Ατομική Διπλωματική Εργασία

ΜΟΝΤΕΛΟΠΟΙΗΣΗ ΑΠΟΔΟΣΗΣ ΜΗ-ΠΤΗΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ ΚΥΡΙΑΣ ΜΝΗΜΗΣ

Σωτήρης Κασινίδης

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μοντελοποίηση απόδοσης μη-πτητικών συστημάτων κύριας μνήμης

Σωτήρης Κασινίδης

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 2022

Ευχαριστίες

Με την ολοκλήρωση της διπλωματικής μου εργασίας δεν θα μπορούσα να παραλείψω τις θερμές ευχαριστίες στα άτομα τα οποία συνέλαβαν στην επιτυχής ολοκλήρωσης αυτού του έργου. Χωρίς την πολύτιμη βοήθεια και συμβουλές που είχα από αυτά τα άτομα δεν θα ήταν δυνατή η ολοκλήρωση της δουλείας μου.

Αρχικά, πρέπει να τονίσω ότι τεράστια σημασία για την επιτυχία μου έπαιξε ο επιβλέπων καθηγητής μου κ. Βώλος επίκουρος καθηγητής στο τμήμα, που με βοηθούσε καθ΄ όλη την διάρκεια της δουλείας μου. Για δύο ακαδημαϊκά εξάμηνα είχαμε εβδομαδιαίες συναντήσεις στις οποίες μου έλυνε τυχών απορίες και συζητούσαμε την πρόοδο μου. Αυτός ο τρόπος με βοήθησε κατά πολύ να αποδίδω καλύτερα και να είμαι πάντα σε τριβή με το θέμα μου. Επίσης ο κ. Βώλος πάντα απαντούσε έγκαιρα στα emails τα οποία του έστελνα και πάντα με τροφοδοτούσε με το κατάλληλο υλικό έτσι ώστε να είμαι σε θέση να μελετήσω και να λύσω τις απορίες μου. Χωρίς αυτών δεν θα ήμουν σε θέση να φέρω εις πέραν επιτυχώς την ΑΔΕ μου.

Ακολούθως, θα ήθελα να ευχαριστήσω τους γονείς μου Λουκά και Θεοδώρα που χωρίς αυτούς δεν θα ήμουν σε αυτή την θέση την εν λόγο στιγμή. Πάντα με στήριζαν σε όλους τους τομείς και με βοηθούσαν να ξεπεράσω όλων των ειδών τις δυσκολίες για να είμαι συγκεντρωμένος και σε θέση να μελετήσω και να προχωρώ με την δουλεία που είχα.

Τέλος, θα ήθελα να δώσω θερμές ευχαριστίες στον προϊστάμενο και στους συναδέλφους στην εργασία μου οι οποίοι πάντα με βοηθούσαν και είχαμε μια άψογη συνεργασία με αποτέλεσμα να έχω την κατάλληλο χρόνο και διάθεση για την διπλωματική μου.

Περίληψη

Τα μη-πτητικά συστήματα κύριας μνήμης[2] (NVMMs) είναι συστήματα που έχουν μη πτητική μνήμη δηλαδή η μνήμη συγκρατεί τα δεδομένα της με την διακοπή του ηλεκτρικού ρεύματος, αντιθέτως με την πτητική μνήμη (RAM) η οποία τα χάνει. Σκοπός αυτής της εργασίας είναι μοντελοποίηση της απόδοσης τέτοιων συστημάτων, δηλαδή στόχος μας ήταν να αναπτύξουμε μοντέλα τα οποία με κάποιες εισόδους του προγράμματος όπως αριθμός στοιχείων, μέγεθος στοιχείων, χαρακτηριστικά υλικού, και άλλα θα είναι σε θέση να προβλέψουν κάποιες μετρικές απόδοσης, όπως ο χρόνος εκτέλεσης και η καθυστέρηση μνήμης για συγκεκριμένα προγράμματα και δομές δεομένων.

Ακολουθήσαμε δύο προσεγγίσεις για την υλοποίηση των μοντέλων μας. Η μια προσέγγιση αφορούσε μετρικές τύπου LLC-loads και LLC-stores και τα misses τους, από τα benchmarks του πακέτου PMDK (περιγράφεται πιο κάτω), εστί ώστε να έχουμε των αριθμό εγγραφών και αναγνώσεων στην Optane μας και με αυτό τον τρόπο να μπορούμε να υπολογίσουμε τον χρόνο εκτέλεσης του προγράμματος. Η δεύτερη προσέγγιση ήταν πιο θεωρητική, και αφορούσε την μοντελοποίηση υλοποίησης της δομής δεδομένων για την οποία περνάμε μετρήσεις. Πιο συγκεκριμένα κατασκευάσαμε ένα αναλυτικό μοντέλο που εκτιμά τις προσπελάσεις μνήμης, που θα υπήρχαν σε κάθε στάδιό της δομής μας, με βάσει την υλοποίηση της δομής δεδομένων hashmap στο PMDK.

Συγκεκριμένα μελετήσαμε τη δομή δεδομένων hashmap από το πακέτο ανάπτυξης PMDK. Το PMDK που με απλά λόγια είναι ένα σύνολο από βιβλιοθήκες με τις οποίες σου δίνεται η δυνατότητα να έχεις απευθείας πρόσβαση σε διευθύνσεις μνήμης μιας μη-πτητικής μνήμης (DAX) αγνοώντας την ιεραρχία της μνήμης (caches). Χρησιμοποιώντας έτοιμα benchmarks του πακέτου συλλέξαμε διάφορες μετρικές, αλλά που θα μα βοηθούσαν στην υλοποίηση των μοντέλων.

Με την βοήθεια του low-level profiler LENS[1], σε πολύ αφαιρετικό επίπεδο ο LENS είναι ένα πρόγραμμα με το οποίο μπορείς να μετρήσεις την απόδοση του συστήματος

σου, μετρήσαμε την καθυστέρηση (latency) και το εύρος ζώνης (bandwidth) της μη-πτητικής μνήμης που είχαμε στην διάθεση μας, που είναι εγκαταστημένη στον server shasta του πανεπιστημίου. Έτσι ώστε να πάρουμε τα αποτελέσματα για το υλικό μας και να τα χρησιμοποιήσουμε στα τα μοντέλα μας.

Τέλος κάναμε μια σύγκριση των αποτελεσμάτων των δύο προσεγγίσεων, τονίζοντας τις διαφορές, τις ομοιότητες και τα πλεονεκτήματα/ μειονεκτήματα που έχει το κάθε μοντέλο.

Περιεχόμενα

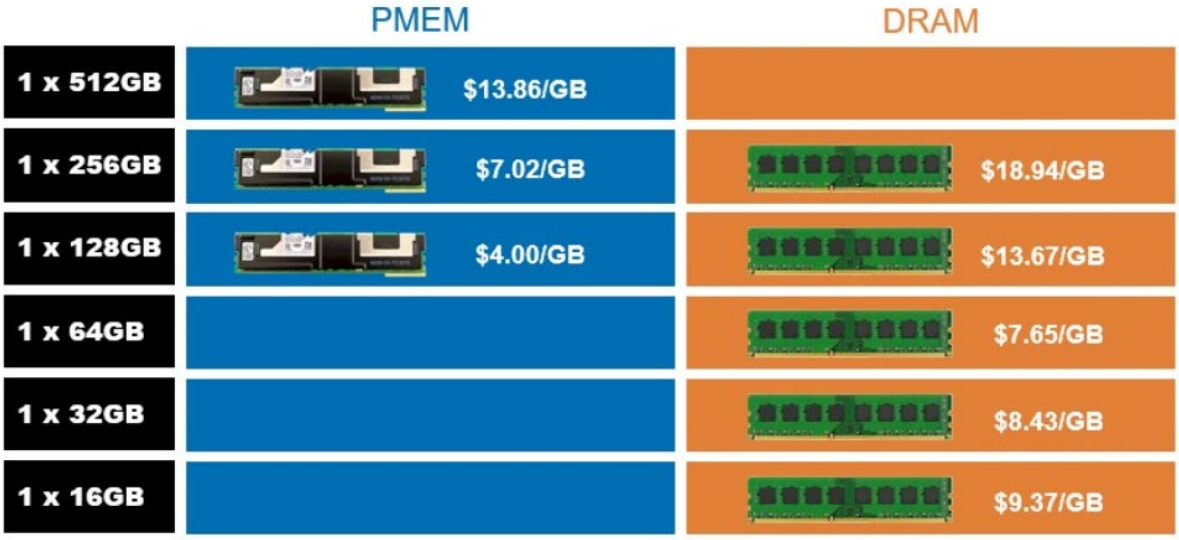
Κεφάλαιο 1

Εισαγωγή

1.1 Παρακίνηση

Πλέον στις μέρες υπάρχει τεράστια υπολογιστική δύναμη στους ηλεκτρονικούς υπολογιστές αλλά αυτό αποκτάται με σχετικά μεγάλο κόστος. Όπως ξέρουμε η υπολογιστική δύναμη σχετίζεται κυρίως στον επεξεργαστή (CPU) και στην διαθέσιμη κύρια μνήμη (RAM) του συστήματος. Εδώ είναι που μια καινούργια τεχνολογία όπως είναι οι μη-πτητικές κύριες μνήμες ήρθαν για να δώσουν μια λύση σε ένα μεγάλο για πολλούς πρόβλημα το κόστος. Η τεχνολογία Intel Optane[2] είναι μια από αυτές, ουσιαστικά αυτή η μνήμη μπορούμε να πούμε ότι τοποθετείται ανάμεσα στην DRAM και τους SSDs/δίσκους στην ιεραρχία μνήμης, δηλαδή ο χρόνος απόκρισης της είναι μεταξύ αυτών των δύο κατηγοριών. Στην αγορά μέχρι στιγμής υπάρχουν DIMMS για Persistent memory μεγέθους 128, 256, και 512 GB σε χωρητικότητα. Πρέπει να τονίσουμε ότι αυτές οι τύπου μνήμες είναι πολύ πιο φθηνές από μνήμες DRAM, συγκεκριμένα το 2021 για να αγοράσει κάποιος 128 GB DRAM θα του κόστιζε περίπου €1650 ενώ για 128 GB PMEM θα χρειαζόταν γύρο στα €480 δηλαδή x3.4 φθηνότερα, στην Εικόνα 1.1 φαίνεται η διαφορά ανάμεσα στις τιμές των δύο κατηγοριών μνήμης. Το σημαντικό εδώ είναι ότι η απόδοση τους είναι αρκετά κοντά όπως θα δούμε και

στην συνέχεια. Κατά την γνώμη μου είναι ένα πολύ ενδιαφέρον θέμα να μπορεί να προβλέψεις αν μια τεχνολογία σαν και αυτή μπορεί να ικανοποιήσει το σύστημα σου, δηλαδή έχεις μια ανεχτή επιβράδυνση στο σύστημα μνήμης αλλά με χαμηλότερο οικονομικό κόστος.



Εικόνα 1.1 Διαφορά τιμης μεταξύ DRAM/PMEM

1.2 Στόχος εργασίας

Στόχος λοιπόν αυτής της εργασίας είναι η ανάπτυξη μοντέλων επίδοσης τα οποία βασίζονται στα θεμέλια της τεχνολογίας που μελετούμε και μας δίνουν μια θεωρητική απάντηση για την απόδοση της. Τα μοντέλα της εργασίας θα είναι σε θέση να προβλέψουν μετρικές απόδοσης όπως ο χρόνος εκτέλεσης κάποιου απλού προγράμματος το οποίο κάνει χρήση μιας δομής δεδομένων στην περίπτωση μας ένα hashmap (η υλοποίηση του θα αναλυθεί πιο κάτω), και με αυτή την δομή γίνονται προσπεράσεις και έγραφες μνήμης πάνω στην τεχνολογία Optane. Με αυτό τον τρόπο παίρνουμε ένας δείγμα το πως θα συμπεριφερθεί η τεχνολογία που μελετούμε, για να αποφασίσουμε αν ικανοποιεί ή όχι τις δικές μας ανάγκες.

1.3 Συνεισφορές

Η κύρια συνεισφορά αυτής της εργασίας είναι ότι κάποιος μπορεί να την χρησιμοποιήσει για να δει αν μια τεχνολογία όπως η Intel Optane είναι κάτι που θα του ήταν χρήσιμο ή και βοηθητικό για τους δικούς του σκοπούς. Αυτό μπορεί να απαντηθεί με το αν ο χρήστης χρησιμοποιήσει τα μοντέλα της εργασίας για να δει αν και εφόσον η απόδοση της τεχνολογίας Optane είναι εντός το ορίων που θα ήθελε για την δική του περίπτωση.

Συγκεκριμένα, αναπτύχθηκαν δύο μοντέλα τα οποία υλοποιούν το ζητούμενο βασισμένα σε διαφορετικές προσεγγίσεις. Σκοπός αυτού του πρώτου μοντέλου είναι να υπολογίσει και να παρουσιάσει το ποσό καλά μπορείς να προβλέψεις το ζητούμενο με δεδομένα που παίρνεις από την εκτέλεση του υπό μελέτη προγράμματος σε τεχνολογία και υλικό που ήδη έχεις στην κατοχή σου ή είναι πολύ ευκολά προσβάσιμο. Το μοντέλο βασίζεται σε μετρήσεις που έγιναν με το εργαλείο perf[4] το οποίο μπορεί να εγκατασταθεί σε κάθε μηχανή Linux, με το perf μετρήσαμε μετρικές όπως LLC-loads & stores καθώς και τα misses τους. Με αυτών των τρόπο ξέραμε πόσες φορές το πρόγραμμα θα χρειαστεί να κάνει προσπέλαση ή να γράψει στην Persistent memory Optane μας, άρα θα μπορούσαμε να υπολογίσουμε και την καθυστέρηση που θα πρόκυπτε.

Το δεύτερο μοντέλο, έχει ως σκοπό να αναδείξει το πόση κατανόηση μπορείς να έχεις για την απόδοση τους προγράμματος σου, λαμβάνοντας υπόψη μόνο την υλοποίηση και τον κώδικα του, χωρίς να έχεις καθόλου στοιχεία για την εκτέλεση του σε πραγματικό υλικό. Πιο συγκεκριμένα η προσέγγιση αυτή μελετά την υλοποίηση της δομής δεομένων το hashmap όπως αυτό υλοποιείται στο PMDK[3] . Δηλαδή μελετήσαμε, ενδελεχώς των κώδικα του hashmap έτσι έστω το μοντέλο μας να είναι σε θέση να προβλέψει τις προσπελάσεις που θα γίνονται στην Optane μνήμη μας με αποτέλεσμα να μπορούμε να υπολογίσουμε την καθυστέρηση και τον χρόνο εκτέλεσης του.

Μια άλλη συνεισφορά αυτής της δουλείας είναι, μια καλύτερη κατανόηση στην απόδοση και στην καθυστέρηση που έχει ένα σύστημα μνήμης. Η απόδοση που θα έχει ένα σύστημα μνήμης βασίζεται σε πολλούς παράγοντες που κάποια από αυτούς ίσως να μη είναι προφανές να τους εντοπίσεις χωρίς να κάνεις πειράματα. Σε αυτή την εργασία έπρεπε να κάνουμε αρκετά πειράματα με διαφορετικές παραμέτρους για να υπολογίσουμε μετρικές που θεωρούσαμε χρήσιμες για την κατασκευή των μοντέλων. Με αυτά τα πειράματα ανακαλύψαμε ότι παράγοντες όπως το σύστημα αρχείων (File System) που χρησιμοποιεί το υλικό το οποίο τρέχαμε πάνω τα πειράματα παίζει σημαντικό ρόλο σε overheads καθυστέρησης της μνήμης. Αυτά είναι κάποια από τα σημαντικά στοιχεία που παρατηρήσαμε που μπορούν να μελετηθούν περεταίρω σε κάποια παροιμία δουλεία.

1.4 Δομή εργασίας

Η εργασία διαχωρίζεται ως εξής:
Στο «Κεφάλαιο 2» αναφέρουμε τα χαρακτηριστικά του συστήματος μας, στο «Κεφάλαιο 3», γίνεται αναφορά και επεξήγηση του low-level profiler που χρησιμοποιήσαμε για να μετρήσουμε την απόδοση (καθυστέρηση και εύρος ζώνης) της τεχνολογίας Optane μας, στο «Κεφάλαιο 4» εξηγούμε το πακέτο PMDK καθώς και τον μαζί με τα χαρακτηριστικά του, στο «Κεφάλαιο 5» γίνεται αναλυτική επεξήγηση της υλοποίησης και του τρόπου προσέγγισης των μοντέλων μας, στο «Κεφάλαιο 6» γίνεται σύγκριση των μοντέλων με αναφορά στις αδυναμίες και πλεονεκτήματα του κάθε μοντέλου, στο «Κεφάλαιο 7» αναφέρουμε κάποιες σχετικές δουλείες παρόμοιες με την εργασία αυτή αναφέρουμε τα συμπεράσματα μας, στο «Κεφάλαιο 8» τονίζουμε τεχνικές που μπορούν να χρησιμοποιηθούν και βελτιώσεις που μπορούν να γίνουν σε μελλοντική δουλεία, και τέλος στο «Κεφάλαιο 9» καταγράφουμε τα συμπεράσματα μας.

Κεφάλαιο 2

Περιγραφή συστήματος

2.1 Χαρακτηριστικά συστήματος

Για να κάνουμε τις μετρήσεις μας όπως είναι ευκολά αντιληπτό χρειαζόμαστε ένα σύστημα. Το σύστημα που χρησιμοποιήθηκε για αυτήν την εργασία είναι ο server shasta.in.ucy.ac.cy, που είναι στο δίκτυο του κλάδου πληροφορικής του ΠΚ. Στον Πίνακα 2.1 φαίνονται τα βασικά χαρακτηριστικά του συστήματος εκ το οποίων τα βασικότερα είναι 80 CPUs Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz, με 32 KB L1 cache, 1024 KB L2 cache, 28 MB L3 cache, και συνολικά το σύστημα είχε DDR4 128 GB RAM 3200MHz. Επίσης ο επεξεργαστής υποστηρίζει σημαίες για Instruction Level Parallelism ILP όπως sse1, sse2, sse3, sse4, και avx512 που ήταν απαραίτητες για το LENS που θα περιγράψουμε στην συνέχεια. Επίσης το σύστημα είναι εξοπλισμένο με 2 DIMMs Intel® Optane™ Persistent Memory (PMEM) [2] 128 GB το καθένα. Πρέπει να αναφέρουμε ότι το σύστημα μας έτρεχε σε Linux CentOS 7.

CPU	2x 40 CPUs Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz 2 threads per core 20 cores per socket
L1 L2 L3	32KB 8-way I$, 32KB 8-way D$, private L2 Cache 1024KB, 16-way, private L3 Cache 28MB, 11-way, shared
DRAM	DDR4, 4x 16GB, 3200MHz,

	DDR4, 4x 16GB, 2933MHz, 2 sockets, 6 channels per socket
NVDRAM	2x Intel Optane DIMM, 128 GB, 2666 MHz 2 sockets, 6 channels per socket
kernel gcc	linux-4.13.16 gcc-c++-7.2.1

Πίνακας 2.1 Χαρακτηριστικά server

Υπάρχουν δύο βασικές λειτουργείες που προσφέρονται με την χρήση της Intel Optane.

- Το πρώτο είναι το memory mode το οποίο είναι πολύ χρήσιμο για την αποθήκευση μεγάλου όγκου δεδομένων σε ταχύτητες της Optane αντί του δίσκου χωρίς να είναι αναγκαία αλλαγή στο κώδικα του προγράμματος το οποίο την χρησιμοποιεί. Ούτε κάποια αλλαγή στο λειτουργικό σύστημα αν απλά αποθηκεύεις τα αρχεία σου στην Optane.
- Και το δεύτερο είναι το app direct mode, με το οποίο μπορεί κάποιος να διαφοροποιήσει την αλληλουχία της μνήμης με το πρόγραμμα του, κάνοντας χρήση των πακέτων ανάπτυξης (development tool kits). Οι προγραμματιστές με χρήση βιβλιοθηκών από το πακέτο ανάπτυξης, μπορούν να έχουν απευθείας πρόσβαση σε διευθύνσεις μνήμης της Optane γλιτώνοντας έτσι καθυστερήσεις που θα υπήρχαν από την ιεραρχία μνήμης. Επίσης υπάρχει και ο συνδυασμός των προαναφερόμενων modes.

2.2 PMEM devices

Όπως αναφέραμε και πιο πριν, το υπό μελέτη αντικείμενο και ουσιαστικά αυτό που μοντελοποιούμε είναι η απόδοση αυτών των PMEM DIMMS. Αφού είναι devices μπορούμε να τα βρούμε κάτω από τον κατάλογο /dev σε ένα μηχάνημα τύπου Linux. Αρχικά για να δημιουργήσουμε κάποιο device που να είναι πάνω στο hardware PMEM, χρησιμοποιήσαμε την εντολή του Linux ndctl[6] ως εξής:

```
sudo ndctl create –namespace pmem0 -mode=fsdax
```

Εδώ σημαντικό είναι το mode να είναι fsdax (File System Direct Access), για να μπορούμε να χρησιμοποιήσουμε το direct mode της Optane μας. Το επόμενο βήμα για να μπορέσουμε να χρησιμοποιήσουμε την τεχνολογία Optane είναι αρχικά να δημιουργήσουμε ένα file system πάνω στο device που είναι η Optane και ακολούθως να κάνουμε mount το device που δημιουργήσαμε πάνω κατάλογο για να έχουμε πρόσβαση. Αυτό το πετύχαμε με τις εντολές:

```
sudo mkfs.xfs /dev/pmem
sudo mount -o dax /dev/pmem /mnt/pmem
```

Στην πρώτη εντολή επιλέγουμε το file system xfs που χρησιμοποιείται και στα υπόλοιπα devices. Στην δεύτερη εντολή το -o dax υποδηλώνει το direct access παρομοίως με το mode στην ndctl εντολή. Με την εκτέλεση αυτής της εντολής μπορούμε να χρησιμοποιήσουμε την Optane μας χρησιμοποιώντας τον κατάλογο /mnt/pmem.

Ακολούθως είχαμε να την ανάγκη να δούμε πληροφορίες για την κατάσταση και το health των PMEM devices που είχαμε δημιουργήσει. Με την εντολή impctl[7] είχαμε την δυνατότητα να δούμε κάποια από τα στοιχεία κατάστασης των devices.

Κεφάλαιο 3

Ο profiler LENS

3.1. Ο profiler LENS

Παρότι η Optane έρχεται με τιμές για την απόδοση της από τον κατασκευαστή, είχαμε την ανάγκη να μετρήσουμε με **ακρίβεια** τις μετρικές απόδοσης που θα χρειαζόμασταν (latency & memory bandwidth). Αυτή είναι και η δουλεία ενός profiler. Χρησιμοποιήσαμε τον Low-level profilEr for Non-volatile (LENS) για μη πτητικές μνήμες , που υλοποιήθηκε από τους Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson and Jishen Zhao στα πλαίσια του άρθρου "Characterizing and Modeling Non-Volatile Memory Systems"[1].

3.1.1. Εγκατάσταση του LENS

Ο LENS υλοποιήθηκε σαν kernel module, δηλαδή εκτελείται απευθείας μέσα στον πυρήνα για να αποφύγει το overhead αλλαγής μεταξύ kernel και user space. Στο σύστημα μας είχαμε εγκατεστημένο τον πυρήνα linux-3.10.0 που δεν ήταν συμβιβάσιμος με το LENS αφού περνάμε διάφορα errors κατά την μεταγλώττιση. Οπότε εγκαταστήσαμε [8] ένα πιο καινούργιο πυρήνα τον linux-4.13.16 που τηρούσε τις προδιαγραφές. Ακολούθως εγκαταστήσαμε το repository [1] του LENS στο σύστημα μας και το μεταγλωττίσαμε με την make.

Ένα ακόμη πρόβλημα που είχαμε με την εγκατάσταση του LENS ήταν, ότι ο gcc που είχαμε ήδη εγκατεστημένο στο σύστημα μας δεν υποστήριζε κάποιές εντολές στον κώδικά, όπως αναφορά σε καταχώρησες όπως είναι οι zmm και ymm. Επίσης δεν υποστήριζε κάποια direct inlines σε functions που ήταν δηλωμένα σε άλλα αρχεία. Για να λύσουμε αυτό το πρόβλημα εγκαταστήσαμε ένα νεότερο version του μεταγλωττιστή τον gcc-c++-7.2.1 από το πακέτο devtoolset-7 [9] και κάναμε compile τον κώδικα με αυτόν. Την νέα έκδοση του μεταγλωττιστή την κατεβάσαμε απευθείας από το διαδίκτυο και κάναμε extract τα αρχεία και compile τον πηγαίο κώδικα.

Αφού ο LENS είναι ένα kernel module, αν εμφανιζόταν κάποιο πρόβλημα την ώρα της εκτέλεσης (run time error) θα αναμέναμε ότι θα χάναμε όλο το σύστημα, όπως και συνέβαινε. Για την ακρίβεια υπήρχε μια μεταβλητή ορισμένη για 256 GB Optane, ενώ η δική μας είναι 128 GB οπότε σε κάποιο σημείο της εκτέλεσης είχαμε segmentation fault μέσα στον kernel με αποτέλεσμα να έχουμε “kernel panic” και να πέφτει ολόκληρο το λειτουργικό. Για να λύσουμε αυτό το πρόβλημα αλλάξαμε την μεταβλητή αυτή για να είναι συμβατή με το δικό μας υλικό.

3.1.2. Μετρήσεις με τον LENS

Το LENS, παρέχει τις εξής λειτουργείες για την ανάλυση και μέτρηση της απόδοσης του συστήματος του χρήστη:

- Buffer prober: σκοπός αυτής της λειτουργίας είναι η ανάλυση της αρχιτεκτονικής των buffers που έχουν τα Optane DIMMs, ως επίσης και χαρακτηριστικών που έχουν.
- Policy prober: σκοπός αυτής της λειτουργίας είναι ανάλυση των πολιτικών για μεταφορά δεδομένων ανάμεσα σε διαφορετικά Optane DIMMs.
- Performance prober: αυτή η λειτουργία διευκολύνει τις δυο πιο πάνω λειτουργείες για την ανάλυση μετρικών απόδοσης όπως memory bandwidth και latency.

Στην Εικόνα 3.1 φαίνεται για την κάθε λειτουργία το microbenchmark που χρησιμοποιείται καθώς και σκοπός και η αρχιτεκτονική που έχει το καθένα.

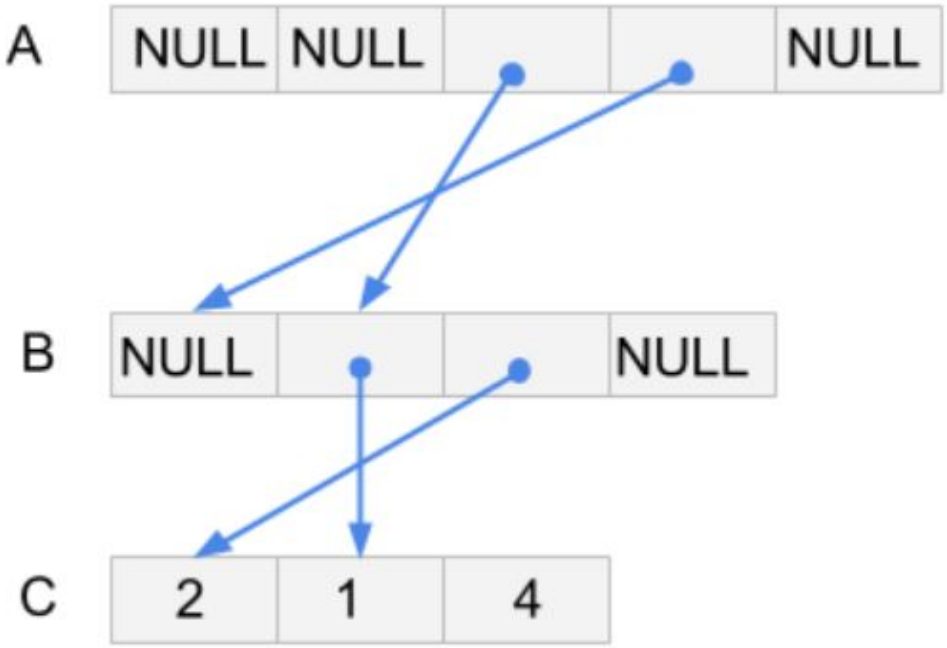
Prober	Microbenchmark	Hardware Behavior	Microarchitecture
Buffer	PtrChasing (64B block)	Buffer overflow	Buffer size
	PtrChasing (various block)	R/W amplification	Buffer entry size
	Read-after-write	Data fast-forwarding	Buffer hierarchy
Policy	Sequential/Strided write	Interleaving speedup	Interleaving scheme
	Overwrite (256B region)	Data migration	Migration latency
	Overwrite (various region)	Data migration	Migration block size
Perf.	Strided write	Stable amplification	Internal bandwidth
	N/A	N/A	Internal latency

Πίνακας 3.1 Λειτουργείες του LENS

Είναι καλό ν αναφέρουμε ότι όλα τα microbencmarks του LENS υλοποιήθηκαν σε επίπεδο assembly για να υπάρχει όσο μεγαλύτερος έλεγχος της μνήμης αλλά και να ελαχιστοποηθούν τα overheads που πιθανόν να προκύπταν από εντολές υψηλού επιπέδου. Αρχικά, χρησιμοποιήσαμε το microbenchmark Pointer Chasing με μεταβλητό block size από την λειτουργία Buffer. Το pointer chasing είναι μια τεχνική όπου γίνονται ακολουθιακά εξαρτόμενες προσπελάσεις στην μνήμη με την χρήση συνδεδεμένης λίστας. Με αυτό τον τρόπο μπορούμε να μετρήσουμε την πραγματική καθυστέρηση του συτήματος μνήμης μας αφού επειδή η προσπελάσεις είναι όλες εξαρτημένες δεν υπάρχει καθόλου παραλληλία στο σύστημα μνήμης. Το microbenchmark PtrChasing, βασίζεται πάνω στην πιο πάνω εν λόγο τεχνική και εκτελεί κάθε φορά εγγραφή ή διάβασμα πάνω στο device που δέχεται ως είσοδο, οπότε αποφασίσαμε πως είναι ένα καλό παράδειγμα για να υπολογίσουμε με αυτόν τον τρόπο την καθυστέρηση της Optane του συστήματος μας. Επίσης η υλοποίηση έγινε με εντολές non-temporal AVX512 οι οποίες παραλείπουν την ιεραρχία μνήμης δηλαδή τα δεδομένα τους δεν αποθηκεύονται στις caches, αλλά στην μνήμη τυχαίας προσπέλασης την PMEM στην περίπτωση μας. Αυτό το microbenchmark το εκτελέσαμε με την ακόλουθη εντολή με στον κατάλογο LENS/scripts του [1].

```
./lens.sh /dev/pmem4 /dev/pmem0 prober/buffer/pointer_chasing.sh
read
```

Σε αυτήν την περίπτωση το /dev/pmem0 πρέπει να είναι ένα PMEM device όπως το αναφέραμε στην ενότητα PMEM devices, ενώ το /dev/pmem4 μπορεί να είναι ή PMEM device ή DRAM emulated device (θα εξηγιθεί αναλυτικά σε μετέπιτα ενότητα).



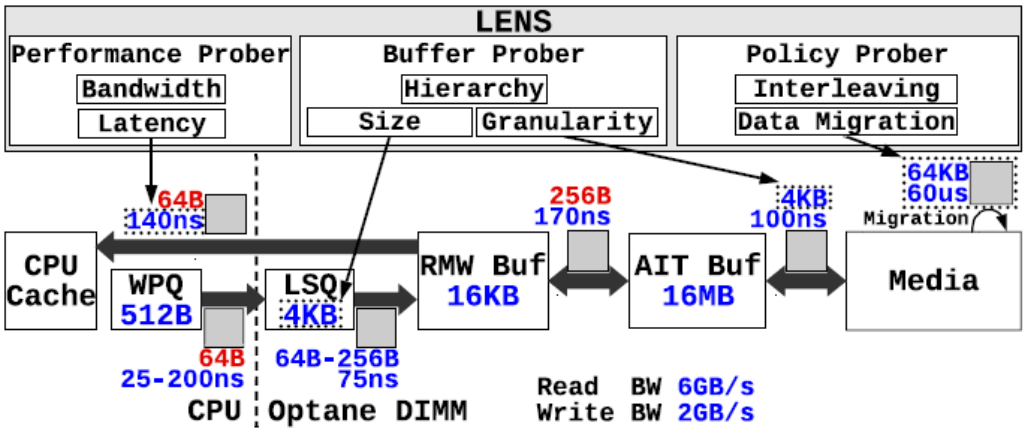
Εικόνα 3.1 Μορφή Pointer Chasing

Ακολουθώς θέλαμε να μετρήσουμε το έυρως ζώνης του συστήματος μας. Αυτό επιλέξαμε να το κάνουμε με το microbenchmark strited write από την λειτουργία performance. Το stride microbenchmark ουσιαστικά γράφει ή διαβάζει σειριακά σε ένα σύνολο από cache lines που βρίσκονται σε σταθερή απόσταση μεταξύ τους, αυξάνοντας σε κάθε ενέργεια το access size. Αφού οι ενέργειες σε σταθερά cache lines τότε υπάρχει χωρική τοπικότητα, άρα θα έχουμε παραλληλία συνεπώς μπορούμε να μετρήσουμε το εύρος ζώνης του συστήματος μνήμης μας. Το συγκεκριμένο microbenchmark το εκτελούμε με την εντολή:

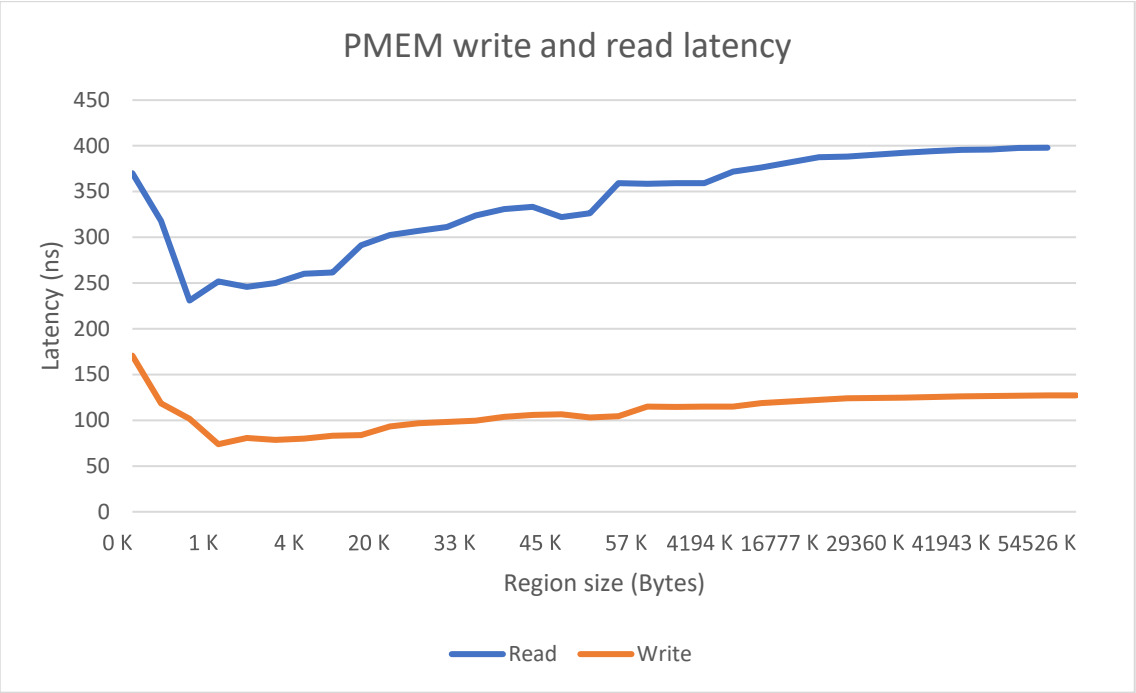
```
./lens.sh /dev/pmem4 /dev/pmem0 prober/performance/stride_bw.sh
read
```

3.2. Αποτελέσματα του LENS

Αφού εκτελέσαμε τα microbenchmarks που επιλέξαμε, έπρεπε να μεταφράσουμε τα αποτελέσματα σε μορφή που θα ήταν εύκολη η ανάλυση τους. Αυτό το κάναμε με έτοιμα python scripts από το repository του LENS. Στην Εικόνα 3.3, βλέπουμε την καθυστέρηση της PMEM μας και για εγγραφή και για διάβασμα για ένα block μεγέθους 64B, σε σχέση με την προσπέλαση του region size. Επίσης, παρατηρούμε ότι αυξάνονται οι χρόνοι όσο αυξάνεται το region size αυτό συμβαίνει κυρίως επειδή υπερχειλίζουν από το μέγεθος του region τα RMW και AIT buffers που φαίνονται στη Εικόνα 3.2. Αυτά τα buffers βρίσκονται μέσα στην Optane και χρησιμοποιούνται για προσωρινή αποθήκευση και για read και για write. Ένας άλλος λόγος που αυξάνεται η καθυστέρηση με την αύξηση του region size είναι επειδή έχουμε και πιο πολλά cache misses αφού βάζουμε μεγαλύτερα region μέσα στις caches.

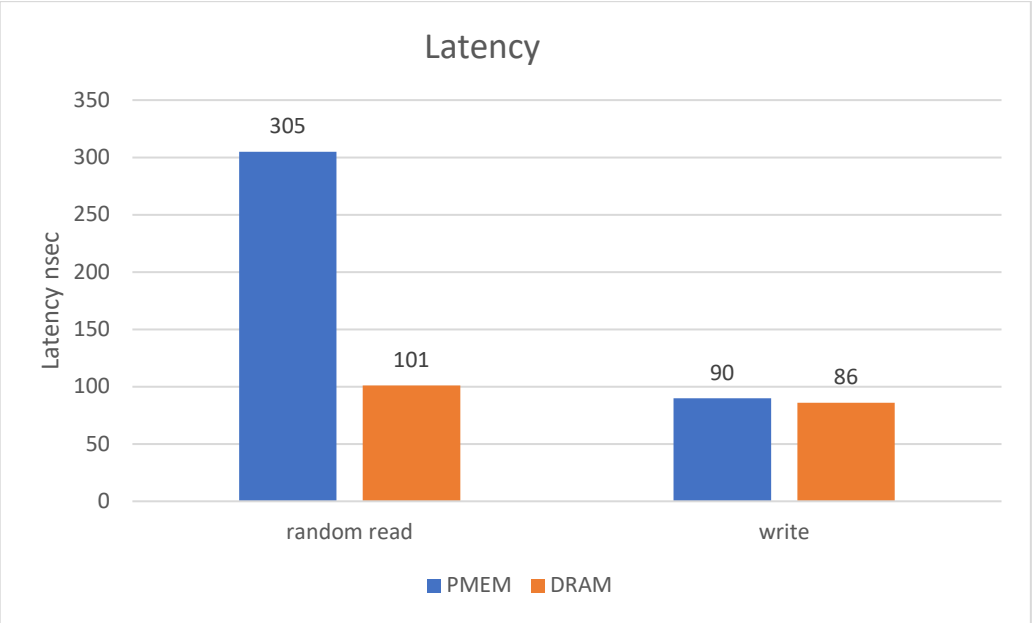


Εικόνα 3.2 Σύνοψη LENS (από το άρθρο [1])



Εικόνα 3.3 Read & write lantecy

Στην Εικόνα 3.4 βλέπουμε την διαφορά για την τελικές τιμές της καθυστέρησης ανάμεσα σε PMEM και DRAM. Αξιοσημείωτο είναι ότι στην καθυστέρηση διαβάσματος έχουμε πολύ μεγαλύτερη διαφορά απ' ότι στην καθυστέρηση εγγραφής ανάμεσα στα δυο συστήματα μνήμης. Χρησιμοποιούμε για τα μοντέλα μας τον μέσο όρο στις μετρήσεις που κάναμε, δηλαδή η τιμή της καθυστέρησης που θα χρησιμοποιούμε θα είναι 305 nsec για το read και 90 nsec για το write.



Εικόνα 3.4 Συγκριση καθυστέρησης ανάμεσα σε PMEM και DRAM

Κεφάλαιο 4

Το Πακέτο Ανάπτυξης PMDK

4.1. Εισαγωγή στο PMDK

Όπως αναφέραμε και στην ενότητα Παρακίνηση με την τεχνολογία Intel Optane Persistent Memory, και συγκεκριμένα στο app direct mode έρχεται και το πακέτο ανάπτυξης Persistent Memory Development Kit (PMDK). Το PMDK προσφέρεται στα πλαίσια του pmem.io [3] , που ουσιαστικά είναι μια γενική αναφορά στην μη πτητική μνήμη και τα χαρακτηριστικά της. Το PMDK μπορεί να ορισθεί ως μια συλλογή βιβλιοθηκών ανοιχτού πηγαίου κώδικα που έχουν αναπτυχθεί για διάφορες περιπτώσεις χρήσης. Κάθε βιβλιοθήκη είναι συντονισμένη, επικυρωμένη στην ποιότητα παραγωγής και πλήρως τεκμηριωμένη. Αυτές οι βιβλιοθήκες παρέχουν σταθερά APIs, έτσι ώστε οι προγραμματιστές να μπορούν να εφαρμόζουν γρήγορα τις δυνατότητες PMEM σε εφαρμογές χωρίς να ανησυχούν για τις λεπτομέρειες εφαρμογής υλικού ή τις αλλαγές γενεών.

4.2. Βιβλιοθήκες του PMDK

Το PMDK προσφέρει πληθώρα από βιβλιοθήκες που μπορούν να χρησιμοποιηθούν χωρίς κάποιες αυστηρές προδιαγραφές ή προ απαιτούμενα συστήματος. Πιο κάτω αναφέρουμε κάποιες από τις πιο βασικές βιβλιοθήκες μαζί με την χρήση/σκοπό τους:

- **libpmemobj**: αυτή η βιβλιοθήκη παρέχει την δυνατότητα για δοσοληψίες πάνω σε αντικείμενα (objects) στην μνήμη, δέσμευση μνήμης, και γενικές

λειτουργείες για προγραμματισμό με μη πτητικές μνήμες, θεωρείτε κατάλληλη για να ξεκινήσεις να δουλεύεις με το PMDK.

- **libpmemblk:** αυτή η βιβλιοθήκη υποστηρίζει ατομικές αλλαγές σε πίνακες από μπλοκ της Persistent μνήμης του ίδιο μεγέθους, είναι χρήσιμη για προγράμματα που κάνουν εγγραφές σε σταθερά cache lines.
- **pmempool:** αυτή η βιβλιοθήκη παρέχει την δυνατότητα για τον έλεγχο πολλαπλών αρχείων που δημιουργήθηκαν από τις άλλες βιβλιοθήκες του πακέτου, είναι χρήσιμη για administrators ή προγραμματιστές κυρίως για debugging.
- **libpmem:** αυτή η βιβλιοθήκη παρέχει χαμηλού επιπέδου έλεγχο και διαχείριση της persistent μνήμης, είναι η βιβλιοθήκη που χρησιμοποιεί και η βιβλιοθήκη libpmemobj για τις μεταβάσεις μνήμης.

Για τους δικούς μας σκοπούς χρησιμοποιήσαμε την βιβλιοθήκη libpmemobj. Η βιβλιοθήκη libpmemobj είναι μια από τις πιο απλές βιβλιοθήκες που περιέχει τον πακέτο, και είναι και η προτεινόμενη για χρήστες που δεν έχουν προηγούμενη εμπειρία με το PMDK. Οπότε ήταν μια καλή επιλογή να χρησιμοποιήσουμε κάποιο από παραδείγματα που περίεχε η βιβλιοθήκη, για αυτή την εργασία. Σε αυτό το σημείο αναφέρουμε πως δεν χρειάστηκε να γράψουμε δικό μας κώδικα με την χρήση της βιβλιοθήκης, αλλά χρησιμοποιήσαμε έτοιμα παραδείγματα που πρόσφερε υλοποιημένα.

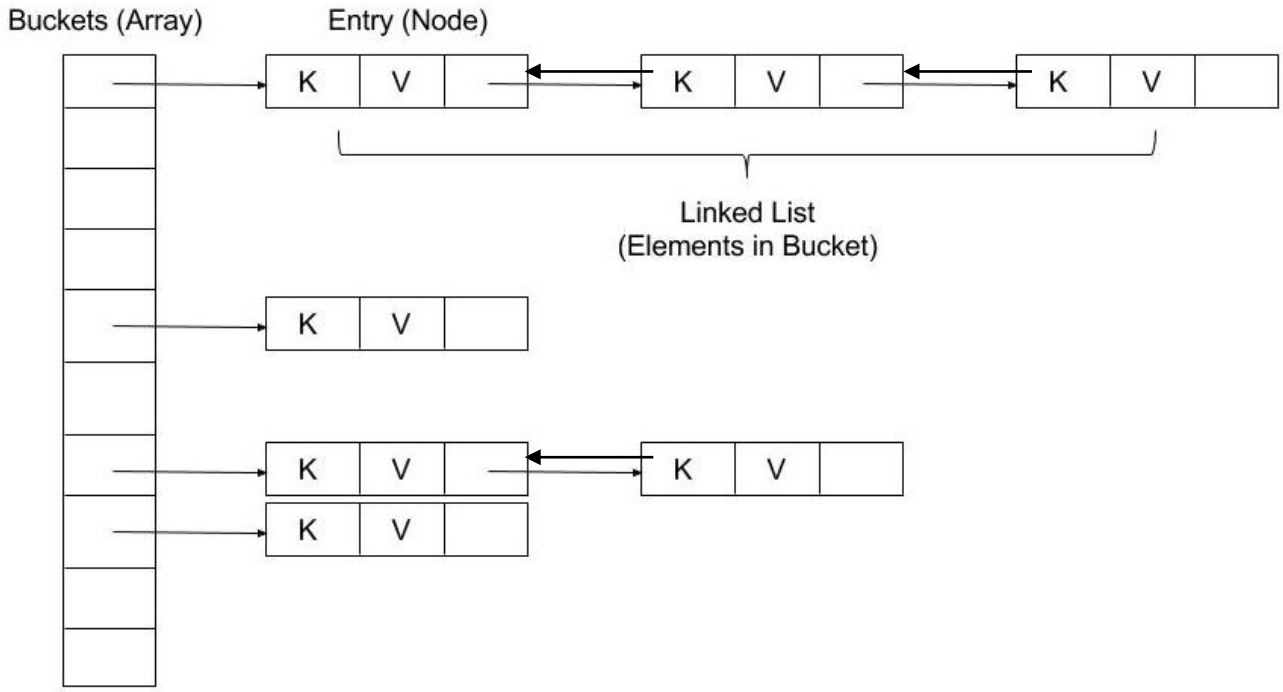
4.2.1. Hashmap στο libpmemobj

Σε αυτή την εργασία επιλέξαμε να χρησιμοποιήσουμε την δομή δεδομένων hashmap όπως αυτή υλοποιείται στην βιβλιοθήκη libpmemobj. Χρησιμοποιούμε το hashmap για να μετρήσουμε την απόδοση της PMEM μας. Για την ακρίβεια χρησιμοποιήσαμε την υλοποίηση hashmap_atomic. Στην Εικόνα 4.1 φαίνεται το πως θα μπορούσαμε να φανταστούμε την υλοποίηση αυτή.

Η κύρια διαφορά με μια κλασική υλοποίηση hashmap[10], δηλαδή ένα πίνακα που έχει μια λίστα σε κάθε είσοδο του πίνακα από ζευγάρια τιμής και κλειδιού, είναι ότι στην δική μας περίπτωση το value από το ζευγάρι <key,value> είναι ένας απευθείας pointer

σε persistent memory, και όχι δεδομένα ή κάποιος κλασικός pointer. Μια άλλη σημαντική διαφορά με κλασική υλοποίηση είναι ότι στην υλοποίηση της βιβλιοθήκης η λίστα που υπάρχει σε κάθε bucket entry είναι μια διπλά συνδεδεμένη λίστα και όχι μονά συνδεδεμένη. Με αυτό τον τρόπο υπάρχει πιο γρήγορη προσπέλαση της λίστας στην αναζήτηση, αλλά από την άλλη υπάρχει μεγαλύτερη σπατάλη χώρου λόγο των επιπλέον pointers. Επίσης όπως φαίνεται και στον όνομα της δομής όλες η πράξεις που μπορείς να κάνεις με αυτή είναι ατομικές, δηλαδή εγγυόνται ότι μόνο πράξη (get, insert, remove) θα χρησιμοποιεί την δομή την ίδια χρονική στιγμή.

Κάποιες ομοιότητες με την κλασική υλοποίηση hashmap είναι, ότι και στις δύο περιπτώσεις το key παραμένει μια ακέραιά τιμή, που αντιπροσωπεύει τον αριθμό του bucket. Επίσης το hashing function γίνεται με τον universal hashing τρόπο.



Εικόνα 4.1 Μορφή του hashmap

Στην Εικόνα 4.1.2, βλέπουμε τους πραγματικούς τύπους δεδομένων της C που απαρτίζουν το hashmap στην βιβλιοθήκη. Ο ειδικός τύπος δεδομένων PMEMoid, είναι ένας απευθείας δείχτης σε persistent memory. Για την ακρίβεια ο τύπος PMEMoid είναι

η αντιστοιχία σε virtual memory της φυσικής μνήμης PMEM και μπορεί να υπολογιστεί εφόσον ξέρουμε την virtual address του αντίστοιχου pool που βρίσκεται το PMEMoid μας και με την πρόσθεση του offset `(void *)((uint64_t)pool + oid.off)`. Οι εικονικές διευθύνσεις των PMEM pools βρίσκονται σε lookup tables παρόμοια με τις εικονικές διευθύνσεις της DRAM.

```
typedef struct pmemoid {
  uint64_t pool_uuid_lo;
  uint64_t off;
} PMEMoid;
```

Εικόνα 4.1.1 PMEMoid

```c
struct entry {
    uint64_t key;
    PMEMoid value;

    /* list pointer */
    POBJ_LIST_ENTRY(struct entry) list;
};

struct entry_args {
    uint64_t key;
    PMEMoid value;
};

POBJ_LIST_HEAD(entries_head, struct entry);
struct buckets {
    /* number of buckets */
    size_t nbuckets;
    /* array of lists */
    struct entries_head bucket[];
};

struct hashmap_atomic {
    /* random number generator seed */
    uint32_t seed;

    /** hash function coefficients */
    uint32_t hash_fun_a;
    uint32_t hash_fun_b;
    uint64_t hash_fun_p;

    /* number of values inserted */
    uint64_t count;
    /* whether "count" should be updated */
    uint32_t count_dirty;

    /* buckets */
    TOID(struct buckets) buckets;
    /* buckets, used during rehashing, null otherwise */
    TOID(struct buckets) buckets_tmp;
};
```

Εικόνα 4.1.2 Κώδικας κορμού του hashmap_atomic

Στην βιβλιοθήκη του hashmap υπάρχουν και άλλες επιλογές υλοποίησης όπως hashmap_tx, και hashmap_rp. Η υλοποίηση hashmap_tx είναι η transactional έκδοση του hashmap όπου χρησιμοποιείτε μονά συνδεμένη λίστα και κάθε δοσοληψία είτε εκτελείτε 100% είτε 0%, δηλαδή κάθε operation ή θα εκτελεστεί πλήρως ή θα τερματιστεί εντελώς σε περίπτωση κάποιου προβλήματος. Η hashmap_rp υλοποίηση η διαφορά που έχει σε σχέση με τις άλλες δύο είναι ότι διαχειρίζεται τις συγκρούσεις του hashing με τον αλγόριθμο του Robin Hood. Σε αυτή την εργασία δεν μελετήσαμε αυτές τις δύο υλοποιήσεις αλλά μπορεί να φανούν βοηθητικές σε μελλοντική δουλεία.

Βέβαιά, στην βιβλιοθήκη libpmemobj υπάρχει πληθώρα από υλοποιημένες δομές δεδομένων προς χρήση. Κάποιες από αυτές είναι queue, map, tree-map, linked-list, list-map, και string-store. Δυστυχώς ούτε αυτές της υλοποιήσεις είχαμε χρόνο να της μελετήσουμε.

4.3. Τα benchmarks του PMDK

Ο τρόπος με τον οποίο κάναμε όλες τις μετρήσεις μας ήταν με την χρήση των benchmarks, που έρχονται με το πακέτο. Γενικά ένα benchmark είναι ένα μετρο-πρόγραμμα το οποίο παίρνει κάποιες παραμέτρους σχετικές με το αντικείμενο προς μελέτη, και υπολογίζει ή/και μετράει κάποιες μετρικές απόδοσης που είναι χρήσιμες για ανάλυση του ζητούμενου. Σημαντικό εδώ, να αναφέρουμε ότι ο τρόπος με των οποίο μπορούμε να χρησιμοποιήσουμε την PMEM με τα benchmarks είναι επιλέγοντας το αρχείο με το οποίο αλληλοεπιδρά το benchmark (δηλαδή γράφει και διαβάζει από αυτό) να είναι κάτω από κατάλογο που είναι mounted σε PMEM device.

4.3.1. Λειτουργείες των benchmarks

Στην δική μας περίπτωση τα benchmarks του πακέτου προσφέρουν δυνατότητα ελέγχου για σχεδόν όλες τις βιβλιοθήκες που περιγράψαμε στη ενότητα Βιβλιοθήκες του PMDK, ακόμη για κάποιες που δεν αναφέραμε. Ο δικός μας σκοπός ήταν να μετρήσουμε την απόδοση του hashmap πάνω σε persistent memory, οπότε μας ήταν χρήσιμες 3 επιλογές του benchmark οι οποίες είναι οι εξής:

- map_get, σε αυτή την επιλογή το benchmark εκτελεί την λειτουργία get πάνω στην δομή δεδομένων που παίρνει σαν είσοδο.
- map_insert, σε αυτή την επιλογή το benchmark εκτελεί την λειτουργία insert πάνω στην δομή δεδομένων που παίρνει σαν είσοδο.
- map_remove, σε αυτή την επιλογή το benchmark εκτελεί την λειτουργία remove πάνω στην δομή δεδομένων που παίρνει σαν είσοδο.

Για την ακρίβεια, επικεντρωθήκαμε στις επιλογές map_get και map_insert που ουσιαστικά μας δίναν το ζητούμενο δηλαδή την καθυστέρηση μνήμης. Αφού με την λειτουργία get μπορείς να μετρήσεις το read το συστήματος μνήμης και με το insert το store. Ακολουθεί ένα παράδειγμα εντολής για την εκτέλεση ένας benchmark με τις παραμέτρους που μας ήταν χρήσιμες.

```
./pmembench map_get --type hashmap_atomic --alloc -n 1000000 -f /mnt/pmem/file -d 1
```

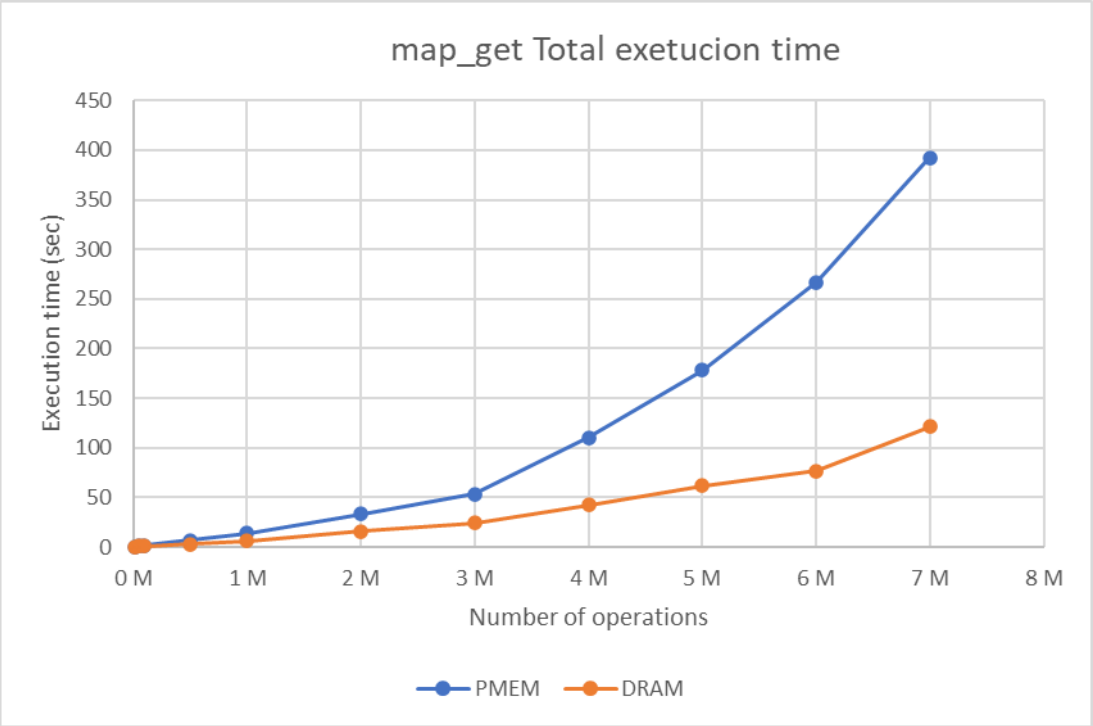
- το map_get δηλώνει την λειτουργία που θέλεις να μετρήσεις
- το type την δομή δεδομένων που θα χρησιμοποιηθεί
- το alloc δηλώνει ότι το στοιχεία που θα γράφονται θα είναι συγκεκριμένου μεγέθους
- το n είναι ο αριθμός των operations που θα εκτελεστούν
- το f είναι το μονοπάτι του αρχείου πάνω στο οποίο αποθηκεύεται η δομή
- το d είναι το μέγεθος του κάθε στοιχείου σε bytes

4.3.2. Αποτελέσματα των benchmarks

Τα benchmarks υπολογίζουν αρκετές μετρικές, κάποιες από αυτές είναι ο συνολικός χρόνος εκτέλεσης από το κομμάτι της διεκπεραίωσης μόνο της λειτουργίας (get, insert,remove), ο μέσος, ο ελάχιστος, και ο μέγιστος από τους χρόνους εκτέλεσης της λειτουργίας. Επίσης ο μέσος όρος, η ελάχιστη, και η μέγιστη καθυστέρηση για ένα operation για την εν λόγω λειτουργία. Για εμάς, η πιο χρήσιμη μετρική ήταν ο συνολικό

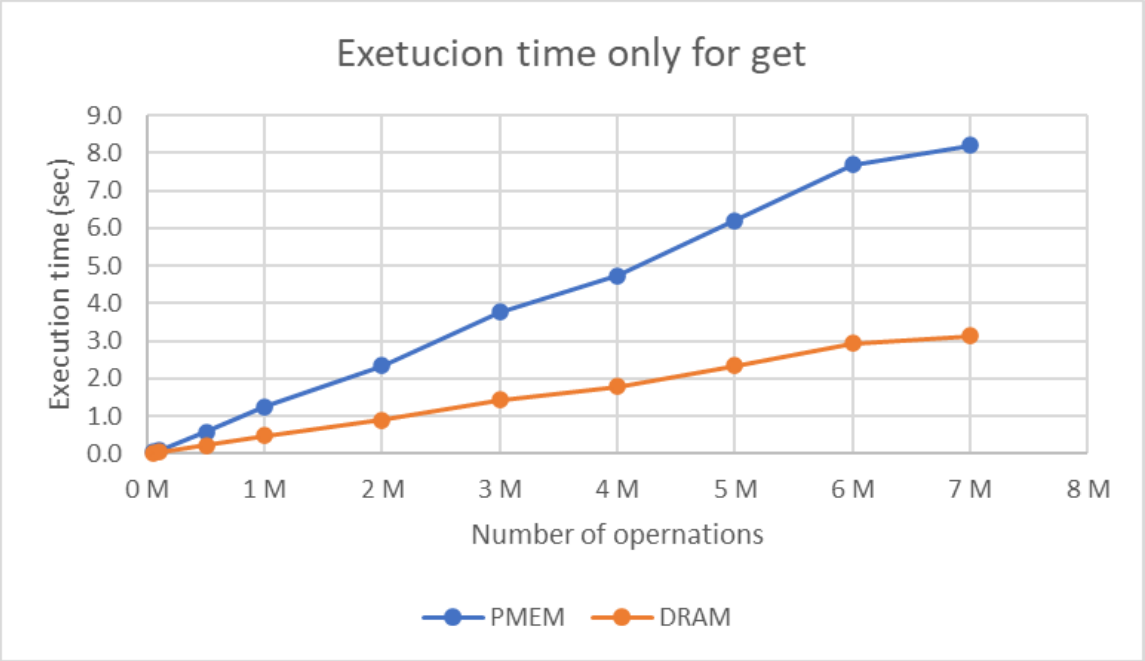
χρόνος εκτέλεσης και η καθυστέρηση μνήμης της λειτουργίας έτσι ώστε να μπορούμε να επαληθεύσουμε τα αποτελέσματα από τα μοντέλα μας.

Στην Εικόνα 4.2 βλέπουμε τον συνολικό χρόνο εκτέλεσης όλου του benchmark, με λειτουργία την map_get πάνω στην δομή hashmap_atomic. Στο παράδειγμα μεταβάλλεται ο αριθμός operations και το μέγεθος των στοιχείων παραμένει σταθερό. Παρατηρούμε ότι ο χρόνος που χρειάζεται η PMEM είναι λίγο μεγαλύτερος αρχικά, αλλά και ότι με την αύξηση των operations, αυξάνεται περισσότερο η διαφορά ανάμεσα σε PMEM και DRAM. Αυτό οφείλεται στην πιο μεγάλο εύρος ζώνης που υπάρχει στην DRAM, που συνεπώς με την αύξηση των ενεργειών είναι σε θέση να μεταφέρει πιο γρήγορα τα cache lines στις caches όταν υπάρχει miss. Ουσιαστικά εδώ ένα πολύ μεγάλο ποσοστό του χρόνου (γύρω στο 95%) είναι στην δημιουργία και αρχικοποίηση του hashmap. Για αυτό στα επόμενα πειράματα όπως και στα μοντέλα θα χρησιμοποιούμε την χρόνο διεκπεραίωσης μόνο της κάθε λειτουργίας.



Εικόνα 4.2 Συνολίκος χρονος εκτέλεσης του map_get

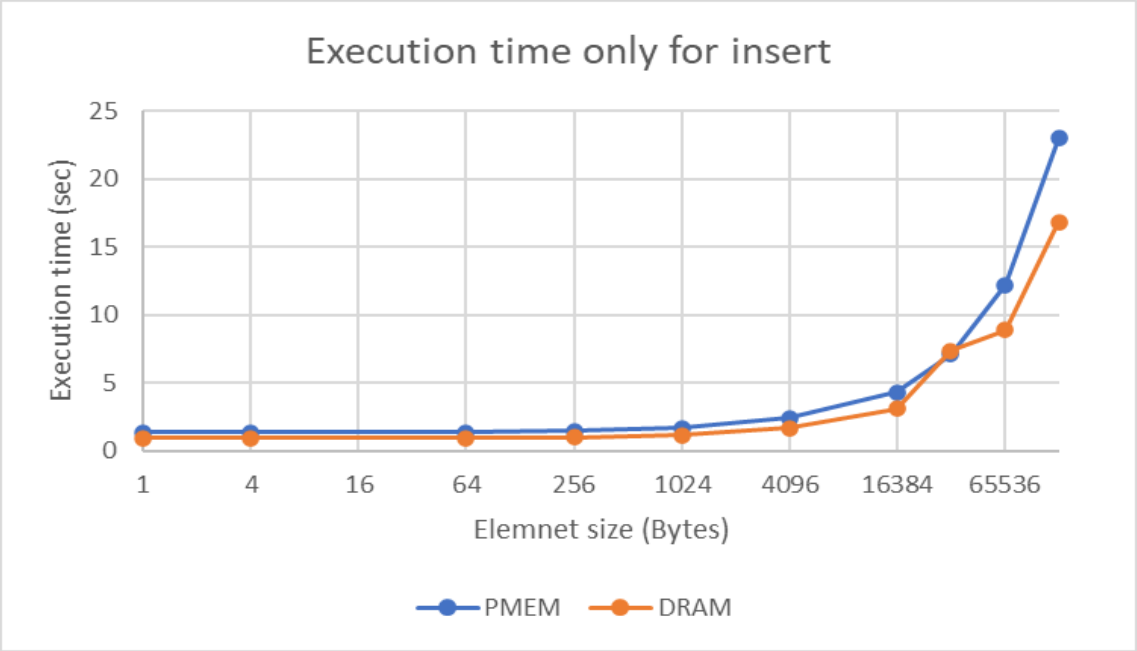
Στην Εικόνα 4.3, μπορούμε να δούμε τον χρόνο που χρειάζετε να εκτελεστεί η λειτουργία map_get, κατά την διάρκεια εκτέλεσης του ανάλογου benchmark. Παρομίως με την προηγούμενη γραφική έχουμε σταθερό μέγεθος στα στοιχεία στο 1B, και μεταβλητό αριθμό από operation. Όπως είναι αναμενομενό, η PMEM χρειάζεται μεγαλύτερο χρόνο να ολοκληρώσει την λειτουργία. Επίσης και σε αυτήν την περίπτωση βλέπουμε ότι σε σχέση με την DRAM, η PMEM έχει ένα μεγαλύτερο ρυθμό αύξησης με την αύξηση του αριθμού των operations. Γνωρίζουμε το γεγονός ότι η PMEM έχει μεγαλύτερη καθυστέρηση από την DRAM, άρα μπορούμε να πούμε είναι και αυτός ένας λόγος για τον οποίο βλέπουμε αυτήν την διαφορά στον ρυθμός αύξησης του χρόνο εκτέλεσης.



Εικόνα 4.3 Χρόνος εκτέλεσης λειτουργίας map_get

Στην Εικόνα 4.4, παρουσιάζεται ο χρόνος που χρειάζεται για να ολοκληρωθεί η λειτουργία insert. Σε αντίθεση με τις προηγούμενες γραφικές σε έχουμε μεταβλητό μέγεθος στοιχείων που φαίνεται στον άξονα X (σε λογαριθμική κλίμακα), παράλληλα έχουμε σταθερό αριθμό από operations που είναι 100,000. Σε αυτήν την περίπτωση βλέπουμε πως η διαφορά μεταξύ σε PMEM και DRAM είναι πιο μικρή απ' ότι στην λειτουργία get. Αυτό οφείλετε στο ότι η διαφορά στην καθυστέρηση εγράφης είναι πιο

μικρή παρ’ ότι στην διαφορά καθυστέρησης του διαβάσματος μεταξύ σε DRAM και PMEM, απ’ ότι αναφέρουμε και στο Κεφάλαιο 2.



Εικόνα 4.4 Χρόνος εκτέλεσης λειτουργίας map_insert

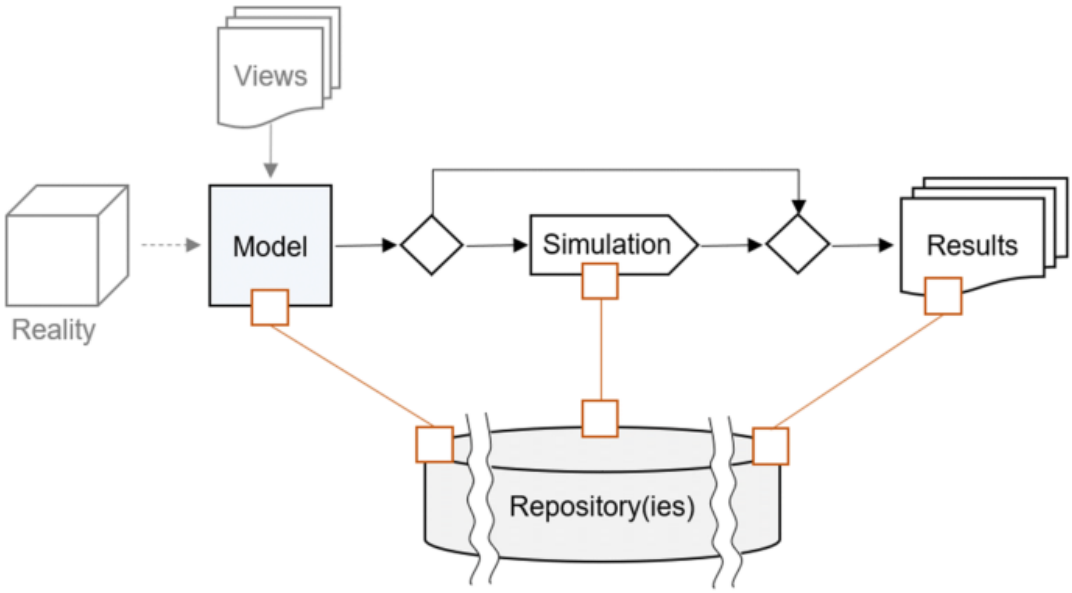
Κεφάλαιο 5

Περιγραφή μοντέλων

5.1. Ορισμός αναλυτικού μοντέλου

Ένα αναλυτικό μοντέλο είναι μια σειρά από μαθηματικές εξισώσεις ή μαθηματικά μοντέλα που έχουν σκοπό να δώσουν μια απάντηση σε κάποιο συγκεκριμένο ερώτημα. Ένα αναλυτικό μοντέλο εκτός από μοντελοποίηση κάποιο μαθηματικού ή φυσικού προβλήματος μπορεί να χρησιμοποιηθεί για υπολογισμό μετρικών απόδοσης ή πιο γενικά απόδοσης κάποιου υπό μελέτη συστήματος ή σχεδιασμού. Αυτό γίνεται και στην δική μας περίπτωση, δηλαδή προσπαθήσαμε να μοντελοποιήσουμε την απόδοση συστημάτων κυρίας μη πτητικής μνήμης (Non Volatile PMEM).

Συγκεκριμένα, όπως φαίνεται και στην Εικόνα 5.1 τα μοντέλα μας δέχονται κάποια δεδομένα από διαθέσιμες πηγές (π.χ. DRAM) ως είσοδο, όπως αριθμός από στοιχεία, μέγεθος στοιχείων και τα χαρακτηριστικά του υλικού, και υπολογίζουν μετρικές απόδοσης όπως ο χρόνος εκτέλεσης, η καθυστέρηση του συστήματος μνήμης, και πιο εξιδεικευμένες μετρικές για κάθε μοντέλο οι οποίες ήταν σημαντικές για κάθε περίπτωση.



Εικόνα 5.1 Παράδειγμα αναλυτικού μοντέλου

5.2. Τι μοντελοποιούμε

Αρχικός μας στόχος ήταν, η μοντελοποίηση της απόδοσης της PMEM μας, και το πρόγραμμα που θα χρησιμοποιούσαμε για να κάνουμε μετρήσεις που θα μας βοηθούσαν να το κάνουμε αυτό ήταν τα benchmarks του πακέτου ανάπτυξης PMDK. Όπως αναφέραμε και στην ενότητα Αποτελέσματα των benchmarks, το συνολικό benchmark ήταν αρκετά περίπλοκο και επίσης είχε αρκετά overheads επιπλέον από την λειτουργία που μας ενδιέφερε. Επιπρόσθετα, ο χρόνος που χρειαζόταν η δημιουργία και η αρχικοποίηση του hashmap ήταν περίπου 90% του συνολικού χρόνου και η λειτουργία που μελετούσαμε μονό γύρω στο 5%. Οπότε το αντικείμενο που αποφασίσαμε να μοντελοποιήσουμε ήταν η λειτουργία που είχαμε σαν παράμετρο καθαυτό και όχι ολόκληρο το benchmark. Ένας άλλος λόγος που μας ώθησε σε αυτή την απόφαση είναι ότι το κομμάτι κώδικα μόνο της λειτουργίας που μας ενδιέφερε ήταν πολύ πιο απλό και πολύ πιο μικρό σε σχέση με την ανάλυση του συνολικού κώδικα.

Συγκεκριμένα, μοντελοποιούμε την λειτουργία map_get της δομής δεδομένων hashmap_atomic. Επίσης, αφού μοντελοποιούμε μόνο την λειτουργία get οι μετρήσεις που πήραμε είναι αποκλείστηκα για το κομμάτι κώδικα που μας ενδιαφέρει.

5.3. Μοντέλο “Performance Counters”

Η πρώτη μας προσπάθεια για σχεδιασμό μοντέλου βασίζεται πάνω σε performance counters. Για την ακρίβεια χρησιμοποιήσαμε το εργαλείο perf [4] του Linux, και τους performance counters του. Ακολούθως, με βάσει αυτά τα αποτελέσματα κτίζουμε τις εξισώσεις του μοντέλου.

5.3.1. Συλλογή δεδομένων

Όπως καταλαμβάνουμε και από την ονομασία του μοντέλου, βασίζεται σε μετρήσεις που έπρεπε να κάνουμε. Η βασική ιδέα σε αυτό το μοντέλο ήταν να κάνουμε μετρήσεις για το benchmark που επιλέξαμε, πάνω σε τεχνολογία που υπάρχει και είναι προσιτή από όλους δηλαδή η τεχνολογία DRAM, και να προσπαθήσουμε να προσομοιώσουμε την απόδοση της PMEM με βάσει αυτά τα δεδομένα.

Το perf, είναι ένα εργαλείο που σου δίνει ένα τεράστιο φάσμα από επιλογές ανάλυσης προγράμματος. Επιλέξαμε να χρησιμοποιήσουμε το perf, επειδή ήταν εργαλείο που είχαμε κάποια εμπειρία μαζί του και επίσης θεωρείτε εύκολη η χρήση του. Εδώ, πρέπει να τονίσουμε πως υπάρχουν πιο εξειδικευμένα εργαλεία ανάλυσης προγράμματος όπως είναι το pcm [14] και το likwid [15], τα οποία δεν τα χρησιμοποιήσουμε αλλά είναι καλή αναφορά για πιθανή μελλοντική δουλεία.

Το Processor Counter Monitor (PCM) είναι μια διεπαφή προγραμματισμού εφαρμογών (API) που περιέχει ένα σύνολο εργαλείων που βασίζονται στο API για την πραγματοποίηση μετρήσεων απόδοσης και ενέργειας των επεξεργαστών Intel® Core™, Xeon®, Atom™ και Xeon Phi™. Το Likwid είναι ένα απλό στην εγκατάσταση και χρήση toolsuite που εκτελείτε από την γραμμής εντολών και προσφέρεται για προγραμματιστές προσανατολισμένους στις μετρήσεις απόδοσης. Το εργαλείο υποστηρίζει μετρήσεις για επεξεργαστές Intel, AMD, ARMv8 και POWER9 στο λειτουργικό σύστημα Linux. Υπάρχει πρόσθετη υποστήριξη για τις GPU της Nvidia.

Από το perf χρησιμοποιήσαμε τρεις από τις βασικές λειτουργίες που παρέχει. Συγκεκριμένα, χρησιμοποιήσαμε τις λειτουργίες perf stat, record, και report/annotate. Η λειτουργία stat δίνει την δυνατότητα να μετρήσεις performance counters που παρέχονται από το σύστημα. Υπάρχει ένα τεράστιο εύρος από performance counters που μπορούν να χρησιμοποιηθούν μαζί με το stat (περισσότερες από 7,000 επιλογές) που έχουν να κάνουν με hardware, software και kernel events. Στην δική μας περίπτωση χρησιμοποιήσαμε τα ακόλουθα events: instructions, cycles, LLC-load-misses, LLC-store-misses, LLC-loads, LLC-stores, cycle_activity.stalls_l3_miss. Για την ακρίβεια χρειαζόμαστε τα instructions και τα cycles του επεξεργαστή για να μπορούμε να υπολογίσουμε τον χρόνο που παίρνει ο επεξεργαστής, ακολούθως θα θέλαμε να ξέρουμε πόσες φορές η πρόσβαση προς την κύρια μνήμη δηλαδή στην περίπτωση μας στην PMEM. Βασιζόμαστε στο πόσες φορές θα έχουμε miss στην Last Level Cache (LLC) και για loads και για stores για να υπολογίσουμε αυτό, και επίσης χρησιμοποιούμε το event stalls_l3_stalls το οποίο μας δίνει συνολικά τους κύκλους μηχανής που σπαταλιόνται για όλα τα miss της LLC. Τις λειτουργείες perf record και report/annotate τις χρησιμοποιήσαμε κυρίως για δικό μας σκοπό, δηλαδή να μελετήσουμε με ακρίβεια τα νούμερα από τους counters μας, ακόμα και τον κώδικα του κάθε symbol (functions, libraries, κτλ...) που αναφέρατε στα αποτελέσματα του perf. Το perf record χρησιμοποιείται όπως και το stat με την διαφορά ότι τα αποτελέσματα γράφονται σε αρχείο με την δυνατότητα ανάλυσης τους από το perf report/annotate.

Στην πρώτη μας προσπάθεια για να χρησιμοποιήσουμε την DRAM για να κάνουμε τις μετρήσεις μας, χρησιμοποιήσαμε το /dev/shm του Linux. Το /dev/shm δεν είναι κάτι άλλο παρά μια κοινή μνήμη στην οποία μπορούν διαφορετικές διεργασίες να ανταλλάζουν δεδομένα χωρίς να πρέπει να γράψουν ή να διαβάσουν στον ή από τον δίσκο. Το /dev/shm συνήθως χρησιμοποιείται για να βελτιώσει την απόδοση προγραμμάτων αλλά και σε βαριά φορτωμένα συστήματα. Το /dev/shm εμφανίζεται στο σύστημα σαν ένα προσωρινό σύστημα αρχείων (temporary file system) και αυτό είναι ο λόγος που δεν ήταν η κατάλληλη επιλογή για εμάς. Συγκεκριμένα, αφού κάναμε τις μετρήσεις μας, με την βοήθεια του perf report παρατηρούσαμε αρκετά overheads σε κύκλους και instructions που ήταν kernel activity. Ουσιαστικά, επειδή χρησιμοποιούμε ένα tmpfs πρέπει να γίνονται και τα κατάλληλα synchronizations οπότε υπάρχει

επιπλέον kernel activity. Επιπλέον θέλαμε να έχουμε και το ίδιο σύστημα αρχείων με αυτό που χρησιμοποιούν τα PMEM devices μας συγκεκριμένα το xfs, για να είναι οι μετρήσεις μας όσο πιο κοντά στην λειτουργία της PMEM και ουσιαστικά να μετρούμε και να μοντελοποιούμε το ίδιο πράγμα στο τέλος. Για να λύσουμε αυτό το πρόβλημα δημιουργήσαμε ένα DRAM emulated device[12]. Το DRAM emulated device ουσιαστικά είναι ένα device που συμπεριφέρεται σαν PMEM αλλά χρησιμοποιεί την DRAM συνεπώς έχει και την απόδοση της. Ο τρόπος με τον οποίο το πετύχαμε αυτό είναι:

1. Βάζουμε στο αρχείο grub το ακόλουθο GRUB_CMDLINE_LINUX="memmap=nn[KMG]!ss[KMG]", όπου nn είναι μέγεθος που θέλουμε να έχει το device, και το ss είναι από που θα ξεκινά στην μνήμη για να δεσμεύσει τα nn bytes.
2. Κάνουμε update το grub file
3. Δημιουργούμε ένα κατάλογο κάτω από το /mnt (πχ /mnt/emdram)
4. mkfs.xfs /dev/pmem4 (το pmem3 είναι το device που δημιουργείται μετα το update του grub)
5. mount -o dax /dev/pmem4 /mnt/emdram

Όπως αναφέραμε και πιο πριν, δεν ήταν σκοπός μας να πάρουμε μετρήσεις για το συνολικό benchmark που εκτελούσαμε, αλλά μόνο για το κομμάτι της λειτουργίας που μελετούσαμε. Το perf δεν παρέχει κάποιο ξεκάθαρο τρόπο για να το πετύχεις αυτό, δηλαδή να κάνες μετρήσεις μόνο για ένα κομμάτι του κώδικα. Έτσι λοιπόν, ο τρόπος που το επιλέξαμε να το καταφέρουμε αυτό ήταν, με το να επικολλήσουμε το perf στην εκτέλεση του benchmark μετά από κάποιο “σήμα”. Μπορείς να επικολλήσεις ένα perf stat σε ένα υπό εκτέλεση πρόγραμμα με την παράμετρο -p <pid> στην εντολή perf stat… ακολουθούμενη από το pid του. Όσον αφορά το “σήμα” που θα στέλναμε για να δηλώσουμε ότι θέλουμε να επικολληθεί, είναι ένα μήνυμα στην οθόνη (printf) μαζί με flush του stdout, το οποίο θα περιμέναμε σε ένα while. Επίσης, βάλαμε το πρόγραμμα να κοιμηθεί για 5 δευτερόλεπτα για να είμαστε σίγουροι ότι το perf θα προλάβει να επικολληθεί πριν να αρχίσει ο κώδικας που θα θάλαμε, χρησιμοποιήσαμε την συνάρτηση sleep η οποία δεν επηρεάζει τις μετρήσεις του perf. Στην Εικόνα 5.3 φαίνεται η υλοποίηση αυτής της ιδέας που περιγράψαμε.

```
//initialization
if ((ret = pmembench_init_workers(workers, bench, args)) != 0) {
        goto out;
    }

//signal to attach perf
printf("begin sleeping for 5s\n");
fflush(stdout);
sleep(5);

//code we want to measure (actual get operation)
unsigned j;
for (j = 0; j < args->n_threads; j++) {
     benchmark_worker_run(workers[j]);
}

for (j = 0; j < args->n_threads; j++) {
   benchmark_worker_join(workers[j]);
   if (workers[j]->ret != 0) {
       ret = workers[j]->ret;
       fprintf(stderr, "thread number %u failed\n", j);
   }

results_store(res, workers, args->n_threads, args->n_ops_per_thread);



#bash script to catch signal
function attach_perf(){
    read -t 0.01 output;
    while [[ $output != *"begin sleeping for 5s"* ]]; do
        #read system stdout every 0.01 sec
        read -t 0.01 output;
    done

    echo "attach perf here";
    #pid is file that contains programm's pid
    pid=$(cat pid);
    perf stat -e instructions,cycles,LLC-load-misses,LLC-store-misses,LLC-
loads,LLC-stores,cycle_activity.stalls_l3_miss -p $pid
}
```

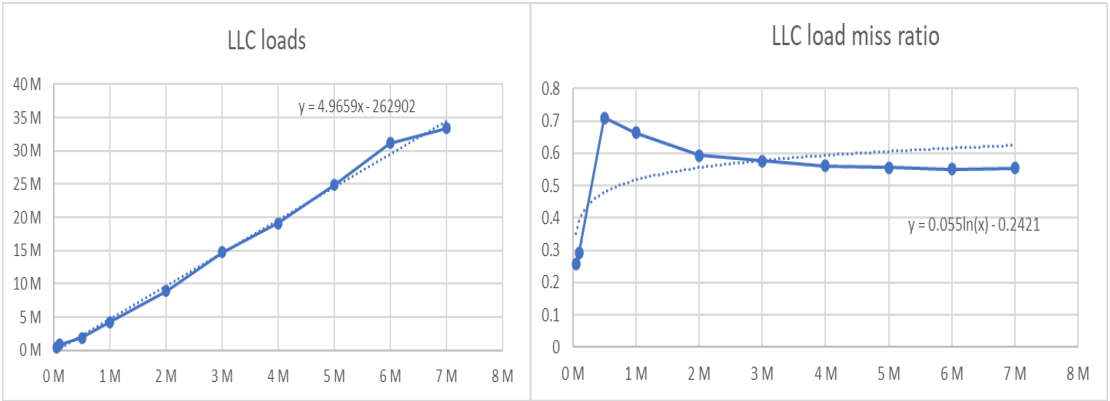
Εικόνα 5.3 Επικόλληση perf σε υπό εκτέλεση πρόγραμμα

Εδώ να αναφέρουμε ότι μια “πιο σωστή” προσέγγιση στην επίλυση του πιο πάνω προβλήματος ήταν η χρήση της βιβλιοθήκης perf_even_open[17] . Η βιβλιοθήκη αυτή σου δίνει την δυνατότητα να χρησιμοποιήσεις του performance counters του perf ρητά με την χρήση της γλώσσας C. Με αυτό τον τρόπο θα μπορούσαμε να πάρουμε της μετρήσεις που θέλουμε για ακριβώς το σημείο του κώδικα που θέλαμε. Λόγο χρόνο δεν είχαμε την ευκαιρία να εξοικειωθούμε και να χρησιμοποιήσουμε την βιβλιοθήκη.

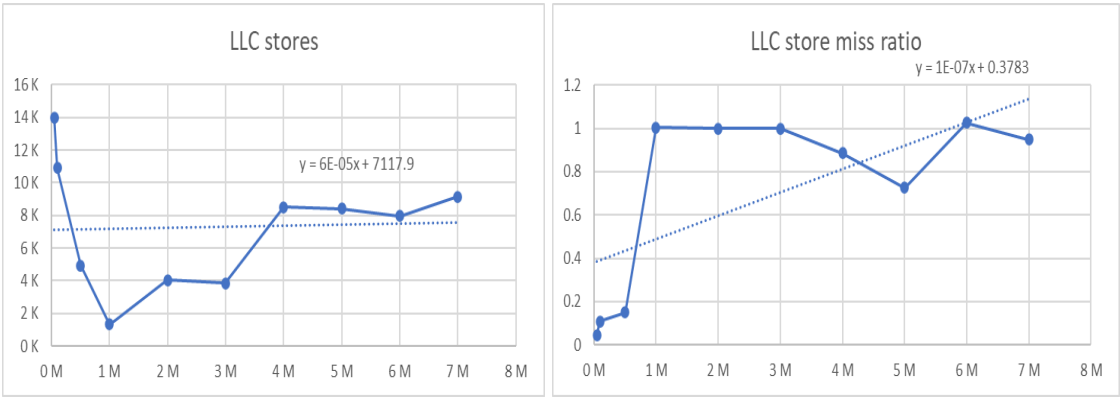
Για να ολοκληρώσουμε την συλλογή δεδομένων, εκτελέσαμε το benchmark με σταθερή λειτουργία την map_get, και σταθερό μέγεθος στοιχείων 1 Byte, αλλά μεταβλητό αριθμό από operations. Οι τιμές που χρησιμοποιήσαμε ήταν 50000, 100000, 500000, 1000000, 2000000. 3000000, 4000000, 5000000, 6000000, 7000000. Και επικολλούσαμε το perf stat όπως αναφέραμε πιο πάνω με τους performance counter που επίσης αναφέραμε.

5.3.2. Δημιουργία μοντέλου

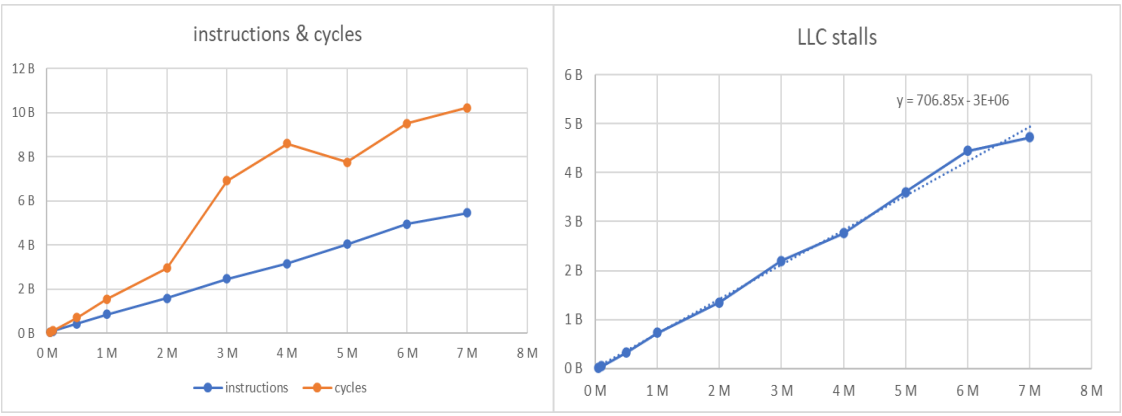
Αφού καταγράψαμε τις μετρήσεις μας, και είχαμε και τα αποτελέσματα του χαρακτηρισμού του υλικού μας από το LENS μπορούσαμε να δημιουργήσουμε το μοντέλο μας. Στις Εικόνες 5.4, 5.5 και 5.6 φαίνονται οι γραφικές παραστάσεις από τα αποτελέσματα των εκτελέσεων μας, για τους αντίστοιχους performance counters. Σημαντικό στις επόμενες γραφικές είναι η διακεκομμένη μπλε γραμμή που υποδηλώνει το curve fitting της αντίστοιχης γραφικής (εξηγούμε στην επόμενη παράγραφο).



Εικόνα 5.4 LLC load



Εικόνα 5.5 LLC store



Εικόνα 5.6 Total instructions/cycles, and LLC stalls

Η τεχνική που χρησιμοποιήσαμε για να καταφέρουμε να μεταφράσουμε τα δεδομένα μας σε εξισώσεις για να τα βάλουμε στο αναλυτικό μας μοντέλο ήταν η τεχνική curve fitting[16]. Ουσιαστικά, το curve fitting είναι η μέθοδος με την οποία προσπαθείς να υπολογίσεις με όσο μεγαλύτερη ακρίβεια είναι δυνατόν τους συντελεστές μια γνωστής μαθηματικής εξίσωσης όπως την γραμμική, πολυωνυμική, gaussian, κτλ.. με σκοπό να "ταιριάζει" πιο πολύ σε μια σειρά από δεδομένα. Στην δική μας περίπτωση, χρησιμοποιήσαμε τα trendlines της Microsoft Excel, που χρησιμοποιούν την μέθοδο αυτή. Αφού είχαμε δημιουργήσει τις γραφικές παραστάσεις και ξέραμε την μορφή τους, επιλέξαμε την εξίσωση που είχε την πιο παρόμοια μορφή περισσότερα στα δεδομένα μας, και πήραμε την τελική εξίσωση που μας υπολογιστικέ. Οι εξισώσεις όπως και τα trendlines φαίνονται στις εικόνες πιο πάνω. Επίσης, αφού τα πειράματα εκτελέστηκαν με μεταβλητό αριθμό από operations τότε αυτή είναι η ανεξάρτητη μεταβλητή τις εξίσωσης μας.

Στη συνέχεια, αφού είχαμε πλέον τρόπο να υπολογίζουμε τις τιμές για τους performance counters που χρησιμοποιούμε, απόμενε να υπολογίσουμε την καθυστέρηση της μνήμης με βάσει αυτούς. Η ιδέα είναι αρκετά απλή, αφού έχουμε τα LLC references και το LLC miss ratio, τα πολλαπλασιάζουμε με την αντίστοιχή καθυστέρηση για προσπέλαση και εγγραφή που ξέρουμε για την PMEM μας, υπολογίζοντας έτσι την συνολική καθυστέρηση που έχουμε για στο σύστημα μνήμης. Επίσης, προσθέτουμε στην καθυστέρηση τον υποθετικό χρόνο που χρειάζονται τα δεδομένα να διαβαστούν ή να εγγραφτούν με βάσει τον εύρος ζώνης και το μέγεθος τους. Με το όρο υποθετικό, εννοούμε ότι δεν υπολογίζουμε το χρόνο αυτό βάσει κάποιας μέτρησης αλλά υποθέτοντας ότι αυτός ο χρόνος είναι ο χρόνος που χρειάζεται το σύστημα μνήμης να στείλει όλα τα δεδομένα size*n την ίδια στιγμή. Στην Εικόνα 5.7 φαίνεται η υλοποίηση του αναλυτικού μοντέλου μας σε γλώσσα προγραμματισμού python. Οι είσοδοι του προγράμματος είναι ο αριθμός των operations «count» και το μέγεθος των στοιχείων «size». Το μοντέλο μπορεί να χρησιμοποιηθεί με την εντολή:

```
python performance_counters_model.py <n_of_operation> <size>
```

$$\boldsymbol{cpu_{fq} = 2.1 * 10^9}$$

$$\boldsymbol{pmem_{read\_bw} = 7.45 * 10^9}$$

$$\boldsymbol{pmem_{latency\_r} = 305 * 10^{-9}}$$

$$\boldsymbol{pmem_{latency\_w} = 90 * 10^{-9}}$$

$$\boldsymbol{pmem_{read\_bw} = 7.45 * 10^9}$$

$$\boldsymbol{instructions = 854 * count}$$

$$\boldsymbol{cycles = 1589 * count}$$

$$\boldsymbol{ipc = instuctions/cycles}$$

$$\boldsymbol{cpu_{time} = \frac{cycles}{cpu_{fq}} * ipc}$$

$$\boldsymbol{LLC_{loads} = |4.96 * count - 262902|}$$

$$\boldsymbol{LLC_{stores} = |6 * 10^{-5} * count - 7117|}$$

$$\boldsymbol{LLC_{load\_miss\_ratio} = |0.055 * ln(count) - 0.24|}$$

$$LLC_{store\_miss\_ratio} = |4 * 10^{-8} * count + 0.81|$$

$$LLC_{stalls} = |706 * count - 3 * 10\text{^}6|$$

$$PMEM_{latency} = \left(LLC_{loads} * LLC_{load_{miss_{ratio}}} * pmem_{latency_r}\right) + \left(LLC_{stores} * LLC_{store_{miss_{ratio}}} * pmem_{latency_w}\right) + \left(count * \frac{size}{pmem_{read_{bw}}}\right)$$

Εικόνα 5.7 Μοντέλο "Performance counters"

5.4. Μοντέλο "Hashmap based"

Σε αυτή την προσπάθεια μας εστιάζουμε στη υλοποίηση του προγράμματος που εξετάζουμε και γενικά σε μια πιο θεωρητική προσέγγιση από την προηγούμενη. Αναλύουμε το τι κάνει το πρόγραμμα έτσι ώστε να μπορούμε να προβλέψουμε τις προσπελάσεις στην μνήμη που θα προκύψουν για υπολογίσουμε την καθυστέρηση και κατ' επέκταση τον χρόνο εκτέλεσης.

5.4.1. Ανάλυση hashmap

Για να αναπτύξουμε αυτό το μοντέλο χρειαζόμασταν μια βαθιά κατανόηση τις υλοποίησης του προγράμματος που μελετούσαμε. Επίσης, έπρεπε να μελετήσουμε και να καταλάβουμε την δομή του hashmap όπως είναι υλοποιημένο στην βιβλιοθήκη που χρησιμοποιούμε. Την δομή του και τα αντικείμενα που αποτελούν το hashmap το χρειαστήκαμε για να έχουμε καλύτερη αντίληψη του κώδικα καθώς και πως αυτός θα αναμέναμε να αλληλοεπιδράσει με το σύστημα μνήμης.

Αρχικά, για να αρχίσουμε να κατανοούμε το hashmap έπρεπε να μελετήσουμε το πως υλοποιείται η μορφή του. Όπως αναφέραμε και στην ενότητα Hashmap στο libpmemobj, και φαίνεται στην **Εικόνα 4.1.2** η μορφή του hashmap περιέχει μια λίστα στην οποία κάθε είσοδος (entry) περιέχει ένα κουβά (bucket), επίσης κρατάμε τον αριθμό συνολικά από τα στοιχεία που περιέχει μέχρι εκείνη την στιγμή το hashmap. Ένας κουβάς αποτελείτε από μια συνδεδεμένη λίστα της οποίας κάθε είσοδος της είναι ένα στοιχείο της μορφής <key,value> και εδώ κρατάμε ένα ακέραιο που είναι ο αριθμός

από τους κουβάδες. Τα στοιχεία με την σειρά τους έχουν το κλασικό κλειδί (key), πάνω στο οποίο υπολογίζεται το hashing δηλαδή σε πιο κουβά θα μπουν. Από την άλλη η τιμή (value) των στοιχείων δεν είναι η κλασική τιμή αλλά ένας δείκτης σε απευθείας persistent memory με την βοήθεια των ειδικών δεικτών της βιβλιοθήκης. Όπως είναι αναμενόμενο ο hashmap κτίζεται δυναμικά, με αποτέλεσμα να πρέπει να γίνεται rebuild η δομή, αλλά δεν ήταν στα στοιχεία που μελετήσαμε σε αυτήν την εργασία.

Αφού είχαμε μια κατανόηση για την μορφή του hashmap, μπορούσαμε να μελετήσουμε το get operation που ήταν ο στόχος μας. Στην Εικόνα 5.8 βλέπουμε την λειτουργία get του hashmap_atomic την οποία χρησιμοποιούσαμε για τα πειράματα μας. Αρχικά, η συνάρτηση αυτή μπορούμε να δούμε ότι δέχεται ένα struct τύπου hashmap_atomic όπως το περιγράψαμε στην προηγούμενη ενότητα που είναι ουσιαστικά η δομή δεδομένων μας, επίσης η συνάρτηση δέχεται και το κλειδί του στοιχείου που θα ψάξει για αυτό εάν υπάρχει και τέλος δέχεται ένα δείκτη σε δομή PMEMobjpool που ουσιαστικά είναι ένα “pool” από αντικείμενα PMEMobj αλλά επειδή δεν κάνουμε χρήση από PMEMpools στο δικό μας πείραμα μπορούμε να το αγνοήσουμε. Στο σώμα της συνάρτησης, βλέπουμε ότι η πρώτη ενέργεια που γίνονται είναι να πάρουμε την λίστα που αποτελεί τους κουβάδες που ουσιαστικά είναι και όλο το hashmap. Στην συνέχεια, γίνεται το hash με τον universal hashing τρόπο [11] έτσι ώστε να βρούμε σε πιο κουβά πρέπει να ψάξουμε. Τέλος, με την χρήση ενός macro γίνεται ένας σειριακός έλεγχος μέσα στον κουβά για να βρεθεί εάν υπάρχει τελικά το στοιχείο στο hashmap ή όχι.

```
/*
 * hm_atomic_get -- checks whether specified value is in the hashmap
 */
PMEMoid
hm_atomic_get(PMEMobjpool *pop, TOID(struct hashmap_atomic) hashmap,
        uint64_t key)
{
    TOID(struct buckets) buckets = D_RO(hashmap)->buckets;
    TOID(struct entry) var;

    uint64_t h = hash(&hashmap, &buckets, key);

    POBJ_LIST_FOREACH(var, &D_RO(buckets)->bucket[h], list)
        if (D_RO(var)->key == key)
            return D_RO(var)->value;

    return OID_NULL;
}
```

Εικόνα 5.8 Λειτουργία get

5.4.2. Δημιουργία μοντέλου

Για να δημιουργήσουμε το μοντέλο μας έπρεπε να πάμε ακόμη ένα βήμα βαθύτερα στην κατανόηση του κώδικα που μελετούσαμε. Δηλαδή, έπρεπε να είμαστε θε θέση να πούμε ποιες από τις εντολές όντως κάνουν προσβάσεις στην PMEM, για να μπορούμε να τις μετρήσουμε.

Στον κώδικα της Εικόνας 5.8 γίνεται χρήση του macro D_RO όπου υπάρχει η ανάγκη για την ανάκτηση τιμής, η υλοποίηση του macro αυτού φαίνεται στην Εικόνα 5.9. Στην ουσία αυτή η μακροεντολή χρησιμοποιεί την συνάρτηση pmemobj_direct_inline που είναι η συνάρτηση η οποία επιστρέφει τον απευθείας δείκτη σε PMEM που έχει το αντικείμενο που περνιέται σαν παράμετρος. Οπότε, η μακροεντολή D_RO κάνει πρόσβαση σε μνήμη PMEM δηλαδή έχει και την καθυστέρηση της, την οποία ξέρουμε. Επίσης δυο από τις 4 εντολές D_RO είναι μέσα σε ένα loop, αλλά επειδή ξέρουμε ότι είναι δυναμικό το μέγεθος αυτής τη επανάληψης και δεν υπάρχει τρόπος να υπολογιστεί θεωρούμε ότι το loop είναι O(1), δηλαδή θα υπολογίσουμε μόνο μια πρόσβαση μνήμης.

Στην Εικόνα 5.10, φαίνεται το μοντέλο και οι εξισώσεις που προκύπτουν εάν προσθέσουμε τις προσβάσεις μνήμης που υπολογίσαμε βάσει του κώδικα. Το count είναι ο αριθμός των operations και το size το μέγεθος των στοιχείων. Το μοντέλο μπορεί να χρησιμοποιηθεί με την εντολή:

```
python hashmap_model.py <n_of_operations> <size>
```

```
#define DIRECT_RO(o) \
    (reinterpret_cast < const __typeof__((o)._type) > \
    (pmemobj_direct((o).oid)))

#define D_RO    DIRECT_RO


/*
 * Returns the direct pointer of an object.
 */
static inline void *
pmemobj_direct_inline(PMEMoid oid)
{
    if (oid.off == 0 || oid.pool_uuid_lo == 0)
        return NULL;

    struct _pobj_pcache *cache = &_pobj_cached_pool;
    if (_pobj_cache_invalidate != cache->invalidate ||
            cache->uuid_lo != oid.pool_uuid_lo) {
        cache->invalidate = _pobj_cache_invalidate;

        if (!(cache->pop = pmemobj_pool_by_oid(oid))) {
            cache->uuid_lo = 0;
            return NULL;
        }

        cache->uuid_lo = oid.pool_uuid_lo;
    }

    return (void *)((uintptr_t)cache->pop + oid.off);
}
```

Εικόνα 5.9 Απευθείας πρόσβαση σε διευθύνσεις PMEM

$$pmem_{read\_bw} = 7.45 * 10^{9}$$

$$pmem_{latency\_r} = 305 * 10^{-9}$$

$$pmem_{latency\_w} = 90 * 10^{-9}$$

$$pmem_{read\_bw} = 7.45 * 10^{9}$$

get hashmap's buckets

$$PMEM_{latency} = count * pmem_{latency_r}$$

hashing function time

$$PMEM_{latency} \mathrel{+}= count * pmem_{latency_r}$$

list the hashmap's buckets (POBJ_LIST_FOREACH)

$$PMEM_{latency} \mathrel{+}= count * pmem_{latency_r}$$

compare the value with the key

$$PMEM_{latency} \mathrel{+}= count * pmem_{latency_r}/3$$

hashmap retrieve data

$$PMEM_{latency} \mathrel{+}= count * (pmem_{latency_r} + \frac{size}{pmem_{bw}})$$

Εικόνα 5.10 Μοντέλο Hashmap based

Κεφάλαιο 6

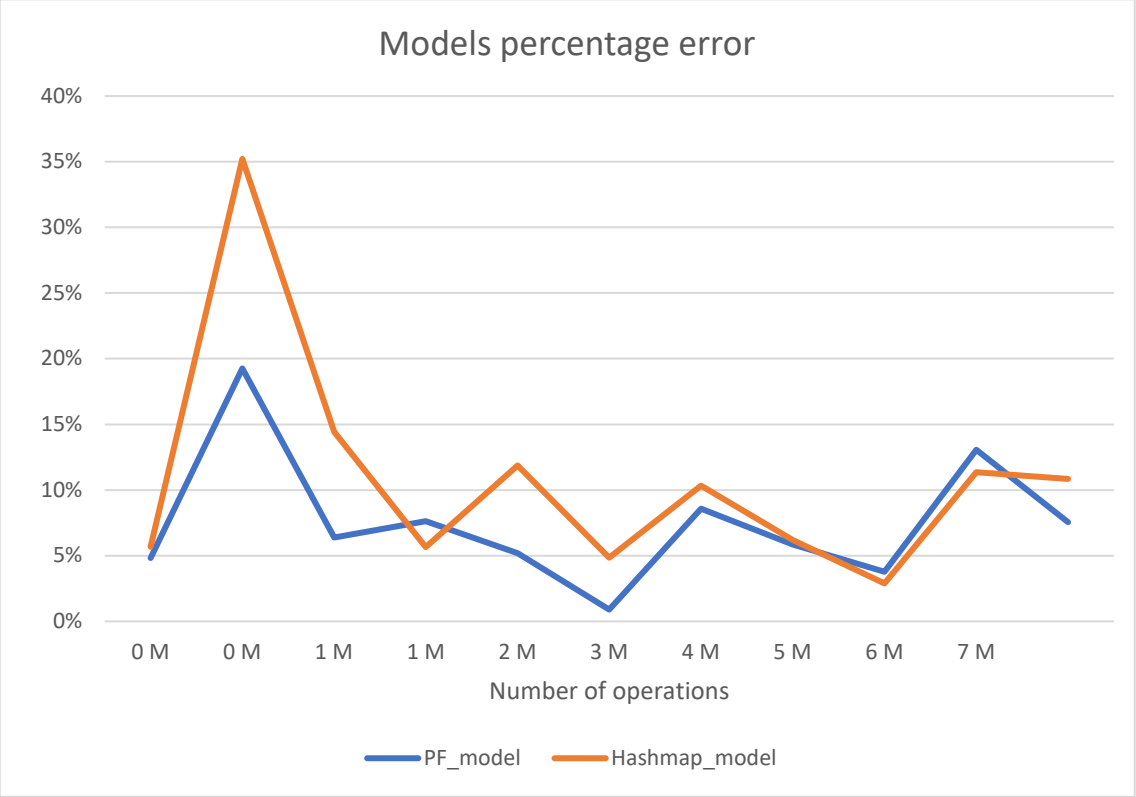
Σύγκριση μοντέλων και παρατηρήσεις

6.1. Αποτελέσματα μοντέλων

Ολοκληρώνοντας την δημιουργία των μοντέλων μας τα χρησιμοποιήσαμε με τις ίδιες παραμέτρους με αυτές που είχαμε χρησιμοποιήσαμε στα πειράματα μας με την πραγματική PMEM. Στον Πίνακα 6.1 φαίνονται τα αποτελέσματα μας μαζί με τις τιμές από την PMEM που μετρήσαμε με το benchmark. Στην Εικόνα 6.1 φαίνεται το ποσοστιαίο σφάλμα ανάμεσα στο αποτέλεσμα του κάθε μοντέλου με την πραγματική τιμή. Το μοντέλο performance counters έχει μέσο όρο σφάλματος 7.5%, ενώ το hashmap based 10.8%.

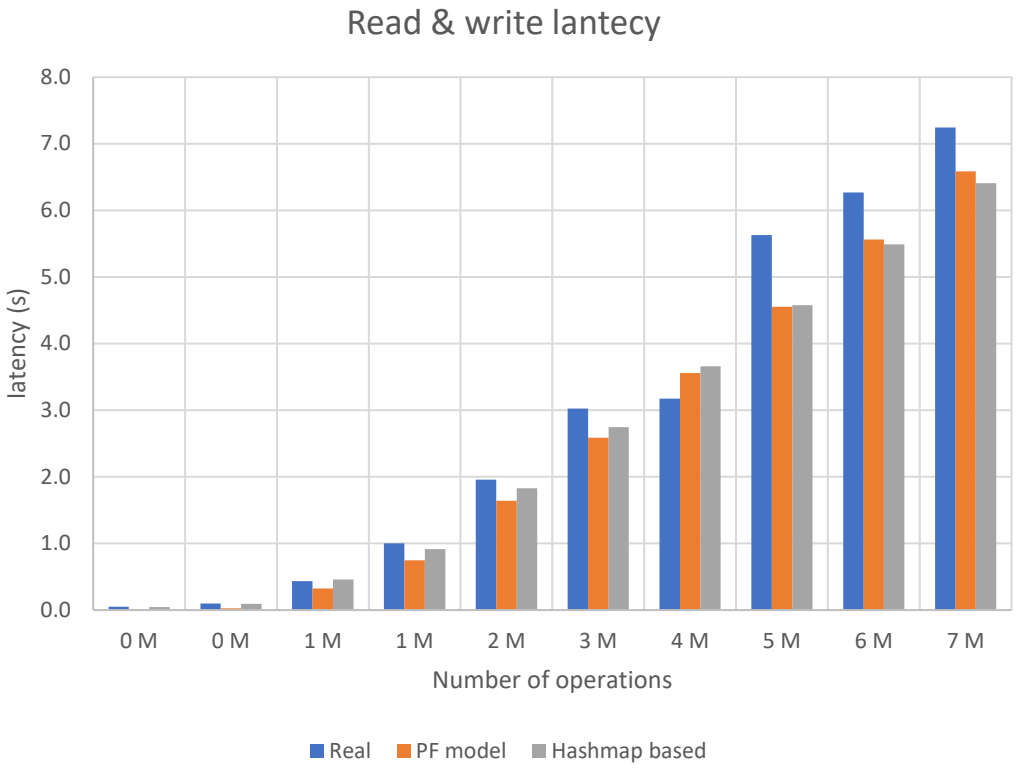
n	Real pmem (s)	PF model (s)	Hashmap model (s)
50,000	0.034	0.032	0.036
100,000	0.086	0.069	0.132
500,000	0.566	0.529	0.661
1,000,000	1.247	1.152	1.322
2,000,000	2.330	2.457	2.644
3,000,000	3.773	3.807	3.965
4,000,000	4.741	5.186	5.287
5,000,000	6.201	6.586	6.609
6,000,000	7.702	8.003	7.931
7,000,000	8.202	9.435	9.253

Πίνακας 6.1 Μοντέλα και πραγματική επίδοση



Εικόνα 6.1 Εκατοστιαίο σφάλμα μοντέλων

Παρατηρούμε ότι το μοντέλο Performance counters προβλέπει καλύτερη την απόδοσης της PMEM παρά το μοντέλο Hashmap based. Αυτό μπορούμε να πούμε ότι είναι λογικό αφού το πρώτο μοντέλο βασίζεται σε πραγματικές μετρήσεις του συστήματος αντιθέτως με το δεύτερο που βασίζεται σε πιο θεωρητική όψη των πραγμάτων.



Εικόνα 6.2 Καθυστέρηση για τα μοντέλα και την πραγματική PMEM

Με το πέρας των πειραμάτων μας είχαμε και κάποια μη αναμενόμενα αποτελέσματα. Αρχικά κάνοντας χρήση του performance counter l3_memory_stalls παρατηρήσαμε ότι κύκλοι πηγαίναν για να περιμένουμε το σύστημα μνήμης είχαν διαφορά από την καθυστέρηση που υπολογίζαμε εμείς με τα μοντέλα μας. Αυτό λοιπόν είναι μια ένδειξη ότι υπάρχει παραλληλία μέσα στο σύστημα μνήμης. Μια άλλη παρατήρηση που δεν περιμέναμε είναι ότι η αρχικοποίηση του hashmap είχε μεγάλη διαφορά μεταξύ DRAM και PMEM παρόλο που η καθυστέρηση εγγραφής των δυο συστημάτων μνήμης είναι αρκετά κοντά (88 και 90 σύμφωνα με τις μετρήσεις μας). Και τέλος ένα άλλο αποτέλεσμα που ήταν απροσδόκητο ήταν το ότι δεν είχαμε επιπλέον αύξηση της καθυστέρησης, στο latency που μετρούσε το benchmark όταν εμείς τρέχαμε το πείραμα με μεγαλύτερα μέγεθοι στα στοιχεία του hashmap.

6.2. Διαφορές μοντέλων

Τα δύο μοντέλα έχουν αρκετές διάφορες, κυρίως όσον αφορά την αξιοποίηση τους. Αρχικά, όπως είναι λογικό αφού έχουμε διαφορετικές προσεγγίσεις στην δημιουργία των μοντέλων θα έχουμε και διαφορετικά δυνατά και αδύνατα σημεία. Το μοντέλο Performance counters αφού βασίζεται σε μετρήσεις τους συστήματος, μπορούμε να πούμε ότι θα είναι σε θέση να προβλέψει και απόδοση για παράδειγμα μιας άλλης δομής δεδομένων εχτός από το hashmap. Αλλά, αν χρησιμοποιήσουμε το ίδιο μοντέλο σε ένα διαφορετικό σύστημα, ή αν αλλάξουμε την τεχνολογία DRAM με κάποιο τρόπο έτσι ώστε να λειτουργεί διαφορετικά, για παράδειγμα την ιδιαιτερότητα που έχει το /dev/shm που αναφέραμε στην ενότητα Συλλογή δεδομένων θα επηρεαστεί η ακρίβεια του μοντέλου μας. Από την άλλη μεριά, το μοντέλο Hashmap based δεν βασίζεται στο υλικό οπότε μπορεί να χρησιμοποιηθεί σε οποιοδήποτε σύστημα με την ίδια ακρίβεια. Βέβαια αυτό το μοντέλο δεν μπορεί να χρησιμοποιηθεί για άλλη δομή δεδομένων αφού όπως ξέρουμε βασίζεται εξ ολοκλήρου στην υλοποίηση της δομής hashmap.

Επομένως, αν κάποιος θέλει να χρησιμοποιήσει τα μοντέλα μας αν ξέρει ότι θα έχει το ίδιο σύστημα για τον επόμενο καιρό μπορεί να βασιστεί στο μοντέλο με τους performance counters αλλιώς θα ήταν καλυτέρα να βασιστεί στο αντίστοιχο μοντέλο που κάνει χρήση του hashmap.

6.3. Αδυναμίες μοντέλων

6.3.1. Bandwidth

Τα μοντέλα μας όπως αναφέραμε ξανά έχουν αρκετές αδυναμίες και μπορούν να βελτιωθούν αρκετά. Ένα πρώτο σημαντικό πρόβλημα είναι η αδυναμία των μοντέλων μας να προσομοιωνόσουν αρκούντος καλά το εύρος ζώνης της PMEM. Ουσιαστικά, δεν κάναμε κάποια αναλυτική μελέτη για το πως το σύστημα χρησιμοποιεί το εύρος ζώνης, σε διαφορές καταστάσεις που μπορεί να βρεθεί. Αυτό συνεπάγεται ότι το μοντέλα μας δεν έχουν την δυνατότητα να συμπεριλάβουν στο αποτέλεσμα μας τυχόν παραλληλία που υπάρχει λόγο του εύρος ζώνης που υπάρχει στο σύστημα μνήμης.

6.3.2. Παραλληλία συστήματος μνήμης

Μια δεύτερη αδυναμία που υπάρχει στα μοντέλα μπορούμε να πούμε ότι είναι το γεγονός ότι δεν λαμβάνουν υπόψη κάποια παραλληλία που μπορεί να υπάρχει στο σύστημα μνήμης. Και τα δύο μοντέλα υπολογίζουν το αποτέλεσμα τους υποθέτοντας ότι τα δεδομένα που έχουν εκτελούνται σειριακά, αφού δεν έχουμε κάτι που να βάζει τον παράγοντα παραλληλία μέσα στις εξισώσεις μας. Ξέρουμε ότι υπάρχει αυτή η παραλληλία στο πραγματικό σύστημα μνήμης παρατηρώντας τα αποτελέσματα που έχουμε.

6.3.3. Προσβάσεις κύριας μνήμης

Ένα αρκετά σημαντικό πρόβλημα μετά μοντέλα μας είναι ότι δεν ξέρουμε με σιγουριά αν όλες οι προσβάσεις που έχουν στην κύρια μνήμη πηγαίνουν στην PMEM και όχι στην DRAM. Στα μοντέλα μας υποθέτουμε ότι όλα τα LLC misses στο “Performance counters” μοντέλο και γενικά όλες οι προσβάσεις στο “Hashmap based” μοντέλο πηγαίνουν προς την PMEM, δηλαδή χρησιμοποιούμε την καθυστέρηση της PMEM. Αυτό δεν ξέρουμε σίγουρα αν ισχύει στην πραγματικότητα πάρα το γεγονός ότι χρησιμοποιούμε το αρχείο που γράφονται τα αποτελέσματα μας να είναι στην PMEM.

Κεφάλαιο 7

Σχετική δουλεία

7.1. Σχετική δουλεία

Μια δουλεία που είναι αρκετά παρόμοια με την δική μας, αλλά και που μας βοήθησε αρκετά είναι το άρθρο “Characterizing and Modeling Non-Volatile Memory Systems”[1]. Μπορούμε να πούμε ότι είναι αρκετά πιο ολοκληρωμένη δουλεία αφού υλοποιήθηκαν και ο χαρακτηρισμός του υλικού (characterization) και η μοντελοποίηση της μη πτητικής μνήμης. Επίσης το LENS που χρησιμοποιούμε στην δική μας εργασία όπως ξανά αναφέραμε υλοποιήθηκε στα πλαίσια αυτού του άρθρου. Η κύρια διαφορά του άρθρου με την δική μας δουλεία είναι ο τρόπος υλοποίησης της μοντελοποίησης της μη πτητικής μνήμης. Συγκεκριμένα στο άρθρο γίνεται κυρίως ανάλυση κάποιων buffers για να επιτευχθεί η μοντελοποίηση. Ένα άλλο άρθρο που είναι παρόμοια δουλεία με την δική μας είναι το “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory”[5]. Σε αυτή την περίπτωση, χρησιμοποιήσαμε την δουλειά αυτή για να κατανοήσουμε και καλυτέρα την χρήση και την συμπεριφορά της μη πτητικής μνήμης.

Κεφάλαιο 8

Συμπεράσματα και Μελλοντική δουλεία

8.1. Συμπεράσματα

Στόχος αυτής της εργασίας ήταν να μοντελοποιήσουμε την απόδοση μη πτητικών συστημάτων κύριας μνήμης, και όπως είδαμε μέσα από την δουλεία αυτή είναι ένα ανοικτό και πολύ ενδιαφέρον θέμα. Επίσης, είδαμε ότι υπάρχουν αρκετά εμπόδια σε αυτή την προσπάθεια στο να φτάσουμε στο στόχο μας. Μερικά από τα προβλήματα που είχαμε τα λύσαμε, άλλα δεν καταφέραμε ή δεν είχαμε τον χρόνο να τα αντιμετωπίσουμε οπότε έμειναν ανοιχτά για μελλοντική δουλεία.

Η εργασίας μας μπορούμε να πούμε ότι χωριστικέ σε τρία στάδια. Το πρώτο στάδιο ήταν ο χαρακτηρισμός του υλικού που είχαμε στην διάθεση μας δηλαδή των Linux server shasta.in.cs.ucy.ac.cy. Το characterization ήταν το πιο ξεκάθαρο κομμάτι της εργασίας αυτής μπορούμε να πούμε επειδή δεν χρειάστηκε να υλοποιήσουμε δικό μας profiler αλλά χρησιμοποιήσαμε έτοιμο, τον LENS. Παρόλα αυτά μας πείρε αρκετό χρόνο λόγω των διάφορων προβλημάτων που συναντήσαμε. Σε αυτό το στάδιο τα πιο σημαντικά προβλήματα που συναντήσαμε και χρειαστήκαμε αρκετό χρόνο να τα λύσουμε ήταν, αρχικά το ότι ο kernel και ο gcc που είχαμε εγκατεστημένα στην διάθεση μας δεν υποστήριζαν τον LENS και έτσι και για τις δυο περιπτώσεις εγκαταστήσαμε νεότερες εκδώσεις. Και το άλλο μεγάλο πρόβλημα που συναντήσαμε ήταν τα “kernel panics” που είχαμε λόγω της διαφοράς της χωρητικότητας (128 GB έναντι 256 GB) της Optane μας, με αυτή που υλοποιήθηκε και ελέγχθηκε ο LENS.

Το δεύτερο στάδιο της εργασίας ήταν η συλλογή δεδομένων που χρειαζόμασταν για να δημιουργήσουμε. Ήταν ένα χρονοβόρο στάδιο επειδή έπρεπε να μελετήσουμε πολύ καλά τα microbenchmarks και τον κώδικα που θα χρησιμοποιήσουμε, αλλά και να βρούμε ποιες από της παραμέτρους που υπήρχαν θα ήταν βοηθητικές για τον δικό μας σκοπό. Και σε αυτό το στάδιο είχαμε κάποια σημαντικά προβλήματα. Ένα πρόβλημα που ήταν και ένα από συμπεράσματα που αποκτήσαμε ήταν, το ότι το /dev/shm έχει διαφορετική συμπεριφορά από ότι είχε ένα /dev/pmem device κυρίως στο kernel activity που είχε, όταν τρέχαμε τα microbenchmarks του PMDK. Επίσης, το άλλο σημαντικό θέμα που είχαμε ήταν τα μη αναμενόμενα αποτελέσματα που περνάμε στις μετρήσεις μας, όσον αφορά την αρχικοποίηση του hashmap όταν εκτελούσαμε ένα πείραμα.

Με την ολοκλήρωση του πρώτου στάδιού, και μετά από ένα σημείο του δευτέρου ξεκινήσαμε το τρίτο στάδιο που ήταν η ανάλυση των δεδομένων και η δημιουργία των μοντέλων μας, αυτό το στάδιο βέβαια ήταν και ο αρχικός μας στόχος. Στην προσπάθεια μας αυτή βρίσκαμε συνεχώς νέες προκλήσεις. Καταλάβαμε πως υπάρχουν αρκετές δυσκολίες στο να μοντελοποίησης ένα τέτοιο σύστημα μνήμης λόγω των πολλών μικρών μικρών παραμέτρων που μπορεί να υπάρχουν και ενδεχόμενος να επηρεάζουν το αποτέλεσμα. Κάποιες αξιοσημείωτες παράμετροι που θεωρούμε ότι έπαιξαν σημαντικό ρόλο είναι η εσωτερική παραλληλία που δύναται μα υπάρχει στο σύστημα μνήμης, η ανεπαρκής μοντελοποίηση του εύρος ζώνης της PMEM μας, και η αβεβαιότητα στο πότε χρησιμοποιείται η PMEM ή η DRAM στα LLC misses. Τέλος, πιστεύουμε ότι με την συνέχεια αυτής της εργασίας ή κάτι παρόμοιο θα υπάρξει ένα πολύ καλό αποτέλεσμα το οποίο μπορεί να έχει πραγματική χρήση.

8.2. Μελλοντική δουλεία και βελτιώσεις

Τα αποτελέσματα αυτής της εργασίας δύνανται να βελτιωθούν αρκετά. Όπως τονίσαμε και πιο πριν έχουμε κάποια προβλήματα στη εργασία μας που δεν μας επέτρεψαν να φτάσουμε με μεγάλη ακρίβεια στον θεωρητικό στόχο μας, δηλαδή να μοντελοποιήσουμε την απόδοση των συστημάτων μη πτητικής μνήμης. Έχοντας υπόψιν αυτό, σε μελλοντική δουλεία μπορούν να αναλυθούν και επιλυθούν αυτά τα

προβλήματα έτσι ώστε να πλησιάσουμε σε ένα καλύτερο αποτέλεσμα. Επίσης, εχτός από την επίλυση των προβλημάτων το αποτέλεσμα μας μπορεί να βελτιωθεί με την χρήση διαφορετικών εργαλείων για παράδειγμα, ή και χρήση μια πιο εξειδικευμένης τεχνικής που θα έχει καλύτερα αποτελέσματα από αυτές που χρησιμοποιήσαμε.

Τα κύρια προβλήματα της εργασίας μας είναι οι αδυναμίες των μοντέλων μας που αναφέρουμε στην ενότητα Αδυναμίες μοντέλων. Μια πιο ευρεία συλλογή από δεδομένα και μετρήσεις πολύ πιθανό να λύσουν προβλήματα όπως η αδυναμία προσημείωσης του εύρους ζώνης του συστήματος και το πρόβλημα του να μην συμπεριλαμβάνουμε την παραλληλία στο σύστημα μνήμης. Επίσης, μπορούν να χρησιμοποιηθούν κάποια προγράμματα ανάλυσης πηγαίου κώδικα έτσι ώστε να υπάρχει καλύτερη γνώση των σημαντικών σημείων του κώδικα που μελετήσαμε και χρησιμοποιήσαμε, με αποτέλεσμα να ξέρουμε με καλή ακρίβεια την αλληλεπίδραση του κώδικα με την μνήμη και συνεπώς τις προσβάσεις σε αυτή.

Από την άλλη μεριά, κάναμε αναφορές κατά την διάρκεια του δοκιμιού μας σε εργαλεία και τεχνικές που θα μπορούσαν να χρησιμοποιηθούν για ένα καλύτερο αποτέλεσμα. Για παράδειγμα, αντί το εργαλείο Perf μπορούν να χρησιμοποιηθούν τα πιο εξειδικευμένα εργαλεία pcm[14] ή το Likwid[15]. Επιπλέον, μπορούν να χρησιμοποιηθούν στα πειράματα πολλαπλές παράμετροι επιλογής που υπάρχουν για παράδειγμα στα microbenchmarks του PMDK. Τέλος, η χρησιμοποίηση μιας επιστημονικής μεθόδου για ανάλυση των δεδομένων που μαζευτήκαν ενδέχεται να βοηθήσει αρκετά. Απ’ ότι μπορούμε να καταλάβουμε είναι ένα πρόβλημα που έχει τεράστιο περιθώριο μελέτης και βελτίωση υφιστάμενων δουλειών.

Βιβλιογραφία

[1] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson and Jishen Zhao "Characterizing and Modeling Non-Volatile Memory Systems", pp 498-502, 2020, Repository https://github.com/TheNetAdmin/LENS/tree/201db92bf461a85ac00ecc847ce250266172e8ab

[2] Intel, "Achieve Greater Insight From Your Data with Intel® Optane™ Persistent Memory", 2019. Retrieved from https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory

[3] Piotr Balcer and, Andy Rudoff, "Persistent Memory Development Kit", 2019. Retrieved from https://pmem.io/pmdk/, Code https://github.com/pmem/pmdk

[4] "perf: Linux profiling with performance counters", 2006. Retrieved form https://perf.wiki.kernel.org/index.php/Main_Page

[5] Jian Yang, Juno Kim, and Morteza Hoseinzadeh, Joseph Izraelevitz "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory", pp 169-186, 2020.

[6] "NDCTL Introduction", Retrieved from https://docs.pmem.io/persistent-memory/getting-started-guide/what-is-ndctl

[7] Intel, "Intel® Optane™ Persistent Memory Start Up Guide", Retrieved from https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel_Optane_Persistent_Memory_Start_Up_Guide.pdf

[8] Ravi Saive, “How to Compile Linux Kernel on CentOS 7” 2018, Retrieved from https://www.tecmint.com/compile-linux-kernel-on-centos-7

[9] pkgs.org, “CentOS SCLo RH x86_64” Site https://centos.pkgs.org/7/centos-sclo-rh-x86_64/devtoolset-7-gcc-c++-7.2.1-1.el7.x86_64.rpm.html

[10] Wikipedia, “Hash table”, Retrieved from https://en.wikipedia.org/wiki/Hash_table

[11] Wikipedia , “Universal hashing”, Retrieved from https://en.wikipedia.org/wiki/Universal_hashing#Hashing_integers

[12] Quoc-Thai V Le, Usharani Upadhyayula, “How to Emulate Persistent Memory Using Dynamic Random-access Memory (DRAM)”, 2016, Retrieved from https://www.intel.com/content/www/us/en/developer/articles/training/how-to-emulate-persistent-memory-on-an-intel-architecture-server.html

[13] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li, “Quartz: A Lightweight Performance Emulator for Persistent Memory Software”, pp 37-49, 2019

[14] “Intel Processor Counter Monitor”, Code https://github.com/opcm/pcm

[15] “Likwid”, Code https://github.com/RRZE-HPC/likwid

[16] “Curve fitting”, Retried from https://en.wikipedia.org/wiki/Curve_fitting

[17] “Perf even open”, Retried from https://man7.org/linux/man-pages/man2/perf_event_open.2.html

Παράρτημα Α

impctl commad

Display select attributes with a comma separated list.

```
ipmctl show -d HealthState,HealthStateReason -dimm
```

For additional health statistics, use the *Show Sensor* command.

```
ipmctl show -a -sensor
```

Sensors are reported for the following health data.

Sensor Name	Description
Health	The current PMem health as reported in the SMART log
MediaTemperature	The current PMem media temperature in Celsius
ControllerTemperature	The current PMem controller temperature in Celsius
PercentageRemaining	Percentage of life remaining
LatchedDirtyShutdownCount	The number of shutdowns without notification over the lifetime of the PMem
UnlatchedDirtyShutdownCount	The number of shutdowns without notification over the lifetime of the PMem. This counter is the same as LatchedDirtyShutdownCount, except it will always be incremented on a dirty shutdown even if Latch System Shutdown Status was not enabled

Installation of LENS

```
git clone
https://github.com/TheNetAdmin/LENS/tree/201db92bf461a85ac00ecc847ce250266172e8ab
```

```
cd LENS
```

```
make -d
```

```
sudo ndctl create –namespace pmem0 -mode=fsdax
```

```
sudo ndctl create –namespace pmem1 -mode=fsdax
```

Μοντέλα

```
import numpy as np
from matplotlib import pyplot as plt
import sys
from math import log
import pandas as pd


def corr_analysis():
    data = pd.read_csv("cor.csv")
    corr_matrix = np.corrcoef(data).round(decimals=3)
    fig, ax = plt.subplots(figsize=(12,6))
    fig.tight_layout()
    im = ax.imshow(corr_matrix)
    im.set_clim(-1, 1)
    ax.grid(False)
    ax.xaxis.set(ticks=range(6), ticklabels= set(data.columns))
    ax.yaxis.set(ticks=range(6), ticklabels= set(data.columns))
    ax.set_ylim(5.5, -0.5)
    for i in range(6):
        for j in range(6):
            ax.text(j, i, corr_matrix[i, j], ha="center", va="center",
color="r")
    cbar = ax.figure.colorbar(im, ax=ax, format="% .2f")
    plt.show()


def main():
    # corr_analysis()
    L1_size = 32 * 10 ** 3
    L2_size = 1024 * 10 ** 3
    L3_size = 28 * 10 ** 6
    if len(sys.argv) >= 2:
        count = int(sys.argv[1])
    if len(sys.argv) == 3:
        size = int(sys.argv[2])

    # hardware resources
    cpu_fq = 2.1 * (10 ** 9)
    pmem_read_bw = .745 * (10 ** 9)
    pmem_latency_r = 305 * (10 ** -9)
    pmem_latency_w = 70 * (10 ** -9)
    dram_latency_r = 100 * (10 ** -9)
    dram_latency_w = 60 * (10 ** -9)

    # calculate the results
    time_overall = 0.0
```

```
    # cpu execution time
    instructions = 854 * count
    cycles = 1589 * count
    ipc = instructions/cycles
    cpu_time = cycles / cpu_fq * ipc
    # time_overall += cpu_time

    # memory latency
    # llc events per operation
    llc_loads = abs(4.966 * count - 262902)
    llc_stores = abs(6e-5 * count - 7117)
    llc_load_miss_ratio = abs(0.055 * log(count) - 0.242)
    llc_store_miss_ratio = abs(4e-8 * count + 0.81)
    l3_stalls = abs(706*count - 3e6)

    # pmem latency with llc loads/stores
    pmem_latency = (llc_loads * llc_load_miss_ratio * pmem_latency_r) + (
        llc_stores * llc_store_miss_ratio * pmem_latency_w
    )

    #bandwidth latency
    size_lat = count * size / pmem_read_bw
    time_overall += pmem_latency + size_lat

    print(time_overall)


if __name__ == "__main__":
    main()
```

```python
# pylint: disable=C,W
import sys


def main():
    if len(sys.argv) > 1:
        count = int(sys.argv[1])
    if len(sys.argv) > 2:
        size = int(sys.argv[2])

    pmem_bw = 7.45 * (10 ** 9)
    pmem_latency_r = 305 * (10 ** -9)
    dram_latency_r = 100 * (10 ** -9)
    pmem_latency_w = 70 * (10 ** -9)

    # calculate the results
    time_overall = 0.0

    # get hashmap's buckets
    time_overall += count * (pmem_latency_r)
    # hashing function time
    time_overall += count * (pmem_latency_r)
    # list the hashmap's buckets (POBJ_LIST_FOREACH)
    time_overall += count * (pmem_latency_r)
    # compare the value with the key
    time_overall += count * (pmem_latency_r) / 3
    # hashmap retrieve data
    time_overall += count * (pmem_latency_r + size / pmem_bw)

    print(time_overall)


if __name__ == "__main__":
    main()
```