ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ &
ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Performance and energy consumption evaluation of a distributed key-value store in x86 and ARM-based architectures

Μελέτη της απόδοσης και της κατανάλωσης ενέργειας κατανεμημένης βάσης δεδομένων σε x86 και ARM αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΒΑΛΣΑΜΙΔΗΣ ΣΤΑΜΑΤΙΟΣ

ΕΠΙΒΛΕΠΩΝ: Σούντρης Δημήτριος, Αναπληρωτής Καθηγητής Ε.Μ.Π.

ΑΘΗΝΑ, ΙΟΥΛΙΟΣ 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ & ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Performance and energy consumption evaluation of a distributed key-value store in x86 and ARM-based architectures

Μελέτη της απόδοσης και της κατανάλωσης ενέργειας κατανεμημένης βάσης δεδομένων σε x86 και ARM αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΒΑΛΣΑΜΙΔΗΣ ΣΤΑΜΑΤΙΟΣ

ΕΠΙΒΛΕΠΩΝ: Σούντρης Δημήτριος, Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 27/07/2017.

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Σούντρης Δημήτριος	Πεκμεστζή Κιαμάλ	Γεώργιος Γκούμας
Αν. Καθηγητής Ε.Μ.Π	Καθηγητής Ε.Μ.Π	Επ. Καθηγητής Ε.Μ.Π

(Υπογραφή)

.....

Βαλσαμίδης Σταμάτιος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Σταμάτιος Βαλσαμίδης, 27/07/2017

Ι. ΠΕΡΙΛΗΨΗ

Καθώς οι καταναλωτικές ανάγκες αυξάνονται, οι διαδικτυακές υπηρεσίες παράγουν έναν άνευ προηγουμένου όγκο δομημένων και μη δεδομένων. Επειδή οι πάροχοι υπηρεσιών αντλούν τεράστια αξία αποκτώντας γρήγορη πρόσβαση σε τέτοια δεδομένα, είναι κρίσιμη η ύπαρξη κατάλληλων υποδομών για αποθήκευση και ανάκτηση δεδομένων. Συνεπώς, οι τεχνολογίες που κλιμακώνονται και η αποθήκευση κλειδιού-τιμής αποτελούν εκ των πραγμάτων πρότυπο για τη δημιουργία τέτοιων υποδομών.

Η διπλωματική αυτή εστιάζει στο Memcached, μία πολύ δημοφιλή, ανοιχτού κώδικα βάση αποθήκευσης κλειδιού-τιμής. Η επίδοσή του και η κατανάλωση ενέργειας μετρούνται σε x86 και βασισμένες σε ARM υλοποιήσεις. Για λόγους σύγκρισης, στην ίδια x86 πλατφόρμα υλοποιούμε το MemC3, το οποίο αποτελεί μια βελτιωμένη έκδοση του Memcached.

Στο κεφάλαιο 1, πραγματοποιείται μια εισαγωγή σε σχετικές έννοιες και τομείς της πληροφορικής, όπως η κατανεμημένη πληροφορική, η πληροφορική υψηλών επιδόσεων και η πληροφορική του σύννεφου, όπως επίσης και η περιγραφή της αποθήκευσης κλειδιού-τιμής. Στο κεφάλαιο 2, παρουσιάζονται μελέτες σχετικές με το Memcached, με τα κύρια σημεία αυτού να τονίζονται. Στο κεφάλαιο 3 παρουσιάζεται το κύριο τμήμα της δουλειάς την οποία πραγματεύεται η διπλωματική αυτή. Διερευνώνται οι αρχιτεκτονικές του Memcached και του MemC3, όπως επίσης και τα benchmarks που χρησιμοποιούνται για τη μέτρηση της επίδοσής τους. Επιπλέον, αναλύονται τα πειραματικά αποτελέσματα. Στο κεφάλαιο 4, συνάγονται συμπεράσματα σχετικά με την επίδοση και την κατανάλωση ενέργειας του Memcached και του MemC3. Τέλος, στο κεφάλαιο 5 προτείνεται σχετική μελλοντική εργασία.

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Αποθήκευση κλειδιού-τιμής, Memcached, Πληροφορική υψηλών επιδόσεων, Κατανεμημένη πληροφορική, Βάσεις δεδομένων, Κατανάλωση ενέργειας

II. ABSTRACT

As consumer needs increase, web services generate an unprecedented amount of structured and unstructured data. Because service providers derive tremendous value from obtaining fast access to such data, it is critical to have the right infrastructure for data storage and retrieval. Consequently, scale-out technologies and key-value stores have become the de facto standard for deploying such infrastructure.

The focus of this thesis is Memcached, a very popular, open-source key-value store. Its performance and energy consumption are measured on an x86 and an ARM-based implementation. For comparison purposes, MemC3 is also implemented on the same x86 platform, which is an optimized version of Memcached.

In Chapter 1, an introduction to related notions and sectors of computing is made, namely distributed computing, high performance computing and cloud computing, as well as key-value storage. In Chapter 2, work related to Memcached is presented, with key points being highlighted. In Chapter 3, this thesis' main body of work is presented. Memcached's and MemC3's architectures are explored, as well as the benchmarks used to measure their performance. Furthermore, the experimental results are analyzed. In Chapter 4, conclusions are drawn regarding Memcached's and MemC3's performance and energy consumption. Finally, in Chapter 5, related future work is proposed.

KEYWORDS: Key-value store, Memcached, High performance computing, Distributed computing, Databases, Energy consumption

III. ΕΥΧΑΡΙΣΤΙΕΣ

Αρχικά, θα ήθελα να ευχαριστήσω θερμά τον αξιότιμο καθηγητή κ. Δημήτριο Σούντρη που μου μετέδωσε το μεράκι του για τον τομέα των μικροϋπολογιστών, VLSI και ενσωματωμένων συστημάτων, τόσο στα πλαίσια των μαθημάτων, όσο και εκτός αυτών. Ακόμη, θα ήθελα να εκφράσω την ευγνωμοσύνη μου για την ευκαιρία που μου έδωσε, επιτρέποντάς μου να ασχοληθώ με ένα τόσο ενδιαφέρον θέμα που αφορά σύγχρονες και εμπορικά χρησιμοποιούμενες τεχνολογίες.

Επιπλέον, θα ήθελα να ευχαριστήσω το δρ. Λάζαρο Παπαδόπουλο για τη συνεχή καθοδήγησή του καθ' όλη τη διάρκεια εκπόνησης της παρούσης διπλωματικής εργασίας, δίχως την οποία κάτι τέτοιο θα ήταν αδύνατον. Ευχαριστώ πολύ και τους Γεώργιο Ζερβάκη και Γιάννη Κούτρα για την πολύτιμη συνεισφορά τους σε τεχνικά ζητήματα.

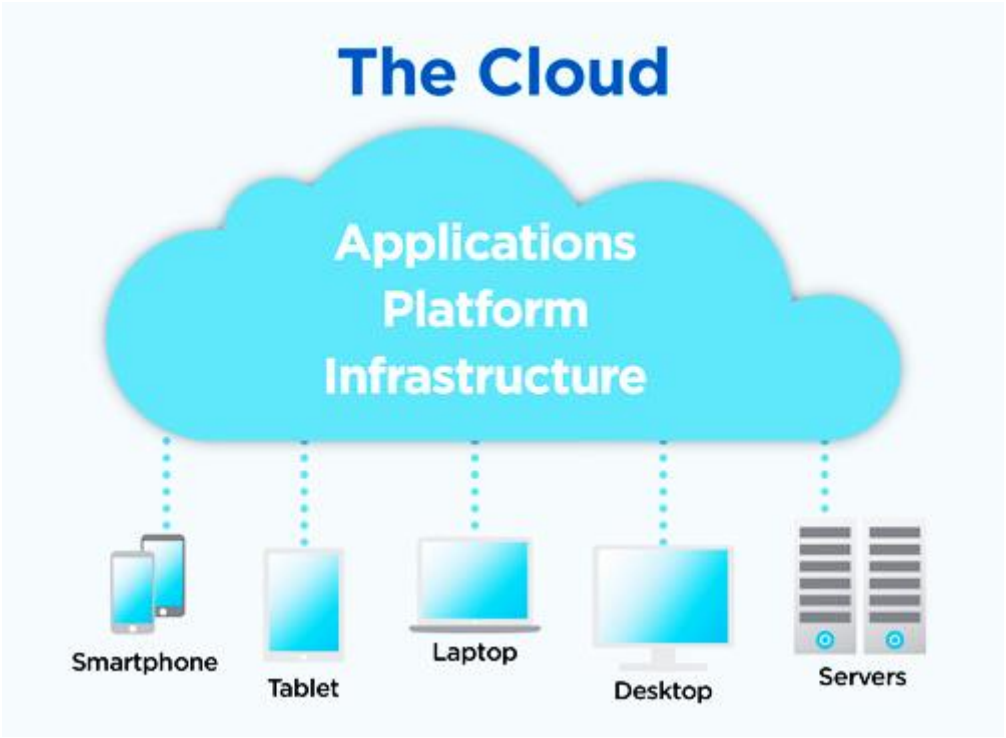
Τέλος, θα ήθελα να ευχαριστήσω θερμά τους στενούς μου φίλους για την συνεχή υποστήριξή τους. Ιδιαίτερα, όμως, θέλω να ευχαριστήσω την οικογένειά μου, η οποία με στήριξε πάσει δυνάμει σε ολόκληρη τη μαθητική και φοιτητική μου πορεία και συνεχίζει να με στηρίζει σε κάθε μου βήμα.

IV. ΕΚΤΕΤΑΜΕΝΗ ΠΕΡΙΛΗΨΗ

Ένας κλάδος της πληροφορικής που έχει επηρεάσει σημαντικά τις σύγχρονες εφαρμογές είναι η πληροφορική υψηλών επιδόσεων (High Performance Computing, HPC), που ασχολείται με τη συσσώρευση υπολογιστικών πόρων με τέτοιο τρόπο ώστε να επιτευχθούν πολύ μεγαλύτερες επιδόσεις σε σύγκριση με ένα επιτραπέζιο υπολογιστή, με σκοπό την επίλυση μεγάλων και δυσεπίλυτων προβλημάτων στον επιστημονικό ή επιχειρηματικό τομέα. Για να επιτευχθούν τέτοιοι στόχοι, οι υπολογιστές οργανώνονται σε συστάδες (clusters). Μία τυπική συστάδα υπολογιστών μπορεί να περιέχει από 4 τερματικά σε μια μικρή επιχείρηση έως 64 τερματικά σε μια μεγαλύτερη επιχείρηση. Αν το κάθε τερματικό διαθέτει τέσσερις υπολογιστικούς πυρήνες, τότε είναι διαθέσιμοι συνολικά από 16 έως 256 πυρήνες στην επιχείρηση.

Πλέον, οι συστάδες και γενικότερα οι υπολογιστικοί πόροι, αλλά και οι βάσεις δεδομένων, δε βρίσκονται σε ένα δωμάτιο. Μπορεί να βρίσκονται σε ξεχωριστά δωμάτια, κτήρια, χώρες ή ακόμη και σε διαφορετικές ηπείρους. Κατά συνέπεια, δημιουργήθηκε η έννοια του κατανεμημένου συστήματος. Συνεπώς, αυτή η εξέλιξη οδήγησε στη δημιουργία ενός ξεχωριστού κλάδου της πληροφορικής, της πληροφορικής κατανεμημένων συστημάτων. Σε ένα κατανεμημένο σύστημα, τα διάφορα μέρη που αποτελούν το σύστημα επικοινωνούν και συντονίζουν τις ενέργειές τους μέσω περάσματος μηνυμάτων. Με αυτό τον τρόπο, οι συνιστώσες του συστήματος αλληλεπιδρούν για να επιτύχουν έναν κοινό σκοπό. Τρία σημαντικά χαρακτηριστικά ενός κατανεμημένου συστήματος είναι η παραλληλία μεταξύ των συνιστωσών, η απουσία κοινού ρολογιού και η ανεξάρτητη βλάβη ή αποτυχία της κάθε συνιστώσας.

Οι ανωτέρω μέθοδοι και έννοιες έχουν συνεισφέρει καθοριστικά στη δημιουργία των μοντέρνων υπηρεσιών, οι οποίες παρέχονται μέσω του «σύννεφου» (cloud). Συνεπώς, η «πληροφορική του σύννεφου» (cloud computing) έχει αλλάξει άρδην την καθημερινή ζωή. Ο παραπάνω κλάδος της πληροφορικής ασχολείται με την παροχή διαμοιραζόμενων υπολογιστικών πόρων και δεδομένων σε υπολογιστές και άλλες συσκευές κατ’ απαίτηση. Η στροφή της πληροφορικής προς αυτή την κατεύθυνση έχει αλλάξει το τοπίο στην ενημέρωση, την κοινωνική δικτύωση, την εκπαίδευση, τις επιστήμες, την υγεία και σε πολλούς άλλους τομείς του δημόσιου και ιδιωτικού βίου.



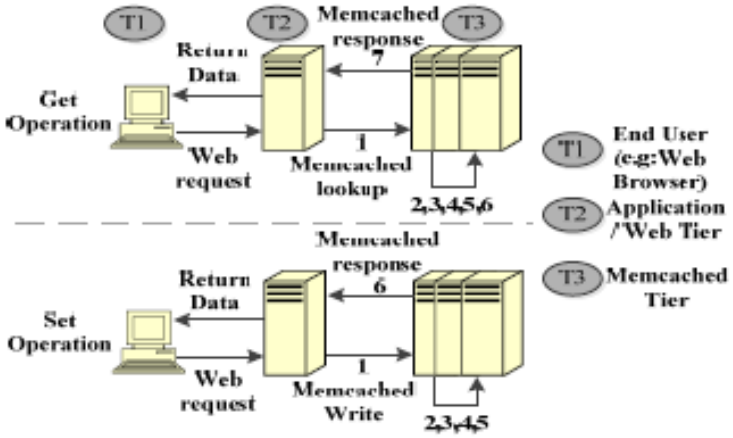
Εικόνα 0.1: Η πληροφορική του σύννεφου αποτελεί μία μεγάλη αλλαγή στην πληροφορική.

Με την πλέον ευρεία χρήση των παραπάνω τεχνικών έχει επιτευχθεί αύξηση της ταχύτητας εξυπηρέτησης του καταναλωτικού κοινού στο χώρο των διαδικτυακών υπηρεσιών. Ωστόσο, όσο οι καταναλωτικές αυξάνονται, οι υπηρεσίες δημιουργούν ένα απροσδόκητα μεγάλο όγκο δομημένων και μη δεδομένων. Ταυτόχρονα, τα εν λόγω δεδομένα διαθέτουν τεράστια αξία για τους παρόχους. Συνεπώς, είναι επιβεβλημένη η ύπαρξη των κατάλληλων υποδομών για την αποθήκευση και την ανάκτησή τους. Τεχνολογίες που κλιμακώνονται αλλά και αποθήκευση με ζεύγη κλειδιού-τιμής (key-value store) έχουν εδραιωθεί ως τέτοιες υποδομές, επιτρέποντας σε μεγάλης κλίμακας συστήματα παροχής περιεχόμενου να ανταποκρίνονται ικανοποιητικά στην ολοένα αυξανόμενη κίνηση και παραγωγή δεδομένων.

Υπάρχουν πολλά συστήματα αποθήκευσης με ζεύγη κλειδιού-τιμής. Το πιο γνωστό τέτοιο σύστημα είναι το Memcached, στο οποίο εστιάζει η παρούσα διπλωματική εργασία. Το Memcached είναι ένα ανοιχτού κώδικα, γενικού σκοπού σύστημα κρυφής κατανεμημένης μνήμης. Επιτυγχάνει επιτάχυνση ιστοτόπων που βασίζονται σε δυναμικές βάσεις δεδομένων μέσω τοποθέτησης δεδομένων και αντικειμένων σε κρυφή μνήμη υλοποιημένη σε μνήμη τυχαίας προσπέλασης (RAM), ώστε να μειωθεί ο αριθμός των αναγνώσεων μιας εξωτερικής πηγής δεδομένων. Η απλή σχεδίασή του διευκολύνει στην εύκολη και γρήγορη ανάπτυξή του, ενώ η προγραμματιστική διεπαφή του είναι διαθέσιμη για τις περισσότερες δημοφιλείς γλώσσες προγραμματισμού. Αρχικά, το Memcached είχε αναπτυχθεί από την Danga Interactive για το LiveJournal, αλλά πλέον χρησιμοποιείται ευρύτατα (ή έχει χρησιμοποιηθεί στο παρελθόν) από πολλά συστήματα, συμπεριλαμβανομένων των MocoSpace, YouTube, Reddit, Survata, Zynga, Facebook, Orange, Twitter, Tumblr και

Wikipedia, ενώ έχει αποτελέσει τη βάση και την έμπνευση για πολλά νεότερα παρόμοια συστήματα, όπως το Redis, το Voldemort και το Amazon ElastiCache.

Το Memcached παρέχει ένα απλό σύνολο λειτουργιών, ανάμεσα στις οποίες είναι οι set, get και delete, με τις δύο πρώτες να είναι οι σημαντικότερες. Η πρώτη γράφει το καθορισμένο ζεύγος κλειδιού-τιμής στον αποθηκευτικό χώρο του Memcached εξυπηρετητή. Η τιμή είναι τυπικά μικρά αντικείμενα, συχνά μεγέθους κάποιων εκατοντάδων bytes. Η δεύτερη (που είναι η πιο συνηθισμένη) ανακτά την τιμή που σχετίζεται με το καθορισμένο από το χρήστη κλειδί, αν αυτή υπάρχει στον εξυπηρετητή.



Εικόνα 0.2: Διάγραμμα αρχιτεκτονικής και χρήσης του Memcached

Μία συστάδα Memcached εξυπηρετητών παρέχουν έναν κατανεμημένο πίνακα κατακερματισμού για την αποθήκευση των αντικειμένων. Το κλειδί του αντικειμένου χρησιμοποιείται για τον καθορισμό του εξυπρετητή εντός της συστάδας όπου θα αποθηκευθεί το αντικείμενο. Μια συνάρτηση κατακερματισμού επιλέγεται ώστε να ισοκατανεμηθούν τα κλειδιά στη συστάδα.

Η διαχείριση της μνήμης πραγματοποιείται μέσω του διανεμητή πλακών (slab allocator). Ο διανεμητής οργανώνει τη μνήμη σε κλάσεις πλακών, καθεμιά από τις οποίες περιέχει προκαθορισμένου, ομοιόμορφου μεγέθους τμήματα μνήμης, τα οποία ξεκινούν από 64 bytes και φτάνουν μέχρι το 1 MB. Το Memcached αποθηκεύει αντικείμενα στη μικρότερη δυνατή κλάση πλάκας και ζητά περισσότερη μνήμη στη μορφή πλακών του 1 MB όταν η αντίστοιχη λίστα αδειάσει.

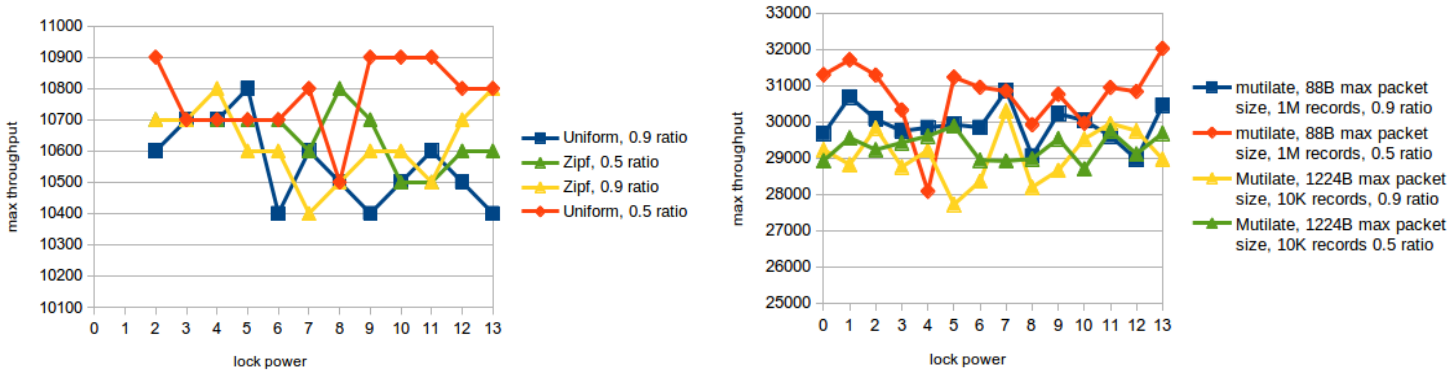
Στα πειράματα που παρουσιάζονται στην παρούσα εργασία χρησιμοποιείται – πέραν του Memcached- μια βελτιωμένη έκδοση του Memcached, το MemC3 (**Mem**cached with **C**LOCK and **C**oncurrent **C**uckoo Hashing). Το MemC3 χρησιμοποιεί ένα βελτιωμένο πίνακα κατακερματισμού που εκμεταλλεύεται την τοπικότητα της κρυφής μνήμης για να περιορίσει τον αριθμό των προσπελάσεων στη μνήμη καθώς και τη δυνατότητα για παραλληλία σε επίπεδο εντολών και μνήμης όταν πρέπει να πραγματοποιηθεί κάποια προσπέλαση. Ακόμη, το MemC3 κάνει χρήση αισιόδοξων σχημάτων κλειδώματος στοχευμένα στη συνηθισμένη περίπτωση, που είναι η ανάγνωση. Τέλος, χρησιμοποιεί μια παραλλαγή του κατακερματισμού cuckoo,

τον αισιόδοξο κατακερματισμό cuckoo, που επιτρέπει πολλαπλές αναγνώσεις και μία εγγραφή να λαμβάνουν χώρα ταυτόχρονα στον πίνακα κατακερματισμού.

Ακολουθούν τα πειραματικά αποτελέσματα, τα οποία πραγματοποιήθηκαν με τη χρήση δύο εφαρμογών αξιολόγησης που μπορούν να στέλνουν αιτήματα στον εξυπηρετητή και να λαμβάνουν απαντήσεις παίρνοντας μετρήσεις. Αυτά είναι: το Data Caching Benchmark, μέρος της σουίτας προγραμμάτων CloudSuite που έχει αναπτυχθεί στο εργαστήριο παράλληλων συστημάτων και αρχιτεκτονικής (PARSA, **PA**rallel **S**ystems **A**rchitecture) του Ομοσπονδιακού Πολυτεχνείου της Λωζάννης (EPFL, **É**cole **P**olytechnique **F**édérale de **L**ausanne) και το Mutilate. Στο πρώτο δίνεται η δυνατότητα προσομοίωσης ενός εξυπηρετητή του Twitter μέσω ενός αντίστοιχου συνόλου δεδομένων, ενώ μπορούν να αλλαχθούν ο αριθμός των TCP συνδέσεων, των νημάτων-εργατών (νημάτων από την πλευρά του πελάτη), η αναλογία αιτημάτων GET/SET και η κατανομή που ακολουθεί η επιλογή του κλειδιού. Στο δεύτερο μπορούν να μεταβληθούν ο αριθμός των TCP συνδέσεων, των νημάτων-εργατών (νημάτων από την πλευρά του πελάτη), η αναλογία αιτημάτων GET/SET και τα χαρακτηριστικά των δεδομένων (μέγεθος κλειδιού, τιμής και αριθμός εγγραφών).

Έχουν παρθεί μετρήσεις για τη μέγιστη διεκπεραιωτική ικανότητα του εξυπηρετητή συναρτήσει της «δύναμης κλειδώματος», που είναι ο λογάριθμος με βάση το 2 του συνόλου των κλειδωμάτων του Memcached, και του αριθμού των νημάτων του εξυπηρετητή. Επίσης, έχουν παρθεί μετρήσεις για το χρόνο απόκρισης του εξυπηρετητή συναρτήσει της διεκπεραιωτικής ικανότητας που στοχεύει να επιτύχει ο εξυπηρετητής. Για τα πειράματα χρησιμοποιήθηκαν το Memcached και το MemC3.

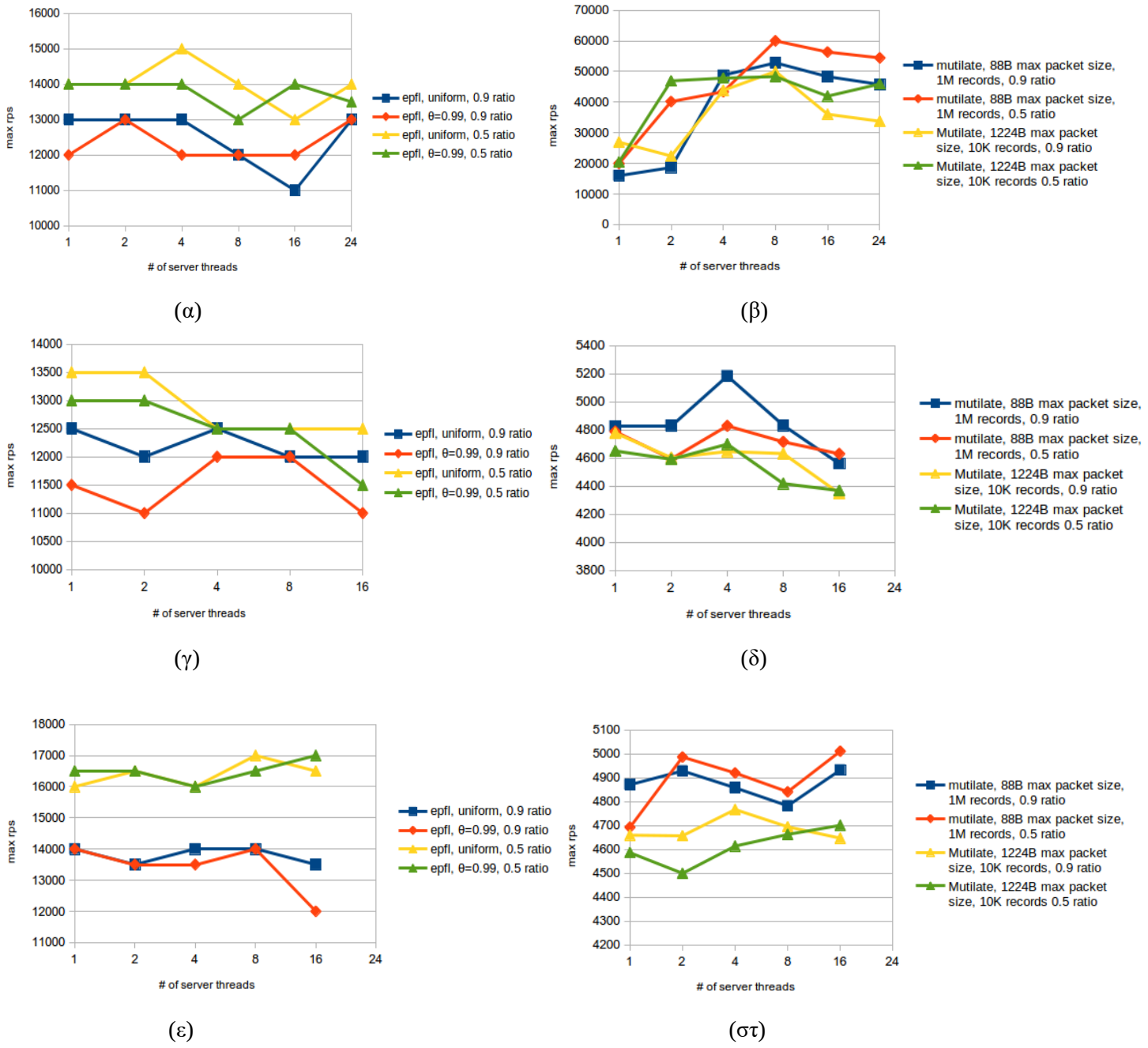
<u>Μέγιστη διεκπεραιωτική ικανότητα συναρτήσει δύναμης κλειδώματος</u>



(α) (β)

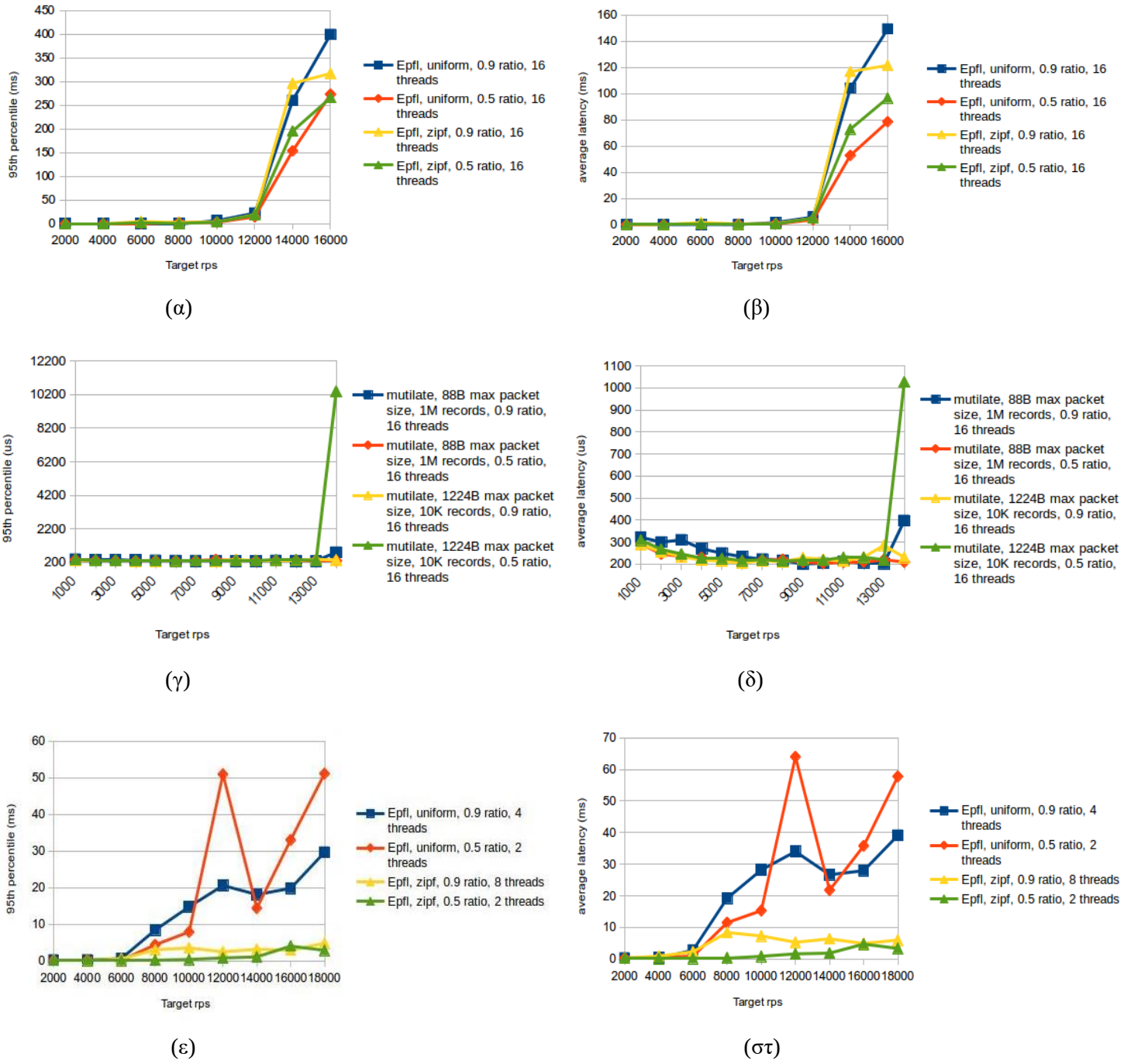
Εικόνα 0.3: (α) Memcached, Data Caching Benchmark, 100 συνδέσεις, 2 νήματα-εργάτες. (β) Memcached, Mutilate, 100 συνδέσεις, 2 νήματα-εργάτες.

Μέγιστη διεκπεραιωτική ικανότητα συναρτήσει αριθμού νημάτων εξυπηρετητή

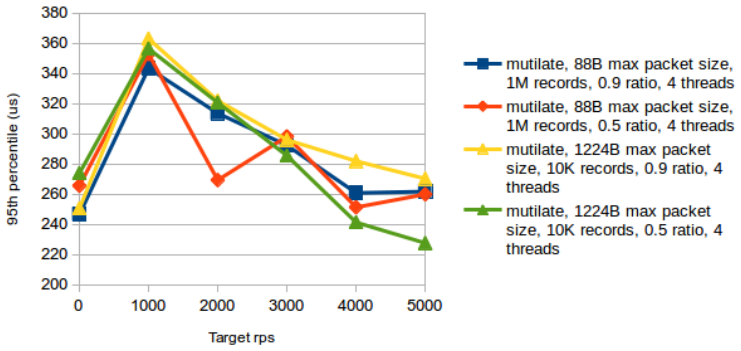


Εικόνα 0.4: (α) Memcached, Data Caching Benchmark, 100 συνδέσεις, 2 νήματα-εργάτες. (β) Memcached, Mutilate, 100 συνδέσεις, 2 νήματα-εργάτες. (γ) Memcached, Data Caching Benchmark, 1 σύνδεση, 1 νήμα-εργάτης. (δ) Memcached, Mutilate, 1 σύνδεση, 1 νήμα-εργάτης. (ε) MemC3, Data Caching Benchmark, 1 σύνδεση, 1 νήμα-εργάτης. (στ) MemC3, Mutilate, 1 σύνδεση, 1 νήμα-εργάτης.

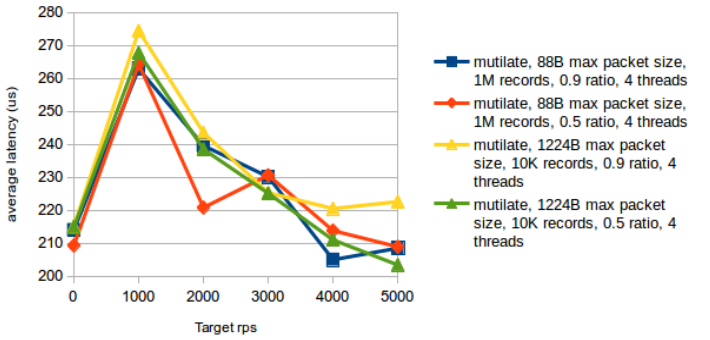
Χρόνος απόκρισης συναρτήσει μέγιστης διεκπεραιωτικής ικανότητας



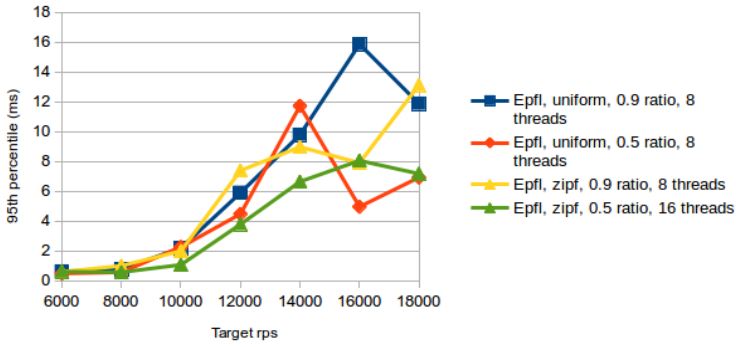
Εικόνα 0.5: (α) Memcached, Data Caching Benchmark, 100 συνδέσεις, 2 νήματα-εργάτες. (β) Memcached, Data Caching Benchmark, 100 συνδέσεις, 2 νήματα-εργάτες. (γ) Memcached, Mutilate, 100 συνδέσεις, 2 νήματα-εργάτες. (δ) Memcached, Mutilate, 100 συνδέσεις, 2 νήματα-εργάτες. (ε) Memcached, Data Caching Benchmark, 1 σύνδεση, 1 νήμα-εργάτης. (στ) Memcached, Data Caching Benchmark, 1 σύνδεση, 1 νήμα-εργάτης.



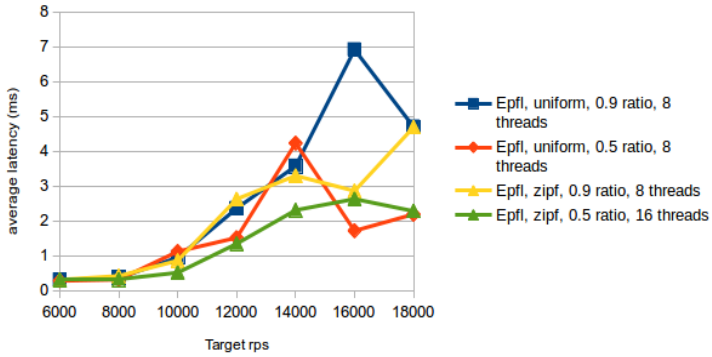
(α)



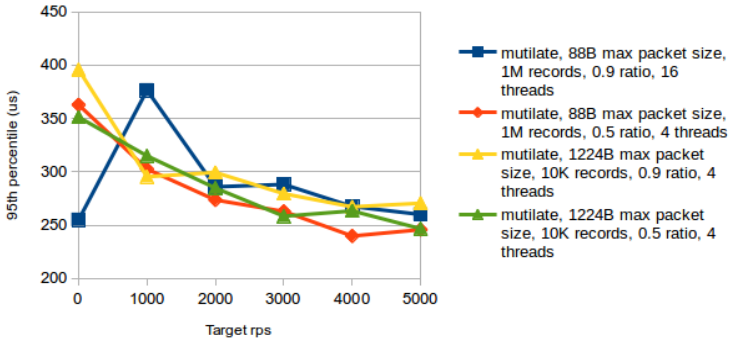
(β)



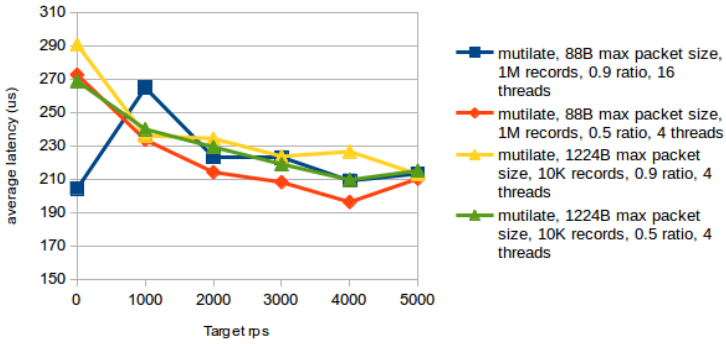
(γ)



(δ)



(ε)



(στ)

Εικόνα 0.6: (α) Memcached, Mutilate, 1 σύνδεση, 1 νήμα-εργάτης. (β) Memcached, Mutilate, 1 σύνδεση, 1 νήμα-εργάτης. (γ) MemC3, Data Caching Benchmark, 1 σύνδεση, 1 νήμα-εργάτης. (δ) MemC3, Data Caching Benchmark, 1 σύνδεση, 1 νήμα-εργάτης. (ε) MemC3, Mutilate, 1 σύνδεση, 1 νήμα-εργάτης. (στ) MemC3, Mutilate, 1 σύνδεση, 1 νήμα-εργάτης.

Συνολικά, τα αποτελέσματα δείχνουν ότι μία αναλογία 0.5 GET/SET αποδίδει μεγαλύτερο ρυθμό διεκπεραίωσης από μία αναλογία 0.9 GET/SET (κατά τη μέτρηση της μέγιστης διεκπεραιωτικής ικανότητας συναρτήσει της δύναμης κλειδώματος, το παραπάνω γίνεται πολύ πιο εμφανές σε υψηλές τιμής της δύναμης κλειδώματος, όπως φαίνεται στις εικόνες 0.3 (α) και (β)), ενώ παρουσιάζουν μεγαλύτερο κατώφλι ρυθμού

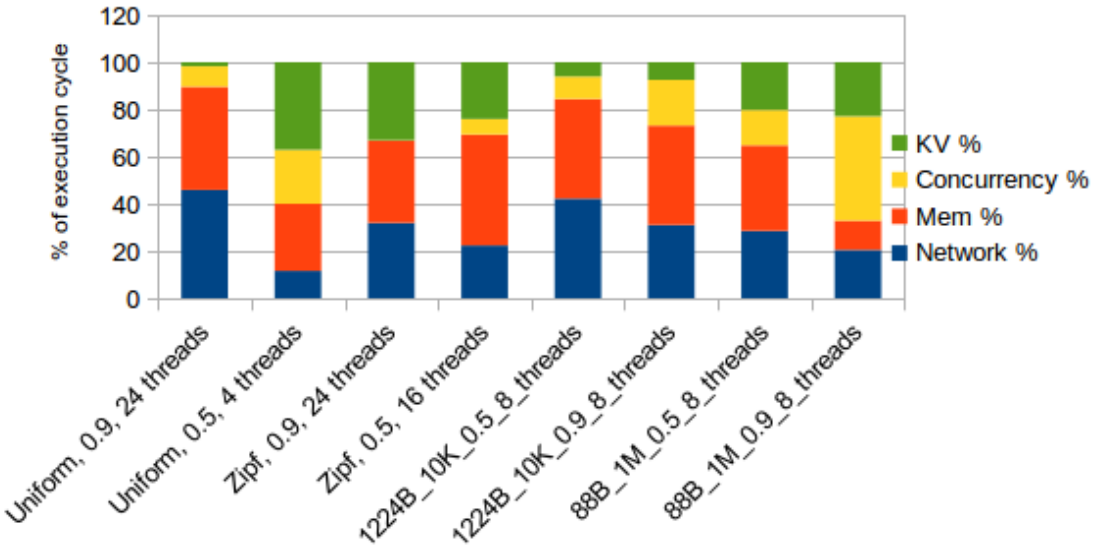
διεκπεραίωσης, προτού η καθυστέρηση αυξηθεί δραματικά. Αξίζει να σημειωθεί πως τα αποτελέσματα για x86 συστήματα δε δείχνουν συνέπεια, όπως φαίνεται στις Εικόνες 0.3 (α) και (β). Ωστόσο, οι διαφορές στη διεκπεραιωτική ικανότητα για τις διάφορες τιμές της δύναμης κλειδώματος (ουσιαστικά, του αριθμού των διαφορετικών κλειδωμάτων) είναι σχετικά μικρές.

Σχετικά με την κατανομή που ακολουθεί η επιλογή των κλειδιών του φορτίου, φαίνεται πως η ομοιόμορφη και η Zipf, οι οποίες χρησιμοποιήθηκαν, δεν παρουσιάζουν ιδιαίτερες διαφορές στην απόδοση του συστήματος. Ωστόσο, ως προς τη διεκπεραιωτική ικανότητα, η ομοιόμορφη κατανομή φαίνεται να δίνει ελαφρώς μεγαλύτερο ρυθμό διεκπεραίωσης σε σύγκριση με τη Zipf κατανομή (Εικόνες 0.3 (α), 0.4 (α), 0.4 (γ)). Από την άλλη, καθώς ο ρυθμός διεκπεραίωσης αυξάνεται, η ομοιόμορφη κατανομή τείνει να «εκτοξεύει» την καθυστέρηση σε μεγαλύτερες τιμές (Εικόνες 0.5 (α) και (β)) ή ταχύτερα (εικόνες 0.5 (ε), 0.5 (στ), 0.6 (γ), 0.6 (δ)) σε σχέση με τη Zipf. Επιπλέον, υψηλότεροι ρυθμοί διεκπεραίωσης επετεύχθησαν χρησιμοποιώντας φορτίο μεγάλο αριθμό πακέτων μικρού μεγέθους, παρά από ένα φορτίο λιγότερων πακέτων μεγάλου μεγέθους (Εικόνες 0.3 (β), 0.4 (β), 0.4 (δ), 0.4 (στ)). Ακόμη, τα γραφήματα του χρόνου απόκρισης συναρτήσει της μέγιστης διεκπεραιωτικής ικανότητας δείχνουν ότι φορτία φορτίο μικρού αριθμού πακέτων μεγάλου μεγέθους προκαλούν ραγδαία αύξηση του χρόνου απόκρισης κατόπιν ενός κατωφλίου πολύ γρηγορότερα από μεγάλο αριθμό πακέτων μικρού μεγέθους (Εικόνες 0.5 (γ), 0.5 (δ)).

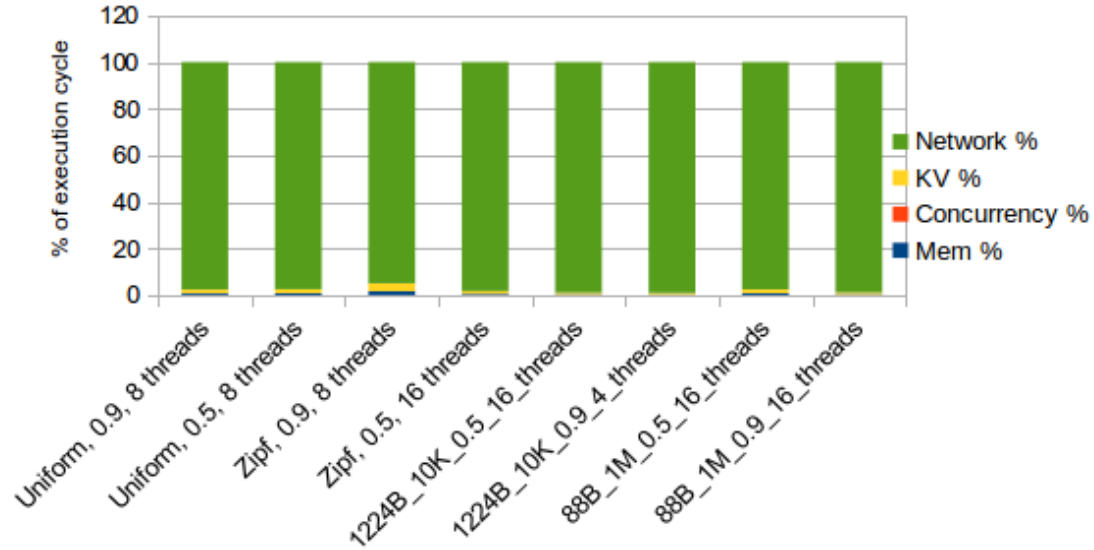
Αναφορικά με τη μέγιστη διεκπεραιωτική ικανότητα συναρτήσει του αριθμού των νημάτων, αξίζει να σημειωθεί ότι το MemC3 παρουσιάζει καλύτερη κλιμάκωση από το Memcached, όταν ο εξυπηρετητής διαθέτει περισσότερα από ένα νήματα, όπως φαίνεται στις Εικόνες 0.4 (ε) και 0.4 (στ) (σε σύγκριση με τα αντίστοιχα διαγράμματα στις Εικόνες 0.4 (γ) και 0.4 (δ)). Επιπρόσθετα, παρατηρούμε ότι περισσότερα παράλληλα αιτήματα οδηγούν σε μεγαλύτερο ρυθμό διεκπεραίωσης από τον εξυπηρετητή, γεγονός που είναι λογικό, αφού μία σύνδεση και ένα νήμα-εργάτης δεν είναι αρκετά για να προκαλέσουν κορεσμό της επεξεργαστικής ισχύος του.

Επιπλέον, πρέπει να σημειωθεί πως το MemC3 παρουσιάζει καλύτερη συμπεριφορά από το Memcached ως προς το χρόνο απόκρισης, με το τελευταίο να υποφέρει από ραγδαίες αυξήσεις του χρόνου απόκρισης πολύ γρηγορότερα από το πρώτο, καθώς αυξάνεται ο ρυθμός διεκπεραίωσης (Εικόνες 0.6 (γ), 0.6 (δ)). Ωστόσο, για χαμηλές τιμές ρυθμού διεκπεραίωσης, το Memcached και το MemC3 παρουσιάζουν παρόμοια συμπεριφορά (Εικόνες 0.6 (α), 0.6 (β), 0.6 (ε), 0.6 (στ)).

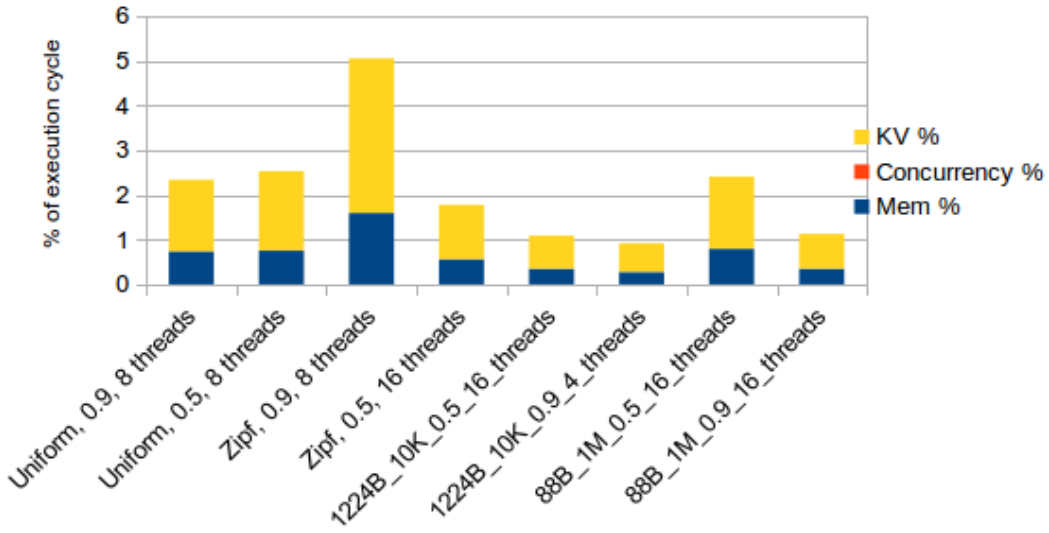
Για να επιτευχθεί μια περισσότερο εν τω βάθει ανάλυση σχετικά με την εκτέλεση του Memcached σε έναν εξυπηρετητή x86 αρχιτεκτονικής, χρησιμοποιήθηκε το εργαλείο SystemTap, το οποίο επιτρέπει την επόπτευση του συστήματος μέσω scripts της ομώνυμης γλώσσας. Συγκεκριμένα, το SystemTap (μαζί με ένα Python script) χρησιμοποιήθηκε για να μετρηθεί ο χρόνος των κλήσεων συστήματος που σχετίζονται με κάθε τμήμα του κύκλου εκτέλεσης του SystemTap (στοίβα δικτύου, έλεγχος παραλληλίας, διαχείριση μνήμης, επεξεργασία κλειδιού-τιμής). Ακολουθούν τα σχετικά πειραματικά αποτελέσματα (Εικόνες 0.7, 0.8, 0.9).



Εικόνα 0.7: Memcached, ανάλυση κύκλου εκτέλεσης.



Εικόνα 0.8: MemC3, ανάλυση κύκλου εκτέλεσης.



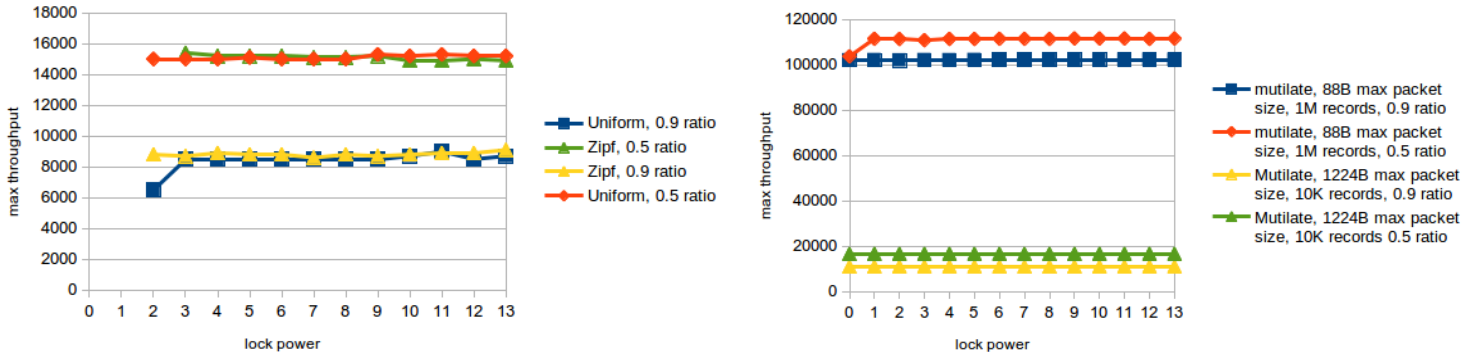
Εικόνα 0.9: MemC3, ανάλυση κύκλου εκτέλεσης χωρίς επεξεργασία δικτύου.

Τα αποτελέσματα, που απεικονίζονται στην Εικόνα 0.7, δείχνουν ότι το μεγαλύτερο εμπόδιο στην απόδοση του συστήματος είναι οι σχετικές με τη μνήμη κλήσεις συστήματος, που καταλαμβάνουν, κατά μέση τιμή, το 36% του κύκλου εκτέλεσης. Οι σχετικές με το δίκτυο κλήσεις συστήματος είναι συνεχώς ένα μεγάλο κομμάτι του κύκλου εκτέλεσης (29% κατά μέση τιμή). Επιπλέον, συμπεραίνουμε ότι το φορτίο με μεγάλο μέγεθος πακέτων και μικρό μέγεθος εγγραφών δεν αποδίδει καλά, λόγω του μικρού ποσοστού που καταλαμβάνει η επεξεργασία κλειδιού-τιμής. Αξίζει να σημειωθεί πως ο έλεγχος παραλληλίας δε φαίνεται να εμποδίζει σημαντικά τις επιδόσεις του Memcached για τις τιμές παραμέτρων που επιλέχθηκαν.

Παρόλη την απουσία παραλληλίας, η ανάλυση του κύκλου εκτέλεσης του MemC3 παρέχει μια καλύτερη εικόνα. Αρχικά, είναι εμφανές ότι το κομμάτι του δικτύου κυριαρχεί έναντι των υπολοίπων (Εικόνα 0.8). Με τη βοήθεια του δεύτερου διαγράμματος (Εικόνα 0.9), στην οποία η συνεισφορά των σχετικών με δίκτυο κλήσεων συστήματος έχει αφαιρεθεί, φαίνεται ότι οι σχετικές με τη μνήμη κλήσεις συστήματος καταλαμβάνουν ένα πολύ μικρό ποσοστό του κύκλου εκτέλεσης. Η επεξεργασία κλειδιού-τιμής καταλαμβάνει ένα αρκετά μεγαλύτερο ποσοστό του κύκλου εκτέλεσης σε σχέση με το κομμάτι που σχετίζεται με τη μνήμη.

Επιπλέον της υλοποίησης σε x86 σύστημα, το Memcached υλοποιήθηκε στον LS2085A επεξεργαστή, ο οποίος διαθέτει υπολογιστικούς πυρήνες βασισμένους στην ARM αρχιτεκτονική. Στα σχετικά πειράματα χρησιμοποιήθηκε μόνο το Memcached με 100 TCP συνδέσεις και 2 νήματα-εργάτες. Τα πειραματικά αποτελέσματα παρατίθενται παρακάτω.

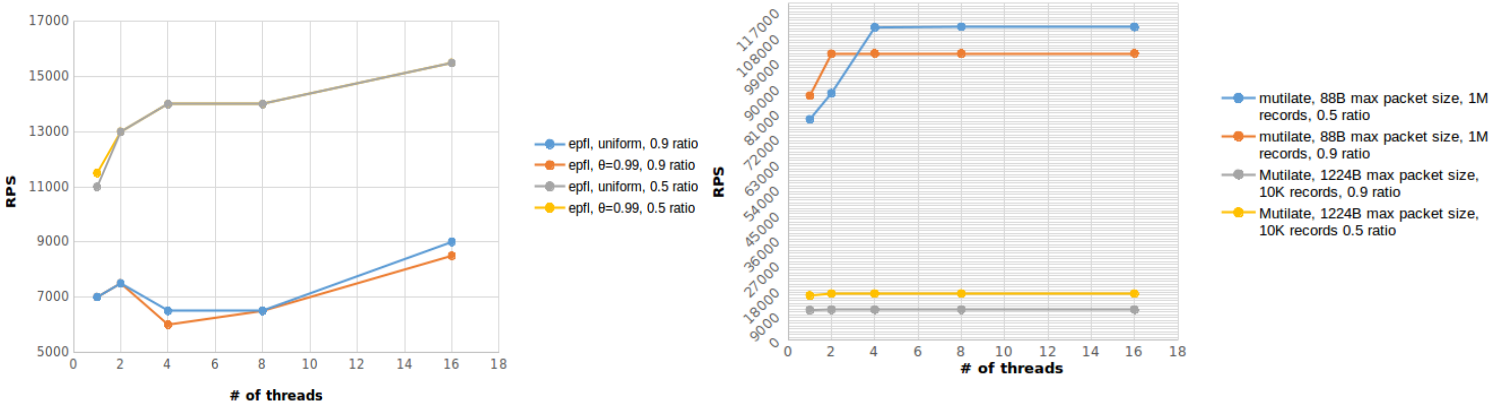
Μέγιστη διεκπεραιωτική ικανότητα συναρτήσει δύναμης κλειδώματος



(α) (β)

Εικόνα 0.10: (α) Memcached, Data Caching Benchmark. (β) Memcached, Mutilate.

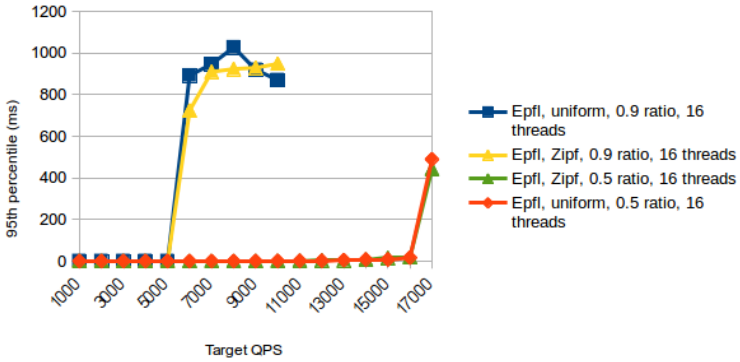
Μέγιστη διεκπεραιωτική ικανότητα συναρτήσει αριθμού νημάτων εξυπηρετητή



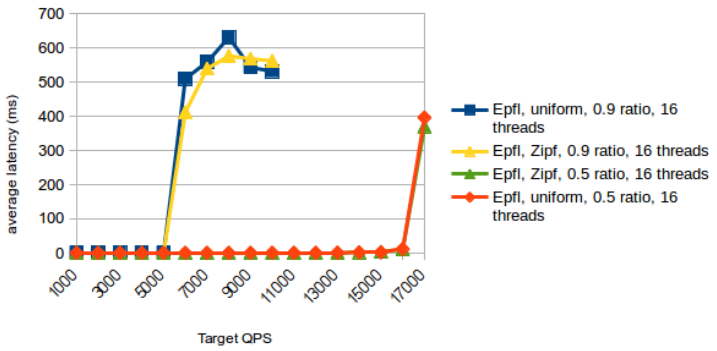
(α) (β)

Εικόνα 0.11: (α) Memcached, Data Caching Benchmark. (β) Memcached, Mutilate.

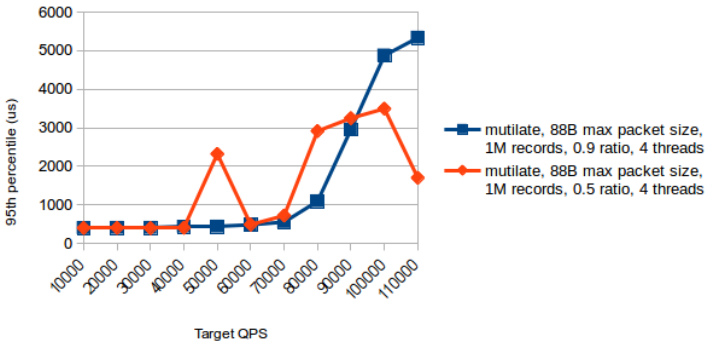
Χρόνος απόκρισης συναρτήσει μέγιστης διεκπεραιωτικής ικανότητας



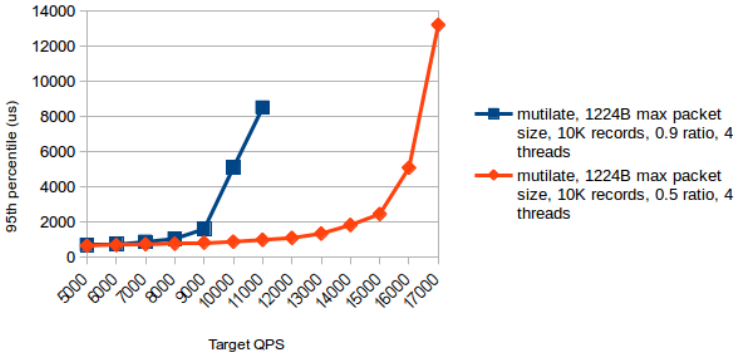
(α)



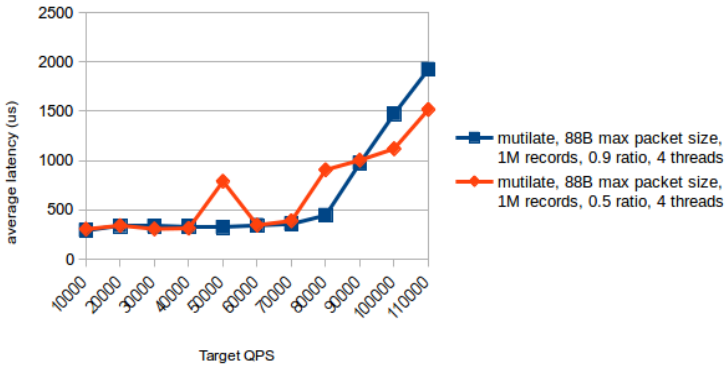
(β)



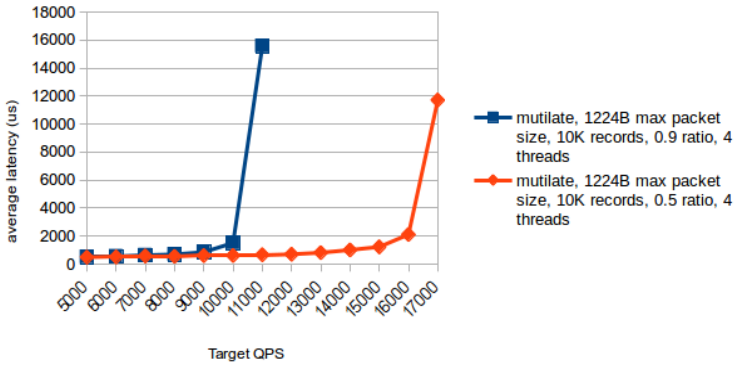
(γ)



(δ)



(ε)



(στ)

Εικόνα 0.12: (α) Memcached, Data Caching Benchmark. (β) Memcached, Data Caching Benchmark. (γ) Memcached, Mutilate. (δ) Memcached, Mutilate. (ε) Memcached, Mutilate.(στ) Memcached, Mutilate.

Τα ανωτέρω διαγράμματα δείχνουν τα ίδια χαρακτηριστικά με την περίπτωση του x86 συστήματος αναφορικά με αλλαγές στις παραμέτρους του φορτίου. Πιο συγκεκριμένα, ο εξυπηρετητής τείνει να αποδίδει καλύτερα με μία αναλογία 0.5 GET/SET σε σχέση με μία αναλογία 0.9 GET/SET, καθώς και μεγαλύτερη διεκπεραιωτική ικανότητα και στιβαρότητα ως προς το χρόνο απόκρισης με μεγάλο μέγεθος εγγραφών και μικρό μέγεθος πακέτων παρά με μικρό μέγεθος εγγραφών και μεγάλο μέγεθος πακέτων. Επιπρόσθετα, η ομοιόμορφη και η Zipf κατανομή που ακολουθούν τα κλειδιά οδηγούν σε παρόμοιες επιδόσεις.

Ωστόσο, ο LS2085A, ο οποίος κάνει χρήση πυρήνων βασισμένων στην αρχιτεκτονική ARM, φαίνεται να επιτυγχάνει υψηλότερους ρυθμούς διεκπεραίωσης σε σχέση με τον επεξεργαστή Intel Xeon, που είναι x86 αρχιτεκτονικής (για τις ίδιες τιμές παραμέτρων), κρατώντας το χρόνο απόκρισης σταθερό (Εικόνες 0.10 (α), 0.10 (β), 0.11 (α), 0.11 (β)). Από την άλλη πλευρά, ο πρώτος φαίνεται να έχει μικρότερη ανοχή σε υψηλή διεκπεραιωτική ικανότητα από το δεύτερο, πράγμα το οποίο φαίνεται από το γεγονός ότι ο χρόνος καθυστέρησης ανεβαίνει περισσότερο όταν ένα κατώφλι ρυθμού διεκπεραίωσης έχει ξεπεραστεί (Εικόνες 0.12 (α), 0.12 (β), 0.12 (γ), 0.12 (δ), 0.12 (ε), 0.12 (στ)). Ακόμη, πρέπει να σημειωθεί πως, με ένα σύνολο δεδομένων που χαρακτηρίζεται από μεγάλο μέγεθος εγγραφών και μικρό μέγεθος πακέτων, ο LS2085A επιτυγχάνει διεκπεραιωτική ικανότητα μία τάξη μεγέθους μεγαλύτερη από την αντίστοιχη που προκύπτει από σύνολο δεδομένων που χαρακτηρίζεται από μικρό μέγεθος εγγραφών και μεγάλο μέγεθος πακέτων (Εικόνες 0.10 (b), 0.11 (b)).

Τέλος, πραγματοποιήθηκαν μετρήσεις για τη διεκπεραιωτική ικανότητα ανά μονάδα ισχύος, ώστε να φανεί αν είναι ωφέλιμη η υλοποίηση του Memcached σε έναν επεξεργαστή βασισμένο σε ARM αρχιτεκτονική από τη σκοπιά της κατανάλωσης ισχύος. Καθώς αυξανόταν ο αριθμός των νημάτων του εξυπηρετητή, τα οποία αναθέτονταν σε συγκεκριμένους επεξεργαστικούς πυρήνες, βρέθηκε ότι η μέγιστη διεκπεραιωτική ικανότητα είναι 165 KRPS (χιλιάδες αιτήματα ανά δευτερόλεπτο) για 5, 6, 7 και 8 νήματα. Κατά συνέπεια, για τη μέτρηση της κατανάλωσης ισχύος, επιλέχθηκε ο ελάχιστος αριθμός νημάτων, δηλαδή 5. Με τη βοήθεια βατομέτρου, μετρήθηκε ότι ο εξυπηρετητής κατανάλωσε 51.6 W σε ηρεμία, 58.5 W με όλους τους πυρήνες ενεργούς και 57.2 W με πέντε πυρήνες ενεργούς. Κατά συνέπεια, η μέγιστη διεκπεραιωτική ικανότητα του LS2085A είναι 160KRPS/57.2W ≅ 2.9 KRPS/Watt. Η σύγκριση με άλλες υλοποιήσεις του Memcached που βρίσκονται στη βιβλιογραφία φαίνονται στον παρακάτω πίνακα.

Υλοποίηση	Ρυθμός Διεκπεραίωσης (KRPS)	KRPS/Watt
MemC3 (NSDI '13)	1500 KRPS	3.8
LS2085A	165 KRPS	2.9
Υλοποίηση σε Xeon (NSDI '14)	700 KRPS	1.8
Υλοποίηση σε Xeon (ISCA'13)	410 KRPS	1
Υλοποίηση σε Xeon (ISCA '15)	300 KRPS	0.8

Πίνακας 0.1: Σύγκριση υλοποίησης του Memcached στον LS2085A με άλλες υλοποιήσεις

Από τα πειραματικά αποτελέσματα μπορούν να αντληθούν κάποια συμπεράσματα σχετικά με τη συμπεριφορά του Memcached. Αρχικά, το Memcached φαίνεται να παρουσιάζει προβλήματα απέναντι σε αιτήματα τύπου GET παρά SET. Αυτό οφείλεται στο γεγονός ότι τα αιτήματα τύπου GET κοστίζουν περισσότερο από τα αιτήματα τύπου SET˙ τα πρώτα απαιτούν 7 κοστοβόρα βήματα για να εκτελεστούν, ενώ τα δεύτερα 6 απλούστερα βήματα. Επιπλέον, το αίτημα GET μπορεί να εμπλέκει κλειδί το οποίο δεν υπάρχει στην κρυφή μνήμη, γεγονός το οποίο επιβάλλει περαιτέρω καθυστέρηση. Ακόμη, πολλά πακέτα με μικρό μέγεθος μειώνουν σημαντικά τη διεκπεραιωτική ικανότητα του Memcached σε σχέση με λίγα πακέτα μεγάλου μεγέθους, διότι τα πρώτα προκαλούν κορεσμό της σωλήνωσης της CPU του εξυπηρετητή, ενώ τα τελευταία δεν καταφέρνουν να προκαλέσουν κάτι τέτοιο. Συγκεκριμένα, το τελευταίο είδος φορτίου περιορίζεται από το εύρος ζώνης της κάρτας διεπαφής δικτύου του εξυπηρετητή, που, στις παραπάνω περιπτώσεις, δεν αξιοποιήθηκε εξ ολοκλήρου. Ωστόσο, στα παραπάνω πειράματα παρατηρήθηκαν καλύτερες επιδόσεις με μικρού μεγέθους πακέτα σε σύγκριση με μεγάλου μεγέθους πακέτα. Αυτό συμβαίνει επειδή χωρούν λιγότερα αντικείμενα στη μνήμη (σε σχέση με την πρώτη περίπτωση) και είναι λίγα σε πλήθος. Συνεπώς, η πιθανότητα να ζητηθεί αντικείμενο που έχει προηγουμένως εκδιωχθεί από την κρυφή μνήμη είναι μεγαλύτερη, οδηγώντας σε χαμηλότερη διεκπεραιωτική ικανότητα. Επιπρόσθετα, πρέπει να σημειωθεί πως η κατανομή που ακολουθεί η επιλογή του κλειδιού δε φαίνεται να επηρεάζει σημαντικά τις επιδόσεις του συστήματος, αν και συνήθως η ομοιόμορφη κατανομή έδινε καλύτερα αποτελέσματα ως προς τη διεκπεραιωτική ικανότητα. Ωστόσο, ο χρόνος εκτέλεσης είναι ένας σημαντικός παράγοντας ώστε να καταδειχθεί αν αυτή η παράμετρος επηρεάζει σημαντικά το σύστημα ή όχι.

Σχετικά με την ανάλυση του κύκλου εκτέλεσης του Memcached, κατέστη φανερό το γεγονός ότι οι σχετικές με τη διαχείριση μνήμης κλήσεις συστήματος αποτελούν σημαντικό εμπόδιο στην απόδοση του Memcached, ανεξάρτητα του είδους του φορτίου. Συνεπώς, συμπεραίνεται ότι το σχήμα διαχείρισης μνήμης με πλάκες δεν είναι ιδιαίτερα αποδοτικό. Επιπλέον, το παρεχόμενο από το λειτουργικό σύστημα POSIX I/O επιβαρύνει σημαντικά το Memcached, καθιστώντας το ακατάλληλο για ένα αποδοτικό σύστημα αποθήκευσης ζεύγους κλειδιού-τιμής.

Αναφορικά με το MemC3, είναι εμφανές ότι μπορεί να επιτύχει πολύ καλύτερες επιδόσεις από το Memcached, χωρίς καν να χρησιμοποιηθούν όλες οι βελτιστοποιήσεις

που έχουν γίνει. Αυτό δείχνει ότι το σχήμα κατακερματισμού του Memcached δεν είναι αποδοτικό, με το αντίστοιχο σχήμα του MemC3 να αποτελεί μια καλύτερη επιλογή.

Τέλος, η υλοποίηση του Memcached στον LS2085A έδειξε καλά αποτελέσματα. Αυστηρά ως προς τις επιδόσεις, επιτεύχθη μεγαλύτερη διεκπεραιωτική ικανότητα σε σχέση με το x86 εξυπηρετητή, ιδιαίτερα για πιο εύκολα αντιμετωπίσιμα φορτία, περιπτώσεις στις οποίες υπήρχε διαφορά μίας τάξης μεγέθους στην απόδοση. Ωστόσο, ο χρόνος απόκρισης αυξάνεται πιο γρήγορα καθώς αυξάνεται η τιμή-στόχος του ρυθμού διεκπεραίωσης στον LS2085A σε σχέση με τον x86 εξυπηρετητή. Ακόμη, ο LS2085A εμφάνισε εξαιρετική απόδοση ανά μονάδα ισχύος, ξεπερνώντας αρκετές υλοποιήσεις σε εξυπηρετητές με Intel Xeon CPUs που αναφέρονται στη βιβλιογραφία. Συνολικά, θα πρέπει να εξεταστεί σοβαρά σαν υποψήφια πλατφόρμα για το Memcached και άλλες συναφείς εφαρμογές.

Στο μέλλον θα ήταν καλό να εξεταστούν οι δυνατότητες του MemC3 αξιοποιώντας πλήρως τις βελτιστοποιήσεις που έχουν γίνει, αλλά και πλήρη παραλληλία. Άλλα παρόμοια συστήματα μπορούν να εξετασθούν για σκοπούς σύγκρισης σε διάφορα σενάρια και συνδυασμούς παραμέτρων. Τα παραπάνω συστήματα αποθήκευσης ζεύγους κλειδιού-τιμής μπορούν να υλοποιηθούν τόσο σε εξυπηρετητή αρχιτεκτονικής x86, όσο στον LS2085A. Επίσης, θα πρέπει να εκτελεστούν πειράματα μεγάλης χρονικής διάρκειας, αλλάζοντας την κατανομή που ακολουθεί η επιλογή των κλειδιών, ώστε να εξακριβωθεί η επίπτωση που έχει το είδος της κατανομής στις επιδόσεις του συστήματος. Τέλος, θα ήταν ενδιαφέρον να γίνει προσπάθεια πλήρους αξιοποιήσεις των υλοποιήσεων που διαθέτει ο LS2085A για δικτυακές εφαρμογές.

V. CONTENTS

1. INTRODUCTION

1.1. HIGH PERFORMANCE COMPUTING

A notion and sector of computing that has impacted modern applications greatly is high performance computing (HPC). High Performance Computing most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business [16].

High-performance computing has become indispensable to the ability of enterprises, scientific researchers, and government agencies to generate new discoveries and to innovate breakthrough products and services [17]. More specifically, high performance computers of interest to small and medium-sized businesses today are really clusters of computers. Each individual computer in a commonly configured small cluster has between one and four processors, and today's processors typically have from two to four cores. In HPC, individual computers in a cluster are often referred to as nodes. A cluster of interest to a small business could have as few as four nodes, or 16 cores. A common cluster size in many businesses is between 16 and 64 nodes, or from 64 to 256 cores [16]. In science, HPC provides the tools and methods needed for data intensive and computationally intensive applications. For example, the Human Genome Project uses HPC in order to determine the sequence of nucleotide base pairs that make up human DNA and of identifying and mapping all of the genes of the human genome from both a physical and functional standpoint [18]. Another well-known example is IBM's Watson, a supercomputer which, in tandem with its cognitive capabilities, tackles challenges in Big Data in healthcare and finance [19].

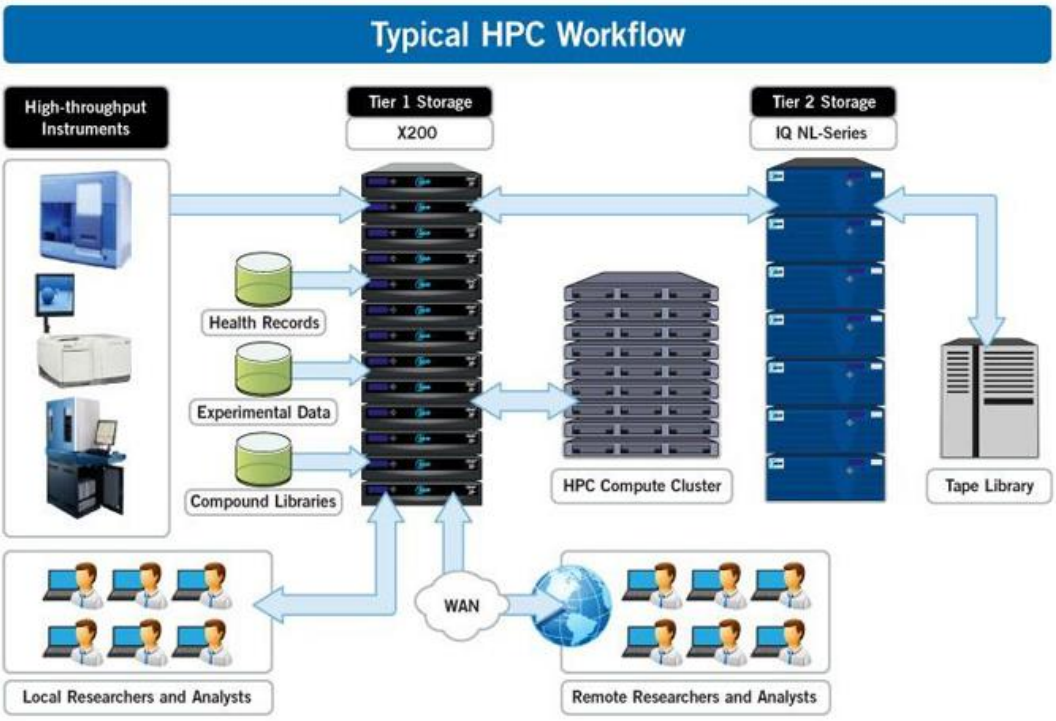


Figure 1.1: Typical HPC Workflow [50].

1.2. DISTRIBUTED COMPUTING

As mentioned before, computers are organized in clusters. Thus, computing resources, as well as databases, tend to exist in different computers, rooms, buildings, countries or continents. Thus, there is a field of computer science that studies distributed systems: distributed computing. A distributed system is a model in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal. Three significant characteristics of distributed systems are: concurrency of components, lack of a global clock, and independent failure of components [20].

The key goals of a distributed system include:

- Transparency: Achieving the image of a single system image without concealing the details of the location, access, migration, concurrency, failure, relocation, persistence and resources to the users.
- Openness: Making the network easier to configure and modify.
- Reliability: Compared to a single system, a distributed system should be highly capable of being secure, consistent and have a high capability of masking errors.
- Performance: Compared to other models, distributed models are expected to give a much-wanted boost to performance.
- Scalability: Distributed systems should be scalable with respect to geography, administration or size.

In the enterprise, distributed computing has often meant putting various steps in business processes at the most efficient places in a network of computers. For example, in the typical distribution using the 3-tier model, user interface processing is performed in the PC at the user's location, business processing is done in a remote computer, and database access and processing is conducted in another computer that provides centralized access for many business processes. Typically, this kind of distributed computing uses the client/server communications model. [21]

1.3. CLOUD COMPUTING

The above notions, methods and models help power modern applications, which exist and are accessible through the cloud. Consequently, it would be fair to say that cloud computing is changing our lives in many ways. We are, as never before, seeing cloud technology impact our world on many levels. The likes of YouTube and Google are testimony to a shift in how people are now interacting with each other. From remote locations to the global center stage, an event can reach the four corners of the planet by going viral. Global has reached its true significance, and we've seen the emerging of the "citizen journalist" on this global stage. [43]

So, what is cloud computing? Cloud computing is a type of Internet-based computing that provides shared computer processing resources and data to computers and other devices on demand. It is a model for enabling ubiquitous, on-demand access to a shared pool of configurable computing resources (e.g. computer networks, servers, storage, applications and services), which can be rapidly provisioned and released with minimal management effort [45], [46].

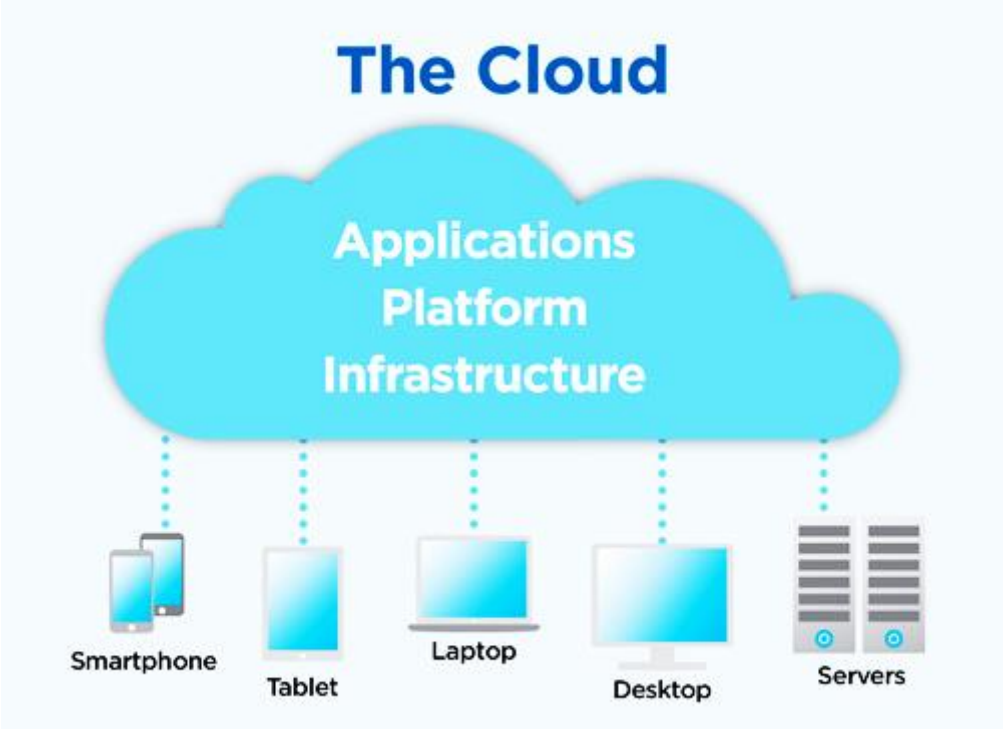


Figure 1.2: Cloud computing represents a major generational shift in enterprise IT [51].

Nowadays, anyone can turn into an instant reporter and live news feeds are constantly streaming the media, at times sparking social upheavals. Through social networks, one can look up forgotten friends and classmates with ease; Facebook is a prime example of this. Twitter has turned into a platform for public figures and politicians in order to convey their views to the public and influence peoples' opinion. Furthermore, social media platforms, in combination with cloud-based information resources, provide an immense data pool from which businesses can get better insights on potential services, innovations and customer requirements, if analyzed properly.

Cloud computing has had an enormous impact on education, as well. Educational institutions have been quick to realize the advantages of cloud technology and have been eagerly adopting it for several reasons, including:

- Ability for the students to access data anywhere, anytime, to enroll in online classes and to participate in group activities.
- The value of combining business automation processes to streamline subscription, class enrollments and assignment tracking, thus reducing expenses significantly.
- Ability for the institutional body to leverage the storage cloud to store the daily 2.5 quintillion bytes of data securely and without the need to cater to a complicated infrastructure.
- The benefit of process billing and charging for education and non-education related activities.
- While these are probably most obvious in a mature and developed market, cloud computing technology also offers benefits to students from developing countries. Access is now instantly available and in many instances free thanks to the proliferation of websites dispensing educational material and cloud knowledge-sharing communities.

As mentioned above, cloud technology also offers other benefits to developing countries since they no longer have the burden of investing in costly infrastructures and can tap into data and applications that are readily available in the cloud. This does not apply only to education, however, as the same can be said about banking, agriculture, health and science. A prime example of this is the telecom industry, with developing countries embracing the smart mobile technology that accelerated development by leaping over the traditional wire and copper infrastructure.

Finally, cloud computing has changed the health sector quite noticeably. There are many reasons why using cloud technology in the healthcare industry is gaining pace. Some examples include:

- Managing non-siloed patient data and sharing it among different parties such as medical professionals or patients checking their own status and treatment follow-ups;
- Reducing operational costs such as data storage;
- Accessing this data through pervasive devices such as mobile phones and going beyond the traditional intranet;
- Implementing a quick solution in a secure environment that is compliant with the Health Insurance Portability and Accountability Act regulations.

While there may be challenges in integrating old or current tools with new technologies and the corresponding level of services, the benefits will outweigh the inhibition to move to the cloud. According to the industry, healthcare will be a growing market in the coming years, running into the billions [43], [44].

1.4. KEY-VALUE STORE

Nowadays, as consumer needs increase, web services generate an unprecedented amount of structured and unstructured data. Because service providers derive tremendous value from obtaining fast access to such data, it is critical to have the right infrastructure for data storage and retrieval. Scale-out technologies and key-value stores have become the de facto standard for deploying such infrastructure, as most of the largest content-serving systems rely on the ability to scale quickly in response to increasing client traffic and data generation [4].

A key-value store, or key-value database, is a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a dictionary or hash. Dictionaries contain a collection of objects, or records, which in turn have many different fields within them, each containing data. These records are stored and retrieved using a key that uniquely identifies the record, and is used to quickly find the data within the database [47].
Many implementations that are not well-suited to traditional relational databases can benefit from a key-value model, which offers several advantages, including:

- Flexible data modeling: Because a key-value store does not enforce any structure on the data, it offers tremendous flexibility for modeling data to match the requirements of the application.
- High performance: Key-value architecture can be more performant than relational databases in many scenarios because there is no need to perform lock, join, union, or other operations when working with objects. Unlike traditional relational databases, a key-value store doesn't need to search through columns or tables to find an object. Knowing the key will enable very fast location of an object.
- Massive scalability: Most key-value databases make it easy to scale out on demand using commodity hardware. They can grow to virtually any scale without significant redesign of the database.
- High availability: Key-value databases may make it easier and less complex to provide high availability than can be achieved with relational database. Some key-value databases use a masterless, distributed architecture that eliminates single points of failure to maximize resiliency.
- Operational simplicity: Some key-value databases are specifically designed to simplify operations by ensuring that it is as easy as possible to add and remove capacity as needed and that any hardware or network failures within the environment do not create downtime [48].

There are many key-value stores that are used commercially. The most well-known key-value store is Memcached, which is the focus of this thesis. Memcached is an open-source, general-purpose distributed memory caching system. It speeds up dynamic

database-driven websites by caching data and objects in RAM to reduce the number of times an external data source must be read. Its simple design promotes quick deployment, ease of development and solves many problems facing large data caches. Its API is available for most popular languages [22]. Memcached was originally developed by Danga Interactive for LiveJournal, but is now used (or has been used) by many other systems, including MocoSpace [23], YouTube [24], Reddit [25], Survata [26], Zynga [27], Facebook [2], [28], [29], Orange [30], Twitter [31], Tumblr [32] and Wikipedia [33] and has been the basis and inspiration for many newer key-value stores.

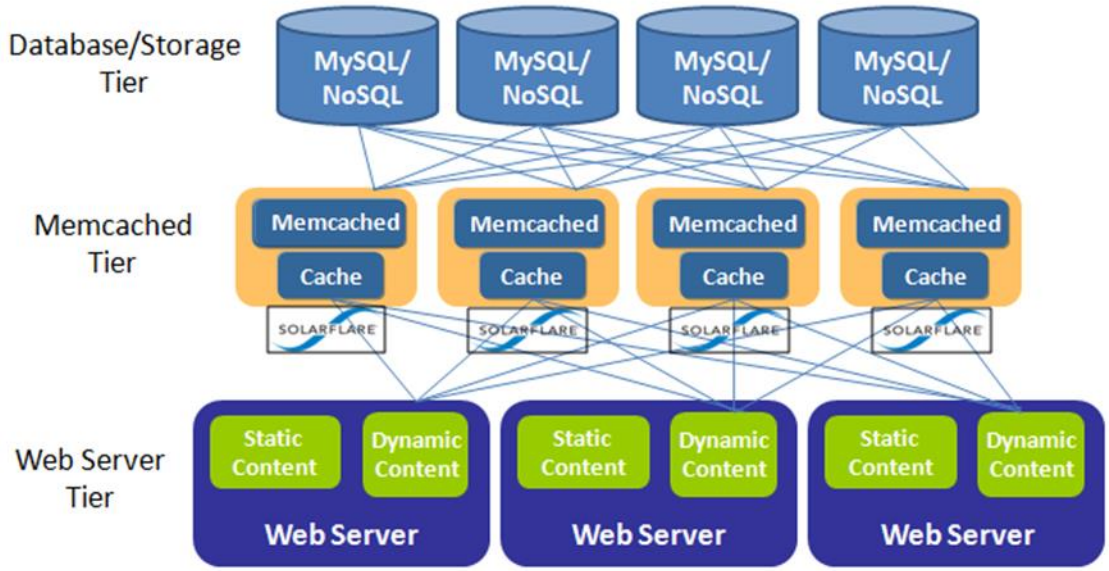


Figure 1.3: Web Server/Memcached/Database tier.

Another very popular key-value store is Redis. Redis (**RE**mote **DI**ctionary **S**erver), which is mainly developed by Salvatore Sanfilippo and is currently sponsored by Redis Labs, is an in-memory database open-source software project implementing a networked, in-memory key-value store with optional durability. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, sorted sets, hyperloglogs, bitmaps and spatial indexes. Redis is used commercially by Twitter [34], Github and Blizzard Interactive [15].

LinkedIn makes use of Voldemort, which is a distributed data store that is designed as a key-value store used for high-scalability storage. Voldemort is neither an object database, nor a relational database. It does not try to satisfy arbitrary relations and the ACID properties, but rather is a big, distributed, fault-tolerant, persistent hash table [35].

Amazon adopts a different approach. Amazon's subsidiary, Amazon Web Services, provides on-demand cloud computing platforms to both individuals, companies and governments, using the technology of the same name. Amazon Web Services uses Amazon ElastiCache, which is a fully managed in-memory data store and cache service. Amazon ElastiCache supports both Memcached and Redis (also called "ElastiCache for Redis"). As a web service running in the computing cloud, Amazon ElastiCache is designed to simplify the setup, operation, and scaling of memcached and Redis deployments. Complex administration processes like patching software, backing

up and restoring data sets and dynamically adding or removing capabilities are managed automatically. Scaling ElastiCache resources can be performed by a single API call [36], [37].

Finally, Facebook turned to a flash memory key-value implementation. Compared with memory, flash provides up to 20 times the capacity per server and still supports tens of thousands of operations per second Thus, Facebook replaced Memcached with McDipper, a flash-based cache server that is Memcache protocol compatible (Memcache is Facebook's Memcached implementation, which is further discussed in chapter 2) [38].

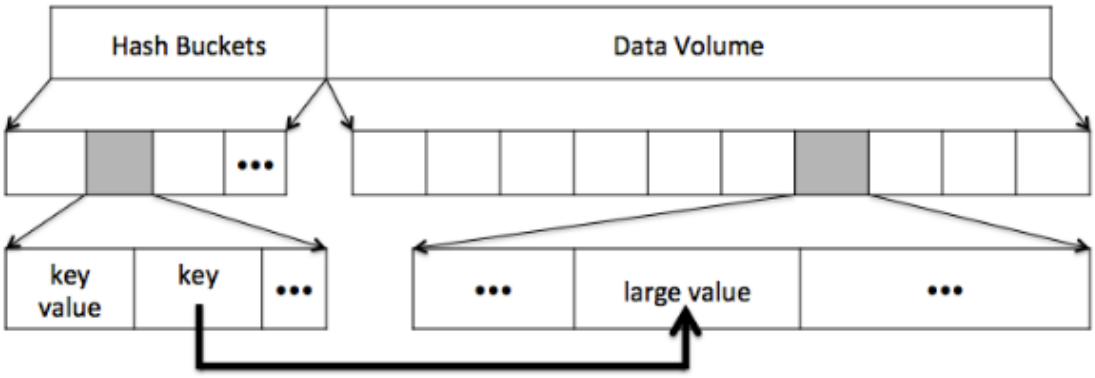


Figure 1.4: McDipper storage layout. [39]

2. MEMCACHED RELATED WORK

As has already been discussed, distributed in-memory key-value stores have become a critical part of the infrastructure for large scale Internet-oriented datacenters. They are deployed at scale across server farms inside companies such as Facebook, Twitter, Amazon and LinkedIn [1]. In particular, Memcached, which is the focus of this thesis, is one of the most popular choices implemented by the industry.

In this chapter, work related to Memcached will be presented. This work is conducted by researchers from both the industry and academia and is primarily focused on optimizing Memcached using varying methods and secondarily on identifying its bottlenecks. Before presenting this work, however, Memcached's advantages will be summarily presented.

2.1. MEMCACHED ADVANTAGES

One of the most fundamental aspects to Memcached is its all-in memory design. By solely utilizing memory, Memcached provides lower latency data access than other comparable storage systems. This low-latency performance is critical for interactive web applications, evidenced by web service providers such as Facebook and Zynga dedicating an entire tier of servers to Memcached [4].

Moreover, Memcached is easy to scale from a cluster perspective. Memcached servers themselves do not directly interact, nor do they require centralized coordination. If either throughput or capacity demands grow, a pool can be scaled simply by directing clients to connect to additional servers; consistent hashing mechanisms immediately map a fraction of keys to the new servers. Operationally, it is useful that key-value throughput can be scaled independent of the back-end storage system (database servers are typically considerably more expensive than Memcached servers) [3].

2.2. MEMCACHED BOTTLENECKS

Unfortunately, Memcached presents poor scalability in terms of performance. It does not achieve the performance that modern hardware is capable of [1]. As such, there has been a dire need to pinpoint Memcached's bottlenecks, which prevent it from scaling well. To that end, a lot of research has been conducted by experts both in the academic and industrial field. Work in [1] suggests that the issue is that Memcached uses the operating system's network stack, heavyweight locks for concurrency control, inefficient data structures, and expensive memory management. These impose high overheads for network processing, concurrency control, and key-value processing. As a result, Memcached shows poor performance and energy efficiency when running on commodity servers.

Furthermore, it is mentioned in [3] that Memcached suffers from architectural inefficiencies. More specifically, the most significant bottlenecks lie in the processor front-end, with poor instruction cache and branch predictor performance. Also, it is shown that object size plays a significant role in Memcached's performance: below 1 KB, performance is CPU-bound, while above 1 KB object size, the performance bottleneck shifts to the network. However, it should be noted that the quality of the NIC used is more important than raw bandwidth [3].

Another bottleneck was the global cache lock that Memcached utilized for concurrency control. In particular, the global cache lock was a major bottleneck for Memcached's performance for more than four threads. However, this has been resolved, as lock striping has been implemented to reduce lock contention [40].

2.3. MEMCACHED OPTIMIZATIONS

2.3.1. SOFTWARE OPTIMIZATIONS

MemC3

MemC3 (**Mem**cached with **C**LOCK and **C**oncurrent **C**uckoo Hashing) is introduced in [10] and features several software optimizations. MemC3 has architectural features can hide memory access latencies and provide performance improvements, leverages workload characteristics and introduces a new hashing scheme. These changes will be presented more thoroughly in chapter 3.

MICA

In [14] MICA (Memory-store with Intelligent Concurrent Access) is presented, which is an in-memory key-value store that achieves high throughput across a wide range of workloads. MICA can provide either store semantics (no existing items can be removed without an explicit client request) or cache semantics (existing items may be removed to reclaim space for new items). Under write-intensive workloads with a skewed key popularity, a single MICA node serves 70.4 million small key-value items per second (Mops), which is 10.8x faster than the next fastest system. For skewed, read-intensive workloads, MICA's 65.6 Mops is at least 4x faster than other systems even after modifying them to use our kernel bypass. MICA achieves 75.5–76.9 Mops under workloads with a uniform key popularity. MICA achieves this through the following techniques:

- Fast and scalable parallel data access: MICA's data access is fast and scalable, using data partitioning and exploiting CPU parallelism within and between cores. Its EREW mode (Exclusive Read Exclusive Write) minimizes costly inter-core communication, and its CREW mode (Concurrent Read Exclusive Write) allows multiple cores to serve popular data. MICA's techniques achieve consistently high throughput even under skewed workloads, one weakness of prior partitioned stores.

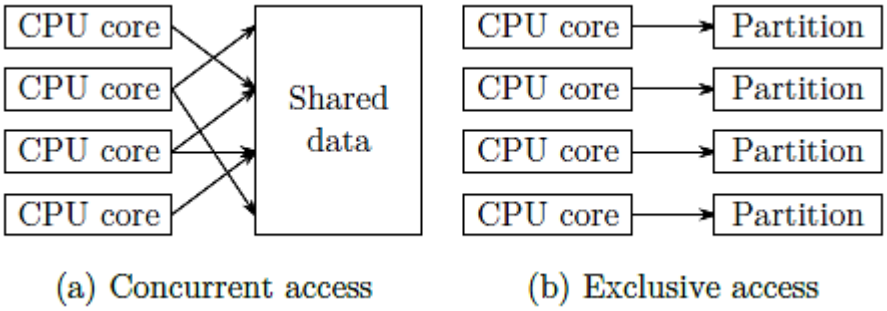


Figure 2.1: Parallel data access models. [14]

- Network stack for efficient request processing: MICA interfaces with NICs directly, bypassing the kernel, and uses client software and server hardware to direct remote key-value requests to appropriate cores where the requests can be processed most efficiently. The network stack achieves zero-copy packet I/O and request processing.
- New data structures for key-value storage: New memory allocation and indexing in MICA, optimized for store and cache separately, exploit properties of key-value workloads to accelerate write performance with simplified memory management.

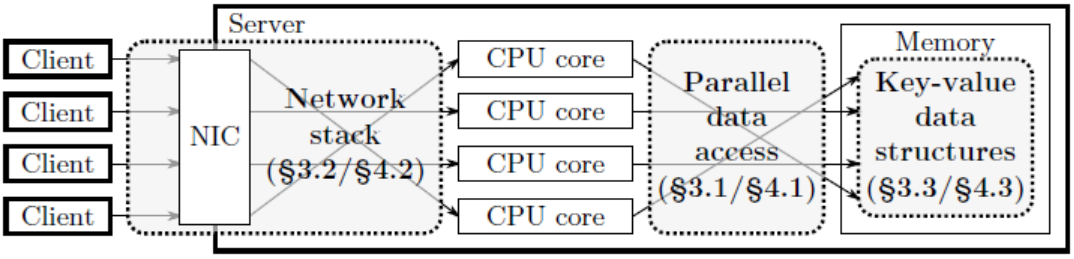


Figure 2.2: Components of in-memory key-value stores. [14]

It should be noted that an optimized MICA design in introduced in [1], which manages to reach approximately 1.2 billion RPS.

Memcache

Facebook engineers utilize Memcached to construct a distributed key-value store that supports Facebook that they call Memcache [2]. Memcache serves as a demand-filled look-aside cache in order to lighten the read load on Facebook's databases, as well as a more general key-value store (e.g. to store pre-computed results from sophisticated machine learning algorithms which can then be used by a variety of other applications).

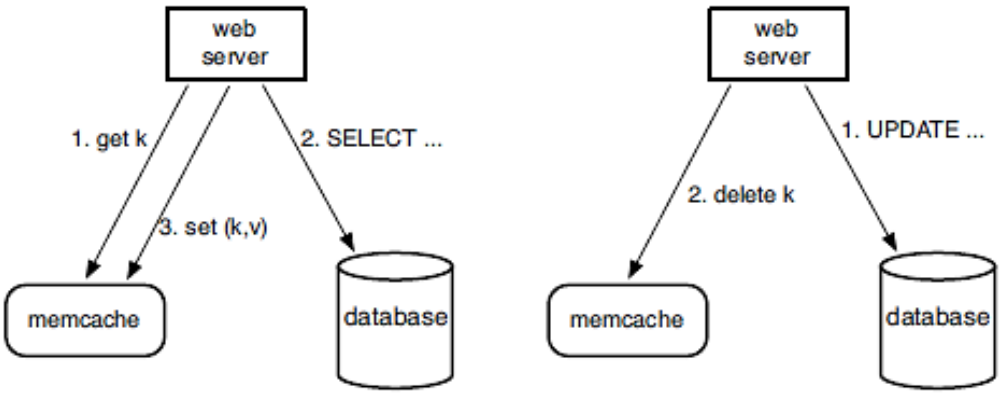


Figure 2.3: Memcache as a demand-filled look-aside cache. The left half illustrates the read path for a web server on a cache miss. The right half illustrates the write path. [2]

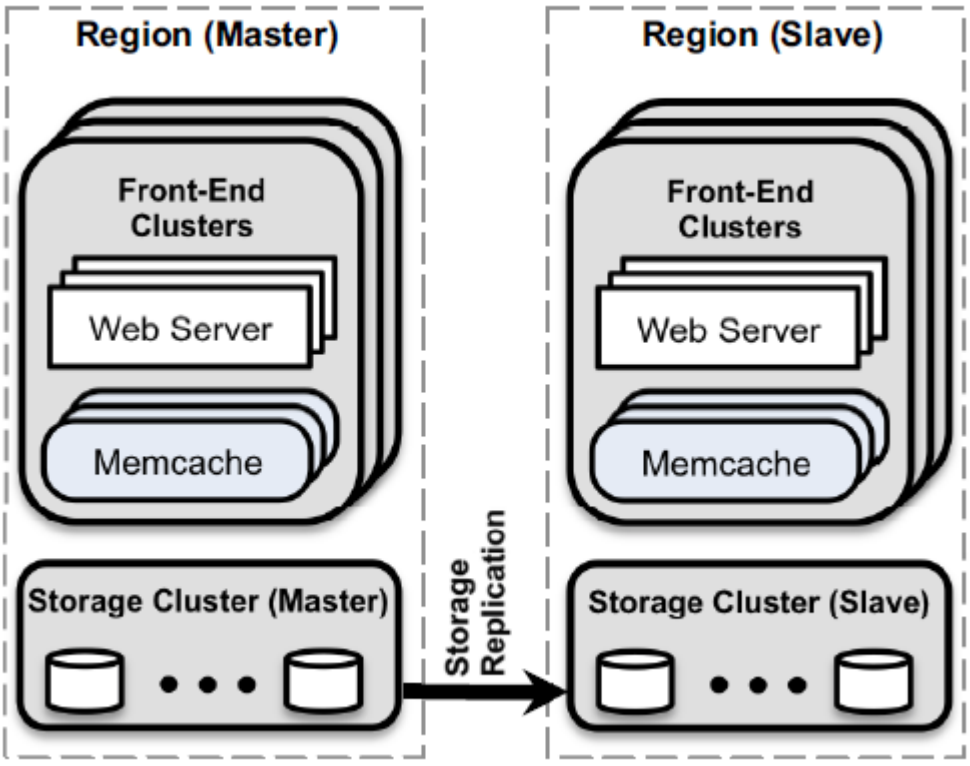


Figure 2.4: Memcache overall architecture. [2]

Some changes are implemented in order to scale out Memcache. First, latency and overhead are reduced by relying on UDP for get requests (which is the common case), while set and delete are performed over TCP. Furthermore, a new mechanism is introduced, leases, to address two problems: stale sets and thundering herds. A stale set occurs when a web server sets a value in Memcache that does not reflect the latest value that should be cached. This can occur when concurrent updates to Memcache get reordered. A thundering herd happens when a specific key undergoes heavy read and write activity. As the write activity repeatedly invalidates the recently set values, many reads default to the more costly path. Also, to accommodate differences between

workloads (access patterns, memory footprints and QoS requirements), a cluster's Memcached servers are partitioned into separate pools.

Moreover, performance optimizations were implemented, namely (1) allow automatic expansion of the hash table to avoid look-up times drifting to O(n), (2) make the server multi-threaded using a global lock to protect multiple data structures, and (3) giving each thread its own UDP port to reduce contention when sending replies and later spreading interrupt processing overhead, with the first two being implemented in the open source version. On top of these optimizations, an adaptive slab allocator was implemented that periodically re-balances slab assignments to match the current workload. It identifies slab classes as needing more memory if they are currently evicting items and if the next item to be evicted was used at least 20% more recently than the average of the least recently used items in other slab classes. If such a class is found, then the slab holding the least recently used item is freed and transferred to the needy class. Finally, a hybrid eviction scheme was introduced that relies on lazy eviction for most keys and proactively evicts short-lived keys when they expire. Short-lived items are placed into a circular buffer of linked lists (indexed by seconds until expiration) – called the Transient Item Cache – based on the expiration time of the item. Every second, all of the items in the bucket at the head of the buffer are evicted and the head advances by one.

Masstree

[12] introduces Masstree, which uses a combination of old and new techniques to achieve high performance. Lookups use no locks or interlocked instructions, and thus operate without invalidating shared cache lines and in parallel with most inserts and updates. Updates acquire only local locks on the tree nodes involved, allowing modifications to different parts of the tree to proceed in parallel. Masstree shares a single tree among all cores to avoid load imbalances that can occur in partitioned designs. The tree is a trie-like concatenation of B+-trees, and provides high performance even for long common key prefixes, an area in which other tree designs have trouble. Query time is dominated by the total DRAM fetch time of successive nodes during tree descent; to reduce this cost, Masstree uses a wide-fanout tree to reduce the tree depth, prefetches nodes from DRAM to overlap fetch latencies, and carefully lays out data in cache lines to reduce the amount of data needed per node. Operations are logged in batches for crash recovery and the tree is periodically checkpointed.

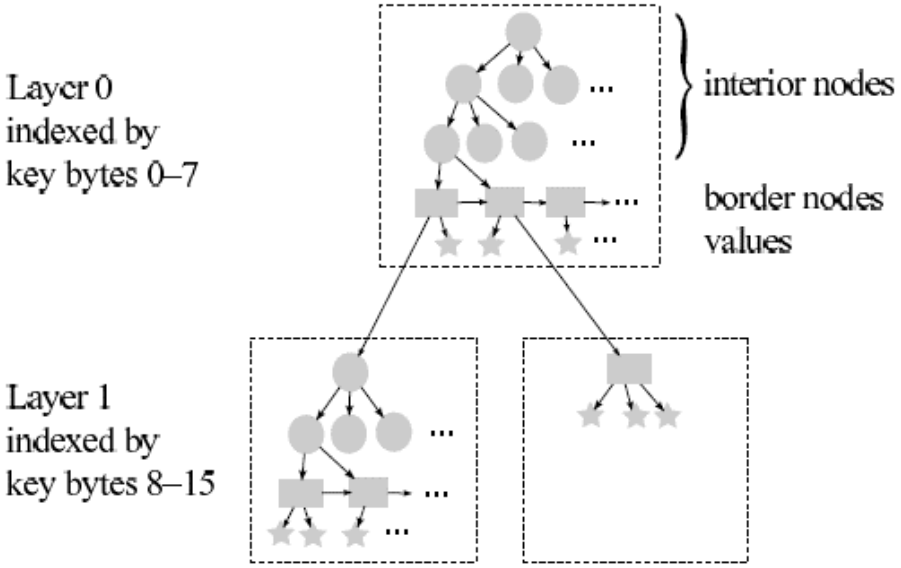


Figure 2.5: Masstree structure: layers of B^+- trees form a trie. [12]

```
struct interior_node:
    uint32_t version;
    uint8_t nkeys;
    uint64_t keyslice[15];
    node* child[16];
    interior_node* parent;

union link_or_value:
    node* next_layer;
    [opaque] value;
```

```
struct border_node:
    uint32_t version;
    uint8_t nremoved;
    uint8_t keylen[15];
    uint64_t permutation;
    uint64_t keyslice[15];
    link_or_value lv[15];
    border_node* next;
    border_node* prev;
    interior_node* parent;
    keysuffix_t keysuffixes;
```

Figure 2.6: Masstree node structures. [12]

2.3.2. RDMA-BASED OPTIMIZATIONS

The following optimizations make use of RDMA (**R**emote **D**irect **M**emory **A**ccess), which reduces end-to-end latency by enabling memory-to-memory data transfers over InfiniBand and Converged Ethernet fabrics [7]. RDMA requests are sent over reliable connections (also called queue pairs) and network failures are exposed as a terminated connection. Requests are sent directly to the NIC without involving the kernel and are serviced by the remote NIC without interrupting the CPU. A memory region must be registered with the NIC before it can be made available for remote access. During registration the NIC driver pins the pages in physical memory, stores the virtual to physical page mappings in a page table in the NIC and returns a region capability that the clients can use to access the region. When the NIC receives an RDMA request, it obtains the page table for the target region, maps the target offset and size into the corresponding physical pages and uses DMA to access the memory. Many

NICs guarantee that RDMA writes (but not reads) are performed in increasing address order [9].

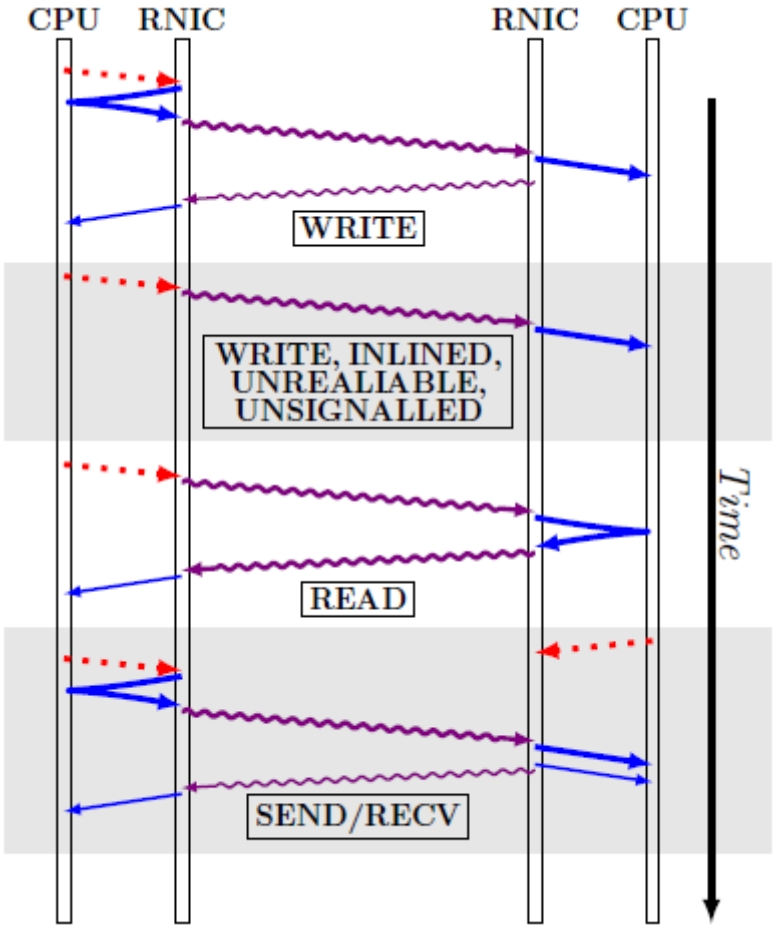


Figure 2.7: Steps in posting RDMA verbs. The dotted arrows are PCIe PIO operations. The solid, straight arrows are DMA operations: the thin ones are for writing the completion events. The thick wavy arrows are RDMA data packets and the thin ones are ACKs. [11]

soNUMA

soNUMA [7] is an architecture, programming model, and communication protocol for distributed, in-memory applications that reduces remote memory access latency to within a small factor (~4x) of local memory. soNUMA leverages two simple ideas to minimize latency. The first is to use a stateless request/reply protocol running over a NUMA memory fabric to drastically reduce or eliminate the network stack, complex NIC, and switch gear delays. The second is to integrate the protocol controller into the node's local coherence hierarchy, thus avoiding state replication and data movement across the slow PCI Express (PCIe) interface.

soNUMA's programming model, which allows for one-sided memory operations that access a partitioned global address space, inspired by RDMA, is provided by the RMC – a simple, hardwired, on-chip architectural block that services remote memory requests through locally cache-coherent interactions and interfaces directly with an on-die network interface. Each operation handled by the RMC is converted into a set of stateless request/reply exchanges between two nodes. The model is exposed through lightweight libraries, which also implement communication and

synchronization primitives in software. An overview of soNUMA and the RMC are illustrated below.

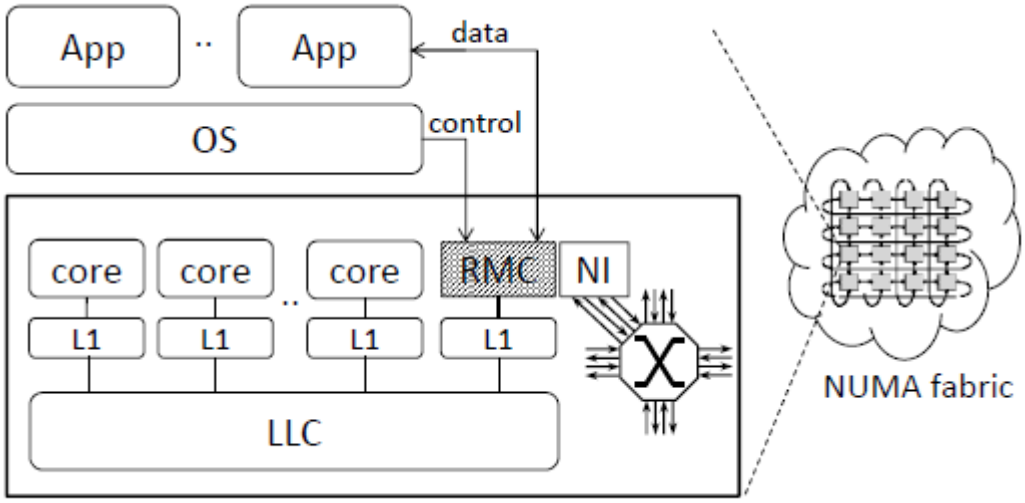


Figure 2.8: soNUMA overview. [7]

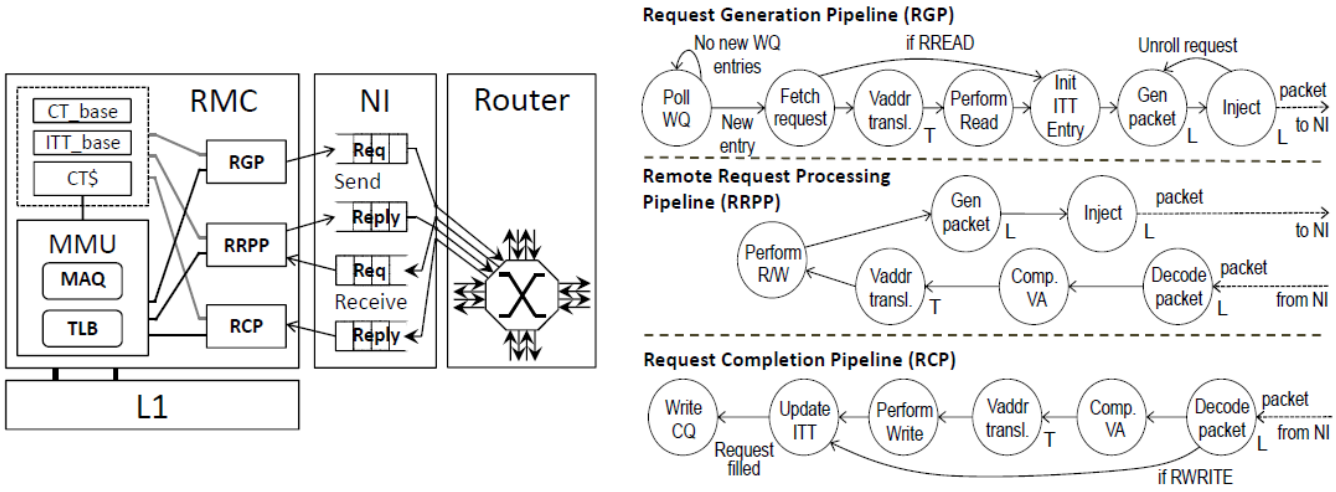


Figure 2.9: RMC inernal architecture and functional overview of the three pipelines. [7]

FaRM

[9] describes FaRM (**Fa**st **R**emote **M**emory), which is a main memory distributed computing platform that exploits RDMA to improve both latency and throughput by an order of magnitude relative to many state of the art main memory systems that use TCP/IP. FaRM exposes the memory of machines in the cluster as a shared address space. Applications can use transactions to allocate, read, write, and free objects in the address space with location transparency. It should be noted that FaRM is a more general-purpose distributed computing platform than a key-value store; key-value stores can be implemented on top of FaRM.

FaRM offers two mechanisms to improve performance where required with only localized changes to the code: lock-free reads that can be performed with a single

RDMA and are strictly serializable with transactions, and support for collocating objects and function shipping to allow applications to replace distributed transactions by optimized single machine transactions.

A new hashtable algorithm on top of FaRM is designed and implemented that combines hopscotch hashing with chaining and associativity to achieve high space efficiency while requiring a small number of RDMA reads for lookups: small object reads are performed with only 1.04 RDMA reads at 90% occupancy. Inserts, updates, and removes are optimized by taking advantage of FaRM's support for collocating related objects and shipping transactions.

Pilaf

In [13] Pilaf is presented, which is a distributed in-memory key-value store that leverages RDMA to achieve high throughput with low CPU overhead. It is argued that the sweet spot in the design space is to restrict the use of RDMA to read-only requests (namely GETs), while letting the server handle all other requests via traditional messaging. Thus, the class of memory access races that can occur is restricted to read-write races; clients might read inconsistent data while the server is concurrently modifying the same memory addresses. The overall architecture is illustrated below.

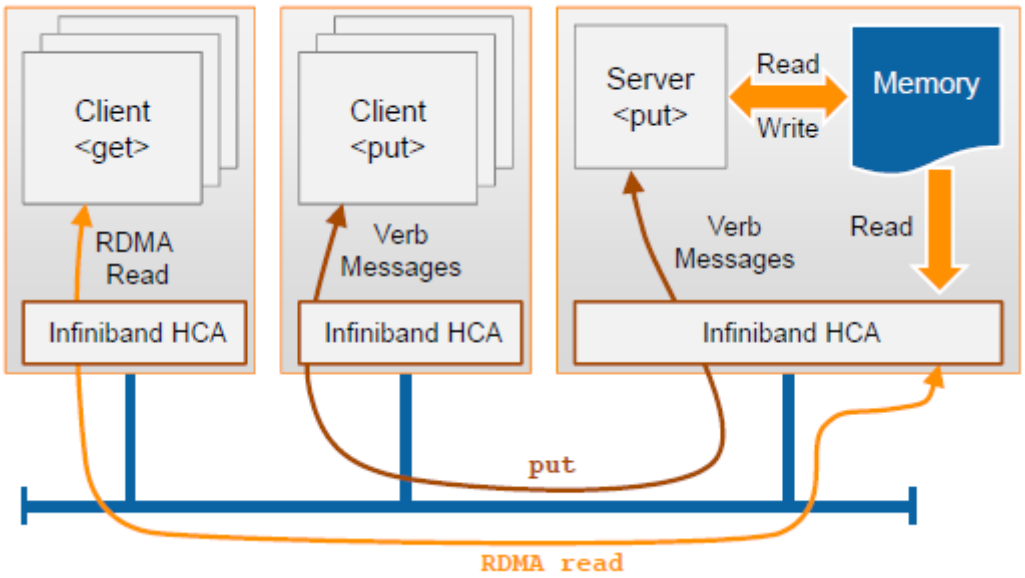


Figure 2.10: Pilaf's overall architecture. [13]

Pilaf uses self-verifying data structures to address read-write races between the server and the clients. A self-verifying data structure consists of checksummed root data objects as well as pointers whose values include a checksum covering the referenced memory area. Starting from a set of root objects with known memory locations, clients are guaranteed to traverse a server's self-verifying data structure correctly, because the checksums can detect any inconsistencies that arise due to concurrent memory writes done by the server. When a race is detected, clients simply retry the operation.

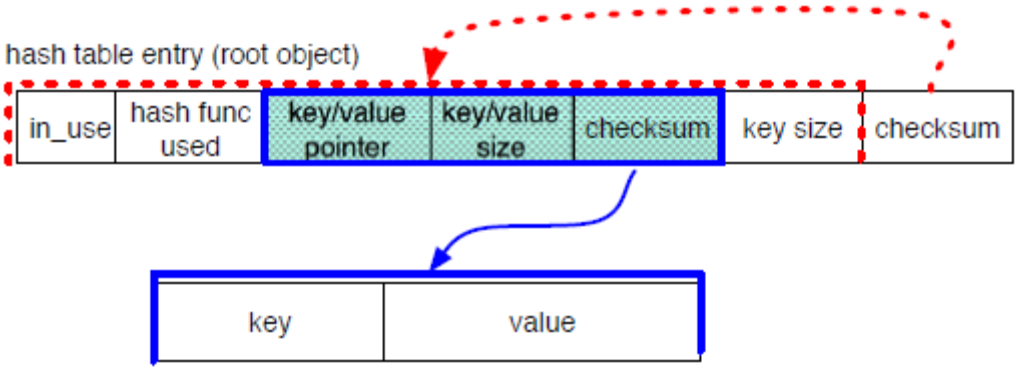


Figure 2.11: Self-verifying hash table structure. [13]

HERD

In [11] HERD is introduced, which, like FaRM and Pilaf, is a key-value cache that leverages RDMA features to deliver low latency and high throughput. However, in HERD, clients transmit their request to the server's memory using RDMA writes over an unreliable connection; RDMA reads sometimes require multiple round trips and, as such, are not adopted. Thus, HERD takes a hybrid approach, using both RDMA and messaging to best effect. The end result is throughput even higher than that of the RDMA-based systems while scaling to several hundred clients.

2.3.3. HARDWARE OPTIMIZATIONS

TSSP

In [3] an SoC architecture is proposed that implements the most latency- and throughput-critical - Memcached task, GET operations, in hardware. This design pairs a hardware GET processing engine and networking stack near the NIC, integrating both with a conventional CPU to handle less latency- and throughput-sensitive operations.

More specifically, the SoC has two memory controllers and a shared interconnect that includes both the processors and the I/O devices. The hardware accelerator can respond to a GET request without any software interaction, but all other request types and memory management are handled by software.

This design leverages several common characteristics of Memcached workloads. First, TSSP optimizes for GETs, as they vastly outnumber SETs and other request types in most Memcached deployments. Second, as Memcached is a best-effort cache, TSSP use UDP, a lighter-weight protocol than TCP that provides more relaxed packet delivery guarantees, for requests that will be handled by the hardware GET accelerator. Requests that must be reliable (e.g., SETs), are transmitted over TCP and will be executed by software, since it is found that simply switching all traffic from TCP to UDP is insufficient to obtain the power-efficiency that TSSP can achieve. Lastly, Memcached's lookup structure (hash table) is split from the key and value storage (slab-allocated memory) to allow the hardware to efficiently perform look-ups, while software handles the more complex memory allocation and management.

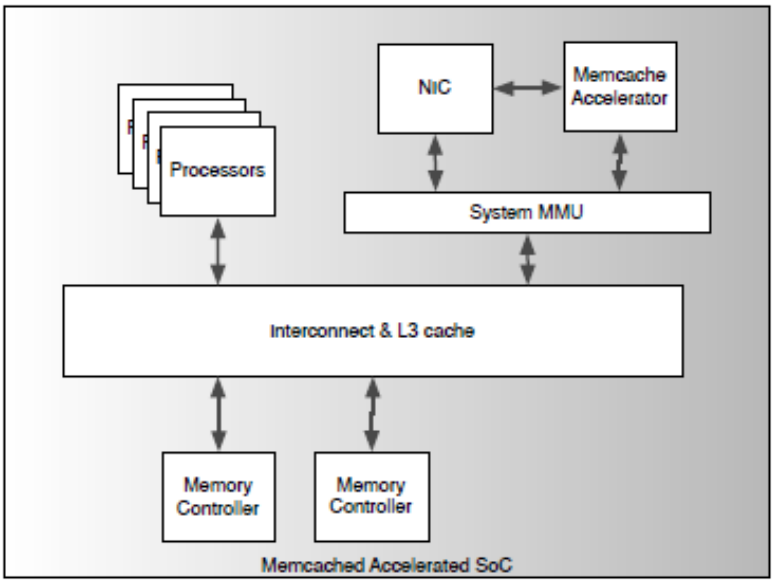


Figure 2.12: TSSP architecture. [3]

Figure 2.12 shows in detail the NIC and the Memcached hardware unit. The flow affinitizer, which normally routes between several hardware queues destined for different cores based on IP address, port, and protocol, has been modified to allow the Memcached accelerator to be a target. Similarly, on the transmit path, the NIC has been modified to allow packets to both be transmitted through the normal DMA descriptor rings as well as from the Memcached hardware. After a packet is routed to the Memcached hardware, it is passed to a UDP Offload Engine which decodes the packet and places the Memcached payload into a buffer for processing. This design requires few NIC modifications and leverages flow affinity features already present in Gigabit NICs to route traffic across the accelerator (UDP on Memcached port) and software (TCP on Memcached port).

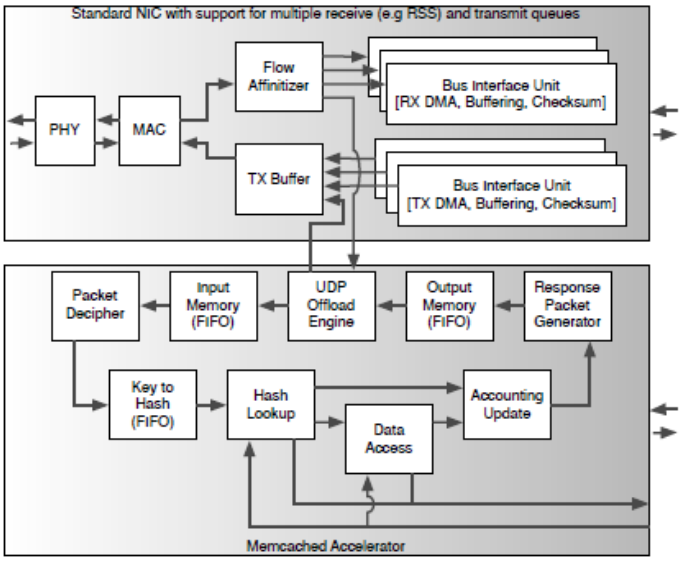


Figure 2.13: NIC and GET accelerator details. [3]

After the traffic is routed, the Memcached hardware deciphers the request and passes control signals along with the key to a hardware hash table implementation. Since one of the design goals is to allow the hardware to respond to GETs without software involvement, the hardware must be able to perform hash-table lookups. This

hash table must be hardware-traversable, so a design is chosen in which the hardware manages all accesses to the hash table (including on behalf of cores) to avoid expensive synchronization between the hardware and software. The hash table and slab memory management scheme are illustrated below.

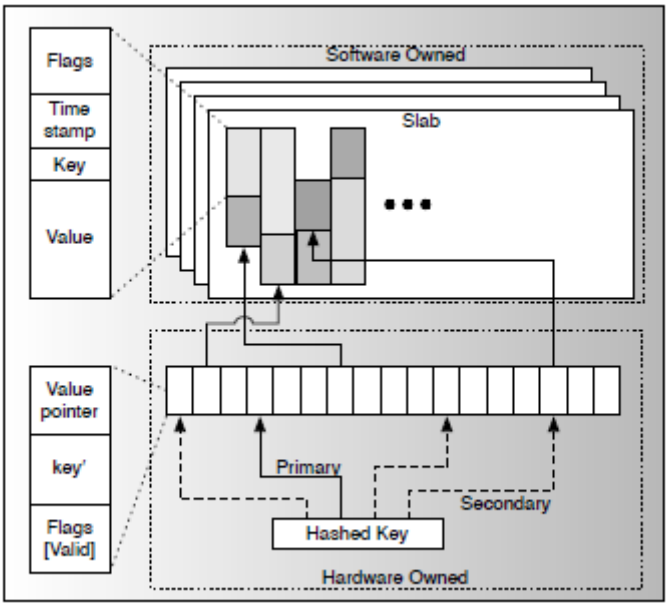


Figure 2.14: TSSP – Hardware and Software data structures. [3]

FPGA-based optimizations

[4] leverages the FPGA to implement a Memcached appliance, thus completely converting the Memcached software to an FPGA implementation. The FPGA is leveraged to provide specialized logic to implement the base functionality of the Memcached server, consisting of accepting requests, finding the requested key-value pair and performing the requested get or set operation.

An FPGA Memcached appliance can ensure tight integration between the networking, compute and memory while removing software overhead. Also, power consumption drops significantly, since FPGAs hold that advantage against a traditional CPU. This design provides performance on-par with baseline servers, while consuming only 9% of the power of the baseline. Scaled at the data center level, performance-per-dollar improves substantially while improving energy efficiency by 3.2X to 10.9X. The overall architecture is illustrated below.

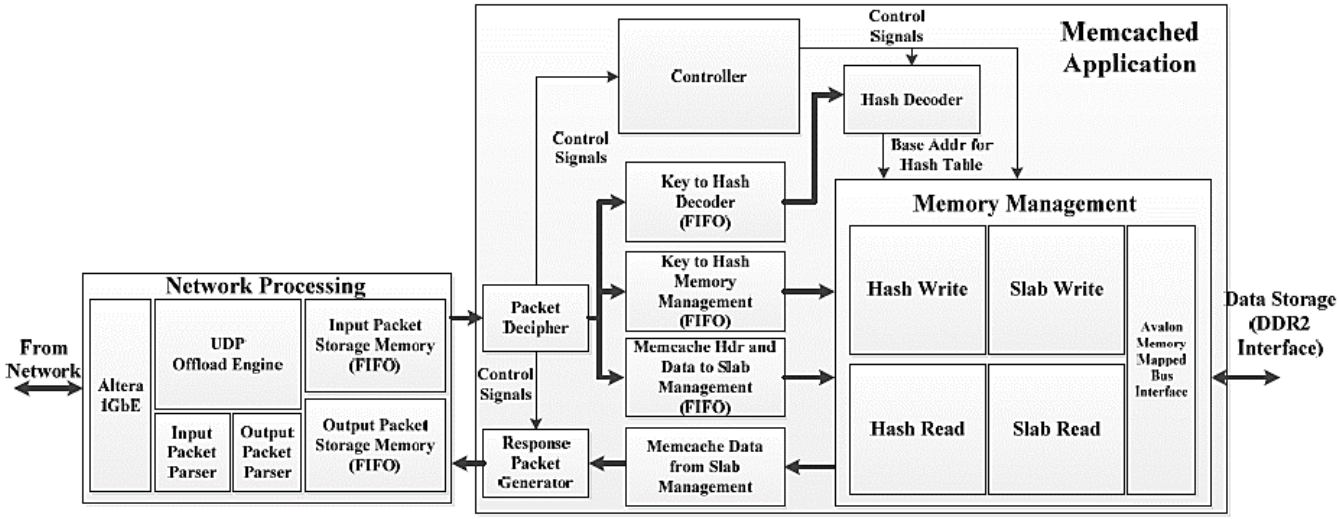


Figure 2.15: Overall FPGA Memcached appliance architecture. [4]

In [5] another Memcached implementation on an FPGA is introduced, which enables the implementation of customized integrated circuits through programming rather than designing and manufacturing custom chips. The circuit itself is designed as a custom-tailored pipeline that fully extracts the parallelism in the application. The prototype demonstrates full line-rate processing, handling up to 13 million of requests per second (RPS), while providing a round trip latency below 4.5 microseconds (us). Power consumption of the FPGA and its subsystem is around 50Watts (W), whereby the FPGA itself consumes less than 15W.

The dataflow architecture (illustrated below) consists of five key processing stages, namely network interface, request parser, hash table, value store and response formatter. Packets are received on the board on a 10Gbps Ethernet interface and streamed back-to-back through these processing stages before being transmitted back into the network. The first stage, the network interface, handles all related processing to Ethernet and includes a full UDP and TCP offload engine. Only the Memcached requests themselves, bar all additional headers, are passed to the request parser together with a connection identifier. The request parser analyzes the Memcached packets to extract key, value, and meta-data information and generates an opcode for the currently supported subset of operations. Currently only ASCII and binary protocols are supported, however further protocols can easily be added with no impact on performance. Independent of the incoming protocol, the request parser normalizes all packets to the format of the standard interface. This information is then passed to the hash table. The hash table's responsibility is to produce an index into the value store for any incoming key. The value store simply supports read or write operations on a corresponding area of memory as defined by the opcode. For SET operations, the presented value is written into memory. In case of GET operations, the retrieved value is added to the packet information as it streams to the response formatter. Finally, the response formatter supports formatting of responses according to the supported protocols.

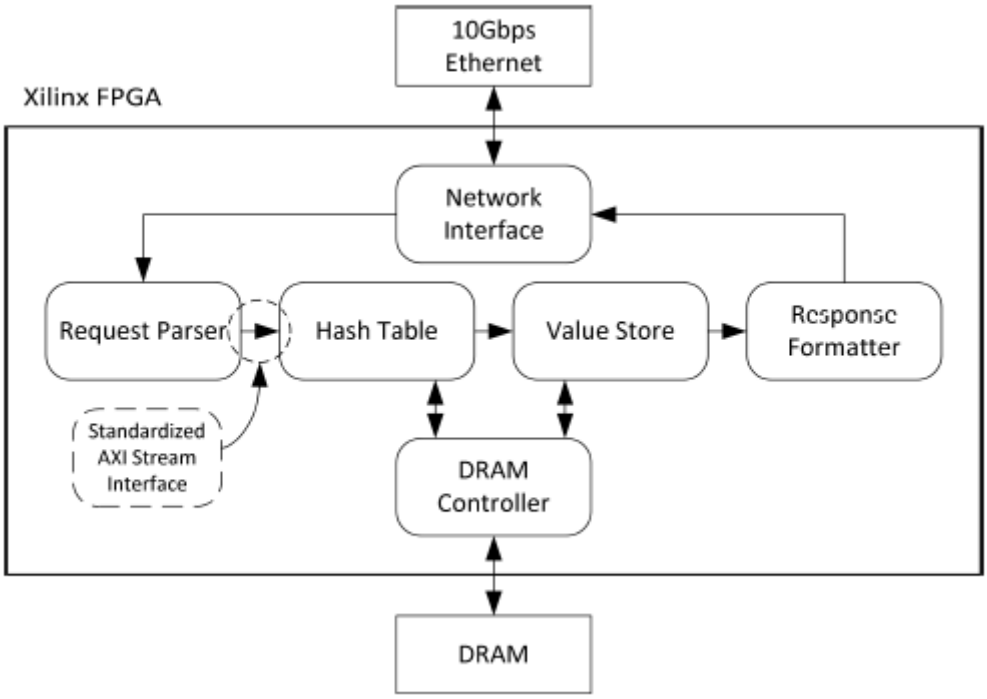


Figure 2.16: Memcached dataflow pipeline. [5]

[6] proposes an FPGA-based in-line accelerator for Memcached. In a conventional server architecture, the NIC's main functionality is to copy the incoming and outgoing packets to/from the memory system. In a conventional server application, the CPU expects packets to be available in the memory after which it will parse the request packets and extract the relevant fields that form the arguments for the request, process the request by performing computation and potentially modifying global data structures, and finally create response packet(s) if required. If the application lives in the user space, additional overhead is imposed to copy the packet data across various buffers/privilege-levels including NIC FIFO, kernel network stack, application level buffers. While zero-copy and user-space networking can be used to minimize the user/kernel distinction at the cost of blurring the protection/privilege separation, such approaches still maintain a strong distinction between the NIC buffer and the processor memory.

An in-line accelerator architecture redraws how computation is done by combining the NIC and the inline accelerator which receives the incoming request packets, processes the request packets by accessing and modifying global application data-structures through a coherent port without involving the CPU, and finally sends the response packets if required. The in-line accelerator processes packets speculatively, assuming the packet is a common case the accelerator can handle. If the accelerator determines it cannot handle the packet, it ”bails out” from the fast path by rolling back what the accelerator did speculatively, and passes the request packet to the CPU cores via the conventional NIC interface.

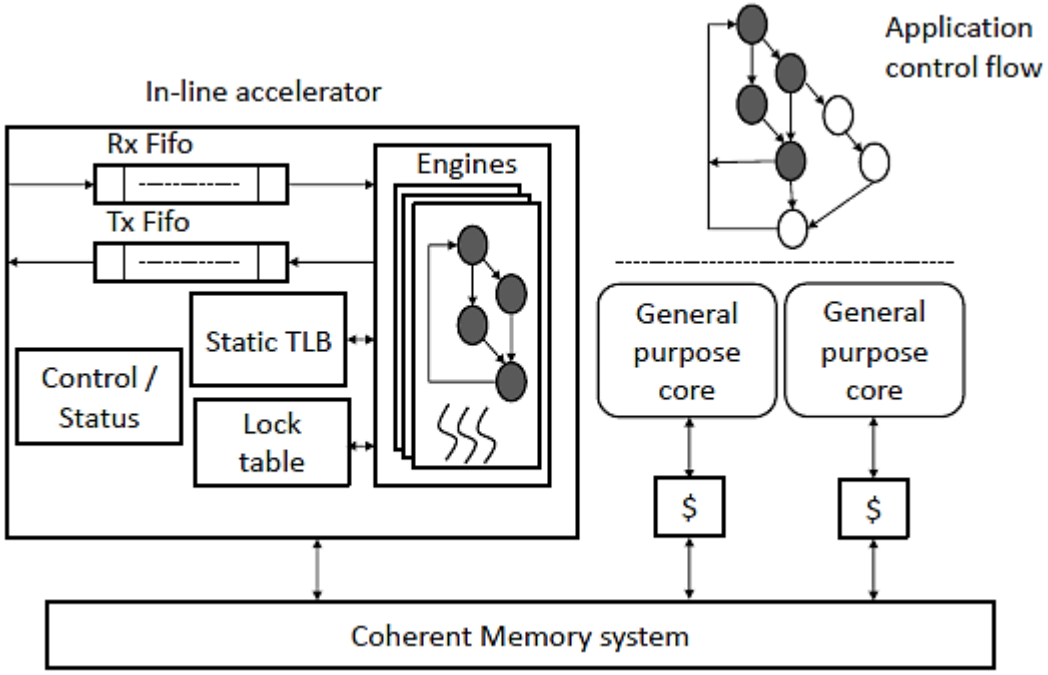


Figure 2.17: In-line accelerator architecture. [6]

The aforementioned design was implemented in a Xilinx Virtex-5 TX240T FPGA and was capable of 583K gets/sec, while consuming less than 2W of power for 64 bytes objects, 6% of the FPGA's LUTs and 1% of the FPGA registers.

Integrated 3D-Stacked Server Designs

In [8] a new approach is adopted, in which density is included as a primary design constraint, rather than solely overall energy consumption and performance. With that in mind, two integrated 3D-stacked architectures are proposed, called Mercury and Iridium. With Mercury, low-power ARM Cortex-A7 cores are tightly coupled with NICs and DRAM, while maintaining high bandwidth and low latency. More specifically, Mercury is able to improve density by 2.9X, power efficiency by 4.9X, throughput (TPS, transactions per second) by 10X and TPS/GB by 3.5X.

Also, to address Memcached servers that require higher density with similar latency targets, but are accessed at much lower rates, Iridium is introduced, a Flash based version of Mercury that further increases density at the expense of throughput while still meeting latency requirements. By replacing the DRAM with NAND Flash, density is improved by 14X, TPS by 5.2X and power efficiency by 2.4X, while still mainiting latency requirements for a bulk of requests. This comes at the expense of 2.8X less TPS/GB due to the much higher density.

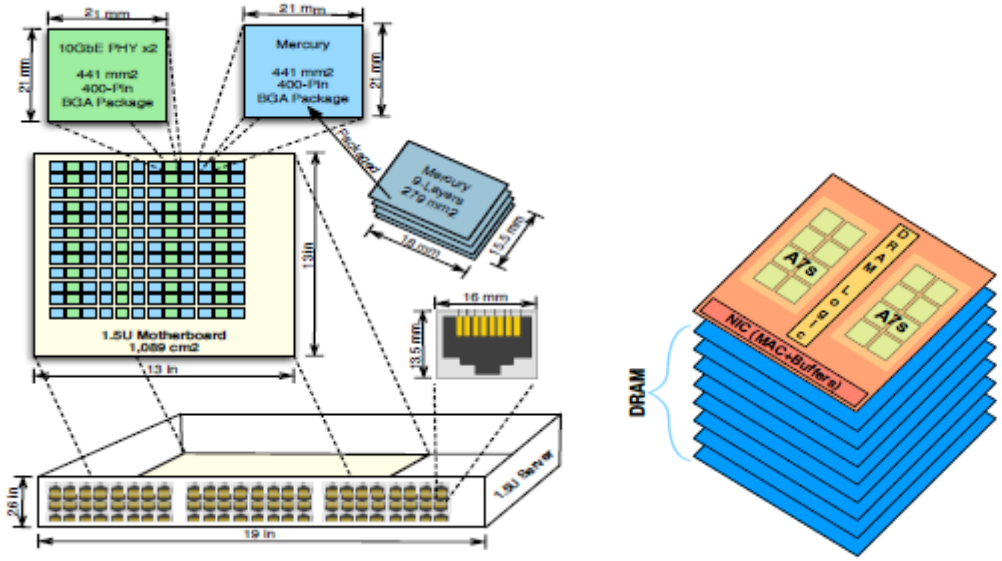


Figure 2.18: Left: 1.5U server with 96 Mercury stacks. Right: The 3D-stacked Mercury architecture. [8]

Implementation on the TILEPro64 Architecture

In [15] Memcached is implemented on the Tilera TILEPro64 64-core CPU. By using the UDP protocol for reads and altering the execution model of Memcached, as illustrated below, the tuned version of Memcached can achieve at least 57% higher throughput on the 64-core Tilera TILEPro64 than low-power x86 servers at comparable latency. When taking power and node integration into account as well, a TILEPro64-based S2Q server with 8 processors handles at least twice as many transactions per second per Watt as the x86-based servers with the same memory footprint. The main reasons for this advantage are the elimination or parallelization of serializing bottlenecks using the on-chip network; and the allocation of different cores to different functions such as kernel networking stack and application modules. It should be noted that the TILEPro64 exhibited near-linear throughput scaling with the number of cores, up to 46 UDP cores.

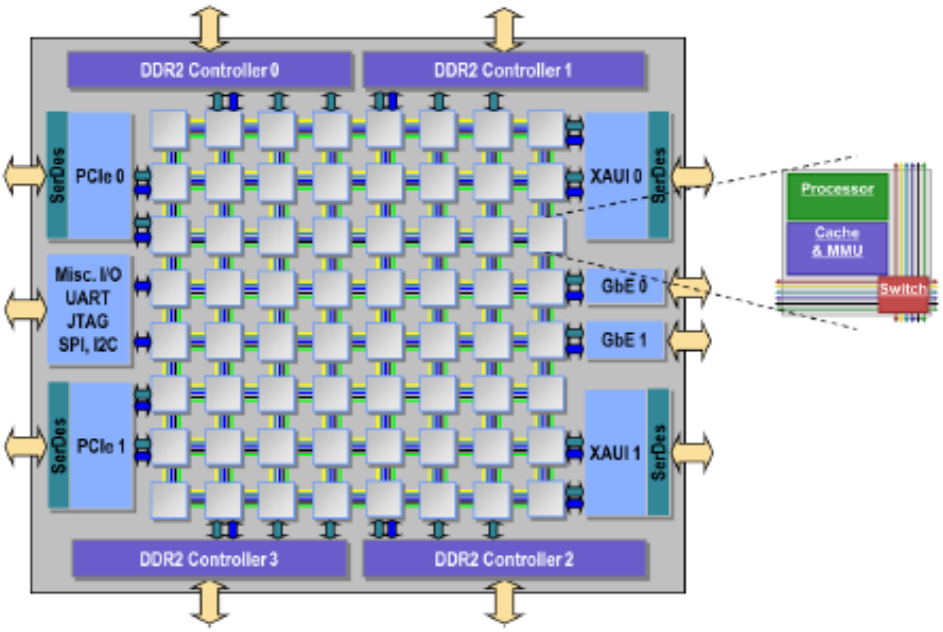


Figure 2.19: High level overview of the Tilera TILEPro64 architecture. [15]

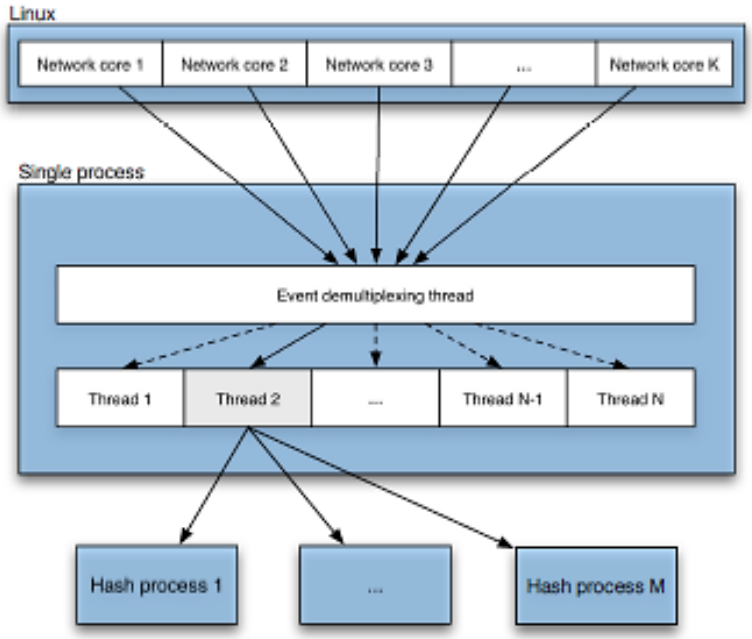


Figure 2.20: Execution model of Memcached on TILEPro64. [15]

3. IMPLEMENTATION IN X86 AND ARM-BASED ARCHITECTURES

3.1. MEMCACHED AND MEMC3

3.1.1. MEMCACHED

3.1.1.1. MEMCACHED COMMANDS

Memcached provides a simple set of operations: set, get and delete [2], among others, with the first two being the most important. In particular, the get command is the most common, since it has been observed that users consume an order of magnitude more content than they create. This behavior results in a workload dominated by fetching data [2].

A get command will retrieve the value associated with the user-specified key, if its located in the Memcached server. If it is not found, it is up to the user to determine where to obtain the proper value (typically a miss will then lead to a database lookup and/or recomputation to determine the appropriate value). A key, which can potentially be an ASCII string up to 250 characters long, is sent to the server in a message including the command (get), the key length and any optional message flags [4].

A get performs the following steps: 1. The request is received at the network interface and is sent to the CPU. 2. The Memcached server will read the data out of the request packet to identify the key. 3. The server performs a hash on the key value to translate the key into a fixed 32-bit value. 4. The value is used to index into a hash table that stores the key-value pairs. 5. If the key is found, its value data is accessed and prepared to be sent back to the client in a response message. 6. The entry corresponding to the key is also promoted in a doubly linked list that is used to perform least recently used (LRU) replacement if the Memcached server is full. 7. The server either sends out a reply to the client with the key-value pair, or a message indicating that the key was not found [4].

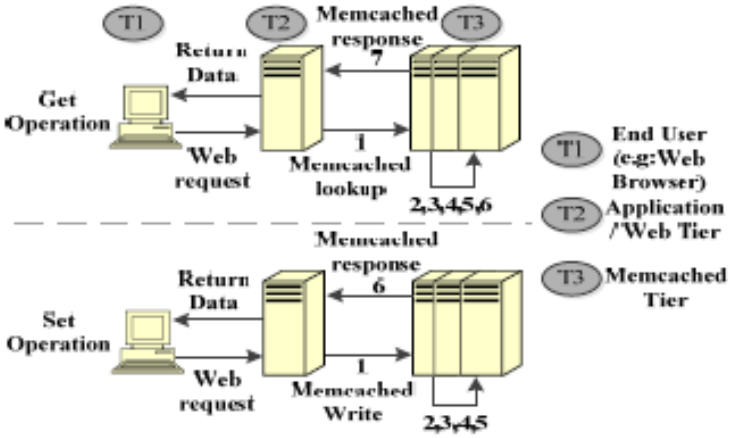


Figure 3.1: Memcached architectural diagram and use case. [4]

A set will write the specified key-value pair into the Memcached server's storage. Values are typically small objects, often a few hundred bytes large. To handle memory management, Memcached uses slab allocation. In slab allocation, Memcached allocates a large chunk of memory and breaks it up into smaller segments of a fixed size according to the slab class' size. This method of allocation reduces the overhead of dynamically allocating and deallocating many small objects. Memory is therefore handled in fixed sizes, with values stored in the smallest slab class that will accommodate the size of value. (Thus, there may be some internal fragmentation per object.) When storing a new value, the LRU list for the slab class is checked to see if the last element can be evicted. If there are no free segments within a slab class, a new slab is allocated if there is free memory [4].

A set performs the following steps: 1. The request is received at the network interface and is sent to the CPU. 2. The server will read the data from the packet to identify the key, value, flags, and total message size. 3. The server then requests a slot from the correct slab memory class to store the key-value pair. The item is promoted to the most recently used position of the slab's LRU list. 4. After copying the data into the slab element, the server performs a hash on the key to determine the hash bucket to store the data. 5. The data is written to the head of the hash bucket. 6. A reply is sent back to the client to indicate the request is completed [4].

3.1.1.2. HASH TABLE

A Memcached cluster provides a lightweight, distributed hash table for storing small objects (up to 1 MB), exposing a simple set/get interface. Each object's key is used to determine which individual Memcached server within the cluster will store the object. Typically, a hash function is chosen to balance keys evenly across the cluster. Individual Memcached servers do not communicate with each other, as each server is responsible for its own independent range of keys. Because the servers do not interact, the performance of a single Memcached server can be used to generalize the behavior of an entire cluster [4].

The hash table data structure is an array of buckets. The array size (k) is always a power of 2 to make finding the correct bucket quicker by taking the value of 2^k -1 and using it as a hash mask. Executing a bit-wise AND (e.g. hash_value & hash_mask) quickly determines the bucket that contains the hash value. The buckets are constructed as single linked-lists of cache items that are NULL-terminated [40].

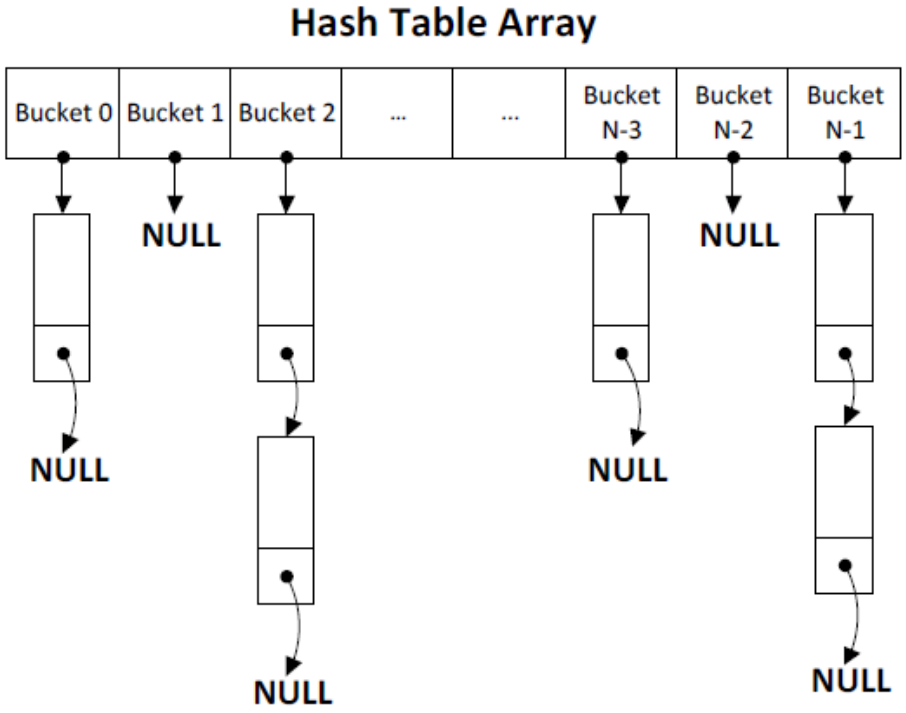


Figure 3.2: Data structure of hash table used to lookup cache items. [40]

3.1.1.3. MEMORY MANAGEMENT

Memcached employs a slab allocator to manage memory. The allocator organizes memory into slab classes, each of which contains pre-allocated, uniformly sized chunks of memory. Memcached stores items in the smallest possible slab class that can fit the item's metadata, key and value. Slab classes start at 64 bytes and exponentially increase in size by a factor of 1.07 up to 1 MB, aligned on 4-byte boundaries. Each slab class maintains a free-list of available chunks and requests more memory in 1MB slabs when its free-list is empty. Once a Memcached server can no longer allocate free memory, storage for new items is done by evicting the least recently used (LRU) item within that slab class. When workloads change, the original memory allocated to each slab class may no longer be enough resulting in poor hit rates [2].

3.1.2. MEMC3

MemC3 (**Mem**cached with **C**LOCK and **C**oncurrent **C**uckoo Hashing) has already been mentioned in chapter 2. It features several software optimizations with regards to Memcached, which will now be presented. First, architectural features can hide memory access latencies and provide performance improvements. In particular, a new hash table design exploits CPU cache locality to minimize the number of memory fetches required to complete any given operation and it exploits instruction-level and memory-level parallelism to overlap those fetches when they cannot be avoided.

Second, MemC3's design also leverages workload characteristics. Since many Memcached workloads are predominately reads Memcached's exclusive, global locking is replaced with an optimistic locking scheme targeted at the common case. Furthermore, many important Memcached workloads target very small objects, so per-object overheads have a significant impact on memory efficiency. For example, Memcached's strict LRU cache replacement requires significant metadata—often more space than the object itself occupies; in MemC3, a compact CLOCK-based approximation is used instead.

Lastly, MemC3 introduces a novel hashing scheme called optimistic cuckoo hashing. Conventional cuckoo hashing achieves space efficiency, but is unfriendly for concurrent operations. Optimistic cuckoo hashing achieves high memory efficiency (e.g., 95% table occupancy); allows multiple readers and a single writer to concurrently access the hash table; and keeps hash table operations cache-friendly. The idea behind finding the cuckoo path is illustrated below [10].

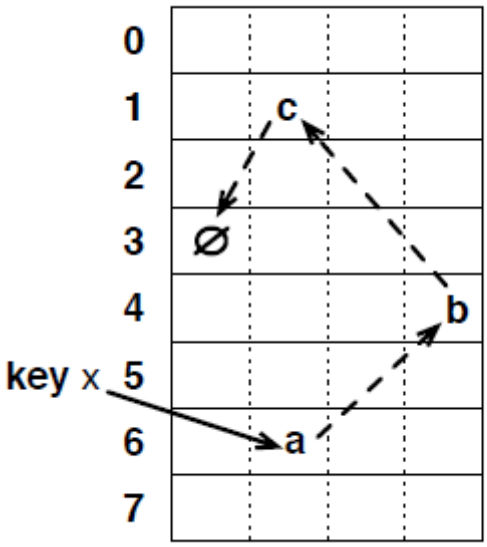


Figure 3.3: Cuckoo path. Ø represents an empty slot. [10]

3.2. BENCHMARKS

3.2.1. DATA CACHING BENCHMARK

This benchmark was developed at PARSA (**PA**rallel **S**ystems **A**rchitecture) Lab EPFL (**É**cole **P**olytechnique **F**édérale de **L**ausanne) and is included in CloudSuite, which is a benchmark suite for cloud services. The Data Caching Benchmark uses the Memcached data caching server, simulating the behavior of a Twitter caching server using the twitter dataset. The metric of interest is throughput expressed as the number of requests served per second. The workload assumes strict quality of service guaranties [41].

A workload parameter that could be changed for the purposes of the experiments was the distribution of the keys in the dataset. The default distribution is uniform. However, the benchmark was modified for the purposes of the experiments that are presented later in this chapter to support a skewed (Zipf) distribution, with the user being able to input the value of the distribution's skewness (theta) parameter ($0 \leq \theta \leq 1$). As such, a comparison between uniform and skewed distribution was made possible. In skewed, theta was picked at 0.99, which is the standard skeweness for YCSB (Yahoo Cloud Benchmark). The user can also change the packets' inter-arrival distribution (constant, exponential).

Another workload parameter that was of critical importance to the experiments was the GET/SET ratio of the requests. In most applications, GETs tend to severely outnumber SETs. As such, 2 configurations were chosen, one with 0.9 and one with 0.5 ratio, respectively.

It should be noted that most parameters that define a dataset could not be altered. The original dataset consumes 300MB of server memory, while the recommended scaled dataset requires around 10GB of main memory dedicated to the Memcached server (scaling factor of 30) [41]. Parameters such as key size, value size or number of records cannot be changed.

The typical procedure followed for running this benchmark is, after having started up the server, the client uses the option –j to "warm up" the server. The server notifies the client when the warmup is over. The benchmark can then be run with the number of TCP connections, worker threads, maximum throughput and other parameters as desired.

3.2.2. MUTILATE

The second benchmark used in the experiments is Mutilate. Mutilate is a Memcached load generator designed for high request rates, good tail-latency measurements, and realistic request stream generation. Mutilate reports the latency (average, minimum, and various percentiles) for get and set commands, as well as achieved QPS and network performance [42].

In contrast with the Data Caching Benchmark, Mutilate allows for the alteration of parameters regarding the dataset. Specifically, the number of records, key and value size had their values changed in order for the experiments to be conducted using datasets with different characteristics. Furthermore, the GET/SET ratio could also be manipulated to provide a different mixture of requests. However, Mutilate does not allow for change in key distribution (although the user can choose the packet inter-arrival distribution, which is automatically adjusted to match the specified throughput value).

The Mutilate benchmark features a list of options which can be used in order to fine-tune an experiment. First, it allows for several agents to be deployed in order to load the Memcached server or server cluster. Second, the experiment can be customized with respect to duration and target throughput. Last, it offers varying levels of verbosity, allowing the user to adjust the level of detail to which he is exposed, while granting debugging potential.

Mutilate offers a few advanced options, as well. Mutilate has a warmup option, for which the warmup time has to be specified, unlike the previous benchmark, in which the server notifies the client that the warmup has been completed. Like the Data Caching Benchmark, Mutilate allows the user to specify the number of connections per server to be established and the maximum depth to pipeline requests. Also, it lets the user assign threads to server in round-robin fashion, as well as customize parameters regarding the transmissions, such as skip some if the previous requests are late, or enforce a minimum time delay between requests. Moreover, Mutilate supports two advanced modes: a) the search mode (--search=*N*:*X*), in which Mutilate searches for the throughput where an *N*-order statistic is under *X* us; e.g. --search 95:1000 means find QPS where 95% of requests are faster than 1000us) and b) the scan mode (--scan=*min*:*max*:*step*), in which Mutilate scans latencies across QPS (**Q**ueries **P**er **S**econd) rates from *min* value to *max* value at *step* intervals.

It should be mentioned that Mutilate offers some agent-mode options. Using these options, the user can run a client in agent mode, enlist a remote agent and adjust the share of QPS of a specific client. It is also possible to define the number of the master client's connections per server, explicitly set its QPS (which is spread across threads and connections) and its connection depth.

The table that follows compares the two benchmarks summarily.

Benchmark	Connection parameters customization	Workload parameters customization	Key distribution choice	Packet Inter-arrival distribution choice	Special Modes
Data Caching Benchmark	✓	✗	✓ (uniform, zipf)	✓ (constant, exponential)	✗
Mutilate	✓	✓	✗	✓ (constant, uniform, normal, generalized pareto, generalized extreme value, exponential)	✓

Table 3.1: Data Caching Benchmark/Mutilate comparison

3.3. EXPERIMENTAL RESULTS

3.3.1. X86 IMPLEMENTATION

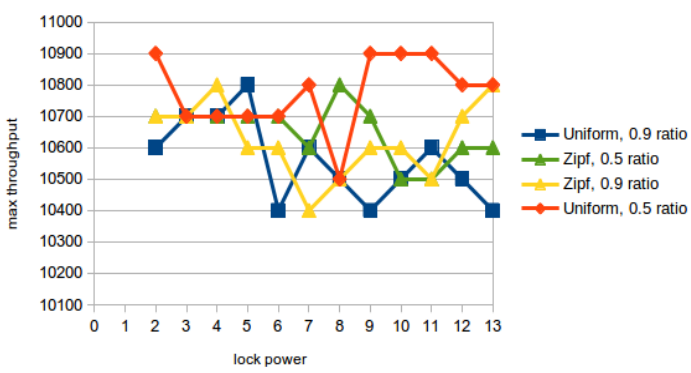
The graphs that follow are the results of the experiments conducted on a server with x86 architecture. Measurements were taken regarding the maximum throughput with respect to lock power and the number of server threads while latency remained under 10 ms (QoS agreement), as well as regarding latency (both the average value and the 95th percentile) with respect to a target throughput. The client is also an x86 machine. It should be noted that both the server and the client are virtual machines (managed via KVM) and are connected via a virtual network, thus eliminating any unrelated network traffic.

In the experiments where throughput is measured with respect to lock power (the number of locks for Memcached), the client sets up 100 TCP connections and utilized 2 threads to communicate with the x86-based Memcached server. In the experiments conducted to determine the system's maximum throughput and to measure latency with respect to maximum throughput, 3 different setups were used on the x86 system: a) Memcached with 100 client/server connections and 2 client threads, b) Memcached with 1 client/server connection and 1 client thread and c) MemC3 with 1 client/server connection and 1 client thread. In every case, the memory reserved for Memcached is 1 GB.

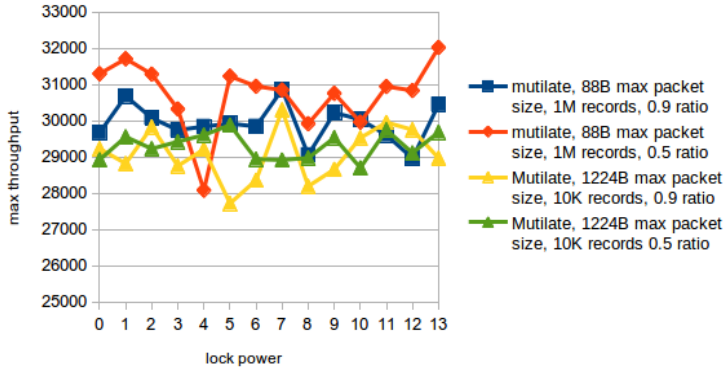
Virtualization tool	Number of threads	Memory	OS	NIC
KVM	24	8 GBs	Fedora release 25	Ethernet (Virtual)

Table 3.2: x86 server specifications

Maximum Throughput vs Lock Power



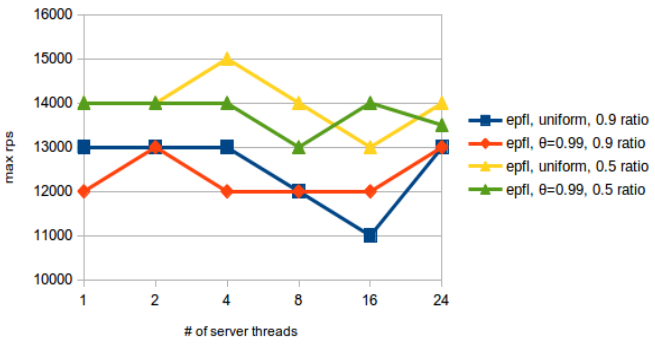
(a)



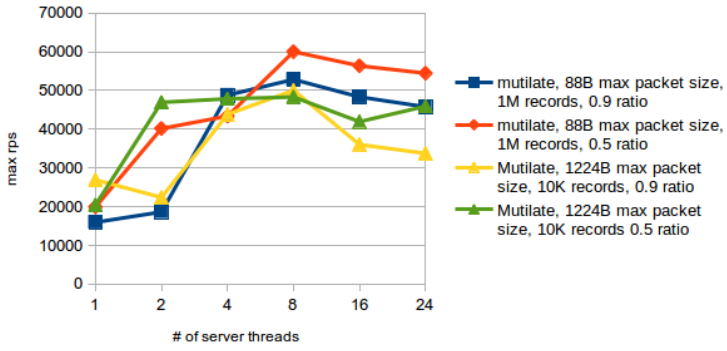
(b)

Figure 3.4: (a) Memcached, Data Caching Benchmark, 100 connections, 2 worker threads. (b) Memcached, Mutilate, 100 connections, 2 worker threads.

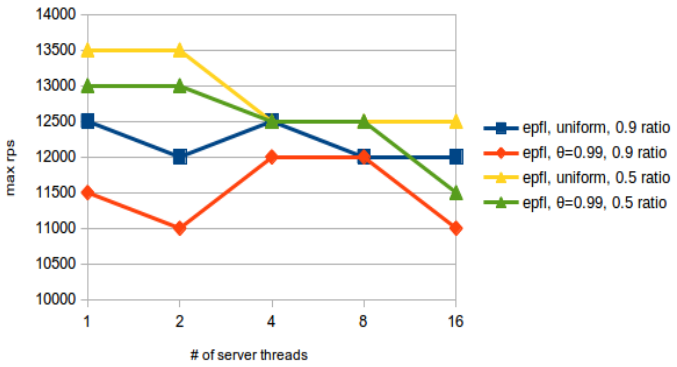
Maximum Throughput vs Server Threads



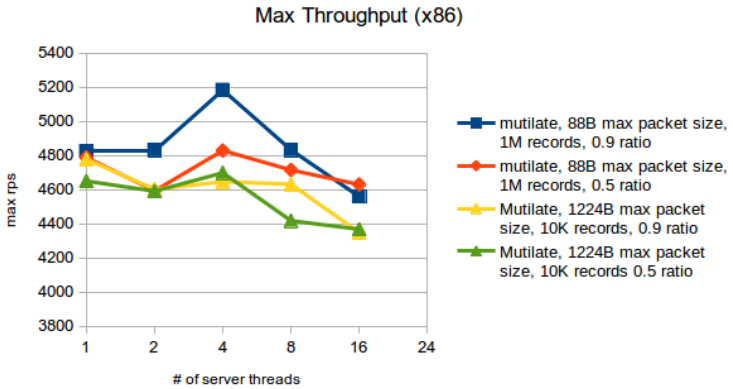
(a)



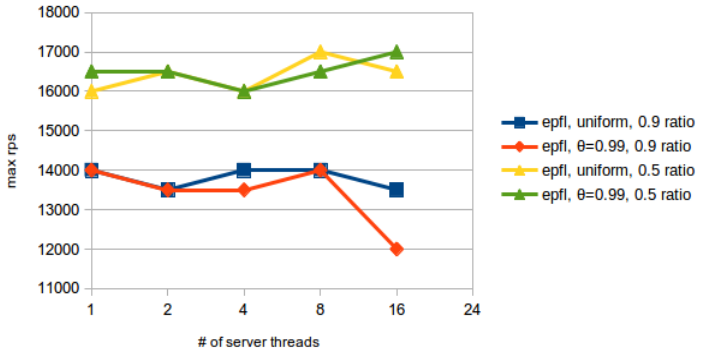
(b)



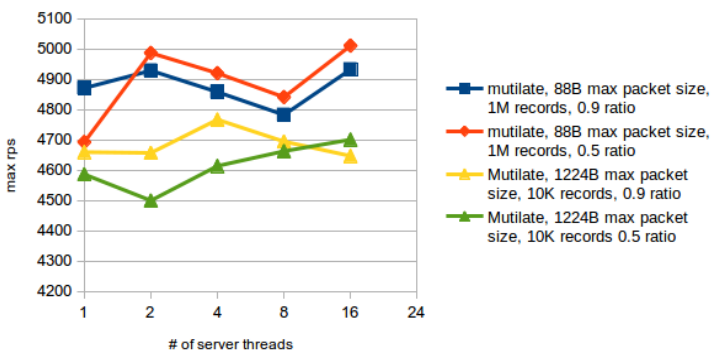
(c)



(d)



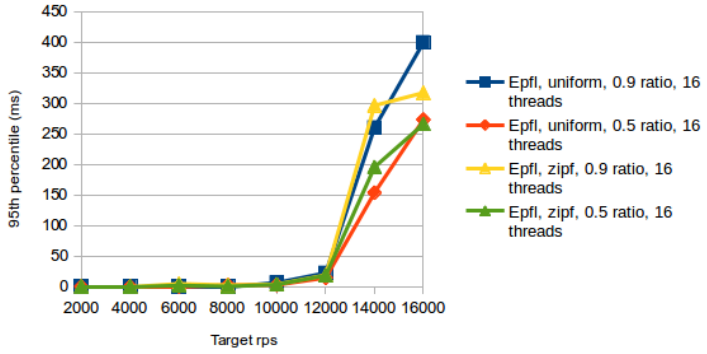
(e)



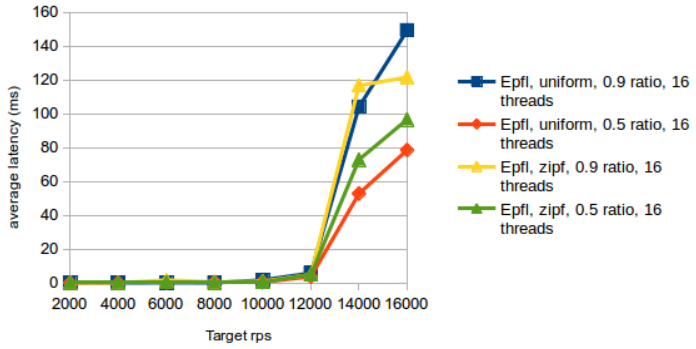
(f)

Figure 3.5: (a) Memcached, Data Caching Benchmark, 100 connections, 2 worker threads. (b) Memcached, Mutilate, 100 connections, 2 worker threads. (c) Memcached, Data Caching Benchmark, 1 connection, 1 worker thread. (d) Memcached, Mutilate, 1 connection, 1 worker thread. (e) MemC3, Data Caching Benchmark, 1 connection, 1 worker thread. (f) MemC3, Mutilate, 1 connection, 1 worker thread

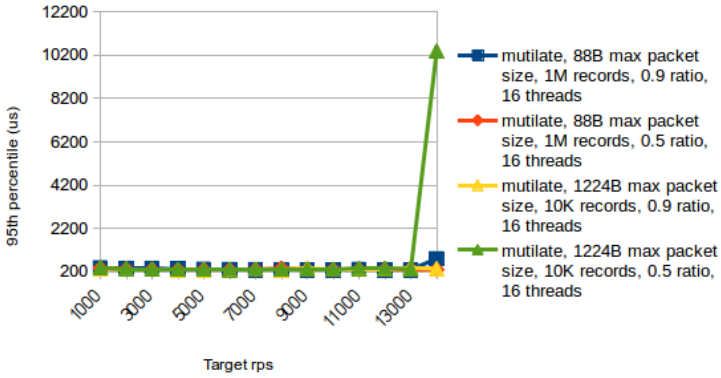
Latency vs Maximum Throughput



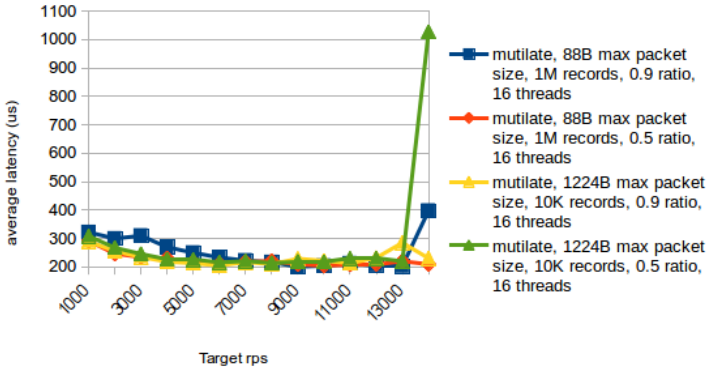
(a)



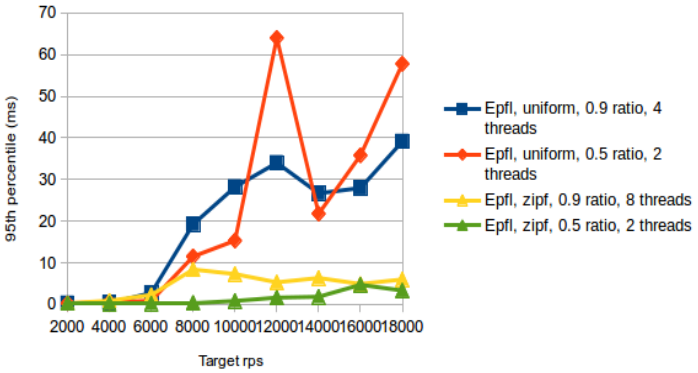
(b)



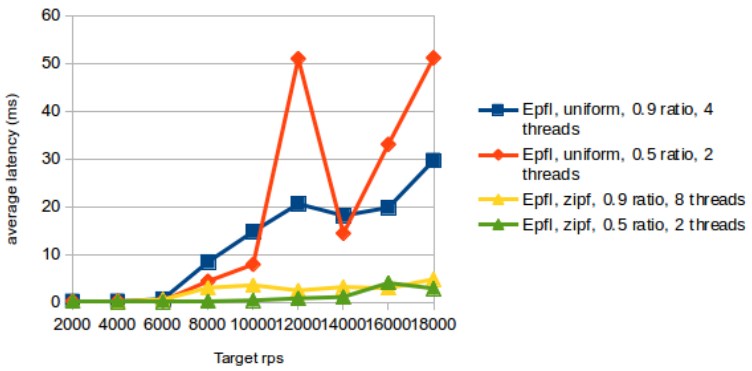
(c)



(d)

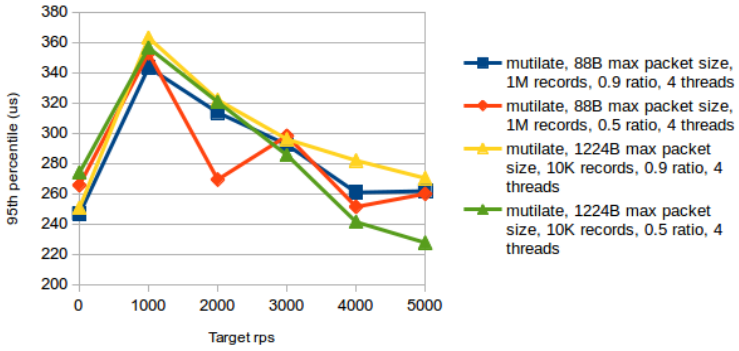


(e)

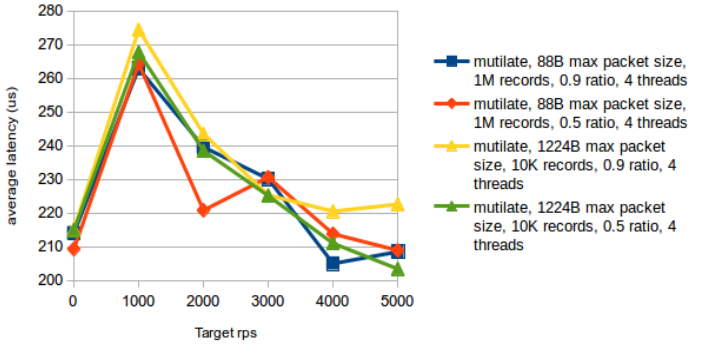


(f)

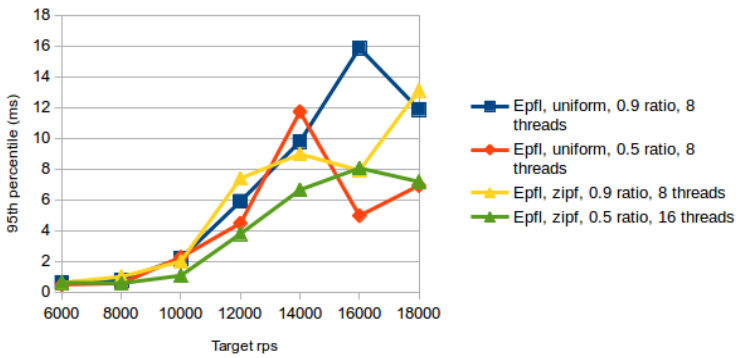
Figure 3.6: (a) Memcached, Data Caching Benchmark, 100 connections, 2 worker threads. (b) Memcached, Data Caching Benchmark, 100 connections, 2 worker threads. (c) Memcached, Mutilate, 100 connections, 2 worker threads. (d) Memcached, Mutilate, 100 connections, 2 worker threads. (e) Memcached, Data Caching Benchmark, 1 connection, 1 worker thread. (f) Memcached, Data Caching Benchmark, 1 connection, 1 worker thread.



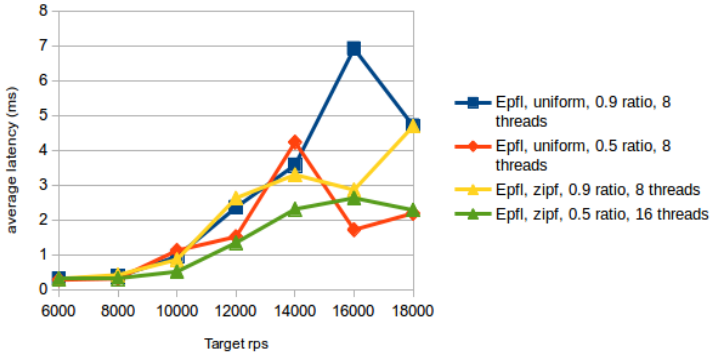
(a)



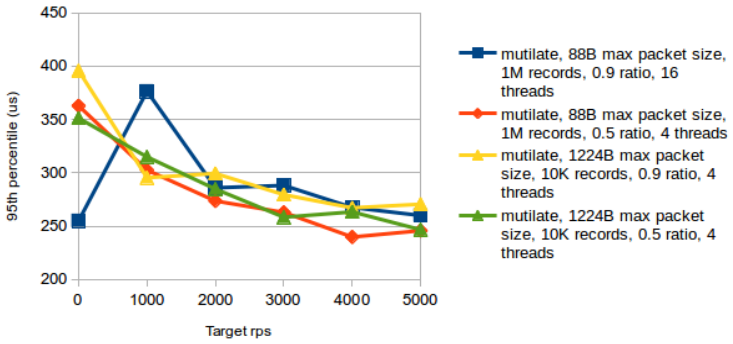
(b)



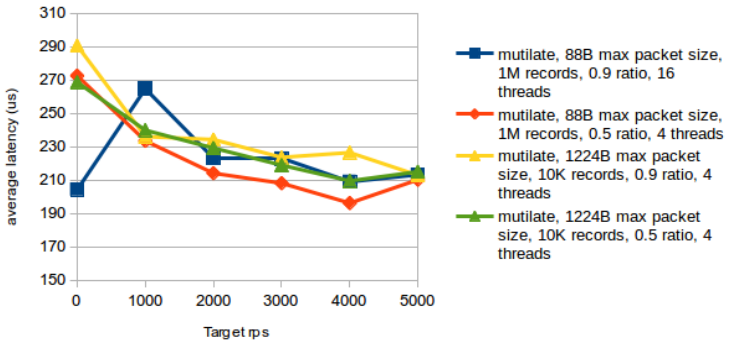
(c)



(d)



(e)



(f)

Figure 3.7: (a) Memcached, Mutilate, 1 connection, 1 worker thread. (b) Memcached, Mutilate, 1 connection, 1 worker thread. (c) MemC3, Data Caching Benchmark, 1 connection, 1 worker thread. (d) MemC3, Data Caching Benchmark, 1 connection, 1 worker thread. (e) MemC3, Mutilate, 1 connection, 1 worker thread. (f) MemC3, Mutilate, 1 connection, 1 worker thread.

On the whole, the results show that a 0.5 GET/SET ratio yields a higher maximum throughput than a 0.9 GET/SET ratio (when measuring maximum throughput with respect to lock power, this becomes more apparent at higher lock power values, as shown in Figures 3.4 (a) and (b)), while having a higher throughput threshold before the latency increases dramatically. One thing that is noted by looking at the Figures 3.4 (a) and (b), however, is that the results for x86 systems showed great inconsistency. However, the differences in throughput for varying values of lock power (in effect, varying number of locks) are relatively small.

As regards to workload distribution, it appears that Uniform and Zipf do not present significant differences in terms of performance. However, throughput-wise, a Uniform distribution appears to yield a slightly greater throughput than a Zipf distribution (Figures 3.4 (a), 3.5 (a), 3.5 (c)). On the other hand, as throughput increases, Uniform distribution leads to latency “blowing up” to higher values (Figures 3.6 (a) and (b)) or faster (Figures 3.6 (e), 3.6 (f), 3.7 (c), 3.7 (d)) than Zipf does. Also, higher throughput was achieved with a workload characterized by a large number of smaller-sized packets, rather than a workload of fewer, larger-sized packets (Figures 3.4 (b), 3.5 (b), 3.5 (d), 3.5 (f)). Moreover, the latency – target throughput graphs show that a smaller number of large packets cause latency to spike more quickly and higher than a larger number of smaller-sized packets (Figures 3.6 (c), 3.6 (d)).

Regarding maximum throughput with respect to the number of server threads, it should be noted that MemC3 seems to scale better than Memcached on more server threads (though we can only speculate as to how much better it scales, due to the lack of client-side parallelism), as seen in Figures 3.5 (e) and 3.5 (f) (compared to the corresponding graphs depicted in Figures 3.5 (c) and 3.5 (d)). Furthermore, we observe that more concurrent requests lead to a higher throughput achieved by the server, which is logical, since 1 connection and 1 client thread are not enough to saturate its processing power.

In addition, it should be mentioned that MemC3 exhibits better behavior latency-wise than Memcached, with the latter suffering from latency spikes much earlier than the former as target throughput increases (Figures 3.7 (c), 3.7 (d)).

However, for low throughput values, Memcached and MemC3 seem to exhibit similar behavior (Figures 3.7 (a), 3.7 (b), 3.7 (e), 3.7 (f)).

3.3.2. SYSTEMTAP

In order to achieve a more in-depth analysis as regards to how Memcached is executed in an x86 server and to identify its bottlenecks, SystemTap scripts were used. SystemTap is a scripting language and tool for dynamically instrumenting running production Linux kernel-based operating systems. More specifically, SystemTap is a tracing and probing tool that allows users to study and monitor the activities of the computer system (particularly, the kernel) in fine detail. It provides information similar to the output of tools like netstat, ps, top, and iostat, but is designed to provide more filtering and analysis options for collected information.

SystemTap offers:

- Flexibility: SystemTap's framework allows users to develop simple scripts for investigating and monitoring a wide variety of kernel functions, system calls, and other events that occur in kernel space. As a result, SystemTap is not so much a tool as it is a system that allows one to develop his/her own kernel-specific forensic and monitoring tools.
- Ease of use: as mentioned earlier, SystemTap allows users to probe kernel-space events without having to resort to instrument, recompile, install, and reboot the kernel. SystemTap scripts (many of which can be found in the official documentation) can demonstrate system forensics and monitoring capabilities not natively available with other similar tools (such as top, oprofile, or ps) [49].

In this case, SystemTap was used to measure the time spent in each system call related to Memcached and MemC3. In [1] it is stated that Memcached's execution cycle can be broken down into four parts:

- Network processing, which consists of system calls related to the networking stack. Memcached uses the Linux kernel stack.
- Concurrency control, which refers to the mechanisms that maintain data consistency while Memcached allows for parallel data access. Memcached uses a set of mutexes for that purpose.
- Memory management, which is how the system allocates and deallocates memory. Memcached employs the SLAB scheme in order to manage memory.
- Key-value processing, which consists of key-value request processing and housekeeping in the local system. In Memcached, this is implemented via a hash function and a hash table, while the eviction policy is LRU.

At first, a script was written (in SystemTap language, which is a C/C++/awk-like scripting language made only for SystemTap), which measured the time that system calls related to each of the aforementioned parts of Memcached's execution cycle by subtracting the timestamp of the system call's entry (when the OS enters kernel mode)

from the timestamp of the system call's return (when the OS leaves kernel mode). However, this did not account for the overlapping that occurred due to system calls entering without waiting for the return of others; as such, measurements were not correct.

To correct the measurements, the same idea was applied, albeit differently. A simple bash script calls four SystemTap scripts, which print the timestamps in the following format: a) X 0 in the case of a system call entry, b) 0 X in the case of a system call return (where X is the timestamp, which is the number of microseconds from the Epoch). Afterwards, a Python script parses the output file containing the above data and returns the total time spent in each part of the execution cycle. However, there was one last problem to be solved. The output files were extremely large to be parsed in a reasonable amount of time, containing up to 6 GBs of data each. Thus, only 1/1000 of the output files were parsed, specifically the latter thousandth of the file, since, at the start of the experiment, the server has not yet reached a stable state. One of the SystemTap scripts used, as well as the Python script, have been added in the Appendix.

The results of the above method are presented below. For Memcached, the setup is 100 connections and 2 worker threads, while for MemC3 it is 1 connection and 1 worker thread. Both are run on an x86 server (virtual machine). The number of server threads for each setup was chosen based on the maximum throughput measured in the corresponding experiments.

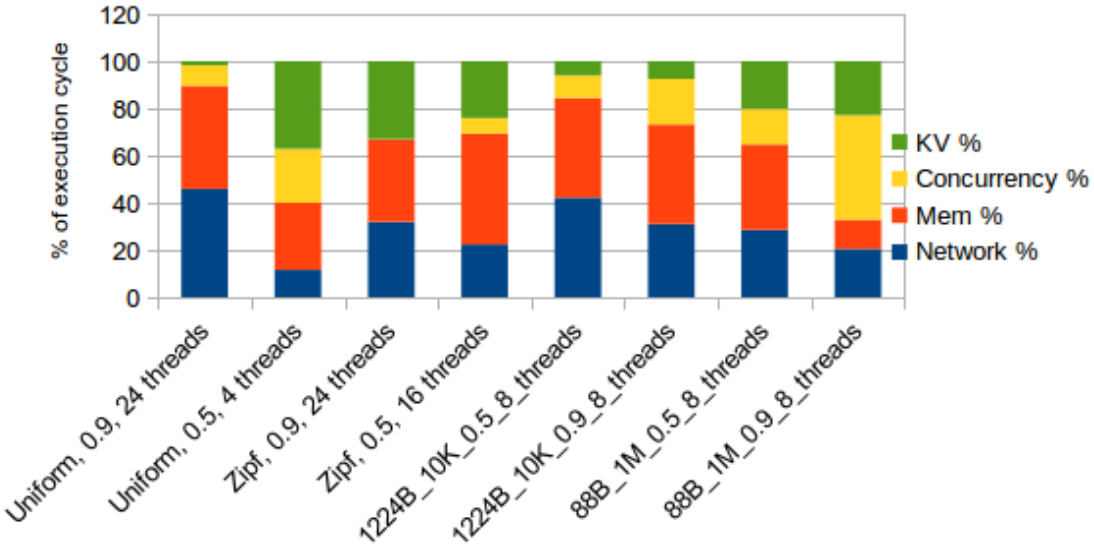


Figure 3.8: Memcached, execution cycle breakdown.

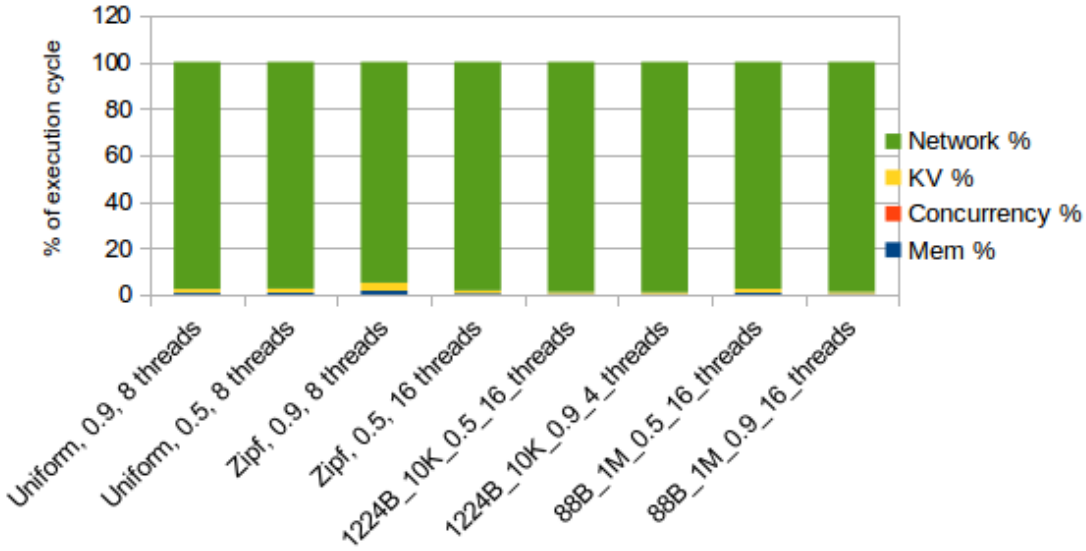


Figure 3.9: MemC3, execution cycle breakdown.

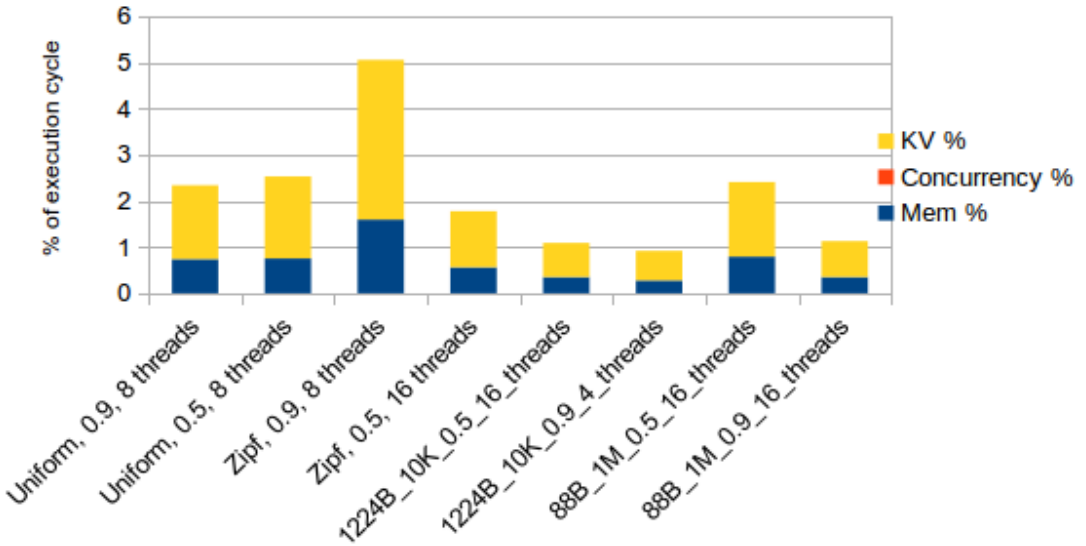


Figure 3.10: MemC3, execution cycle breakdown without network processing.

Memcached's execution cycle breakdown graphs provide a more in-depth view of its performance. The results, depicted in Figure 3.8, show that the greatest impedance to the application's performance are memory-related system calls, taking up 36% of the execution cycle (on average). Network-related system calls are consistently a large part of the execution cycle as well (29% on average). It is concluded that large packet/small record size workload performs badly, due to the small percentage that key-value processing takes up. It should also be noted that concurrency control does not seem to be impeding Memcached's performance greatly for these setups.

Finally, MemC3's execution cycle breakdown provides a little more insight, despite the lack of concurrency. First, it is apparent that network system calls are severely more frequent than any other part of the execution cycle (Figure 3.9). With the

help of the second graph (Figure 3.10), in which the contribution of the network-related system calls has been removed, it can be seen that memory-related system calls take up a very small percentage of the execution cycle. Key-value storage takes up a much larger portion of the execution cycle in comparison to memory.

3.3.3. COMMUNICATION PROCESSOR IMPLEMENTATION

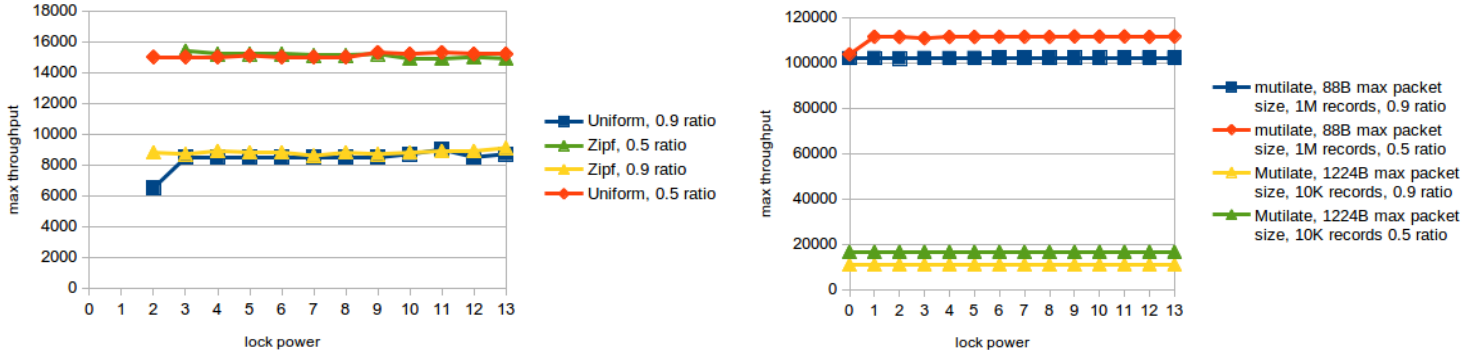
The graphs that follow are the results of the experiments conducted on a server with ARM architecture. More specifically, the machine used is Freescale's QorIQ LS2085A communication processor. The client is an x86 computer connected to the server via a 1 GbE port. As with the previous set of experiments, measurements were taken regarding the maximum throughput with respect to lock power and the number of server threads while latency remained under 10 ms (QoS agreement), as well as regarding latency (both the 95th percentile and the average value) with respect to a target throughput. However, the only configuration used was with 100 TCP connections and 2 worker threads.

The table below presents the communication processor's specifications.

Communication Processor	Number of cores	Memory	OS	NIC
QorIQ LS2085A	Up to 8x ARM Cortex- A57	8 GBs	Ubuntu	1 Gb Ethernet

Table 3.3: Communication processor specifications [52].

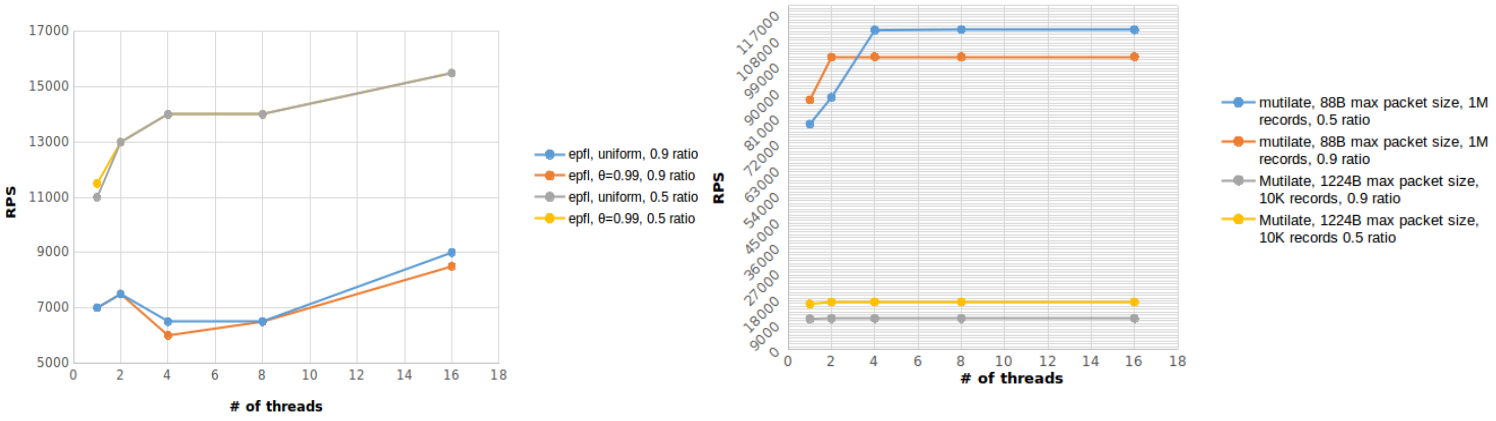
Maximum Throughput vs Lock Power



(a) (b)

Figure 3.11: (a) Memcached, Data Caching Benchmark. (b) Memcached, Mutilate.

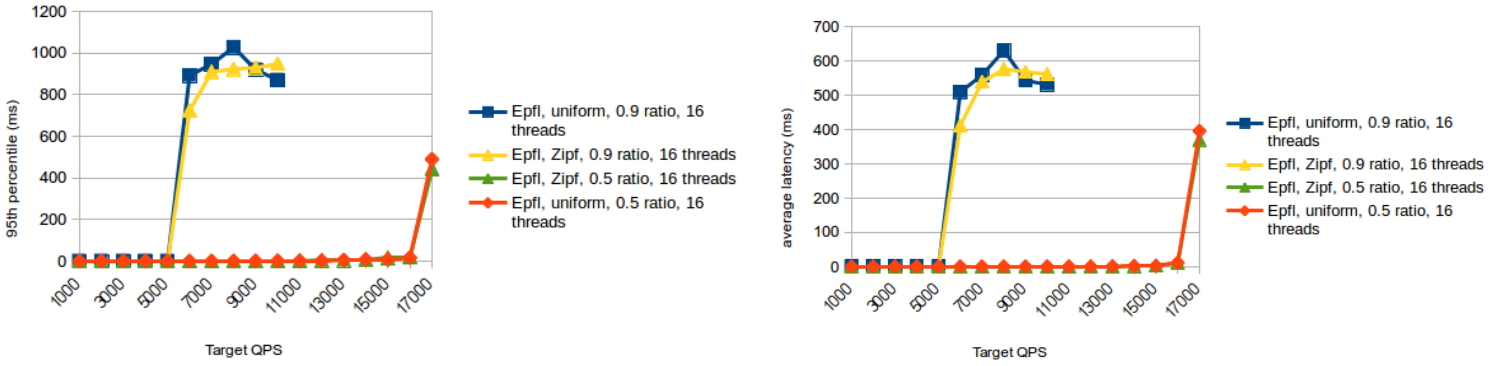
Maximum Throughput vs Server Threads



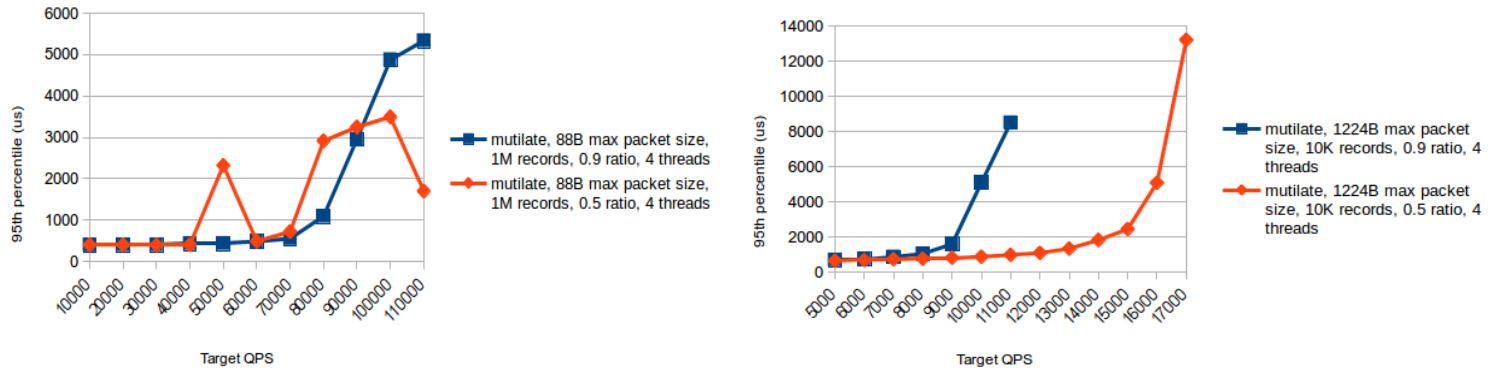
(a) (b)

Figure 3.12: (a) Memcached, Data Caching Benchmark. (b) Memcached, Mutilate.

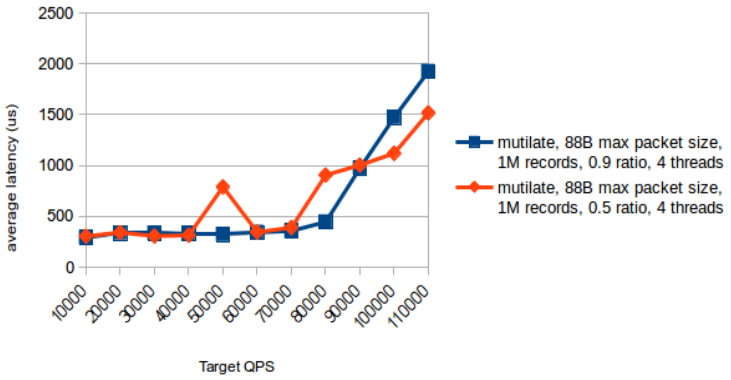
Latency vs Maximum Throughput



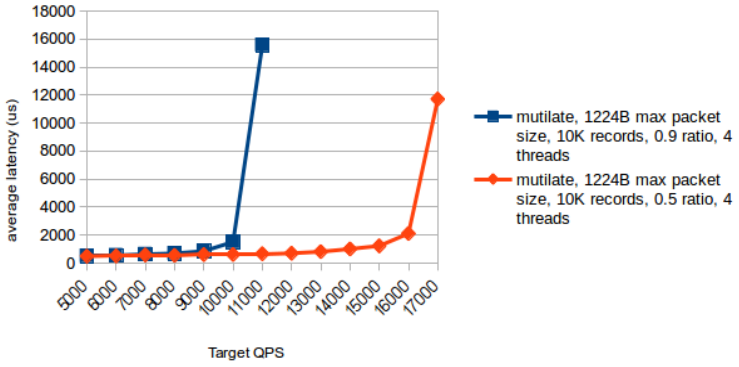
(a) (b)



(c) (d)



(e)



(f)

Figure 3.13: (a) Memcached, Data Caching Benchmark. (b) Figure 3.32: Memcached, Data Caching Benchmark. (c) Memcached, Mutilate. (d) Memcached, Mutilate. (e) Memcached, Mutilate. (f) Memcached, Mutilate.

The above graphs show the same characteristics as in the x86 case regarding changes in the workload parameters. More specifically, the server tends to yield better results with a 0.5 GET/SET ratio than with a 0.9 GET/SET ratio and a higher maximum throughput and robustness to latency with a large record size/small packet size workload than a small record size/large packet size workload. Furthermore, Uniform and Zipf key distributions lead to similar performance.

However, the LS2085A communication processor, which makes use of cores based on ARM architecture, seems to achieve higher maximum throughput than the server making use of the x86-based Intel Xeon processor (for the same configuration), while keeping latency constant (Figures 3.11 (a), 3.11 (b), 3.12 (a), 3.12 (b)). On the other hand, the former exhibits less tolerance to higher target throughput than the latter, which can be observed by the fact that latency shows a higher spike when a certain target throughput threshold has been crossed (Figures 3.13 (a), 3.13 (b), 3.13 (c), 3.13 (d), 3.13 (e), 3.14 (f)). Also, it should be noted that, with a dataset characterized by a large record and small packet size, the LS2085A processor achieves throughput an order of magnitude greater than the throughput achieved with a dataset characterized by a small record and large packet size (Figures 3.11 (b), 3.12 (b)).

3.3.4. Communication processor/x86 Comparison on Power Consumption

Since LS2085A features a multi-core ARM architecture CPU, it is worth exploring its power consumption while running Memcached, as well as its performance (throughput) per watt. The results will be compared to implementation results found in the related work.

The measurements that follow were taken using the following configuration: 12288 MBs of memory, GET/SET ratio of 0.95, 128 B object size, 64 B key size, 64 connections and 4 worker threads. While increasing Memcached threads bound to specific cores, it was found that, without violating the QoS agreement, the server achieved a maximum throughput of 165 KRPS for 5, 6, 7 and 8 threads. As such, the

least number of threads was chosen, meaning 5. A wattmeter was set up in order to measure the power consumed by the server. It was found that the server consumed 51.6 W while idle, 58.5 W with all cores active and 57.2 W with 5 cores active. This leads us to conclude that the maximum throughput per watt of the LS2085A communication processor is 160KRPS/57.2W ≅ 2.9 KRPS/Watt. The comparison with Memcached's implementations found in the related work is shown in the following table.

Implementation	KRPS/Watt	Maximum Latency
MemC3 (NSDI '13)	3.8	0.1 ms
LS2085A	2.9	10 ms
Xeon implementation (NSDI '14)	1.8	1ms
Xeon implementation (ISCA'13)	1	5 ms
Xeon implementation (ISCA '15)	0.8	0.1 ms

Table 3.4: Performance/watt comparison between Memcached's implementation on the LS2085A communication processor and others found in the related work.

4. CONCLUSIONS

From the experimental results presented in chapter 3, some conclusions can be drawn about Memcached's behavior. These can be summarized as follows:

- Regarding the workload, Memcached appears to be struggling with GETs rather than SETs. This occurs because GETs are costlier than SETs; as explained in chapter 3, GETs require 7 costly steps to be executed, rather than 6 simpler steps. The latter are relatively less costly than the former, because the GET request might involve a key that does not exist in the cache, which can delay the completion of the request.
- Workloads with a large number of records and small packet size stress Memcached significantly more than workloads with fewer records but larger sized packets. This happens because packets with size smaller than 1 KB are CPU-bound, while packets with size larger than 1 KB are network-bound. In other words, the former saturate the CPU's pipeline, while the latter saturate the NIC first [3]. However, in this case, the NIC was never saturated; there was either a virtual network that featured virtual interfaces that were characterized by a large bandwidth, or a physical 1 GbE port. The lower throughput that was observed for fewer, larger-sized packets (compared to many small-sized packets) was due to the fact that, since the total number of items that could be contained was lower, the probability of a certain item being evicted before it was requested again was higher. As such, if the available memory is not enough, large packets can stress the Memcached server, even if the NIC has enough bandwidth.
- The key distribution didn't seem to have a significant effect on Memcached's performance, although for the uniform distribution (which is, in theory, the best case scenario) the maximum throughput was slightly higher in most cases. This leads to the conclusion that this is not a deciding factor for Memcached's performance, however longer experiments (with respect to time), which could not be conducted, could prove otherwise and reveal a larger performance gap between uniform and zipf distribution. Experiments with a longer duration would reveal if skewness in key choice stresses the server heavily or not.

As regards Memcached's execution cycle breakdown, it was revealed that:

- Memory-related system calls are a serious impedance to its performance, regardless of workload type. It can be concluded that the SLAB memory management scheme is not very efficient, which is why many key-value stores do not employ it.
- The OS-provided POSIX I/O induces a significant overhead on Memcached's performance, thus making it unsuitable for a highly efficient key-value store.

Regarding MemC3, it was shown that:

- It can achieve much better performance than Memcached in every way, even without making use of all its optimizations. This shows that Memcached's

hashing scheme, the Jenkins algorithm, is not very efficient. MemC3's optimistic cuckoo hashing scheme seems to perform much better, making it a better choice.

Finally, Memcached's implementation on the LS2085A communication processor showed promising results. More specifically:

- Strictly performance-wise, the LS2085A communication processor achieves a higher maximum throughput than the x86 server, especially for easier-to-handle workloads, in which cases it can achieve throughput an order of magnitude higher. The same cannot be said about its performance when it is trying to achieve a higher target throughput; latency rises faster than on an x86-based server.
- The LS2085A communication processor showed great performance per watt while running Memcached, surpassing many implementations running on servers with Intel Xeon CPUs. On the whole, it should be considered as a possible candidate for running Memcached or other key-value stores.

The conclusions that were drawn are shown in the table below.

Implementation	**Workload-related bottlenecks**	**Execution cycle-related bottlenecks**	**Performance**	**Performance/Watt**
x86 (Memcached)	GETs; many small-sized packets; large-sized packets (depends on available memory)	Memory-related system calls; secondarily Network-related system calls	13.5 KRPS (without parallelism); 60 KRPS (with parallelism)	1.8 KRPS/Watt
x86 (MemC3)	GETs	Network-related system calls	17 KRPS (without parallelism)	3.8 KRPS/Watt
Communication Processor (Memcached)	GETs; many small-sized packets; large-sized packets (depends on available memory)	-	111 KRPS (with parallelism)	2.9 KRPS/Watt

Table 4.1: Summary of conclusions.

5. FUTURE WORK

First, experiments quite long in duration (e.g. lasting days) can be conducted in order to determine the true impact of the distribution that key choice follows. Furthermore, MemC3 should be run using full parallelism and optimizations, in order to gauge its true potential and compare it with Memcached. Other optimized versions of Memcached (e.g. MICA) and key-value stores like Redis should also have their performances measured in different configurations and scenarios, in order to determine which is superior and in which cases, since some may perform better in specific workloads than others.

It would also be interesting to implement MemC3 on the LS2085A communication processor, in order to conduct a comparison between MemC3 and Memcached, especially on the performance-per-watt level. This could be done for other key-value stores, as well. Last, it should be attempted to draw the communication processor's full potential by utilizing its unique characteristics that make it (in theory) ideal for such applications. The LS2085A communication processor features advanced, high-performance datapath and network peripheral interfaces required for networking, telecom/datacom, wireless infrastructure, military and aerospace applications. Attempts should be made to make use of its network-oriented design in order to achieve higher performance and performance per watt, especially on key-value stores, which are of critical importance to large scale datacenters.

REFERENCES

[1] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, Pradeep Dubey, "Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-Value Store Server Platform", ISCA '15, June 2015.

[2] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, Venkateshwaran Venkataramani "Scaling Memcache at Facebook", USENIX Association,10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13).

[3] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, Thomas F. Wenisch "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached", ISCA '13 Tel Aviv, Israel.

[4] Sai Rahul Chalamalasetti, Alvin AuYoung, Kevin Lim, Parthasarathy Ranganathan, Mitch Wright, Martin Margala, "An FPGA Memcached Appliance", FPGA'13, February 11–13, 2013, Monterey, California, USA.

[5] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bar, Zsolt Istvan "Achieving 10Gbps line-rate key-value stores with FPGAs"

[6] Maysam Lavasani, Hari Angepat, Derek Chiou, "An FPGA-based In-line Accelerator for Memcached", Department of Electrical and Computer Engineering, The University of Texas at Austin, 2014.

[7] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, Boris Grot, "Scale-Out NUMA", ASPLOS '14, March, 2014, Salt Lake City, Utah, USA.

[8] Anthony Gutierrez, Michael Cieslak, Bharan Giridhar, Ronald G. Dreslinski, Luis Ceze, Trevor Mudge, "Integrated 3D-Stacked Server Designs for Increasing Physical Density of Key-Value Stores", ASPLOS '14, March 1–4, 2014, Salt Lake City, Utah, USA.

[9] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, Miguel Castro "FaRM: Fast Remote Memory", 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14), April, 2014, Seattle, WA, USA.

[10] Bin Fan, David G. Andersen, Michael Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing", USENIX Association 10th, USENIX Symposium on Networked Systems Design and Implementation (NSDI '13).

[11] Anuj Kalia, Michael Kaminsky, David G. Andersen, "Using RDMA Efficiently for Key-Value Services", SIGCOMM'14, August, 2014, Chicago, IL, USA.

[12] Mao, Yandong, Edward W. Kohler, and Robert Morris, "Cache Craftiness for Fast Multicore Key-Value Storage", 7th ACM European Conference on Computer Systems: April, 2012, Bern, Switzerland.

[13] Christopher Mitchell, Yifeng Geng, Jinyang Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store", USENIX Association 2013 USENIX Annual Technical Conference (USENIX ATC '13).

[14] Hyeontaek Lim, Dongsu Han, David G. Andersen, Michael Kaminsky, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage", NSDI, 2014.

[15] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, Kenneth Steele, “Power and Performance Evaluation of Memcached on the TILEPro64 Architecture”, October 10, 2011.
[16] https://insidehpc.com/hpc-basic-training/what-is-hpc/
[17] http://www2.itif.org/2016-high-performance-computing.pdf
[18] Robert Krulwich (2001-04-17). Cracking the Code of Life (Television Show). PBS.
[19] http://www.hpctoday.com/state-of-the-art/ibm-watson-the-future-problem-solving-supercomputer/].
[20] Coulouris, George; Jean Dollimore; Tim Kindberg; Gordon Blair (2011). Distributed Systems: Concepts and Design (5th Edition). Boston: Addison-Wesley. ISBN 0-132-14301-1.
[21] https://www.techopedia.com/definition/18909/distributed-system
[22] Memcached.org
[23] MocoSpace Architecture - 3 Billion Mobile Page Views a Month. High Scalability (2010-05-03). Retrieved on 2013-09-18.
[24] Cuong Do Cuong (Engineering manager at YouTube/Google) (June 23, 2007). Seattle Conference on Scalability: YouTube Scalability (Online Video - 26th minute). Seattle: Google Tech Talks.
[25] Steve Huffman on Lessons Learned at Reddit Archived 2010-05-17 at the Wayback Machine.
[26] https://www.survata.com/jobs/
[27] http://gigaom.com/2010/06/08/how-zynga-survived-farmville/
[28] https://web.archive.org/web/20100527004122/http://developers.facebook.com/opensource/
[29] https://www.facebook.com/note.php?note_id=39391378919&ref=mf
[30] https://partner.orange.com/
[31] https://blog.twitter.com/official/en_us/a/2008/its-not-rocket-science-but-its-our-work.html
[32] http://www.jobscore.com/jobs2/tumblr/engineer-core-applications-group/cvAFBCbcyr47JbiGakhP3Q?ref=rss&sid=68
[33] https://www.mediawiki.org/wiki/Memcached
[34] http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale-105tb-ram-39mm-qps-10000-ins.html
[35] http://www.project-voldemort.com/voldemort/
[36] https://aws.amazon.com/elasticache/redis/
[37] http://docs.aws.amazon.com/AmazonElastiCache/latest/UserGuide/Scaling.html
[38] https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/
[39] https://gigaom.com/2013/03/05/facebook-kisses-dram-goodbye-builds-memcached-for-flash/
[40] Alex Wiggins, Jimmy Langston, ‘Enhancing the Scalability of Memcached’, May 2012.
[41] http://parsa.epfl.ch/cloudsuite_old/memcached.html
[42] https://github.com/leverich/mutilate/blob/master/README.md

[43] https://www.ibm.com/blogs/cloud-computing/2013/04/how-cloud-computing-is-impacting-everyday-life/
[44] https://www.ibm.com/blogs/cloud-computing/2013/04/how-cloud-computing-is-impacting-everyday-life-2/
[45] Hassan, Qusay (2011). "Demystifying Cloud Computing". The Journal of Defense Software Engineering. CrossTalk. 2011 (Jan/Feb): 16–21.
[46] Peter Mell and Timothy Grance (September 2011). The NIST Definition of Cloud Computing (Technical report). National Institute of Standards and Technology: U.S. Department of Commerce. doi:10.6028/NIST.SP.800-145. Special publication 800-145.
[47] https://www.valentina-db.com/en/valentina-key-value-database
[48] http://basho.com/resources/key-value-databases/
[49] SystemTap 3.0, SystemTap Beginners Guide
[50] http://www.ndm.net/isilon/high-performance-computing
[51] http://www.datamation.com/cloud-computing/what-is-cloud-computing.html
[52] http://www.nxp.com/products/microcontrollers-and-processors/arm-processors/qoriq-layerscape-arm-processors/development-resources/qoriq-ls2085a-rdb-reference-design-board:LS2085A-RDB

APPENDICES

1. SYSTEMTAP SCRIPT FOR CREATION OF FILE TO BE PARSED CONTAINING NETWORK-RELATED SYSTEM CALL TIMESTAMPS

```
#systemtap script for creation of file to be parsed containing
network-related system call timestamps
probe syscall.socket {
      if     (execname()=="memcached")     printf("%lu     0\n",
gettimeofday_us())
}

probe syscall.bind {
      if     (execname()=="memcached")     printf("%lu     0\n",
gettimeofday_us())
}

probe syscall.epoll_ctl {
      if     (execname()=="memcached")     printf("%lu     0\n",
gettimeofday_us())
}

probe syscall.epoll_wait {
      if     (execname()=="memcached")     printf("%lu     0\n",
gettimeofday_us())
}

probe syscall.read {
      if     (execname()=="memcached")     printf("%lu     0\n",
gettimeofday_us())
}

probe syscall.sendmsg {
      if     (execname()=="memcached")     printf("%lu     0\n",
gettimeofday_us())
}

probe syscall.socket.return {
      if     (execname()=="memcached")     printf("0     %lu\n",
gettimeofday_us())
}

probe syscall.bind.return {
```

```
      if     (execname()=="memcached")     printf("0     %lu\n",
gettimeofday_us())
}

probe syscall.epoll_ctl.return {
      if     (execname()=="memcached")     printf("0     %lu\n",
gettimeofday_us())
}

probe syscall.epoll_wait.return {
      if     (execname()=="memcached")     printf("0     %lu\n",
gettimeofday_us())
}

probe syscall.read.return {
      if     (execname()=="memcached")     printf("0     %lu\n",
gettimeofday_us())
}

probe syscall.sendmsg.return {
      if     (execname()=="memcached")     printf("0     %lu\n",
gettimeofday_us())
}
```

2. PYTHON PARSING SCRIPT

```
# Python parsing script
# Import modules
import pandas as pd
import numpy as np
# Insert filepath
filepath                                                              =
"/home/valsamidis/measurements_systemtap_memcached_100_connect
ions_2_workers/measurements/epfl_uniform_0.9_24_threads/mem_ou
t.txt"

# Open file
f = pd.read_csv(filepath,
                sep = " ",
                names = ["entry_time","return_time"])

def merge(list1, list2):

      merged_list = []
```

```
      while (list1 and list2):
            if list1[0][0] > list2[0][0]:
                  merged_list.append(list2[0])
                  list2[0:1] = []
            else:
                  merged_list.append(list1[0])
                  list1[0:1] = []
      if list1:
            merged_list = merged_list + list1
      else:
            merged_list = merged_list + list2

      return merged_list

# Create lists
entry_time_list = list(f["entry_time"])
return_time_list = list(f["return_time"])

# Create shorter lists
initial_entry_length = len(entry_time_list)
initial_return_length = len(return_time_list)
factor_entry = initial_entry_length // 1000
factor_return = initial_return_length // 1000

entry_time_list = entry_time_list[-factor_entry:]
return_time_list = return_time_list[-factor_return:]

# Remove zeroes
i = 0
while (i < initial_entry_length):
      try:
            if (entry_time_list[i] == 0):
                  del entry_time_list[i]
                  initial_entry_length = initial_entry_length - 1
            else:
                  i = i + 1
      except IndexError:
            break

i = 0
while (i < initial_return_length):
      try:
            if (return_time_list[i] == 0):
                  del return_time_list[i]
                  initial_return_length = initial_return_length -
1
            else:
                  i = i + 1
```

```
      except IndexError:
            break

# Get lists' length
entry_length = len(entry_time_list)
return_length = len(return_time_list)

# Create arrays
entry_time_tuples = [(entry_time_list[0] , 1)]
return_time_tuples = [(return_time_list[0], -1)]

for i in range(1, entry_length):
      entry_time_tuples       =       entry_time_tuples       +
[(entry_time_list[i], 1)]
for i in range(1, return_length):
      return_time_tuples      =       return_time_tuples      +
[(return_time_list[i], -1)]

# Concatenate and sort
full_array = merge(entry_time_tuples, return_time_tuples)

# Start with entry
i = 0
while (full_array[0][1] == -1):
      full_array[0:1] = []

# Initializations
start = 0
total_time = 0
offset = 0
flag = True
full_array_length = len(full_array)

# Main program
for i in range(full_array_length):
      offset = offset + full_array[i][1]
      if (offset == 0):
            total_time = total_time + (full_array[i][0] - start)
            flag = True
      if ((offset == 1) and (flag == True)):
            start = full_array[i][0]
            flag = False

# Print Output
print("Total time spent: " + str(total_time) + "\n")
```

LIST OF FIGURES

Benchmark, 1 connection, 1 worker thread. (d) MemC3, Data Caching Benchmark, 1 connection, 1 worker thread. (e) MemC3, Mutilate, 1 connection, 1 worker thread. (f) MemC3, Mutilate, 1 connection, 1 worker thread.

[44] Figure 3.8: Memcached, execution cycle breakdown.

[45] Figure 3.9: MemC3, execution cycle breakdown.

[46] Figure 3.10: MemC3, execution cycle breakdown without network processing.

[47] Figure 3.11: (a) Memcached, Data Caching Benchmark. (b) Memcached, Mutilate.

[48] Figure 3.12: (a) Memcached, Data Caching Benchmark. (b) Memcached, Mutilate.

[49] Figure 3.13: (a) Memcached, Data Caching Benchmark. (b) Figure 3.32: Memcached, Data Caching Benchmark. (c) Memcached, Mutilate. (d) Memcached, Mutilate. (e) Memcached, Mutilate. (f) Memcached, Mutilate.

LIST OF TABLES