Ατομική Διπλωματική Εργασία

EXPERIMENTAL ANALYSIS OF CACHING STRATEGIES FOR HETEROGENEOUS MEMORY

Αντρέας Γ. Λαός

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2026

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πειραματική Ανάλυση Στρατηγικών Κρυφής Μνήμης για Ετερογενή Μνήμη

Αντρέας Γ. Λαός

Επιβλέπων Καθηγητής

Χάρης Βώλος

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2026

Ευχαριστίες

Θα ήθελα να εκφράσω την ειλικρινή μου ευγνωμοσύνη στον επιβλέποντα καθηγητή μου, κ. Χάρη Βώλο, για την πολύτιμη καθοδήγηση, τις εύστοχες παρατηρήσεις και την υποστήριξη του καθ΄ όλη την διάρκεια εκπόνησης της παρούσας εργασίας. Η εμπειρία και οι γνώσεις του υπήρξαν καθοριστικές για την ολοκλήρωση της.

Επίσης, θέλω να ευχαριστήσω θερμά την οικογένεια μου και τους φίλους μου για την αμέριστη συμπαράσταση και την ηθική τους υποστήριξη όλα αυτά τα χρόνια των σπουδών μου.

Περίληψη

Η αυξανόμενη ζήτηση για υπολογιστική ισχύ στα σύγχρονα κέντρα δεδομένων, καθιστά την κατανάλωση ενέργειας ένα από τους πιο σημαντικούς παράγοντες κόστους και επιβάρυνσης του περιβάλλοντος. Η αντικατάσταση της DRAM με NVM αποτελεί μια από τις πιο υποσχόμενες πλέον κατευθύνσεις για την βελτίωση της ενεργειακής απόδοσης, διατηρώντας παράλληλα υψηλή χωρητικότητα μνήμης. Ωστόσο, η χρήση NVM ακολουθείτε από σημαντικές προκλήσεις απόδοσης και συνέπειας δεδομένων, καθώς η τεχνολογία NVM παρουσιάζει σημαντικά υψηλότερες καθυστερήσεις ανάγνωσης και εγγραφής σε σχέση με την DRAM.

Στην παρούσα εργασία εξετάζεται η αποτελεσματικότητα διαφορετικών πολιτικών caching σε ένα υβριδικό σύστημα μνήμης που αποτελείται από DRAM ως cache μπροστά από NVM. Τα συστήματα αυτά αντιμετωπίζουν ένα θεμελιώδες δίλημμα: η Write-Through εγγυάται συνέπεια δεδομένων αλλά επιβαρύνει την απόδοση λόγω της υψηλής καθυστέρησης εγγραφής στην NVM, ενώ η Write-Back προσφέρει καλύτερη απόδοση χωρίς όμως consistency. Κεντρικό ερώτημα αποτελεί το κατά πόσο η εφαρμογή μιας επιλεκτικής πολιτικής, όπου η Write-Through εφαρμόζεται αποκλειστικά στα objects που απαιτούν συνέπεια δεδομένων και η Write-Back στα υπόλοιπα, μπορεί να μειώσει σημαντικά το write traffic στην NVM χωρίς να υπάρξει δυσανάλογη επιβάρυνση στην απόδοση.

Για την μελέτη αυτή υλοποιήθηκε ένα εργαλείο δυναμικής ανάλυσης βασισμένο στο Intel PIN framework, το οποίο καταγράφει τα memory accesses μιας εφαρμογής με ακρίβεια επιπέδου allocation site. Το εργαλείο εφαρμόστηκε στο Memcached 1.6.14 με ρεαλιστικό read-only workload ενός εκατομμυρίου κλειδιών. Τα αποτελέσματα τροφοδοτήθηκαν στον προσομοιωτή DineroIV για ανάλυση της συμπεριφοράς της cache υπό διαφορετικές πολιτικές και μεγέθη.

Τα πειραματικά αποτελέσματα δείχνουν ότι η Selective Policy επιτυγχάνει δραματική μείωση του write traffic στην NVM. Υποδεικνύεται ότι η επιλεκτική εφαρμογή πολιτικών caching αποτελεί μια αποδοτική στρατηγική για συστήματα ετερογενούς μνήμης.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Η ραγδαία ανάπτυξη των υπηρεσιών cloud computing και των κέντρων δεδομένων τις τελευταίες δεκαετίες έχει οδηγήσει σε εκθετική αύξηση της ζήτησης για υπολογιστικούς πόρους και χωρητικότητα μνήμης. Η DRAM αποτελεί εδώ και δεκαετίες την κυρίαρχη τεχνολογία κύριας μνήμης στα σύγχρονα συστήματα, προσφέροντας υψηλές ταχύτητες πρόσβασης και αξιοπιστία. Ωστόσο, η DRAM παρουσιάζει σημαντικά μειονεκτήματα που καθίστανται όλο και πιο κρίσιμα στο πλαίσιο των σύγχρονων κέντρων δεδομένων όπως η υψηλή κατανάλωση ενέργειας τόσο κατά τη λειτουργία όσο και σε κατάσταση αναμονής, καθώς και περιορισμένη πυκνότητα αποθήκευσης σε σχέση με το κόστος της [25], δηλαδή η DRAM έχει χαμηλότερη χωρητικότητα ανά τετραγωνικό χιλιοστό chip σε σχέση με NVM τεχνολογίες.

Στο πλαίσιο αυτό, η Non-Volatile Memory (NVM) έχει αναδειχθεί ως μια από τις πλέον υποσχόμενες τεχνολογίες για τη βελτίωση της ενεργειακής απόδοσης των κέντρων δεδομένων [1]. Τεχνολογίες όπως η Phase Change Memory (PCM) και η Intel Optane προσφέρουν σημαντικά χαμηλότερη κατανάλωση ενέργειας, υψηλότερη πυκνότητα αποθήκευσης και το κρίσιμο χαρακτηριστικό της μη-πτητικότητας, δηλαδή της διατήρησης των δεδομένων ακόμα και χωρίς παροχή ρεύματος [18]. Ωστόσο, οι τεχνολογίες NVM παρουσιάζουν σημαντικά υψηλότερες καθυστερήσεις πρόσβασης σε

σχέση με τη DRAM, ιδιαίτερα στις εγγραφές, καθιστώντας την ολική αντικατάσταση της DRAM μη πρακτική για εφαρμογές που απαιτούν υψηλές επιδόσεις [18].

Η σημερινή καλύτερη προσέγγιση για την αντιμετώπιση αυτού του συμβιβασμού είναι η υιοθέτηση υβριδικών συστημάτων μνήμης, στα οποία η DRAM λειτουργεί ως cache υψηλής ταχύτητας μπροστά από μια μεγαλύτερη και ενεργειακά αποδοτικότερη NVM [1]. Η προσέγγιση αυτή επιτρέπει την αξιοποίηση των πλεονεκτημάτων και των δύο τεχνολογιών, ενώ παράλληλα εγείρει σημαντικά ερωτήματα σχετικά με τις πολιτικές διαχείρισης της cache που εφαρμόζονται σε αυτό το υβριδικό περιβάλλον [1].

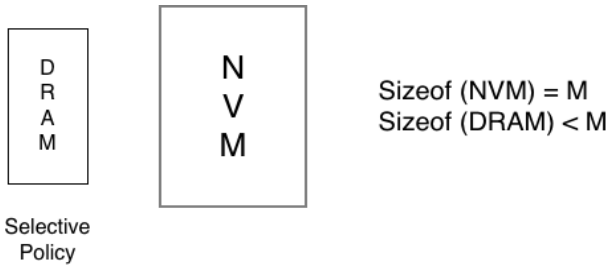
1.2 Πρόβλημα και Ερώτημα Έρευνας

Η χρήση των υβριδικών συστημάτων μνήμης εισάγει ένα θεμελιώδες δίλημμα στη διαχείριση της πολιτικής cache. Στα συστήματα αυτά, η DRAM λειτουργεί ως cache μπροστά από την NVM και κάθε εγγραφή πρέπει να διαχειριστεί με βάση μια από τις δύο βασικές πολιτικές, Write-Through ή Write-Back [18].

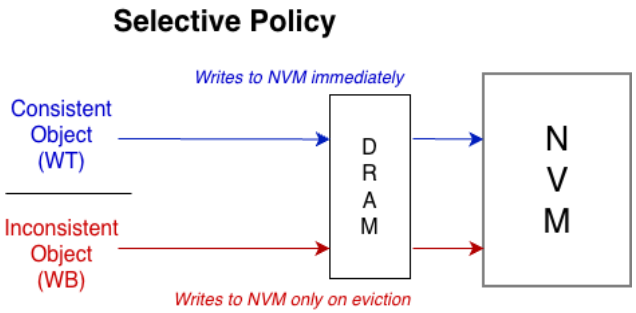
Παρά τα προφανή πλεονεκτήματα της NVM που αναφέρονται στο 1.1, η αντικατάσταση της DRAM δεν είναι τετριμμένη. Το βασικό ερώτημα που καλείται να απαντήσει η παρούσα εργασία είναι κατά πόσο είναι εφικτή η αξιοποίηση της NVM ως μερικής αντικατάστασης της DRAM στα κέντρα δεδομένων, με τρόπο που να διατηρεί υψηλή απόδοση και παράλληλα να επιτυγχάνει σημαντική βελτίωση της ενεργειακής αποδοτικότητας.

Με την πολιτική Write-Through, κάθε εγγραφή προωθείται αμέσως στην NVM, εγγυώμενη τη συνέπεια των δεδομένων (data consistency) ανά πάσα στιγμή. Ωστόσο, λόγω της υψηλής καθυστέρησης για εγγραφές της NVM, η πολιτική αυτή επιβαρύνει σημαντικά την απόδοση του συστήματος [18]. Αντίθετα, η πολιτική Write-Back καταγράφει τις εγγραφές αρχικά μόνο στην DRAM και τις προωθεί στην NVM κατά το eviction των cache blocks, επιτυγχάνοντας καλύτερη απόδοση αλλά χωρίς εγγύηση για συνέπεια δεδομένων στην NVM [18]. Το δίλημμα επιλογής πολιτικής cache αποτελεί το κύριο εμπόδιο για την αποτελεσματική αξιοποίηση της NVM σε πραγματικά συστήματα.

Στο πλαίσιο αυτό προκύπτει το ερώτημα, κατά πόσο είναι απαραίτητη η εφαρμογή ίδιας πολιτικής σε όλα τα memory objects μιας εφαρμογής. Παρατηρήθηκε ότι στις πραγματικές εφαρμογές συνυπάρχουν δύο κατηγορίες objects, τα persistent objects τα οποία αποθηκεύουν κρίσιμα δεδομένα που πρέπει πάντα να είναι ενημερωμένα στην NVM, και τα volatile objects, τα οποία αποτελούν εφήμερα runtime δεδομένα που χάνονται κατά την επανεκκίνηση και δεν απαιτούν consistency. Η διάκριση αυτή οδηγεί στο κεντρικό ερώτημα έρευνας της παρούσας εργασίας το οποίο είναι, αν μπορεί μια Selective Write Policy που εφαρμόζει Write-Through αποκλειστικά στα persistent objects και Write-Back τα volatile, να μειώνει σημαντικά το write traffic προς την NVM, διατηρώντας παράλληλα την εγγύηση συνέπειας δεδομένων (data consistency);



Σχήμα 1.1: Αρχιτεκτονική Υβριδικού Συστήματος



Σχήμα 1.2: Selective Policy Architecture

1.3 Στόχοι

Η παρούσα εργασία έχει ως κεντρικό στόχο την πειραματική διερεύνηση της εφαρμοσιμότητας και της αποτελεσματικότητας μιας Selective Write Policy σε υβριδικά συστήματα μνήμης DRAM+NVM, με σκοπό την επίτευξη ενεργειακά αποδοτικής λειτουργίας χωρίς δυσανάλογη επιβάρυνση της απόδοσης. Προς επίτευξη του στόχου αυτού, ορίζονται οι παρακάτω επιμέρους στόχοι.

Πρώτον, η σχεδίαση και υλοποίηση ενός εργαλείου δυναμικής ανάλυσης μνήμης βασισμένο στο Intel PIN framework [21], το οποίο να επιτρέπει την καταγραφή των memory accesses μιας εφαρμογής με ακρίβεια επιπέδου allocation site. Η δυνατότητα αυτή είναι απαραίτητη για τη διάκριση μεταξύ persistent και volatile memory objects κατά την εκτέλεση.

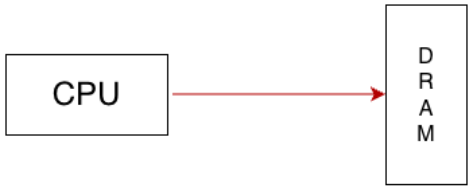
Δεύτερον, η εφαρμογή του εργαλείου στην εφαρμογή Memcached 1.6.14 [10] με ρεαλιστικό workload, και η κατηγοριοποίηση των memory objects της εφαρμογής σε persistent και volatile μέσω ανάλυσης του πηγαίου κώδικα.

Τρίτον, η ποσοτική σύγκριση των πολιτικών Write-Back, Write-Through και Selective ως προς το write traffic προς την NVM και την απόδοση του συστήματος, μετρούμενη μέσω του Average Memory Access Time (AMAT), με τη χρήση του προσομοιωτή cache DineroIV [7].

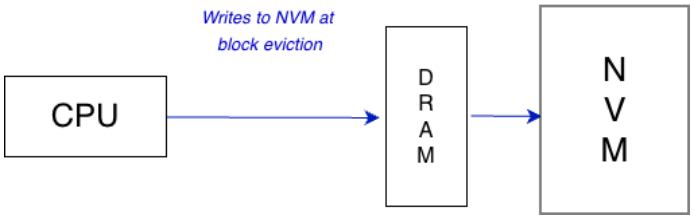
Τέταρτον, η αξιολόγηση της Selective Write Policy ως βιώσιμης στρατηγικής για υβριδικά συστήματα μνήμης, εξετάζοντας κατά πόσο επιτυγχάνει ικανοποιητική μείωση του write traffic διατηρώντας παράλληλα αποδεκτή απόδοση συγκριτικά με τις καθιερωμένες πολιτικές.

Οι τοπολογίες που θα μελετηθούν σε βάθος είναι οι εξής:

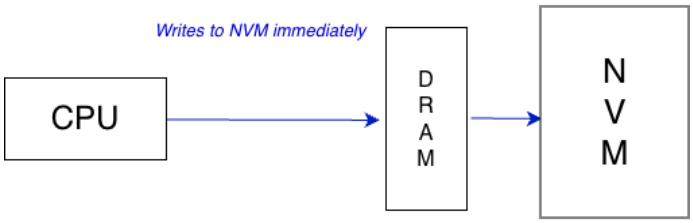
Baseline



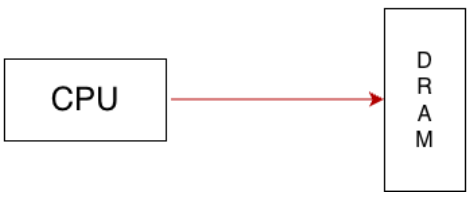
Write-Back Policy



Write-Through Policy



Selective Policy



Κεφάλαιο 2

Υπόβαθρο

2.1 Ιεραρχία Μνήμης και CPU Cache

Η ιεραρχία μνήμης αποτελεί θεμελιώδη αρχή σχεδιασμού των σύγχρονων υπολογιστικών συστημάτων. Η βασική ιδέα στηρίζεται στην παρατήρηση ότι διαφορετικές τεχνολογίες μνήμης παρουσιάζουν διαφορετικά χαρακτηριστικά ταχύτητας, χωρητικότητας και κόστους [22]. Στην κορυφή της ιεραρχίας βρίσκονται οι καταχωρητές του επεξεργαστή, ακολουθούμενοι από τα επίπεδα cache (L1, L2, L3), την κύρια μνήμη (DRAM) και τέλος την δευτερεύουσα αποθήκευση. Στο πλαίσιο της παρούσας εργασίας, το σενάριο που εξετάζεται τροποποιεί αυτή την ιεραρχία αντικαθιστώντας την DRAM ως κύρια μνήμη με ένα υβριδικό σύστημα, στο οποίο η DRAM λειτουργεί ως cache υψηλής ταχύτητας μπροστά από την NVM.

Η CPU cache αποτελεί κρίσιμο στοιχείο της ιεραρχίας μνήμης, καθώς γεφυρώνει το χάσμα ταχύτητας μεταξύ του επεξεργαστή και της κύριας μνήμης. Η λειτουργία της βασίζεται στις αρχές της χρονικής τοπικότητας (temporal locality), σύμφωνα με την οποία δεδομένα που προσπελάστηκαν πρόσφατα είναι πιθανό να προσπελαστούν ξανά σύντομα, και της χωρικής τοπικότητας (spatial locality), σύμφωνα με την οποία δεδομένα που βρίσκονται κοντά σε ήδη προσπελασμένα δεδομένα είναι πιθανό να προσπελαστούν στο άμεσο μέλλον [22]. Όταν ο επεξεργαστής αναζητά δεδομένα που

βρίσκονται ήδη στην cache, συμβαίνει cache hit, ενώ στην αντίθετη περίπτωση συμβαίνει cache miss και τα δεδομένα φορτώνονται από το επόμενο επίπεδο ιεραρχίας.

Ένα κρίσιμο μέτρο απόδοσης της cache είναι το miss rate, δηλαδή το ποσοστό των προσβάσεων που δεν βρίσκουν τα δεδομένα στην cache και αναγκάζονται να τα αναζητήσουν στο επόμενο επίπεδο της ιεραρχίας. Συνδυαστικά μέτρο απόδοσης αποτελεί το Average Memory Access Time (AMAT), το οποίο εκφράζει τον μέσο χρόνο πρόσβασης στην μνήμη λαμβάνοντας υπόψη τόσο τα cache hits όσο και τα cache misses [22]. Ο τύπος υπολογισμού του AMAT ορίζεται ως εξής:

$$AMAT = T\_hit + MissRate \times T\_miss$$

Όπου T_hit είναι ο χρόνος πρόσβασης σε περίπτωση hit, MissRate το ποσοστό των misses και T_miss ο χρόνος πρόσβασης στο επόμενο επίπεδο της ιεραρχίας σε περίπτωση miss.

Η cache είναι οργανωμένη σε cache lines ή αλλιώς cache blocks. Αυτές αποτελούν τη μονάδα μεταφοράς δεδομένων μεταξύ των επιπέδων ιεραρχίας μνήμης [22]. Όταν ένα cache block τροποποιείται από εγγραφή, χαρακτηρίζεται ως dirty, δηλαδή το περιεχόμενο του στην cache δεν συμφωνεί πλέον με το αντίστοιχο περιεχόμενο του στο κατώτερο επίπεδο μνήμης. Όταν η cache εξαντλήσει τον διαθέσιμο της χώρο και χρειαστεί να φιλοξενήσει νέα δεδομένα, πρέπει να αντικαταστήσει κάποιο από τα υπάρχοντα blocks. Αυτή η διαδικασία ονομάζεται eviction. Η επιλογή του block προς αντικατάσταση κατά το eviction καθορίζεται από την replacement policy. Στην περίπτωση που το block προς αντικατάσταση είναι dirty, πρέπει να εγγραφεί στο κατώτερο επίπεδο ιεραρχίας πριν αντικατασταθεί. Ο τρόπος με τον οποίο γίνεται η διαχείριση των εγγραφών καθορίζεται από την εκάστοτε write policy. Οι write policies που εφαρμόζονται στο πλαίσιο της παρούσας εργασίας αναλύονται στο υποκεφάλαιο 2.3.

2.2 Non-Volatile Memory (NVM)

Η Non-Volatile Memory (NVM) αποτελεί μια κατηγορία τεχνολογιών μνήμης που διατηρούν τα αποθηκευμένα δεδομένα ακόμα και χωρίς παροχή ρεύματος, σε αντίθεση με την πτητική DRAM η οποία χάνει το περιεχόμενο της κατά την απώλεια

τροφοδοσίας [18]. Πέραν της μη πτητικότητας, οι τεχνολογίες NVM προσφέρουν σημαντικά πλεονεκτήματα έναντι της DRAM, όπως υψηλότερη πυκνότητα αποθήκευσης, χαμηλότερη κατανάλωση ενέργειας σε κατάσταση αναμονής και χαμηλότερο κόστος ανά gigabyte [18].

Τα πλεονεκτήματα της NVM έναντι της DRAM απορρέουν από τον θεμελιωδώς διαφορετικό μηχανισμό αποθήκευσης δεδομένων που χρησιμοποιεί. Σε αντίθεση με την DRAM, η οποία αποθηκεύει δεδομένα ως ηλεκτρικό φορτίο σε πυκνωτές που απαιτούν συνεχή ανανέωση για την διατήρηση του περιεχομένου τους, η NVM αξιοποιεί υλικά που μεταβάλλουν μονίμως τις φυσικές τους ιδιότητες κατά την εγγραφή [18]. Το χαρακτηριστικό αυτό προσδίδει στην NVM τα πλεονεκτήματα που προαναφέρθηκαν. Ωστόσο, ο ίδιος μηχανισμός εισάγει και ένα σημαντικό μειονέκτημα. Η φυσική μεταβολή των υλικών κατά την εγγραφή απαιτεί περισσότερο χρόνο σε σχέση με την απλή φόρτιση πυκνωτή της DRAM, με αποτέλεσμα η καθυστέρηση ανάγνωσης της NVM να είναι περίπου 2-3 φορές υψηλότερη και η καθυστέρηση εγγραφής έως και 10 φορές υψηλότερη από αυτή της DRAM [18]. Οι τιμές αυτές επηρεάζουν άμεσα το AMAT του συστήματος, καθιστώντας την επιλογή της κατάλληλης write policy κρίσιμο παράγοντα απόδοσης σε υβριδικά συστήματα DRAM+NVM. Οι δύο πλέον σημαντικές τεχνολογίες NVM που σχετίζονται με την παρούσα εργασία είναι η Phase Change Memory (PCM), η οποία αξιοποιεί αλλαγές φάσης υλικών για την αποθήκευση δεδομένων, και η Intel Optane, βασισμένη στην τεχνολογία 3D XPoint, η οποία αποτέλεσε την πρώτη εμπορικά διαθέσιμη byte-addressable NVM [18].

Ένας επιπλέον περιορισμός των τεχνολογιών NVM είναι η περιορισμένη αντοχή εγγραφής (write endurance), δηλαδή ο περιορισμένος αριθμός εγγραφών που μπορεί να δεχτεί κάθε κελί μνήμης πριν αχρηστευτεί [18]. Ο περιορισμός αυτός καθιστά την μείωση του write traffic προς την NVM όχι μόνο ζήτημα απόδοσης αλλά και ζήτημα διάρκειας ζωής του υλικού, ενισχύοντας περαιτέρω την σημασία της Selective Write Policy που εξετάζεται στην παρούσα εργασία.

Λαμβάνοντας υπόψη τα παραπάνω χαρακτηριστικά, το σενάριο που εξετάζεται στην παρούσα εργασία αξιοποιεί την DRAM ως cache υψηλής ταχύτητας μπροστά από την NVM. Η προσέγγιση αυτή επιτρέπει την εκμετάλλευση της υψηλής απόδοσης της

DRAM για τις συχνές προσβάσεις, ενώ παράλληλα αξιοποιεί την ενεργειακή αποδοτικότητα και τη μεγάλη χωρητικότητα της NVM για την αποθήκευση των δεδομένων.

2.3 Write-Back και Write-Through Policies

Κρίσιμη απόφαση σχεδιασμού σε κάθε σύστημα cache και ιδιαίτερα σε υβριδικά συστήματα μνήμης οπού η κύρια μνήμη είναι NVM, είναι η επιλογή της κατάλληλης write policy . Η write policy επηρεάζει με άμεσο τρόπο την απόδοση του συστήματος όσο και την συνέπεια των δεδομένων αφού καθορίζει πότε και πώς προωθούνται στο κατώτερο επίπεδο της ιεραρχίας μνήμης οι εγγραφές του συστήματος που πραγματοποιούνται στην cache [18].

Με Write-Through policy, κάθε εγγραφή στην cache προωθείται ταυτόχρονα και στο κατώτερο επίπεδο μνήμης. Με αυτή την πολιτική cache εξασφαλίζεται ανά πάσα στιγμή πλήρη συνέπεια μεταξύ του περιεχομένου της cache και της κύριας μνήμης [22]. Το βασικότερο πλεονέκτημα της πολιτικής Write-Through είναι ότι εγγυάται data consistency, αυτό σημαίνει ότι τα δεδομένα στην κύρια μνήμη είναι πάντα ενημερωμένα, ανεξαρτήτως της κατάστασης της cache. Παρ' όλ' αυτά, στο πλαίσιο υβριδικών συστημάτων DRAM+NVM, η Write-Through εισάγει σημαντικό performance penalty, καθώς κάθε εγγραφή απαιτεί άμεση πρόσβαση στην αργή NVM, αυξάνοντας σημαντικά το συνολικό write traffic και το AMAT [18].

Από την άλλη πλευρά, με την πολιτική Write-Back, οι εγγραφές καταγράφονται αρχικά μόνο στην cache και το block στο οποίο έγινε η εγγραφή επισημαίνεται ως dirty. Αργότερα γίνεται προώθηση των δεδομένων στο κατώτερο επίπεδο μνήμης κατά το eviction του dirty block από την cache [22]. Με αυτή την προσέγγιση μειώνεται σημαντικά το write traffic προς την NVM, αφού πολλές εγγραφές στο ίδιο block συγχωνεύονται σε μια μόνο εγγραφή προς την NVM [18][30]. Ωστόσο βασικό μειονέκτημα της πολιτικής Write-Back είναι η έλλειψη εγγύησης data consistency αφού σε περίπτωση απώλειας τροφοδοσίας τα dirty blocks που δεν έχουν ακόμα εγγραφεί στην κύρια μνήμη χάνονται, καταστρέφοντας την ακεραιότητα των δεδομένων [18][27].

Ο παρακάτω πίνακας συνοψίζει συγκριτικά τα βασικά χαρακτηριστικά των δύο πολιτικών:

	Write-Through	**Write-Back**
Data Consistency	Εγγυημένη	Μη εγγυημένη
Write Traffic	Υψηλό	Χαμηλό
NVM Latency Penalty	Υψηλό	Χαμηλό
Dirty Blocks	Δεν υπάρχουν	Υπάρχουν

Πίνακας 2.1: Σύγκριση βασικών χαρακτηριστικών Write-Through και Write-Back.

Το δίλημμα μεταξύ Write-Through και Write-Back οδηγεί στο κεντρικό ερώτημα της παρούσας εργασίας. Έχει παρατηρηθεί ότι στις πραγματικές εφαρμογές δεν απαιτούν όλα τα memory objects την ίδια πολιτική. Για ορισμένα objects είναι απαραίτητη η εγγύηση data consistency ενώ σε κάποια άλλα όχι. Η διαπίστωση αυτή αποτέλεσε το έναυσμα για την μελέτη μιας Selective Write Policy, η οποία εφαρμόζει πολιτική Write-Through αποκλειστικά στα persistent objects, δηλαδή σε αυτά που χρειάζονται data consistency, και Write-Back στα volatile objects με στόχο την επίτευξη του βέλτιστου συνδυασμού απόδοσης και συνέπειας δεδομένων.

2.4 Intel PIN Dynamic Binary Instrumentation

Το Intel PIN αποτελεί ένα framework dynamic binary instrumentation. Αυτός ο όρος αναφέρεται στην διαδικασίας εισαγωγής πρόσθετου κώδικα ανάλυσης σε ένα ήδη μεταγλωττισμένο(compiled) εκτελέσιμο πρόγραμμα (binary) κατά την διάρκεια της εκτέλεσης του, δηλαδή δυναμικά, χωρίς να απαιτείται πρόσβαση στον πηγαίο κώδικα ή επαναμεταγλώττιση [21]. Με τον τρόπο αυτό γίνεται εφικτή η παρακολούθηση και η ανάλυση της συμπεριφοράς οποιουδήποτε εκτελέσιμου προγράμματος κατά την εκτέλεση του, ανεξαρτήτως της γλώσσας προγραμματισμού στην οποία έχει αναπτυχθεί. Το framework έχει αναπτυχθεί από την Intel και χρησιμοποιείται ευρέως στην ακαδημαϊκή κοινότητα για ανάλυση απόδοσης, ανίχνευση σφαλμάτων και καταγραφή της συμπεριφοράς της μνήμης εκτελέσιμων προγραμμάτων.

Η λειτουργία του PIN framework είναι βασισμένη στον μηχανισμό Just-In-Time (JIT) μεταγλώττισης. Ο όρος JIT αναφέρεται στη μεταγλώττιση του κώδικα τη στιγμή που αυτός πρόκειται να εκτελεστεί, αντί να έχει μεταγλωττιστεί εξ ολοκλήρου εκ των προτέρων. Μέσω του μηχανισμού αυτού το framework παρεμβάλλεται μεταξύ του εκτελέσιμου προγράμματος και του λειτουργικού συστήματος [21], αναλύοντας τον κώδικα του προγράμματος σε επίπεδο εντολών. Επίσης, εισάγει δυναμικά πρόσθετο κώδικα ανάλυσης στα σημεία που ορίζει ο χρήστης, πριν την εκτέλεση κάθε τμήματος κώδικα. Η διαδικασία αυτή πραγματοποιείται χωρίς την τροποποίηση του αρχικού εκτελέσιμου αρχείου, επιτρέποντας την παρακολούθηση κάθε εντολής του προγράμματος με ελάχιστη επίδραση στην λογική εκτέλεσής του.

Κεντρικό στοιχείο του PIN framework είναι η έννοια του Pintool. Αυτή η έννοια αποτελεί ένα custom plugin που αναπτύσσει ο χρήστης και το οποίο ορίζει ποια γεγονότα θα παρακολουθούνται και πώς θα αναλύονται [21]. Ένα pintool μπορεί να καταγράφει οποιαδήποτε πτυχή της εκτέλεσης ενός προγράμματος, όπως κλήσεις συναρτήσεων, εντολές φόρτωσης και αποθήκευσης δεδομένων ή δυναμικές εκχωρήσεις μνήμης. Από το PIN παρέχεται πλούσιο API για την ανάπτυξη Pintools. Αυτό επιτρέπει την πρόσβαση σε λεπτομερείς πληροφορίες για κάθε εντολή, όπως η διεύθυνση μνήμης που προσπελαύνεται, το μέγεθος της πρόσβασης και το instruction pointer.

Στο πλαίσιο της εργασίας αυτής, αναπτύχθηκε ένα custom Pintool το οποίο καταγράφει τα memory accesses της εφαρμογής Memcached κατά την εκτέλεση της. Με τις καταγραφές αυτές μπορεί να συσχετίσει κάθε πρόσβαση με το allocation site που δημιούργησε το αντίστοιχο memory object αφού γίνεται και ανίχνευση των εκχωρήσεων μνήμης. Αυτή του η δυνατότητα καθώς και η ικανότητα του PIN να λειτουργεί σε multi-threaded περιβάλλοντα, το καθιστά το κατάλληλο εργαλείο για τους σκοπούς της παρούσας ανάλυσης. Η αναλυτική περιγραφή του Pintool που υλοποιήθηκε παρουσιάζεται στο Κεφάλαιο 3.

2.5 Memcached

Το Memcached είναι ένα ανοιχτού κώδικα, κατανεμημένο σύστημα caching μνήμης υψηλής απόδοσης. Είναι σχεδιασμένο για την αποθήκευση δεδομένων σε μορφή ζευγών κλειδιού-τιμής (key-value pairs) στην κύρια μνήμη [10]. Χρησιμοποιείται ευρέως σε production περιβάλλοντα ως ενδιάμεσο επίπεδο caching μεταξύ της εφαρμογής και της βάσης δεδομένων. Στόχος είναι η μείωση της καθυστέρησης των προσβάσεων σε συχνά χρησιμοποιούμενα δεδομένα και την ανακούφιση του φόρτου των βάσεων δεδομένων. Εταιρίες όπως Facebook, Twitter και Wikipedia αξιοποιούν το Memcached για την εξυπηρέτηση εκατομμυρίων αιτημάτων ανά δευτερόλεπτο [10].

Αρχιτεκτονικά, το Memcached κάνει χρήση slab allocator για να διαχειριστεί την μνήμη. Ο slab allocator οργανώνει την μνήμη σε κλάσεις σταθερού μεγέθους (slab classes) ώστε να αποφύγει τυχόν κατακερματισμό της μνήμης [10][9]. Κάθε αποθηκευμένο αντικείμενο (item) το τοποθετεί στην κατάλληλη slab class ανάλογα με το μέγεθός του. Για την αναζήτηση αντικειμένων στην μνήμη χρησιμοποιείται ένας Hash Table, ενώ η διαχείριση της χωρητικότητας βασίζεται σε πολιτική LRU (Least Recently Used), αντικαθιστώντας τα λιγότερο πρόσφατα χρησιμοποιημένα αντικείμενα όταν εξαντλείται ο διαθέσιμος χώρος [9]. Επιπλέον το Memcached υποστηρίζει multi-threaded εκτέλεση επιτρέποντας την παράλληλη εξυπηρέτηση πολλαπλών αιτημάτων από worker threads [29].

Η επιλογή για χρήση του Memcached ως εφαρμογή-στόχου στο πλαίσιο της παρούσας εργασίας είναι βασισμένη σε τρεις κύριους λόγους. Για αρχή, αποτελεί ένα ρεαλιστικό και ευρέως διαδεδομένο workload που αντιπροσωπεύει πραγματικές συνθήκες λειτουργίας κέντρων δεδομένων. Δεύτερον, η αρχιτεκτονική του περιλαμβάνει διαφορετικές κατηγορίες memory objects με διαφορετικές απαιτήσεις στην συνέπεια των δεδομένων, καθιστώντας το Memcached κατάλληλο για την μελέτη της Selective Write Policy. Τέλος, η multi-threaded φύση του επιτρέπει την αξιολόγηση του Pintool σε ρεαλιστικές συνθήκες παράλληλης εκτέλεσης.

Για την δημιουργία του workload στα πειράματα που εκτελέστηκαν στα πλαίσια της παρούσας εργασίας χρησιμοποιήθηκε το εργαλείο mcperf, το οποίο αποτελεί έναν

εξειδικευμένο load generator για το Memcached [19]. Ο mcperf generator επιτρέπει στον χρήστη τον καθορισμό του αριθμού των κλειδιών που θα φορτωθούν, το μέγεθος τους καθώς και το μέγεθος των τιμών. Επιτρέπει επίσης τον καθορισμό του ρυθμού των αιτημάτων ανά δευτερόλεπτο και της κατανομής των προσβάσεων. Για την παρούσα εργασία, ο mcperf χρησιμοποιήθηκε για την φόρτωση ενός εκατομμυρίου κλειδιών στο Memcached σε πρώτη φάση και μετά την παραγωγή ρεαλιστικού read-only workload κατά τη φάση μέτρησης (measurement phase). Η αναλυτική περιγραφή των παραμέτρων που χρησιμοποιήθηκαν για το workload παρουσιάζεται στο Κεφάλαιο 4.

2.6 DineroIV Cache Simulator

Ο DineroIV είναι ένας trace-driven προσομοιωτής cache. Επιτρέπει την προσομοίωση της συμπεριφοράς ιεραρχιών cache, βασισμένο σε ακολουθίες προσβάσεων μνήμης που έχουν καταγραφεί από πραγματικές εκτελέσεις προγραμμάτων [7]. Ο όρος trace-driven αναφέρεται στο ότι ο προσομοιωτής δεν εκτελεί το ίδιο το πρόγραμμα, αλλά επεξεργάζεται ένα αρχείο trace που περιέχει την ακολουθία των προσβάσεων μνήμης που παράχθηκε κατά την πραγματική εκτέλεση του προγράμματος.

Ο προσομοιωτής δέχεται σαν είσοδο ένα αρχείο trace. Στο αρχείο αυτό, κάθε γραμμή περιγράφει μια πρόσβαση μνήμης που έγινε στο πρόγραμμα αναφέροντας τον τύπο της πρόσβασης (load ή store) και την διεύθυνση μνήμης που προσπελαύνεται [7]. Παράλληλα, ο χρήστης ορίζει τις παραμέτρους της cache που θέλει να προσομοιώσει όπως το μέγεθος της cache, το μέγεθος του cache block και την write policy. Εξίσου σημαντική παράμετρος που ορίζει ο χρήστης, είναι η associativity. Η μνήμη χωρίζεται σε blocks, όπου κάθε block είναι μια συνεχόμενη περιοχή μνήμης. Κάθε διεύθυνση μνήμης ανήκει σε ένα συγκεκριμένο block. Η cache με τη σειρά της διαιρείται σε sets και κάθε block μνήμης αντιστοιχίζεται σε ένα συγκεκριμένο set βάσει της διεύθυνσης του [22]. Η παράμετρος associativity καθορίζει πόσα slots έχει το κάθε set. Δηλαδή καθορίζει πόσα διαφορετικά blocks μπορούν να ανατεθούν σε ένα συγκεκριμένο set ταυτόχρονα.

Αν η associativity είναι χαμηλή τότε αυξάνεται η πιθανότητα conflict misses, δηλαδή οι περιπτώσεις που κάποια blocks τα οποία χρησιμοποιούνται συχνά αντιστοιχίζονται στο

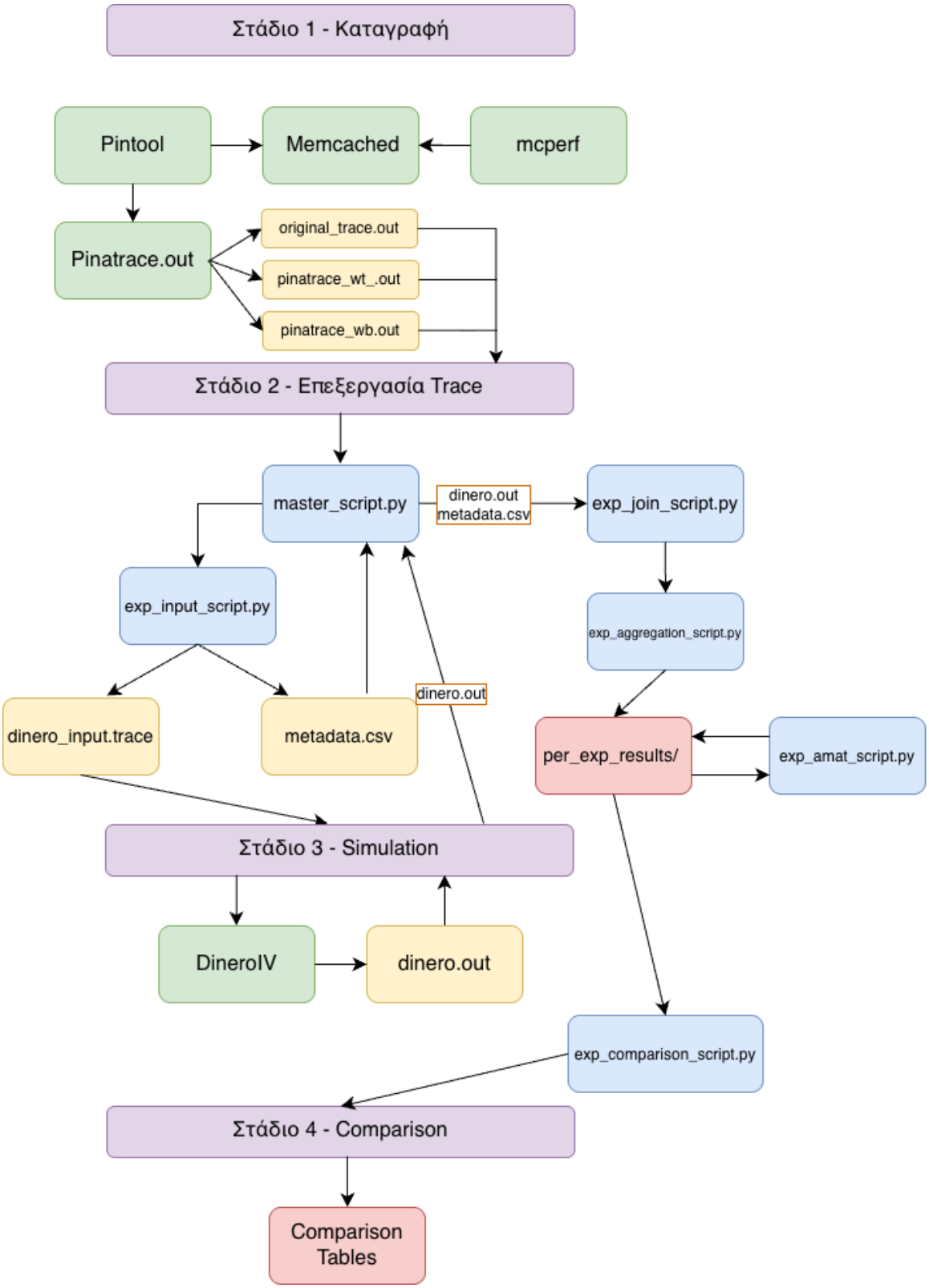
ίδιο set και αναγκαστικά το ένα εκδιώκει το άλλο. Αυτό έχει ως αποτέλεσμα την αύξηση του miss rate. Αντίθετα, με υψηλότερη associativity μειώνονται τα conflict misses και κατ' επέκταση το miss rate. Ωστόσο η αυξημένη associativity έχει ως αποτέλεσμα την αύξηση της πολυπλοκότητας του υλικού [22].

Στο πλαίσιο της παρούσας εργασίας, ο DineroIV χρησιμοποιήθηκε ως προσομοιωτής της συμπεριφοράς της CPU Cache υπό διαφορετικές write policies και διαφορετικά μεγέθη cache. Αξίζει να σημειωθεί ότι ο προσομοιωτής δεν μοντελοποιεί από μόνος του latencies. Για τον υπολογισμό του AMAT πραγματοποιείται επιπλέον επεξεργασία των αποτελεσμάτων του, η οποία περιγράφεται αναλυτικά στο Κεφάλαιο 4.

Κεφάλαιο 3

Σχεδιασμός και Υλοποίηση του Pintool

3.1 Αρχιτεκτονική Συστήματος



Σχήμα 3.1: Pipeline του συστήματος

Το σύστημα που αναπτύχθηκε στο πλαίσιο της παρούσας εργασίας οργανώνεται σε τέσσερα διακριτά στάδια επεξεργασίας. Τα στάδια αυτά μετατρέπουν την εκτέλεση μιας πραγματικής εφαρμογής σε ποσοτικά αποτελέσματα, ώστε να είναι δυνατή η σύγκριση των write policies υπό διαφορετικά δομημένες cache. Το παραπάνω σχεδιάγραμμα απεικονίζει την συνολική αρχιτεκτονική του συστήματος.

Το πρώτο στάδιο όπως απεικονίζεται αφορά την καταγραφή των memory accesses της εφαρμογής. Το Pintool εκτελείται μέσα στη διεργασία του Memcached υπό τον JIT μηχανισμό του PIN framework. Παράλληλα με το Memcached και το Pintool, ο mcperf generator παράγει το workload που εκτελεί η εφαρμογή. Το Pintool είναι σε θέση που μπορεί να καταγράφει memory allocations, deallocations(frees) και κάθε memory access (load/store). Επίσης συσχετίζει κάθε access με το allocation site που δημιούργησε το αντίστοιχο memory region. Το trace με τα αποτελέσματα εγγράφονται στο pinatrace.out αρχείο, το οποίο περιέχει την πλήρη ακολουθία των memory accesses με per allocation site attribution. Δηλαδή, αντιστοιχίζεται το κάθε memory access με το region στο οποίο έγινε το access.

Ενδιάμεσο βήμα πριν το δεύτερο στάδιο, ο χρήστης εκτελεί χειροκίνητα το segregate_trace.py. Αυτό το script διαβάζει το pinatrace.out και δημιουργεί δύο νέα αρχεία. Το πρώτο αρχείο που δημιουργεί είναι το pinatrace_wt.out, το οποίο περιέχει αποκλειστικά τα accesses των persistent objects. Το δεύτερο αρχείο είναι το pinatrace_wb.out, το οποίο περιέχει τα accesses των iinconsistent objects.

Το δεύτερο στάδιο αφορά την επεξεργασία του trace που παράγει το Pintool. Το master_script.py είναι αυτό που συντονίζει την εκτέλεση του pipeline με τα επιμέρους scripts που επεξεργάζονται το trace. Στην αρχή της εκτέλεσης του, το master_script.py ζητά από τον χρήστη να επιλέξει ποιο trace επιθυμεί να χρησιμοποιήσει, ανάλογα με το πείραμα που θέλει να εκτελέσει. Η διαδικασία αρχίζει με την ανάγνωση και τροποποίηση του trace από το exp_input_script.py, το οποίο παράγει δύο αρχεία, το dinero_input.trace, το οποίο είναι το trace με τα memory accesses μορφοποιημένο έτσι ώστε να είναι συμβατό με τον DineroIV και το metadata.csv, το οποίο αντιστοιχίζει κάθε access με το allocation site του, περιέχει δηλαδή τα metadata.

Το τρίτο στάδιο αφορά την προσομοίωση της CPU cache μέσω του DineroIV. Ο προσομοιωτής παίρνει ως είσοδο το dinero_input.trace και με τις καθορισμένες από τον χρήστη παραμέτρους της cache, παράγει το dinero.out, το οποίο περιέχει λεπτομερή στατιστικά για hits/misses καθώς και το write traffic (bytes to memory) για κάθε memory access. Η προσομοίωση εκτελείται ξεχωριστά για κάθε write policy και κάθε μέγεθος cache.

Το τέταρτο στάδιο αφορά την ανάλυση και σύγκριση των αποτελεσμάτων. Το exp_join_script.py αντιστοιχεί τις γραμμές του dinero.out και του metadata.csv με βάση το ID κάθε εγγραφής. Αποτέλεσμα αυτού είναι το joined_results.csv, το οποίο συνδυάζει τα metadata κάθε access με το αντίστοιχο αποτέλεσμα της προσομοίωσης (hit/miss). Το αρχείο αυτό διαγράφεται μετά την επεξεργασία για εξοικονόμηση χώρου. Στην συνέχεια το exp_aggregation_script.py επεξεργάζεται το joined_results.csv και αυτό παράγει στατιστικά σε επίπεδο συνόλου (global_stats.txt) αλλά και ανά allocation site (per_alloc_caller_stats.csv) και ανά base address (per_base_address_stats.csv). Ομοίως το exp_amat_script.py παίρνει σαν input το dinero_global_stats.txt και το global_stats.txt που παρήγαγε το aggregation script για να υπολογίσει το AMAT για DRAM και NVM. Τα αποτελέσματα όλων των scripts αποθηκεύονται σε ξεχωριστό φάκελο για κάθε πείραμα. Τέλος, το exp_comparison_script.py παράγει τους συγκριτικούς πίνακες που παρουσιάζονται στο Κεφάλαιο 5.

3.2 Κεντρικές Δομές Δεδομένων

Το Pintool που δημιουργήθηκε στο πλαίσιο της παρούσας εργασίας βασίζεται σε δύο κεντρικές δομές δεδομένων, το Region Map και το Thread Context (TLS).

Η πρώτη και σημαντικότερη δομή είναι το Region Map. Αποτελεί τον κεντρικό κατάλογο όλων των memory regions που δεσμεύτηκαν από την εφαρμογή κατά την εκτέλεση της και έχουν καταγραφεί. Κάθε εγγραφή αποθηκεύεται ως struct Region:

```
struct Region {
ADDRINT start; // αρχική διεύθυνση
size_t size;        // μέγεθος σε bytes
```

```
string tag;          // heap:malloc, heap:calloc,
mmap κλπ.
string alloc_file;   // source file του allocation
site
INT32 alloc_line;    // source line του allocation
site
};
```

Ο καθολικός κατάλογος ή αλλιώς χάρτης των regions ορίζεται ως:

```
static std::map<ADDRINT, Region> g_regions;
```

Η επιλογή std:map αντί για std:unordered_map δεν είναι τυχαία. Αυτή η δομή καλείται να λύσει το πρόβλημα όπου δεδομένης μιας διεύθυνσης μνήμης που προσπελαύνεται, πρέπει να βρεθεί ποιο region την περιέχει. Ένα region ξεκινά σε διεύθυνση start και εκτείνεται μέχρι start + size. Άρα για να γίνει το lookup δεν είναι μια απλή αναζήτηση με γνωστό κλειδί αλλά, πρέπει να γίνει range lookup για το ποιο region έχει start ≤ address < start + size. Η ταξινομημένη δομή του std:map επιτρέπει την εύρεση του κατάλληλου region μέσω upper_bound(address), η οποία επιστρέφει τον πρώτο κόμβο με start > address και άρα αρκεί μόνο να ελεγχθεί ο προηγούμενος κόμβος. Η λειτουργία αυτή με την χρήση upper_bound έχει πολυπλοκότητα O(log n) και καλείται για κάθε load/store access, οπότε η αποδοτικότητα είναι κρίσιμη.

Η δεύτερη δομή είναι το Thread Context (TLS – Thread Local Storage). Αυτό αποθηκεύει την per-thread κατάσταση που χρειάζεται το Pintool κατά την εκτέλεση. Κάθε thread έχει το δικό του ThreadCtx το οποίο περιέχει δεδομένα σχετικά με το thread, όπως role, OS thread ID, καθώς και τα λεγόμενα pending slots τα οποία αποθηκεύουν προσωρινά τα ορίσματα των allocators πριν επιστρέψουν. Το TLS είναι απαραίτητο γιατί το PIN δεν επιτρέπει πρόσβαση στα ορίσματα μιας συνάρτησης στο IPOINT_AFTER όπως το size, παρά μόνο στην επιστρεφόμενη διεύθυνση, δηλαδή τη διεύθυνση που επέστρεψε ο allocator. Άρα χωρίς TLS τα ορίσματα που δεν ήταν προσβάσημα από IPOINT_AFTER θα καταγράφονταν με λανθασμένη τιμή. Με TLS όμως τα ορίσματα αποθηκεύονται στο IPOINT_BEFORE μέσω TLS και χρησιμοποιούνται στο IPOINT_AFTER μαζί με την επιστρεφόμενη διεύθυνση.

```
// BEFORE: αποθήκευση ορισμάτων στο TLS
static VOID BeforeMallocTLS( THREADID tid, size_t sz,
ADDRINT caller_ip) {
    tc->hasPendingMalloc = true;
    tc->pendingMallocSize = sz;
    tc->pendingMallocCallerIp = caller_ip;
}

// AFTER: χρήση ορισμάτων + επιστρεφόμενη διεύθυνση
static VOID AfterMallocTLS(THREADID tid, ADDRINT ret) {
    AfterMalloc(ret,tc->pendingMallocSize, tc-
>pendingMallocCallerIp);
    tc->hasPendingMalloc = false;
}
```

Ιδιαίτερη προσοχή απαιτεί το calloc, για το οποίο χρησιμοποιείται stack pending slots αντί για single slot λόγω reentrant κλήσεων αφού η glibc καλεί εσωτερικά calloc κατά την αρχικοποίηση της. Αυτό θα είχε ως αποτέλεσμα ένα single slot να αντικαθίσταται από την εκκρεμή εγγραφή της reentrant κλήσης.

3.3 Runtime Parameters (Knobs)

Το Pintool παρέχει ένα σύνολο από παραμέτρους εκτέλεσης (knobs), οι οποίες ελέγχουν την συμπεριφορά του κατά την εκτέλεση χωρίς να είναι απαραίτητη η επαναμεταγλώττιση [21]. Κάθε knob επηρεάζει τόσο το μέγεθος και την ποιότητα του παραγόμενου trace όσο και την απόδοση της εφαρμογής ανάλογα με το πόσα queries per second (QPS) μπορεί να στείλει ο mcperf generator. Οι τιμές που επιλέχθηκαν για τα πειράματα της παρούσας εργασίας προέκυψαν από συστηματική σύγκριση διαμορφώσεων και αντιπροσωπεύουν τη βέλτιστη ισορροπία μεταξύ πληρότητας και ακρίβειας του trace.

use_libc_hooks

Το knob use_libc_hooks ελέγχει αν το Pintool εγκαθιστά hooks στις συναρτήσεις εκχώρησης μνήμης της libc όπως malloc, calloc, realloc, free κλπ. Είναι απαραίτητο να διευκρινιστεί ο τρόπος με τον οποίο λειτουργεί το dynamic linking, ώστε να γίνει κατανοητή η λειτουργία του αναφερόμενου knob. Όταν το Memcached καλεί malloc ή calloc, η κλήση δεν εκτελείται από κώδικα μέσα στο ίδιο το executable του Memcached. Αντίθετα, επιλύεται δυναμικά και εκτελείται από την libc.so [6], η οποία είναι η shared library της C standard library. Όταν το knob use_libc_hooks έχει τιμή 0 το Pintool αναζητά συναρτήσεις όπως malloc και calloc μόνο μέσα στο executable του Memcached αλλά οι συναρτήσεις αυτές δεν επιλύονται εσωτερικά. Οι πραγματικές malloc και calloc βρίσκονται στη libc και καλούνται μέσω dynamic linking [24]. Αυτό έχει ως αποτέλεσμα κανένα allocation να μην γίνεται hook, το Region Map παραμένει άδειο και το trace δεν παράγει καμία εγγραφή. Με τιμή 1 όμως, το Pintool εγκαθιστά hooks απευθείας στις συναρτήσεις της libc.so. Έτσι είναι σε θέση να καταγράψει όλα τα allocations ανεξαρτήτως από ποιο τμήμα κώδικα προέρχονται.

main_exe_only

Το knob main_exe_only ελέγχει για ποιες εντολές load/store το PIN εισάγει instrumentation. Δηλαδή, κατά το JIT compilation, το PIN αναλύει κάθε εντολή πριν την εκτελέσει και αποφασίζει αν θα εισάγει επιπλέον κώδικα που θα τρέχει μαζί με τον κύριο κώδικα για να κάνει κάποιου είδους ανάλυσης. Ο επιπλέον κώδικας που εισάγεται ονομάζεται callback και καλείτε κάθε φορά που το PIN συναντά την συγκεκριμένη εντολή. Στην περίπτωση μας, το PIN κάνει instrument για κάθε load/store και εισάγει τον κώδικα ανάλυσης RecordRead/RecordWrite αντίστοιχα έτσι ώστε να καταγράφονται οι προσβάσεις στο trace. Το knob main_exe_only με τιμή 1, αναγκάζει το PIN να κάνει instrument μόνο εντολές που ανήκουν στο main executable φιλτράροντας εντολές που ανήκουν σε shared libraries. Οι δύο κύριες libraries που εντοπίστηκαν και φιλτράρονται είναι η libevent, η οποία αφορά αποκλειστικά τον μηχανισμό δικτύου και event loop (~22 accesses/query) και η libpthread της οποίας οι εντολές αφορούν synchronization locks για τα item_locks (~30 accesses/query). Το συμπέρασμα είναι ότι καμιά από τις δύο δεν αγγίζει τα key-value data objects που

μελετήθηκαν στην παρούσα εργασία και άρα η απώλεια τους από το trace είναι αποδεκτή.

trace_libc_memops

Το knob trace_libc_memops εισήχθη ως λύση σε ένα πρόβλημα που προκύπτει από τον συνδυασμό των main_exe_only και use_libc_hooks. Συγκεκριμένα, με main_exe_only=1, οι εντολές πρόσβασης load/store των shared libraries δεν γίνονται instrument συμπεριλαμβανομένης και της libc. Αυτό σημαίνει ότι όταν το Memcached δέχεται ένα query και εξυπηρετεί ένα GET request, η memcpy της libc που εκτελεί την αντιγραφεί των δεδομένων δεν γίνεται instrument ποτέ και οι προσβάσεις τις δεν καταγράφονται στο trace. Η ανάγκη της memcpy προέρχεται από όταν το Memcached δέχεται GET request και εντοπίζει το αντίστοιχο item στον slab page ώστε να στείλει την τιμή του πίσω στον client μέσω network. Για να γίνει όμως αυτό, καλείται η memcpy της libc η οποία αντιγράφει τα 200 bytes value από το slab page, όπου το item είναι αποθηκευμένο, σε ένα response buffer από τον οποίο θα σταλούν μέσω socket. Η memcpy εκτελεί αυτή την αντιγραφή μέσω εντολών load/store, όπου διαβάζει (load) chunk bytes από την πηγή και τα γράφει (store) στον προορισμό. Χωρίς να γίνεται το σωστό instrumentation αυτών των εντολών χάνονται ~134 accesses/query που αντιστοιχούν ακριβώς σε αυτές τις προσβάσεις στα item values.

Μια υποψήφια λύση στο πρόβλημα αυτό θα ήταν να οριστεί το main_exe_only σε 0, ώστε να γίνονται instrument όλες οι libraries. Αυτό όμως απορρίφθηκε στην πορεία γιατί εισάγει ~22 accesses/query θόρυβο από libevent internals και libpthread synchronization τα οποία παραμορφώνουν τα αποτελέσματα.

Στην συνέχεια διαπιστώθηκε ότι ο ορισμός του trace_libc_memops=1 αποτελεί την βέλτιστη λύση αφού διατηρεί το main_exe_only=1 για να αποκλείει τον θόρυβο από libevent και libpthread, αλλά εξαιρείται από τον αποκλεισμό η libc. Ωστόσο, ακόμα και με trace_libc_memops=1, ο callback RecordRead/RecordWrite καταγράφει ένα access της libc μόνο αν η διεύθυνση που προσπελαύνεται ανήκει σε tracked region, δηλαδή σε region που έχει ήδη καταγραφεί μέσω allocation. Με αυτό το τρόπο αποφεύγεται η καταγραφή των internals της libc και καταγράφονται μόνο οι προσβάσεις της σε

application data. Αυτή η μέθοδος έχει ως αποτέλεσμα την αύξηση των accesses/query από 257 σε 381, με κόστος ~33% μείωση QPS (175.4 πριν, 120.8 μετά) λόγω της επιπλέον instrumentation των εντολών της libc.

worker_only

Το knob worker_only ελέγχει κατά πόσο το Pintool καταγράφει προσβάσεις για όλα τα threads της εφαρμογής ή μόνο για το worker thread. Πιο συγκεκριμένα το Memcached έχει εκτελεστεί με παράμετρο -t 1, η οποία σηματοδοτεί ότι η εφαρμογή θα τρέξει με 1 worker thread. Το working thread βρίσκεται στο critical path και στην ουσία εκτελεί την πραγματική εργασία του Memcached. Η αρχιτεκτονική των threads είναι δομημένη έτσι ώστε ο Dispatcher thread δέχεται νέες TCP connections και τις αναθέτει στον worker thread μέσω connection queue. Ο ίδιος δεν επεξεργάζεται requests και δεν αγγίζει τα key-value data objects, δηλαδή δεν έρχεται σε επαφή με τα πραγματικά δεδομένα. Ο worker thread τώρα επεξεργάζεται κάθε GET request, εκτελεί hash lookup για να βρει το key, διαβάζει item από το slab page και στέλνει την απάντηση πίσω στον client. Στην ουσία είναι το μόνο thread που αγγίζει τα data objects που μελετήθηκαν στην παρούσα εργασία. Τα background threads εκτελούν εργασίες υποδομής όπως LRU eviction και hash table maintenance. Αυτά αγγίζουν infrastructure objects αλλά όχι τα user data. Τα infrastructure objects είναι αυτά που χρησιμοποιούνται για την διαχείριση και λειτουργία του Memcached ως σύστημα και άρα δεν περιέχουν user data.

Με worker_only=0 το Pintool θα κατέγραφε accesses και από τα άλλα threads, με αποτέλεσμα να εισάγεται θόρυβος από accesses σε infrastructure objects που δεν σχετίζονται με την επεξεργασία των queries.

Για να επιτευχθεί το φιλτράρισμα αυτό ώστε να αφαιρεθεί ο θόρυβος και να επικεντρωθεί η μελέτη στον worker thread, το Pintool χρειάζεται την πληροφορία αναγνώρισης του worker thread. Η αναγνώριση αυτή δεν είναι τετριμμένη, το PIN δεν γνωρίζει εκ των προτέρων τους ρόλους των threads μιας εφαρμογής. Η λύση σε αυτό το πρόβλημα βρέθηκε μέσω ανάλυσης του πηγαίου κώδικα του Memcached. Ο worker στο Memcached δημιουργείται από τον dispatcher και ξεκινά την εκτέλεσή του καλώντας τη συνάρτηση worker_libevent. Αυτή η συνάρτηση είναι το entry point του worker

thread, δηλαδή είναι η πρώτη συνάρτηση που εκτελεί το thread και τρέχει για όλη την διάρκεια της ζωής του, επεξεργαζόμενη συνεχώς τα εισερχόμενα requests μέσω του event loop της libevent. Αυτό σημαίνει ότι κάθε thread που εκτελεί worker_libevent είναι εξ ορισμού worker thread.

Με βάση αυτή τη γνώση, δημιουργήθηκε η συνάρτηση HookMemcachedThreadRoles με σκοπό την αναγνώριση του worker thread. Συγκεκριμένα, εγκαθιστά hook στην είσοδο της worker_libevent μέσω IPOINT_BEFORE [21] το οποίο καθορίζει το κάλεσμα του callback αμέσως πριν την εκτέλεση της εντολής.

```
static VOID HookMemcachedThreadRoles(IMG img) {
     if (!IMG_IsMainExecutable(img)) return;
     RTN r = RTN_FindByName(img, "worker_libevent");
     RTN_InsertCall(r, IPOINT_BEFORE,
     AFUNPTR(MarkWorkerThread),
     IARG_THREAD_ID, IARG_END);
}
```

Στον κώδικα της παραπάνω συνάρτησης, η RTN_FindByName είναι PIN API call [21] που ψάχνει μέσα στο image του main executable, δηλαδή στο φορτωμένο Memcached binary στην μνήμη, για συνάρτηση με όνομα «worker_libevent». Όταν την βρει επιστρέφει ένα RTN handle το οποίο είναι στην ουσία ένα αναγνωριστικό που αντιπροσωπεύει την συνάρτηση στο PIN και επιτρέπει την εγκατάσταση callbacks σε αυτή. Στην συνέχεια η RTN_InsertCall είναι και αυτή ένα PIN API call [21] που χρησιμοποιεί το RTN handle για να εγκαταστήσει την callback συνάρτηση MarkWorkerThread στην είσοδο της worker_libevent με IPOINT_BEFORE. Έτσι κάθε φορά που οποιοδήποτε thread ξεκινά την εκτέλεση της worker_libevent, το PIN καλεί αυτόματα την MarkWorkerThread πριν εκτελεστεί οποιαδήποτε εντολή της συνάρτησης, ώστε να ορίσει το tc->role = ROLE_WORKER στο TLS του thread αυτού σημαδεύοντας το μόνιμα ως worker για όλη την διάρκεια εκτέλεσης του προγράμματος. Τα υπόλοιπα threads είναι ορισμένα ως ROLE_OTHER.

trace_stack

Το knob trace_stack χρησιμοποιείται για να ελέγχει την καταγραφή προσβάσεων σε περιοχές στοίβας. Κάθε thread διαθέτει δική του περιοχή στοίβας στην μνήμη η οποία χρησιμοποιείται για αποθήκευση τοπικών μεταβλητών, arguments συναρτήσεων και δεδομένων εκτέλεσης. Στο Memcached, το stack του worker thread δεσμεύεται από glibc κατά τη δημιουργία του thread μέσω mmap με allocation site allocatestack.c:373. Το region του stack καταχωρείται στο map με tag stack:pthread και άρα με trace_stack=1 θα καταγραφούν όλες οι προσβάσεις στη στοίβα μαζί και του worker thread.

Το πρόβλημα με το knob αυτό να έχει τιμή 1 είναι ότι τα accesses αυτά που καταγράφονται δεν αφορούν τα key-value data objects που μελετά η εργασία αυτή, αλλά αφορούν εσωτερικά δεδομένα εκτέλεσης του thread όπως τοπικές μεταβλητές και ορίσματα συναρτήσεων. Συγκεκριμένα παρατηρήθηκε ότι καταγράφεται ένα τεράστιο ποσοστό προσβάσεων που αφορούν εσωτερικά δεδομένα και αποτελούν θόρυβο. Έτσι, για να φιλτραριστούν αυτά τα accesses το Pintool ελέγχει στο hot path του RecordRead και RecordWrite αν το Region έχει tag stack:pthread, αν έχει αγνοεί εντελώς την πρόσβαση. Αυτό ελέγχεται εξολοκλήρου από το knob.

```
if (found && snap.tag == "stack:pthread" &&
    !KnobTraceStack.Value()) return;
```

trace_globals

Το knob trace_globals ελέγχει αν γίνεται καταγραφή των global sections του main executable ως regions στο map και αν γίνεται καταγραφή προσβάσεων σε αυτά. Τα global sections είναι τμήματα του binary και περιέχουν στατικά δεδομένα. Συγκεκριμένα, το .data section περιέχει καθολικές και στατικές μεταβλητές που αρχικοποιούνται με 0 και το .got section περιέχει τον πίνακα καθολικών offset για dynamic linking, δηλαδή κρατά τις διευθύνσεις συναρτήσεων και μεταβλητών από shared libraries. Στην περίπτωση του Memcached, αυτά τα sections περιέχουν κυρίως στατικές μεταβλητές υποδομής όπως global counters, configuration flags και pointers

που δεν σχετίζονται με τα heap objects που μελετά η παρούσα εργασία. Η συνάρτηση που καταχωρεί κάθε non-executable section ως region στο Region Map είναι η TrackImageGlobals και καλείται κατά τη φόρτωση του image. Με trace_globals=0, η συνάρτηση επιστρέφει αμέσως χωρίς να καταχωρεί τίποτα, αποτρέποντας την καταγραφή accesses σε αυτά τα sections.

trace_untracked

Το knob trace_untracked κυρίως δημιουργήθηκε για debug αφού ελέγχει κατά πόσο καταγράφονται προσβάσεις σε διευθύνσεις που δεν ανήκουν σε κανένα tracked region του Region map, δηλαδή διευθύνσεις για τις οποίες το FindRegionWithOff η οποία κάνει το region lookup, δεν βρίσκει αντίστοιχο region. Αυτό συμβαίνει μόνο σε περίπτωση όπου μια εντολή προσπελαύνει μνήμη που δεν δεσμεύτηκε μέσω κάποιου από τους allocators για τους οποίους έχουμε hook, όπως για παράδειγμα στατικά buffers που δεν δεσμεύονται κατά compile time. Με trace_untracked=1 αυτά τα accesses καταγράφονται ως UNTRACKED στο trace, προσθέτοντας θόρυβο χωρίς χρήσιμη πληροφορία για την ανάλυση την παρούσα μελέτη.

src_debug

Το src_debug είναι επίσης χρήσιμο μόνο για debugging. Αυτό που κάνει είναι να ενεργοποιεί verbose diagnostics για την διαδικασία εύρεσης source locations κάθε allocation site. Είναι χρήσιμο μόνο κατά την ανάπτυξη και αποσφαλμάτωση του tool, με σημαντικό κόστος απόδοσης.

Knob	*Περιγραφή*
use_libc_hooks	Εγκαθιστά hooks στις συναρτήσεις εκχώρησης μνήμης της libc (malloc, calloc, realloc, free). Απαραίτητο για την καταγραφή allocations αφού οι συναρτήσεις αυτές επιλύονται δυναμικά από τη libc.so και όχι από το main executable.
main_exe_only	Περιορίζει την instrumentation των load/store εντολών αποκλειστικά στο main executable, φιλτράροντας shared libraries (libevent: ~22 accesses/query, libpthread: ~30 accesses/query) που δεν αγγίζουν τα key-value data objects.
trace_libc_memops	Εξαιρεί τη libc από τον αποκλεισμό του main_exe_only, επιτρέποντας την καταγραφή προσβάσεων της libc (π.χ.

	memcpy) σε tracked regions. Ανακτά ~134 accesses/query από αντιγραφή item values με κόστος ~33% μείωση QPS.
worker_only	Περιορίζει την καταγραφή αποκλειστικά στον worker thread, ο οποίος αναγνωρίζεται μέσω hook στη συνάρτηση `worker_libevent`. Αποκλείει dispatcher και background threads που δεν επεξεργάζονται user data.
trace_stack	Απενεργοποιεί την καταγραφή προσβάσεων στη στοίβα (tag: stack:pthread). Οι προσβάσεις αυτές αφορούν εσωτερικά δεδομένα εκτέλεσης (τοπικές μεταβλητές, ορίσματα) και αποτελούν θόρυβο για την παρούσα ανάλυση.
trace_globals	Απενεργοποιεί την καταγραφή global sections του binary (.data, .got). Περιέχουν στατικές μεταβλητές υποδομής που δεν σχετίζονται με τα heap objects που μελετώνται.
trace_untracked	Απενεργοποιεί την καταγραφή προσβάσεων σε διευθύνσεις που δεν ανήκουν σε κανένα tracked region. Χρήσιμο μόνο για debugging — με τιμή 1 εισάγει θόρυβο χωρίς χρήσιμη πληροφορία.
src_debug	Απενεργοποιεί verbose diagnostics για την εύρεση source locations κάθε allocation site. Χρήσιμο μόνο κατά την ανάπτυξη του tool με σημαντικό κόστος απόδοσης.

Πίνακας 3.1: Runtime parameters (knobs)

3.4 Παρακολούθηση Memory Allocations

3.4.1 BEFORE/AFTER Pattern

Το Pintool που δημιουργήθηκε για να καταγράψει ένα allocation event χρειάζεται δύο βασικές πληροφορίες, πρώτον την αρχική διεύθυνση της περιοχής μνήμης που δεσμεύεται και το μέγεθος του. Η αρχική διεύθυνση είναι η επιστρεφόμενη τιμή του allocator, για παράδειγμα η malloc επιστρέφει την διεύθυνση της μνήμης που δέσμευσε. Το μέγεθος είναι το όρισμα που πέρασε ο κώδικας στον allocator, όπως για παράδειγμα το sz στην malloc(sz).

Το πρόβλημα που δημιουργείται είναι ότι αυτές οι δύο πληροφορίες δεν είναι διαθέσιμες ταυτόχρονα σε κανένα σημείο εκτέλεσης. Στο IPOINT_BEFORE, δηλαδή αμέσως πριν εκτελεστεί ο κώδικας του allocator, τα ορίσματα είναι διαθέσιμα αλλά η επιστρεφόμενη διεύθυνση δεν έχει ακόμα παραχθεί. Στο IPOINT_AFTER, δηλαδή αμέσως μετά την επιστροφή, η διεύθυνση είναι διαθέσιμη αλλά το PIN δεν επιτρέπει

πρόσβαση στα ορίσματα του allocator [21][5]. Άρα χωρίς την ύπαρξη του επιπλέον μηχανισμού με IPOINT_BEFORE, το μέγεθος του region δεν είναι προσβάσιμο και κατά συνέπεια άγνωστο στο IPOINT_AFTER, με αποτέλεσμα την καταγραφή λανθασμένων τιμών. Αυτό ακριβώς το φαινόμενο παρατηρήθηκε αρχικά για regions τα οποία λόγο άγνωστου size καταγράφονταν με size=0 ή με λανθασμένη μη μηδενική τιμή.

Η λύση αυτού είναι το BEFORE/AFTER pattern όπου τα ορίσματα αποθηκεύονται προσωρινά στο TLS του thread στο IPOINT_BEFORE και χρησιμοποιούνται στη συνέχεια στο IPOINT_AFTER μαζί με την επιστρεφόμενη διεύθυνση.

```
// IPOINT_BEFORE: αποθήκευση ορισμάτων στο TLS
static VOID BeforeMalloc(THREADID tid, size_t sz, ADDRINT
caller_ip) {
        ThreadCtx* tc = SafeCTX(tid, "BeforeMalloc");
        if (!tc) return;
        tc->hasPendingMalloc = true;
        tc->pendingMallocSize = sz;
        tc->pendingMallocCallerIp = caller_ip;
}

// IPOINT_AFTER: χρήση ορισμάτων + επιστρεφόμενη διεύθυνση
static VOID AfterMalloc(THREADID tid, ADDRINT ret) {
        ThreadCtx* tc = SafeCTX(tid, "AfterMalloc");
        if (!tc || !tc->hasPendingMalloc) return;

        // Καταγραφή region με ret + pendingMallocSize
        tc->hasPendingMalloc = false;
}
```

Το BEFORE/AFTER pattern εφαρμόζεται για τους allocators που υποστηρίζουν απλή hook εγκατάσταση μέσω RTN_InsertCall, όπως malloc, mmap, munmap και strdup. Για

calloc και realloc allocators απαιτείται διαφορετική προσέγγιση λόγω ειδικών απαιτήσεων που αναλύονται πιο κάτω στο 3.4.3.

3.4.2 Ποιοι Allocators γίνονται Hook και Γιατί

Για να είναι δυνατή η καταγραφή των memory allocations της εφαρμογής του Memcached, το Pintool πρέπει να εγκαταστήσει hooks σε ένα ευρύ σύνολο allocators. Η επιλογή των allocators που γίνονται hook βασίζεται σε δύο βασικά κριτήρια. Το πρώτο κριτήριο είναι η ανάγκη να καλύπτονται όλοι οι τρόποι με τους οποίους το Memcached δεσμεύει και αποδεσμεύει μνήμη. Το δεύτερο, είναι επίσης η ανάγκη να αποφεύγεται η εγκατάσταση hooks σε allocators άλλων libraries που θα εισήγαγαν θόρυβο. Για τους λόγους αυτούς, τα hooks εγκαθίστανται αποκλειστικά σε συναρτήσεις της libc, jemalloc και tcmalloc [13].

Μετά από δοκιμές προστέθηκαν αρκετά hooks ώστε να καλύπτονται οι πιο πιθανοί allocators [13] οι οποίοι οργανώνονται στις παρακάτω κατηγορίες:

Κατηγορία	*Allocator*	*Περιγραφή*
Heap Allocators	malloc, calloc, realloc, reallocarray, aligned_alloc, memalign, valloc, pvalloc, posix_memalign	Καλύπτουν όλες τις μορφές δυναμικής εκχώρησης μνήμης από το heap.
Deallocators	free, cfree	Καλύπτουν την αποδέσμευση heap memory.
Memory mapped allocators	mmap, mmap64, munmap, mremap	Καλύπτουν εκχωρήσεις μνήμης μέσω memory mapping, που χρησιμοποιούνται τόσο από τον glibc allocator εσωτερικά για μεγάλα allocations όσο και απευθείας από το Memcached για τα slab pages.
String Allocators	Strdup, strndup	Καλύπτουν εκχωρήσεις για αντίγραφα strings.
Jemalloc	Je_malloc, je_free,	Καλύπτουν την περίπτωση που το

	je_calloc, je_realloc	Memcached έχει μεταγλωττιστεί με jemalloc αντί για τον default glibc allocator.

Πίνακας 3.2: Allocators που έγιναν hooked

Ένα πρακτικό αλλά σημαντικό ζήτημα που έπρεπε να αντιμετωπιστεί είναι ότι κάθε allocator, όπως malloc, εμφανίζεται υπό πολλαπλά ονόματα στη libc. Αυτές οι διαφορετικές εκδοχές ονοματολογίας των allocators ονομάζεται aliases και υλοποιούν την ίδια συνάρτηση. Για malloc που αναφέρθηκε παραπάνω, εμφανίζεται υπό τα εξής aliases:

Malloc	Το δημόσιο όνομα που χρησιμοποιεί ο προγραμματιστής.
__libc_malloc	Το εσωτερικό όνομα που χρησιμοποιεί η libc όταν καλεί malloc από άλλες συναρτήσεις της.
__GI___libc_malloc	Χρησιμοποιεί το πρόθεμα __GI_ (Global Internal) για να εξασφαλίσει ότι η libc καλεί τη δική της υλοποίηση και όχι κάποια εξωτερικά ορισμένη παραλλαγή [15].
__GI_malloc	Συντόμευση για το παραπάνω [15].

Πίνακας 3.3: Malloc aliases

Όλα αυτά δείχνουν στον ίδιο εκτελέσιμο κώδικα, απλά εκτίθενται υπό διαφορετικά ονόματα στον symbol table της libc.so. Αν δεν γίνουν hook σωστά όλα τα aliases, ορισμένα allocations θα διέφευγαν της καταγραφής. Για τον λόγο αυτό πρέπει να γίνουν hook ρητά όλα τα γνωστά aliases κάθε allocator.

3.4.3 RTN_InsertCall vs RTN_ReplaceSignature μέθοδοι

Για να γίνει η εγκατάσταση των hooks στους allocators, έγινε η χρήση δύο διαφορετικών PIN API κλήσεων στο Pintool, ανάλογα με τον allocator. Οι κλήσεις που χρησιμοποιήθηκαν είναι η RTN_InsertCall και RTN_ReplaceSignature [21].

Η RTN_InsertCall χρησιμοποιήθηκε για απλούς allocators όπως malloc, mmap, munmap και strdup. Σε αυτή την προσέγγιση γίνεται χρήση του BEFORE/AFTER pattern αυτούσια όπως περιγράφηκε στο 3.4.1. Πιο συγκεκριμένα, εγκαθίσταται ένα callback στο IPOINT_BEFORE για να γίνει η αποθήκευση των ορισμάτων και ένα στο IPOINT_AFTER για την καταγραφή της επιστρεφόμενης διεύθυνσης του allocation, δηλαδή τη διεύθυνση μνήμης του region που δεσμεύτηκε [21].

Για τους allocators calloc και realloc διαπιστώθηκε η ανάγκη για χρήση διαφορετικής προσέγγισης μέσω RTN_ReplaceSignature. Η RTN_ReplaceSignature είναι ένα PIN API call που κάνει αντικατάσταση ολόκληρης της υλοποίησης μιας συνάρτησης με κώδικα που ορίζει ο χρήστης [21]. Σε αντίθεση με την RTN_InsertCall που απλά εισάγει callbacks πριν ή μετά τη συνάρτηση και αφήνει την αρχική υλοποίηση να εκτελεστεί κανονικά, η RTN_ReplaceSignature αναλαμβάνει πλήρως τον έλεγχο και η αρχική υλοποίηση καλείται μόνο αν ο χρήστης το επιλέξει ρητά μέσω PIN_CallApplicationFunction. Η PIN_CallApplicationFunction είναι PIN API call που δίνει την δυνατότητα στο Pintool να καλέσει την αρχική υλοποίηση μιας συνάρτησης που αντικαταστάθηκε [21]. Σε περίπτωση που δεν γινόταν χρήση της συνάρτησης αυτής, η αρχική λειτουργία του allocator δεν θα εκτελούταν ποτέ. Το Pintool πρέπει να την καλέσει ρητά περνώντας τα ορίσματα και λαμβάνοντας την επιστρεφόμενη τιμή. Οι λόγοι που απαιτείται αυτή η προσέγγιση για calloc και realloc είναι διαφορετικοί.

Αρχικά, για calloc αντιμετωπίζονται τα εξής:

Το πρώτο πρόβλημα είναι το ίδιο με όλους τους allocators, όπου δεν είναι δυνατή η πρόσβαση στο size στο IPOINT_AFTER. Ωστόσο για calloc υπάρχουν επιπλέον δύο προβλήματα που καθιστούν αδύνατη την απλή χρήση RTN_InsertCall.

Το δεύτερο πρόβλημα είναι οι reentrant κλήσεις που γίνονται καθώς η glibc καλεί εσωτερικά calloc κατά την αρχικοποίηση της [13], πριν ακόμα ξεκινήσει η κανονική εκτέλεση της εφαρμογής. Αυτό σημαίνει ότι μπορεί να γίνει μια reentrant κλήση calloc ενώ μια άλλη βρίσκεται σε εκκρεμή κατάσταση στο TLS και με ένα single slot θα γινόταν αντικατάσταση της εκκρεμές εγγραφής με αποτέλεσμα να δημιουργούνταν λανθασμένα regions και μερικά θα αγνοούνταν. Η λύση έρχεται με την χρήση του stack αντί για single slot στο TLS.

```
struct PendingCalloc {
     size_t n, sz;
     ADDRINT callerIp;
     UINT64 seq;
};
static constexpr int kCallocStackMax = 64;
PendingCalloc
pendingCallocStack[kCallocStackMax];
int pendingCallocTop;
UINT64 callocSeq;
```

Ο callocSeq είναι ένας μονοτονικά αυξανόμενος μετρητής, ο οποίος δηλαδή μόνο αυξάνεται και ποτέ δεν μειώνεται, που αποδίδει μοναδικό αναγνωριστικό(identifier) σε κάθε κλήση calloc. Κατά το IPOINT_BEFORE, γίνεται αποθήκευση του identifier στο stack όπως κάθε άλλο όρισμα. Στο IPOINT_AFTER, γίνεται χρήση του identifier για να αντιστοιχιστεί η επιστρεφόμενη διεύθυνση με τα σωστά ορίσματα που βρίσκονται στο stack, ακόμα και αν πολλαπλές reentrant κλήσεις είναι ταυτόχρονα σε εκκρεμή κατάσταση.

Το τρίτο πρόβλημα αφορά τον τρόπο που η glibc εκθέτει το public σύμβολο της calloc. Αυτό το πρόβλημα προκαλείται γιατί, όταν ένα πρόγραμμα καλεί μια συνάρτηση από shared library, η κλήση της συνάρτησης δεν πηγαίνει απευθείας στην υλοποίηση της. Ο πρώτος σταθμός της κλήσης είναι στο PLT stub, ένα μικρό κομμάτι κώδικα που είναι υπεύθυνο για την εύρεση της πραγματικής διεύθυνσης της συνάρτησης μέσω του GOT (Global Offset Table) [6]. Το PLT stub δηλαδή δεν περιέχει τον πραγματικό κώδικα της

συνάρτησης του calloc αλλά κάνει jump σε αυτή αφού βρει την διεύθυνση της πραγματικής υλοποίησης από τον GOT.

Το public σύμβολο calloc εκτίθεται μέσω PLT stub στην glibc [15]. Όταν το PIN εφαρμόζει RTN_InsertCall στο calloc, γίνεται hook το PLT stub και όχι η πραγματική υλοποίηση του calloc. Αυτό έχει ως αποτέλεσμα να μην είναι αξιόπιστα τα προσβάσημα από το callback ορίσματα και η επιστρεφόμενη τιμή. Άρα ακόμα και αν το BEFORE/AFTER pattern εφαρμοστεί, τα ορίσματα που θα αποθηκευτούν στο TLS θα είναι λανθασμένα ή απροσδιόριστα [21].

Αντίθετα, τα aliases __libc_calloc και __GI___libc_calloc εκτίθενται μέσω PLT stub, δείχνουν απευθείας στην πραγματική υλοποίηση [15]. Για αυτό τον λόγο τα aliases μπορούν να γίνουν hook αξιόπιστα μέσω RTN_InsertCall με BEFORE/AFTER pattern. Από την άλλη το public calloc απαιτεί RTN_ReplaceSignature, η οποία αντικαθιστά όλη την συνάρτηση αντί να εισάγει callbacks στο PLT stub και έτσι εξασφαλίζεται πρόσβαση στα πραγματικά ορίσματα και την επιστρεφόμενη τιμή [21].

```
static VOID* Calloc_Replacement(CONTEXT* ctxt, THREADID
tid,
    AFUNPTR orig, size_t n, size_t sz, ADDRINT caller_ip)
{
    VOID* ret = nullptr;
    tc->inCalloc++;
    PIN_CallApplicationFunction(ctxt, tid,
CALLINGSTD_DEFAULT, orig, nullptr,
        PIN_PARG(void*), &ret,
        PIN_PARG(size_t), n,
        PIN_PARG(size_t), sz,
        PIN_PARG_END());
    AfterCalloc((ADDRINT)ret, n, sz, caller_ip);
    tc->inCalloc--;
    return ret;
}
```

Ο counter inCalloc χρησιμοποιείται για να ανιχνεύονται οι reentrant κλήσεις — αν inCalloc > 1 κατά την είσοδο σε Calloc_Replacement, πρόκειται για reentrant κλήση από τα internals της glibc και αντιμετωπίζεται αναλόγως.

Για το realloc,

Εδώ επίσης χρησιμοποιείται RTN_ReplaceSignature αλλά για διαφορετικό λόγο. Το realloc εκτελεί δύο διαφορετικές πράξεις ατομικά. Εκτελεί ένα free για το παλιό region και ένα alloc για το νέο region που δημιουργείται με το νέο μέγεθος. Αν γινόταν χρήση του RTN_InsertCall και BEFORE/AFTER pattern δεν θα ήταν δυνατό να εγγυηθεί η καταγραφή των δύο πράξεων ατομικά. Με RTN_ReplaceSignature η ατομικότητα εξασφαλίζεται γιατί ο έλεγχος παραμένει στο Pintool για όλη τη διάρκεια της κλήσης.

```
static VOID* Realloc_Replacement(CONTEXT* ctxt, THREADID
tid,
    AFUNPTR orig, ADDRINT oldp, size_t sz, ADDRINT
caller_ip)
{
    VOID* ret = nullptr;
    tc->inRealloc++;
    PIN_CallApplicationFunction(...);
    AfterRealloc((ADDRINT)ret, oldp, sz, caller_ip); //
FREE(old) + ALLOC(new)
    tc->inRealloc--;
    return ret;
}
```

Ο counter inRealloc αποτρέπει ψευδή AfterMalloc events από εσωτερικές malloc κλήσεις που μπορεί να γίνουν στο realloc.

3.4.4 Source Location Resolution

Για κάθε allocation event που καταγράφεται το Pintool πρέπει να προσδιορίσει το αρχείο του πηγαίου κώδικα και την γραμμή από την οποία προέρχεται η κλήση του allocator, αυτό ονομάζεται allocation site. Η πληροφορία αυτή είναι κρίσιμη για την κατηγοριοποίηση των memory objects σε persistent και volatile, όπως περιγράφεται στο Κεφάλαιο 4.

Από το PIN είναι προσβάσιμο το caller IP της κλήσης, μέσω του IARG_RETURN_IP argument στο IPOINT_BEFORE. Το IARG_RETURN_IP στο PIN αντιστοιχεί στο return address που είναι η διεύθυνση της επόμενης εντολής μετά το call instruction [21] [26]. Άρα το caller_ip που αποθηκεύεται στο TLS δεν είναι η διεύθυνση του ίδιου του call instruction, αλλά η διεύθυνση αμέσως μετά από αυτό. Για να βρεθεί η σωστή γραμμή κώδικα χρειάζεται η διεύθυνση IP-1, δηλαδή η αμέσως προηγούμενη διεύθυνση στο call instruction.

Για να αντιμετωπιστούν αυτά τα προβλήματα υιοθετήθηκε μια 4-tier προσέγγιση η οποία δοκιμάζει τέσσερις μεθόδους μέχρι να βρεθεί έγκυρο αποτέλεσμα.

```
static std::string ResolveSourceLocation(ADDRINT ip, const
std::string& imgName, ADDRINT imgOff) {
    // Tier 1: PIN_GetSourceLocation στο exact IP (= return
address)
    std::string file; INT32 line = 0;
    PIN_GetSourceLocation(ip, nullptr, &line, &file);
    if (!file.empty() && line > 0)
        return file + ":" + std::to_string(line);

    // Tier 2: PIN_GetSourceLocation στο IP-1 (= μέσα στο
call instruction)
    PIN_GetSourceLocation(ip - 1, nullptr, &line, &file);
    if (!file.empty() && line > 0)
        return file + ":" + std::to_string(line);
```

```
    // Tier 3: addr2line στο exact offset
    std::string result = RunAddr2line(imgName, imgOff);
    if (!result.empty()) return result;

    // Tier 4: addr2line στο IP-1 offset
    result = RunAddr2line(imgName, imgOff - 1);
    if (!result.empty()) return result;

    return "UNKNOWN:0";
}
```

Το Tier 1 κάνει χρήση του PIN API PIN_GetSourceLocation στο exact IP (return address) [21]. Αυτό μερικές φορές είναι αρκετό για να επιστραφεί η σωστή γραμμή, όπου τα debug symbols είναι πλήρη και προσβάσημα. Το Tier 2 εφαρμόζει την ίδια μέθοδο στο IP-1, δηλαδή μέσα στο call instruction. Αυτό είναι το πιο συνηθισμένο fallback γιατί το return address συχνά δεν αντιστοιχεί άμεσα σε debug symbol [21]. Το Tier 3, αυτό που κάνει είναι να καλεί εξωτερικά το εργαλείο addr2line [12] με το image-relative offset του IP, ως fallback όταν το PIN API αποτυγχάνει εντελώς. Και τέλος το Tier 4 συνδυάζει addr2line με IP-1 offset. Αν αποτύχουν και τα τέσσερα στάδια τότε το allocation site καταγράφεται ως UNKNOWN:0 και το memory object δεν μπορεί να κατηγοριοποιηθεί και γι' αυτό αντιμετωπίζεται ως volatile.

Ωστόσο, η χρήση addr2line επαναλαμβανόμενα ρίχνει σημαντικά την απόδοση του συστήματος γιατί είναι μια χρονοβόρα εργασία. Για αυτό τον λόγο δημιουργήθηκε ο g_src_cache, ο οποίος είναι ένας unordered map που αποθηκεύει τα αποτελέσματα της resolution για κάθε IP, ώστε να γίνεται επίλυση κάθε διεύθυνσης μια φόρα. Στην πράξη, ο αριθμός των μοναδικών allocation site είναι πεπερασμένος και όχι ιδιαίτερα μεγάλος. Αυτό σε συνδυασμό με το cache για τα allocation sites βοηθούν σημαντικά στην πολυπλοκότητα του μηχανισμού.

3.4.5 Tag Assignment

Για κάθε region που καταχωρείται στο Region Map, καταγράφεται και ένα tag. Το tag είναι ένα string το οποίο περιγράφει τον τύπο και την προέλευση του region. Χρησιμοποιείται αποκλειστικά για το φιλτράρισμα των accesses στο hot path. Για παράδειγμα, regions με tag stack:pthread ή mmap:ldlinux φιλτράρονται και δεν καταγράφονται στο trace. Η απόδοση του tag γίνεται κατά την καταγραφή του alloc event και είναι βασισμένη στον τύπο του allocator και στο allocation site.

Τα tags οργανώνονται στις εξής κατηγορίες:

Heap tags, με βάση τον allocator που χρησιμοποιήθηκε	
heap:malloc	regions που δεσμεύτηκαν μέσω malloc, strdup, aligned_alloc κ.λπ.
heap:calloc	regions που δεσμεύτηκαν μέσω calloc
heap:realloc	regions που δεσμεύτηκαν μέσω realloc ή reallocarray
mmap tags, με βάση το **source file που αντιστοιχεί στο caller_ip** του mmap call	
mmap	κανονικά mmap regions από την εφαρμογή ή τη libc
mmap:ldlinux	regions που δεσμεύτηκαν από τον dynamic linker
stack:pthread	stack regions των threads που δεσμεύτηκαν από τη glibc
Global tags, αποδίδονται από τη TrackImageGlobals βάσει του ονόματος του section	
global:data	.data section
global:bss	.bss section
global:<sname>	οποιοδήποτε άλλο non-executable section (π.χ .got, .rodata κ.λπ.)

Πίνακας 3.4: Κατηγορίες των tags

3.4.6 Καταγραφή ALLOC/FREE Events

Μετά την επιστροφή του allocator, αφού το Pintool αποκτήσει πληροφορία για το μέγεθος και την επιστρεφόμενη διεύθυνση μέσω του BEFORE/AFTER pattern που περιγράφηκε στο 3.4.1, εκτελούνται ατομικά δύο πράξεις. Πρώτον, δημιουργείται μια νέα εγγραφή τύπου Region με τα πεδία start, size, tag, alloc_file και alloc_line και

εισάγεται στο g_regions map κάτω από g_regions_lock κλείδωμα. Μετά, καλείται η PrintAlloc η οποία καταγράφει στο trace μια γραμμή μορφής:

```
T0 [ALLOC] Region_start=0x7fb544018e80 Size=256
Type=heap:calloc Caller=/memcached-1.6.14/cache.c:84
Caller_IMG=memcached+0x1a2b4 Caller_PC=0x55a2bd4252b4
```

Το πεδίο type αντιστοιχεί στο tag του region, το Caller στο allocation site (file:line) που προσδιορίστηκε μέσω του 4-tier source location resolution του 3.4.4, και το Caller_IMG στο image-relative offset της εντολής κλήσης.

Είναι επίσης σημαντικό να αναφερθεί ότι καθώς το Memcached εκτελείται σε multi-thread περιβάλλον, τα threads μπορούν να καλούν allocators ταυτόχρονα. Έτσι, τόσο η εισαγωγή στο g_regions map όσο και η εγγραφή στο trace αποτελούν κοινούς πόρους που πρέπει να προστατευτούν από race conditions. Για τον λόγο αυτό χρησιμοποιούνται δύο ξεχωριστά locks, το g_regions_lock για το map και το g_events_lock για το trace αρχείο. Έτσι εξασφαλίζεται η ορθή λειτουργία σε multi-threaded περιβάλλον [21]. Η αναλυτική περιγραφή του μηχανισμού για τον συγχρονισμό παρουσιάζεται στο υποκεφάλαιο 3.7.

Σε αντίθεση με τους υπόλοιπους allocators για τους οποίους απαιτείται BEFORE/AFTER pattern όπως περιγράφηκε στο 3.4.1, η free() γίνεται hook μόνο στο IPOINT_BEFORE, δηλαδή πριν εκτελεστεί η πραγματική αποδέσμευση της μνήμης [21]. Αυτό γιατί η free() δεν έχει επιστρεφόμενη τιμή που να είναι απαραίτητη, άρα το IPOINT_AFTER δεν προσφέρει καμία χρήσιμη πληροφορία. Κατά την εκτέλεση της BeforeFree, το Pintool αναζητά το pointer που δόθηκε στη free(), στο g_regions map και διακρίνονται τρείς περιπτώσεις.

Στην πρώτη περίπτωση, το pointer αντιστοιχεί ακριβώς στο start address ενός καταγεγραμμένου region. Αφού βρεθεί το region, αφαιρείται από το map με g_regions.erase() και καλείται η PrintFreeKnown που γράφει στο trace την γραμμή για [FREE] με τα πεδία Region_start, Region_size, Type, Free_ptr και Caller. Στην δεύτερη περίπτωση, το pointer δεν αντιστοιχεί σε αρχή region αλλά βρίσκεται μέσα στα όρια

ενός καταγεγραμμένου region, δηλαδή πρόκειται για interior pointer. Αυτό αποτελεί ανώμαλη κατάσταση αφού η free() δεν ορίζεται για interior pointers. Το pintool σε αυτή την περίπτωση δεν διαγράφει το region από το map και αντί για κανονικό Free event γράφει στο trace γραμμή ANOMALY free_interior με το offset του pointer μέσα στο region. Αυτή η δομή ελέγχου δημιουργήθηκε αποκλειστικά για σκοπούς debug και δεν αποτελεί φυσιολογική συμπεριφορά του Memcached. Στην τρίτη περίπτωση, το pointer δεν αντιστοιχεί σε κανένα καταγεγραμμένο region. Εδώ καλείται η PrintFreeUnknown η οποία γράφει γραμμή [FREE – UNKNOWN] με μόνο το Free_ptr και το Caller. Αυτή η περίπτωση εμφανίζεται συνήθως όταν η glibc δεσμεύει εσωτερικά μνήμη μέσω mmap χωρίς να καλέσει malloc() ή calloc() [14][20]. Από τη συμπεριφορά αυτή της glibc το Pintool δεν μπορεί να καταγράψει ποτέ το αντίστοιχο allocation στο g_regions map. Όταν αργότερα καλείται η free() για αυτή τη διεύθυνση μνήμης το Pintool τη βλέπει αλλά δεν βρίσκει το αντίστοιχο region στο map.

3.5 Καταγραφή Memory Accesses

Για την καταγραφή memory accesses το Pintool κατά την εκκίνηση του, καταχωρεί την συνάρτηση Instructions ως instrumentation callback μέσω της συνάρτησης INS_AddInstrumentFunction από PIN API [21]. Στο PIN οι instrumentation callbacks δεν εκτελούνται στο hot path, αλλά γίνεται η κλήση τους από το PIN κατά το JIT compilation, μια φορά για κάθε εντολή που εμφανίζεται πρώτη φορά και ο σκοπός τους είναι να αποφασίσουν το analysis callback που θα εγκατασταθεί σε αυτή [5]. Η Instruction είναι αυτή που εγκαθιστά τα πραγματικά analysis callbacks τα οποία θα εκτελούνται αργότερα στο hot path και εκτελείται μόνο μια φορά για κάθε εντολή [21].

Η Instruction συνάρτηση καλεί στην συνέχεια την βοηθητική συνάρτηση ShouldInstrumentIns, η οποία είναι υπεύθυνη να αποφασίσει αν η εντολή θα γίνει instrument ή όχι. Η συνάρτηση αυτή επιτρέπει πάντα στο main executable να γίνει instrument ανεξαρτήτως knobs. Η απόφαση για shared libraries εξαρτάται από τα knobs main_exe_only και trace_libc_memops που αναλύθηκαν στο 3.3. Αν δηλαδή το main_exe_only είναι απενεργοποιημένο επιτρέπονται όλα τα images, αν είναι ενεργοποιημένο όμως επιτρέπονται επιπλέον μόνο τα libc images εφόσον το trace_libc_memops είναι και αυτό ενεργοποιημένο. Αν η εντολή περάσει από το φίλτρο

της ShouldInstrumentIns συνάρτησης γίνεται έλεγχος μέσω INS_MemoryOperandCount (PIN API call) αν η εντολή περιέχει memory operands. Αν έχει read operand εγκαθιστά την RecordRead (analysis callback) και αντίστοιχα για write εγκαθιστά την RecordWrite (analysis callback) μέσω INS_InsertPredicatedCall στο IPOINT_BEFORE [21]. Η επιλογή για χρήση της INS_InsertPredicatedCall έναντι της INS_InsertCall έγινε για να εξασφαλιστεί ότι το analysis callback καλείται μόνο όταν η εντολή εκτελείται πραγματικά, αποφεύγοντας κλήσεις για εντολές που ακυρώνονται, γεγονός συχνό για αρχιτεκτονικές x86 με REP prefix ή σε αρχιτεκτονικές με conditional execution [17].

Για κάθε πρόσβαση read/write, η RecordRead και RecordWrite λαμβάνουν τη διεύθυνση μνήμης που προσπελαύνεται και καλούν την συνάρτηση FindRegionWithOff για να γίνει lookup στο map και να προσδιοριστεί αν η διεύθυνση ανήκει σε κάποιο region από το map. Η FindRegionWithOff αποκτά κλείδωμα g_regions_lock και εκτελεί upper_bound(a) στο g_regions map. Το upper_bound(a) επιστρέφει τον πρώτο κόμβο που βρίσκει με start > a και στην συνέχεια πηγαίνει ένα κόμβο πίσω και ελέγχει αν ισχύει start $\leq$ a < start + size. Αν ισχύει η ανισότητα, επιστρέφει ένα snapshot του Region αποθηκεύοντας το στην μεταβλητή out και το offset (a – start) της διεύθυνσης πρόσβασης μέσα στο Region. Με αυτό τον τρόπο το g_regions_lock αποδεσμεύεται αμέσως μετά την αντιγραφή του region struct στην μεταβλητή out (snapshot) και οι RecordRead και RecordWrite μπορούν να γράψουν στο trace χωρίς το lock, αποφεύγοντας έτσι περιττό blocking σε multi-threaded environment. Αν όμως δεν βρεθεί αντίστοιχο Region και το knob trace_untracked είναι 0, το access αγνοείται χωρίς να γίνει οποιαδήποτε εγγραφή στο trace. Σε περίπτωση εύρεσης του region, γράφεται στο trace μια γραμμή της παρακάτω μορφής:
T0 ROLE=WORKER OS_TID=12345 0x55a2bd4252b4: load base:0x7fb544018e80 full:0x7fb544018e88 tag=heap:calloc off=8 size=8 ip_img=memcached+0x1a2b4 ALLOC_CALLER=cache.c:84

Πεδίο	*Περιγραφή*
T0	Thread ID
ROLE	Ρόλος του Thread (WORKER/OTHER).

OS_TID	Πραγματικό OS thread ID.
0x55a2bd4252b4	Instruction pointer της εντολής που εκτέλεσε το access.
Load/store	Τύπος πρόσβασης
Base	Αρχική διεύθυνση του region στο οποίο ανήκει η πρόσβαση.
full	Ακριβής διεύθυνση μνήμης που προσπελάστηκε.
Off	Offset μέσα στο region, δηλαδή η διαφορά full – base σε bytes.
Size	Μέγεθος της πρόσβασης σε bytes.
Tag	Τύπος του Region, αναλύθηκε στο υποκεφάλαιο 3.4.5
Ip_img	Image – relative offset της εντολής
ALLOC_CALLER	Allocation site του region (file:line) στο οποίο έγινε η πρόσβαση. Κλειδί για την κατηγοριοποίηση σε persistent/volatile.

Πίνακας 3.5: Πεδία γραμμής από πρόσβαση στο trace.

3.6 Μηχανισμός Ελέγχου Tracing Window (SIGUSR2)

Για την καταγραφή του trace είναι σημαντικό να οριστεί με ακρίβεια το χρονικό παράθυρο για το οποίο θα είναι ενεργή η καταγραφή των memory accesses που πρέπει να μελετηθούν. Στα πειράματα της παρούσας εργασίας η εκτέλεση χωρίζεται σε δύο φάσεις, το load phase και το measurement phase. Κατά το load phase φορτώνονται τα κλειδιά στην μνήμη και κατά το measurement phase εκτελείται το read-only workload που αποτελεί το αντικείμενο μέτρησης της μελέτης. Ο διαχωρισμός αυτός γίνεται γιατί η καταγραφή των memory accesses πρέπει να περιορίζεται αποκλειστικά στο measurement phase καθώς στο load phase εισάγονται accesses που δεν αντιπροσωπεύουν τη steady-state συμπεριφορά της εφαρμογής και θα αλλοίωναν τα αποτελέσματα της προσομοίωσης. Ο στόχος της παρούσας εργασίας είναι η μελέτη του write traffic που παράγεται ακόμα και κάτω από καθαρά read-only συνθήκες, γι’ αυτό και η καταγραφή κατά το load phase θα νόθευε το δείγμα προς μελέτη.

Για να γίνει ο διαχωρισμός αυτός σε load και measurement phase υλοποιήθηκε ένας μηχανισμός ελέγχου του tracing window μέσω SIGUSR2 signal. Κανονικά σε Linux συστήματα τα signals παραδίδονται απευθείας στη διεργασία όπως στην περίπτωση της παρούσας εργασίας στο Memcached. Το PIN ωστόσο τρέχει ανάμεσα στο λειτουργικό

σύστημα και την εφαρμογή, οπότε παρεμβάλλεται και βλέπει πρώτο κάθε signal πριν αυτό φτάσει στο Memcached [17]. Κατά την εκκίνηση του Pintool καταχωρείται ο handler OnSigUsr2 μέσω PIN_InterceptSignal (SIGUSR2, OnSigUsr2, 0) [17], δηλώνοντας στο PIN ότι κάθε φορά που φτάνει SIGUSR2 πρέπει να καλέσει πρώτα τον OnSigUsr2 και αυτός αποφασίζει τι θα γίνει με το signal. Ο handler εναλλάσσει την τιμή του global flag g_trace_memops μεταξύ TRUE/FALSE, καταγράφει την αλλαγή στο log αρχείο και επιστρέφει FALSE, το οποίο για PIN σημαίνει ότι το signal δεν προωθείται στην εφαρμογή [17]. Αποτέλεσμα αυτού είναι η άγνοια του Memcached για το γεγονός αποστολής του SIGUSR2 και η λειτουργία του ορθά δεν διαταράσσεται καθώς το Memcached δεν διαθέτει handler για SIGUSR2.

Το global flag g_trace_memops ορίζεται ως volatile BOOL στον κώδικα. Το volatile είναι C/C++ qualifier το οποίο αναγκάζει τον compiler να διαβάζει την τιμή της μεταβλητής αποκλειστικά από την μνήμη και όχι από register ή cache του compiler. Χωρίς Volatile ο compiler θα μπορούσε να κάνει optimization. Optimization γίνεται καθώς ο compiler βλέπει ότι η g_trace_memops διαβάζεται σε κάθε κλήση RecordRead/RecordWrite στον κώδικα χωρίς να αλλάζει η τιμή του και αποφασίζει ότι είναι ασφαλές να διαβαστεί μια φορά από την μνήμη και να αποθηκευτεί σε register για μελλοντική χρήση καθώς αυτό αποτελεί ταχύτερη λύση από διαρκή ανάγνωση από την μνήμη. Αυτό θα είχε ως αποτέλεσμα οι RecordRead και RecordWrite να διαβάζουν παλιά τιμή από register ακόμα και μετά την αλλαγή στην τιμή από handler ενώ με χρήση volatile εξασφαλίζεται ότι κάθε ανάγνωση της flag αντικατοπτρίζει πάντα την πιο πρόσφατη τιμή. Η χρήση volatile έναντι mutex ή lock έγινε γιατί η εναλλαγή τιμής του flag είναι ατομικό operation σε x86 και δεν απαιτείται πλήρης συγχρονισμός, το σημαντικό είναι να επιτευχθεί η αποφυγή του compiler optimization. Αν λοιπόν οι τιμή του flag g_trace_memops είναι FALSE οι RecordRead και RecordWrite επιστρέφουν χωρίς να καταγράψουν προσβάσεις στο trace.

Στην πράξη, το g_trace_memops flag έχει FALSE αρχική τιμή και έτσι η καταγραφή memory accesses είναι απενεργοποιημένη κατά την εκκίνηση. Όταν το load phase ολοκληρωθεί, αμέσως πριν ξεκινήσει το measurement phase ο χρήστης στέλνει χειροκίνητα signal χρησιμοποιώντας την εντολή kill -SIGUSR2 <pid>, όπου pid είναι το process ID του Memcached. Ο handler τότε αναλαμβάνει να αλλάξει την τιμή του

flag σε TRUE και από εκείνη την στιγμή οι RecordRead και RecordWrite αρχίζουν την καταγραφή memory accesses στο trace. Όταν το measurement phase ολοκληρωθεί, ένα δεύτερο kill -SIGUSR2 <pid> απενεργοποιεί ξανά την καταγραφή.

3.7 Multi-threading & Συγχρονισμός

Το Memcached είναι multi-threaded server. Κατά την εκτέλεση του workload υπάρχουν πολλαπλά threads τα οποία είναι υπεύθυνα για να εξυπηρετούν GET requests ταυτόχρονα. Το Pintool από την πλευρά του τρέχει μαζί με την εφαρμογή Memcached στο ίδιο address space [5] και αυτό έχει ως αποτέλεσμα τα analysis callbacks όπως RecordRead και RecordWrite, καθώς και τα allocator hooks εκτελούνται ταυτόχρονα από πολλαπλά threads. Αυτό το γεγονός αφήνει απροστάτευτες τις κοινές δομές δεδομένων του Pintool από race conditions. Για να αποφευχθούν οποιασδήποτε μορφής race conditions γίνεται χρήση PIN locks μέσω PIN_GetLock και PIN_ReleaseLock [17]. Τα standard POSIX mutexes που αποτελούν τον κλασικό μηχανισμό συγχρονισμού του Linux μέσω pthread_mutex_lock/pthread_mutex_unlock δεν μπορούν να χρησιμοποιηθούν γιατί είναι πιθανό να προκληθεί deadlock, αφού η διεργασία που θα κρατά το lock διακόπτεται αλλά το lock παραμένει ενεργό. Το PIN για αυτό τον λόγο ορίζει αυστηρή ιεραρχία για την απόκτηση locks η οποία αποτελείται από [17]:

$$\text{Application locks} \rightarrow \text{PIN internal locks} \rightarrow \text{Tool locks}$$

Αυτό σημαίνει ότι τα locks του Pintool (Pin_GetLock) βρίσκονται πάντα χαμηλότερα ιεραρχικά από τα locks εφαρμογής και η σειρά απόκτησης τους δεν αντιστρέφεται ποτέ. Deadlock θα προκληθεί μόνο όταν δύο threads πάρουν τα ίδια locks με αντίστροφη σειρά. Με την εφαρμογή της ιεραρχίας αυτής το φαινόμενο αυτό αποφεύγεται αφού έχοντας δεσμευμένο ένα lock υψηλότερου επιπέδου, όπως στην περίπτωση που μελετάμε που είναι δεσμευμένο το Application lock, μπορεί να πάρει μόνο lock χαμηλότερου επιπέδου.

Ένα παράδειγμα είναι όταν ένα worker thread του Memcached κρατά POSIX mutex M1 (πχ item lock για ένα key) και εκτελεί εντολή πρόσβασης load. Το PIN τότε διακόπτει(interrupt) την λειτουργία του Memcached και τρέχει το RecordRead callback το οποίο εγκαθίσταται στην εντολή κατά το JIT compilation μέσω

INS_InsertPredicatedCall. Το RecordRead callback δεν αγγίζει καθόλου τα δεδομένα του Memcached και δεν χρειάζεται να έχει πρόσβαση σε καμία δομή που προστατεύεται από το M1. Η δουλειά του RecordRead callback είναι να κάνει lookup στο g_regions map και γράψει στο trace. Αυτά είναι αποκλειστικά δομές του Pintool που προστατεύονται από g_regions_lock και g_events_lock αντίστοιχα, εσωτερικά του Pintool. Τα PIN locks αυτά είναι εντελώς ανεξάρτητα από το M1 mutex. Το thread παίρνει tool lock καθώς έχει στην κατοχή του application lock. Η συμπεριφορά αυτή του thread ακολουθεί αυστηρά την ιεραρχία που θέτει το PIN με αποτέλεσμα την σωστή λειτουργία του συστήματος και την αποφύγει deadlocks.

Αξίζει να σημειωθεί ότι για σκοπούς της παρούσας εργασίας το Memcached έχει εκτελεστεί μόνο με ένα μοναδικό worker thread. Ο λόγος που οδήγησε στην εξής απόφαση ήταν εξολοκλήρου οι δυνατότητες του DineroIV simulator που χρησιμοποιήθηκε για την προσομοίωση της cache συμπεριφοράς. Ο DineroIV είναι single-threaded σύστημα και γι' αυτό αν γινόταν καταγραφή πολλαπλών worker threads το trace θα περιείχε interleaved accesses από διαφορετικά worker threads τα οποία ο DineroIV δεν θα μπορούσε να χειριστεί ορθά καθώς θα αντιμετώπιζε το trace ως ένα ενιαίο sequential stream με αποτέλεσμα να χανόταν η πραγματική σειρά εκτέλεσης των εντολών. Παρ' όλ' αυτά, ο συγχρονισμός υλοποιήθηκε πλήρως ώστε το Pintool να έχει ορθή συμπεριφορά και σε multi-threaded περιβάλλοντα.

g_regions_lock:

Το g_regions_lock χρησιμοποιείται για να προστατεύσει το g_regions map, το οποίο αποτελεί την σημαντικότερη δομή του Pintool. Η δομή αυτή τροποποιείται από allocator hooks όπως AfterMalloc, AfterCalloc κλπ. Επίσης, γίνεται πρόσβαση σε αυτή από RecordRead και RecordWrite μέσω της συνάρτησης FindRegionWithOff κατά την διεξαγωγή του lookup. Σε περίπτωση που η προστασία από locks θα ήταν ελλιπής ή δεν υπήρχε, ένα thread που εκτελεί εντολή αποδέσμευσης και διαγράφει το region από το map καθώς ένα άλλο thread κάνει lookup προκαλεί race condition. Το Pintool έγινε optimized ώστε το lock να αποκτάται για το ελάχιστο δυνατό διάστημα, όπως αναλύθηκε στο 3.5, η συνάρτηση FindRegionWithOff αποδεσμεύει το g_regions_lock

αμέσως μετά την αντιγραφή του region struct ώστε να μην μπλοκάρονται τα υπόλοιπα threads κατά την διάρκεια της εγγραφής στο trace.

g_events_lock:

Το g_events_lock δημιουργήθηκε για να προστατεύει τις εγγραφές στο trace αρχείο, λόγο του ότι το trace είναι global αρχείο και πολλαπλά threads μπορούν να γράψουν σε αυτό μέσω RecordRead και RecordWrite, allocator hooks και OnSigUsr2. Χωρίς την προστασία που παρέχει το g_events_lock οι εγγραφές θα παρεμβάλλονταν μεταξύ τους και το trace θα ήταν αναξιόπιστο.

g_hook_lock:

Το g_hook_lock είναι υπεύθυνο για να προστατεύει το g_hooked_rtn_addrs το οποίο αποτελεί ένα set με αποθηκευμένες τις διευθύνσεις των routines που έχουν ήδη γίνει hook. Κάθε φορά που φορτώνεται ένα image καλείται η TryMarkHooked η οποία ελέγχει αν η routine έχει ήδη γίνει hook, αν δεν έχει γίνει την καταχωρεί. Ο λόγος για τον οποίο είναι αναγκαίο το g_hook_lock είναι η αποφυγή διπλής εγκατάστασης hook στην ίδια routine όταν δύο threads φτάσουν ταυτόχρονα στο ImageLoad για το ίδιο image. Αυτό θα προκαλούσε διπλές εγγραφές στο trace και θα το έκανε αναξιόπιστο για την μελέτη.

g_stack_lock:

Αυτό το lock προστατεύει το g_stack_regions, μια δομή που καταγράφει τα stack regions για κάθε thread.

g_src_cache_lock AND g_img_cache_lock:

Αυτά τα locks χρησιμοποιούνται για την προστασία των caches για source location και image resolution αντίστοιχα όπως αναλύονται στο 3.8.

Ωστόσο, για allocator hooks η προστασία μέσω locks δεν είναι αρκετή, απαιτείται και Thread Local Storage (TLS). Όπως έχει ήδη αναλυθεί στο 3.4.3, κάθε thread διατηρεί το δικό του ThreadCTX struct το οποίο είναι υπεύθυνο για την προσωρινή αποθήκευση της κατάστασης του allocator μεταξύ του BEFORE και AFTER hook. Το g_tls_key είναι ένα κλειδί που δημιουργείται από το PIN μέσω PIN_CreateThreadDataKey [17] και χρησιμοποιείται από ThreadCTX και επιτρέπει σε κάθε thread να αποθηκεύει και να ανακτά το δικό του ThreadCTX μέσω PIN_GetThreadData(g_tls_key, tid). Στην ουσία είναι ένας global δείκτης που για κάθε thread δείχνει σε διαφορετικό αντικείμενο.

Τέλος, όπως ήδη έχει αναλυθεί στο 3.6, το g_trace_memops flag έχει οριστεί ως volatile αντί να προστατεύεται από lock αφού η εναλλαγή τιμής του από το OnSigUsr2 handler είναι atomic operation σε x86. Το volatile εξασφαλίζει ότι θα αποφευχθεί το optimization από τον compiler.

3.8 Βελτιστοποιήσεις Απόδοσης

Παρατηρήθηκε ότι κατά την εκτέλεση του Pintool, καλούνται δύο λειτουργίες πολύ συχνά με σημαντικό αντίκτυπο στην απόδοση του. Η πρώτη λειτουργία είναι η source location resolution η οποία προσδιορίζει το file:line για κάθε allocation site και η δεύτερη είναι η image resolution η οποία είναι υπεύθυνη για να προσδιορίσει το image-relative offset για κάθε instruction pointer. Οι λειτουργίες αυτές έχουν αναλυθεί στο 3.4.4 πιο λεπτομερώς. Η πρώτη λειτουργία καταλήγει σε κλήση addr2line και η δεύτερη σε κλήση PIN API για αναζήτηση image, όπου και στις δύο περιπτώσεις επιβαρύνετε η απόδοση του Pintool αφού είναι ακριβές λειτουργίες και εκτελούνται αρκετές φορές στο σύστημα. Για αυτό τον λόγο έχουν δημιουργηθεί caches για την κάθε μια που αποθηκεύουν τα αποτελέσματα και αποφεύγουν επαναλαμβανόμενες ακριβές αναζητήσεις.

Το g_img_cache είναι στην ουσία ένα unordered map από instruction pointer (ADDRINT) σε ζεύγος δύο τιμών. Η πρώτη τιμή είναι το image name, δηλαδή το όνομα του εκτελέσιμου ή της shared library, όπως Memcached ή libc.so στην οποία ανήκει η εντολή. Η δεύτερη τιμή είναι το image-relative offset, δηλαδή η απόσταση σε bytes από την αρχή του image μέχρι την συγκεκριμένη εντολή. Το offset αυτό

παραμένει σταθερό σε κάθε εκτέλεση ανεξαρτήτως του πού φορτώθηκε το εκτελέσιμο στη μνήμη λόγω ASLR. Οι τιμές αυτές σχηματίζουν ένα πεδίο μορφής ip_img=Memcached+0x1a2b4. Κάθε φορά που χρειάζεται το image-relative offset ενός instruction pointer η GetImgFromIpCached ελέγχει αν υπάρχει ήδη στην g_img_cache, αν ναι επιστρέφετε αμέσως χωρίς την εκτέλεση του lookup. Αν όμως δεν υπάρχει, γίνεται κλήση της GetImgFromIp για την ακριβή αναζήτηση και αποθηκεύει το αποτέλεσμα στο g_img_cache για μελλοντική χρήση. Το cache προστατεύεται από το g_img_cache_lock.

Το g_src_cache είναι ένα unordered_map από string key σε SrcLoc struct. Το string key είναι συνδυασμός του image path και του image-relative offset, δηλαδή μια μοναδική ταυτότητα για κάθε συνδυασμό εκτελέσιμου και θέσης εντολής μέσα σε αυτό. Το SrcLoc struct περιέχει 3 πεδία:

Πεδίο	*Περιγραφή*	*Παράδειγμα*
File	Όνομα αρχείου πηγαίου κώδικα.	Cache.c
Line	Αριθμός γραμμής μέσα στο αρχείο.	84
Col	Η στήλη μέσα στο αρχείο. (Δεν χρησιμοποιείται στο Pintool)	10

Πίνακας 3.6: Πεδία SrcLoc struct.

Τα πεδία αυτά μαζί αποτελούν το allocation site, δηλαδή το ακριβές σημείο στον πηγαίο κώδικα από το οποίο έγινε η δέσμευση μνήμης.

Η λογική που ακολουθεί αυτή η cache είναι ίδια με της g_img_cache. Όταν χρειάζεται το source location ενός instruction pointer, η ResolveSrcWithAddr2line συνάρτηση ελέγχει πρώτα αν υπάρχει ήδη στο g_src_cache, αν υπάρχει επιστρέφει αμέσως χωρίς να καλέσει το addr2line. Σε περίπτωση όμως που δεν υπάρχει, καλείται το addr2line, το οποίο είναι ένα εξωτερικό εργαλείο υπεύθυνο για την μετατροπή μιας διεύθυνσης μνήμης σε ονόματα αρχείων και αριθμός γραμμών πηγαίου κώδικα [12]. Το αποτέλεσμα αυτού, αποθηκεύεται στο g_src_cache για μελλοντική χρήση. Είναι σημαντικό να σημειωθεί ότι το g_src_cache μπορεί να κρατά και αρνητικά

αποτελέσματα όπως `??:?` για την περίπτωση όπου το addr2line αποτύχει να βρει το source location ώστε να μην κληθεί ξανά το addr2line για το ίδιο instruction pointer στο μέλλον. Το cache προστατεύεται από το g_src_cache_lock.

Κεφάλαιο 4

Μεθοδολογία Πειραμάτων

4.1 Επισκόπηση Πειραματικής Μεθοδολογίας

Ο στόχος των πειραμάτων που έγιναν για σκοπούς τις παρούσας εργασίας είναι η αξιολόγηση μιας selective write policy η οποία θα εφαρμόζει Write-Through (WT) αποκλειστικά στα persistent objects και Write-Back (WB) στα volatile objects, σε σύγκριση με μια pure WT και μια pure WB policy. Το κεντρικό ερευνητικό ερώτημα είναι αν η selective policy μπορεί να μειώσει το write traffic προς τη NVM έναντι της pure WT, διατηρώντας παράλληλα την εγγύηση consistency που η pure WB αδυνατεί να προσφέρει.

Για την διεξαγωγή αυτών των πειραμάτων και την αξιολόγηση του ερευνητικού ερωτήματος αναπτύχθηκε ένα πειραματικό pipeline το οποίο αποτελείται από τέσσερα στάδια. Στο πρώτο στάδιο το Pintool που έχει αναλυθεί στο Κεφάλαιο 3, καταγράφει τα memory accesses του Memcached κατά το measurement phase, αποδίδοντας, μεταξύ άλλων, κάθε πρόσβαση που γίνεται στο allocation site που δημιούργησε το αντίστοιχο memory object. Στο δεύτερο στάδιο, έχει δημιουργηθεί ένα script (segregate_trace.py) αποκλειστικά για τους σκοπούς της παρούσας εργασίας, το οποίο χωρίζει το trace σε δύο διαφορετικά αρχεία, κρατώντας το αυθεντικό. Το πρώτο αρχείο που δημιουργείτε

είναι το pinatrace_wt.out το οποίο περιέχει τις προσβάσεις για τα persistent objects και το δεύτερο είναι το pinatrace_wb.out για τις προσβάσεις των volatile objects. Και τα δύο αρχεία περιέχουν και τις εγγραφές για δεσμεύσεις/αποδεσμεύσεις. Η κατηγοριοποίηση αναλύεται στο υποκεφάλαιο 4.5. Στο τρίτο στάδιο, ο DineroIV cache simulator προσομοιώνει τη συμπεριφορά της cache για κάθε trace ξεχωριστά, χρησιμοποιώντας διαφορετικά μεγέθη για cache και write policies για κάθε πείραμα. Στο τέταρτο στάδιο είναι εκεί που τα αποτελέσματα αποκτούν σημασία αφού εδώ δημιουργήθηκαν python scripts τα οποία υπολογίζουν το write traffic και το AMAT για κάθε πολιτική και cache size.

Οι write policies που συγκρίνονται είναι η pure WT, η pure WB και η selective policy. Όσο αφορά τη selective policy, λόγω του ότι τα δύο traces προσομοιώνονται χωριστά, τα αποτελέσματα αποτελούν lower bound, δηλαδή δεν λαμβάνεται υπόψη η αλληλεπίδραση μεταξύ persistent και volatile objects στην κοινή cache και έτσι αφαιρούνται από το πείραμα evictions που προκαλούνται από την αλληλεπίδραση αυτή. Για τον λόγο αυτό υπολογίζεται το upper bound το οποίο λαμβάνει υπόψη το worst case σενάριο αυτής της αλληλεπίδρασης. Κάθε στάδιο του pipeline αναλύεται εκτενώς στα επόμενα υποκεφάλαια.

4.2 Workload – Memcached & mcperf

Για την παραγωγή του trace έγινε χρήση του Memcached 1.6.14 ως εφαρμογή υπό μελέτη και για load generator χρησιμοποιήθηκε το mcperf. Το Memcached εκκινήθηκε με την εντολή:

memcached -t 1 -p 11211 -u $USER -m 512

Παράμετρος	*Εξήγηση*
-t 1	Μόνο ένα worker thread.
-p 11211	Port επικοινωνίας.
-m 512	512MB μνήμη – Αρκετή για 1 εκατομμύριο κλειδιά.
-u $USER	Ορισμός χρήστη εκτέλεσης.

Πίνακας 4.1: Παράμετροι Memcached

Επίσης εκτελέστηκε με τα εξής knobs:

Knob	*Τιμή*
use_libc_hooks	1
main_exe_only	1
trace_libc_memops	1
worker_only	1
trace_stack	0
trace_globals	0
trace_untracked	0
src_debug	0

Πίνακας 4.2: Knobs

Η εκτέλεση του workload χωρίζεται σε δύο φάσεις, load phase και measurement phase, όπως έχει ήδη αναλυθεί στο υποκεφάλαιο 3.6. Η καταγραφή των δύο φάσεων ελέγχεται μέσω του μηχανισμού SIGUSR2.

Load Phase:

Κατά το load phase φορτώνονται 1 εκατομμύριο κλειδιά στο Memcached χωρίς να γίνεται καταγραφή προσβάσεων από το Pintool. Η εντολή που χρησιμοποιήθηκε είναι:

```
./mcperf -s 127.0.0.1 --binary --loadonly \
  -r 1000000 -K 30 -V 200 \
  --keycache_capacity=1000000 \
  --keycache_reuse=1 \
  --keyorder=uniform:1000000 \
  --keycache_regen=100
```

Το –binary ορίζει χρήση binary protocol για την επικοινωνία mcperf – memcached αντί για text protocol. Έγινε επιλογή binary protocol γιατί είναι ταχύτερο λόγο του ότι αποφεύγει το parsing του text.

Το –loadonly ορίζει ότι το mcperf εκτελεί μόνο load phase, δηλαδή είναι υπεύθυνο να φορτώσει keys και values στο Memcached χωρίς να εκτελεί οποιουδήποτε είδους measurements. Αρχικοποιεί το memcached.

Η παράμετρος -r 1000000 καθορίζει ότι θα φορτωθούν 1 εκατομμύριο κλειδιά στο Memcached αντιπροσωπεύοντας αρκετά ρεαλιστικές συνθήκες για το πείραμα της μελέτης.

Οι παράμετροι -K 30 και -V 200 έχουν την default τιμή τους. Καθορίζουν το μέγεθος key και value bytes αντίστοιχα.

Το –keycache_capacity=1000000 ορίζει το μέγεθος του keycache του mcperf. Το keycache είναι η εσωτερική δομή που αποθηκεύει τα keys που θα χρησιμοποιηθούν για GET/SET. Ορίστηκε ίσο με τον αριθμό των keys (-r 1000000) γιατί αν ήταν μικρότερο ο mcperf θα επέλεγε keys μόνο από το subset για GET/SET και θα προκαλούσε τεχνητό locality.

Το –keycache_reuse=1 καθορίζει ότι το keycache αλλάζει σε κάθε request χωρίς επανάχρηση. Γενικά η παράμετρος αυτή καθορίζει πόσες φορές θα χρησιμοποιηθεί το ίδιο key cache πριν αλλαχτεί. Αν η τιμή ήταν μεγάλη (πχ 500) το ίδιο υποσύνολο από keys θα χρησιμοποιούνταν πολλές φορές (500) πριν αλλάξει. Αυτό θα δημιουργούσε ένα τεχνητό locality. Με reuse=1 και uniform keyorder εξασφαλίζεται ότι δεν ευνοείται κανένα υποσύνολο keys.

Το –keyorder=uniform:1000000 ορίζει uniform κατανομή πρόσβασης. Αυτό σημαίνει ότι κάθε key έχει την ίδια πιθανότητα να επιλεγεί από το σύνολο 1 εκατομμυρίου keys. Αυτό σημαίνει ότι προκαλείται χαμηλό locality αφού πολύ σπάνια θα ζητηθεί το ίδιο key δύο φορές διαδοχικά. Επιλέχθηκε για να μελετηθεί η συμπεριφορά του συστήματος υπό worst-case cache pressure, όπου η cache δεν μπορεί να εκμεταλλευτεί πλήρως το temporal locality.

Το –keycache_regen=100 σημαίνει 100% ανανέωση του keycache κάθε φορά που γίνεται regeneration, δηλαδή πλήρης αντικατάσταση των κλειδιών σε regeneration. Αν

η τιμή ήταν μικρή, τότε μόνο ένα μικρό ποσοστό των keys θα άλλαζε κάθε φορά και το keycache θα παρέμενε σχεδόν σταθερό το οποίο θα προκαλούσε καλό locality. Σε συνδυασμό με το –keycache_reuse=1 εξασφαλίζεται ότι κάθε request ζητά τυχαίο key από ολόκληρο το dataset.

Measurement Phase:

Αμέσως μετά το load phase, στο measurement phase, ο χρήστης στέλνει kill -SIGUSR2 <pid> για να ενεργοποιηθεί η καταγραφή memory accesses στο trace:

```
./mcperf -s 127.0.0.1 --binary --noload \
  -r 1000000 -K 30 -V 200 \
  -u 0.00 -T 1 -c 4 -d 4 \
  -q 400 -t 210 -i exponential \
  --keycache_capacity=1000000 \
  --keycache_reuse=1 \
  --keyorder=uniform:1000000 \
  --keycache_regen=100
```

Οι παράμετροι –binary, -r, -K, -V, --keycache_capacity, --keycache_reuse, --keyorder και –keycache_regen παραμένουν ίδιες με το load phase για συνέπεια του workload.

Το –noload ορίζει ότι το mcperf εκτελεί μόνο measurement χωρίς load phase. Η χρήση του noload γίνεται αφού το Memcached έχει φορτωθεί με –loadonly. Αν γινόταν χρήση χωρίς να προηγηθεί load phase τότε η μνήμη του memcached θα ήταν άδεια και θα προκαλούνταν cold misses.

Με την παράμετρο -u 0.00 ορίζεται update ratio 0%, αυτό σημαίνει ότι στο measurement phase γίνονται μόνο GET requests και το workload είναι read-only. Η επιλογή αυτή έγινε γιατί ο στόχος της παρούσας εργασίας είναι να μελετηθεί το αναπόφευκτο write-traffic που παράγεται ακόμα και υπό καθαρά read-only συνθήκες. Δηλαδή να μελετηθεί το write traffic που προέρχεται αποκλειστικά από metadata updates όπως refcount increments και LRU timestamp updates για κάθε GET request.

Η παράμετρος -T 1 ορίζει 1 thread στον mcperf generator. Με 1 thread και -c 4, -d 4 υπάρχουν ήδη 16 concurrent requests τα οποία είναι αρκετά για να φορτώσουμε τον single worker thread του Memcached. Περισσότερα threads θα πολλαπλασίαζαν το concurrency και άρα το QPS, με κίνδυνο να γίνει υπερφόρτωση του Memcached υπό PIN overhead. Τα requests θα συσσωρεύονταν στην ουρά του Memcached προκαλώντας timeouts και αναξιόπιστα αποτελέσματα. Ένας επιπλέον λόγος που προκάλεσε επιφύλαξη για χρήση περισσότερων threads είναι ότι σύμφωνα με τα λεγόμενα των authors του mcperf, threads <= cores. Αυτό και το γεγονός ότι το PIN και το Memcached κατέχουν ήδη cores για την εκτέλεση τους θα μπορούσε να προκληθεί context switching λόγω contention με τα threads του Memcached και PIN. Με -T 1 ο mcperf χρησιμοποιεί 1 core αποφεύγοντας context switching και εξασφαλίζοντας σταθερό και προβλέψιμο timing των requests.

Οι παράμετροι -c 4 και -d 4 ορίζουν 4 TCP connections και depth 4 αντίστοιχα. Όσα περισσότερα TCP connections τόσα περισσότερα concurrent requests θα δημιουργούνται. Το depth ορίζει πόσα requests μπορώ να στείλω και να περιμένουν χωρίς να δώσουν respond. Σύμφωνα με τους authors του mcperf το default (1 connection, depth=1) δεν είναι ρεαλιστικό αφού αντιπροσωπεύει ένα client που στέλνει ένα request και περιμένει απάντηση πριν στείλει το επόμενο, δηλαδή ακολουθεί σειριακή εκτέλεση. Με -c 4 και -d 4 δημιουργούνται 4 connections x 4 outstanding requests = 16 concurrent requests που αντιπροσωπεύουν ένα πιο ρεαλιστικό σενάριο. Παρ' όλ' αυτά, παραμένει θεωρητικά συντηρητικό έναντι του guideline για >50 connections γιατί με single worker thread στο Memcached και PIN overhead θα υπήρχε υψηλότερο concurrency το οποίο θα δημιουργούσε bottleneck στον client και όχι στο Memcached.

Η παράμετρος -q 400 ορίζει ότι ο στόχος είναι 400 QPS. Επιλέχθηκε αρκετά υψηλό ώστε το bottleneck να είναι από την πλευρά του Memcached και όχι από mcperf, δηλαδή να μην μένει αδρανής το Memcached επειδή τα απεσταλμένα requests από mcperf είναι ανεπαρκής. Παρ' όλ' αυτά από επιλογή δεν είναι τόσο υψηλό ώστε να μην κατακλύσει τον single worker thread υπό PIN overhead. Στην πράξη το πραγματικό QPS ήταν ~121 γιατί το Memcached υπό PIN overhead δεν μπορούσε να εξυπηρετήσει 400 QPS.

Η παράμετρος -t 210 είναι υπεύθυνη για να ορίζει διάρκεια measurement 210 seconds. Αυτή η επιλογή του έγινε ώστε με το πραγματικό QPS να παραχθούν αρκετά queries για στατιστικά έγκυρα αποτελέσματα και επίσης το trace να είναι αρκετά μεγάλο για αξιόπιστο simulation.

Η παράμετρος τώρα -i exponential ορίζει exponential inter-arrival distribution, αυτό σημαίνει ότι γίνεται μια bursty άφιξη requests όπου κάποτε φτάνουν πολλά requests μαζί και άλλες φορές δεν φτάνει κανένα. Αυτό αντιπροσωπεύει ρεαλιστικό traffic σε αντίθεση με το uniform inter-arrival που θα ήταν steady και τεχνητά ομοιόμορφο.

Όταν τώρα ολοκληρωθεί το measurement phase στέλνεται δεύτερο kill -SIGUSR2 <pid> για να απενεργοποιηθεί η καταγραφή προσβάσεων στο trace. Συνολικά το measurement phase παρήγαγε ~25.000 GET requests σε διάρκεια 210 δευτερολέπτων με ~121 QPS αποδίδοντας trace με 9.685.053 memory accesses μεγέθους 2.08GB όπου και ήταν ο σκοπός να δημιουργηθεί ένα trace τέτοιου μεγέθους λόγο περιορισμού χωρητικότητας στην μηχανή. Η διάρκεια των 210 seconds επιλέχθηκε βάσει δοκιμαστικών εκτελέσεων που έγιναν με διαφορετικούς αριθμούς requests, από τις οποίες υπολογίστηκε ο απαιτούμενος χώρος στην μνήμη για αποθήκευση του trace ανά requests. Βάσει αυτών των μετρήσεων, με QPS ~121 και -t 210 seconds εκτιμήθηκε ότι το trace θα έφτανε κοντά στα 2GB που ήταν το μέγιστο που επέτρεπε η διαθέσιμη χωρητικότητα του συστήματος.

4.3 Cache Simulator – DineroIV

Για να γίνει η προσομοίωση της cache συμπεριφοράς ήταν απαραίτητη η επιλογή ενός trace-driven simulator. Σε ένα trace-driven simulator η είσοδος είναι ένα trace αρχείο, δηλαδή μια ακολουθία από memory accesses που καταγράφηκαν από πραγματική εκτέλεση του προγράμματος. Ο simulator διαβάζει το trace αρχείο που παίρνει ως input και προσομοιώνει τη συμπεριφορά της cache για κάθε access, υπολογίζοντας hits, misses και write traffic. Αυτή η προσέγγιση είναι κατάλληλη για τις ανάγκες που έχει η παρούσα εργασία γιατί επιτρέπει την προσομοίωση του ίδιου workload πολλές φορές υπό διαφορετικές παραμέτρους για την cache όπως διαφορετικά cache sizes, write

policies, associativity και πολλές άλλες παραμέτρους χωρίς να χρειάζεται εκτέλεση του Memcached ξανά και η παραγωγή trace για κάθε πείραμα.

Πριν την τελική απόφαση για χρήση του DineroIV εξετάστηκαν δύο εναλλακτικοί simulators. Ο ένας εκ των δύο, ο CacheSimulator [2], απορρίφθηκε λόγο του ότι δεν μπορούσε να υποστηρίξει 64-bit addresses ενώ το trace που παράγει το Pintool περιέχει 64-bit διευθύνσεις μνήμης και η χρήση του CacheSimulator θα απαιτούσε τροποποίηση μεγάλου ποσοστού του κώδικα του. Ο δεύτερος, ChampSim [3], είναι ένας πλήρης microarchitecture simulator που χρησιμοποιείται κυρίως για μελέτη branch prediction και prefetching, ωστόσο δεν διέθετε υλοποιημένη Write-Through πολιτική ενώ είναι σημαντική προϋπόθεση για την διεξαγωγή της παρούσας εργασίας.

Ο DineroIV είναι trace-driven cache simulator [7]. Επιλέχθηκε γιατί καλύπτει όλες τις ανάγκες της παρούσας εργασίας όπως, την υποστήριξη 64-bit addresses, διαθέτει Write-Back και Write-Through πολιτικές, είναι lightweight και είναι διαθέσιμος ελεύθερα για μη εμπορική χρήση, γεγονός που τον καθιστά ευρέως χρησιμοποιημένο για ακαδημαϊκούς σκοπούς.

Παρ' όλ' αυτά, ο DineroIV δεν είναι timing simulator, δηλαδή δεν υπάρχει έννοια χρόνου ή cycles, μόνο references. Το κύριο αποτέλεσμα της προσομοίωσης είναι πληροφορίες hit rate και miss rate, όπου στην συνέχεια έγινε τροποποίηση στον simulator ώστε η πληροφορία hit/miss να αποτυπώνετε στο trace για κάθε εγγραφή πρόσβασης που περιέχει. Επίσης για τις ανάγκες τις εργασίας ήταν απαραίτητος ο υπολογισμός AMAT(Average Memory Access Time), λειτουργία την οποία δεν παρείχε ο DineroIV. Η λειτουργία AMAT αναπτύχθηκε μετέπειτα με το script exp_amat_script.py το οποίο επεξεργάζεται τα αποτελέσματα του DineroIV και υπολογίζει το AMAT βάσει της μεθοδολογίας που αναλύεται στο υποκεφάλαιο 4.7.

<u>DineroIV output:</u>

Ο DineroIV παράγει στατιστικά αποτελέσματα για κάθε simulation. Χαρακτηριστικά, εξάγει τις εξής πληροφορίες:

Demand Fetches	Συνολικός αριθμός block-level lookups που πραγματοποιήθηκαν από demand references. Δηλαδή οι φορές που η cache έλεγξε αν

	ένα block βρίσκεται ήδη σε αυτήν. Χωρίζεται σε Read και Write υποκατηγορίες.
Demand Misses	Συνολικός αριθμός προσβάσεων από τις οποίες το block δεν βρέθηκε στην cache (DRAM). Χωρίζεται σε Read και Write υποκατηγορίες.

Πίνακας 4.3: Ανάλυση DineroIV output

Σημαντικό να αναφερθεί ότι ο DineroIV δεν επιβάλλει περιορισμούς μεγέθους ή ευθυγράμμισης στις προσβάσεις μνήμης, γι' αυτό μια εντολή μπορεί να αγγίζει δεδομένα που εκτείνονται σε N συνεχόμενα cache blocks. Σε μια τέτοια περίπτωση ο DineroIV καταγράφει N ξεχωριστές προσβάσεις (block lookups) για μια εντολή. Αυτό συμβαίνει γιατί το κάθε block έχει μέγεθος 64bytes και σε περιπτώσεις όπου η πρόσβαση είναι μεγαλύτερη πρέπει να γίνει lookup για το block που περιέχει τα υπόλοιπα bytes δεδομένων. Οι προσβάσεις τέτοιας μορφής αναφέρονται στα στατιστικά που εξάγει ο DineroIV ως Multi-block references [8].

Από τα στατιστικά του DineroIV εξάγονται τα read/write hits και misses ως εξής:

```
Read Misses = Demand Misses (Read)
Write Misses = Demand Misses (Write)
Read Hits = Demand Fetches (Read) – Demand Misses (Read)
Write Hits = Demand Fetches (Write) – Demand Misses (Write)
Total = Demand Fetches (Total)
```

4.4 Παραμετροποίηση DineroIV

Για κάθε trace ο DineroIV εκτελείται με συγκεκριμένο συνδυασμό παραμέτρων [7]. Η εκτέλεση γίνεται αυτόματα μέσω script το οποίο κτίζει δυναμικά την εντολή του DineroIV για κάθε πείραμα βάσει των παραμέτρων που θα δώσει ο χρήστης μετά από μια σειρά ερωτήσεων που κάνει το script στον χρήστη. Παρακάτω αναλύονται οι παράμετροι που χρησιμοποιήθηκαν στα πειράματα της παρούσας εργασίας.

Input Format (-informat D):

Ο DineroIV δέχεται το trace σε συγκεκριμένο format. Το -informat D καθορίζει ότι η είσοδος έχει Din format. Αυτό σημαίνει ότι κάθε γραμμή αποτελείται από τον τύπο πρόσβασης (r – read, w – write), τη διεύθυνση μνήμης και το μέγεθος της πρόσβασης σε bytes. Το exp_input_script.py μετατρέπει το trace του Pintool σε αυτό το format ώστε να είναι αποδεκτό input για τον DineroIV.

Cache sizes (-l1-dsize):

Τα πειράματα έχουν εκτελεστεί για cache sizes από 1KB έως 512MB, καλύπτοντας ένα ευρύ φάσμα με σκοπό να αντιπροσωπεύει τόσο μικρές caches όσο και μεγάλες caches. Στο μοντέλο της παρούσας εργασίας η DRAM λειτουργεί ως cache μπροστά από την NVM, άρα τα cache sizes αντιπροσωπεύουν διαφορετικά μεγέθη DRAM cache. Ο σκοπός της πειραματικής μελέτης σε μεγάλο εύρος cache sizes είναι να αποτυπωθεί η συμπεριφορά των write policies σε όλο το φάσμα των δυνατών μεγεθών, από το worst-case με πολύ μικρό μέγεθος DRAM, μέχρι το best-case όπου η DRAM cache είναι αρκετά μεγάλη ώστε να χωράει το μεγαλύτερο μέρος του working set.

Block size (-l1-dbsize 64):

Το block size ορίστηκε σε 64 bytes, τυπική τιμή για σύγχρονες caches. Αντιστοιχεί στο μέγεθος μιας cache line και είναι η τιμή που χρησιμοποιείται ευρέως στην βιβλιογραφία για μελέτες cache συμπεριφοράς.

Associativity (-l1-dassoc 8):

Το associativity όπως αναλύθηκε στο 2.4 καθορίζει πόσα blocks μπορούν να ανατεθούν στο ίδιο set ταυτόχρονα. Ορίστηκε σε 8-way set associative, μια τιμή που αντιπροσωπεύει ένα ρεαλιστικό επίπεδο για σύγχρονες cache.

Replacement Policy:

Χρησιμοποιείται LRU πολιτική η οποία είναι default για DineroIV και αντιπροσωπεύει ρεαλιστική συμπεριφορά cache.

Write Policy:

Η write policy ορίζεται ανά trace είτε Write-Back για volatile, είτε Write-Through για persistent και για το original trace τα πειράματα μελετούν και τις δύο πολιτικές ώστε τα αποτελέσματα να χρησιμοποιηθούν για ως σημείο σύγκρισης με την selective policy.

Write Allocation Policy (-l1-dwalloc):

Η παράμετρος για write allocation policy ορίζεται επίσης ανάλογα με το trace στο οποίο γίνεται το simulation.

Για το trace pinatrace_wb.out το οποίο περιέχει τις προσβάσεις για volatile objects:

Χρησιμοποιείται -l1-dwalloc a: write-allocate, σε κάθε write miss φέρνεται το block στη cache (DRAM). Αυτό είναι και η πιο φυσιολογική επιλογή για WB γιατί οι επόμενες εγγραφές στο ίδιο block θα είναι hits.

Για το trace pinatrace_wt.out το οποίο περιέχει τις προσβάσεις για persistent objects:

Χρησιμοποιείται -l1-dwalloc n: no-write-allocate για write-through policy, σε κάθε miss στην cache (DRAM) το επιθυμητό block δεν φέρνεται στην cache αλλά γίνεται εγγραφή απευθείας στην NVM. Με αυτό τον τρόπο προσομοιώνουμε σωστά την υπό μελέτη μεθοδολογία της παρούσας εργασίας για την οποία αγνοούμε την λειτουργία της DRAM για προσβάσεις σε persistent objects αφού δεν έρχεται το block στην DRAM για τα write-misses. Κάθε write miss αντιπροσωπεύει ένα πραγματικό NVM write που δεν μπορεί να αποφευχθεί.

4.5 Κατηγοριοποίηση Persistent/Volatile Objects

Για την επίτευξη της selective policy που βασίζετε η μελέτη της παρούσας εργασίας, είναι σημαντικό να γίνει σωστός διαχωρισμός των memory objects του Memcached στις κατηγορίες Persistent και Volatile objects. Ο διαχωρισμός αυτός βασίζεται στην ανάλυση του Memcached 1.6.14 source code ώστε να καθοριστούν ποια objects θα προσομοιωθούν με Write-Through και ποια με Write-Back policy.

Η κατηγοριοποίηση έγινε βάσει τριών κριτηρίων:

i. Τί αποθηκεύει το object – Δηλαδή αν περιέχει δεδομένα που είναι επίμονα προς την μνήμη ή αν είναι runtime δεδομένα που χάνονται και αναδημιουργούνται σε κάθε επανεκκίνηση του server.
ii. Σε συνέχεια του πρώτου, μελετήθηκε αν το object πρέπει αναγκαστικά να είναι συνεπές στην NVM. – Δηλαδή αν σε τυχόν power failure η απώλεια του object θα οδηγούσε σε σημαντική απώλεια δεδομένων ή και αδυναμία ανάκτησης τους.
iii. Από την άλλη πλευρά μελετήθηκε αν η απώλεια του εν λόγω object είναι αποδεκτή και το Memcached μπορεί να το δημιουργήσει ξανά κατά την επανεκκίνηση του server χωρίς απώλεια δεδομένων.

Τα objects που πληρούν το δεύτερο κριτήριο έχουν κατηγοριοποιηθεί ως persistent και προσομοιώνονται με Write-Through policy. Τα υπόλοιπα τα οποία εμφανίζονται εφήμερα, μη επίμονα προς την μνήμη και η απώλεια τους είναι αποδεκτή, έχουν κατηγοριοποιηθεί ως volatile και προσομοιώνονται με Write-Back policy.

Persistent Objects:

Σε αυτή την κατηγορία, μετά από ανάλυση του source code, έχουν ενταχθεί τα slabs.c:615 και assoc.c:121. Το slabs.c:615 φαίνεται να είναι το allocation site για τα πραγματικά key-value data του Memcached. Κάθε αποθηκευμένο item περιλαμβάνει το header, key και την τιμή του. Αυτά τα items είναι τα δεδομένα που αποθηκεύει ο χρήστης στο Memcached και πρέπει να παραμένουν ενημερωμένα στην NVM καθ' όλη την διάρκεια της ζωής τους. Σε power failure είναι κρίσιμο τα δεδομένα αυτά να παραμένουν αποθηκευμένα και ανακτήσιμα.

File: slabs.c | Line:615:

```
// function: do_slabs_alloc()
if (p->sl_curr != 0) {
    // return off our freelist
    it = (item *)p->slots;
```

```
            p->slots = it->next;
            if (it->next) it->next->prev = 0;
            it->it_flags &= ~ITEM_SLABBED;
            it->refcount = 1;
            p->sl_curr--;
            ret = (void *)it;
        }
```

Η do_slabs_alloc() εντοπίζει ένα διαθέσιμο block μνήμης από το freelist του slab allocator και το επιστρέφει για αποθήκευση νέου key-value item. Το freelist είναι μια λίστα από pre-allocated blocks από την μνήμη που είναι διαθέσιμα για επαναχρησιμοποίηση. Στην περίπτωση του Memcached, όταν ένα item διαγραφεί, τοποθετείται στο freelist του αντίστοιχου slab class. Με την εγγραφή νέου item με ίδιο μέγεθος ο allocator παίρνει block από freelist αντί να δεσμευτεί μνήμη από το λειτουργικό σύστημα. Στον κώδικα βλέπουμε το head του freelist (p->slots) και τον αριθμό τον διαθέσιμων blocks (p->sl_curr). Αν το freelist είναι άδειο (sl_curr == 0), τότε δεσμεύεται νέο slab page από τη μνήμη.

Το assoc.c:121 είναι το allocation site του hash table index, δηλαδή είναι η δομή που επιτρέπει την εύρεση των keys στην μνήμη. Σε περίπτωση που το hash table δεν είναι ενημερωμένο πλήρως, τα αποθηκευμένα keys είναι αδύνατο να βρεθούν ακόμα και αν τα δεδομένα τους υπάρχουν στην μνήμη και είναι πλήρως ενημερωμένα. Αυτό καθιστά την κατηγοριοποίηση του σε persistent object αναπόφευκτη.

File: assoc.c | Line:121:

```
static void assoc_expand(void) {
    old_hashtable = primary_hashtable;
    primary_hashtable = calloc(hashsize(hashpower +
1), sizeof(void *));
    if (primary_hashtable) {
        hashpower++;
        expanding = true;
```

```
            ...
        }
    }
```

Το Memcached υλοποιεί το hash table ως ένα πίνακα από pointers και κάθε θέση του πίνακα αυτού είναι ένα bucket που δείχνει στο head ενός linked list από items. Όταν δύο keys έχουν το ίδιο hash value, όταν δηλαδή ανήκουν στο ίδιο bucket, τα αντίστοιχα items δημιουργούν μια αλυσίδα(chaining) μέσω του πεδίου h_next του κάθε item. Η συνάρτηση assoc_expand() είναι υπεύθυνη για να δεσμεύσει νέο και μεγαλύτερο hash table όταν τα αποθηκευμένα items ξεπεράσουν το 150% της τρέχουσας χωρητικότητας και το chaining είναι σε βαθμό που επηρεάζει την απόδοση της αναζήτησης. Όσο περισσότερα items υπάρχουν σε κάθε bucket, τόσο μεγαλώνει και η αλυσίδα που πρέπει να διασχιστεί για κάθε αναζήτηση. Το assoc.c:121 είναι το calloc που δεσμεύει το νέο hash table. Στην περίπτωση που το hash table δεν είναι σωστό τα keys δεν μπορούν να εντοπιστούν στην μνήμη και γι' αυτό είναι σημαντικό να είναι επίμονο στην μνήμη το object αυτό.

Volatile Objects:

Τα volatile objects είναι αυτά που αντιπροσωπεύουν τα εφήμερα runtime δεδομένα του Memcached, δομές δηλαδή που δημιουργούνται κατά την εκκίνηση ή κατά την διάρκεια εκτέλεσης του συστήματος και σε κάθε επανεκκίνηση αναδημιουργούνται. Επίσης η απώλεια αυτών, σε power failure δεν θα προκαλέσει απώλεια δεδομένων και το Memcached μπορεί να επανεκκινήσει χωρίς κανένα πρόβλημα. Έτσι αυτά τα objects μπορούν να προσομοιωθούν με write-back policy.

File: cache.c | Line:19 AND Line:84:

```
/* cache.c – cache_create() – line 19 */
cache_t* cache_create(const char *name, size_t bufsize,
size_t align) {
    cache_t* ret = calloc(1, sizeof(cache_t));
    ...
```

```
}

/* cache.c – do_cache_alloc() – line 84 */
void* do_cache_alloc(cache_t *cache) {
    ...
    } else if (cache->limit == 0 || cache->total < cache-
>limit) {
        object = ret = malloc(cache->bufsize);
        ...
    }
}
```

Το cache.c είναι υπεύθυνο για την διαχείριση ενός pool από connection objects, ένα σύνολο δηλαδή από pre-allocated conn structs έτοιμα για επαναχρησιμοποίηση. Τα conn structs είναι δομές που αντιπροσωπεύουν TCP connections και αποθηκεύουν την κατάσταση του(socket, read buffer, I/O function pointers και TCP state). Το Memcached διατηρεί μια freelist από conn objects που έχουν ήδη αποδεσμευτεί και περιμένουν για επαναχρησιμοποίηση αντί να δεσμεύετε νέο conn struct σε κάθε σύνδεση. Η cache_create() συνάρτηση από cache.c:19 δεσμεύει το cache_t struct που διαχειρίζεται αυτό το freelist και η do_cache_alloc() από cache.c:84 δεσμεύει νέο conn object με malloc μόνο όταν το freelist είναι άδειο. Και τα δύο objects είναι runtime infrastructure και δεν περιέχουν δεδομένα του χρήστη που πρέπει να είναι επίμονα στην μνήμη.

File: memcached.c | Line:656:

```
/* memcached.c – conn_new() – line 656 */
conn *conn_new(const int sfd, enum conn_states init_state,
                const int event_flags,
                const int read_buffer_size,
                enum network_transport transport,
                struct event_base *base, void *ssl) {
    conn *c;
```

```
    ...
    if (NULL == c) {
        if (!(c = (conn *)calloc(1, sizeof(conn)))) {
            fprintf(stderr, "Failed to allocate connection
object\n");
            return NULL;
        }
        c->read = NULL;
        c->sendmsg = NULL;
        c->write = NULL;
        c->rbuf = NULL;
        c->rsize = read_buffer_size;
        if (c->rsize) {
            c->rbuf = (char *)malloc((size_t)c->rsize);
        }
        ...
    }
}
```

Η conn_new() του αρχείου memcached.c:656 είναι υπεύθυνη για την δέσμευση νέου conn struct για κάθε νέο TCP connection που έρχεται στο Memcached. Ο conn struct αποθηκεύει ολόκληρη την κατάσταση του TCP connection, το read buffer (rbuf) για τα εισερχόμενα δεδομένα, τα function pointers για I/O operations (read, write,sendmsg), το socket file descriptor (sfd) και το μέγεθος του read buffer (rsize). Κάθε φορά που ένας client συνδέεται δημιουργείται ένα νέο conn struct και στην αποσύνδεση το conn struct χάνεται. Άρα δεν υπάρχει νόημα να διατηρηθεί την NVM.

Τα υπόλοιπα objects ακολουθούν την ίδια λογική και είναι εφήμερα runtime δεδομένα που δεν περιέχουν user data που επίσης αναδημιουργούνται κατά την επανεκκίνηση.

Allocation Site (Object)	*Περιγραφή*	*Κατηγορία*
slabs.c:615	Key-value data + item metadata	Persistent

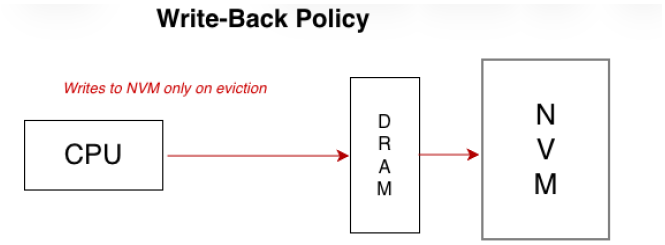
	(header, key, value)	
assoc.c:121	Hash table index — εύρεση keys	Persistent
cache.c:84	Object cache freelist — runtime pool για connection structs	Volatile
memcached.c:656	Connection structs (TCP state machine)	Volatile
thread.c:1051	Worker thread structures	Volatile
thread.c:1042	Item mutex locks (pthread_mutex_t)	Volatile
bipbuffer.c:51	Network I/O buffers ανά connection	Volatile
logger.c:824	Per-thread logger + bipbuffer	Volatile
cache.c:19	cache_t struct (cache manager)	Volatile
items.c:1273	LRU bump buffer	Volatile
thread.c:434	Connection queue ανά thread	Volatile
memcached.c:484	Connection pointer array	Volatile
arena.c:518/509	glibc allocator internals	Volatile

Πίνακας 4.4: Persistent vs Volatile Objects Description

4.6 Μεθοδολογία Υπολογισμού Write Traffic

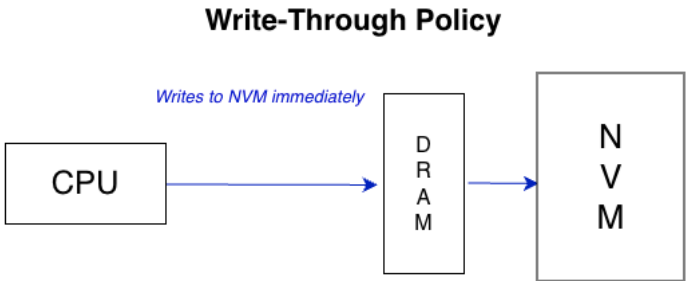
Το write traffic αντιπροσωπεύει τον συνολικό όγκο δεδομένων που κινούνται προς την μνήμη κατά την εκτέλεση των πειραμάτων. Για να υπολογιστεί δεν χρειάζεται κάποια επιπλέον επεξεργασία, ότι είναι απαραίτητο το δίνει ο DineroIV στο output του όταν εξάγει την τιμή bytes_to_memory. Το bytes_to_memory το σύνολο των bytes που στάλθηκαν προς την μνήμη κατά την διάρκεια του simulation.

Για Write-Back policy, τα writes κατευθύνονται προς την NVM όταν θα γίνει το eviction του dirty block από την DRAM cache [11]. Αυτό σημαίνει ότι το metric του bytes_to_memory αντιπροσωπεύει τα dirty evictions που γίνονται. Κατά το eviction στην μνήμη γράφεται ολόκληρο το block (64bytes) είτε η αλλαγή που έγινε ήταν μικρότερη από το block είτε όχι, είναι ανεξάρτητο δηλαδή από το πόσα bytes τροποποιήθηκαν [22][28].



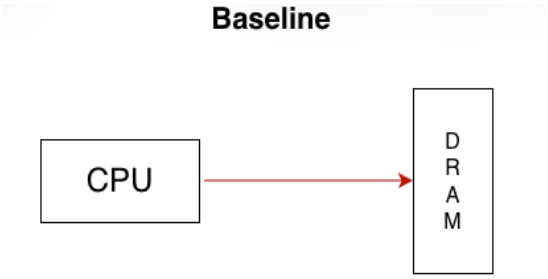
Σχήμα 4.1: Write-Back Policy

Για Write-Through policy με no-write allocate όλα τα writes ανεξαρτήτως αν είναι hit ή miss γράφονται απευθείας στην NVM [22][11]. Όταν γίνεται write hit το block σημαίνει υπάρχει ήδη στην cache μόνο από προηγούμενο read γιατί κανένα write δεν μπορεί να φέρει block με no write allocate στην cache. Τότε λοιπόν η εγγραφή των τροποποιημένων blocks γίνεται και στην cache (DRAM) αλλά και στην μνήμη NVM ταυτόχρονα[11]. Σε write miss ωστόσο, με no write allocate το block δεν γίνεται fetch από την μνήμη, τα τροποποιημένα bytes εγγράφονται απευθείας στην NVM χωρίς να αγγιχθεί η DRAM [22]. Και στις δύο περιπτώσεις με write-through γράφονται στην NVM μόνο τα bytes που τροποποιήθηκαν, όχι ολόκληρο το block [28].



Σχήμα 4.2: Write-Through Policy

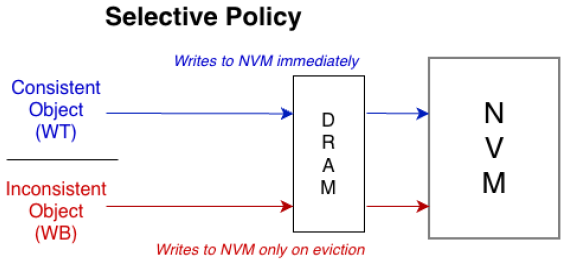
Για το Baseline σενάριο δεν υπάρχει cache όπως είδαμε και στα σχεδιαγράμματα τοπολογίας στο υποκεφάλαιο 1.3. Κάθε πρόσβαση οδηγείται απευθείας στην DRAM. Δεν υπάρχει NVM στο baseline.



Σχήμα 4.3: Baseline

Για την selective πολιτική το write traffic υπολογίζεται ως το άθροισμα του write traffic των simulations για volatile objects με WB και για persistent objects με WT. Αυτό είναι το lower bound, δηλαδή είναι το best-case scenario.

Lower Bound = WB write traffic (volatile) + WT write traffic (persistent)



Σχήμα 4.4: Selective Policy

Το lower bound είναι μια παραδοχή της παρούσας μελέτης αφού λόγω segregation, όπου τα δύο traces για persistent και volatile γίνονται simulate ξεχωριστά, δεν ανταγωνίζονται για τον χώρο στην ίδια DRAM cache. Στην πραγματικότητα τα δύο αυτά simulations εκτελούνται σαν ένα και μοιράζονται την ίδια DRAM cache. Κάθε φορά που ένα persistent object κάνει read miss, όχι write miss λόγο no write allocate, φέρνει ένα block στην DRAM cache και ενδέχεται να εκδιώξει ένα dirty volatile block. Αυτό προκαλεί ένα επιπλέον write στην NVM που δεν αποτυπώνεται στο lower bound – best case scenario.

Για να ληφθεί υπόψη το worst case scenario πρέπει να υπολογιστεί το upper bound. Η παραδοχή είναι ότι κάθε persistent miss κάνει πάντα evict ένα dirty volatile block όπως εξηγήθηκε παραπάνω. Γι' αυτό τον λόγο στο upper bound η υπόθεση είναι ότι σε κάθε miss από persistent object γίνεται eviction και επιπλέον 64bytes γίνονται write στην NVM.

$$\text{Upper bound} = \text{Lower bound} + \text{persistent\_misses} * 64$$

Στην πραγματικότητα το πραγματικό write traffic βρίσκεται κάπου στην μέση lower – upper bound. Ωστόσο, για την παρούσα εργασία λήφθηκε υπόψη μόνο το lower bound και αυτό αποτελεί γνωστή παραδοχή της μελέτης.

4.7 Μεθοδολογία υπολογισμού AMAT (cycles/access)

Η αρχιτεκτονική μνήμης αποτελείται από δύο βασικά σενάρια:

i. Baseline: CPU ⇒ DRAM (κύρια μνήμη, χωρίς cache)
ii. Experimental: CPU ⇒ DRAM (cache) ⇒ NVM (κύρια μνήμη)

Στο experimental σενάριο η DRAM είναι cache μπροστά από την NVM χωρίς να υπάρχουν L1/L2 CPU cache στο μοντέλο. Σκοπός είναι να μελετηθεί αν η DRAM μπορεί να αντικατασταθεί από NVM με αποδεκτό κόστος απόδοσης, χρησιμοποιώντας την DRAM ως cache μπροστά από την NVM.

Επίπεδο	*Latency*	*Source*
T_DRAM	100 cycles	[18] [4]
T_NVM_read	300 cycles (~ DRAMx3)	[18]
T_NVM_write	1000 cycles	[23] [16]

Πίνακας 4.5: Κόστος προσβάσεων

Baseline AMAT:

Στο baseline δεν υπάρχει cache, υπάρχει μόνο DRAM ως κύρια μνήμη. Κάθε access κατευθύνεται απευθείας στην DRAM και άρα κάθε πρόσβαση έχει σταθερό κόστος:

$$AMAT\_baseline = T\_DRAM = 100 \text{ cycles}$$

Στατιστικά DineroIV & AMAT:

Ο υπολογισμός του AMAT είναι βασισμένος στα στατιστικά που εξάγει ο DineroIV. Συγκεκριμένα παράγει το Demand Fetches το οποίο είναι το σύνολο των προσβάσεων και το Demand Misses που είναι το σύνολο των προσβάσεων για τις οποίες το block δεν βρισκόταν στην cache, όπως ακριβώς αναλύθηκαν στο υποκεφάλαιο 4.3. Τα μεγέθη αυτά λειτουργούν σε cache block επίπεδο καθώς όπως ήδη αναλύθηκε κάθε memory reference που αγγίζει N συνεχόμενα blocks καταγράφεται ως N ξεχωριστά block lookups. Όπως αναλύθηκε επίσης από DineroIV έχουμε τα εξής:

```
Read Misses = Demand Misses (Read)
Write Misses = Demand Misses (Write)
Read Hits = Demand Fetches (Read) – Demand Misses (Read)
Write Hits = Demand Fetches (Write) – Demand Misses (Write)
Total = Demand Fetches (Total)
```

AMAT για WB Policy:

Access	*Κόστος*	*Αιτιολόγηση*
Read Hit	100c	T_DRAM: Block υπάρχει στη DRAM.
Write Hit	100c	T_DRAM: WB δεν γράφει στη NVM κατά hit.
Read Miss	100c + 300c	T_DRAM + T_NVM_read: fetching block from NVM.
Write Miss	100c + 1000c	T-DRAM (lookup) + T_NVM_write: dirty eviction.

Πίνακας 4.6: Write Back AMAT

$$AMAT\_WB = (Read\_Hit * 100 + Write\_Hit * 100 + Read\_Miss * 400 +$$

Write_Miss * 1100) / Total accesses

AMAT για WT Policy:

Το WT εκτελείται με no write allocate, κάθε write πηγαίνει απευθείας στην NVM χωρίς να γίνετε fetch το block στην cache.

Access	*Κόστος*	*Αιτιολόγηση*
Read Hit	100c	Block υπάρχει στη DRAM
Write Hit	100c + 1000c	T_DRAM + T_NVM_write: WT γράφει στη NVM
Read Miss	100c + 300c	T_DRAM + T_NVM_read: fetch block από NVM
Write Miss	100c + 1000c	T_DRAM + T_NVM_write: WT γράφει στη NVM

Πίνακας 4.7: Write Through AMAT

AMAT_WT = (Read_Hit * 100 +
Write_Hit * 1100 +
Read_Miss * 400 +
Write_Miss * 1100) / Total accesses

AMAT για Selective Policy:

Για Selective policy εφαρμόζεται WB στα volatile και WT στα persistent objects. Λόγο του ότι τα δύο simulations τρέχουν ξεχωριστά στην μεθοδολογία της εργασίας, το Selective AMAT υπολογίζεται ως weighted average των δύο επιμέρους AMAT σε κάθε πείραμα:

AMAT_Selective = (Incons_accesses × incons_AMAT_WB +
Cons_accesses × cons_AMAT_WT) /
(Incons_accesses + Cons_accesses)

Υπολογίζεται δηλαδή το AMAT για τα δύο ξεχωριστά πειράματα Volatile WB και Persistent WT και στην συνέχεια υπολογίζεται το weighted average. Αυτό αποτελεί lower bound αφού δεν λαμβάνεται υπόψη η αλληλεπίδραση μεταξύ των δύο objects στην κοινή DRAM όπως αναλύθηκε στο 4.6.

Κεφάλαιο 5

Αποτελέσματα & Ανάλυση

Πριν από την παρουσίαση των αποτελεσμάτων είναι σημαντικό να αναφερθούν μερικά βασικά χαρακτηριστικά:

◊ Item size = ~289 bytes (key = 30 bytes + Value = 200 bytes + Metadata)
Αυτό συλλέχθηκε από το Memcached με την χρήση Memcached stats. Από τα Memcached stats είναι προσβάσιμο το μέγεθος που καταλαμβάνουν τα δεδομένα που φορτώθηκα στην μνήμη του Memcached. Χαρακτηριστικά, STATS bytes = 289000000 / 1 εκατομμύριο keys που φορτώθηκαν. Λόγο όμως του slab allocator που χωρίζει την μνήμη σε 1MB slab pages και χωρίζει την κάθε μια σε σταθερού μεγέθους chunks τα διαθέσιμα chunks χωρίζονται σε slab classes. Το item εμπίπτει στο Class 6 με Chunk = 304B και γι' αυτό έχει ~15B padding.

Από Memcached stats:

STAT 6:chunk_size 304

STAT items:6:mem_requested 289000000

STAT 6:used_chunks 1000000

STAT items:6:evicted 0

STAT limit_maxbytes

◊ Dataset size = ~ 290MB (Από persistent objects – items)

◊ QPS = 121

◊ Total queries = 25.410

◊ Total Memory Accesses = 9.685.053 (61% reads + 39% εσωτερικά writes)

Τα αναπόφευκτα εσωτερικά writes του Memcached είναι ενημερώσεις reference counters, διαχείριση LRU λιστών, ενημερώσεις connection state machine και thread queues.

Persistent objects — 3 writes/query:

- do_item_get (items.c:959): refcount++
- do_item_remove (items.c:542): refcount--
- do_item_bump (items.c:1051-1055): LRU position update

Volatile objects — ~143.9 writes/query:

- cache.c:84 (response pool): ~78.2/query — resp_allocate, resp_add_iov, memset, resp_free, resp_finish
- Memcached.c:656 (connection state): ~27.6/query — conn_set_state, drive_machine, reset_cmd_handler
- Thread.c:1051 (thread queues): ~25.2/query
- Thread.c:1042 (item locks): ~11.8/query
- Λοιπά: ~1.1/query

◊ Accesses per Query = 381.2

◊ Persistent objects accesses = 1.054.989 – 42.2 per query

◊ Volatile objects accesses = 8.658.186 – 346.5 per query

Ονοματολογία Πειραμάτων:

Original WB	Πειράματα με το original trace (persistent + volatile objects) με Write-Back policy.
Original WT	Πειράματα με το original trace (persistent + volatile objects) με Write-Through policy.

Persistent WT	Πειράματα για Persistent objects μόνο, με Write-Through policy.
Volatile WB	Πειράματα για Volatile objects μόνο, με Write-Back policy.
Selective	Συνδυασμός αποτελεσμάτων Persistent WT και Volatile WB, για τον υπολογισμό των αποτελεσμάτων Selective policy.

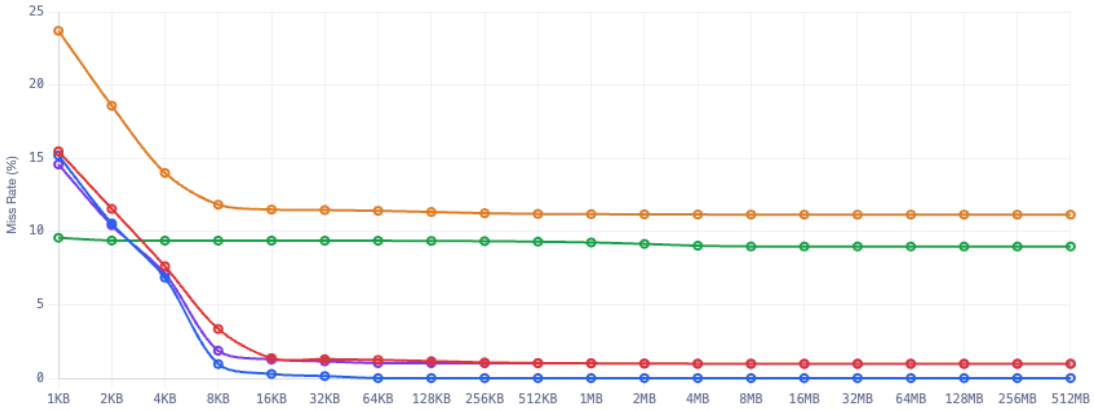
Πίνακας 5.1: Ονοματολογία Πειραμάτων

5.1 Miss rate

Στον παρακάτω πίνακα παρουσιάζονται τα αποτελέσματα από την εκτέλεση των πειραμάτων για τα 3 κύρια σενάρια που μελετώνται στην παρούσα εργασία, καθώς και οι στήλες Volatile WB και Persistent WT για τις οποίες μελετάτε η συμπεριφορά των objects στην μεταβολή του cache size.

Cache Size	*Original WB*	*Original WT*	*Volatile WB*	*Persistent WT*	*Selective (lower-bound)*
1KB	15.4828%	23.7032%	15.2000%	9.5782%	14.5894%
2KB	11.5598%	18.5895%	10.5633%	9.3911%	10.4360%
4KB	7.6365%	14.0025%	6.8587%	9.3870%	7.1333%
8KB	3.3540%	11.8358%	0.9608%	9.3865%	1.8759%
16KB	1.3678%	11.5074%	0.2902%	9.3855%	1.2780%
32KB	1.3014%	11.4714%	0.1460%	9.3824%	1.1492%
64KB	1.2481%	11.4195%	0.0089%	9.3769%	1.0264%
128KB	1.1679%	11.3412%	0.0089%	9.3669%	1.0253%
256KB	1.0818%	11.2580%	0.0089%	9.3489%	1.0234%
512KB	1.0317%	11.2089%	0.0089%	9.3159%	1.0198%
1MB	1.0149%	11.1926%	0.0089%	9.2588%	1.0136%
2MB	1.0034%	11.1811%	0.0089%	9.1608%	1.0030%
4MB	0.9902%	11.1680%	0.0089%	9.0409%	0.9899%
8MB	0.9842%	11.1620%	0.0089%	8.9878%	0.9842%
16MB	0.9837%	11.1615%	0.0089%	8.9837%	0.9837%
32MB	0.9837%	11.1615%	0.0089%	8.9837%	0.9837%
64MB	0.9837%	11.1615%	0.0089%	8.9837%	0.9837%
128MB	0.9837%	11.1615%	0.0089%	8.9837%	0.9837%
256MB	0.9837%	11.1615%	0.0089%	8.9837%	0.9837%
512MB	0.9837%	11.1615%	0.0089%	8.9837%	0.9837%

Πίνακας 5.2: Αποτελέσματα Miss Rate



Γράφημα 5.1: Γραφική Παράσταση Miss Rate

Παρατηρείται ότι τα volatile objects (Write Back) παρουσιάζουν αισθητή πτώση miss rate στην μετάβαση του cache size από 4KB σε 8KB, από 6.86% σε 0.96%, δηλαδή μείωση 86%. Αυτό συμβαίνει γιατί το working set τους αποτελείται κυρίως από infrastructure δομές του Memcached (connection structs, thread queues, LRU metadata) οι οποίες προσπελαύνονται από το Memcached επανειλημμένα για κάθε query. Λόγο αυτού, είναι αρκετό να γίνει μια φορά miss, να γίνουν fetch από την μνήμη τα hot cache lines που χρησιμοποιούνται συχνότερα και να είναι αρκετά μεγάλη η cache ώστε να τα χωράει. Από τα 64KB και μετά, καθώς το cache size μεγαλώνει, βλέπουμε το miss rate για volatile να σταθεροποιείται και να φτάνει τιμή 0.0089%, πρακτικά μηδενική.

Αντίθετα τα persistent objects (Write through) παρατηρείται ότι διατηρούν ένα σταθερό miss rate σχετικά σε όλο το εύρος τιμών παρ' όλο που υπάρχει μια μικρή πτώση του ποσοστού από cache size 16MB και για όσο μεγαλώνει. Δηλαδή παρατηρείται να κινείται από ~9.38% που παραμένει σταθερό μέχρι 16MB, στα ~8.98% και σταθεροποιείται. Αυτή η συμπεριφορά οφείλεται σε πολλούς παράγοντες. Αρχικά, το μεγάλο working set των persistent objects, 1 εκατομμύριο κλειδιά (~290MB), τα οποία δεν χωράνε ποτέ πλήρως στην cache. Η ομοιόμορφη κατανομή που ακολουθούν τα queries (uniform distribution), εξαλείφει την temporal locality και κάνει την επιλογή των keys πιο τυχαία. Επίσης, ο σημαντικότερος λόγος είναι ότι το load phase αγνοείται από το pintool για να γίνει εστίαση σε read-only trace, η cache στο measurement phase κάνει cold start. Αυτό έχει ως αποτέλεσμα το dataset να μην είναι loaded στην cache από την αρχή και τα cold misses να είναι αναπόφευκτα. Ακόμα και αν το cache size

ήταν πολύ μεγάλο ποτέ δεν θα μειωνόταν το miss rate αφού με uniform distribution σε συνδυασμό με cold start της cache, το dataset θα χρειαζόταν πολύ μεγάλο αριθμό queries για να φορτωθεί πλήρως στην cache. Αυτοί είναι οι βασικότεροι λόγοι που δικαιολογούν την συμπεριφορά του miss rate για persistent objects.

Η Selective policy φαίνεται να κυριαρχείται από την συμπεριφορά των volatile objects αφού ~90% των συνολικών προσβάσεων είναι από αυτά. Έτσι παρατηρείται μια πτώση από 14,59% στο 1KB, σε 0.98% από 16MB και μετά όπου θεωρητικά ακουμπά μηδενική τιμή.

Οι μετρήσεις Original WB και Original WT αντικατοπτρίζουν το συνδυασμένο working set ολόκληρου του trace (persistent + volatile objects). Το miss rate για Original WB σταθεροποιείται στο ~0,98% για μεγάλα size cache, ενώ το miss rate για Original WT παραμένει σταθερά υψηλό στο ~11,16% λόγω χρήσης no write allocate για allocate policy το οποίο εμποδίζει την τοποθέτηση blocks στην cache από την μνήμη όταν γίνεται write miss. Επίσης επηρεάζεται και από το γεγονός ότι τα volatile objects είναι αυτά που καταλαμβάνουν το μεγαλύτερο ποσοστό από τα accesses per query. Αυτό αποτελεί εμπόδιο στα persistent blocks για την παραμονή τους στην cache.

Το σημαντικότερο εύρημα από αυτά τα σενάρια, αποτελεί η σύγκριση Selective policy με Original WB και Original WT. Έναντι του Original WT η Selective policy υπερτερεί σε όλα τα cache sizes. Στα 8KB cache size το miss rate του Selective policy είναι 1.88% έναντι 11.84% του Original WT, δηλαδή περίπου 85% χαμηλότερο, ενώ για μεγαλύτερα cache sizes η διαφορά φτάνει γύρο στο 90%. Αυτό οφείλεται στο ότι το no write allocate δεν επιτρέπει να τοποθετηθούν τα blocks στην cache μετά από write miss, αυξάνοντας τεχνητά το miss rate. Έναντι του Original WB το Selective policy είναι ελαφρώς καλύτερο σε μικρά έως μεσαία cache sizes ενώ παρατηρείται ότι για μεγαλύτερα μεγέθη cache οι δύο πολιτικές συγκλίνουν προς το 0.98%. Η Selective policy δεν φαίνεται να θυσιάζει miss rate σε σχέση με Original WB.

Ο τύπος για τον υπολογισμό του Selective Miss Rate:

Selective Miss Rate = (Incons_ReadMisses +

Incons_WriteMisses +

Cons_ReadMisses +

Cons_WriteMisses) ÷

(Incons_Total + Cons_Total) × 100

Ο πίνακας τιμών που εφαρμόζονται στον τύπο για εξαγωγή του Selective Miss Rate:

Cache	*Cons. WT Read Misses*	*Cons. WT Write Misses*	*Cons. WT Total Acc.*	*Incons. WB Read Misses*	*Incons. WB Write Misses*	*Incons. WB Total Acc.*	*Selective Miss Rate*
1KB	101.049	0	1.054.989	692.285	623.761	8.658.186	14.5894%
2KB	99.075	0	1.054.989	408.851	505.738	8.658.186	10.4360%
4KB	99.032	0	1.054.989	152.500	441.341	8.658.186	7.1333%
8KB	99.027	0	1.054.989	50.470	32.717	8.658.186	1.8759%
16KB	99.016	0	1.054.989	24.958	165	8.658.186	1.2780%
32KB	98.983	0	1.054.989	12.531	113	8.658.186	1.1492%
64KB	98.925	0	1.054.989	675	99	8.658.186	1.0264%
128KB	98.820	0	1.054.989	675	99	8.658.186	1.0254%
256KB	98.630	0	1.054.989	675	99	8.658.186	1.0234%
512KB	98.282	0	1.054.989	675	99	8.658.186	1.0198%
1MB	97.679	0	1.054.989	675	99	8.658.186	1.0136%
2MB	96.645	0	1.054.989	675	99	8.658.186	1.0030%
4MB	95.381	0	1.054.989	675	99	8.658.186	0.9899%
8MB	94.820	0	1.054.989	675	99	8.658.186	0.9842%
16MB	94.777	0	1.054.989	675	99	8.658.186	0.9837%
32MB	94.777	0	1.054.989	675	99	8.658.186	0.9837%
64MB	94.777	0	1.054.989	675	99	8.658.186	0.9837%
128MB	94.777	0	1.054.989	675	99	8.658.186	0.9837%
256MB	94.777	0	1.054.989	675	99	8.658.186	0.9837%
512MB	94.777	0	1.054.989	675	99	8.658.186	0.9837%

Πίνακας 5.3: Αναλυτικός υπολογισμός Selective Miss Rate με απόλυτους αριθμούς misses ανά κατηγορία object.

Αξίζει να σημειωθεί ότι ο λόγος για τον οποίο τα Persistent WT Write Misses είναι μηδενικά για όλα τα cache sizes προκύπτει καθαρά από το access pattern του Memcached. Κάθε GET request εκτελεί πρώτα ανάγνωση του hash table node (assoc_find(), assoc.c:121) και στην συνέχεια ανάγνωση του item header (item_get(), slabs.c:615). Αυτά προκαλούν read miss και το block φορτώνεται στην cache DRAM

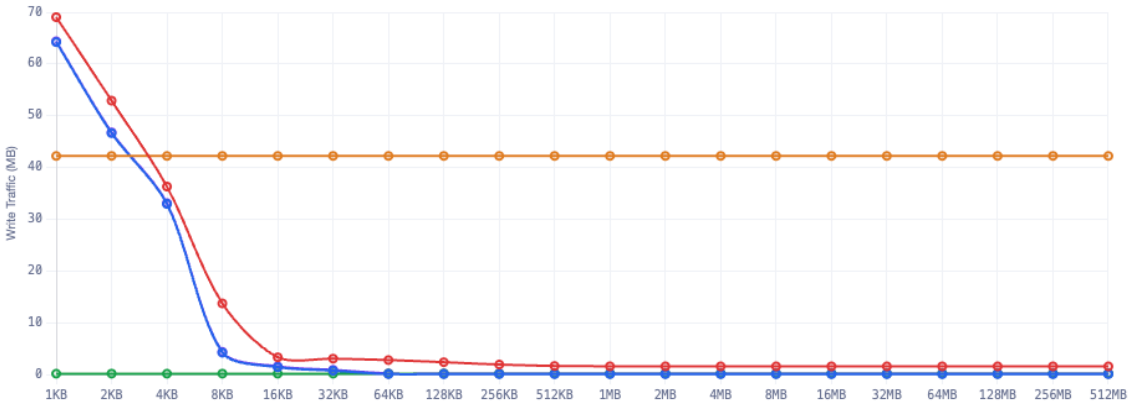
από την μνήμη NVM. Τα metadata writes που ακολουθούν μετά και αφορούν τα φορτωμένα blocks, οδηγούν σε write hit.

5.2 Write Traffic (Bytes to Memory)

Μια σημαντική έννοια που μελετάτε στην παρούσα εργασία είναι το write traffic. Το write traffic μελετά τα bytes που γράφονται στην NVM κατά την εκτέλεση του workload. Στον πίνακα 5.4 παρουσιάζονται τα αποτελέσματα για κάθε write policy σε κάθε cache size που μελετήθηκε. Στο σχήμα 5.2 απεικονίζει γραφικά την εκτέλεση του write traffic συναρτήσει του cache size.

Cache Size	*Original WB*	*Original WT*	*Volatile WB*	*Persistent WT*	*Selective (lower-bound)*
1KB	69.050 MB	42.229 MB	64.231 MB	0.144 MB	64.375 MB
2KB	52.908 MB	42.229 MB	46.625 MB	0.144 MB	46.769 MB
4KB	36.303 MB	42.229 MB	32.942 MB	0.144 MB	33.086 MB
8KB	13.716 MB	42.229 MB	4.196 MB	0.144 MB	4.339 MB
16KB	3.313 MB	42.229 MB	1.397 MB	0.144 MB	1.541 MB
32KB	3.044 MB	42.229 MB	0.726 MB	0.144 MB	0.870 MB
64KB	2.771 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
128KB	2.358 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
256KB	1.901 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
512KB	1.637 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
1MB	1.573 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
2MB	1.563 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
4MB	1.557 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
8MB	1.554 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
16MB	1.554 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
32MB	1.554 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
64MB	1.554 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
128MB	1.554 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
256MB	1.554 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB
512MB	1.554 MB	42.229 MB	0.046 MB	0.144 MB	0.190 MB

Πίνακας 5.4: Write Traffic ανά cache size.



Σχήμα 5.2: Απεικόνιση write traffic.

Ο τύπος υπολογισμού του Selective Write Traffic:

Selective Bytes To Memory = Incons_BytesToMemory + Cons_BytesToMemory

Αποτελεί το lower bound, δηλαδή το best-case scenario. Δεν λαμβάνεται υπόψη αλληλεπίδραση στην κοινή DRAM cache.

Το πρώτο στοιχείο που παρατηρείται είναι η σταθερότητα του Original WT στα 42.229MB για κάθε cache size. Αυτό συμβαίνει γιατί η Write through policy γράφει κάθε τροποποιημένο byte άμεσα στην NVM ανεξαρτήτως cache size. Η Write through γράφει συγκεκριμένα μόνο τα τροποποιημένα bytes και όχι ολόκληρο το block. Έτσι το συνολικό write traffic αντικατοπτρίζει καθαρά τον όγκο των εγγραφών του workload.

Για τα πειράματα Original WB παρατηρείται υπερβολικά μεγάλος αριθμός write traffic προς την μνήμη για μικρές caches και αυτό οφείλεται στα επαναλαμβανόμενα evictions που γίνονται λόγω των μικρών cache sizes. Όσο το cache size αυξάνεται το write traffic μειώνεται μέχρι να σταθεροποιηθεί στα 8MB και μετά για cache size, με τα bytes to memory να ανέρχονται στα 1.554MB μέχρι τα 512MB cache size. Αυτό μας παραπέμπει στο γεγονός ότι τα evictions μειώνονται σημαντικά και μεγάλο μέρος του working set παραμένει στην cache προκαλώντας Write Hits.

Το Persistent WT επίσης φαίνεται να παραμένει σταθερό στα 0.144MB για όλα τα cache sizes. Ο λόγος της συμπεριφοράς αυτής είναι ο ίδιος με το Original WT αφού σε κάθε write που εκτελείται για Hit ή Miss η write through γράφει πάντα στην NVM μόνο τα τροποποιημένα bytes. Είναι εμφανές ότι το write traffic για Persistent WT είναι σημαντικά μικρότερο. Αυτό προκαλείται από το πολύ μικρότερο ποσοστό writes που εκτελούνται σε κάθε πείραμα από Persistent objects, όπως φαίνεται και στον πίνακα 5.5. Τα writes για Persistent objects είναι μόλις 75.380 έναντι 3.649.742 που είναι τα writes των Volatile objects, δηλαδή περίπου 49 φορές λιγότερα writes. Η Selective policy το εκμεταλλεύεται αυτό και το γεγονός ότι μόνο τα Persistent objects είναι απαραίτητο να παραμένουν συνεπές στην μνήμη και εφαρμόζει μόνο σε αυτά Write Through.

Experiment	*Total Accesses*	*Reads*	*Reads %*	*Writes*	*Writes %*
Original WB	9.713.175	5.804.319	59.8%	3.725.122	38.4%
Original WT	9.713.175	5.804.319	59.8%	3.725.122	38.4%
Volatile WB	8.658.186	4.843.053	55.9%	3.649.742	42.2%
Persistent WT	1.054.989	961.266	91.1%	75.380	7.1%
Selective	9.713.175	5.804.319	59.8%	3.725.122	38.4%

Πίνακας 5.5: Συνολικός αριθμός προσβάσεων και ανάλυση σε κατηγορίες, ανά πείραμα.

Στα πειράματα Volatile WB εμφανίζεται μια συμπεριφορά παρόμοια με αυτήν στα πειράματα Original WB. Παρατηρείται ένα υπερβολικά μεγάλο write traffic για μικρά cache sizes που δικαιολογείται από τα πολλαπλά dirty evictions που προκαλούνται και στην συνέχεια το write traffic πέφτει δραματικά όταν τα cache sizes αυξηθούν. Κατά την αύξηση των cache sizes τα hot cache lines των volatile objects χωράνε στην cache και ελαχιστοποιείται το write traffic καθώς μειώνονται τα επαναλαμβανόμενα dirty evictions από την cache. Η μεταβολή αυτή είναι εμφανής από τα 8KB cache size όπου το write traffic αρχίζει να μειώνεται από 32.942MB στα 4.196MB και σταθεροποιείται στα 0.046MB από τα 64KB cache size και μετά.

Η συμπεριφορά της Selective policy στο write traffic είναι ένας συνδυασμός των Persistent και Volatile objects και τις συμπεριφοράς τους στο write traffic. Χειρίζεται τα volatile objects με Write Back ώστε να μην προκαλείται write συνεχώς στην μνήμη,

πάρα μόνο σε eviction το οποίο με μικρό cache θα μπορούσε να είναι πρόβλημα αφού η Write Back πολιτική γράφει στην μνήμη ολόκληρο το block και όχι μόνο τα τροποποιημένα bytes. Παρ' όλ' αυτά, το πρόβλημα αυτό αντιμετωπίζεται αποτελεσματικά με αύξηση του cache size. Τα persistent objects, τα χειρίζεται μέσω Write through γιατί είναι απαραίτητο να έχουν συνέπεια στην μνήμη. Αναλογικά με το Original WT θα έπρεπε να προκαλεί μεγάλο traffic αλλά λόγω του ότι είναι πολύ μικρός ο αριθμός εγγραφών των persistent objects σε κάθε query σε σχέση με τα volatile και επίσης επειδή γίνεται εγγραφή μόνο των τροποποιημένων bytes επιτυγχάνεται σημαντικά χαμηλό write traffic.

Η Selective policy εκμεταλλεύεται το καλύτερο χαρακτηριστικό από κάθε policy. Δηλαδή, εφαρμόζει Write Through σε persistent objects γιατί είναι απαραίτητο για λόγους συνέπειας των δεδομένων, γράφοντας αποκλειστικά τα τροποποιημένα bytes και Write Back σε volatile objects όπου η εγγραφή τους στην μνήμη δεν είναι ζήτημα συνέπειας, κρατώντας τα δεδομένα στην cache για να αποφευχθεί η συνεχής εγγραφή στην μνήμη. Έτσι με Selective policy αποφεύγεται το κύριο μειονέκτημα κάθε πολιτικής, η συνεχής εγγραφή στην μνήμη της Write Through και η ασυνέπεια της Write Back καθώς και η εγγραφή ολόκληρου του block που προκαλεί η Write Back κατά τα dirty evictions.

Γι' αυτούς τους λόγους η Selective policy φαίνεται σε μικρά cache sizes να έχει μεγάλο write traffic, ακριβώς όπως το πείραμα Original WB που επίσης έχουν πολύ κοντινές τιμές, με την Selective να προκαλεί ελαφρώς μικρότερο write traffic για κάθε cache size. Όταν το cache size είναι αρκετά μεγάλο για να χωράει τα hot cache lines των volatile objects, από τα 4KB στα 8KB είναι εμφανής μια σημαντική πτώση στο write traffic το οποίο επίσης παραπέμπει στην συμπεριφορά του Original WB. Ωστόσο, η Selective έχει μικρότερο write traffic και σε μεγαλύτερες cache αφού με την χρήση Write Through για κάθε persistent write εγγράφονται μόνο τα τροποποιημένα bytes.

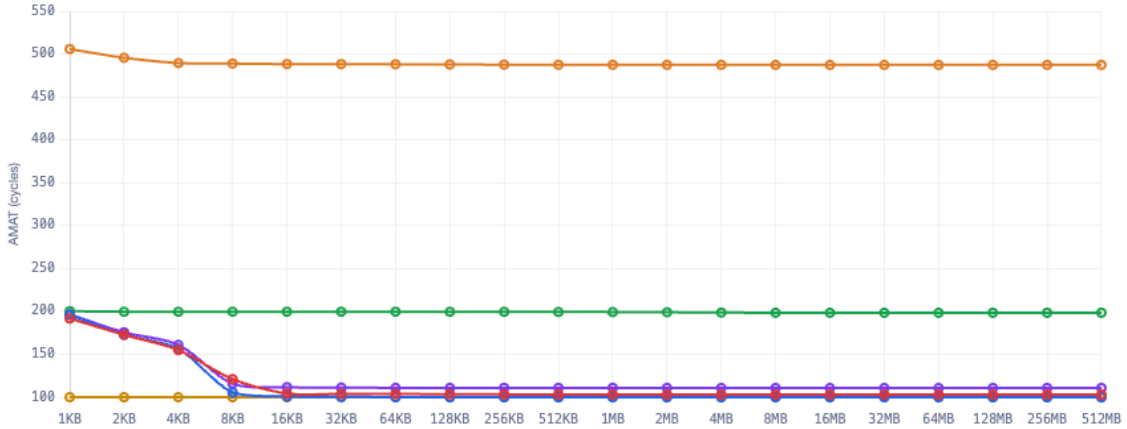
5.3 AMAT (Average Memory Access Time)

Το Average Memory Access Time (AMAT) εκφράζει τον μέσο χρόνο ανά πρόσβαση (memory access) σε cycles. Το Baseline είναι 100c που εκφράζει τον χρόνο πρόσβασης

στη DRAM. Στον Πίνακα 5.6 παρουσιάζονται τα αποτελέσματα για κάθε σενάριο και στο Σχήμα 5.3 απεικονίζεται γραφικά η συμπεριφορά του AMAT συναρτήσει του cache size. Λεπτομέρειες για τον υπολογισμό του AMAT έχουν αναλυθεί στο υποκεφάλαιο 4.7.

Cache Size	*Original WB*	*Original WT*	*Volatile WB*	*Persistent WT*	*Selective*	*Baseline*
1KB	191.6506c	520.6397c	196.0301c	200.1856c	196.4814c	100.0000c
2KB	172.7021c	509.2487c	172.5779c	199.6243c	175.5155c	100.0000c
4KB	154.9353c	501.1872c	156.2579c	199.6120c	160.9668c	100.0000c
8KB	121.0039c	499.0796c	105.5275c	199.6106c	115.7463c	100.0000c
16KB	104.2103c	498.4390c	100.8838c	199.6075c	111.6066c	100.0000c
32KB	103.9172c	498.3387c	100.4472c	199.5981c	111.2164c	100.0000c
64KB	103.7540c	498.1830c	100.0348c	199.5816c	110.8470c	100.0000c
128KB	103.5117c	497.9482c	100.0348c	199.5517c	110.8438c	100.0000c
256KB	103.2526c	497.6987c	100.0348c	199.4977c	110.8379c	100.0000c
512KB	103.1024c	497.5513c	100.0348c	199.3988c	110.8271c	100.0000c
1MB	103.0517c	497.5022c	100.0348c	199.2273c	110.8085c	100.0000c
2MB	103.0174c	497.4679c	100.0348c	198.9333c	110.7766c	100.0000c
4MB	102.9778c	497.4284c	100.0348c	198.5738c	110.7375c	100.0000c
8MB	102.9598c	497.4104c	100.0348c	198.4143c	110.7202c	100.0000c
16MB	102.9583c	497.4090c	100.0348c	198.4021c	110.7189c	100.0000c
32MB	102.9583c	497.4090c	100.0348c	198.4021c	110.7189c	100.0000c
64MB	102.9583c	497.4090c	100.0348c	198.4021c	110.7189c	100.0000c
128MB	102.9583c	497.4090c	100.0348c	198.4021c	110.7189c	100.0000c
256MB	102.9583c	497.4090c	100.0348c	198.4021c	110.7189c	100.0000c
512MB	102.9583c	497.4090c	100.0348c	198.4021c	110.7189c	100.0000c

Πίνακας 5.6: AMAT (cycles) ανά cache size για κάθε πείραμα.



Σχήμα 5.3: Γραφική εξέλιξη AMAT συναρτήσει του cache size.

5.3.1 Original WT AMAT

Το πρώτο που παρατηρείται είναι το πολύ κακό performance του Original WT όπου το AMAT παραμένει σταθερά πολύ υψηλό. Οι τιμές του κυμαίνονται από 497.41c για μεγαλύτερα cache sizes, μέχρι και 520.64c για μικρά cache sizes, αφού όπως αναλύθηκε στο υποκεφάλαιο 4.7 η τιμή AMAT εξαρτάται άμεσα από το πλήθος write hits/misses και read hits/misses. Παρ' όλ' αυτά η διαφορά είναι μικρή γιατί με πολιτική Write Through κάθε write κάνει εγγραφή στην μνήμη είτε είναι miss είτε hit και έτσι σε κάθε write χρεώνεται το write latency για NVM.

Cache	*Read Hits*	*Read Misses*	*Write Hits*	*Write Misses*	*Total*	*AMAT*
1KB	5.034.083	847.636	2.376.761	1.454.695	9.713.175	520.6397c
2KB	5.402.891	478.828	2.504.658	1.326.798	9.713.175	509.2487c
4KB	5.663.900	217.819	2.689.184	1.142.272	9.713.175	501.1872c
8KB	5.732.139	149.580	2.831.404	1.000.052	9.713.175	499.0796c
16KB	5.752.881	128.838	2.842.558	988.898	9.713.175	498.4390c
32KB	5.756.126	125.593	2.842.808	988.648	9.713.175	498.3387c
64KB	5.761.169	120.550	2.842.811	988.645	9.713.175	498.1830c
128KB	5.768.770	112.949	2.842.811	988.645	9.713.175	497.9482c
256KB	5.776.850	104.869	2.842.811	988.645	9.713.175	497.6987c
512KB	5.781.622	100.097	2.842.811	988.645	9.713.175	497.5513c
1MB	5.783.211	98.508	2.842.811	988.645	9.713.175	497.5022c
2MB	5.784.320	97.399	2.842.811	988.645	9.713.175	497.4679c
4MB	5.785.601	96.118	2.842.811	988.645	9.713.175	497.4284c
8MB	5.786.183	95.536	2.842.811	988.645	9.713.175	497.4104c
16MB	5.786.230	95.489	2.842.811	988.645	9.713.175	497.4090c
32MB	5.786.230	95.489	2.842.811	988.645	9.713.175	497.4090c
64MB	5.786.230	95.489	2.842.811	988.645	9.713.175	497.4090c
128MB	5.786.230	95.489	2.842.811	988.645	9.713.175	497.4090c
256MB	5.786.230	95.489	2.842.811	988.645	9.713.175	497.4090c
512MB	5.786.230	95.489	2.842.811	988.645	9.713.175	497.4090c

Πίνακας 5.7: Original Trace – Write Through policy – Read/Write Hits & Misses ανά cache size.

Από τον Πίνακα 5.7 παρατηρείται ότι καθώς τα write και read misses πέφτουν και τα read και write hits ανεβαίνουν, η τιμή του AMAT μειώνεται. Αυτό συμβαίνει λόγο του ότι τα writes κοστίζουν περισσότερο από τα reads για Write Though πολιτική με την DRAM ως cache και την NVM ως κύρια μνήμη. Για write στην NVM το AMAT είναι

1100c και για read 400c αφού πρώτα γίνεται lookup στην DRAM. Τα write misses σταθεροποιούνται πιο γρήγορα από τα read misses και έχοντας μεγαλύτερη βαρύτητα φαίνεται το AMAT να επηρεάζεται περισσότερο από την μεταβολή των write misses. Όταν και τα δύο σταθεροποιηθούν η τιμή του AMAT εμφανίζεται σταθερή στα 497.41 cycles.

5.3.2 Original WB AMAT

Παρατηρώντας το AMAT για το Original trace με Write Back policy, είναι εμφανή η διαφορά στην απόδοση σε σχέση με την πολιτική Write Through για το αντίστοιχο πείραμα. Αυτό δικαιολογείται από την μεθοδολογία της Write Back η οποία γράφει στην NVM μόνο κατά dirty eviction. Έτσι αποφεύγει το συνεχές και μεγάλο κόστος για εγγραφή στην NVM. Αυτό είναι επωφελές για ίδια objects στα οποία γίνονται writes πολύ συχνά όπως είναι τα volatile objects τα οποία όπως προαναφέρθηκε γίνονται accessed πολλές φορές για κάθε query. Τα dirty evictions καθώς μεγαλώνει η cache μειώνονται και έτσι μειώνονται και οι εγγραφές στην NVM επηρεάζοντας θετικά την τιμή του AMAT. Τέλος παρατηρείται ότι το AMAT στο σενάριο αυτό με μεγάλο cache size τείνει να φτάσει τιμές baseline και συγκεκριμένα σταθεροποιείται στην τιμή 102.96 cycles per access.

Cache	*Read Hits*	*Read Misses*	*Write Hits*	*Write Misses*	*Total*	*AMAT*
1KB	5.005.067	876.652	3.204.233	627.223	9.713.175	191.6506c
2KB	5.286.496	595.223	3.303.855	527.601	9.713.175	172.7021c
4KB	5.584.359	297.360	3.387.068	444.388	9.713.175	154.9353c
8KB	5.707.769	173.950	3.679.626	151.830	9.713.175	121.0039c
16KB	5.750.341	131.378	3.829.974	1.482	9.713.175	104.2103c
32KB	5.755.488	126.231	3.831.277	179	9.713.175	103.9172c
64KB	5.760.627	121.092	3.831.320	136	9.713.175	103.7540c
128KB	5.768.390	113.329	3.831.345	111	9.713.175	103.5117c
256KB	5.776.746	104.973	3.831.355	101	9.713.175	103.2526c
512KB	5.781.604	100.115	3.831.356	100	9.713.175	103.1024c
1MB	5.783.243	98.476	3.831.357	99	9.713.175	103.0517c
2MB	5.784.355	97.364	3.831.357	99	9.713.175	103.0174c
4MB	5.785.637	96.082	3.831.357	99	9.713.175	102.9778c
8MB	5.786.220	95.499	3.831.357	99	9.713.175	102.9598c
16MB	5.786.267	95.452	3.831.357	99	9.713.175	102.9583c
32MB	5.786.267	95.452	3.831.357	99	9.713.175	102.9583c

64MB	5.786.267	95.452	3.831.357	99	9.713.175	102.9583c
128MB	5.786.267	95.452	3.831.357	99	9.713.175	102.9583c
256MB	5.786.267	95.452	3.831.357	99	9.713.175	102.9583c
512MB	5.786.267	95.452	3.831.357	99	9.713.175	102.9583c

Πίνακας 5.8: Original trace – Write Back – Read/Write Hits & Misses ανά cache size

Από τον Πίνακα 5.8 παρατηρείται ότι τα write misses μειώνονται δραματικά με την αύξηση του cache size, από 627.223 σταθεροποιούνται στα 99 από 1MB και πάνω. Επίσης τα Read Misses μειώνονται σταδιακά με την αύξηση του cache size και σταθεροποιούνται πιο αργά από τα write misses στην τιμή 95.452 cycles per access από 16MB και πάνω. Αυτά τα δύο σε συνδυασμό με το ότι τα Write misses έχουν περισσότερη βαρύτητα για AMAT λόγο χρήσης NVM, έχουν ως αποτέλεσμα την σταδιακή μείωση του AMAT μέχρι να σταθεροποιηθεί στην σχεδόν baseline τιμή των 102.96 cycles per access.

5.3.3 Volatile WB AMAT

Το AMAT για volatile objects σε Write Back policy φαίνεται να έχει πολύ παρόμοια συμπεριφορά με το AMAT του Original WB. Και στα δύο σενάρια φαίνεται να μειώνεται σταδιακά και να σταθεροποιείται στην συνέχεια με την μεταβολή του cache size. Επίσης παρατηρείται ότι στο εύρος όπου σταθεροποιείται πλησιάζει την τιμή του baseline, περίπου 100.03 cycles per access. Αυτό σημαίνει ότι η μέθοδος να αφαιρεθούν τα persistent objects και να ακολουθούν μόνο τα volatile objects Write Back πολιτική δεν επηρεάζει αρνητικά τον υπολογισμό του AMAT το οποίο ήταν αναμενόμενο αφού σύμφωνα με τον παρακάτω πίνακα τα Read Misses μειώνονται δραματικά.

Cache	*Read Hits*	*Read Misses*	*Write Hits*	*Write Misses*	*Total*	*AMAT*
1KB	4.209.825	692.285	3.132.315	623.761	8.658.186	196.0301c
2KB	4.493.259	408.851	3.250.338	505.738	8.658.186	172.5779c
4KB	4.749.610	152.500	3.314.735	441.341	8.658.186	156.2579c
8KB	4.851.640	50.470	3.723.359	32.717	8.658.186	105.5275c
16KB	4.877.152	24.958	3.755.911	165	8.658.186	100.8838c
32KB	4.889.579	12.531	3.755.963	113	8.658.186	100.4472c
64KB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
128KB	4.901.435	675	3.755.977	99	8.658.186	100.0348c

256KB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
512KB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
1MB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
2MB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
4MB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
8MB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
16MB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
32MB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
64MB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
128MB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
256MB	4.901.435	675	3.755.977	99	8.658.186	100.0348c
512MB	4.901.435	675	3.755.977	99	8.658.186	100.0348c

Πίνακας 5.9: Volatile trace – Write Back - Read/Write Hits & Misses ανά cache size

Από τον Πίνακα 5.9 σε σύγκριση με τον Πίνακα 5.8 είναι ευδιάκριτη η μεγάλη διαφορά στις τιμές των Read Misses. Αυτή η διαφορά οφείλεται στην αφαίρεση των read misses από persistent objects για τα οποία όπως αναφέρθηκε στο υποκεφάλαιο 5.1, λόγο access pattern του Memcached σε κάθε GET πρώτα γίνεται hash table lookup και μετά write. Έτσι κάθε access από persistent object που γίνεται πρώτη φορά μαρκάρεται ως read miss και στην συνέχεια γίνεται write hit. Αυτό αποδεικνύεται και από την σύγκριση οποιουδήποτε πειράματος του Volatile WB αθροίζοντας τα read misses του με τα read misses από το αντίστοιχο πείραμα για Persistent WT, τα οποία έχουν ως αποτέλεσμα τα read misses για Original WB στο αντίστοιχο πείραμα.

Επίσης παρατηρείται ότι τα Read misses και Write misses μειώνονται όσο το cache size αυξάνεται μέχρι να σταθεροποιηθούν στα 675 και 99 αντίστοιχα. Αυτό συμβαίνει γιατί όσο μεγαλώνει το cache size τα συχνά προσβάσημα δεδομένα, τα οποία για volatile objects γίνονται accessed πολλαπλές φορές ανά query, αρχίζουν να χωράνε στην cache. Γι' αυτό τον λόγο προκαλούνται λιγότερα dirty evictions και το AMAT μειώνεται αισθητά, τόσο που φτάνει κοντά στο baseline. Ο λόγος που τα υπολειπόμενα read misses και write misses παραμένουν όσο και αν μεγαλώσει το cache size είναι τα cold misses, αφού η πρώτη φορά που θα προσπελαστεί ένα block δεν μπορεί να είναι στην cache λόγο ανάγκης read-only trace για τη μελέτη του worst-case scenario.

Τέλος όπως αναφέρεται στην εισαγωγή, μετά από ανάλυση των διευθύνσεων των writes και του source code, τα volatile objects παράγουν περίπου 143 writes/query. Αυτά είναι

τα αναπόφευκτα writes που γίνονται εσωτερικά του Memcached για κάθε GET request και αποτελούνται από response handling, connection state, thread synchronization κ.α. Δεν μελετώνται client writes.

5.3.4 Persistent WT AMAT

Cache	*Read Hits*	*Read Misses*	*Write Hits*	*Write Misses*	*Total*	*AMAT*
1KB	878.560	101.049	75.380	0	1.054.989	200.1856c
2KB	880.534	99.075	75.380	0	1.054.989	199.6243c
4KB	880.577	99.032	75.380	0	1.054.989	199.6120c
8KB	880.582	99.027	75.380	0	1.054.989	199.6106c
16KB	880.593	99.016	75.380	0	1.054.989	199.6075c
32KB	880.626	98.983	75.380	0	1.054.989	199.5981c
64KB	880.684	98.925	75.380	0	1.054.989	199.5816c
128KB	880.789	98.820	75.380	0	1.054.989	199.5517c
256KB	880.979	98.630	75.380	0	1.054.989	199.4977c
512KB	881.327	98.282	75.380	0	1.054.989	199.3988c
1MB	881.930	97.679	75.380	0	1.054.989	199.2273c
2MB	882.964	96.645	75.380	0	1.054.989	198.9333c
4MB	884.228	95.381	75.380	0	1.054.989	198.5738c
8MB	884.789	94.820	75.380	0	1.054.989	198.4143c
16MB	884.832	94.777	75.380	0	1.054.989	198.4021c
32MB	884.832	94.777	75.380	0	1.054.989	198.4021c
64MB	884.832	94.777	75.380	0	1.054.989	198.4021c
128MB	884.832	94.777	75.380	0	1.054.989	198.4021c
256MB	884.832	94.777	75.380	0	1.054.989	198.4021c
512MB	884.832	94.777	75.380	0	1.054.989	198.4021c

Πίνακας 5.10: Persistent trace – Write Through - Read/Write Hits & Misses ανά cache size

Τα Persistent objects ακολουθούν Write Through policy (no write allocate) και έτσι κάθε write προωθείται άμεσα στην NVM χωρίς να γίνεται fetch το block σε write miss. Από τον Πίνακα 5.10 παρατηρείται ότι το AMAT των persistent objects παραμένει πολύ υψηλό για όλο το εύρος μεγέθους της cache. Συγκεκριμένα παίρνει τιμές από 200.19 cycles για 1KB μέχρι 198.40 cycles για 16MB και πάνω όπου παραμένει σταθερό σε αυτή τη τιμή. Δεν δείχνει κάποια βελτίωση και δεν πλησιάζει ποτέ το baseline των 100 cycles.

Η αιτία αυτής της συμπεριφοράς είναι δομικής φύσεως αφού τα persistent objects εκτελούν περίπου 3 εγγραφές για κάθε query. Αυτές οι εγγραφές όπως αναλύθηκε στην αρχή του κεφαλαίου είναι αποκλειστικά από refcount και LRU position update αφού το trace που μελετάτε είναι read-only. Κάθε write hit που προκαλείται έχει κόστος 1100c και λόγω της write through πολιτικής η οποία γράφει πάντα στην NVM αυτό το κόστος είναι αναπόφευκτο ανεξαρτήτως cache size.

Το αμέσως επόμενο βαρύτερο κόστος που δικαιολογεί την σταθερότητα του AMAT σε persistent objects με Write Through, είναι τα σταθερά υψηλά read misses. Αυτό συμβαίνει για τον ίδιο λόγο που αναλύθηκε στο υποκεφάλαιο 5.1, όπου στα persistent objects με uniform distribution και cold start για την cache, κάθε query αγγίζει σχεδόν πάντα key που δεν έχει προσπελαστεί ξανά, με αποτέλεσμα κάθε τέτοια πρόσβαση να είναι cold miss. Ακόμα και σε περίπτωση όπου όλα τα persistent objects βρεθούν στην cache μετά από ένα αριθμό queries, με την προϋπόθεση ότι η cache μπορεί να φιλοξενήσει το μέγεθος των δεδομένων, αυτός ο αριθμός Read Misses θα παραμείνει λόγω cold cache. Στην περίπτωση της παρούσας μελέτης όμως το σενάριο αυτό δεν έχει εξεταστεί γιατί η έρευνα βασίστηκε σε 25410 queries τα οποία δεν μπορούν να φέρουν στην cache όλα τα unique keys.

Τέλος παρατηρείται ότι τα write misses έχουν σταθερή τιμή 0 για όλο το εύρος τιμών του cache size. Αυτό οφείλεται αποκλειστικά στο ότι κάθε GET request εκτελεί πρώτα read για κάθε item (do_item_get()), το οποίο φορτώνει το block στην cache και έτσι τα writes που ακολουθούν (refcount, LRU) βρίσκουν το block ήδη στην cache με write hit.

5.3.5 Selective AMAT

Το AMAT για Selective policy υπολογίζεται ως weighted average των AMAT των persistent WT και volatile WB με βάρος τον αριθμό των accesses κάθε κατηγορίας. Αποτελεί lower bound γιατί δεν λαμβάνεται υπόψη η αλληλεπίδραση των objects στην κοινή DRAM cache.

Selective AMAT= (Vol_AMAT×Vol_Total+Pers_AMAT×Pers_Total) /
(Vol_Total+Pers_Total)

Από τον πίνακα 5.6 παρατηρείται ότι το Selective AMAT ξεκινά από 196.48c για cache size ίσο με 1KB και μειώνεται σταδιακά μέχρι να σταθεροποιείται στα 110.72c από τα 16MB cache size και πάνω. Αυτό σημαίνει ότι είναι μόλις 10.72% πάνω από το baseline.

Η συμπεριφορά αυτή εξηγείται πλήρως από την σύνθεση των persistent WT και volatile WB. Τα volatile objects αποτελούν σχεδόν το 90% των προσβάσεων (8.658.186 από τις 9.713.175) και όπως ήδη αναλύθηκε το AMAT τους για μεγάλες cache μνήμες σταθεροποιείται στα 100.3c που πρακτικά είναι ταυτόσημο με το baseline. Τα persistent objects από την άλλη, αποτελούν το 11% των accesses (1.054.989) και έχουν AMAT περίπου 198c λόγω του αναπόφευκτου κόστους που προκαλεί η Write Through policy κάνοντας εγγραφές στην μνήμη σε κάθε write, το οποίο όμως εγγυάται συνέπεια δεδομένων. Το weighted average όμως κυριαρχείται από το 90% των volatile objects και αυτό έχει ως αποτέλεσμα η Selective Policy να πλησιάζει το baseline αντιγράφοντας σχεδόν πλήρως την συμπεριφορά των Volatile WB μετρήσεων παρά το υψηλό AMAT των persistent objects.

Συγκρίνοντας τις μετρήσεις από Selective Policy με τις μετρήσεις για Original WT και Original WB για cache sizes 16MB και πάνω, εκεί δηλαδή που και οι τρείς σταθεροποιούνται, η Selective policy με 110.72c είναι σημαντικά πιο αποδοτική από Write Through με 497.41c, με την διαφορά τους να φαίνεται σχεδόν 5 φορές καλύτερη η απόδοση της Selective policy. Σε σύγκριση τώρα με την Write Back policy η διαφορά τους είναι μόλις 7.76c η οποία αντιπροσωπεύει το αναπόφευκτο κόστος της εγγύησης για συνέπεια των δεδομένων.

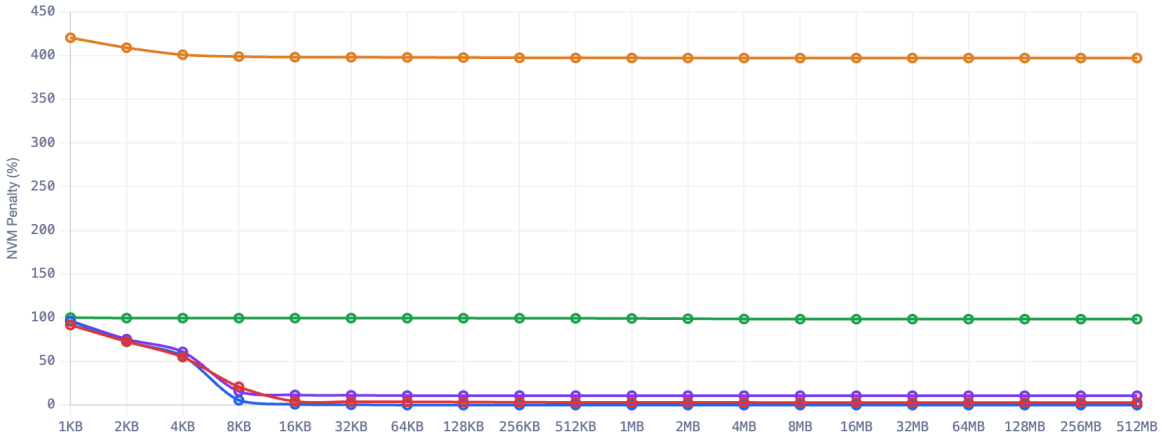
5.4 NVM Penalty

Το NVM Penalty εκφράζει το επιπλέον κόστος που εισάγει η ύπαρξη της NVM ως δεύτερο επίπεδο μνήμης (κύρια μνήμη), σε σχέση με το baseline το οποίο αποτελείται αποκλειστικά από DRAM όπου κάθε πρόσβαση είτε write είτε read έχει κόστος 100 cycles. Ο τύπος που χρησιμοποιήθηκε για τον υπολογισμό του είναι ο εξής:

$$\text{NVM Penalty} = [(\text{AMAT} - 100)/100] \times 100\% = (\text{AMAT} - 100)\%$$

Cache Size	*Original WB*	*Original WT*	*Volatile WB*	*Persistent WT*	*Selective*
1KB	91.65%	420.64%	96.03%	100.19%	96.48%
2KB	72.70%	409.25%	72.58%	99.62%	75.52%
4KB	54.94%	401.19%	56.26%	99.61%	60.97%
8KB	21.00%	399.08%	5.53%	99.61%	15.75%
16KB	4.21%	398.44%	0.88%	99.61%	11.61%
32KB	3.92%	398.34%	0.45%	99.60%	11.22%
64KB	3.75%	398.18%	0.03%	99.58%	10.85%
128KB	3.51%	397.95%	0.03%	99.55%	10.84%
256KB	3.25%	397.70%	0.03%	99.50%	10.84%
512KB	3.10%	397.55%	0.03%	99.40%	10.83%
1MB	3.05%	397.50%	0.03%	99.23%	10.81%
2MB	3.02%	397.47%	0.03%	98.93%	10.78%
4MB	2.98%	397.43%	0.03%	98.57%	10.74%
8MB	2.96%	397.41%	0.03%	98.41%	10.72%
16MB	2.96%	397.41%	0.03%	98.40%	10.72%
32MB	2.96%	397.41%	0.03%	98.40%	10.72%
64MB	2.96%	397.41%	0.03%	98.40%	10.72%
128MB	2.96%	397.41%	0.03%	98.40%	10.72%
256MB	2.96%	397.41%	0.03%	98.40%	10.72%
512MB	2.96%	397.41%	0.03%	98.40%	10.72%

Πίνακας 5.11: NVM Penalty ανά cache size για κάθε σενάριο.



Σχήμα 5.4: Γραφική αναπαράσταση NVM Penalty.

Όπως είναι αναμενόμενο το Original WT παρουσιάζει το υψηλότερο NVM Penalty για όλες τις τιμές του cache size. Αυτό σημαίνει ότι το σύστημα με Write Through policy είναι σχεδόν 5 φορές πιο αργό από ένα αποκλειστικά DRAM σύστημα ανεξαρτήτως cache size. Το penalty αυτό είναι ουσιαστικά αναπόφευκτο με Write Through, αφού

κάθε write προωθείται αμέσως στην NVM της οποίας κάθε εγγραφή κοστίζει 1100c και ο αριθμός writes είναι σταθερός ανεξαρτήτως cache size.

Το Original WB φαίνεται να έχει το μικρότερο penalty σε σύγκριση με Original WT και Selective policy. Φαίνεται να ξεκινά από 91.65% στο 1KB και μειώνεται δραματικά μέχρι να σταθεροποιηθεί στο 2.96% από 8MB cache size και πάνω. Το penalty αυτό είναι σχεδόν μηδενικό για μεγάλες cache, καθώς με WB policy τα dirty evictions ελαχιστοποιούνται όταν τα frequently accessed blocks χωράνε στην cache. Ωστόσο, η Write Back Policy δεν εγγυάται συνέπεια δεδομένων και αυτό έχει ως αποτέλεσμα δεδομένα όπως τα key-value items και ο hash table να ενδέχεται να χαθούν σε περίπτωση αποτυχίας του συστήματος αν δεν προλάβει να γίνει eviction.

Στα Persistent WT πειράματα οι τιμές για NVM penalty παραμένουν σταθερά υψηλές στο γύρω από το 100% για όλα τα cache sizes, δύο φορές δηλαδή πιο αργό από το baseline. Στα πειράματα για Volatile WB ωστόσο, η τιμή του penalty για μεγάλα cache sizes τείνει προς το 0%, καμία δηλαδή επιβάρυνση.

Η Selective policy, η οποία συμπεραίνεται από τον συνδυασμό των Persistent WT και Volatile WB σταθεροποιείται στο 10.72% NVM penalty. Σε σύγκριση με Write Through και Write Back policies μπορεί να μην έχει όσο καλή απόδοση έχει η Write Back αλλά η διαφορά μεταξύ τους είναι μόλις 7.76 ποσοστιαίες μονάδες. Η κρίσιμη όμως διαφορά της Selective policy έναντι της Write Back είναι ότι μπορεί να εγγυηθεί data consistency ενώ η Write Back όχι.

Κεφάλαιο 6

Συμπεράσματα

Η παρούσα εργασία εξέτασε το κεντρικό ερώτημα, αν μπορεί μια Selective Policy η οποία εφαρμόζει Write-Through στα objects που πρέπει να είναι συνεπής στην μνήμη χωρίς απώλειες σε τυχών διακοπή του συστήματος και Write-Back στα objects που δεν χρειάζονται συνέπεια αφού αναδημιουργούνται σε κάθε επανεκκίνηση του συστήματος. Κύριος στόχος της εργασίας είναι να εξεταστεί κατά πόσο μπορεί να γίνει διατηρώντας την υψηλή απόδοση και μειώνοντας το write traffic προς την NVM.

Για να απαντηθεί το ερώτημα αυτό, υλοποιήθηκε ένα πλήρες πειραματικό pipeline. Έγινε χρήση custom Intel PIN tool για καταγραφή memory accesses και δεσμεύσεων/αποδεσμεύσεων μνήμης στο Memcached 1.6.14. Στην συνέχεια έγινε κατηγοριοποίηση των memory objects σε persistent και volatile μέσω ανάλυσης του source code και προσομοίωση cache συμπεριφοράς με τον DineroIV simulator για cache sizes από 1KB μέχρι 512MB.

Τα πειράματα απάντησαν το ερώτημα αυτό καταφατικά. Η Selective policy επιτυγχάνει 99.6% μείωση του write traffic έναντι της Write-Through πολιτικής για cache sizes από 64KB και άνω (0.190 MB vs 42.229 MB). Ακόμα και για μικρές cache τα αποτελέσματα δεν έδειξαν αντίθετο αποτέλεσμα αλλά παρουσίασαν μείωση ύψους 89.7%. Το σύστημα με Selective policy παρουσίασε μείωση απόδοσης μόλις 10.72% σε σχέση με ένα αποκλειστικά DRAM σύστημα με NVM penalty 10.72% έναντι 397.41% καθυστέρησης που απέδωσε η Write-Through πολιτική. Σε επίπεδο AMAT η Selective policy απέδειξε ότι είναι περίπου 4.5 φορές καλύτερη από την Write-Through με 110.7 cycles και 497.4 cycles αντίστοιχα.

Έναντι της Write-Back, η Selective παρουσιάζει ελαφρά χειρότερες επιδόσεις. Η διαφορά τους στο NVM penalty, δηλαδή η απόκλιση της απόδοσης τους από το baseline είναι μόλις 7.76 ποσοστιαίες μονάδες (10.72% vs 2.96%) και στο AMAT μόλις 7.76 cycles (110.7c vs 102.96c). Το Write Traffic της Selective φαίνεται περίπου

87.8% χαμηλότερο από το Write-Back για μεγάλες caches (0.190 MB vs 1.554 MB), αφού η Write-Through policy που χρησιμοποιείται για τα persistent objects γράφει μόνο τα τροποποιημένα bytes αντί για ολόκληρο το dirty block. Το κρίσιμο όμως πλεονέκτημα της Selective έναντι της Write Back policy είναι ότι εγγυάται πλήρη συνέπεια των δεδομένων για τα key-value items και τον hash table στην NVM, κάτι που η Write Back δεν μπορεί να εγγυηθεί.

Τα ευρήματα αυτά επιβεβαιώνονται από την ανάλυση των memory accesses. Τα persistent objects εκτελούν μόλις ~3 writes/query (refcount++, refcount--, LRU update) ενώ τα volatile objects εκτελούν ~143.9 writes/query από εσωτερικές δομές του Memcached (response handling, connection state, thread synchronization). Αυτή η θεμελιώδης ασυμμετρία, λίγα bytes που χρειάζονται consistency και πολλά bytes που δεν τη χρειάζονται, είναι αυτό που κάνει την Selective policy αποτελεσματική.

Επιπλέον η μελέτη αποκάλυψε ότι ακόμα και σε ένα read-only workload, το Memcached παράγει σημαντικό εσωτερικό write traffic (~39.1% των memory accesses είναι writes) που δεν μπορεί να αποφευχθεί. Η Selective policy διαχειρίζεται αυτό το αναπόφευκτο write traffic με τον βέλτιστο τρόπο, εφαρμόζοντας Write-Through μόνο εκεί που είναι απαραίτητο για consistency.

Κεφάλαιο 7

Μελλοντικές Εργασίες

Στην παρούσα εργασία εξετάστηκε η Selective policy κάτω από πολύ συγκεκριμένες πειραματικές συνθήκες όπως read-only workload, uniform distribution και single threaded simulation. Τα αποτελέσματα είναι ενθαρρυντικά ωστόσο υπάρχουν μερικές κατευθύνσεις που μπορούν να διερευνηθούν στο μέλλον.

Read & Write Workload

Η παρούσα μελέτη χρησιμοποίησε αποκλειστικά read-only workload (-u 0.00) για να απομονωθεί το αναπόφευκτο write traffic που παράγει το Memcached κατά GET requests. Θα ήταν ενδιαφέρον να μελετηθεί η επέκταση της παρούσας εργασίας με σκοπό την μελέτη της Selective policy καθώς στο measurement phase το Memcached δέχεται και SET requests εκτός από GETs. Με την εισαγωγή SET requests κάθε εγγραφή νέου item στο slab θα αποτελεί persistent write μεγέθους ~304B, το οποίο είναι εξαιρετικά μεγαλύτερο από τα μέχρι στιγμής writes που μελετήθηκαν με μέγεθος ~2B των metadata. Αυτό θα έχει ως αποτέλεσμα να αυξηθεί σημαντικά το write traffic των persistent objects και κατ' επέκταση το NVM penalty της Selective policy. Σκοπός είναι η εύρεση του breakeven point για τα SET requests όπου η Selective policy παύει να υπερτερεί των άλλων policies.

Υλοποίηση Native Selective Cache Simulator

Ένας σημαντικός περιορισμός της παρούσας μεθοδολογίας είναι ότι η Selective policy προσομοιώνεται μέσω δύο χωριστών εκτελέσεων του DineroIV, μία για τα persistent objects με WT και μία για τα volatile objects με WB, με αποτέλεσμα τα δύο traces να μην ανταγωνίζονται για χώρο στην κοινή DRAM cache. Αυτό οδηγεί στην παραδοχή του lower bound που αναλύθηκε στο Κεφάλαιο 4. Μια σημαντική μελλοντική κατεύθυνση θα ήταν η ανάπτυξη ενός custom cache simulator ο οποίος να υποστηρίζει natively τη Selective policy. Δηλαδή να εφαρμόζει αυτόματα WT για προσβάσεις σε

persistent objects και WB για προσβάσεις σε volatile objects, χρησιμοποιώντας το ενιαίο trace ως είσοδο. Με αυτόν τον τρόπο τα δύο είδη objects θα ανταγωνίζονται για τον ίδιο cache χώρο, αποτυπώνοντας ρεαλιστικά τις αλληλεπιδράσεις μεταξύ τους και εξαλείφοντας την ανάγκη υπολογισμού lower/upper bound. Το αποτέλεσμα θα ήταν η άμεση μέτρηση του πραγματικού write traffic και AMAT της Selective policy χωρίς παραδοχές.

Multi-threaded Cache Simulation

Ο DineroIV cache simulator που χρησιμοποιήθηκε στην παρούσα εργασία είναι single-threaded, με αποτέλεσμα το trace να περιορίζεται σε έναν worker thread (-t 1). Σε πραγματικά παραγωγικά συστήματα το Memcached εκτελείται με πολλαπλούς worker threads που μοιράζονται την ίδια cache. Η χρήση ενός multi-threaded cache simulator θα επέτρεπε την προσομοίωση με πολλαπλούς workers, αποτυπώνοντας πιο ρεαλιστικά τις επιπτώσεις του cache contention μεταξύ threads στη συμπεριφορά της Selective policy.

Βιβλιογραφία

[1] Arun KP and D. Mishra, "Kindle: A Comprehensive Framework for Exploring OS-Architecture Interplay in Hybrid Memory Systems," in Proc. IEEE International Symposium on Workload Characterization (IISWC), 2024.

[2] M. Bhargava, CacheSimulator [Online]. Available: https://github.com/muditbhargava66/CacheSimulator

[3] ChampSim [Online]. Available: https://github.com/ChampSim/ChampSim

[4] S. Cho, "CS/COE1541: Introduction to Computer Architecture — Memory Hierarchy," University of Pittsburgh, 2013. [Online]. Available: https://people.cs.pitt.edu/~cho/cs1541/current/handouts/lect-memh_4up.pdf

[5] R. Cohn, "Dynamic Instrumentation with Pin," Intel. [Online]. Available: https://cscads.rice.edu/Cohn-Pin.pdf

[6] U. Drepper, "How To Write Shared Libraries," Red Hat, 2011. [Online]. Available: https://www.akkadia.org/drepper/dsohowto.pdf

[7] J. Edler and M. D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator," University of Wisconsin. [Online]. Available: https://pages.cs.wisc.edu/~markhill/DineroIV/

[8] J. Edler and M. D. Hill, "DineroIV Source Notes." [Online]. Available: https://github.com/danluu/dineroIV/blob/master/NOTES

[9] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2013, pp. 371–384. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final197.pdf

[10] B. Fitzpatrick, "Distributed Caching with Memcached," Linux Journal, 2004.

[11] GeeksforGeeks, "Write Through and Write Back in Cache." [Online]. Available: https://www.geeksforgeeks.org/computer-organization-architecture/write-through-and-write-back-in-cache/

[12] GNU Binutils, "addr2line." [Online]. Available: https://sourceware.org/binutils/docs/binutils/addr2line.html

[13] GNU C Library Manual, "Replacing malloc." [Online]. Available: https://sourceware.org/glibc/manual/latest/html_node/Replacing-malloc.html

[14] GNU C Library Manual, "Virtual Memory Allocation and Paging." [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_3.html

[15] GNU/bminor, "include/libc-symbols.h," glibc. [Online]. Available: https://github.com/bminor/glibc/blob/master/include/libc-symbols.h

[16] T. Hirofuchi and R. Takano, "A Prompt Report on the Performance of Intel Optane DC Persistent Memory Module,"

arXiv:2002.06018, 2020.

[17] Intel, "Pin User Guide." [Online]. Available: https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/index.html

[18] J. Izraelevitz et al., "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," arXiv:1903.05714, 2019. [Online]. Available: https://arxiv.org/pdf/1903.05714

[19] J. Leverich and C. Kozyrakis, "Reconciling High Server Utilization and Sub-millisecond Quality-of-Service," in Proc. 9th European Conference on Computer Systems (EuroSys), 2014. [Online].Available: http://csl.stanford.edu/~christos/publications/2014.mutilate.eurosys.pdf

[20] D. Mikheev, "Memory Allocation and Deallocation in glibc." [Online]. Available: https://dimamik.com/posts/general/memory_allocation_and_deallocation_in_glibc/

[21] C.-K. Luk et al., "PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation," in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2005.

[22] D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 5th ed. Morgan Kaufmann, 2014.

[23] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in Proc. 36th International Symposium on Computer Architecture (ISCA), 2009.

[24] Red Hat Developer, "Securing malloc in glibc: Why malloc hooks had to go," 2021. [Online]. Available: https://developers.redhat.com/articles/2021/08/25/securing-malloc-glibc-why-malloc-hooks-had-go

[25] A. Shehabi et al., "2024 United States Data Center Energy Usage Report," Lawrence Berkeley National Laboratory, U.S. Department of Energy, 2024. [Online]. Available: https://escholarship.org/uc/item/32d6m0d1

[26] voidsecurity.in, "Data Structure Recovery using PIN," 2015. [Online]. Available: https://www.voidsecurity.in/2015/04/data-structure-recovery-using-pin-and.html

[27] G. Wang, X. Che, H. Wei, C. Pei, and J. Hu, "Crash Consistency in DRAM-NVM-Disk Hybrid Storage System," arXiv:2408.04238, 2024. [Online]. Available: https://arxiv.org/abs/2408.04238

[28] H. Weatherspoon, "Caches (Writing)," Cornell University CS 3410, 2013. [Online]. Available: https://www.cs.cornell.edu/courses/cs3410/2013sp/lecture/18-caches3-w.pdf

[29] J. Yang et al., "An Empirical Analysis on Memcached's Replacement Policies," in Proc. International Symposium on Memory Systems (MEMSYS), 2023. [Online]. Available: https://www.memsys.io/wp-content/uploads/2023/09/2.pdf

[30] D. Zhang, L. Ju, M. Zhao, X. Gao, and Z. Jia, "Write-back aware shared last-level cache management for hybrid main memory," in Proc. Design Automation Conference (DAC), 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7544413

Δηλώνω ότι η παρούσα διατριβή και οποιοδήποτε συνοδευτικό λογισμικό αποτελούν δικό μου πρωτότυπο έργο, εκπονημένο σε πλήρη συμμόρφωση με τον Κώδικα Δεοντολογίας για την Έρευνα του Πανεπιστημίου Κύπρου (https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/**), και με πλήρη ευθύνη εκ μέρους μου για την ακρίβεια, την ακεραιότητα και την εγκυρότητα όλου του περιεχομένου.**