογενή συστήματα μνήμης.
- **Διενεργούμε μια εκτενή πειραματική αξιολόγηση του πλαισίου μας**, συγκρίνοντάς το με βασικούς αλγορίθμους και προηγμένες στρατηγικές τοποθέτησης δεδομένων. Τα αποτελέσματα δείχνουν ότι το SPID επιτυγχάνει μείωση του χρόνου εκτέλεσης κατά 30.82% και μειώνει την κατανάλωση ενέργειας κατά 31.61% κατά μέσο όρο.

Το υπόλοιπο κεφάλαιο οργανώνεται ως εξής. Η Ενότητα 1.2 παρουσιάζει την σχετική βιβλιογραφία και περιγράφει τις βασικές εξελίξεις της έρευνας. Η Ενότητα 1.3 περιγράφει τη λύση τοποθέτησης δεδομένων που προτείνουμε, ενώ η Ενότητα 1.4 αναλύει την πειραματική αξιολόγηση. Τέλος, η Ενότητα 1.5 ολοκληρώνει την εργασία.

1.2 Σχετική Βιβλιογραφία

Τα τελευταία χρόνια, έχουν πραγματοποιηθεί πολλαπλές μελέτες που εστιάζουν στην τοποθέτηση δεδομένων σε ετερογενή συστήματα κύριας μνήμης. Κατηγοριοποιούμε τη σχετική βιβλιογραφία με βάση το επίπεδο ανάλυσης στο οποίο εφαρμόζεται η τοποθέτηση, δηλαδή την τοποθέτηση δεδομένων σε επίπεδο δομής/αντικειμένου και σε επίπεδο σελίδας.

Τοποθέτηση Δεδομένων σε Επίπεδο Δομής/Αντικειμένου: Οι συγγραφείς του [14] προτείνουν διαστρωμάτωση και μετανάστευση αντικειμένων σε επίπεδο πυρήνα για ετερογενή συστήματα μνήμης. Στο [20], παρουσιάζεται ένας profiler που παρέχει κατευθύνσεις για την τοποθέτηση δεδομένων σε ετερογενή μνήμη, ενώ στο [21], προτείνεται δυναμική βελτιστοποίηση των τύπων δεδομένων για καλύτερη τοποθέτηση. Στο ίδιο επίπεδο, στο [22], παρουσιάζεται μια μέθοδος τοποθέτησης δομών δεδομένων που λαμβάνει υπόψη τις εγγραφές για cached και uncached NVMs. Οι συγγραφείς του [23] προτείνουν ένα πλαίσιο χρόνου εκτέλεσης για την τοποθέτηση δεδομένων προσαρμοσμένου επιπέδου ανάλυσης για εφαρμογές βασισμένες σε γραφήματα, ενώ στο [27], οι συγγραφείς εφαρμόζουν τοποθέτηση δεδομένων λαμβάνοντας υπόψη τις φάσεις της εφαρμογής.

Τοποθέτηση Δεδομένων σε Επίπεδο Σελίδας: Οι λύσεις τοποθέτησης δεδομένων σε επίπεδο σελίδας έχουν εξερευνηθεί ευρέως για τη βελτιστοποίηση της χρήσης και της απόδοσης μνήμης σε ετερογενή συστήματα. Οι συγγραφείς του [17] προτείνουν έναν αλγόριθμο τοποθέτησης σελίδων βασισμένο σε

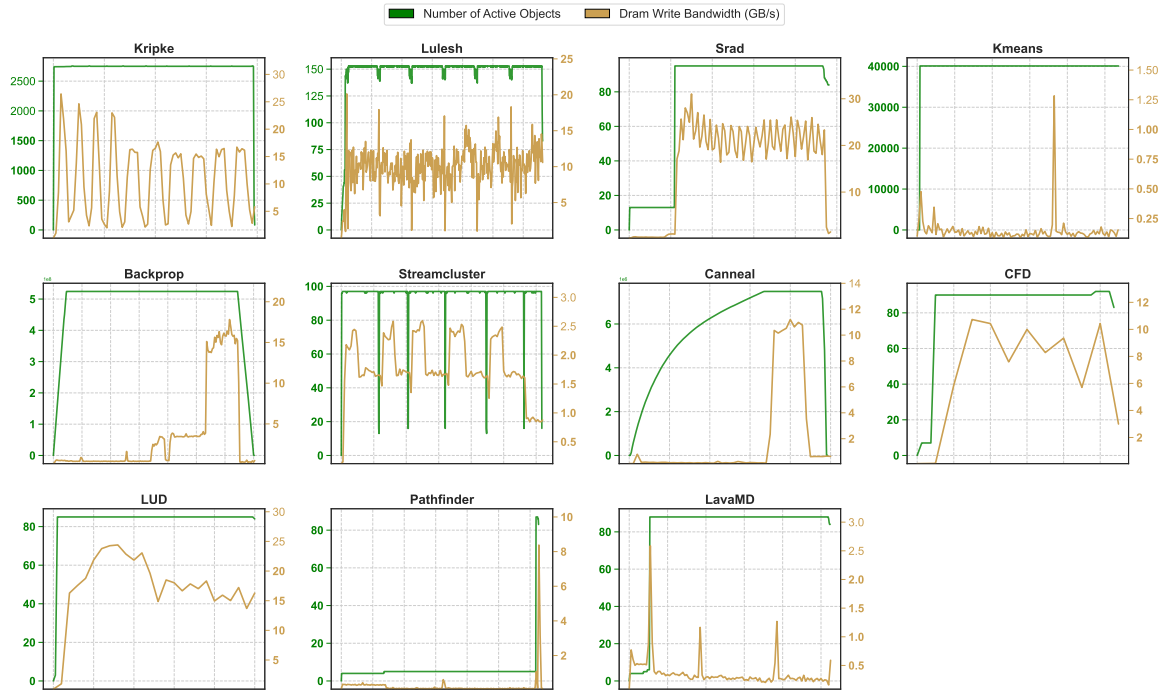

Figure 1.3.1: Συσχέτιση του αριθμού των ενεργών αντικειμένων που έχουν δεσμευτεί στη μνήμη (πράσινο) και του εύρους ζώνης εγγραφής μνήμης (κίτρινο), συναρτήσει του χρόνου, για όλα τα benchmarks που εξετάστηκαν.

LSTM (Long short-term memory) που προβλέπει μελλοντικά μοτίβα πρόσβασης ώστε να τοποθετεί τις σελίδες αποτελεσματικά στα επίπεδα μνήμης. Στο [16], προτείνεται μια προσέγγιση ενισχυτικής μάθησης (reinforcement learning) για την τοποθέτηση σελίδων, ενώ στο [18], οι συγγραφείς παρουσιάζουν έναν online μηχανισμό τοποθέτησης δεδομένων με καθοδήγηση από profiling, ειδικά σχεδιασμένο για την τοποθέτηση δεδομένων σε επίπεδο σελίδας σε περιβάλλοντα HPC. Τέλος, στο [19], προτείνεται ένας μηχανισμός τοποθέτησης σελίδων σε επίπεδο λειτουργικού συστήματος, που λειτουργεί χωρίς να απαιτεί τροποποιήσεις στις εφαρμογές.

Παρόλο που προηγούμενες έρευνες έχουν εξερευνήσει στρατηγικές τοποθέτησης δεδομένων για ετερογενή συστήματα μνήμης σε διάφορα επίπεδα ανάλυσης, αυτές οι προσεγγίσεις συνήθως παρουσιάζουν δύο κύρια μειονεκτήματα:

1. Βασίζονται σε δαπανηρούς και ενεργοβόρους μηχανισμούς profiling, και

2. Εισάγουν σημαντική επιβάρυνση χρόνου εκτέλεσης λόγω εκτεταμένης παρακολούθησης (monitoring) και απαίτησης εξαγωγής πληροφορίας κατά τον χρόνο εκτέλεσης.

Εξ όσων γνωρίζουμε, αυτή είναι η πρώτη εργασία που προτείνει μια λύση χαμηλού κόστους για την τοποθέτηση δεδομένων σε ετερογενή συστήματα DRAM/NVM, εξαλείφοντας την εξάρτηση της αποτελεσματικότητας της λύσης από λεπτομερές profiling και άλλες ενεργοβόρες διαδικασίες. Επιπλέον, σε αντίθεση με προηγούμενες έρευνες, αυτή η εργασία εισάγει το πρώτο πλαίσιο δυναμικής τοποθέτησης δεδομένων βασισμένο σε spikes για ετερογενή συστήματα DRAM/NVM. Η προσέγγισή μας αξιοποιεί την ανίχνευση spikes στις προσβάσεις μνήμης για τον εντοπισμό περιοχών μνήμης με υψηλό εύρος ζώνης και ενεργών αντικειμένων, επιτρέποντας την προσαρμοστική τοποθέτηση δεδομένων κατά τον χρόνο εκτέλεσης, χωρίς να βασίζεται σε δαπανηρό profiling ή μόνιμο monitoring.

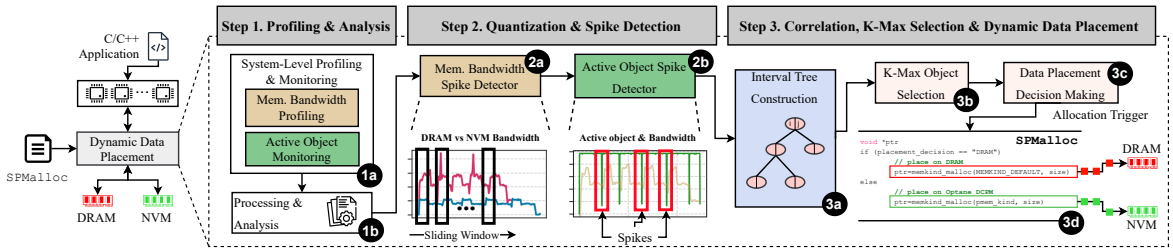

Figure 1.3.2: Επισκόπηση του πλαισίου SPID

1.3 Προτεινόμενη μεθοδολογία

Σε αυτή την Ενότητα, παρουσιάζουμε το SPID, μια λύση δυναμικής τοποθέτησης δεδομένων βασισμένης σε αιχμές (spike-based) για ετερογενή συστήματα DRAM/NVM. Οι θεμελιώδεις σχεδιαστικές αρχές της μεθοδολογίας μας καθοδηγούνται από την παρατήρηση της υψηλής χρονικής συσχέτισης μεταξύ της χρήσης εύρους ζώνης μνήμης και των δεσμεύσεων μνήμης για πολλές εφαρμογές. Στο Σχήμα 1.3.1 απεικονίζεται η συσχέτιση της κατανάλωσης εύρους ζώνης μνήμης (κίτρινο) και των ενεργών αντικειμένων δέσμευσης μνήμης (πράσινο) στο χρόνο για όλες τις εφαρμογές που εξετάστηκαν. Ο αριστερός άξονας Y δείχνει τον αριθμό των ενεργών αντικειμένων μνήμης, ενώ ο δεξιός άξονας Y δείχνει το εύρος ζώνης εγγραφής μνήμης κατά τη διάρκεια της εκτέλεσης της αντίστοιχης εφαρμογής.

Οι εξεταζόμενες εφαρμογές αποκαλύπτουν ορισμένα διαφορετικά μοτίβα συσχέτισης μεταξύ των ενεργών δεσμεύσεων και της χρήσης εύρους ζώνης μνήμης. Στην περίπτωση των *Lulesh*, *LavaMD* και *Pathfinder*, παρατηρούμε ότι οι περιοχές υψηλού εύρους ζώνης εγγραφής ευθυγραμμίζονται στενά με spikes στον αριθμό των ενεργών δεσμευμένων αντικειμένων, δείχνοντας άμεση συσχέτιση μεταξύ spikes δεσμεύσεων και χρήσης εύρους ζώνης. Στο *Streamcluster*, οι περιοχές υψηλού εύρους ζώνης εγγραφής μνήμης παρουσιάζουν μια μικρή χρονική μετατόπιση σε σχέση με τα spikes ενεργών αντικειμένων, όπου πολλαπλά spikes δεσμεύσεων οδηγούν σε αντίστοιχα spikes εύρους ζώνης μετά από μικρή καθυστέρηση. Αντίθετα, εφαρμογές όπως τα *Srad*, *Lud*, *Cfd* και *Kripke* εμφανίζουν ένα εκτεταμένο μοτίβο συσχέτισης, όπου ένα μοναδικό spike δεσμεύσεων ξεκινά μια παρατεταμένη περίοδο υψηλού εύρους ζώνης μνήμης. Τέλος, σε εφαρμογές όπως τα *Backprop* και *Kmeans*, παρατηρούμε πως ένα spike στην αρχή του προγράμματος ακολουθείτε τόσο από περιοχές χαμηλού όσο και υψηλού εύρους ζώνης μνήμης.

> ⋆ ***Κύριο Συμπέρασμα 1:*** *Οι περιοχές έντονων δυναμικών δεσμεύσεων μνήμης συσχετίζονται χρονικά με spikes στη χρήση εύρους ζώνης μνήμης.*

Η παραπάνω παρατήρηση καθοδηγεί τις βασικές επιλογές σχεδιασμού του SPID, το οποίο είναι η πρώτη εργασία που βασίζεται σε προηγμένους μηχανισμούς ανίχνευσης σε spikes, μέσω της συσχέτισης spikes εύρους ζώνης (BW-spikes) και ενεργών αντικειμένων(Active object-spikes), καθοδηγώντας την αποτελεσματική τοποθέτηση δεδομένων και τη διαχείριση πόρων σε ετερογενή συστήματα μνήμης. Το Σχήμα 1.3.2 παρουσιάζει μια επισκόπηση της προτεινόμενης μεθοδολογίας. Ως είσοδος, παρέχεται η επιθυμητή εφαρμογή σε C/C++. Η προτεινόμενη μεθοδολογία αποτελείται από τρεις διακριτές φάσεις, οι οποίες είναι οι εξής i) *Profiling & Analysis* (Ενότητα 1.3.1), ii) *Quantization & Spike Detection* (Ενότητα 1.3.2) και iii) *Correlation, K-Max Selection & Dynamic Data Placement* (Ενότητα 1.3.3), οι οποίες περιγράφονται λεπτομερώς στο υπόλοιπο της Ενότητας αυτής. Ο Πίνακας 1.1 συνοψίζει τις βασικές παραμέτρους του συστήματος που συζητούνται στη συνέχεια αυτής της Ενότητας.

1.3.1 Βήμα 1: Profiling & Analysis

Η αρχική φάση του SPID έχει σχεδιαστεί για να διεξάγει μια ολοκληρωμένη και μικρού φόρτου διαδικασία profiling της στοχευμένης εφαρμογής, διευκολύνοντας μια συστηματική ταξινόμηση των περιοχών spikes που περιγράφονται στην Ενότητα 1.3.2. Το profiling ξεκινά με το monitoring σε επίπεδο συστήματος/εφαρμογής και τη συλλογή δεδομένων (1a), όπου καταγράφονται βασικές μετρικές όπως το εύρος

ζώνης μνήμης και οι δεσμεύσεις μνήμης. Όπως φαίνεται στο Σχήμα 1.3.1, πραγματοποιούμε profiling εύρους ζώνης μνήμης εκτελώντας την εφαρμογή, ώστε να αξιολογήσουμε τη συσχέτιση μεταξύ της κατανάλωσης εύρους ζώνης μνήμης και των μοτίβων δεσμεύσεων μνήμης. Ειδικότερα, επικεντρωνόμαστε στο εύρος ζώνης εγγραφών, το οποίο χρησιμεύει ως βασικός δείκτης για τον εντοπισμό μεταβάσεων φάσης και την κατανόηση της δυναμικής συμπεριφοράς χρήσης μνήμης κατά την εκτέλεση.

Για να αντιμετωπιστεί το σημαντικό κόστος που συνδέεται συνήθως με την ανάλυση σε επίπεδο αντικειμένων, η μεθοδολογία μας αξιοποιεί μια αποδοτική στρατηγική που βασίζεται στην παρακολούθηση ενεργών αντικειμένων. Αυτό το βήμα περιλαμβάνει την παρακολούθηση της δυναμικής δέσμευσης bytes και του αριθμού των ενεργών αντικειμένων καθ' όλη τη διάρκεια της εκτέλεσης, επιτρέποντάς μας να καταγράψουμε τα γενικά πρότυπα χρήσης μνήμης χωρίς να απαιτείται λεπτομερής ανάλυση σε επίπεδο αντικειμένων. Με τη συνδυασμένη χρήση του εύρους ζώνης μνήμης και της παρακολούθησης ενεργών αντικειμένων, έχουμε αποτελεσματικό profiling της συμπεριφοράς μνήμης της εφαρμογής, το οποίο διευκολύνει τον εντοπισμό σημείων συμφόρησης της απόδοσης.

Τα δεδομένα που συλλέγονται επεξεργάζονται και αναλύονται (1b) για να εντοπιστούν κρίσιμες περιοχές και μεταβάσεις φάσης στα επόμενα βήματα. Τα επεξεργασμένα δεδομένα προωθούνται στο επόμενο βήμα, το οποίο παρουσιάζεται στην Ενότητα 1.3.2.

1.3.2 Βήμα 2: Quantization & Spike Detection

Σε αυτό το βήμα, ο κύριος στόχος είναι η ανίχνευση και ταξινόμηση διακριτών φάσεων της εφαρμογής σε δύο βασικές κατηγορίες: τις *Περιοχές με Spikes στη χρήση του Εύρους Ζώνης Μνήμης* (*Memory Bandwidth Spike Regions*), που χαρακτηρίζονται από υψηλή χρήση εύρους ζώνης μνήμης, και τις *Περιοχές με Spikes Ενεργών Αντικειμένων* (*Active Object Spike Regions*), που χαρακτηρίζονται από πολλαπλές δεσμεύσεις αντικειμένων μνήμης. Η ταξινόμηση αυτών των περιοχών βασίζεται σε δύο νέους μηχανισμούς ανίχνευσης: (i) τον *Memory Bandwidth Spike Detector* (2a) και (ii) τον *Active Object Spike Detector* (2b), οι οποίοι περιγράφονται αναλυτικά στη συνέχεια αυτής της υποενότητας.

Memory Bandwidth Spike Detector: Για την ανίχνευση διακριτών φάσεων της εφαρμογής, παρουσιάζουμε έναν αλγόριθμο ποσοτικοποίησης, σχεδιασμένο να διαιρεί την εφαρμογή σε διακριτά χρονικά διαστήματα με βάση τα μοτίβα εύρους ζώνης μνήμης. Ο αλγόριθμός μας χρησιμοποιεί έναν μηχανισμό ολισθαίνον παραθύρου (sliding window) για την ανίχνευση φάσεων υψηλού εύρους ζώνης. Στο Σχήμα 1.3.3 απεικονίζονται οι κυματομορφές του εύρους ζώνης εγγραφής για Dram (κόκκινο) και NVM (μπλε) για κάθε benchmark. Αυτές είναι οι κυματομορφές πάνω στις οποίες λειτουργεί το ολισθαίνον παράθυρο με στόχο την εύρεση περιοχών όπου η NVM υστερεί σε μεγάλο βαθμό σε σύγκριση με τη Dram όσον αφορά στο εύρος ζώνης, και συνεπώς στον χρόνο εκτέλεσης. Όπως θα αναμέναμε, οι κυματομορφές αυτές παρουσιάζουν πανομοιότυπα μοτίβα εύρους ζώνης, με την NVM να σημειώνει μικρότερες τιμές λόγω της περιορισμένης απόδοσής της στις εγγραφές, με αποτέλεσμα χρονικά μεγαλύτερες κυματο-

Denotation	Description
$BW_{\text{DRAM}}(t)$	Εύρος ζώνης εγγραφής της DRAM τη χρονική στιγμή t
$\overline{BW}_{\text{NVM}}(s)$	Μέσος όρος εύρους ζώνης εγγραφής της NVM πάνω στο ολισθαίνον παράθυρο s
t_i	Χρονική στιγμή έναρξης ενός spike εύρους ζώνης
t_j	Χρονική στιγμή λήξης ενός spike εύρους ζώνης
θ	Συντελεστής κατωφλίου για spikes εύρους ζώνης
α_i	Ο αριθμός των ενεργών αντικειμένων τη χρονική στιγμή i
$\overline{cd}$	Η μέση διαφορά μεταξύ δύο διαδοχικών αριθμών των ενεργών αντικειμένων
sp_i^{i+1}	Δυαδική συνάρτηση για την ανίχνευση των spikes

Table 1.1: Κύριες παράμετροι για την υλοποίηση του SPID

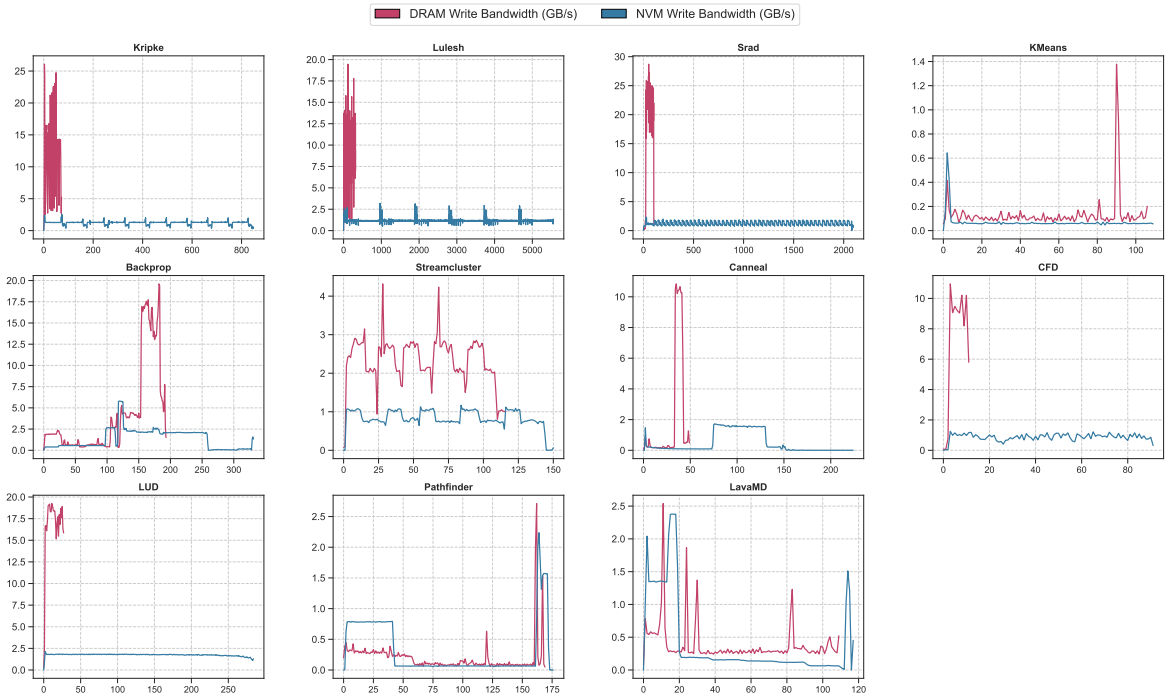

Figure 1.3.3: Εύρος ζώνης εγγραφής της Dram (κόκκινο) έναντι της NVM (μπλε) κατά τη διάρκεια εκτέλεσης (secs.) των benchmarks .

μορφές. Μια περιοχή με spike εύρους ζώνης αντιστοιχεί σε ένα χρονικό διάστημα $[t_i, t_j]$, όπου: t_i είναι η αρχική χρονική στιγμή που το εύρος ζώνης εγγραφής στη DRAM υπερβαίνει ένα προκαθορισμένο όριο σε σχέση με τον μέσο όρο του εύρους ζώνης εγγραφής στη NVM εντός του τρέχοντος ολισθαίνον παραθύρου σταθερού μήκους s (Εξίσωση 1.3.1), και t_j είναι η τελική χρονική στιγμή που το εύρος ζώνης εγγραφής στη DRAM πέφτει κάτω από ένα παρόμοιο όριο (Εξίσωση 1.3.2). Η μεθοδολογία μας καταγράφει αποτελεσματικά τις φάσεις υψηλής χρήσης μνήμης, επιτρέποντας την προσαρμοστική τοποθέτηση δεδομένων και τη βελτιστοποίηση της χρήσης πόρων σε ετερογενή περιβάλλοντα μνήμης. Ας θεωρήσουμε ότι θ είναι ένας συντελεστής κατωφλίου που ορίζει ένα spike στο εύρος ζώνης της DRAM σε σχέση με τον μέσο όρο του εύρους ζώνης της NVM. Τότε, οι στιγμές έναρξης t_i και λήξης t_j μιας αιχμής εύρους ζώνης ορίζονται ως εξής:

$$t_i = \min\{t : BW_{\text{DRAM}}(t) > \theta \cdot \overline{BW}_{\text{NVM}}(s)\} \tag{1.3.1}$$

$$t_j = \min\{t > t_i : BW_{\text{DRAM}}(t) < \theta \cdot \overline{BW}_{\text{NVM}}(s)\} \tag{1.3.2}$$

όπου $BW_{\text{DRAM}}(t)$ είναι το εύρος ζώνης εγγραφής στη DRAM τη χρονική στιγμή t και $\overline{BW}_{\text{NVM}}(s)$ είναι ο μέσος όρος του εύρους ζώνης εγγραφής στη NVM κατά το ολισθαίνον παράθυρο s. Το όριο θ καθορίζεται πειραματικά και η επιλογή του αφήνεται στο χρήστη.

Active Object Spike Detector: Ομοίως, ανιχνεύουμε περιοχές με spikes ενεργών αντικειμένων εξετάζοντας τη μεταβολή στις διαδοχικές τιμές του αριθμού των ενεργών αντικειμένων. Αυτή τη φορά, εργαζόμαστε πάνω στις κυματομορφές του Σχήματος 1.3.1 και συγκεκριμένα ασχολούμαστε με την πράσινη κυματομορφή, η οποία παρουσιάζει τον αριθμό τον ενεργών αντικειμένων συναρτήσει του χρόνου. Αναλυτικότερα, παράγουμε διακριτές τιμές για τον αριθμό των ενεργών αντικειμένων, σχηματίζοντας μια ακολουθία στοιχείων α_i, που υποδηλώνει τον αριθμό των ενεργών αντικειμένων συναρτήσει του χρόνου εκτέλεσης, ως εξής: $\{\alpha_0, \alpha_1, \ldots, \alpha_n\}$. Δεδομένης της σειράς αυτών των τιμών, ο αλγόριθμός μας υπολογίζει τις απόλυτες διαφορές μεταξύ των διαδοχικών τιμών, που αναπαρίστανται ως $\{|a_1 - a_0|, \ldots, |a_n - a_{n-1}|\}$. Υπολογίζουμε τη μέση απόλυτη διαφορά, που ορίζεται ως κρίσιμο κατώφλι

$\overline{cd}$ από τις τιμές α_i, όπως φαίνεται στην Εξίσωση 1.3.3. Η συνάρτηση sp_i^{i+1} (Εξίσωση 1.3.4) εντοπίζει spikes ενεργών αντικειμένων συγκρίνοντας τις διαφορές των αντικειμένων με το κρίσιμο κατώφλι. Συγκεκριμένα, η συνάρτηση επιστρέφει $sp_i^{i+1} = 1$ όταν $|\alpha_{i+1} - \alpha_i|$ ξεπερνά το $\overline{cd}$, και $sp_i^{i+1} = 0$ διαφορετικά. Spikes τα οποία είναι γειτονικά συγχωνεύονται σε μεγαλύτερα διαστήματα, καταγράφοντας περιόδους υψηλής δραστηριότητας αντικειμένων, χωρίς εκτενή ανάλυση σε επίπεδο αντικειμένου.

$$\overline{cd} = \frac{1}{n}\sum_{i=1}^{n}|\alpha_{i+1} - \alpha_i| \tag{1.3.3}$$

$$\mathrm{sp}_i^{i+1} = \begin{cases} 1, & \text{if } |\alpha_{i+1} - \alpha_i| > \overline{cd} \\ 0, & \text{if } |\alpha_{i+1} - \alpha_i| \leq \overline{cd} \end{cases} \quad \forall i \in \{0, .., n-1\} \tag{1.3.4}$$

1.3.3 Βήμα 3: Correlation, K-Max Selection & Dynamic Data Placement

Μετά την αναγνώριση των περιοχών spikes εύρους ζώνης μνήμης και ενεργών αντικειμένων (1.3.2), ο επόμενος στόχος μας είναι η συσχέτισή τους και η πιθανοτική εξερεύνηση των spikes ενεργών αντικειμένων που είναι πιθανότερο να σχετίζονται με κάθε επερχόμενο spike εύρους ζώνης. Για να το επιτύχουμε αυτό, χρησιμοποιούμε ένα *interval tree* [28] για να καταγράψουμε και να εξαγάγουμε αποτελεσματικά τα spikes ενεργών αντικειμένων που εμφανίζονται στο χρονικό διάστημα μεταξύ διαδοχικών spikes εύρους ζώνης (3a). Τα *interval trees* επιτρέπουν ταχεία εξαγωγή σε χρόνο $O(\log N)$, καθιστώντας τα κατάλληλα για γρήγορη διαπέραση και εξερεύνηση.

Κάθε κόμβος στο *interval tree* αντιστοιχεί στο χρονικό διάστημα $[t_i, t_j]$ ενός spike ενεργών αντικειμένων. Για την εξερεύνηση των spikes ενεργών αντικειμένων μέσα στο εξεταζόμενο χρονικό διάστημα που ορίζεται από δύο διαδοχικά spikes εύρους ζώνης, το *interval tree* επιστρέφει τους κόμβους/διαστήματα που επικαλύπτονται με αυτό. Αφού ανακτηθούν τα σχετικά spikes, ο αλγόριθμός μας φιλτράρει και επιλέγει τα *K-max* (τα K μεγαλύτερα) spikes βάσει των δεσμευμένων bytes (allocated bytes) στα επιλεγμένα spikes ενεργών αντικειμένων (3b), τα οποία είναι και τα πιο κρίσιμα για αποδοτική τοποθέτηση.

Τέλος, η δυναμική τοποθέτηση δεδομένων πραγματοποιείται κατά την εκτέλεση (3c). Τα K spikes ενεργών αντικειμένων που εντοπίστηκαν προηγουμένως χαρακτηρίζονται ως *hot regions*, ενώ οι υπόλοιπες περιοχές χαρακτηρίζονται ως *cold regions*. Κάθε περιοχή συσχετίζεται με τον συνολικό αριθμό των bytes που έχουν κατανεμηθεί σε αυτήν. Για να μετριαστεί η πιθανή υποβάθμιση της απόδοσης που προκαλείται από τον πιο αργό χρόνο πρόσβασης της NVM, οι *hot regions* δρομολογούνται για τοποθέτηση στη μνήμη DRAM.

Αφού εντοπιστούν τα spikes ενεργών αντικειμένων για τοποθέτηση στη γρηγορότερη μνήμη, ο αλγόριθμος αξιοποιεί τα ντετερμινιστικά μοτίβα δέσμευσης μνήμης των benchmarks για να λάβει αποφάσεις τοποθέτησης κατά την εκτέλεση για τις *cold regions*. Κάθε περιοχή διαχειρίζεται ως ξεχωριστή μονάδα, με τα στοιχεία της αντίστοιχης περιοχής να τοποθετούνται στη NVM. Οι *cold regions* που αντιστοιχίζονται στη NVM είναι επιρρεπείς σε φαινόμενα όπως το write amplification και το write throttling [29], λόγω των συχνών μικρών δεσμεύσεων αντικειμένων.

Για την αντιμετώπιση αυτών των προβλημάτων, εφαρμόζεται ένας μηχανισμός εναλλασσόμενης τοποθέτησης, όπου διαδοχικές *cold regions* ομαδοποιούνται σε μεγαλύτερες ομάδες. Τέλος, η τοποθέτηση δεδομένων πραγματοποιείται μέσω της ενσωματωμένης βιβλιοθήκης μας, που ονομάζεται `SPMalloc` (3d), η οποία περιγράφεται λεπτομερώς στην Ενότητα 1.3.4.

1.3.4 Τεχνική Υλοποίηση

Σχεδιάζουμε τη βιβλιοθήκη `SPMalloc`, μια ανοιχτού κώδικα βιβλιοθήκη, η οποία αντικαθιστά τις συναρτήσεις `malloc, calloc, realloc, free` και `new/delete` για τις γλώσσες προγραμματισμού C και C++, αντίστοιχα. Οι κλήσεις προς τη βιβλιοθήκη μας αναχαιτίζονται κατά την εκτέλεση χρησιμοποιώντας το `LD_PRELOAD`, συνδέοντας τη βιβλιοθήκη στο εκτελέσιμο αρχείο.

Η βιβλιοθήκη μας χρησιμοποιεί το *memkind API* [30], το οποίο επιτρέπει στους προγραμματιστές να τοποθετούν δεδομένα είτε στη μνήμη DRAM είτε στη μνήμη Optane DCPM. Στο Σχήμα 1.3.4 παρουσιάζεται η σύγκριση του χρόνου εκτέλεσης των benchmarks που περιγράφονται στην Ενότητα 1.4.1 για δυναμικές δεσμεύσεις στη μνήμη DRAM, οι οποίες πραγματοποιούνται μέσω της προεπιλεγμένης `malloc()` και της `SPMalloc()` που παρέχουμε. Παρατηρούμε μια ελάχιστη μέση διαφορά απόδοσης της τάξης του 1.8%, γεγονός που υποδεικνύει ότι οι δεσμεύσεις μνήμης μέσω της βιβλιοθήκης μας επιβάλλουν αμελητέα επιβάρυνση κατά την εκτέλεση.

Ως αποτέλεσμα, θεσπίζουμε μια ενοποιημένη πολιτική δέσμευσης μνήμης, κατά την οποία όλες οι δυναμικές δεσμεύσεις διενεργούνται μέσω του *memkind API*, διευκολύνοντας τη διαβάθμιση και τον έλεγχο της μνήμης. Επιπλέον, η `SPMalloc` είναι εμπλουτισμένη με δυνατότητες profiling, ώστε να υποστηρίζει τη διαχείριση μνήμης σε πραγματικό χρόνο.

Για να μετριάσουμε την επιβάρυνση που σχετίζεται με εργαλεία πλήρους profiling των αντικειμένων [21], ενσωματώνουμε έναν profiler βασισμένο σε δειγματοληψία μέσα στη βιβλιοθήκη. Ο profiler αυτός παρακολουθεί και καταγράφει τη χρήση μνήμης, παρακολουθώντας τα bytes που δεσμεύονται και τον αριθμό των ενεργών αντικειμένων κατά τη διάρκεια της εκτέλεσης. Σχεδιασμένη ως ένα επιπλέον thread στο παρασκηνίο της εφαρμογής προς εκτέλεση, η `SPMalloc` επιτρέπει τη συνεχή λήψη αποφάσεων χωρίς να επηρεάζει παρεμβατικά την απόδοση της εφαρμογής, παρέχοντας πληροφορίες για τις δεσμεύσεις μνήμης και την τοποθέτηση σε πραγματικό χρόνο για αποτελεσματική διαχείριση μνήμης.

1.4 Αξιολόγηση

1.4.1 Πειραματική Διάταξη

Τα πειράματα διεξήχθησαν σε έναν υψηλών προδιαγραφών server που αποτελείται από δύο 20-core Intel Xeon Gold 5218R CPUs @ 2.10 GHz, με 4x32GB DDR4 DIMMs και 6x256GB Optane DC NVDIMMs. Η Intel Optane DCPM διαμορφώθηκε σε App-Direct mode με το σύστημα αρχείων EXT4-DAX. Για την ανάπτυξη των εφαρμογών χρησιμοποιήθηκε η έκδοση 1.11 του Persistent Memory Development Kit (PMDK) και ο gcc-13.3.

Η προτεινόμενη λύση μας αξιολογείται σε 11 πραγματικά benchmarks, που προέρχονται από τα `PARSEC` [31], `Rodinia` [32] και `CORAL-2` [33], καλύπτοντας ένα ευρύ σύνολο τομέων εφαρμογών, όπως ML workloads, επεξεργασία εικόνας και φυσική. Εστιάζουμε σε εφαρμογές με έντονη χρήση μνήμης, όπως τα *Streamcluster, LUD, Kripke, CFD, Backprop, Srad*, *Lulesh*, και σε ισορροπημένα workloads μεταξύ απαιτήσεων επεξεργασίας και μνήμης, όπως τα *Canneal, LavaMD* και *Pathfinder*.

Συγκρίνουμε την προτεινόμενη πολιτική τοποθέτησης με 5 πολιτικές τοποθέτησης:

1. *DRAM-all*, όπου όλα τα δεδομένα τοποθετούνται στη DRAM,
2. *Optane-all*, όπου όλα τα δεδομένα τοποθετούνται στη NVM,
3. *Round-Robin*, όπου τα δεδομένα τοποθετούνται κυκλικά σε κάθε τύπο μνήμης,
4. *Random*, όπου τα δεδομένα τοποθετούνται τυχαία, και
5. *Phase-based* [27], που αποτελεί μια σύγχρονη λύση τοποθέτησης δεδομένων. Για αυτήν, οι διακριτές φάσεις χωρίζονται σε χρονικά διαστήματα των 5 δευτερολέπτων.

Οι εξεταζόμενες πολιτικές δυναμικής τοποθέτησης δεδομένων αξιολογούνται με βάση την απόδοση, την κατανάλωση ενέργειας και την επίδρασή τους στη διάρκεια ζωής της NVM, σύμφωνα με τον αριθμό εγγραφών που πραγματοποιούνται στα DIMMs της NVM. Η κατανάλωση ενέργειας και το εύρος ζώνης μνήμης μετριούνται χρησιμοποιώντας το Intel PCM [34], ένα εργαλείο που επιτρέπει την παρακολούθηση της ισχύος και του εύρους ζώνης σε πραγματικό χρόνο μέσω hardware counters, ενώ για τη μέτρηση του αριθμού των εγγραφών και αναγνώσεων, χρησιμοποιούμε το εργαλείο `ipmctl` [35], το οποίο αποτελεί το πρότυπο για τη ρύθμιση, διαχείριση και παρακολούθηση των NVDIMMs.

1.4.2 Αποτελέσματα

Performance Analysis: Στο Σχήμα 1.4.1 παρουσιάζεται ο κανονικοποιημένος χρόνος εκτέλεσης (επάνω) και η κατανάλωση ενέργειας (κάτω) για όλα τα εξεταζόμενα benchmarks, αντίστοιχα. Τα αποτελέσματα για τον χρόνο εκτέλεσης κανονικοποιούνται ως προς την πολιτική τοποθέτησης *Optane-all*.

Η πολιτική *DRAM-all* παρέχει τον βέλτιστο χρόνο εκτέλεσης σε όλα τα benchmarks, λόγω της βέλτιστης καθυστέρησης τόσο για τις λειτουργίες ανάγνωσης όσο και για τις λειτουργίες εγγραφής, ενώ η πολιτική *Optane-all* παρουσιάζει τον μεγαλύτερο χρόνο εκτέλεσης λόγω της σημαντικής επιβάρυνσης από τις εγγραφές. Παρατηρούμε ότι το SPID υπερέχει των αξιολογούμενων πολιτικών τοποθέτησης, επιτυγχάνοντας κατά μέσο όρο 28.86%, 34.16% και 29.43% μικρότερο χρόνο εκτέλεσης σε σύγκριση με τις λύσεις *Round-Robin*, *Random* και *Phase-based*, αντίστοιχα. Οι πολιτικές τοποθέτησης *Round-Robin* και *Random* είναι υπερβολικά απλοϊκές για να αποδώσουν αποτελεσματικά σε ένα ευρύ φάσμα εφαρμογών.

Επιπλέον, το SPID επιτυγχάνει απόδοση που είναι μόλις 15.15% πιο αργή από τη βέλτιστη λύση σε DRAM (*DRAM-only*), κατά μέσο όρο, αποδεικνύοντας την αποτελεσματικότητά του στην αξιοποίηση τόσο της DRAM όσο και της NVM για βέλτιστη απόδοση σε ποικίλους φόρτους εργασίας.

Το κύριο πλεονέκτημα του SPID σε σχέση με την προσέγγιση *Phase-based* έγκειται στο γεγονός ότι η τελευταία βασίζεται μόνο σε φάσεις που ορίζονται από το εύρος ζώνης μνήμης, αγνοώντας τη συμπεριφορά των δεσμεύσεων μνήμης εντός αυτών των φάσεων, η οποία είναι κρίσιμη για τη συνολική απόδοση. Αυτό οδηγεί σε υποβάθμιση της απόδοσης. Το SPID ξεπερνά αυτόν τον περιορισμό, εντοπίζοντας αποτελεσματικά κρίσιμες περιοχές δέσμευσης μνήμης. Αυτή η διάκριση επιτρέπει στο SPID να υπερέχει σαφώς της προσέγγισης *Phase-based* σε benchmarks όπου οι κρίσιμες δεσμεύσεις δεν ακολουθούνται άμεσα από περιοχές υψηλής κατανάλωσης εύρους ζώνης (π.χ. *backprop*, *srad*, *Kripke*, *Canneal*), επιτυγχάνοντας κατά μέσο όρο 3.7× αύξηση της ταχύτητας απόδοσης.

Τέλος, για benchmarks που χαρακτηρίζονται ως μη εντατικά στη χρήση μνήμης (π.χ. *pathfinder, lavaMD, Kmeans*), όλες οι πολιτικές παρουσιάζουν παρόμοια απόδοση, υποδεικνύοντας ότι οι στρατηγικές τοποθέτησης μνήμης έχουν ελάχιστο αντίκτυπο σε τέτοιες περιπτώσεις.

> ⋆ ***Κύριο Συμπέρασμα 2:*** *Το SPID εντοπίζει και δίνει προτεραιότητα σε κρίσιμες περιοχές δέσμευσης μνήμης, βελτιστοποιώντας την απόδοση μέσω τοποθετήσεων που αντικατοπτρίζουν τόσο τη καθοριστική σημασία του εύρους ζώνης όσο και των δεσμεύσεων μνήμης.*

Ανάλυση Κατανάλωσης Ενέργειας: Η κατανάλωση ενέργειας στα DIMMs του συστήματος μνήμης για τα εξετασμένα benchmarks αποτυπώνεται στο Σχήμα 1.4.1 (κάτω) σε λογαριθμική κλίμακα. Τα αποτελέσματα που προέκυψαν ευθυγραμμίζονται στενά με τους χρόνους εκτέλεσης, δείχνοντας ότι το SPID επιτυγχάνει 29.81%, 34.86%, και 30.15% βελτιωμένη κατανάλωση ενέργειας σε σχέση με τις πολιτικές τοποθέτησης *Round-Robin*, *Random*, και *Phase-based*, αντίστοιχα. Η βελτιστοποιημένη κατανάλωση ενέργειας του SPID αποδίδεται κυρίως σε δύο βασικούς παράγοντες: (i) τα αντικείμενα με υψηλή συχνότητα εγγραφών τοποθετούνται στη DRAM, αποφεύγοντας την υψηλή κατανάλωση ενέργειας που σχετίζεται με τις εγγραφές σε NVM, και (ii) τα αντικείμενα με μικρή συχνότητα εγγραφών κατανέμονται σε NVM χαμηλής κατανάλωσης, βοηθώντας στη διατήρηση της απόδοσης ενώ καταναλώνουν λιγότερη ενέργεια. Επιπλέον, η στρατηγική δυναμικής τοποθέτησης βασισμένη σε spikes που χρησιμοποιεί το SPID διασφαλίζει ότι οι περιοχές μνήμης διαχειρίζονται αποτελεσματικά, λόγω του μηχανισμού monitoring ενεργών αντικειμένων, ο οποίος ενισχύει τη λήψη αποφάσεων τοποθέτησης, σε αντίθεση με την *Phase-based*, όπου τα ενεργά αντικείμενα δεν λαμβάνονται υπόψη. Αυτή η δυναμική προσέγγιση αντιτίθεται στις παραδοσιακές στατικές μεθόδους τοποθέτησης, οι οποίες συχνά οδηγούν σε υποβέλτιστη ενεργειακή απόδοση λόγω λιγότερο ευφυών αναθέσεων μνήμης.

> ⋆ ***Κύριο Συμπέρασμα 3:*** *Το SPID μειώνει τη σπατάλη ενέργειας λόγω της αποτελεσματικής τοποθέτησης των spikes και non-spikes σε DRAM και NVM, αντίστοιχα.*

NVM Accesses & Lifetime Analysis: Η αποτελεσματικότητα μιας πολιτικής τοποθέτησης δεδομένων πρέπει να αξιολογείται με βάση την ικανότητά της να σέβεται την περιορισμένη αντοχή της NVM σε

εγγραφές. Οι εγγραφές που πραγματοποιούνται στη NVM λειτουργούν ως δείκτες φθοράς της NVM [21]. Σκοπός μας είναι η επέκταση της διάρκειας ζωής της NVM, και γι' αυτό αξιολογούμε τις αναγνώσεις και τις εγγραφές στην Optane, δίνοντας ιδιαίτερη προσοχή στη μείωση των εγγραφών. Η Εικόνα 1.4.2 απεικονίζει τον συνολικό αριθμό των εγγραφών (πάνω) και των αναγνώσεων (κάτω), αντίστοιχα, για τις πολιτικές *Optane-all* (baseline), *Phase-based* και SPID στα εξεταζόμενα benchmarks. Τα αποτελέσματά μας δείχνουν ότι το SPID επιτυγχάνει 73.03% και 33.02% λιγότερες εγγραφές κατά μέσο όρο σε σύγκριση με τις *Optane-all* και *Phase-based*, αντίστοιχα. Ομοίως, σε ό,τι αφορά τις αναγνώσεις, το SPID επιτυγχάνει 70.81% και 30.3% λιγότερες αναγνώσεις κατά μέσο όρο σε σύγκριση με τις *Optane-all* και *Phase-based*, αντίστοιχα. Η επιλεκτική τοποθέτηση μη εντατικών εγγραφών σε NVM και η αποδοτική τοποθέτηση που επιτυγχάνεται στο σύστημα DRAM/NVM οδηγούν στη μείωση της συχνότητας των εγγραφών στη NVM. Ως αποτέλεσμα, το SPID όχι μόνο παρέχει βελτιωμένη απόδοση σε σχέση με όλες τις άλλες πολιτικές που εξετάστηκαν, αλλά μειώνει επίσης τη φθορά, βελτιώνοντας τη συνολική διάρκεια ζωής της NVM πιο αποτελεσματικά από άλλες λύσεις.

> ⋆ ***Κύριο Συμπέρασμα 4:*** *Το SPID επεκτείνει τη διάρκεια ζωής της NVM μειώνοντας σημαντικά τις εγγραφές, εξισορροπώντας την πρόσβαση μεταξύ DRAM και NVM, βελτιώνοντας έτσι τη συνολική αντοχή και διάρκεια ζωής.*

Ανάλυση Πολιτικής Τοποθέτησης: Τέλος, προκειμένου να αποκτήσουμε μια βαθύτερη κατανόηση της συμπεριφοράς του SPID, εξετάζουμε την κατανομή δεδομένων στο ετερογενές σύστημα μνήμης DRAM/NVM. Τα pie plots που παρουσιάζονται στο Σχήμα 1.4.3 απεικονίζουν το ποσοστό των bytes που έχουν κατανεμηθεί σε DRAM και NVM κατά την εκτέλεση κάθε benchmark με χρήση του SPID. Αυτές οι κατανομές μπορούν να κατηγοριοποιηθούν σε τρεις διακριτές κατηγορίες:

- **Ισορροπημένη Κατανομή**: Για benchmarks με έντονη χρήση μνήμης, όπως τα *Kripke*, *Lulesh*, *Srad*, *Canneal*, *Streamcluster* και *CFD*, τα οποία παρουσιάζουν επίσης πολύπλοκα μοτίβα κατανομής, απαιτούνται κρίσιμες αποφάσεις τοποθέτησης δεδομένων. Το SPID επιτυγχάνει μια ισορροπημένη κατανομή bytes μεταξύ DRAM και NVM για αυτά τα benchmarks, εξασφαλίζοντας ότι τα write-intensive αντικείμενα τοποθετούνται στη DRAM, ενώ τα non write-intensive αντικείμενα κατανέμονται στη NVM χαμηλής κατανάλωσης ενέργειας.
- **Κατανομή Υπέρ της NVM**: Για benchmarks με χαμηλή χρήση μνήμης που βασίζονται κυρίως στην αξιοποίηση του CPU, όπως τα *kmeans*, *pathfinder* και *lavaMD*, το SPID κατανέμει κυρίως δεδομένα στη NVM. Αυτή η προσέγγιση βελτιστοποιεί την απόδοση ενώ ελαχιστοποιεί την κατανάλωση ενέργειας, αξιοποιώντας τα χαρακτηριστικά χαμηλής κατανάλωσης της NVM.
- **Κατανομή Υπέρ της DRAM**: Τέλος, για ορισμένα benchmarks, όπως τα *Lud* και *Backprop*, το SPID κατανέμει όλα τα bytes στη DRAM. Αυτή η συμπεριφορά οφείλεται σε δύο παράγοντες: (i) τα benchmarks αυτά εμφανίζουν φάσεις έντονης χρήσης μνήμης κατά την εκτέλεση, οδηγώντας σε bandwidth spikes, και (ii) χαρακτηρίζονται από ένα αρχικό spike ενεργών αντικειμένων—συνήθως στην αρχή του προγράμματος—μετά το οποίο δεν εμφανίζονται περαιτέρω δεσμεύσεις μνήμης. Συνεπώς, το SPID τοποθετεί το αρχικό spike στη DRAM, με αποτέλεσμα το σύνολο των δεδομένων να παραμένει στη DRAM.

Το Σχήμα 1.4.4 παρέχει πρόσθετες πληροφορίες για τις προαναφερθείσες κατανομές. Συγκεκριμένα, απεικονίζει τα **μη μηδενικά** κατανεμημένα bytes (κόκκινο) και το bandwidth της DRAM (κίτρινο) κατά τη διάρκεια εκτέλεσης κάθε benchmark. Όπως παρατηρείται, benchmarks όπως τα *Lulesh*,*Kripke*, *Streamcluster* και *Canneal* που παρουσιάζουν πολύπλοκα μοτίβα κατανομής, επιτυγχάνουν μια ισορροπημένη κατανομή μεταξύ DRAM και NVM. Αντίθετα, benchmarks με χαμηλή χρήση μνήμης, όπως τα *Kmeans* και *LavaMD*, διαχειρίζονται αποκλειστικά μέσω της Optane. Τέλος, benchmarks με έντονη χρήση μνήμης, όπως τα *Backprop* και *Lud*, που δεσμεύουν όλη τη μνήμη που χρειάζονται στην αρχή της εκτέλεσης, διαχειρίζονται εξ ολοκλήρου από τη DRAM.

Το διάγραμμα violin που παρουσιάζεται στο Σχήμα 1.4.5 απεικονίζει το μέσο ποσοστό των bytes που έχουν κατανεμηθεί στη DRAM και στη NVM για όλα τα αξιολογημένα benchmarks. Η ανάλυσή μας αποκαλύπτει ότι το 52.04% των δεδομένων έχει κατανεμηθεί στη DRAM και το 47.96% στη NVM, με

τα ποσοστά κατανομής να παρουσιάζουν ελάχιστη μεταβλητότητα και να ευθυγραμμίζονται στενά με τις τιμές της διάμεσου. Αυτή η κατανομή αναδεικνύει την ικανότητα του SPID να εξισορροπεί αποτελεσματικά την τοποθέτηση δεδομένων μεταξύ των δύο τύπων μνήμης, συμβάλλοντας στην ισορροπημένη απόδοση (Σχήμα 1.4.1 κορυφή), στην κατανάλωση ενέργειας (Σχήμα 1.4.1 βάση) και στις προσβάσεις μνήμης (Σχήμα 1.4.2), όπως συζητήθηκε νωρίτερα σε αυτή την ενότητα.

⋆ ***Κύριο Συμπέρασμα 5:*** *Η ισορροπημένη τοποθέτηση του SPID αποτελεί τον βασικό παράγοντα που επιτρέπει τη συν-βελτιστοποίηση της απόδοσης, της ενέργειας και των προσβάσεων εγγραφής.*

1.5 Συμπεράσματα και Μελλοντική δουλειά

1.5.1 Συμπεράσματα

Σε αυτή την εργασία, παρουσιάζουμε το SPID, μια καινοτόμα προσέγγιση για profiling χαμηλού κόστους και τοποθέτηση δεδομένων, σχεδιασμένη για ετερογενή συστήματα μνήμης DRAM/NVM. Το SPID αξιοποιεί αποτελεσματικά τη συσχέτιση μεταξύ έντονων φάσεων δέσμευσης μνήμης και περιοχών με υψηλό εύρος ζώνης εγγραφών για τη βελτιστοποίηση της τοποθέτησης δεδομένων. Η μεθοδολογία μας παρουσιάζει βελτιώσεις στην απόδοση κατά μέσο όρο 30.82% σε σχέση με προηγούμενες τεχνικές, ενώ μειώνει την κατανάλωση ενέργειας κατά μέσο όρο 31.61% και διαχειρίζεται αποτελεσματικά την περιορισμένη αντοχή των NVM μειώνοντας τον αριθμό εγγραφών. Επιπλέον, παρουσιάζουμε την SPMalloc, μια επεκτάσιμη και ανοιχτού κώδικα βιβλιοθήκη, σχεδιασμένη να αναχαιτίζει τις δυναμικές δεσμεύσεις δεδομένων κατά τον χρόνο εκτέλεσης, επιτρέποντας την ακριβή παρακολούθηση των μοτίβων δέσμευσης μνήμης κατα την εκτέλεση της επιθυμητής εφαρμογής, εισάγοντας ελάχιστο επιπλέον κόστος.

1.5.2 Μελλοντική δουλειά

Υπάρχουν αρκετές υποσχόμενες κατευθύνσεις για μελλοντική εργασία, οι οποίες θα μπορούσαν να κάνουν αυτό το πλαίσιο πιο αξιόπιστο, ευέλικτο και αποδοτικό. Αυτές οι βελτιώσεις θα μπορούσαν να στοχεύουν τόσο στις δυνατότητες και τις λειτουργίες ανάλυσης της SPMalloc, όσο και στον αλγόριθμο και τη μεθοδολογία τοποθέτησης του SPID. Με την αντιμετώπιση αυτών των περιοχών, το SPID μπορεί να εξελιχθεί σε ένα πιο ανθεκτικό και ευφυές πλαίσιο για τη διαχείριση ετερογενούς μνήμης. Μερικές πιθανές βελτιστοποιήσεις περιλαμβάνουν:

- **Λεπτότερη Ανάλυση στο Στάδιο Profiling:** Η βελτίωση του σταδίου profiling μέσω ενσωμάτωσης πιο προηγμένων εργαλείων παρακολούθησης, όπως ένα προσαρμοσμένο εργαλείο Intel Pin, το οποίο θα λειτουργεί σε συνδυασμό με τις υπάρχουσες διαδικασίες παρακολούθησης του SPMalloc. Τα αρχικά δεδομένα profiling που παρέχονται από την SPMalloc θα μπορούσαν να καθοδηγήσουν αυτά τα εργαλεία να εστιάσουν σε συγκεκριμένες χρονικές φάσεις, μειώνοντας τον φόρτο και βελτιώνοντας την αποδοτικότητα. Αυτή η προσέγγιση θα μπορούσε να επιτρέψει στο SPID να προσαρμόσει την ανάλυσή του στη δυναμική συμπεριφορά των εφαρμογών πιο αποτελεσματικά.
- **Προσαρμοστικότητα σε Θορυβώδη Περιβάλλοντα:** Τα πειράματά μας πραγματοποιήθηκαν σε ελεγχόμενο περιβάλλον, όπως συζητήθηκε στην Ενότητα 1.4.1 όπου ο θόρυβος ήταν ελάχιστος. Για να βελτιωθεί η αξιοπιστία του πλαισίου, μελλοντική εργασία θα μπορούσε να επικεντρωθεί στη βελτίωση των σταδίων profiling και τοποθέτησης ώστε να λειτουργούν αποτελεσματικά ακόμα και υπό υψηλό φορτίο συστήματος και θορυβώδεις συνθήκες. Αυτό θα αύξανε την προσαρμοστικότητα του SPID σε πραγματικά σενάρια, καθιστώντας το κατάλληλο για διάφορα και απρόβλεπτα περιβάλλοντα.
- **Ευφυή Εργαλεία Τοποθέτησης:** Η ενσωμάτωση προηγμένων εργαλείων στο στάδιο τοποθέτησης για πιο ακριβείς και προσαρμοσμένες αποφάσεις. Για παράδειγμα, η ενσωμάτωση μοντέλων μηχανικής μάθησης και ευρετικών μεθόδων θα μπορούσε να επιτρέψει στο SPID να προβλέψει μελλοντικά μοτίβα δέσμευσης μνήμης με βάση τα δεδομένα profiling της SPMalloc. Αυτές οι

δυνατότητες πρόβλεψης θα μπορούσαν να οδηγήσουν σε εξυπνότερες στρατηγικές τοποθέτησης που μεγιστοποιούν την απόδοση και ελαχιστοποιούν τα σημεία συμφόρησης στη μνήμη.

- **Επέκταση Λειτουργικότητας της SPMalloc:** Η επέκταση των δυνατοτήτων παρακολούθησης της SPMalloc ώστε να καταγράφει επιπλέον μετρήσεις, όπως τη διάρκεια ζωής των αντικειμένων δέσμευσης, θα μπορούσε να προσφέρει βαθύτερες γνώσεις για τα μοτίβα χρήσης μνήμης. Παράλληλα, η προσθήκη περισσότερης λογικής στις λειτουργίες δέσμευσης και αποδέσμευσης θα μπορούσε να διευκολύνει καλύτερες αποφάσεις τοποθέτησης σε πραγματικό χρόνο. Για παράδειγμα, η ενσωμάτωση αλγορίθμων για την παρακολούθηση και κατηγοριοποίηση των συχνοτήτων πρόσβασης στη μνήμη ή των μεγεθών της, θα μπορούσε να επιτρέψει πιο δυναμικές πολιτικές δέσμευσης.
- **Ενσωμάτωση με Διαχείριση σε Επίπεδο Συστήματος:** Μια άλλη πιθανή κατεύθυνση είναι η εξερεύνηση της ενσωμάτωσης με διαχειριστές μνήμης σε επίπεδο λειτουργικού συστήματος ή συστήματα εικονικής μνήμης για τη βελτίωση των αποφάσεων τοποθέτησης. Εργαζόμενο σε επίπεδο ολόκληρου του συστήματος, το SPID θα μπορούσε να λαμβάνει αποφάσεις τοποθέτησης που λαμβάνουν υπόψη όλη τη χρήση της μνήμης στο σύστημα και όχι μόνο την εφαρμογή υπό δοκιμή.
- **Υποστήριξη για Αναδυόμενες Τεχνολογίες Μνήμης:** Τα μελλοντικά ετερογενή συστήματα μνήμης μπορεί να περιλαμβάνουν νέες τεχνολογίες πέρα από Optane και DRAM, όπως το CXL (Compute Express Link) ή η μη πτητική RAM. Η επέκταση των στρατηγικών profiling και τοποθέτησης του SPID για να υποστηρίζουν αυτούς τους νέους τύπους μνήμης θα το καθιστούσε περισσότερο ευέλικτο και έτοιμο για μελλοντική χρήση.

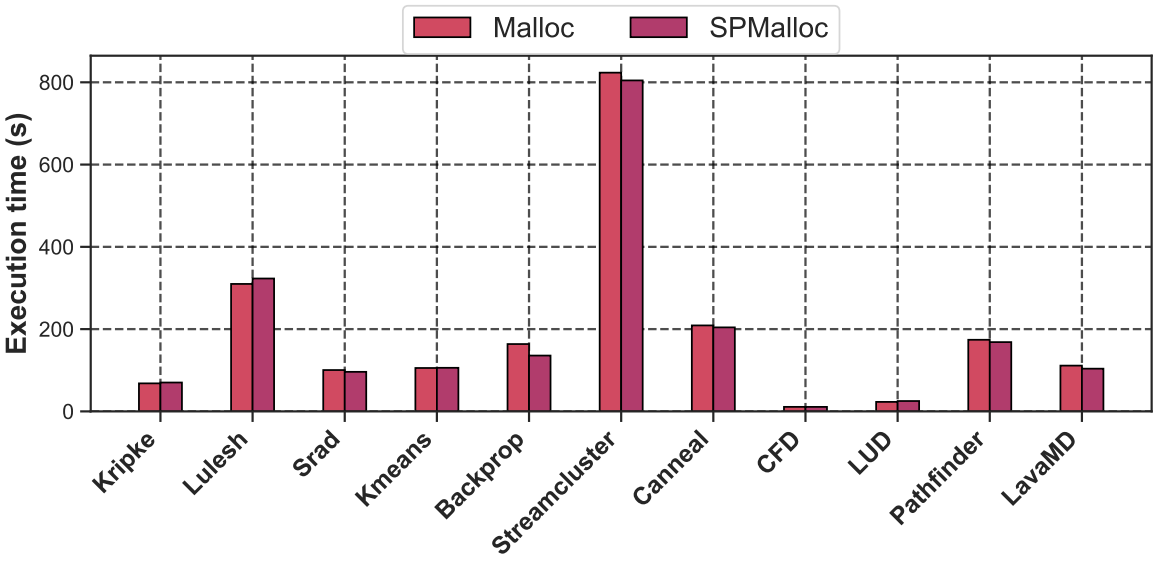

Figure 1.3.4: Σύγκριση της χρονικής καθυστέρησης της `SPMalloc` έναντι της `malloc` .

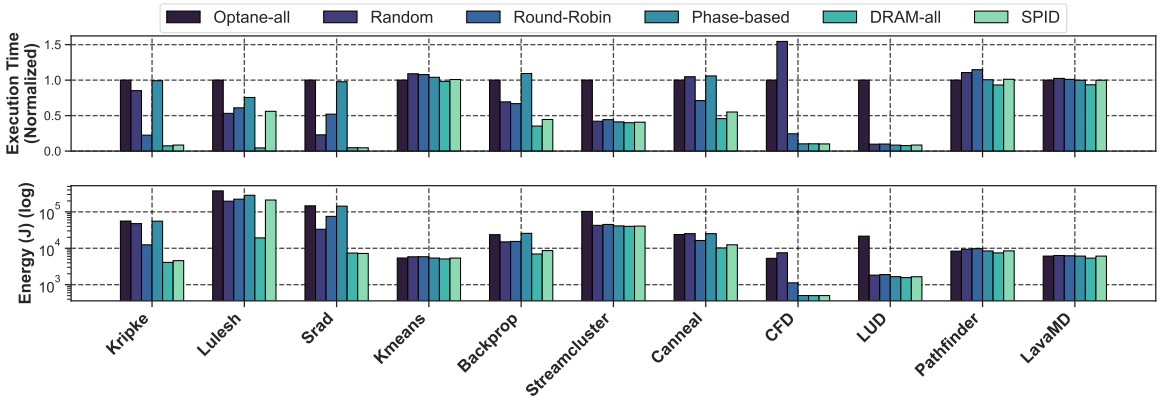

Figure 1.4.1: Σύγκριση του χρόνου εκτέλεσης (επάνω) και της κατανάλωσης ενέργειας (κάτω) του `SPID` σε σχέση με τις εξεταζόμενες πολιτικές τοποθέτησης δεδομένων για τις εξεταζόμενες εφαρμογές.

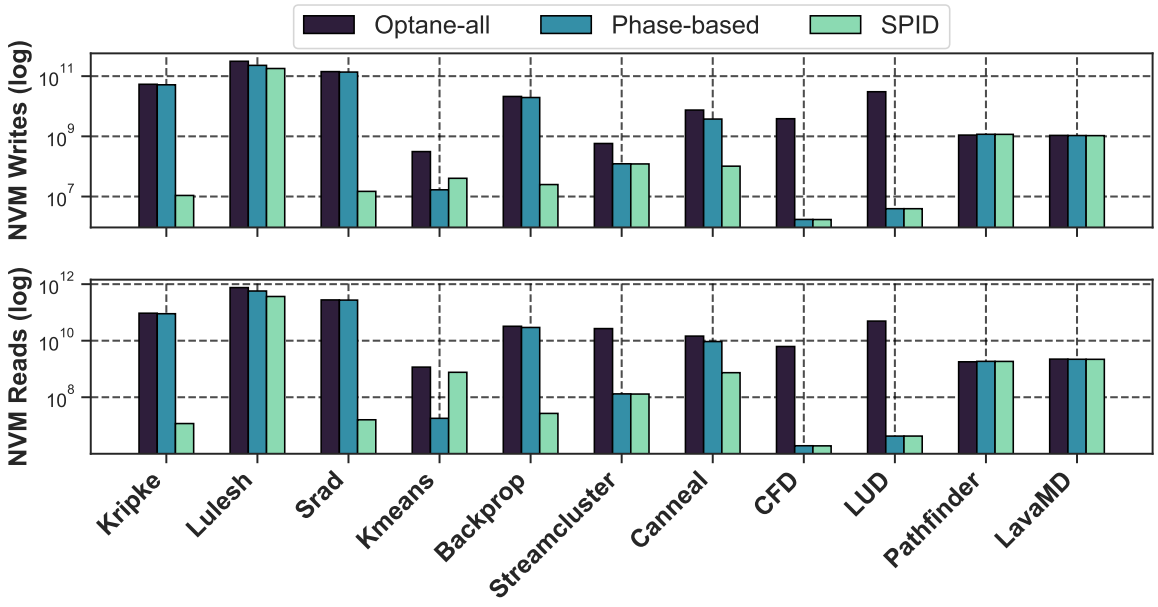

Figure 1.4.2: Αριθμός εγγραφών (πάνω) και αναγνώσεων (κάτω) σε NVM για τα `SPID`, `Optane-all` και `Phase-based` για όλες τις εξεταζόμενες εφαρμογές.

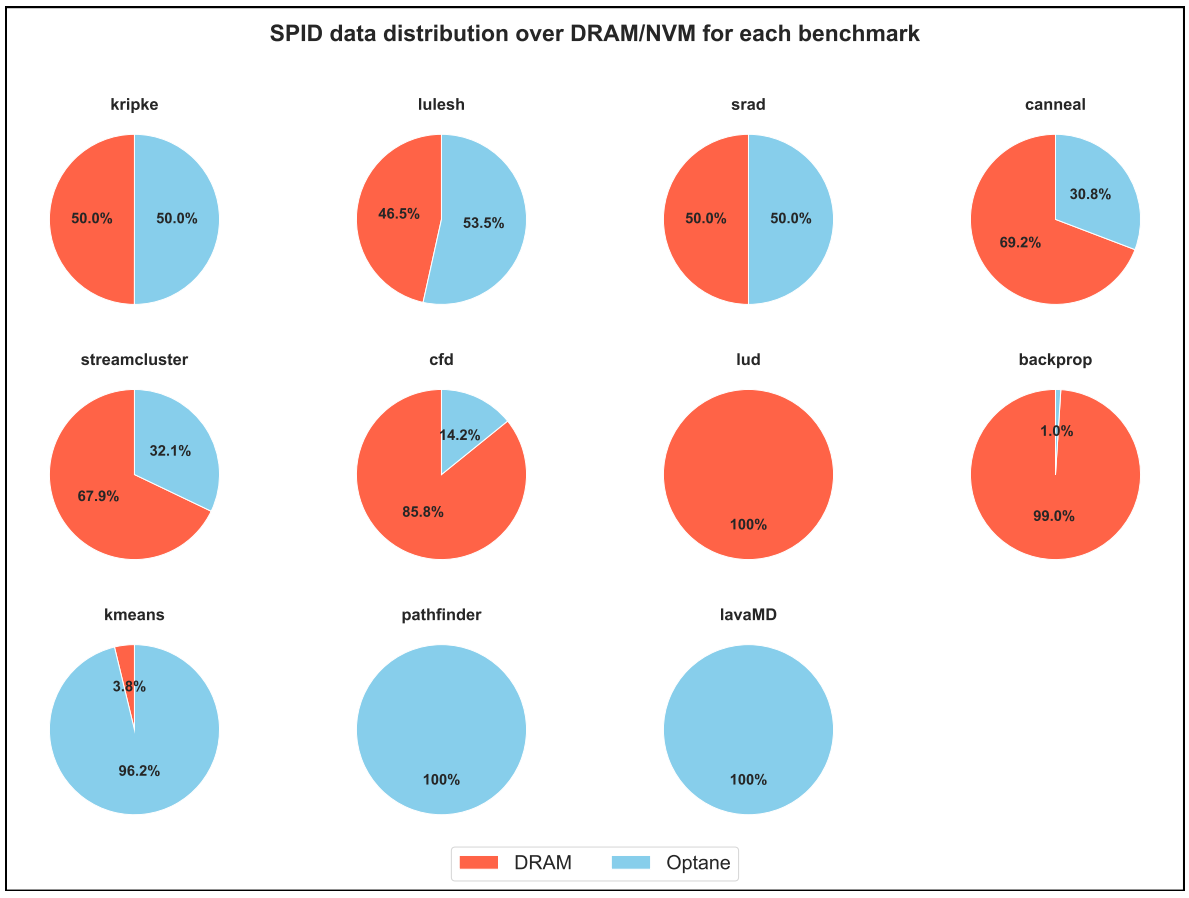

Figure 1.4.3: Κατανομή δεδομένων μέσω του `SPID` σε DRAM/NVM για κάθε benchmark .

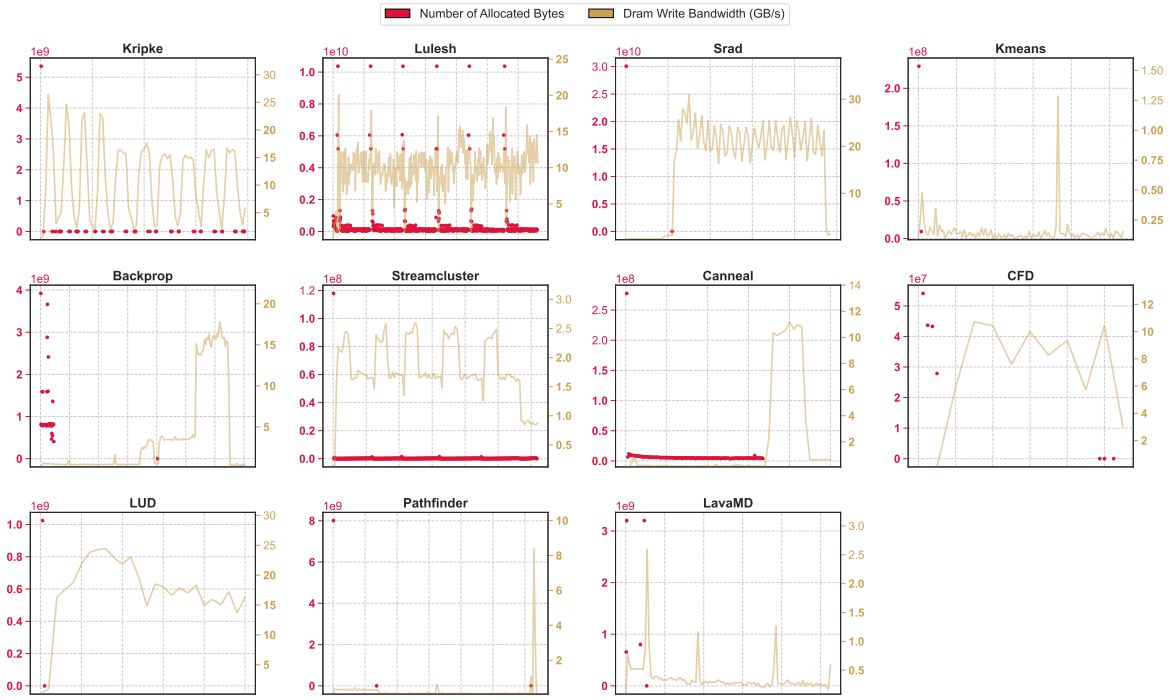

Figure 1.4.4: Συσχέτιση των δεσμευμένων bytes (κόκκινο) και του εύρους ζώνης μνήμης (κίτρινο), συναρτήσει του χρόνου εκτέλεσης, για όλα τα benchmarks .

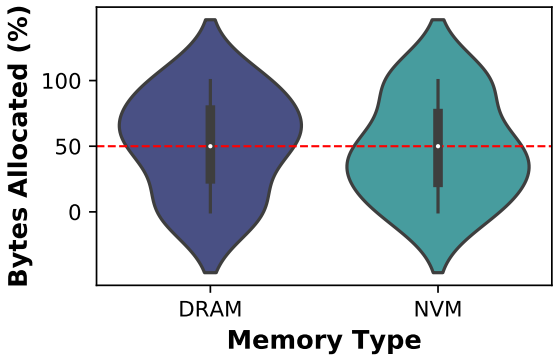

Figure 1.4.5: Κατανομή δεδομένων μέσω του SPID σε DRAM/NVM για το σύνολο των benchmarks .

Chapter 2

Introduction

Over recent years, the growth of emerging memory technologies integrated into modern High Performance Computing (HPC) and Cloud systems has introduced a novel architectural paradigm, known as heterogeneous memory system. To address the scalability and sustainability limitations of traditional DRAM technologies [1], [2], Non-Volatile Memories (NVM), also referred to as Persistent Memories (PM), have gained significant traction. Technologies such as 3D-XPoint [3], Resistive RAM (ReRAM) [4], and Spin Transfer Torque Magnetic RAM (STT-MRAM) [5] serve as key examples of NVMs. These technologies typically offer data persistence, reduced energy consumption, and higher density than DRAM, resulting in increased memory capacities at lower costs [6]. However, NVMs also present challenges, such as higher access latency, lower bandwidth, and limited write endurance. They also exhibit asymmetrical latency and energy consumption between read and write operations [7], leading to trade-offs in the integration of NVM technologies with traditional DRAM.

Modern commercial platforms and vendors, such as Google, Amazon, and Alibaba [8], [9], [10], combine conventional DRAM DIMMs with Intel Optane DC Persistent Memory, which is, up to now, the only commercially available NVM technology. These systems aim to expand memory capacity sustainably and cost-effectively, offering users a diverse range of memory technologies. A variety of application domains are supported on these platforms, including machine learning, deep learning [11], database workloads [12], and scientific/big data applications [13]. These workloads demand high processing throughput with minimal I/O bottlenecks, thus placing considerable stress on the main memory subsystem.

With the rapid evolution of heterogeneous memory systems and increasingly diverse application requirements, both academic and industrial communities are stressing efforts to develop efficient data placement algorithms [14], [15], [16]. Recent advancements in placement policies support data placement at multiple granularity levels, such as page-level [16], [17], [18], [19], object-level [14], [20], [21], [22], and adaptive-level [23]. However, these approaches require extensive profiling techniques, including instrumentation [24], [25] and detailed object-level analysis [21]. Such methods, while precise, are often resource-intensive and incur significant runtime overhead, limiting their scalability and applicability in real, large-scale memory systems. For instance, research presented in [26] indicates that full memory profiling can impose up to 53% performance overhead, where the overhead increases significantly as the size and complexity of the target workload grow.

To overcome these constraints, novel data placement solutions need to optimize profiling processes and minimize run-time overhead while optimizing placement accuracy. In this context, we introduce `SPID`, the first work to propose a lightweight, spike-based dynamic data placement framework specifically designed for heterogeneous DRAM/NVM systems, eliminating the dependency of the efficiency of our solution to detailed profiling and other resource-intensive processes. `SPID` reduces the dependency on exhaustive object-level profiling, and leverages the detection of memory access spikes to identify high-bandwidth and active memory regions, enabling runtime-adaptive data placement. This targeted

approach allows SPID to dynamically adapt data placement while maintaining low profiling overhead, thus providing a practical and scalable solution for real-time applications. **The novel contributions of this work are as follows:**

- **We introduce SPID, a lightweight, scalable spike-based data placement algorithm** for heterogeneous DRAM/NVM systems. SPID leverages advanced spike detection mechanisms, through correlating bandwidth and active objects spikes and provides adaptive decision-making to optimize data placement.
- **We develop and integrate SPMalloc**, an open-source library that intercepts dynamic memory allocation calls, integrates SPID's functionality and enables seamless data placement across heterogeneous memory systems.
- **We conduct a comprehensive experimental evaluation of our framework**, comparing it against baseline algorithms and state-of-the-art data placement strategies. Results demonstrate that SPID achieves an average reduction in execution time by 30.82% and reduces energy consumption by 31.61%.

The remainder of this work is organized as follows: Chapter 3 reviews the related work and highlights the key advancements achieved through this research. Chapter 4 provides the theoretical foundation necessary for understanding the subsequent chapters. Chapter 5 introduces our proposed data placement solution, with Chapters 6 and 7 focusing on the technical implementation of the solution and the experimental evaluation, respectively. Finally, Section 8 presents the conclusion of this work.

Chapter 3

Related Work

Over the last years, multiple works have been conducted that focus on data placement over heterogeneous main memory systems. We classify the related literature based on the granularity that the placement is applied on, i.e structure/object-level and page-level data placement.

Structure/Object-level Data Placement: Authors of [14] propose kernel-level object tiering and migration for heterogeneous memory systems. In [20], authors introduce a profiler that provides guidance for data placement in heterogeneous memory, while in [21], authors propose dynamic data type refinement to optimize placement. On the same level, in [22], a write-aware data structure placement for cached and uncached NVMs is proposed. Authors of [23] propose a runtime framework for adaptive granularity data placement for graph-based applications, while in [27] the authors perform data placement through considering individual application phases.

Page-level Data Placement: Page-level data placement solutions are widely explored to optimize memory usage and performance in heterogeneous memory systems. Authors of [17] propose an LSTM-based page placement algorithm that predicts future access patterns to place pages effectively in memory tiers. In [16], the authors introduce a reinforcement learning approach for page placement, while in [18], the authors present an online, profile-guided data tiering mechanism designed specifically for page-level data placement in HPC environments. Last, authors of [19] propose an OS-level, application-transparent page placement mechanism that operates without requiring modifications to applications.

While previous research has explored data placement strategies for heterogeneous memory systems across various granularity levels, these approaches typically suffer from two major drawbacks: i) they rely on costly and resource-intensive profiling mechanisms, and ii) they introduce significant execution time overhead due to extensive monitoring and runtime inference requirements. To the best of our knowledge, this is the first work to introduce a lightweight solution for data placement in heterogeneous DRAM/NVM systems, eliminating the dependency of the efficiency of our solution to detailed profiling and other resource-intensive processes. Additionally, in contrast to prior research, this work introduces the first spike-based dynamic data placement framework for heterogeneous DRAM/NVM systems. Our approach leverages the detection of memory access spikes to identify high-bandwidth and active memory regions, enabling runtime-adaptive data placement without relying on expensive profiling or persistent monitoring.

Chapter 4

Theoretical background

This chapter lays the theoretical foundation essential for understanding the remainder of this work. It begins with an in-depth analysis of memory architecture and hierarchy, followed by an introduction to the fundamentals of persistent memory and heterogeneous memory systems. The chapter concludes by addressing the challenges associated with programming persistent memory and exploring the various data placement strategies employed by modern programmers for heterogeneous memory systems.

4.1 Memory Architecture: An Overview

Memory architecture is a critical aspect of computer systems, determining how data is stored, accessed, and managed across various memory levels. It involves organizing memory in a hierarchical structure, designed to balance performance, capacity, cost, and energy efficiency. This structure enables faster access to frequently used data while providing larger storage for less frequently accessed information. The memory hierarchy is essential for optimizing overall system performance, as it mitigates the speed gap between the processor and memory. Understanding the basics of memory hierarchy, its key characteristics, and the challenges faced by modern memory systems is fundamental to improving computing efficiency and scalability. This section explores these areas, providing insights into how memory systems are structured and the ongoing efforts to address their limitations.

4.1.1 Basics of Memory Hierarchy

Authors of [36] describe the fundamental concepts of the memory architecture and hierarchy. In general, a hierarchy is an organizational structure where items are arranged in ranked levels, often based on their importance or priority. In computing, several systems utilize a hierarchical framework. For instance, most file systems organize files within a structured tree-like arrangement, assigning them specific locations.

Similarly, a computer memory hierarchy categorizes memory components based on their access speeds and response times. This structure typically consists of multiple memory levels, each offering varying performance characteristics and speeds of access.

Memory hierarchy is essentially employed to organize memory in such a way that data access time can be minimized, thus improving system performance. In hierarchical memory systems, processor (CPU) registers are at the top of a pyramid-like structure (level 0) while optical disks and tape backup devices are at the bottom (level 4). This system was developed based on a type of program behavior known as "locality of references." This behavior refers to the tendency of programs to access instructions that have addresses or memory locations near one another in order to speed up access and improve performance.

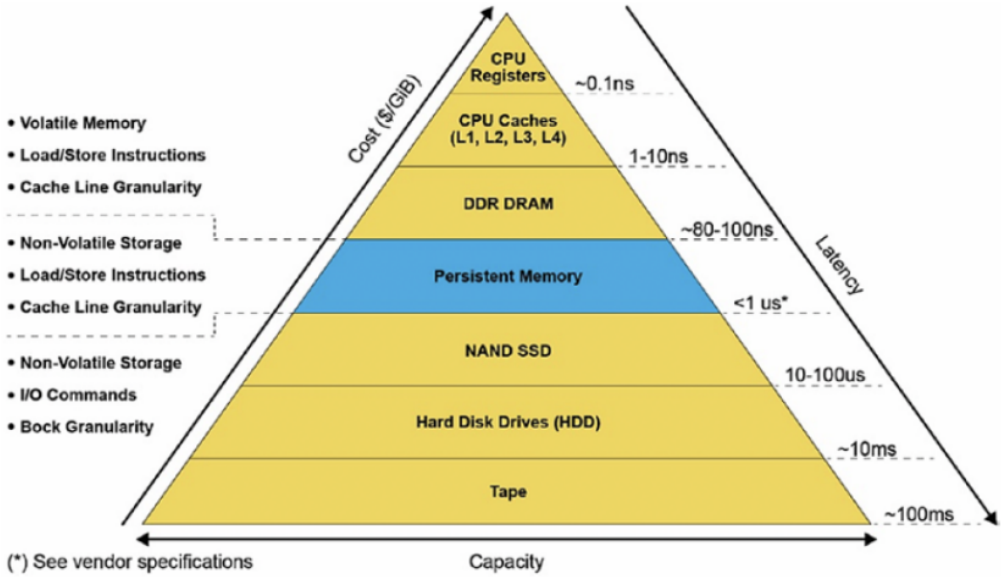

Figure 4.1.1: Memory Pyramid, adapted from [37] .

Fig. 4.1.1 illustrates the hierarchical pyramid of memory, which can be categorized into five levels, based on speed and usage. Those levels are the following :

- **Level 0:** CPU registers
- **Level 1:** Cache memory
- **Level 2:** Primary memory or main memory
- **Level 3:** Secondary memory or magnetic disks or solid-state storage
- **Level 4:** Tertiary memory or optical disks or magnetic tapes

As a general rule, the cost and capacity of memory levels are inversely related to speed. CPU registers are the fastest but smallest and most expensive, while tertiary memory devices are the slowest, largest, and least costly. The hierarchy levels can be further analyzed as such :

Level 0: CPU Registers

CPU registers are small, high-speed memory locations inside the processor used to store data needed for operations. They have the shortest access time and are the most expensive type of memory, typically sized in kilobytes. Implemented using static RAM (SRAM) and digital flip-flops, registers include components like the program counter, status word register, and accumulator to handle data storage and calculations efficiently.

Level 1: Cache Memory

Cache memory stores frequently accessed program code or data for rapid retrieval by the CPU. When the processor needs information, it first checks the cache; if the data is not found, it accesses the main memory. Cache memory is typically smaller than CPU registers, measured in megabytes (MB), and is implemented using static RAM (SRAM). It is often integrated into the processor but can also exist as a separate integrated circuit (IC).

Level 2: Primary/Main Memory

Primary memory, often called random access memory (RAM), serves as the main storage unit for active programs and data. It communicates directly with the CPU and peripheral devices through the I/O processor. RAM is typically implemented using dynamic RAM (DRAM), though main memory may also include read-only memory (ROM).

Inactive programs and data are transferred to auxiliary memory to free up space for currently active tasks. Main memory is less expensive and larger than registers or cache, with capacities typically measured in gigabytes.

Level 3: Secondary Storage

Secondary storage includes devices like magnetic disks, which store programs and data on one or both sides. Multiple disks can be stacked on a spindle to increase storage capacity. In modern systems, magnetic disks are often replaced by faster, non-mechanical solid-state drives (SSDs).

Serving as backup storage, secondary storage is much larger and cheaper than cache or main memory, with capacities often reaching up to 20 terabytes (TB).

Level 4: Tertiary Storage

Tertiary storage, including magnetic tapes and optical disks, is primarily used for long-term data archiving and backup. These auxiliary storage devices store duplicate copies of data or programs not needed for immediate use.

Tertiary storage is the slowest and least expensive memory type, with capacities typically ranging from 1 TB to 20 TB.

4.1.2 Memory Hierarchy Characteristics

The key characteristics of a memory hierarchy include the following:

- **Capacity:** Capacity refers to the amount of information a memory device can store. As we progress down the memory hierarchy pyramid, the storage capacity increases significantly.
- **Access Time:** Access time represents the duration between initiating a read/write request and the moment the data becomes available. Moving from the top to the bottom of the hierarchy, access time increases. CPU registers, located at the top, have the shortest access time and are the fastest. Conversely, storage devices like magnetic tapes at the bottom of the hierarchy have the longest access times.
- **Performance:** The absence of a memory hierarchy creates a speed gap between CPU registers and main memory, leading to increased access times and reduced system performance. By optimizing the memory hierarchy, fewer levels are needed to access and manipulate data, thereby improving overall performance.
- **Cost per Bit:** Cost per bit is determined by dividing the total memory cost by the number of accessed bits. As we move downward in the memory hierarchy, the cost per bit decreases. This trend arises because internal memory, found higher in the hierarchy, is more expensive than external memory.
- **Durability:** Durability refers to the ability of a memory device to withstand wear and maintain functionality over time. For example, DRAM is highly durable, especially for frequent read and write operations. In contrast, technologies like Intel Optane, while offering high performance, are less durable, particularly under heavy write workloads.

These characteristics are critical considerations when designing and implementing modern memory systems. They directly influence system performance, cost-efficiency, and reliability, ensuring that memory configurations meet the demands of increasingly complex and data-intensive applications. By balancing these factors, designers can optimize the overall efficiency and scalability of computing systems.

4.1.3 Challenges in Modern Main Memory Systems

In recent years, DRAM has faced several significant challenges that have hindered overall computer performance. As a key component in computing systems, RAM became a major bottleneck to the growth of performance. However, these challenges have been actively addressed by both research and development departments in technology companies and by computer scientists in research institutes. Their efforts have led to innovative solutions and improvements. Various sources have been compared to ensure the inclusion of the most recent advancements and discussions regarding the challenges faced by main memory systems. Authors of [38], point out some of the most prevalent challenges of modern main memory systems:

Memory Capacity : The doubling of core counts occurs every two years, whereas DRAM capacity doubles every three years, leading to a 30% decrease in memory capacity per core every two years. The demand for memory capacity continues to grow rapidly, driven by data-intensive applications, and this trend is expected to persist. Certain areas of computer science, such as advanced neural network models or driver assistance systems, are particularly memory-intensive. These fields often struggle with limited memory bandwidth, which remains a significant bottleneck in the evolution of hardware performance.

Energy Consumption For a long time, memory systems were designed primarily for performance, with energy consumption being a secondary concern. For example, about 50% of the energy in IBM servers was consumed by the off-chip memory hierarchy. In response, solutions like the CapMaestro project [39] were introduced to improve power management in data centers, allowing for a 50% increase in server capacity under the same power infrastructure.

Currently, DRAM accounts for around 50% of total system power. To address this, low-power DRAM variants, such as DDR3L, LPDDR3, and LPDDR4, have been developed, with LPDDR4 consuming 40% less power than DDR4. Persistent Memory (PMEM), which offers a unique balance of performance and power efficiency compared to traditional DRAM, is also being explored as a potential solution to reduce energy consumption while maintaining high data throughput.

Given the significant power consumption of DRAM, it is crucial to develop new low-power solutions. However, accurate measurement of DRAM's power usage remains challenging due to limited control over DRAM commands in most systems and the lack of specialized monitoring tools for DRAM power consumption.

Scaling The scaling of DRAM faces significant challenges due to its reliance on capacitors to store charge. As DRAM cells shrink, their reliability diminishes, leading to reduced sensing capabilities and increased leakage. Access transistors also need to remain large enough to ensure low leakage and high retention time. While reducing DRAM size can lower costs, it also compromises performance, as smaller capacitors become more prone to noise and failure.

DRAM scaling is reaching its physical limits. While processor transistors have already reached the 3 nm and 2 nm nodes, DRAM is expected to struggle with further miniaturization beyond 10 nm. This limitation impacts the overall performance growth of computers.

To address this, researchers are exploring alternatives like STT-MRAM, ReRAM, and PCM, which offer higher densities, lower power consumption, and increasing scalability. These technologies are advancing rapidly, with STT-MRAM and ReRAM already in production with 12 nm processes. PCM is also actively used by various companies, and its market is projected to grow significantly in the coming years.

4.2 Persistent Memory (PMEM) and the rise of Heterogeneous Memory Systems

4.2.1 Persistent Memory

What Is Persistent Memory?

As mentioned in [40], persistent memory (PMEM) is a type of high-performance solid-state memory that retains data even when the system loses power. Unlike volatile memory, like RAM, which clears its data when powered off, persistent memory keeps the information stored. This capability allows for faster data access and improved system performance when the system is rebooted.

How Does Persistent Memory Work?

Persistent memory (PMEM) is based on non-volatile memory (NVM) components that store data even without power. This data is directly accessible by the central processing unit (CPU), allowing it to bypass the delays of traditional storage devices like hard disk drives (HDDs) or solid-state drives (SSDs). PMEM resides on the memory bus, enabling it to function like regular system memory while retaining data like an SSD.

PMEM fits seamlessly into the memory hierarchy, positioned between volatile memory and storage devices as depicted in Fig. 4.1.1. Technologies like Optane act similarly to RAM but also retain data like an SSD, bridging the gap between fast memory and durable storage.

Figure 4.2.1 illustrates a typical CPU microarchitecture with three cache levels (L1, L2, L3) and a memory controller connected to three memory channels, each with DRAM and persistent memory.

In systems where CPU caches are not power-fail protected, any unflushed data in the CPU caches will be lost during a power failure or crash. However, in systems with power-fail protected caches, modified data is flushed to persistent memory to prevent loss during a crash.

Data moves through the cache hierarchy from L3 (larger, slower) to L2, and then to L1 (fastest, holding data for immediate CPU operations). If the CPU cannot find data in the caches, it accesses it from memory. When data is written, it is first placed in the L1 cache and may be evicted to L2, L3, and eventually written to the memory device via the memory controller.

In systems without persistent memory, data is written to non-volatile storage like SSDs or HDDs, and application software ensures data integrity through flushing operations like msync() or fsync(). In persistent memory systems, the application is responsible for ensuring data integrity directly.

Benefits of Persistent Memory

Integrating persistent memory (PMEM) into enterprise systems offers several key advantages:

- **Enhanced performance:** PMEM provides faster access to data, improving overall system performance.
- **Reduced latency:** By bypassing traditional storage devices, PMEM reduces latency.
- **Versatility:** PMEM offers different operating modes, enabling access to diverse capabilities.
- **Improved scalability:** PMEM enables more memory capacity without significant cost increases.
- **Non-volatility:** It ensures data remains accessible even during power losses, crashes, or shutdowns, providing data persistence and reliability.
- **Better total cost of ownership (TCO):** Larger memory sizes can be achieved without dramatically increasing costs, offering more affordable solutions compared to traditional RAM.
- **Better data security:** PMEM can include encryption add-ons for improved data protection in-memory.

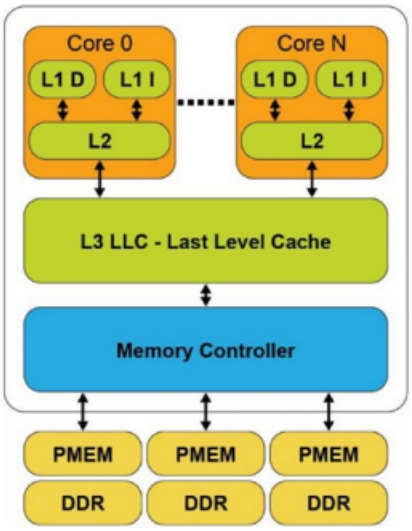

Figure 4.2.1: CPU cache and memory hierarchy. Adapted from [37].

Differences of Persistent Memory when compared against Volatile and Traditional Storage :

DRAM vs. Persistent Memory

Dynamic Random Access Memory (DRAM) is fast but volatile, meaning it loses data when power is lost. While DRAM offers features like buffering, registers, and error correction, it cannot retain data during power failures or crashes. As DRAM constitutes a significant portion of server costs, persistent memory (PMEM) offers a combination of speed, resilience, and non-volatility, making it a strong choice for applications requiring quick data access and retention. Additionally, PMEM tends to be cheaper per gigabyte than DRAM.

SSDs and HDDs vs. Persistent Memory

Traditional storage devices like SSDs and HDDs are durable but have slower data access speeds compared to the main memory. Persistent memory addresses this limitation by offering fast access times while maintaining data persistence. Unlike flash storage, PMEM is directly connected to the memory bus, providing faster access and greater efficiency in enterprise storage systems.

4.2.2 Heterogeneous Memory Systems

Combined with the previous discussion on DRAM and persistent memory technologies like Optane, which have varying latencies and performance characteristics, this highlights the need for heterogeneous memory systems. As highlighted in [41], memory vendors offer different types of memory optimized for specific system requirements. For example, RLDRAM is designed for low access latency but consumes more power, while LPDDR offers lower power consumption but has higher latency and lower bandwidth. HBM, on the other hand, provides higher bandwidth but is designed for bandwidth-sensitive workloads.

However, there is no single memory type that excels in all areas—latency, bandwidth, and power consumption. Homogeneous memory systems, where a single type of memory is used, often fail to meet the demands of diverse workloads. This inefficiency is due to the varying memory access behaviors of different applications, as shown by benchmarks such as SPEC CPU2006. For instance, computation-heavy workloads like gcc may work well with LPDDR for reduced power consumption, while memory-intensive tasks like mcf or milc benefit from memory modules with low latency or high bandwidth.

Thus, heterogeneous memory systems, which combine multiple types of memory, offer the flexibility to optimize performance and energy efficiency by assigning applications to the most suitable memory module. This approach is essential in a landscape where workloads are increasingly diverse in their memory access patterns.

4.3 Intel Optane DC Memory

4.3.1 Intel Optane DC Persistent Memory Module

Izraelevitz et al. [42] provide a thorough explanation of the basic performance and functionality of the Intel Optane DC Persistent Memory Module. The Intel Optane DC Persistent Memory Module (Optane DC PMM) is the first commercially available NVDIMM that introduces a new memory tier between volatile DRAM and block-based storage. Unlike traditional storage devices, such as SSDs, which connect through interfaces like PCIe, the Optane DC PMM leverages a byte-addressable memory interface, delivering superior performance. While similar to DRAM in many respects, it offers higher density and data persistence. Upon its introduction, the Optane DC PMM is offered in three capacities: 128 GB, 256 GB, and 512 GB, providing an innovative solution for workloads that require both speed and data retention.

The Intel Optane DC Persistent Memory Module (Optane DC PMM) operates similarly to traditional DRAM DIMMs by connecting directly to the memory bus and interfacing with the integrated memory controller (iMC) on the CPU. It is designed to work with Intel's second-generation Xeon Scalable processors (Cascade Lake), where each CPU supports two iMCs, and each iMC can manage three memory channels. This allows a CPU socket to support up to six Optane DC PMMs, reaching a maximum of 6 TB of memory.

To ensure data persistence, Optane DC PMM integrates with Intel's asynchronous DRAM refresh (ADR) feature, which guarantees that CPU stores reaching the ADR domain will survive a power failure and be flushed to the NVDIMM within a hold-up time of less than 100 µs. The iMC communicates with the Optane DC PMM via a DDR-T interface, which shares the mechanical and electrical interface with DDR4 but uses a distinct protocol for variable latencies. The Optane DC PMM uses a 72-bit data bus, similar to DDR4, and transfers data in 64-byte cache-line granularity.

Upon receiving memory access requests, the on-DIMM controller handles address translation and access to the Optane DC media. Like SSDs, the controller manages wear leveling and bad-block management through an address indirection table (AIT), which resides in the Optane DC media, while a copy of the AIT is stored in on-DIMM DRAM. The memory access granularity is 256 bytes, leading to write amplification as smaller CPU stores are handled as read-modify-write operations.

In terms of power efficiency, Optane DC PMMs consume less power than DRAM when idle since they do not require constant refreshing. The modules feature two configurable power budgets: an average power budget for sustained workloads and a peak power budget for burst traffic, both of which can be customized by the user.

Intel Optane technology is built on 3D-XPoint, a groundbreaking non-volatile memory technology co-developed by Intel and Micron. Unlike NAND flash, 3D XPoint features a unique cross-point architecture that allows for faster, low-latency data access and higher endurance. It combines the speed of DRAM with the persistence of storage, making it ideal for applications requiring rapid, non-volatile memory. With its 3D design and resistive memory mechanism, 3D XPoint offers improved density and durability, positioning Optane as a revolutionary solution for both memory and storage needs.

4.3.2 Operation Modes

Memory Mode, App Direct Mode, and a sliding scale of allocations in between are among the operating modes in which the PMMs can be further adjusted after being placed on a server. An overview of those modes is illustrated at Fig. 4.3.1 and can be further analyzed as such:

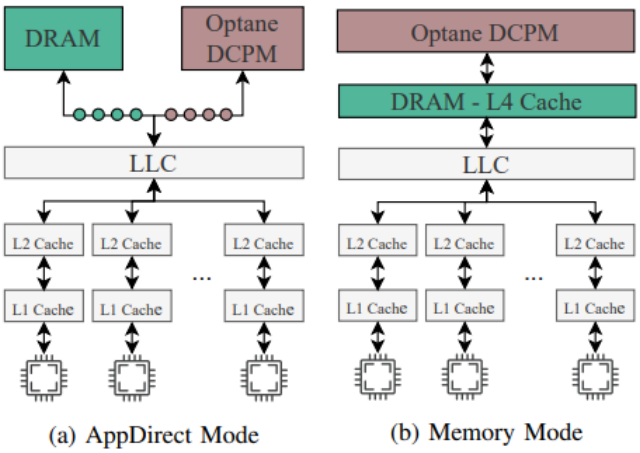

Figure 4.3.1: Overview of Intel Optane DCPM operating modes as adapted from [43].

Memory Mode for Optane DC Persistent Memory PMMs

The Optane PMMs function similarly to DRAM in memory mode. The persistent memory keeps the data "volatile," but the volatile key is cleared each power cycle, thus no special software or application modifications are required. The host memory controller controls the persistent memory, which is utilized in memory mode as an extension of DRAM. Persistent memory to DRAM ratios are not fixed; instead, they can vary depending on the requirements of the application. Naturally, everything that hits the DRAM cache (near memory) will have a latency profile of less than 100 nanoseconds. The persistent memory, also known as distant memory, will receive any cache misses and provide latency in the sub-microsecond range.

App Direct Mode for Optane DC Persistent Memory

App Direct mode is another feature of Optane DC persistent memory. Certain persistent memory-aware programs are required for this mode. This option exploits the persistent behavior of this memory type while maintaining memory-like byte addressability. Persistent memory maintains cache coherence and provides DMA and RDMA functionality in App Direct mode. Additionally, persistent memory can be set up as storage via App Direct. Here, persistent memory uses conventional read/write instructions and functions in blocks, just like SSDs. This is compatible with current file systems, provides block-level atomicity, and allows for block size customization (4K, 512B). Users only require an NVDIMM driver in order to use the storage over app direct. Compared to conventional enterprise class SSDs, this mode offers faster performance, lower latency, and greater endurance, as well as capacity scaling.

4.3.3 Intel Optane Characteristics and Basic Performance

Latency, Write Endurance and Power Consumption on Optane and various memory Technologies

Takahiro et al. [44] report that DRAM exhibits average read and write latencies of approximately 95 ns, with minimal differences between the two (1–2 ns). In contrast, Intel Optane DC Persistent Memory Modules (DCPMM) demonstrate significantly higher latencies, with a read latency of up to 374.1 ns and a write latency of 391.2 ns, marking a 400.1% and 407.1% increase over DRAM, respectively. Additionally, memory accesses involving write-backs in DCPMM show a latency of 458.4 ns, which is 16.1% higher than read-only accesses. These results underscore the latency disparity between DRAM and DCPMM, highlighting the latter's slower performance in both read and write operations.

Liu et al. [45] report on the latency, write endurance, and power consumption characteristics of various memory technologies, including Flash SSD, DRAM, PCM, STT-RAM, ReRAM, and Intel Optane DC

Memory Technology	Read Latency (ns)	Write Latency (ns)	Write Endurance (times)	Standby Power
Flash SSD	25,000	200,000	10^5	zero
DRAM	80	80	$>10^{16}$	Fresh power
PCM	50-80	150-1000	10^8	zero
STT-RAM	6	13	10^{15}	zero
ReRAM	10	50	10^{11}	zero
Intel Optane DCPMM	169 (Sequential), 305 (Random)	90	10^8	zero

Figure 4.3.2: Latency, Write endurance and Power Consumption of memories including Flash SSD, DRAM, PCM, STT-RAM, ReRAM, and Intel Optane DC Persistent Memory [45]

Persistent Memory Modules (DCPMM). They note that while non-volatile memory modules (NVMM) offer benefits such as higher density and lower energy usage, they face challenges like 6–30× higher write latencies and 5–10× greater write power consumption compared to DRAM. Additionally, NVMM has significantly lower write endurance (10^8 cycles) compared to DRAM's 10^{16} cycles, making NVMM unsuitable as a direct DRAM replacement.

Instead, Liu et al. suggest hybrid memory architectures that combine DRAM and NVMM to leverage the strengths of both technologies. However, the integration of NVMM introduces challenges in performance optimization, energy efficiency, cost, wear leveling, and data persistence, requiring advances in memory hierarchy design, management, and allocation schemes.

The study also highlights that the behavior of Intel Optane DCPMM differs from prior expectations based on simulations. Optane DCPMM exhibits 2–3× higher read latency than DRAM but lower write latency. These findings emphasize the need to revisit and optimize existing persistent memory designs for real-world NVMM devices. Fig. 4.3.2 provides their results on the latencies, write endurance and power consumption of the aforementioned memory types.

Bandwidth

Izraelevitz et al. [42] investigate the maximum read and write bandwidth of memory devices using the MLC tool [46] to generate sequential read and write operations across varying thread counts. By gradually increasing the number of threads, they identify the saturation point for memory bandwidth, using up to 23 threads to minimize CPU contention.

Their findings reveal that for read operations, PM-Optane scales with increasing thread count but at a slower rate compared to PM-LDRAM. For non-cached writes, PM-Optane reaches its peak bandwidth with just four threads, after which scalability halts. In contrast, PM-LDRAM continues to scale more effectively.

The experiments show that six interleaved Optane DC PMMs achieve a maximum read bandwidth of 39.4 GB/sec and a maximum write bandwidth of 13.9 GB/sec. Reads scale effectively with thread count, reaching peak throughput at 17 threads, while writes saturate at just four threads. These results highlight the asymmetric scalability of Optane DC memory.

The results of their experiments are diplayed in Fig. 4.3.3. The graph illustrates memory performance as the number of threads increases for read operations (left) and write operations (right). Optane DC demonstrates good scalability for read operations as thread count increases, while write bandwidth reaches saturation with just four threads. Remote DRAM exhibits a distinct access pattern, peaking at approximately 35 GB/sec due to bus bandwidth limitations.

Benefits of Intel Optane DC Persistent Memory

As highlighted by [47], Intel Optane offers numerous benefits. For end users, Intel Optane DC persistent memory modules provide a multitude of advantages. To begin with, the modules provide a far more economical means of scaling a server's DRAM footprint. The total TCO of an organization's server investment is improved because persistent memory can be meshing with the DRAM layer, which causes the effective usable DRAM footprint to scale more quickly with persistent memory. Additionally, some

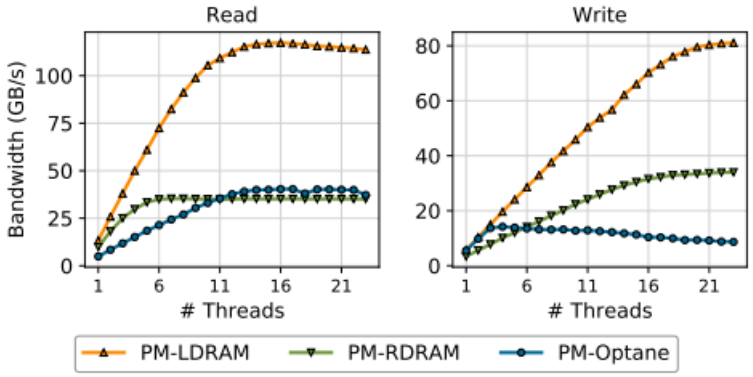

Figure 4.3.3: Sequential memory bandwidth with different number of threads [42].

servers might be able to take advantage of new chances to combine workloads since they can process more data faster. When it comes to value, a second point can be made: persistent memory modules are more affordable than DRAM. Organizations can choose to build their servers with less DRAM but more Optane DC persistent memory to maintain a reasonable, or larger, memory footprint for workloads that might not require as much of the nanosecond latency that DRAM offers. This is because Optane DC persistent memory modules are more affordable than DRAM.

As the name makes clear, the persistent memory modules are just that—persistent. This results in quicker server reboots since the PMMs don't need to be updated with new data. When it comes to memory resident databases, this is quite crucial. Restoring all of the in-memory data after a server reboot can take a very long time. Persistent memory has shown to be quite beneficial for independent software vendors (ISVs) that specialize in high-performance databases in these situations where speedy operation is crucial. Intel has really provided evidence to support this claim. They discovered that a columnar store reloading the entire 1.3TB dataset into DRAM took 20 minutes on a DRAM-only server. An entire system restart in that server before persistent memory was 32 minutes; 12 minutes for the OS, 20 minutes for the data. It took 13.5 minutes on the same server with Optane DC persistent memory. Although it appears remarkable at first glance, the fact that the data component lasted only one and a half minutes—a 13X gain—makes it much more astounding.

The first hardware-encrypted memory was Intel Optane DC persistent memory modules, which also provide on-module encryption. The modules employ a 256-bit AES-XTP encryption engine for data at rest security. The encryption key is lost in Memory Mode and is generated every boot if the DRAM cache loses its contents. In App Direct Mode, a key that is only accessible by the Intel Optane DC controller and is kept in a security metadata region on the module is used to encrypt persistent media. A passphrase is required to unlock the Intel Optane DC persistent memory, which is locked during a power outage. For safe repurposing or disposal at the end of life, the modules also offer DIMM overwrite and secure cryptographic erase. Finally, revision control mechanisms are available and signed firmware versions are permitted.

4.4 Programming Persistent Memory and PMDK

4.4.1 What's different?

The core concept of programming persistent memory is understanding that it combines characteristics of both memory and storage. Unlike traditional memory, persistent memory retains its data after crashes or power failures, making consistency across application runs crucial. Developers must address challenges like recovery strategies and maintaining consistent data structures, similar to storage.

Existing storage techniques, such as write-ahead logging, can be applied to persistent memory, but they

may not fully exploit its advantages. Persistent memory allows direct, byte-granular access through memory mapping, eliminating the need for traditional block-based storage APIs and enabling higher performance and simpler data paths.

4.4.2 Common Challenges when Programming Pmem

Authors of [37], [48] and [49], point out the core challenges presented when programming PMEM:

Consistency :

Consistency ensures that a transaction transitions a data structure only between valid states. In persistent memory, thread-safe updates typically rely on locking, which naturally aligns with consistency points. Locks prevent threads from observing intermediate or invalid states, and releasing a lock signals that the data structure is in a consistent state that other threads can safely access.

Start-Time Responsibilities :

Start-time responsibilities in persistent memory programming involve tasks like detecting platform details, available instructions, and media failures. Unlike traditional storage, where such checks are handled in the storage stack by the operating system, persistent memory allows direct access, removing the kernel from the data path once the memory is mapped. Skipping these initial checks can lead to significant issues, such as unnecessary cache flushing on hardware that doesn't require it, reducing performance, or ignoring media errors, which could result in the use of corrupted data and unpredictable behavior. Ensuring these responsibilities are addressed is essential for production-quality programming.

Atomicity :

Atomicity in persistent memory ensures that a set of operations either completes fully or not at all, even during system failures. Common methods include redo logging (logging full changes for forward recovery), undo logging (logging information to reverse partial changes), and atomic pointer updates (safely swapping pointers to old and new data). A significant challenge arises when transactions involve memory allocation and deallocation, such as adding nodes to data structures. If a transaction is interrupted, memory leaks or inconsistent states can occur unless all allocations and deallocations are managed as part of the atomic operation, adding complexity to implementation.

Crash Consistency :

Crash consistency [48], in persistent memory (PM) is a critical challenge due to its non-volatile nature. The issue arises because the maximum atomic CPU write size is limited to 8 bytes. Writing data larger than this can result in partial updates and inconsistencies during system crashes. Furthermore, the order in which data is persisted in write-back caches often differs from the store instruction order, necessitating the use of memory barriers and cache line flushes to maintain consistency. CPUs now provide specific instructions (e.g., clflush, clflushopt, clwb, sfence, etc.) to address these issues, and techniques like logging or copy-on-write (CoW) can ensure consistency for larger data. However, these methods introduce significant overhead from additional writes, impacting performance. Tools like PMDK help manage these challenges by implementing persistence instructions efficiently.

Write Amplification :

Write amplification [49], refers to the phenomenon where the amount of data written to storage exceeds the amount of actual data being modified, which can significantly reduce the lifespan and performance of storage devices. In the context of non-volatile memories (NVMs), reducing write amplification is crucial for enhancing the endurance and efficiency of NVM-based systems, as frequent writes can degrade the memory over time. Kargar et al. address write amplification by redesigning existing data structures and systems, rather than building new ones from scratch. They focus on structures like LSM-trees, B+-Trees, and hash-based indexing, tailoring them to reduce unnecessary writes. Techniques such as separating keys and values in LSM-trees, using XOR linked lists, and placing mutable data in

DRAM and immutable data in NVM are proposed. These methods aim to optimize write operations, increase NVM lifespan, and enhance overall system performance.

4.4.3 Persistent Memory Development Kit

The unique characteristics of persistent memory make it a complex technology to work with, as highlighted in previous chapters. Anyone who has worked with persistent memory software can attest to the challenges it presents. To simplify this, Intel developed the Persistent Memory Development Kit (PMDK). This toolkit was designed to be the go-to library for persistent memory, offering solutions to the common difficulties encountered in persistent memory programming.

The PMDK [50], has grown into a comprehensive collection of open-source libraries and tools designed to simplify the management and access of persistent memory devices for both application developers and system administrators. Developed alongside the evolving operating system support for persistent memory, these libraries fully leverage the features exposed by the OS interfaces.

Building on the SNIA NVM programming model [51], the PMDK libraries vary in complexity, with some offering simple wrappers around OS primitives for ease of use, while others provide sophisticated data structures and algorithms for persistent memory. This diversity means that you, as a developer, need to choose the right level of abstraction for your specific needs.

Although created by Intel to support its hardware, the PMDK is designed to be vendor- and platform-neutral, meaning it is not restricted to Intel processors or persistent memory devices. It can also be used on any platform that exposes the necessary operating system interfaces, including Linux and Windows. Intel encourages contributions from individuals, hardware vendors, and ISVs to enhance the PMDK.

The PMDK is licensed under the BSD 3-Clause License, allowing developers to incorporate it into both open-source and proprietary software. This flexibility means you can integrate only the components of PMDK that meet your requirements.

As of the time of writing, the following libraries are currently part of the PMDK:

- **libpmemobj**: This library provides a transactional object store, supporting memory allocation, transactions, and general persistent memory programming features. It's ideal for developers new to persistent memory.
- **libpmem**: Offering low-level persistent memory support, this library serves as the foundation for the other libraries. It's particularly useful for developers creating custom persistent memory algorithms, though most developers will likely use libpmemobj, which calls libpmem internally.
- **libpmem2**: A newer version of libpmem, libpmem2 offers a more universal, platform-agnostic interface. Like libpmem, it provides low-level persistent memory support and is useful for custom algorithm development, but most developers will use libpmemobj for its higher-level memory allocation and transaction support.
- **libpmempool**: This library supports offline pool management and diagnostics for persistent memory.
- **pmempool**: A tool for managing persistent memory pool files created by the PMDK libraries. It is useful for both system administrators and software developers for troubleshooting and debugging.
- **daxio**: A utility for performing I/O on Device DAX devices or zeroing out a Device DAX device.
- **pmreorder**: A utility for checking the consistency of a persistent memory program.

4.4.4 Memkind API

The Memkind API [30], a component of the Persistent Memory Development Kit (PMDK), is a versatile and easy-to-use memory allocator built on top of the widely used jemalloc. It is designed to enhance memory management by reducing fragmentation and supporting scalable concurrency. Memkind enables efficient utilization of diverse types of memory in modern systems, including DRAM, NVDIMM, and High-Bandwidth Memory (HBM), by extending the standard malloc API with an additional argument that specifies the desired memory type. One of its key features is a transparent mode, which facilitates memory tiering across different memory types without requiring modifications to existing applications. Additionally, Memkind provides flexibility by supporting file-backed memory allocations on specified devices or user-defined memory regions. These capabilities make it a powerful tool for optimizing application performance on systems with hybrid memory architectures, particularly those leveraging persistent memory

4.5 Placement

With the advent of heterogeneous memory systems, the development of effective data placement methods has become a critical area of research. These systems combine multiple memory types, such as high-speed DRAM and non-volatile memory (NVM), each with distinct performance, durability, and cost characteristics. Efficient data placement is pivotal for optimizing the use of these diverse memory tiers, directly influencing system performance, energy efficiency, and cost-effectiveness.

This section explores the challenges inherent in placing and migrating data across different memory types, the varying granularity levels at which data placement occurs, and the diverse strategies employed by researchers and practitioners to address these challenges.

Among the many tasks faced by developers of heterogeneous memory systems, data placement stands out as the most complex and critical. It serves as the cornerstone of system design, determining the overall performance and operational efficiency of applications. Poor placement strategies can lead to bottlenecks, excessive latency, and increased energy consumption, while effective approaches can unlock the full potential of heterogeneous memory technologies.

4.5.1 Challenges of Data Placement

Modern programmers face numerous challenges when designing algorithms or methods for data placement in heterogeneous memory systems. These challenges arise due to the complexity of managing two distinct memory types, such as DRAM and persistent memory (PMem), each with unique characteristics. Some of the key challenges are described below:

- **Overheads:**

  Implementing tools or algorithms to make data placement decisions often introduces significant overhead. This occurs because the placement algorithms must interact with the allocation procedures of the target applications, adding additional complexity to the memory system, which is already strained by application execution. Moreover, data migration decisions—moving data between memory tiers to optimize placement—consume even more memory bandwidth and processing power, exacerbating the overhead. These migration operations, if not carefully managed, can lead to performance bottlenecks and increased latency, undermining the benefits of heterogeneous memory systems.

- **Access Pattern Variability:**

  Data placement algorithms must account for the dynamic nature of application memory access patterns. Even with prior profiling information about data behavior, these algorithms need to predict, at runtime, whether upcoming data should be placed in DRAM (for fast access) or PMem (for larger capacity and persistence). This variability creates a significant challenge because applications often exhibit unpredictable or irregular access patterns. To address this,

programmers increasingly rely on advanced techniques, such as machine learning models, to predict memory access behavior. For example, some researchers employ reinforcement learning or neural networks to classify "hot" versus "cold" data. While these methods can improve accuracy, they require additional expertise in machine learning, as well as computational resources, further complicating implementation.

- **Scalability with System Complexity:**

  As heterogeneous memory systems grow in complexity—integrating multiple memory tiers, distributed architectures, or accelerators—the challenge of scaling data placement algorithms becomes increasingly prominent. A placement strategy that works well for a single-node system may fail to perform effectively in large, distributed setups. For instance, ensuring data locality while minimizing migration costs becomes exponentially harder as the number of nodes or memory types increases. Programmers must design scalable solutions that balance performance across the system without overloading communication links or memory channels.

4.5.2 Granularity of Data Placement

One of the most critical decisions in data placement for heterogeneous memory systems is choosing the granularity level—the detail level at which data is managed and placed. The granularity level directly impacts performance, implementation complexity, and system overhead, requiring programmers to balance trade-offs. Below, we explore the three commonly used granularity levels:

Page-Grained Placement:

Page-Grained Placement operates at the level of operating system pages, typically 4KB in size, offering a practical balance between precision and simplicity. This method is widely adopted because it integrates seamlessly with existing virtual memory systems, as presented in [52], reducing the need for extensive software changes. While it sacrifices the precision of finer-grained methods, it remains effective for many workloads by enabling reasonable performance improvements with moderate complexity. However, its larger granularity can lead to inefficiencies, especially when managing smaller or highly fragmented data structures.

Fine-Grained Placement:

Fine-Grained Placement provides the highest level of precision by managing data at the byte or cache-line level (typically 64–128 bytes). This method enables optimal placement of frequently accessed ("hot") data in faster memory tiers, maximizing performance for latency-sensitive workloads. However, this precision comes at the cost of significantly increased complexity and overhead. Fine-grained placement requires sophisticated tools to monitor memory access patterns and detailed logging, which can strain system resources. While highly efficient for specific applications, its implementation is challenging and may not scale well for larger, more dynamic workloads.

Object-Level Placement:

Object-Level Placement uses application-specific data structures, such as arrays, lists, or objects, as the basis for data placement decisions. This granularity allows for fine-tuned memory optimizations by aligning placement strategies with the application's inherent memory behavior. For example, the Dynamic Data Type Refinement (DDTR) methodology, as authors of [53] present, enables the restructuring and optimization of data objects to improve performance in heterogeneous memory systems. By leveraging techniques like DDTR, object-level placement can exploit detailed knowledge of application data layouts to enhance both performance and energy efficiency. However, this approach often requires extensive modifications to the application's source code and demands a deep understanding of its data access patterns. While it offers significant optimization potential, object-level placement can be labor-intensive and challenging to generalize, making it better suited for workloads with predictable and well-defined memory requirements.

4.5.3 Placement Methods

Last but not least, an essential component of the data placement process is the method used to implement it. Different approaches to data placement offer unique advantages and trade-offs, each managing placement decisions from a distinct perspective. Below, we explore the most commonly used methods:

OS-Driven Placement:

Operating system (OS)-driven placement relies on the OS to dynamically manage data placement across memory tiers. The system monitors metrics such as access frequency, latency requirements, or bandwidth usage and makes automated decisions to move data between tiers. Examples of such policies include Linux NUMA and AutoNUMA [54], which aim to optimize memory locality and access patterns in multi-node environments. These approaches simplify data placement by abstracting decisions away from developers, enabling broad compatibility across applications. However, they often lack the fine-grained control that developers might need for specific workloads. As a result, OS-driven placement is effective for general use cases but can fall short for applications requiring precise memory allocation strategies.

Manual Placement:

Manual placement gives developers direct control over data placement decisions. Programmers design tools or integrate APIs into their applications to dictate where data should reside at runtime. Using frameworks like the Persistent Memory Development Kit (PMDK) or the memkind API, developers can manually allocate memory to different tiers based on workload characteristics. This method allows for fine-grained optimizations, as programmers can tailor placement decisions to the unique requirements of their applications. However, manual placement can be labor-intensive and requires a deep understanding of both the application's behavior and the underlying memory system. While it provides maximum flexibility, it is best suited for expert users or critical applications where performance is paramount.

Hardware-Assisted Placement:

Hardware-assisted placement leverages specialized hardware, such as FPGAs or ASICs, to accelerate and automate placement decisions. By utilizing the parallel processing capabilities of these devices, hardware can dynamically analyze data access patterns and optimize placement in real-time. As pointed out by researchers in [55], modern hardware has the potential to scale placement decision-making as systems grow in complexity, offering a solution to the increasing computational demands of heterogeneous memory management. Hardware-assisted placement is particularly useful in high-performance scenarios where minimizing latency is critical. However, it requires significant initial investment in hardware design and may be less flexible than software-driven approaches.

Hybrid Approaches:

Hybrid methods combine elements of OS-driven, manual, and hardware-assisted placement to balance their strengths and weaknesses. By integrating automated OS-level management with targeted developer interventions and hardware accelerations, hybrid approaches distribute the complexity of profiling and placement decisions. For instance, developers can use OS tools to handle general memory allocations while employing manual or hardware-assisted techniques for specific performance-critical data paths. This flexibility allows hybrid systems to achieve higher performance and adaptability, making them a robust choice for managing diverse and complex workloads.

Chapter 5

Proposed methodology

In this Chapter, we present SPID, a spike-based dynamic data placement solution for heterogeneous DRAM/NVM systems. The fundamental design principles of our methodology are motivated by the observation of high temporal correlation between memory bandwidth utilization and memory allocations for multiple applications. Figure 5.0.1 shows the correlation of memory bandwidth consumption (yellow) and the active allocations (green) through time for *Lulesh*, *Streamcluster*, *Pathfinder* and *Srad* applications, respectively. The left Y-axis illustrates the number of active allocated objects, while the right Y-axis shows the memory write bandwidth throughout the execution of the corresponding application.

The examined applications reveal three distinct patterns of correlation between active allocations and memory bandwidth usage. In the case of *Lulesh* and *Pathfinder*, we observe that regions of high memory bandwidth align closely with spikes in the number of allocated objects, showing a direct correlation between allocation spikes and bandwidth usage. In *Streamcluster*, high-memory bandwidth regions exhibit a slight temporal shift relative to allocation spikes, with multiple allocation peaks leading to corresponding bandwidth peaks after a short delay. Finally, applications such as *Srad* display an extended correlation pattern, where a single allocation spike initiates a prolonged period of high memory bandwidth.

> ⋆ ***Key Outcome 1:*** *Regions of intense dynamic memory allocation correlate temporally with spikes in memory bandwidth usage.*

This insight enables the key design choices for SPID, which is the first work that relies on advanced spike detection mechanisms, through correlating bandwidth and active objects spikes, guiding efficient data placement and resource management within heterogeneous memory systems. Figure 5.0.2 illustrates an overview of the proposed methodology. As input, we provide the target C/C++ application. The proposed methodology consists of three distinct phases, i.e. i) *Profiling & Analysis* (Sec. 5.1), ii) *Quantization & Spike Detection* (Sec. 5.2) and iii) *Correlation, K-Max Selection & Dynamic Data Placement* (Sec. 5.3), which are detailed in the rest of this Section. Table 5.1 summarizes the key system parameters discussed in the rest of this section.

5.1 Profiling & Analysis

The initial phase of SPID is designed to conduct a comprehensive, lightweight profiling of the target application, facilitating a systematic classification of spiking regions described in Sec. 5.2. This profiling begins with system-level monitoring and data collection (1a), where key metrics such as memory bandwidth and allocations are gathered. As shown in Fig. 5.0.1, we perform memory bandwidth profiling by running the application to assess the correlation between memory bandwidth consumption

and memory allocation patterns. In particular, our focus is on write bandwidth, which serves as a key indicator for identifying phase transitions and understanding the dynamic behavior of memory usage during execution.

To overcome the significant overhead typically associated with object-level profiling, our methodology utilizes an efficient strategy based on active object monitoring. This step involves tracking the runtime allocation of bytes and the number of active objects throughout the execution, which allows us to capture the overall memory usage patterns without the need for fine-grained object-level analysis. By combining both memory bandwidth and active object monitoring, we effectively profile the application's memory behavior, enabling us to detect performance bottlenecks. We post-process and analyze the gathered metrics (1b) to identify critical patterns and phase transitions in the next steps. The processed data are propagated to the next step, presented in Sec. 5.2.

5.2 Quantization & Spike Detection

In this step, the primary objective is to detect and classify distinct application phases into two key regions: *Memory Bandwidth Spike Regions*, which exhibit high memory bandwidth utilization, and *Active Object Spike Regions*, characterized by increased object allocations. The classification of these regions is based on two novel detection mechanisms: (i) the *Memory Bandwidth Spike Detector* (2a) and (ii) the *Active Object Spike Detector* (2b), both of which are described in detail in the rest of this section.

Memory Bandwidth Spike Detector: To identify distinct application phases, we introduce a quantization algorithm, designed to partition the application into discrete intervals based on memory bandwidth patterns. Our algorithm utilizes a sliding window mechanism to detect high bandwidth phases. A bandwidth spike region corresponds to a time interval $[t_i, t_j]$, where: t_i is the initial timestamp at which DRAM write bandwidth exceeds a predefined threshold relative to the average NVM bandwidth within the current sliding window of constant length s (Eq. 5.2.1) and t_j is the ending timestamp when the DRAM write bandwidth drops below a similar threshold relative to NVM bandwidth (Eq. 5.2.2). Our methodology effectively captures high-intensity memory phases, allowing adaptive data placement and optimized resource utilization in heterogeneous memory environments. Let θ be a threshold factor that defines a spike in DRAM bandwidth relative to the average NVM bandwidth. Then, the start t_i and end t_j times of a bandwidth spike can be defined as follows:

$$t_i = \min\{t : BW_{\text{DRAM}}(t) > \theta \cdot \overline{BW}_{\text{NVM}}(s)\} \tag{5.2.1}$$

$$t_j = \min\{t > t_i : BW_{\text{DRAM}}(t) < \theta \cdot \overline{BW}_{\text{NVM}}(s)\} \tag{5.2.2}$$

Denotation	Description
$BW_{\text{DRAM}}(t)$	DRAM write bandwidth at time t
$\overline{BW}_{\text{NVM}}(s)$	Average NVM write bandwidth over sliding window s
t_i	Start timestamp of a bandwidth spike
t_j	End timestamp of a bandwidth spike
θ	Threshold factor for bandwidth spike
α_i	The number of active objects at time i
$\overline{cd}$	Average consecutive active object difference
sp_i^{i+1}	Binary function that identifies spike regions

Table 5.1: Key parameters for `SPID` formulation

where $BW_{\text{DRAM}}(t)$ is the DRAM write bandwidth at time t, $\overline{BW}_{\text{NVM}}(s)$ is the average NVM write bandwidth over the sliding window s. Thrshold θ is derived through experiments.

Active Object Spike Detector: Similarly, we identify active object spike regions by examining the variation in consecutive values of the number of active objects. We derive discrete values for the amount of active objects, forming a sequence of elements α_i, indicating the run-time number of active allocated objects, as follows: $\{\alpha_0, \alpha_1, \ldots, \alpha_n\}$. Given the array of active object counts, our algorithm calculates the absolute consecutive differences between these values, represented as follows: $\{|a_1 - a_0|, \ldots, |a_n - a_{n-1}|\}$. We derive the average absolute consecutive difference, defined as the critical threshold, $\overline{cd}$ from the α_i values, as illustrated in Eq. 5.2.3. The sp_i^{i+1} is a binary function (Eq. 5.2.4) which identifies spike regions among active objects by comparing each object's absolute consecutive differences, i.e $|\alpha_{i+1} - \alpha_i|$ with the average absolute consecutive difference, $\overline{cd}$. Specifically, the function outputs $sp_i^{i+1} = 1$ when $|\alpha_{i+1} - \alpha_i|$ exceeds the average difference threshold $\overline{cd}$, indicating a spike, and $sp_i^{i+1} = 0$ otherwise. Consecutive detected spikes are merged into larger intervals, thus allowing to capture extended periods of high memory object activity. This classification enables detection of high-activity regions within heterogeneous memory systems, supporting lightweight profiling without extensive object-level analysis.

$$\overline{cd} = \frac{1}{n}\sum_{i=1}^{n}|\alpha_{i+1} - \alpha_i| \tag{5.2.3}$$

$$\text{sp}_i^{i+1} = \begin{cases} 1, & \text{if } |\alpha_{i+1} - \alpha_i| > \overline{cd} \\ 0, & \text{if } |\alpha_{i+1} - \alpha_i| \leq \overline{cd} \end{cases} \quad \forall i \in \{0, .., n-1\} \tag{5.2.4}$$

5.3 Correlation, K-Max Selection & Dynamic Data Placement

After identifying memory bandwidth and active objects spike regions (5.2), we aim to perform their correlation and perform probabilistic mapping of the active object spikes most likely associated with each upcoming bandwidth spike. In order to achieve this, we employ an interval tree [28] to efficiently capture and query active object spikes occurring within the time frame between consecutive bandwidth spike regions (3a). Interval trees enable fast query in $O(logN)$ time, thus making them sufficient for fast traversal and exploration. Each single node in the interval tree corresponds to the time interval $[t_i, t_j]$ of an active object spike. For the exploration of active object spikes within the examined time interval defined by two consecutive bandwidth spikes, the interval tree returns those nodes/intervals that overlap with the latter. Once the relevant spikes are retrieved, our algorithm filters through selecting the *K-max* spikes based on the allocated bytes within each bandwidth spike (3b), which are the most critical for efficient placement.

Finally, dynamic data placement is performed at run-time (3c). The K active object spikes detected earlier are flagged as *hot regions*, while the remaining regions are characterized as *cold regions*. Each region is associated with the total number of bytes allocated within it. To mitigate potential performance degradation caused by the slower access latency of NVM, the *hot regions* are prioritized for placement on DRAM memory. After identifying active object spikes for faster memory placement, the algorithm leverages the deterministic allocation patterns of benchmarks to make runtime placement decisions for the *cold regions*. Each region is managed as a distinct unit, with elements of the corresponding region placed on NVM. *Cold regions* assigned to NVM are susceptible to phenomena, such as write amplification and write throttling [29] due to frequent small object allocations. To address these issues, an alternating placement mechanism is employed, where consecutive *cold regions* are grouped to bigger groups. Finally, the data placement is performed through our integrated library, namely `SPMalloc` (3d), which is detailed in Sec. 6.3.

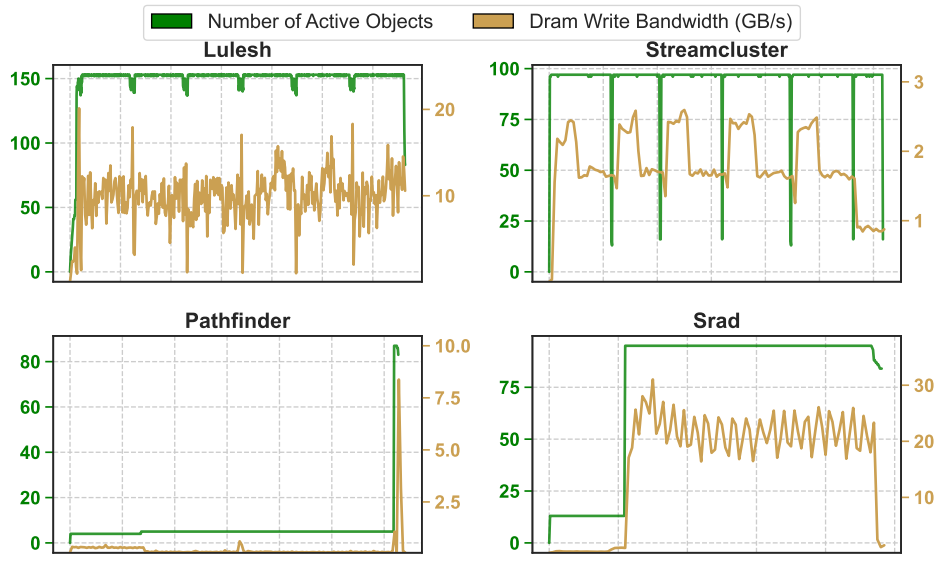

Figure 5.0.1: Correlation of active memory allocated objects (green) and memory bandwidth (yellow) over the execution time, for *Lulesh*, *Streamcluster*, *Pathfinder* and *Srad* benchmarks.

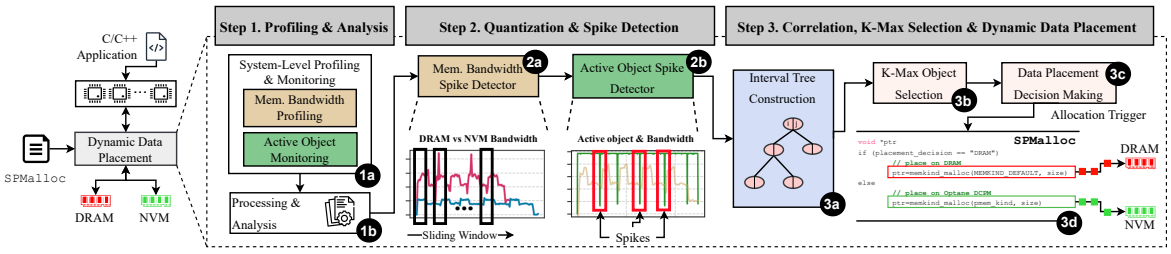

Figure 5.0.2: Overview of `SPID` framework

Chapter 6

Technical Implementation

In this chapter, we present SPMalloc, our custom, open-source library designed to intercept and replace standard memory allocation function calls *(malloc, calloc, realloc, and free)* with custom implementations tailored for heterogeneous memory systems. We begin by discussing the motivation for designing such a library, emphasizing the growing need for precise memory management in modern applications. We then explore the commonly used methods to achieve function interception and compare their advantages and limitations.

Additionally, we detail the numerous challenges encountered during the design and implementation process, such as ensuring compatibility with C++ applications and resolving issues like recursion in function calls. For each challenge, we describe the solutions adopted, showcasing the thought process behind our design choices. The chapter also includes a detailed, code-based analysis of the library, supported by figures illustrating the core implementation, to provide a comprehensive understanding of its functionality and integration.

6.1 Technical Implementation: An Overview

A crucial aspect of our work involves devising an effective strategy for placing data across two distinct memory types: DRAM and Optane. There are various approaches to achieve this goal, each with its own set of challenges and benefits. Previous studies, such as [52], have often relied on simulators to explore methodologies for data placement between memory types. Simulators simplify experimentation by abstracting away hardware complexities and providing greater control over allocation and placement, enabling researchers to test different algorithms and granularities without being constrained by real-world hardware limitations. However, these advantages come at the cost of fidelity, as simulators may not fully capture the intricacies of hardware behavior and performance variability present in physical systems.

The granularity of data placement is another key consideration in the design of such systems. For instance, page-level placement, a widely used approach, involves migrating entire memory pages between memory types. This is often achieved using operating system mechanisms such as NUMA node migration or kernel-level extensions [54]. While this method is relatively straightforward to implement, it lacks the precision required to target specific data structures or objects, potentially leading to suboptimal performance for workloads with fine-grained memory requirements.

In contrast, object-level placement allows for much finer control by enabling the allocation of individual objects to specific memory types. However, implementing object-level placement is significantly more challenging, as it requires custom allocation tools capable of intercepting and managing individual allocation calls. Given that we work on a physical system equipped with Optane DIMMs, and aim to achieve the precision and flexibility beyond what page-level methods provide, we focus on a solution

that operates at the object level. To achieve this, we designed a custom library that intercepts and redirects *malloc*, *calloc*, *realloc*, and *free* calls in C, and *new*/*delete* in C++ to custom allocation routines, enabling us to precisely control the placement of allocation objects in the target application. In this chapter, we discuss the challenges encountered during the implementation of this approach and detail the final solution we developed to address them.

6.2 Memory Allocation Basics and Linking Methods

6.2.1 Memory Allocation Basics

Modern programs predominantly allocate memory using functions such as malloc, calloc, and realloc from the GNU C Library (glibc) for C, and the new operator in C++. These functions manage heap memory, returning a pointer that indicates the memory location allocated for the requested object. This allocated memory remains reserved until explicitly released by calling free in C or delete in C++. The seamless functioning of these allocators is a cornerstone of modern operating systems and runtime environments, making them integral to program execution.

Replacing these standard allocation functions with custom implementations introduces significant challenges. Problems such as memory leaks, segmentation faults, and improper synchronization in multithreaded environments are common pitfalls. Despite these difficulties, there is a growing need to modify or bypass these functions in order to optimize memory utilization, especially in the context of emerging heterogeneous memory systems, which integrate technologies like DRAM and Optane.

Developing a completely new memory allocator tailored for heterogeneous memory systems would be a complex task, requiring extensive effort to ensure optimal performance and reliability. Furthermore, establishing trust in a novel allocator can be difficult without widespread community validation. Instead, a more practical and efficient approach is to create custom versions of the existing allocation functions (malloc, calloc, realloc, free, and their counterparts) that leverage widely accepted tools and APIs.

To achieve this, we will focus on designing a custom library that intercepts calls to these allocation functions. This library will enable precise control over memory placement, allowing objects to be dynamically allocated on either DRAM or Optane, based on the specific requirements of the application.

6.2.2 Linking Methods

Linking [56] is the process of combining one or more object files, produced by the assembler, with necessary libraries to create an executable program. Object files are binary representations of code, where memory locations for functions and variables are either absolute or virtual (relative to a base address). References to external libraries, which are initially unresolved, are addressed during linking by mapping them to their actual memory locations.

Linking can be performed in two ways:

- **Static Linking:** Embeds all necessary library functions directly into the executable.
- **Dynamic Linking:** Resolves library references at runtime, allowing libraries to be shared across programs.

The two linking methods can be further analyzed as follows [57]:

Static Linking: When an executable file is launched, all the necessary contents of the binary are loaded into the process's virtual address space. Many programs also rely on functions from system libraries, which must be included during execution. In the case of static linking, these library functions are embedded directly within the program's executable binary file. As a result, the program is fully self-contained and ready to execute as soon as it is loaded into memory.

Static linking is performed during the compilation phase of the program. It involves combining a collection of relocatable object files and command-line arguments into a fully linked object file, which can then be directly loaded and executed.

A significant drawback of static linking is that each program must include its own copy of the necessary system library functions, leading to redundancy. This approach can be inefficient in terms of physical memory and disk space, as multiple copies of the same library functions are stored and loaded for each program. In contrast, dynamic linking resolves this issue by allowing a single instance of the system library to be loaded into memory and shared among multiple programs.

Dynamic Linking: A dynamically linked program includes a small statically linked function that is executed when the program starts. This function is responsible for mapping the required dynamic libraries into memory and initiating their execution. The dynamic linker identifies the necessary libraries, along with the variables and functions the program requires, by analyzing metadata in the program and the libraries. It then maps these libraries into the program's virtual address space, typically at runtime, and resolves references to the symbols they contain.

The exact memory locations where shared libraries are mapped are not predetermined. To accommodate this, shared libraries are compiled as position-independent code (PIC), allowing them to execute correctly regardless of their memory address.

The core advantage is that Dynamic linking reduces the memory requirements of a program. A dynamically linked library (DLL) is loaded into memory only once and can be shared by multiple applications, saving memory space. Furthermore, dynamic linking lowers application support and maintenance costs by enabling updates to shared libraries without the need to recompile or redistribute dependent programs.

In this work, we use benchmarks from various benchmark suites. Most of these benchmarks compile into a single executable, but the source files that contribute to this executable are numerous and often distributed across multiple directories. Attempting to statically link our custom library to all these source files would require significant modifications to each individual file, assuming we could even identify all of them. This approach would be cumbersome, error-prone, and time-consuming. In contrast, dynamically linking our library to the benchmark is a much simpler and more reliable solution. By leveraging the LD_PRELOAD mechanism [58], we can preload the contents of our library before the main program begins execution, without requiring any modifications to the source files of the benchmark. This allows us to seamlessly intercept memory allocation calls at runtime. As a result, we opted for dynamic linking for our library, marking it as a key design decision in our implementation.

6.3 Creating the Library, Challenges and Solutions

6.3.1 Allocation Functions

Our objective is to create custom implementations of the standard memory allocation functions *(malloc, calloc, realloc, and free)* that **retain their original names but introduce our custom functionality**. Specifically, our custom implementations must enable the placement of allocation objects either in DRAM or Optane, depending on the requirements. For placing objects in Optane, we rely on the Memkind API, which is explicitly analyzed in Chapter 4, using the memkind_malloc() function (or the equivalent functions for calloc and realloc). For DRAM allocations, however, we need a mechanism to invoke the original allocation functions from within our custom implementations. This requires retrieving the original symbols of these functions.

To the best of our knowledge, there are two primary methods to achieve this:

malloc Hooks [59]: A mechanism previously available in glibc to intercept allocation calls. However, this approach has been deprecated in modern versions of glibc, making it unsuitable for our implementation.

dlsym(): A function provided by the dynamic linking library, which allows the retrieval of the original symbols of functions at runtime. By calling dlsym with the appropriate symbol name (e.g., "malloc"), we can bypass our custom functions and access the standard implementation.

While dlsym is a versatile and widely used solution, it introduces a significant challenge in our context. dlsym itself internally calls calloc, which becomes problematic when we intercept and replace calloc with our custom implementation. Specifically, when dlsym is used for the first time to retrieve the original symbol of malloc, it inadvertently triggers our custom calloc implementation, which has not yet been fully initialized. This results in an infinite recursion that causes a segmentation fault. Addressing this issue is a critical part of our implementation strategy.

This challenge led us to make another critical design choice: leveraging the Memkind API not only for Optane memory allocations but also for DRAM allocations. One of Memkind's key features is its ability to allocate memory in DRAM using the memkind_malloc function. By specifying the kind management parameter (the first argument of memkind_malloc) as MEMKIND_DEFAULT, we ensure that memory allocations are directed to the system's main DRAM. This approach provides a seamless and efficient way to handle DRAM allocations without relying on the standard malloc function.

To validate that this method does not introduce any significant overhead compared to the original malloc function, we ran all benchmarks using both allocators — our memkind_malloc-based implementation and the standard malloc. As shown in Fig. 6.5.6, there was no noticeable difference in execution times between the two allocators across all benchmarks. This result eliminates concerns about potential performance penalties associated with using memkind_malloc for DRAM allocations and reinforces the robustness of our approach.

6.3.2 Free Function

Implementing the custom free function presents another significant challenge. When freeing an allocated object, the only available information is its pointer, which indicates the memory location of the object but provides no details about the memory type (DRAM or Optane) from which it should be deallocated. This creates the need for an efficient mechanism to determine the memory type associated with the pointer during deallocation.

One straightforward approach to solve this issue is to allocate an additional byte alongside every memory allocation. This extra byte could serve as a flag, set to 1 for DRAM and 0 for Optane (or vice versa), effectively encoding the memory type with the pointer. During deallocation, the custom free function would read this byte to determine the memory type and then invoke the appropriate deallocation function, such as memkind_free with MEMKIND_DEFAULT for DRAM or the respective kind option for Optane. While simple and conceptually sound, this approach proved to be highly inefficient in practice. Experimental results showed that this method introduced substantial overhead, drastically increasing the execution time of the benchmarks and making it unsuitable for practical use.

To address this issue, we leveraged a key feature of the Memkind API: the *memkind_detect_kind(void *ptr)* function. This function can identify the kind associated with a given pointer, returning MEMKIND_DEFAULT for DRAM allocations or the specific kind used for Optane allocations. By using this function, we simplified the logic of our custom free function, eliminating the need for additional metadata in memory allocations while significantly improving performance and maintainability.

However, the authors of the Memkind API caution that memkind_detect_kind incurs a "non-trivial performance overhead." To assess its impact, we ran all benchmarks with this implementation and carefully measured the performance during memory deallocation. As illustrated in Fig. 6.5.6, our results showed no noticeable performance overhead, even in benchmarks with frequent memory operations. Based on these findings, we chose to implement our custom free function using memkind_detect_kind, as it provides a clean, efficient, and reliable solution for managing memory deallocation across DRAM and Optane.

Fig. 6.5.1 illustrates the implementation of our custom malloc, calloc, realloc, and free functions, as

described earlier. A global variable named dram is used to control the allocation behavior: when dram = 1, allocations are directed to DRAM, and when dram = 0, they are allocated to Optane. This variable provides a simple yet effective mechanism for dynamically specifying the target memory type for allocations within our heterogeneous memory system. To ensure compatibility with C++ applications, our library also incorporates custom implementations of the new and delete operators. Fig. 6.5.2 shows the implementation of these functions, where we simply redirect their functionality to our custom malloc and free equivalents. This approach ensures seamless integration with C++ applications while maintaining the custom memory allocation behavior. The library is compiled into a shared object file (.so), which can then be preloaded into the target executable using LD_PRELOAD. This redirection ensures that all allocation functions—whether in C or C++—are intercepted and routed to our custom implementations.

6.3.3 The constructor and destructor Functions

An important aspect of designing a shared library is the use of *__attribute__((constructor))* and *__attribute__((destructor))* functions. These special functions play a vital role in the lifecycle of the shared library by allowing us to define initialization and finalization routines.

The constructor function is executed immediately before the main() function of the target executable begins execution. This makes it the ideal location to perform any setup required by the shared library. In our implementation, the constructor function initializes Optane memory by creating a file-backed kind of memory using the Memkind API. Additionally, it sets a global variable named "initialized" to 1, signaling that the library is fully initialized and ready to manage memory placement between DRAM and Optane. This flag acts as a safeguard, ensuring that all memory operations proceed only after proper initialization.

Conversely, the destructor function is executed right after the target executable finishes execution, making it essential for cleaning up resources. In our implementation, the destructor function resets the initialized flag to 0, signaling that the program has concluded, and no further memory placements should occur. This step ensures a clean termination of the library's operations, preventing potential memory corruption or resource leakage.

By leveraging the constructor and destructor functions, we ensure seamless integration of the shared library with any target application, providing robust initialization and cleanup mechanisms. Fig. 6.5.3 illustrates the straightforward yet crucial implementation of these functions in C.

6.4 Implementing Object Monitoring Capabilities to Our Library

6.4.1 Monitoring

One of the fundamental design choices in creating a shared library to intercept allocation function calls (malloc, calloc, realloc, and free) is that it grants us complete control and visibility over the memory allocation behavior of the target application. By capturing the call arguments and return values of these functions, we gain access to critical information such as memory pointers, the sizes of allocated objects, and the total number of objects being allocated or freed during the program's execution.

This level of access is invaluable for analyzing the allocation patterns and behavior of the target application. It enables us to identify allocation hotspots and phases, which are key to understanding how memory is utilized. Such insights play a crucial role in optimizing memory placement decisions, particularly in heterogeneous memory systems like DRAM and Optane. By studying these patterns, we can make informed decisions about which data should reside on DRAM for performance and which can be placed on Optane for capacity. This understanding forms the foundation of efficient and intelligent memory management.

To abstract such information effectively, it is essential to adopt a method that tracks allocation behavior

without significantly impacting the program's execution or introducing excessive overhead. Monitoring every individual allocation object, including its size and corresponding pointer, can result in substantial runtime overhead and the generation of enormous log files. This issue becomes particularly pronounced in benchmarks with a high frequency of allocations, making this approach impractical.

To address the challenges of excessive log sizes and runtime overhead, we made a key design choice: instead of logging every individual allocation, we aggregate allocation behavior over user-defined timestamps. Specifically, we periodically log summary information, such as the total allocated bytes and the number of active allocated objects, at intervals defined by the user (0.25 seconds by default). This allows users to make trade-offs between finer-grained monitoring and runtime overhead by adjusting the timestamp duration and logging frequency. Shorter intervals provide more detailed insights, while longer intervals minimize overhead.

The implementation of this monitoring capability requires that we extend the functionality of the allocation and deallocation functions. Each allocation function increments a global counter, *allocated_bytes*, by the size of the allocation and increases one more counter, *active_objects*, by one. Correspondingly, each deallocation function decrements the *active_objects* counter by one. To ensure thread safety, we utilize mutex locks to manage updates to these counters, preventing race conditions in multi-threaded applications.

The actual logging of this aggregated information is handled by a background thread, which operates independently of the allocation and deallocation calls. This thread is triggered at the specified intervals and logs the collected information to a file descriptor or output of the user's choice. By offloading the logging functionality to a separate thread, we minimize interference with the execution of the main program and reduce runtime overhead significantly.

The initialization and finalization of the background thread are seamlessly integrated into the library's *__attribute__((constructor))* and *__attribute__((destructor))* functions, ensuring the monitoring mechanism operates only during the program's execution. Fig. 6.5.4 illustrates the implementation of the background thread, as well as the additional lines of code required to initialize and terminate its functionality.

6.4.2 Placement

Another important role of the background thread is assisting in object placement decisions. As described in Chapter 5, our placement strategy is guided by the output of the *SPID* algorithm, which provides an input array to the library for making placement decisions. This array alternates between bytes to be allocated on Optane (even indices) and bytes to be allocated on DRAM (odd indices). For example, if the input array is [1 kB, 2 kB, 3 kB], the algorithm instructs the placement of the first 1 kB on Optane, the next 2 kB on DRAM, and the final 3 kB on Optane.

The background thread can easily integrate this functionality. Instead of logging allocation data, it monitors the allocation progress against the SPID input array. For each allocation, it tracks whether the specified number of bytes (as defined in the input array) has been allocated to the correct memory type. Once the target number of bytes for a memory type is reached, the thread switches the memory type (via the dram variable) to ensure subsequent allocations follow the instructions of the array.

Fig. 6.5.5 illustrates the modifications to the background thread for integrating this placement logic. The updated thread continuously checks whether the allocated bytes match the specified values for the current memory type, switches memory type when necessary, and ensures proper adherence to the placement strategy.

6.5 Concluding, SPMalloc

We design `SPMalloc`, an open-source library, which replaces the `malloc, calloc, realloc, free` and `new/delete` operations for C and C++ programming languages, respectively. The calls to our library are intercepted at runtime using `LD_PRELOAD`, linking the library to the target executable. Our

library makes use of the *memkind API* [30], which allows developers to place data either on DRAM or Optane DCPM. Figure 6.5.6 illustrates the execution time comparison of the benchmarks presented in Section 7.1 for dynamic allocations on DRAM performed through the default `malloc()` and our `SPMalloc()`. We observe a minimal average performance difference of 1.8%, indicating that memory allocations executed via our API impose negligible runtime overhead. As a result, we establish a unified allocation policy whereby all dynamic allocations are directed through the *memkind API* to facilitate memory tiering and control. Furthermore, `SPMalloc` is enhanced with profiling support to aid in real-time memory management decisions. To mitigate the overhead associated with exhaustive object profiling tools [21], we incorporate a sampling-based profiler within the library. This profiler monitors and logs memory usage by tracking the bytes allocated and the number of active objects throughout the execution. Designed as a background thread, `SPMalloc` enables continuous decision making without intrusively affecting application performance, providing real-time allocation insights and placement for effective memory management.

```c
void *malloc(size_t size)
{
    if(initialized == 0) {
        return memkind_malloc(MEMKIND_DEFAULT, size);
    }
    else{
        if(dram == 1){
            return memkind_malloc(MEMKIND_DEFAULT, size);
        }
        else{
            return memkind_malloc(pmem_kind, size);
        }
    }
}


void *calloc(size_t num, size_t size) {
     if(initialized == 0){
        return memkind_calloc(MEMKIND_DEFAULT, num, size);
     }
    else {
        if(dram == 1){
            return memkind_calloc(MEMKIND_DEFAULT, num, size);
        }
        else{
            return memkind_calloc(pmem_kind, num, size);
        }
    }
}


void *realloc(void *ptr,size_t size)
{
    if(ptr == NULL) return malloc(size);
    memkind_t kind = memkind_detect_kind(pt
    if(kind == MEMKIND_DEFAULT)  return  me                    c(MEMKIND_DEFAULT,ptr,size);
    else  return memkind_realloc(pmem_kind,ptr,size);
}


void free(void *ptr){
    if(ptr == NULL) return;
    memkind_t kind = memkind_detect_kind(ptr);
    if(kind == MEMKIND_DEFAULT)  memkind_free(MEMKIND_DEFAULT,ptr);
    else memkind_free(pmem_kind,ptr);
}
```

Figure 6.5.1: Custom functions for *malloc*, *calloc*, *realloc* and *free*, using the Memkind Api .

```cpp
#include <iostream>

void* operator new(size_t size) {
    void* ptr = malloc(size);
    if (!ptr) {
        throw std::bad_alloc();
    }
    return ptr;
}

void* operator new[](size_t size) {
    void* ptr = malloc(size);
    if (!ptr) {
        throw std::bad_alloc();
    }
    return ptr;
}

void operator delete(void* ptr) noexcept {
    free(ptr);
}

void operator delete[](void* ptr) noexcept {
    free(ptr);
}
```

Figure 6.5.2: Redirecting *new* and *delete* functions of C++ to behave like the custom function we created .

```c
struct memkind *pmem_kind=NULL;
int err=0, initialized = 0, dram = 1;


__attribute__((constructor)) void init_memkind() {
    err = memkind_create_pmem("/mnt/pmem0", 0, &pmem_kind);
    if(err){
        return;
    }
    initialized = 1;
}

static void __attribute__((destructor)) del(void)
{
    initialized = 0;
}
```

Figure 6.5.3: Constructor and Destructor functions that are used to initialize and terminate the functionality of our library when using LD_preload .

```c
int err = 0, initialized = 0, dram = 1;
pthread_t timer_thread;
void *timer_threadproc(void *arg);
int done = 0;
float seconds_elapsed = 0.0;
unsigned long  allocated_objects = 0;
unsigned long  allocated_bytes = 0;
pthread_mutex_t alloc_mutex = PTHREAD_MUTEX_INITIALIZER;


__attribute__((constructor)) void init_memkind() {
    initialized = 1;
    done = 0;
    if (pthread_create(&timer_thread, NULL, &timer_threadproc, NULL) != 0) {
        fprintf(stderr, "Failed to create timer thread\n");
    }
}

__attribute__((destructor)) void del(void) {
    initialized = 0;
    done = 1;
    pthread_join(timer_thread, NULL);
}


void *timer_threadproc(void *arg) {
    while (!done) {
        usleep(250000);
        seconds_elapsed += 0.25;
        pthread_mutex_lock(&alloc_mutex);
        fprintf(stderr, "Allocated: %lu bytes (second %f) (objects alive = %lu)\n", allocated_bytes, seconds_elapsed, allocated_objects);
        allocated_bytes = 0;
        pthread_mutex_unlock(&alloc_mutex);
    }
    return NULL;
}

void *malloc(size_t size) {
    pthread_mutex_lock(&alloc_mutex);
    void *p = memkind_malloc(MEMKIND_DEFAULT, size);
    if (initialized == 1) {
        allocated_bytes += size;
        allocated_objects += 1;
    }
    return p;
    pthread_mutex_unlock(&alloc_mutex);
}
```

Figure 6.5.4: Extending *SPMalloc* to perform monitoring on the Active Objects and Allocated Bytes over the course of the program's execution .

```c
void *timer_threadproc(void *arg) {
    while (!done) {
        usleep(10);
        seconds_elapsed += 0.00001;
        if(i >= switches) {
            done = 1;
            return NULL;
        }
        else {
            if (allocated_bytes >= allocations[i]  && i % 2 == 0){
                pthread_mutex_lock(&alloc_mutex);
                dram = 1;
                fprintf(stderr, "Allocated: %llu bytes on optane (second %f) (objects alive = %llu)\n", allocated_bytes, seconds_elapsed, allocated_objects);
                allocated_bytes = 0;
                i += 1;
                pthread_mutex_unlock(&alloc_mutex);
            }
            else if (allocated_bytes >= allocations[i]  && i % 2 == 1){
                pthread_mutex_lock(&alloc_mutex);
                dram = 0;
                fprintf(stderr, "Allocated: %llu bytes on dram (second %f)  (objects alive = %llu)\n", allocated_bytes, seconds_elapsed, allocated_objects);
                allocated_bytes = 0;
                i += 1;
                pthread_mutex_unlock(&alloc_mutex);
            }
        }
    }
    return NULL;
}
```

Figure 6.5.5: Extending the Functionality of the background thread, to perform placement decision during exeuction .

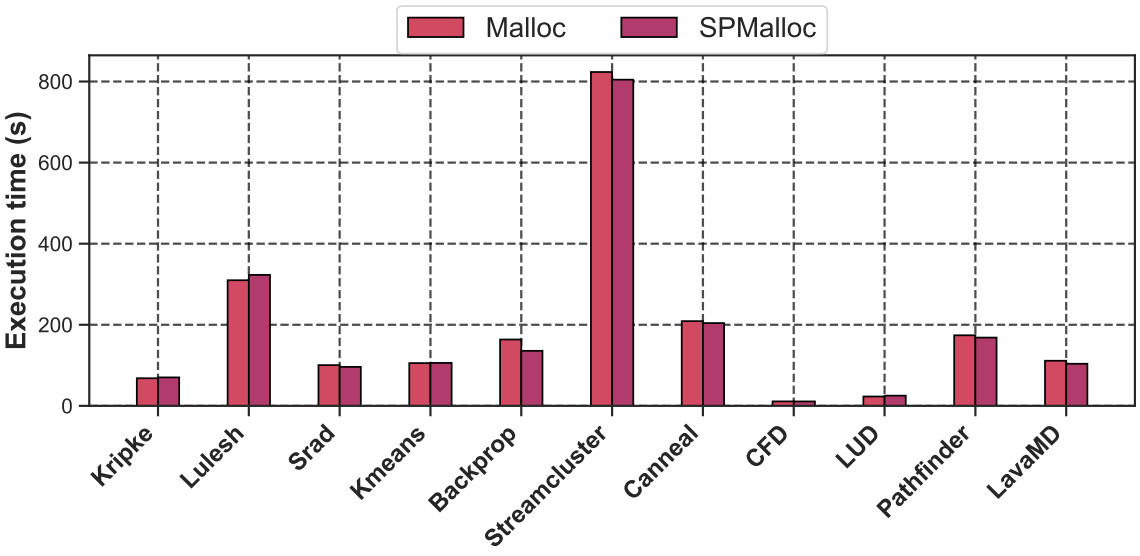

Figure 6.5.6: Comparison of `SPMalloc` and `malloc` overhead.

Chapter 7

Experimental Evaluation

7.1 Experimental Setup

The experiments were conducted on a high-end server consisting of two 20-core Intel Xeon Gold 5218R CPUs @ 2.10 GHz, with 4x32GB DDR4 DIMMs and 6x256GB Optane DC NVDIMMs. Intel Optane DCPM was configured in App-Direct mode with the EXT4-DAX filesystem. Version 1.11 of the Persistent Memory Development Kit (PMDK) and gcc-13.3 were employed for the deployment of the applications. Our proposed solution is evaluated over 11 real-life benchmarks derived from `PARSEC` [31], `Rodinia` [32] and `CORAL-2` [33] suites, covering a set of application domains, e.g. ML workloads, image processing and physics. We focus on memory-intensive applications, namely *Streamcluster, LUD, Kripke, CFD, Backprop, Srad, Lulesh* and on balanced workloads between compute and memory demands, i.e. *Canneal, LavaMD* and *Pathfinder*. Table 7.1 summarizes the benchmarks we experimented with, including their respective suites and core functionalities

We compare our proposed placement policy against 5 placement policies: i) *DRAM-all*, where all data are placed on DRAM, ii) *Optane-all*, where data are placed on NVM, iii) *Round-Robin*, where data are placed in round-robin way on each memory, iv) *Random*, where data are randomly placed and v) *Phase-based* [27], which is a state-of-the-art data placement solution implemented from scratch. For the latter, the discrete phases are split into 5-second chunks. The examined dynamic data placement policies are evaluated based on performance, energy consumption and their impact on the NVM's lifetime, based on the number of write operations performed on the persistent memory DIMMs. The energy consumption and memory bandwidth are measured utilizing the Intel PCM [34], a tool that allows power and bandwidth run-time monitoring over DIMMs through hardware counters, while for measuring the number of read and write accesses, we utilize the `ipmctl` tool [35], which is the standard for configuring, managing and monitoring the NVDIMMs.

7.2 Experimental Evaluation

Performance Analysis: Fig. 7.1.1 illustrates the normalized execution time (top) and the energy consumption (bottom) for all the examined benchmarks, respectively. The execution time results are normalized over the *Optane-all* placement policy. The *DRAM-all* provides the optimal execution time across all benchmarks, due to its optimal latency for both read and write operations, while the *Optane-all* policy exhibits the highest execution time due to significant write access overhead. We observe that `SPID` outperforms the evaluated placement policies, achieving 28.86%, 34.16%, and 29.43% less execution time on average, compared to *Round-Robin*, *Random*, and *Phase-based* solutions, respectively. *Round-Robin* and *Random* placement policies are too simplistic to perform efficiently across a wide range of applications. Furthermore, `SPID` achieves a performance that is only 15.15% slower than the optimal DRAM-only solution on average, demonstrating its effectiveness in utilizing

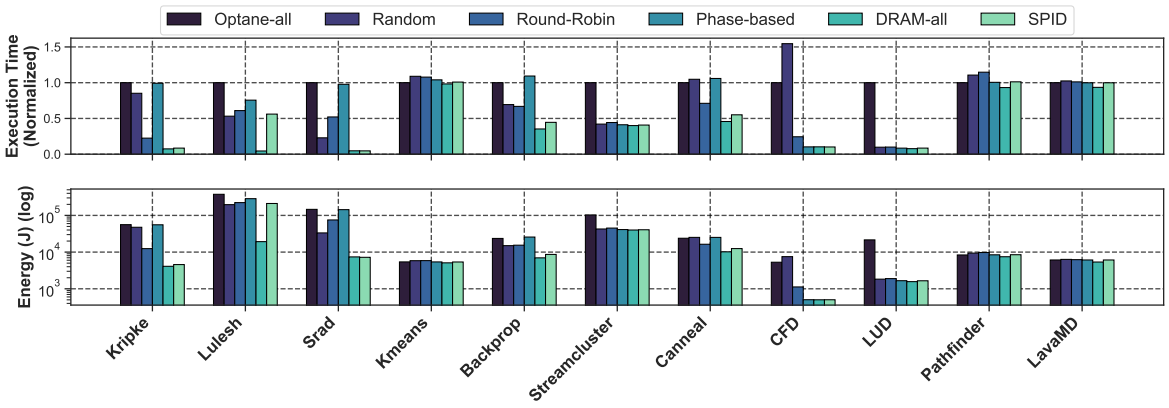

Figure 7.1.1: Execution time (top) and energy consumption (bottom) comparison of the SPID againts the examined data placement policies for the examined applications.

both DRAM and NVM for optimized performance across diverse workloads.

The key advantage of SPID compared to the *Phase-based* is that the latter relies only on bandwidth-defined phases, disregarding allocation behavior within those phases, which is critical for the overall performance, thus leading to performance degradation. SPID overcomes this limitation, through effectively identifying critical allocation regions. This distinction enables SPID to clearly outperform the *Phase-based* approach in benchmarks where critical allocations are not directly adjacent to bandwidth-intensive regions (e.g. *backprop*, *srad*, *Kripke*, *Canneal*), by achieving 3.7× performance speedup on average. Last, for benchmarks that are characterized as non-memory-intensive (e.g. *pathfinder, lavaMD, Kmeans*) all policies demonstrate similar performance, indicating that memory placement strategies have minimal impact in such cases.

> ⋆ ***Key Outcome 2:*** *SPID effectively identifies and prioritizes critical allocation regions by optimizing performance through placements that reflect both bandwidth and allocation dynamics.*

Energy Consumption Analysis: The energy consumption on memory DIMMs for the examined benchmarks is depicted in Fig. 7.1.1 (bottom) in log. scale. The derived results closely align with the execution times, showing that SPID achieves 29.81%, 34.86%, and 30.15% improved energy consumption compared to the *Round-Robin*, *Random*, and *Phase-based* placement policies, respectively. The optimized energy consumption of SPID is primarily attributed to two key factors: (i) write-intensive objects are placed on DRAM, thus avoiding the high energy consumption associated with write operations on NVM, and (ii) non-write-intensive objects are allocated to low-power NVM, which helps in maintaining performance while consuming less power. Additionally, the spike-based dynamic placement strategy used by SPID ensures that memory regions are effectively handled, due to the active object monitoring mechanism, which boosts placement decision making, in contrast to *Phase-based*, where active objects are not considered. This dynamic approach contrasts with traditional static placement methods, which often result in suboptimal energy efficiency due to less intelligent or fixed memory assignments.

> ⋆ ***Key Outcome 3:*** *SPID reduces energy waste due to effective spike and non-spike intensive placement on DRAM and NVM, respectively.*

NVM Accesses & Lifetime Analysis: The effectiveness of a data placement policy should be evaluated on its ability to respect the limited NVM write endurance. Write accesses performed on the NVM act as high-level indicators for the NVM-wear [21]. Therefore, aiming to extend the NVM lifespan, we evaluate the read and write accesses on Optane, focusing particularly on minimizing write operations. Fig. 7.2.1 illustrates the overall number of write (top) and read (bottom) accesses, respec-

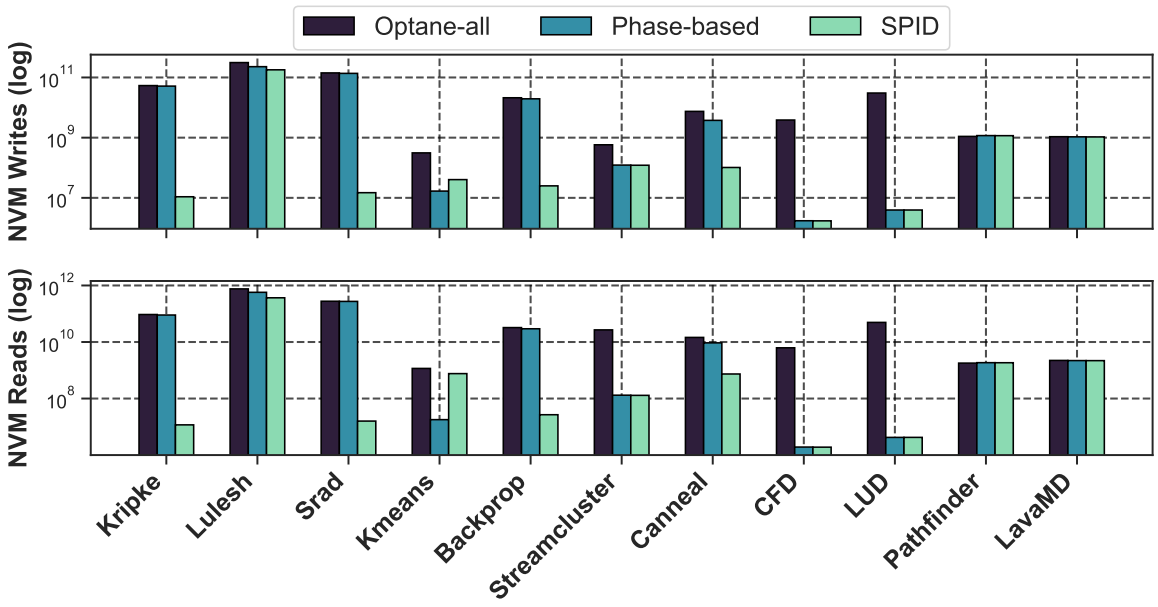

Figure 7.2.1: Number of NVM write (top) and read (bottom) accesses for the `SPID`, `Optane-all` and `Phase-based` for all the examined applications.

tively for the *Optane-all* (baseline), *Phase-based* and `SPID` policies across the examined benchmarks. Our results indicate that `SPID` achieves 73.03% and 33.02% less write accesses on average compared to *Optane-all* and *Phase-based*, respectively. Similarly to the write accesses, `SPID` achieves 70.81% and 30.3% less read accesses on average compared to *Optane-all* and *Phase-based*, respectively. The selective placement of non-write-intensive objects on NVMs and the balanced placement achieved across the DRAM/NVM system leads to reduced frequency of write operations performed on the NVM. As a result, `SPID` not only delivers improved performance over all other policies examined but also minimizes wear, enhancing the overall longevity of NVM more effectively than alternative solutions.

> ⋆ ***Key Outcome 4:*** *`SPID` extends NVM lifespan by significantly reducing write operations, balancing access across DRAM and NVM, thereby enhancing overall endurance and longevity.*

Placement Policy Analysis: Last but not least, in order to gain deeper understanding of the behavior of `SPID`, we examine the distribution of data across the heterogeneous DRAM/NVM system. The pie charts presented in Fig. 7.2.2 illustrate the percentage of bytes allocated to DRAM and NVM during the execution of each benchmark using SPID. These distributions can be categorized into three distinct patterns:

- **Balanced Distribution**: For memory-intensive benchmarks such as Kripke, Lulesh, Srad, Canneal, Streamcluster, and CFD, which also exhibit complex allocation patterns, critical memory decisions are required. SPID achieves a balanced allocation of bytes between DRAM and NVM for these benchmarks, ensuring that write-intensive objects are placed on DRAM, while non-write-intensive objects are allocated to low-power NVM.
- **NVM-Favored Distribution**: For benchmarks with lower memory pressure that primarily rely on CPU utilization, such as kmeans, pathfinder, and lavaMD, SPID predominantly allocates data to NVM. This approach optimizes performance while minimizing energy consumption by leveraging the low-power characteristics of NVM.
- **DRAM-Favored Distribution**: Finally, certain benchmarks, namely Lud and Backprop, see SPID allocate all bytes to DRAM. This behavior is driven by two factors: (i) these benchmarks exhibit memory-intensive phases during execution, leading to bandwidth spikes, and (ii) they

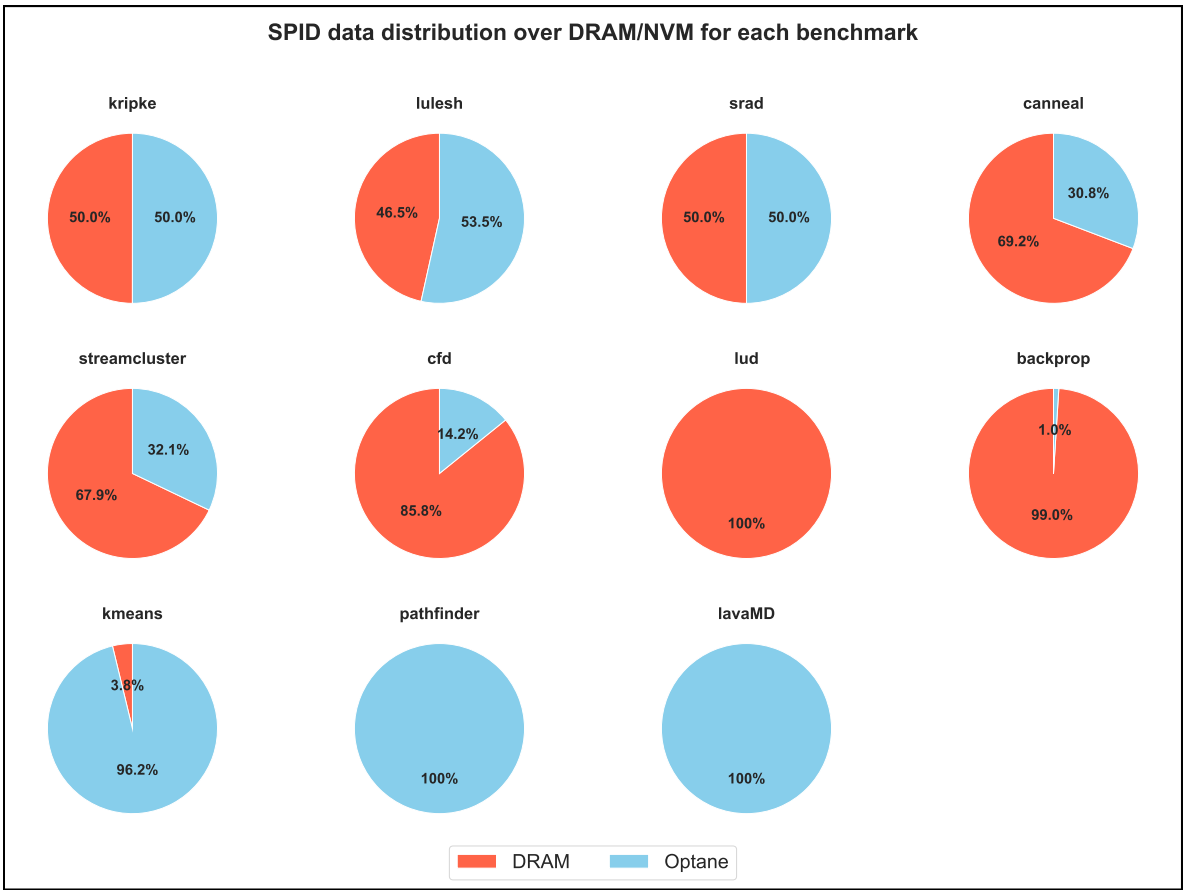

Figure 7.2.2: SPID data distribution over DRAM/NVM

feature an initial allocation spike—often at the program's start—after which no further allocations occur. Consequently, SPID allocates the initial spike to DRAM, resulting in the entirety of the data residing in DRAM.

Fig. 7.2.3 provides additional insight into the aforementioned distributions. Specifically, it illustrates the **non-zero** allocated bytes (red) and the DRAM bandwidth (yellow) throughout the execution of each benchmark. As observed, benchmarks such as Lulesh, Kripke, Streamcluster and Canneal which exhibit complex allocation patterns, achieve an even distribution between DRAM and NVM. Conversely, benchmarks with low memory intensity, such as Kmeans and LavaMD, are exclusively managed using Optane. Finally, memory-intensive benchmarks like Backprop and Lud, which allocate all their memory at the start of execution, are entirely handled by DRAM.

The violin plot shown in Fig. 7.2.4 illustrates the average percentage of bytes allocated to DRAM and NVM for all the evaluated benchmarks. Our analysis reveals that 52.04% of the data is allocated to DRAM and 47.96% to NVM, with the placement percentages show minimal variation and closely align with the median values. This distribution highlights *SPID*'s ability to effectively balance data placement between the two memory types, contributing to balance performance (Fig. 7.1.1 top), energy consumption (Fig. 7.1.1 bottom) and memory accesses (Fig. 7.2.1), as discussed earlier in this section.

⋆ ***Key Outcome 5:*** *The balanced placement of SPID is the key enabling factor for performance, energy and write accesses co-optimization.*

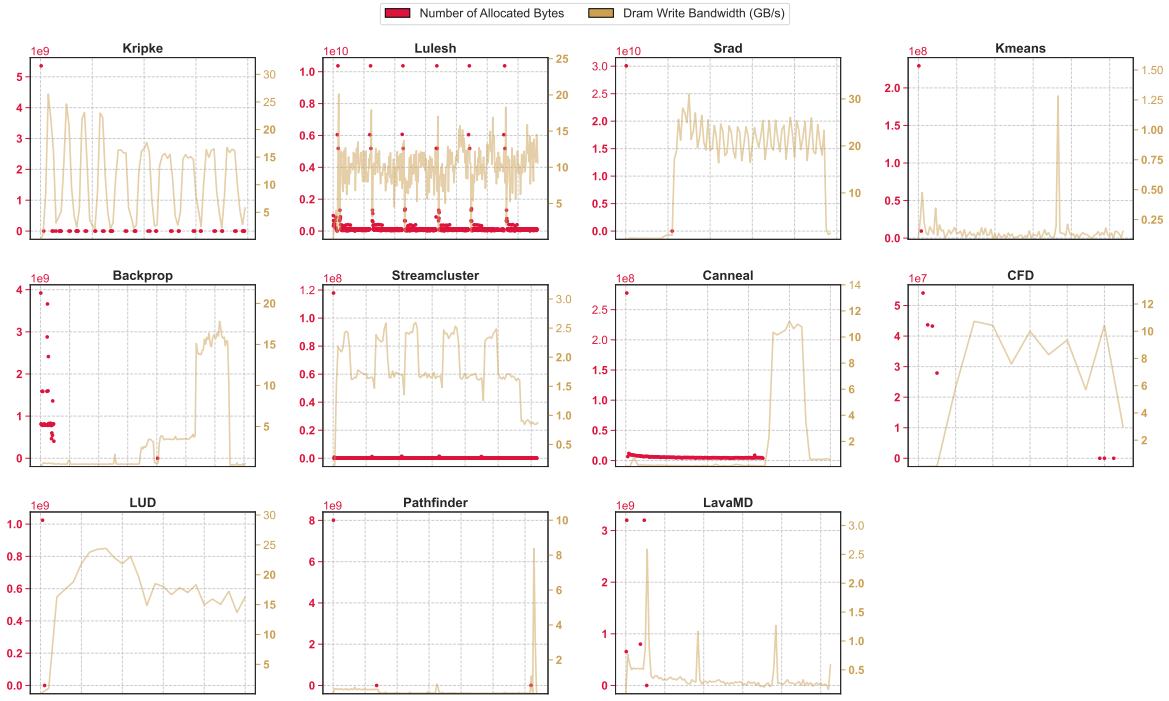

Figure 7.2.3: Correlation of allocated bytes (red) and memory bandwidth (yellow) over the execution time, for all benchmarks.

Benchmark Name	Benchmark Suite	Description
LULESH	CORAL-2	Simulates shock hydrodynamics for unstructured meshes in physics-based applications.
Kripke	CORAL-2	Proxy application for solving linear Boltzmann transport equations using sweep-based algorithms.
Streamcluster	PARSEC	Performs online clustering for data mining, often used in real-time applications.
Canneal	PARSEC	Uses simulated annealing for optimizing chip design layouts under constraints.
Kmeans	Rodinia	Clusters datasets into groups based on similarity using the k-means algorithm.
LavaMD	Rodinia	Simulates molecular dynamics for calculating particle interactions in a 3D space.
CFD	Rodinia	Performs computational fluid dynamics simulation for solving Navier-Stokes equations.
Backprop	Rodinia	Implements backpropagation for training neural networks in machine learning tasks.
Srad	Rodinia	Applies speckle reducing anisotropic diffusion for image denoising in medical imaging.
Pathfinder	Rodinia	Uses dynamic programming for pathfinding and traversal in 2D grids.
LUD	Rodinia	Performs LU decomposition to solve systems of linear equations efficiently.

Table 7.1: Benchmark details including name, suite, and description.

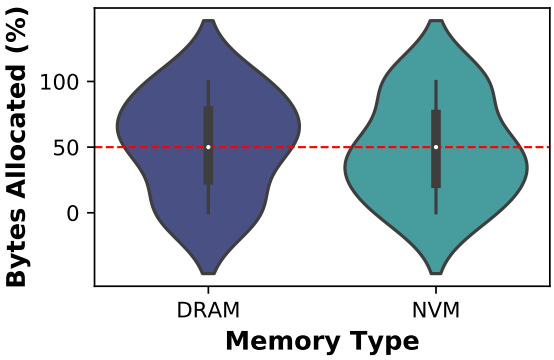

Figure 7.2.4: `SPID` data distribution over DRAM/NVM

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this work, we introduce `SPID`, a novel, low-overhead profiling and data placement approach designed for heterogeneous DRAM/NVM memory systems. `SPID` effectively leverages the correlation between intense allocation phases and high write bandwidth regions to optimize data placement. Our methodology demonstrates performance improvements of 30.82% over prior techniques on average, while reducing power consumption by an average of 31.61% and effectively handling the limited NVM endurance by reducing the number of write accesses. Additionally, we present `SPMalloc`, a custom, scalable, and open-source library designed to intercept dynamic data allocations at runtime. while enabling precise monitoring of allocation patterns within the target application, introducing minimal overhead.

8.2 Future Work

There are several promising directions for future work that could make this framework more reliable, versatile, and efficient. These improvements could target both the profiling capabilities and functionalities of SPMalloc as well as the placement algorithm and methodology of SPID. By addressing these areas, SPID can evolve into a more robust and intelligent framework for heterogeneous memory management. Some potential optimizations include:

- **Finer Granularity in Profiling:** Enhancing the profiling stage by incorporating more sophisticated monitoring tools, such as a custom Intel Pin tool, to work in conjunction with SPMalloc's existing monitoring procedures. The initial profiling data provided by SPMalloc could guide these tools to focus on specific time phases, thereby reducing overhead and improving efficiency. This approach could enable SPID to adapt its profiling to the dynamic behavior of workloads more effectively.
- **Adaptability in Noisy Environments:** Our experiments were conducted under a controlled setup, as discussed in Chapter 7, where noise was minimized. To improve the framework's reliability, future work could focus on enhancing the profiling and placement stages to perform effectively even under high system load and noisy conditions. This would increase SPID's adaptability to real-world scenarios, making it suitable for diverse and unpredictable environments.
- **Intelligent Placement Tools:** Incorporating advanced tools into the placement stage to enable more accurate and context-aware decisions. For example, integrating machine learning models and heuristics could allow SPID to predict future allocation patterns based on SPMalloc's profiling data. Such predictive capabilities could lead to smarter placement strategies that maximize performance and minimize memory bottlenecks.

- **Extending SPMalloc's Functionality:** Expanding the monitoring capabilities of SPMalloc to track additional metrics, such as the lifetime of allocation objects, could provide deeper insights into memory usage patterns. Similarly, adding more logic to the allocation and deallocation functions could facilitate better placement decisions in real time. For instance, integrating algorithms to track and categorize memory access frequencies or sizes could allow for more dynamic allocation policies.
- **Integration with System-Level Management:** Another potential direction is to explore integration with OS-level memory managers or virtual memory systems to enhance placement decisions. By working at a system-wide level, SPID could make placement decisions that account for all memory usage on the system, not just the benchmark or workload under test.
- **Support for Emerging Memory Technologies:** Future heterogeneous memory systems may incorporate new technologies beyond Optane and DRAM, such as CXL (Compute Express Link) or non-volatile RAM. Extending SPID's profiling and placement strategies to accommodate these new memory types would make it future-proof and more versatile.

Bibliography

[1] S. Shiratake, "Scaling and performance challenges of future dram," in *2020 IEEE international memory workshop (IMW)*, IEEE, 2020, pp. 1–3.

[2] L. Orosa et al., "Codic: A low-cost substrate for enabling custom in-dram functionalities and optimizations," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2021, pp. 484–497.

[3] J. Yang, B. Li, and D. J. Lilja, "Exploring performance characteristics of the optane 3d xpoint storage technology," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 5, no. 1, pp. 1–28, 2020.

[4] V. Gupta, S. Kapur, S. Saurabh, and A. Grover, "Resistive random access memory: A review of device challenges," *IETE Technical Review*, vol. 37, no. 4, pp. 377–390, 2020.

[5] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2013, pp. 256–267.

[6] Y. Guo, W. Xiao, Q. Liu, and X. He, "A cost-effective and energy-efficient architecture for die-stacked dram/nvm memory systems," in *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, IEEE, 2018, pp. 1–2.

[7] A. Shanbhag, N. Tatbul, D. Cohen, and S. Madden, "Large-scale in-memory analytics on intel® optane™ dc persistent memory," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, 2020, pp. 1–8.

[8] F. Li, "Cloud-native database systems at alibaba: Opportunities and challenges," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2263–2272, 2019.

[9] M. Hennecke, "Daos: A scale-out high performance storage stack for storage class memory," *Supercomputing frontiers*, vol. 40, 2020.

[10] *Google Cloud and Intel Optane DCPM*, Accessed: 15-11-2024.

[11] A. Eisenman et al., "Bandana: Using non-volatile memory for storing deep learning models," *Proceedings of machine learning and systems*, vol. 1, pp. 40–52, 2019.

[12] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "Oltp through the looking glass, and what we found there," in *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, 2018, pp. 409–439.

[13] C. Wang et al., "Panthera: Holistic memory management for big data processing over hybrid memories," in *PLDI*, 2019, pp. 347–362.

[14] S. Kannan, Y. Ren, and A. Bhattacharjee, "Klocs: Kernel-level object contexts for heterogeneous memory systems," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 65–78.

[15] H. Liu, R. Liu, X. Liao, H. Jin, B. He, and Y. Zhang, "Object-level memory allocation and migration in hybrid memory systems," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1401–1413, 2020.

[16] E. Karimov, T. Evenblij, S. Alinezhad Chamazcoti, and F. Catthoor, "Parl: Page allocation in hybrid main memory using reinforcement learning," *Available at SSRN 4857236*,

[17] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A hybrid memory page scheduler with machine intelligence," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 37–48.

[18] M. B. Olson, B. Kammerdiener, M. R. Jantz, K. A. Doshi, and T. Jones, "Online application guidance for heterogeneous memory systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 3, pp. 1–27, 2022.

[19] H. A. Maruf et al., "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 742–755.

[20] S. Wen, L. Cherkasova, F. X. Lin, and X. Liu, "Profdp: A lightweight profiler to guide data placement in heterogeneous memory systems," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 263–273.

[21] M. Katsaragakis, L. Papadopoulos, C. Baloukas, and D. Soudris, "Memory management methodology for application data structure refinement and placement on heterogeneous dram/nvm systems," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2022, pp. 748–753.

[22] I. Peng, K. Wu, J. Ren, D. Li, and M. Gokhale, "Demystifying the performance of hpc scientific applications on nvm-based memory systems," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2020, pp. 916–925.

[23] Y. Chen, I. B. Peng, Z. Peng, X. Liu, and B. Ren, "Atmem: Adaptive data placement in graph applications on heterogeneous memories," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 293–304.

[24] C.-K. Luk et al., "Pin: Building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[25] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[26] E. D. Berger, "Scalene: Scripting-language aware profiling for python," *arXiv preprint arXiv:2006.03879*, 2020.

[27] J. Klinkenberg et al., "Phase-based data placement optimization in heterogeneous memory," in *CLUSTER 2024-International Conference on Cluster Computing*, IEEE, 2024.

[28] A. Pal and M. Pal, "Interval tree and its applications," *Advanced Modeling and Optimization*, vol. 11, no. 3, pp. 211–224, 2009.

[29] E. Lee, J. Kim, H. Bahn, S. Lee, and S. H. Noh, "Reducing write amplification of flash storage through cooperative data management with nvm," *ACM Transactions on Storage (TOS)*, vol. 13, no. 2, pp. 1–13, 2017.

[30] *Memkind API*, Accessed: 15-11-2024.

[31] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[32] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*, Ieee, 2009, pp. 44–54.

[33] Lawrence Livermore National Laboratory, *Coral-2 benchmarks*, Accessed: 2024-10-18, 2023. [Online]. Available:

[34] *Intel PCM*, Accessed: 15-11-2024.

[35] *IPCMTL*, Accessed: 15-11-2024.

[36] R. Awati and I. Wigmore, *What is hierarchy (memory hierarchy)?* Accessed: 2024-12-05, 2024.

[37] S. Scargall, *Programming persistent memory: A comprehensive guide for developers*. Springer Nature, 2020.

[38] K. Biriukov, "Current and upcoming challenges of the main memory system (2021)," *International Journal of Computer Applications*, vol. 975, p. 8887, 2021.

[39] Y. Li et al., "Capmaestro: Exploiting power redundancy, data center-wide priorities, and stranded power for boosting data center performance," *IBM Research Report RC25680*, 2018.

[40] Pure Storage, *What is persistent memory?* Accessed: 2024-12-15.

[41] A. Narayan, T. Zhang, S. Aga, S. Narayanasamy, and A. Coskun, "Moca: Memory object classification and allocation in heterogeneous memory systems," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2018, pp. 326–335.

[42] J. Izraelevitz et al., "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[43] M. Katsaragakis, C. Baloukas, L. Papadopoulos, V. Kantere, F. Catthoor, and D. Soudris, "Energy consumption evaluation of optane dc persistent memory for indexing data structures," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, IEEE, 2022, pp. 75–84.

[44] T. Hirofuchi and R. Takano, "A prompt report on the performance of intel optane dc persistent memory module," *IEICE TRANSACTIONS on Information and Systems*, vol. 103, no. 5, pp. 1168–1172, 2020.

[45] H.-K. Liu et al., "A survey of non-volatile main memory technologies: State-of-the-arts, practices, and future directions," *Journal of Computer Science and Technology*, vol. 36, pp. 4–32, 2021.

[46] Intel Corporation, *Intel® memory latency checker v3.11b*, Accessed: 2024-12-15.

[47] StorageReview, *Intel optane dc persistent memory module (pmm)*, Accessed: 2024-12-15.

[48] Z. Chen, Y. Hua, B. Ding, and P. Zuo, "Lock-free concurrent level hashing for persistent memory," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 799–812.

[49] S. Kargar and F. Nawab, "Extending the lifetime of nvm: Challenges and opportunities," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 3194–3197, 2021.

[50] *Persistent memory is revolutionary*, Accessed: 2024-12-15.

[51] J. Handy, *What is snia's persistent memory programming model?* The SSD Guy Blog, Published: March 19, 2019, Accessed: 2024-12-15.

[52] M. Katsaragakis, K. Stavrakakis, D. Masouros, L. Papadopoulos, and D. Soudris, "Adjacent lstm-based page scheduling for hybrid dram/nvm memory systems," in *14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023)*, Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

[53] M. Katsaragakis, C. Baloukas, L. Papadopoulos, and F. Catthoor, "Performance, energy and nvm lifetime-aware data structure refinement and placement for heterogeneous memory systems," *Authorea Preprints*, 2023.

[54] D. Moura, D. Mossé, and V. Petrucci, "Performance characterization of autonuma memory tiering on graph analytics," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2022, pp. 171–184.

[55] M. G. Wrighton and A. M. DeHon, "Hardware-assisted simulated annealing with application for fast fpga placement," in *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, 2003, pp. 33–42.

[56] N. Bhargav, *Static vs. dynamic linking*, Reviewed by Milos Simic, Published on Baeldung CS, Accessed: 2024-12-15.

[57] GeeksforGeeks, *Static and dynamic linking in operating systems*, Accessed: 2024-12-15.

[58] Baeldung, *What is the ld_preload trick?* Reviewed by Bruno Fontana, Published on Baeldung, Accessed: 2024-12-15.

[59] GNU, *Memory allocation hooks*, Accessed: 2024-12-15.