Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

A client-server synchronization model for concurrent data structures implemented in SCC

Διπλωματικη Εργασια

του

ΤΑΣΟΥΛΑ ΖΩΗ - ΓΕΡΑΣΙΜΟΥ

Επιβλέπων: Δημήτριος Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2016

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

A client-server synchronization model for concurrent data structures implemented in SCC

Διπλωματικη Εργασια

του

ΤΑΣΟΥΛΑ ΖΩΗ - ΓΕΡΑΣΙΜΟΥ

Επιβλέπων: Δημήτριος Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Μαρτίου 2016.

.....	.....	.....
Δημήτριος Σούντρης	Νεκτάριος Κοζύρης	Κιαμάλ Πεκμεστζή
Αναπληρωτής Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2016

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

.....

Τασούλας Ζώης - Γερασιμος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Αντικείμενο της διπλωματικής εργασίας είναι η αξιολόγηση κάποιον ταυτόχρονων δομών δεδομένων στο πολυπύρηνο σύστημα Single-chip Cloud Computer της εταιρίας Intel. Συγκεκριμένα κατά τη διάρκεια της διπλωματικής εργασίας μελετήθηκαν και αξιολογήθηκαν για διάφορες παραμέτρους οι δομές δεδομένων της στοίβας, της FIFO ουράς προτεραιότητας και του διονυμικού σωρού μεγίστου.

Το σύστημα Single-chip Cloud Computer είναι ένας υπολογιστής γενικού σκοπού με 48 πυρήνες, η δημιουργία του οποίου προορίζεται για ερευνητικούς σκοπούς και μελέτη των πολυπύρηνων συστημάτων. Ο μεγάλος αριθμός πυρήνων και το μοναδικό αυτό σύστημα μας παρότρυνε να το επιλέξουμε για να μελετήσουμε τις επιδώσεις και την συμπεριφορά πολυπύρηνων συστημάτων. Ένα επίσης ενδιαφέρον στοιχείο που μας οδήγησε να επιλέξουμε την έρευνα πάνω σε αυτόν τον υπολογιστή είναι η ιδιαιτερότητα του μοντέλου μνήμης, γεγονός που μας δημιούργησε το ενδιαφέρον να δοκιμάσουμε το μοντέλο πελάτη εξυπηρετητή για την οργάνωση και των συγχρονισμό ταυτόχρονων δομών δεδομένων.

Στη διάρκεια της διπλωματικής συγκρίναμε την συμπεριφορά και την επίδοση των δομών δεδομένων που αναφέραμε. Χρησιμοποιήσαμε ένα πλήθος διαφορετικών σεναρίων αιτημάτων και διεργασιών από τους πυρήνες και λάβαμε μετρήσεις τόσο για την χρονική επίδοση, όσο και για την κατανάλωση ενέργειας από κάθε δομή. Ακόμη δοκιμάσαμε κατά πόσο επηρεάζει την επίδοση η κατανομή των πυρήνων κατά τη δέσμευσή τους, αν θα είναι συνεχείς ή διασκορπισμένοι. Επίσης είδαμε κατά πόσο η θέση του πυρήνα εξυπηρετητή επηρεάζει την επίδοση, και πώς θα συμπεριφέρονταν οι δομές και το σύστημα στην περίπτωση που μεσολαβούσε κάποια καθυστέρηση ή η ενασχόληση με κάποια άλλη δουλειά από τους πυρήνες, μεταξύ δύο διαδοχικών αιτημάτων για αλλαγές στη δομή δεδομένων.

Λέξεις Κλειδιά

Ταυτόχρονες δομές δεδομένων, πολυπύρηνα συστήματα, συγχρονισμός, μοντέλο πελάτη εξυπηρετητή, SCC

Abstract

The purpose of this diploma thesis is to evaluate concurrent data structures with the multicore system by Intel, the Single-chip Cloud Computer. Specifically during the work for this thesis we examined and evaluated through different parameters the data structures of stack, FIFO queue and binary max heap.

The Single-chip Cloud Computer system is a general purpose computer of 48 cores. Its creation was meant for research purposes and further study of multicore systems. The big number of cores and this unique system made us choose it to examine the performance and the behaviour of multicore systems. Another interesting fact that led us choose to do our research with this computer is the special memory model that the Single-chip Cloud Computer has, which created the curiosity to test and evaluate the performance of a client-server model for the synchronization and the organizing of the concurrent data structures.

During this thesis we compared the behaviour and the performance of the data structures we mentioned above. We used a set of different scenarios consisting of a different set of requests and transactions from the cores. We gathered measurements both for the time performance and also for the power consumption by every data structure. In addition to this, we tested how the position of the allocated cores, if they are continuously allocated or distributed, affects performance. Also we wanted to see how and if the position of the server core in the client-server model affects performance and how the data structures and the system will behave and perform in case there would be a delay or some time spent by cores to handle another task, between two consecutive requests for transactions with the data structure.

Keywords

Concurrent data structures, multicore systems, synchronization, client server model, SCC

Ευχαριστίες

Θα ήθελα να ευχαριστήσω στο σημείο αυτό τον επιβλέποντα καθηγητή κ. Δημήτριο Σούντρη που με εμπιστεύτηκε και μου έδωσε την ευκαιρία να ασχοληθώ με το θέμα αυτό. Τον ευχαριστώ επίσης για την πολύ καλή μας συνεργασία και επικοινωνία και για την προθυμία του να με βοηθήσει και να με καθοδηγήσει στις ακαδημαϊκές μου επιλογές.

Επίσης ευχαριστώ ιδιαίτερα τον υποψήφιο διδάκτορα κ. Λάζαρο Παπαδόπουλο για την καθοδήγηση και την άριστη συνεργασία μας κατά την διάρκεια εκπόνησης της διπλωματικής αυτής εργασίας. Η επικοινωνία μας ήταν άριστη και ήταν πάντα διαθέσιμος ώστε να λύνουμε τα θέματα που ανέκυπταν και να προχωράμε στην περάτωση της εργασίας αυτής. Να ευχαριστήσω επίσης τον κ. Γιώργο Χατζηκωνσταντή γιατί ήταν πρόθυμος και διαθέσιμος να συμβάλει με την βοήθειά του και την εμπειρία στο κομμάτι των μετρήσεων της κατανάλωσης ενέργειας.

Τέλος θα ήθελα να ευχαριστήσω την οικογένειά μου, την μητέρα μου και τις αδερφές μου, για την υποστήριξή τους, την αγάπη τους και την υπομονή τους όλα αυτά τα χρόνια. Ήταν ένα σημαντικό κομμάτι της προσπάθειάς μου και χωρίς αυτές πολλά πράγματα θα ήταν δύσκολα. Να ευχαριστήσω επίσης την Αλεξάνδρα αλλά και όλους τους φίλους με τους οποίους ήρθαμε σε επαφή λόγω του Εθνικού Μετσόβιου Πολυτεχνείου ή γενικότερα των σπουδών. Έκαναν την διαδρομή μέχρι την απόκτηση του πτυχίου πιο ευχάριστη και πολλές φορές ήταν η αφορμή να διδαχθώ και να μάθω πολλά πράγματα.

Περίληψη διπλωματικής

Όπως φαίνεται από τον τίτλο, το θέμα της διπλωματικής αυτής εργασίας είναι η μελέτη, σχεδίαση και αξιολόγηση ενός μοντέλου συγχρονισμού πελάτη-εξυπηρετητή, για ταυτόχρονες δομές δεδομένων, υλοποιώντας το για τον υπολογιστή Single-chip Cloud Computer της εταιρίας Intel.

Η επιλογή του ειδικού αυτού υπολογιστή έγινε γιατί είναι ένα σύστημα που διαθέτει 48 επεξεργαστικούς πυρήνες. Σχεδιάστηκε από την Intel για ερευνητικούς σκοπούς και διατίθεται για έρευνα πάνω σε διάφορους τομείς των πολυπύρηνων συστημάτων, όπως λειτουργικά συστήματα για πολυπύρηνα συστήματα, τρόποι επικοινωνίας των πυρήνων, βέλτιστες πρακτικές για αποδοτική επικοινωνία, χρήση μνήμης και εξοικονόμηση ενέργειας σε πολυπύρηνους υπολογιστές γενικού σκοπού. Ο λόγος που δημιουργήθηκε το ενδιαφέρον για τη μελέτη του μοντέλου πελάτη-εξυπηρετητή είναι η ύπαρξη μιας ιδιαιτερότητας στο σύστημα μνημών του Single-chip Cloud Computer. Αργότερα θα σχολιάσουμε εκτενέστερα την οργάνωση και την ιεραρχία των μνημών, αλλά αυτό που προξενεί ενδιαφέρον και είναι αφορμή για έρευνα και μελέτη είναι η ύπαρξη μιας πολύ γρήγορης, μικρής μεν στο μέγεθος, μνήμης εντός κάθε ψηφίδας του συστήματος.

Η ιδιαίτερη αυτή μνήμη ήταν η αφορμή για να δοκιμάσουμε πώς μπορεί ένα μοντέλο συγχρονισμού που θα βασίζεται στην ανταλλαγή μηνυμάτων και δεδομένων μικρού μεγέθους να αποδώσει, σε σύγκριση με άλλα μοντέλα. Συγκεκριμένα δοκιμάσαμε κάποια μοντέλα με κλειδώματα και τα συγκρίναμε με αυτό του πελάτη-εξυπηρετητή. Το κύριο αντικείμενο της διπλωματικής αυτής εργασίας είναι η σύγκριση των μοντέλων αυτών σε διάφορους τομείς, όπως χρονική επίδοση, κατανάλωση ενέργειας, δικαιοσύνη στην ολοκλήρωση αιτημάτων και κάποιες άλλες κατηγορίες συγκρίσεων που θα δούμε παρακάτω.

Για τις διεργασίες και την ολοκλήρωση της διπλωματικής αυτή εργασίας χρησιμοποιήθηκαν κυρίως εργαλεία ελεύθερου λογισμικού όπου αυτό ήταν δυνατό. Επίσης για τη συγγραφή χρησιμοποιήθηκε η τυπογραφική σουίτα LaTeX.

Στο κείμενο που ακολουθεί θα δούμε με σειρά κάποια θεωρητικά στοιχεία για αρχές συγχρονισμού, τόσο παραδοσιακές όσο και κάποιες προχωρημένες αρχές που εμφανίστηκαν πρόσφατα. Στη συνέχεια θα παρουσιάσουμε με περισσότερες λεπτομέρειες το σύστημα Single-chip Cloud Computer καθώς και τις υλοποιήσεις που χρησιμοποιήσαμε για τις μετρήσεις μας. Έπειτα την βάση και τη δομή των σεναρίων που χρησιμοποιήσαμε για να αξιολογήσουμε τις διάφορες δομές δεδομένων καθώς και θα σχολιάσουμε ποια μεγέθη και ιδιότητες μετρήσαμε. Τέλος θα σχολιάσουμε τα αποτελέσματα που εξάγαμε και θα προτείνουμε κάποιες μελλοντικές

προεκτάσεις της δουλειάς μας για αυτή τη διπλωματική εργασία.

1.1 Μέθοδοι συγχρονισμού

Μιλώντας για τις αρχές συγχρονισμού των ταυτόχρονων δομών δεδομένων, αυτές είναι απαραίτητες στα πολυπύρηνα συστήματα και όταν μιλάμε για ταυτόχρονες ή παράλληλες διεργασίες. Αυτό προκύπτει από το γεγονός ότι είτε υπάρχει δυνατότητα πολλαπλών προσβάσεων ταυτόχρονα στη μνήμη, είτε το υλικό δεν επιτρέπει κάτι τέτοιο και τα αιτήματα πρόσβασης στη μνήμη σειριοποιούνται, πρέπει να υπάρχει ένας μηχανισμός που να εγγυάται την ακεραιότητα των αιτημάτων, δηλαδή την ακεραιότητα των δεδομένων. Πρέπει επίσης να είμαστε σίγουροι ότι η κατάσταση της μνήμης θα είναι συνεπής έπειτα από κάποια διεργασία με τα δεδομένα της, ότι τα δεδομένα που θα διαβάζονται ή θα γράφονται θα είναι τα επιθυμητά και τέλος ότι δεν θα προκύπτουν απροσδιόριστες καταστάσεις.

Όπως έχουμε δει από την αρχιτεκτονική των υπολογιστών ή από τον τρόπο λειτουργίας των μεταγλωττιστών, πολλές εντολές μπορεί να αναδιαταχθούν, μπορεί για εξοικονόμηση χρόνου και βελτίωση της επίδοσης μια πρόσβαση στη μνήμη να μην γίνει ακριβώς όταν ζητείται αλλά αργότερα όταν το σύστημα κρίνει ότι είναι η καταλληλότερη στιγμή. Επίσης όταν υπάρχουν ταυτόχρονα αιτήματα πρέπει να προκύπτει με σαφήνεια πιο αίτημα θα πραγματοποιηθεί πότε και προφανώς να μην έχουμε αλληλοκαλύψεις αιτημάτων και εντολών που μπορούν να οδηγήσουν σε ασυνέπεια δεδομένων ή λανθασμένες αναγνώσεις και εγγραφές στοιχείων.

Για την αποφυγή όλων των παραπάνω κινδύνων και την προστασία των δεδομένων, της κατάστασης της μνήμης και της λειτουργίας των εφαρμογών και των προγραμμάτων μας, είναι απαραίτητοι οι μηχανισμοί συγχρονισμού. Ο συγχρονισμός μπορεί να γίνεται είτε από το υλικό είτε με λογισμικό. Υπάρχουν δε αλγόριθμοι και μηχανισμοί συγχρονισμού που παρότι υλοποιούνται με λογισμικό χρειάζονται υποστήριξη από ειδικό ή συγκεκριμένο υλικό, αυτοί οι μηχανισμοί ονομάζονται υβριδική. Με λίγα λόγια ο συγχρονισμός είναι ένας μηχανισμός που αναλαμβάνει να εγγυειθή την σωστή και συνεπή λειτουργία προγραμμάτων που εκτελούνται παράλληλα και αλληλεπιδρούν με την μνήμη. Με διάφορους τρόπους, υλικού, λογισμικού ή υβριδικούς, αναλαμβάνει να διαθέτει την πρόσβαση στη μνήμη σε κάθε πυρήνα ή νήμα προγράμματος με τρόπο ασφαλή για τα δεδομένα.

Κάποιες από τις παραδοσιακές μεθόδους συγχρονισμού είναι τα κλειδώματα, οι ατομικές λειτουργίες, μνήμη συναλλαγών (transactional memory) και οι παρακολουθητές και οι μεταβλητές συνθηκών (monitors and conditional variables). Ξεκινώντας από τα κλειδώματα, είναι ίσως ο πιο απλός σε ιδέα μηχανισμός και πολλές φορές αρκετά απλός στην υλοποίηση. Η ιδέα του μοντέλου αυτού για συγχρονισμό είναι η ύπαρξη ενός ή περισσοτέρων κλειδωμάτων τα οποία θα προστατεύουν τμήματα της δομής ή ολόκληρη τη δομή. Για να αλληλεπιδράσει ένας πυρήνας ή νήμα με την δομή θα πρέπει να αποκτήσει τον έλεγχο του κλειδώματος. Το κλείδωμα μπορεί να είναι μια δομή λογισμικού η οποία να υποστηρίζεται από ατομικές πράξεις στο υλικό ή άλλες λειτουργίες υλικού. Κάθε φορά που κάποιος θέλει να αποκτήσει πρόσβαση στη μνήμη, διεκδικεί το κλείδωμα και υπάρχει συναγωνισμός ανάμεσα στους πυρήνες ή τα νήματα. Όποιος αποκτήσει το κλείδωμα μπορεί να προχωρήσει και να έχει πρόσβαση στη

δομή ενώ οι υπόλοιποι που δεν το έλαβαν κάνουν μια άλλη εργασία, ανάλογα με το τι θέλει ο προγραμματιστής ή με το ποιος είναι ο σκοπός του προγράμματος. Παραδείγματος χάρη, οι πυρήνες που δεν θα καταφέρουν να λάβουν το κλείδωμα στην κατοχή τους μπορούν να επιστρέψουν κάνοντας κάποια άλλη εργασία και να το διεκδικήσουν αργότερα, μπορούν να συνεχίσουν να το ζητάνε μέχρι να το λάβουν ή ακόμα και να μπουν σε κατάσταση αναμονής για κάποιον χρόνο και να το διεκδικήσουν αργότερα. Κάθε μία στρατηγική έχει θετικά και αρνητικά σημεία οπότε επιλέγεται ανάλογα την εφαρμογή.

Πλέον υπάρχουν πολλές ιδέες και τρόποι υλοποίησης κλειδωμάτων που δεν μπορούν να εξαντληθούν σε αυτό το κείμενο. Όταν πρέπει να διαλέξουμε ανάμεσα στις διάφορες υλοποιήσεις κλειδωμάτων, χρειάζεται να κάνουμε μια επιλογή και να ζυγίσουμε παράγοντες όπως, επίδοση του μηχανισμού και της στρατηγικής του κλειδώματος, ευκολία στη χρήση και ευκολία στην υλοποίηση. Αν και παρακάτω θα μιλήσουμε περισσότερο για τις υλοποιήσεις που χρησιμοποιήθηκαν για τα πειράματά μας, ας αναφέρουμε εδώ ότι χρησιμοποιήσαμε απλές προσεγγίσεις με coarse grain κλειδώματα και busy waiting.

Η επόμενη τεχνική συγχρονισμού που χρησιμοποιείται ευρέως και αξίζει την αναφορά μας είναι οι ατομικές λειτουργίες. Η ατομικότητα εγγυάται την απομόνωση από διεργασίες που τρέχουν ταυτόχρονα και η ατομική αρχή εγγυάται ότι οι αλληλεπιδράσεις με την μνήμη φαίνονται σαν να έγιναν άμεσα και σαν να μην υπήρχε χρονική επικάλυψη ανάμεσα σε διαφορετικά αιτήματα.

Οι ατομικές λειτουργίες μπορούν να υλοποιηθούν με δύο τρόπους, μέσω υλικού ή λογισμικού. Η υλοποίηση μέσω υλικού περιλαμβάνει είτε πρωτόκολλα συνέπειας μεταξύ κρυφών μνημών ή μέσω καταχωρητών test&set. Και στις δύο αυτές περιπτώσεις πρέπει το διαθέσιμο υλικό να έχει ήδη μια από αυτές τις δομές, δηλαδή να έχει σχεδιαστεί ώστε να παρέχει στους χρήστες και προγραμματιστές αυτή τη δυνατότητα. Η άλλη περίπτωση είναι υλοποίηση ατομικών λειτουργιών μέσω δομών λογισμικού. Για αυτές τις περιπτώσεις χρησιμοποιούμε είτε έλεγχο σε μεταβλητές είτε ελέγχο στην χρονική σφραγίδα μιας λειτουργίας, δηλαδή την χρονική στιγμή που κάποια λειτουργία σημειώθηκε ως εκτελεσμένη. Αν μετά τον έλεγχο των χρόνων προκύψει επικάλυψη ή σύγκρουση λειτουργιών που μπορεί να οδηγήσει σε λανθασμένα ή αβέβαια δεδομένα, οι εντολές αυτές ακυρώνονται και πρέπει να επαναληφθούν ή ξεχνιούνται, ανάλογα με τις προγραμματιστικές επιλογές που έχουμε κάνει.

Στην συνέχεια έχουμε την μνήμη συναλλαγών, μια πιο προχωρημένη μέθοδο συγχρονισμού. Η μνήμη συναλλαγών είναι εμπνευσμένη από τους μηχανισμούς συναλλαγών βάσεων δεδομένων (database transactions mechanisms). Είναι μια μέθοδος συγχρονισμού υψηλότερου επιπέδου και αφαιρεί βάρος από τον προγραμματιστή αφού αυτός απλά σημειώνει σε ποια σημεία θέλει συγχρονισμό και το σύστημα αναλαμβάνει και εγγυάται ότι εκεί που ζητήθηκε θα υπάρξει ατομικότητα και συνέπεια δεδομένων. Με τον τρόπο αυτό μειώνονται οι πιθανότητες λαθών, μη αποδοτικών υλοποιήσεων και χρήσεων του συγχρονισμού αφού ο προγραμματιστής δεν επιβαρύνεται με αυτά, έχουν ασχοληθεί πριν από αυτόν άλλοι ώστε να υλοποιήσουν και να παρέχουν συγχρονισμό με μνήμη συναλλαγών, είτε μέσω υλικού είτε μέσω λογισμικού.

Η τελευταία από τις απλές μεθόδους συγχρονισμού που θα σχολιάσουμε είναι οι παρακολουθητές και οι μεταβλητές συνθηκών. Οι παρακολουθητές είναι δομές που συνδυάζουν

δεδομένα, μεθόδους και συγχρονισμό σε ένα πακέτο, όπως οι κλάσεις στις γλώσσες προγραμματισμού συνδυάζουν δεδομένα και μεθόδους. Η ύπαρξη παρακολουθητών πρέπει να συνοδεύεται από υποστήριξη υλικού, είτε μέσω παροχής ατομικών λειτουργιών είτε μέσω της δυνατότητας να απενεργοποιούνται οι διακοπές (interruprts). Η ιδέα για τους παρακολουθητές προέκυψε ώστε να αντιμετωπίσει προβλήματα που εμφάνιζαν τα κλειδώματα.

Τελειώνοντας την αναφορά μας στις μεθόδους και τις αρχές του συγχρονισμού θα αναφέρουμε δυο προχωρημένες τεχνικές συγχρονισμού, οι οποίες δεν χρησιμοποιούνται ευρέως στην πράξη αλλά μελετούνται ερευνητικά και παρουσιάζουν ενδιαφέρον. Οι μέθοδοι αυτοί είναι η ενσωματωμένη μνήμη συναλλαγών και τα C-κλειδώματα. Η ενσωματωμένη μνήμη συναλλαγών σαν ιδέα προήλθε από την απλή μνήμη συναλλαγών που αναφέραμε πιο πάνω. Αρχικά προορίζονταν για χρήση σε ενσωματωμένα συστήματα αλλά πλέον δοκιμάζεται η χρήση της και σε υπολογιστές γενικού σκοπού. Η διαφορά της ενσωματωμένης μνήμης είναι ότι με προγραμματιστικές επιλογές και με πρωτόκολλα που χρησιμοποιούνται, προσπαθεί να καταναλώσει λιγότερη ενέργεια μειώνοντας το πλήθος των υποθετικών εκτελέσεων ώστε να μην χάνεται ενέργεια από τις αποτυχημένες υποθέσεις. Επίσης με βάση έρευνες μπορεί ακόμα να πετύχει και βελτίωση στην απόδοση. Το C-κλείδωμα τώρα, είναι μια μέθοδος συγχρονισμού που βασίζεται στο υλικό και απευθύνεται σε ενσωματωμένα συστήματα. Συνδυάζει τα κλειδώματα με την μνήμη συναλλαγών ώστε να προσφέρει έναν μηχανισμό συγχρονισμού που θα προσφέρει καλύτερη επίδοση. Με την υποστήριξη επιπλέον υλικού, οι πυρήνες αλληλεπιδρούν με την μνήμη με βάση τις αρχές της μνήμης συναλλαγών, αλλά αν διαπιστωθεί κάποια σύγκρουση στα δεδομένα τότε ο τρόπος συγχρονισμού αλλάζει και οι πυρήνες διεκδικούν κλειδώματα για να έχουν πρόσβαση στα δεδομένα αυτά. Με τον τρόπο αυτό προσπαθούμε να πετύχουμε μικρότερη κατανάλωση ενέργειας και να συνδυάσουμε τα πλεονεκτήματα των κλειδωμάτων με αυτά της μνήμης συναλλαγών.

1.2 SCC και υλοποιήσεις

Ας δούμε τώρα την δομή του υπολογιστικού συστήματος SCC με περισσότερες λεπτομέρειες. Όπως έχουμε αναφέρει νωρίτερα το SCC αποτελείται από 48 πυρήνες. Οι πυρήνες αυτοί είναι οργανωμένοι ανά ζευγάρια σε ψηφίδες, δηλαδή συνολικά έχουμε 24 ψηφίδες με δύο πυρήνες στην κάθε μία. Οι ψηφίδες αυτές είναι διαταγμένες σε ένα πλέγμα 6×4 και κάθε ψηφίδα έχει έναν διακομιστή (router) εντός της, ο οποίος αναλαμβάνει όλη την επικοινωνία με δομές εκτός της ψηφίδας, δηλαδή με τις υπόλοιπες ψηφίδες, με την κεντρική μνήμη ή με τον υπολογιστή που διαχειρίζεται όλο το σύστημα του SCC.

Για να λειτουργήσει το σύστημα SCC πρέπει να είναι συνδεδεμένο σε έναν υπολογιστή γενικού σκοπού, 64-bit ο οποίος μπορεί να τρέχει λειτουργικό GNU\Linux ή Windows και μέσω αυτού γίνεται η διαχείριση του SCC. Οι πυρήνες μπορούν επίσης να τρέξουν μια προσαρμοσμένη διανομή GNU\Linux αλλά αυτό δεν είναι απαραίτητο και εξαρτάται από τις επιλογές και τον στόχο του προγραμματιστή. Δηλαδή οι πυρήνες μπορούν να χρησιμοποιηθούν και χωρίς έτοιμο λειτουργικό, αν ο προγραμματιστής θέλει να έχει πρόσβαση σε χαμηλό επίπεδο προγραμματισμού των πυρήνων.

Η αρχιτεκτονική των επεξεργαστών είναι IA P54C και είναι κατασκευασμένοι σε τεχνολογία $45nm$ high K metal gate CMOS και σε όλο το chip υπάρχουν 1,3 δισεκατομμύρια τρανζίστορ. Το chip επίσης διαθέτει τέσσερις ελεγκτές μνήμης (memory controllers) οι οποίοι χρησιμοποιούνται από τους πυρήνες για να επικοινωνήσουν με την κεντρική μνήμη. Τον ποιο ελεγκτή θα χρησιμοποιήσει κάθε πυρήνας εξαρτάται από τη θέση του πυρήνα και το τμήμα της μνήμης που θέλει να προσπελάσει. Η διαδρομή δηλαδή που θα ακολουθήσει το αίτημα κάποιου πυρήνα εξαρτάται από το σύστημα και δεν το καθορίζει άμεσα ο προγραμματιστής. Επίσης από το σύστημα επιλέγεται η διαδρομή που θα ακολουθήσει κάποιο αίτημα ή μήνυμα από κάποιον πυρήνα προς κάποιον άλλο. Ο προγραμματιστής δεν επιλέγει και δεν καθορίζει με ποιον τρόπο θα κινηθεί το μήνυμα μέσα στον δίαυλο του chip για να φθάσει στον προορισμό του.

Το SCC επίσης έχει την υλική υποδομή και μας προσφέρει προγραμματιστικά εργαλεία έτσι ώστε να μπορούμε να ελέγξουμε και να μεταβάλλουμε την συχνότητα με την οποία δουλεύουν οι πυρήνες ακόμα και την τάση τροφοδοσίας τους. Για την συχνότητα έχουμε 24 διαιρέτες, δηλαδή κάθε ψηφίδα μπορεί να έχει την δική της συχνότητα, ενώ για την τάση έχουμε 7 πεδία τιμών που μπορούμε να επιλέξουμε και οι πυρήνες μπορούν να ρυθμιστούν σε ομάδες των 4. Οπότε έχουμε 6 σύνολα πυρήνων που μπορούν να έχουν διαφορετική τάση.

Ας περάσουμε τώρα να δούμε λίγο την οργάνωση της μνήμης στο σύστημα μας. Γενικά η μνήμη του συστήματος χωρίζεται σε κρυφή μνήμη εντός των ψηφίδων, καταχωρητές ανταλλαγής μηνυμάτων (message passing buffers) και αυτοί εντός των ψηφίδων και στην κύρια μνήμη η οποία βρίσκεται εκτός του chip.

Σχετικά με την κρυφή μνήμη, υπάρχει κρυφή μνήμη δύο επιπέδων. Κάθε πυρήνας έχει συνολικά 32 KB κρυφή μνήμη πρώτου επιπέδου, L1, από αυτή, τα 16 KB είναι κρυφή μνήμη εντολών και τα άλλα 16 KB κρυφή μνήμη δεδομένων. Επίσης κάθε πυρήνας έχεις τη δικιά του κρυφή μνήμη δευτέρου επιπέδου, L2, η οποία είναι εννοποιημένη και έχει συνολική χωρητικότητα 256 KB.

Σε κάθε ψηφίδα επίσης υπάρχουν 16 KB μνήμης καταχωρητών ανταλλαγής μηνυμάτων. Αυτό σημαίνει ότι συνολικά στο chip έχουμε 382 KB τέτοιας μνήμης, αφού υπάρχουν 24 ψηφίδες. Η μνήμη αυτή είναι τύπου SRAM οπότε είναι αρκετά γρήγορη. Κάθε πυρήνας έχει δικαίωμα να γράψει και να διαβάσει σε οποιονδήποτε καταχωρητή, ανεξάρτητα σε ποια ψηφίδα βρίσκεται. Επομένως είναι μια μοιραζόμενη, κατανεμημένη μνήμη εντός του chip.

Η κύρια μνήμη εκτός του chip μπορεί να κυμανθεί μεταξύ 16 GB και 64 GB. Κάθε ελεγκτής μνήμης, από τους 4 που υπάρχουν, μπορεί να διευθυνσιοδοτήσει από 4 έως 16 GB και κάθε πυρήνας μπορεί να δει έως 4 GB κύριας μνήμης. Η κύρια αυτή μνήμη είναι τύπου DRAM και μπορεί να λειτουργήσει τόσο ως ιδιωτική όσο και ως μοιραζόμενη. Τα δεδομένα από την μνήμη αυτή αντιγράφονται στις κρυφές μνήμες των δύο επιπέδων κάτι όμως που δεν συμβαίνει με τα δεδομένα που προέρχονται από καταχωρητές ανταλλαγής μηνυμάτων. Τα δεδομένα που προέρχονται από αυτούς τους καταχωρητές είναι κατάλληλα σημειωμένα και δεν παραμένουν στην κρυφή μνήμη.

Ένα τελευταίο χαρακτηριστικό των μνημών που θα φανεί σημαντικό στην επεξήγηση των αποτελεσμάτων αργότερα είναι το ότι δεν υπάρχει πρωτόκολλο συνάφειας μεταξύ των κρυφών

μνημών των διαφόρων πυρήνων. Αυτό έχει ως συνέπεια, όταν χρησιμοποιούμε την κύρια μνήμη ως μοιραζόμενη, να είμαστε αναγκασμένη κάθε φορά που μεταβάλλουμε κάποιο μοιραζόμενο δεδομένο, για να είναι αυτό εμφανές σε όλους τους πυρήνες να πρέπει να γραφτεί αμέσως στην κύρια μνήμη. Ακόμα, αν θέλουμε να διαβάσουμε ένα μοιραζόμενο δεδομένο, το οποίο το έχουμε διαβάσει και νωρίτερα, δεν πρέπει να κοιτάμε στην κρυφή μνήμη αλλά πρέπει να το φέρνουμε εκ νέου από την κύρια μνήμη, γιατί μπορεί κάποιος άλλος πυρήνας στο μεταξύ, να το έχει μεταβάλει. Για να λύσουμε όλο αυτό το πρόβλημα, το προγραμματιστικό μοντέλο του SCC μας παρέχει μια εντολή καθαρισμού της κρυφής μνήμης από τα μοιραζόμενα δεδομένα. Αυτό σημαίνει ότι πριν διαβάσουμε ένα δεδομένο της μοιραζόμενης μνήμης ή αφού γράψαμε σε αυτό, πρέπει να εκτελέσουμε την εντολή εκκένωσης της κρυφής μνήμης από τα μοιραζόμενα δεδομένα.

Τελειώνοντας την αναφορά και την παρουσίαση του συστήματος, πρέπει να αναφερθούμε στο προγραμματιστικά μοντέλα που είναι διαθέσιμα και σε αυτό που χρησιμοποιήσαμε για την ανάπτυξη των εφαρμογών που χρειάστηκαν για τις εργασίες της διπλωματικής αυτής.

Αρχικά να πούμε ότι αν κάποιος δεν έχει πρόσβαση σε κάποιο φυσικό μηχάνημα SCC, τότε υπάρχει και ένας προσομοιωτής του περιβάλλοντος. Αυτός λειτουργεί σε υπολογιστές με λειτουργικά συστήματα GNU\Linux ή Windows και βασίζεται στην γλώσσα OpenMp για να υλοποιήσει τις παραλληλίες. Ωστόσο, όσοι χρησιμοποιούν τον προσομοιωτή αυτόν θα πρέπει να είναι ενήμεροι ότι δεν είναι ασφαλές να εξάγονται συμπεράσματα ως προς μετρήσεις γιατί οι χρόνοι και άλλα στοιχεία, όπως η κατανάλωση είναι πολύ πιθανόν να διαφέρουν από το πραγματικό σύστημα.

Στο πραγματικό σύστημα τώρα, όπως είπαμε και πριν υπάρχει η επιλογή να φορτώσουμε κάποια ειδική διανομή GNU\Linux για τους πυρήνες ή να τους χρησιμοποιήσουμε χωρίς λειτουργικό σύστημα, στην Baremetal εκδοχή, όπως λέγεται. Προφανώς αν επιλέξουμε να φορτώσουμε λειτουργικό σύστημα στους πυρήνες θα έχουμε την δυνατότητα να χρησιμοποιήσουμε και να επωφεληθούμε από πολλά έτοιμα εργαλεία λογισμικού που είναι συμβατά με το λειτουργικό. Οι εφαρμογές που γράφουμε για τους πυρήνες του SCC μπορούν να είναι είτε σε γλώσσα C είτε σε γλώσσα Fortran, γιατί για αυτές τις δύο γλώσσες παρέχονται μεταγλωττιστές από την Intel.

Για την περαιτέρω βοήθεια και στήριξη της προγραμματιστικής κοινότητας του SCC, παρέχεται μια επιπλέον προγραμματιστική διεπαφή, με την μορφή μιας βιβλιοθήκης που εμπλουτίζει τη γλώσσα C. Η προγραμματιστική αυτή διεπαφή λέγεται RCCE και παρέχει έναν αριθμό έτοιμων συναρτήσεων που βοηθάνε σημαντικά και απλοποιούν το προγραμματιστικό έργο. Για την ανάπτυξη των εφαρμογών για τις μετρήσεις αυτής της διπλωματικής, χρησιμοποιήσαμε την βιβλιοθήκη RCCE και τις συναρτήσεις που προσφέρει για επικοινωνία των πυρήνων, για υλοποίηση σημαιών που είναι χρήσιμες προγραμματιστικές δομές για συγχρονισμό και επικοινωνία αλλά και άλλες συναρτήσεις που βοήθησαν σημαντικά.

Ας δούμε τώρα με περισσότερες λεπτομέρειες τις υλοποιήσεις που χρησιμοποιήσαμε για τα διάφορα μοντέλα κάθε δομής δεδομένων του αξιολογήσαμε. Για την στοίβα υλοποιήσαμε τρία μοντέλα, ένα πελάτη-εξυπηρετητή, ένα μοντέλο πελάτη-εξυπηρετητή με εξάλειψη, θα εξηγήσουμε παρακάτω τι σημαίνει αυτό, και ένα μοντέλο με ένα κλείδωμα για όλη τη δομή.

Για την FIFO ουρά υλοποιήσαμε επίσης τρία μοντέλα, ένα πελάτη-εξυπηρετητή, ένα μοντέλο με ένα κλείδωμα και ένα μοντέλο με δύο κλειδώματα. Τέλος για τον σωρό υλοποιήσαμε δύο μοντέλα, ένα πελάτη εξυπηρετητή και ένα με ένα κλείδωμα.

Ξεκινώντας ας μιλήσουμε πρώτα για την δομή των υλοποιήσεων της στοίβας. Όπως αναφέραμε και πιο πάνω, για την ανάπτυξη των εφαρμογών μας χρησιμοποιήσαμε την διεπαφή RCCE και προγραμματίσαμε σε γλώσσα C. Στην πρώτη υλοποίηση, στο μοντέλο πελάτη-εξυπηρετητή, από τους N πυρήνες που δεσμεύουμε, οι $N-1$ είναι πελάτες και ένας πυρήνας είναι ο εξυπηρετητής. Ο ρόλος του εξυπηρετητή είναι να λαμβάνει αιτήματα από τους πελάτες και να τα υλοποιεί. Η δομή δεδομένων βρίσκεται στην ιδιωτική μνήμη του εξυπηρετητή και μόνο αυτός έχει πρόσβαση σε αυτήν. Οι πελάτες στέλνουν τα αιτήματά τους στον εξυπηρετητή και περιμένουν μέχρι να υλοποιηθούν για να προχωρήσουν στο επόμενο. Η ανταλλαγή αυτή γίνεται μέσω της μνήμης των καταχωρητών ανταλλαγής μηνυμάτων. Συγκεκριμένα οι δομή δεδομένων δέχεται ακεραίους των 32 bit ως στοιχεία. Επομένως οι πελάτες γράφουν στον τοπικό τους καταχωρητή των αριθμό που θέλουν να προσθέσουν στη δομή, ειδοποιούν μέσω σημαιών τον εξυπηρετητή για το αίτημά τους, ο εξυπηρετητής όταν κληθεί να υλοποιήσει αυτό το αίτημα διαβάζει από τον καταχωρητή του πελάτη τον ακέραιο και τον προσθέτει στην δομή. Έπειτα ενημερώνει κατάλληλα τον πελάτη ότι το αίτημα έχει ολοκληρωθεί. Αντίστοιχα όταν κάποιος πελάτης ζητάει να αφαιρεθεί κάποιο στοιχείο από τη δομή περιμένει μέχρι ο εξυπηρετητής να γράψει τον ακέραιο στον καταχωρητή τού πελάτη, από όπου ο δεύτερος τον διαβάζει και συνεχίζει την λειτουργία του.

Η δομή των προγραμμάτων είναι η εξής, αρχικά κάνουμε τις κατάλληλες δεσμεύσεις μνήμης για την δομή, για τις σημαίες που θα χρησιμοποιήσουμε και για τους καταχωρητές ανταλλαγής μηνυμάτων. Στη συνέχεια αρχικοποιούμε την δομή μας ώστε να υπάρχουν κάποια δεδομένα μέσα από την αρχή. Έπειτα οι πυρήνες χωρίζονται σε δύο μέρη και το καθένα εκτελεί διαφορετικό κομμάτι κώδικα. Οι πελάτες εισέρχονται σε έναν βρόχο, ο οποίος τρέχει καθορισμένες από εμάς φορές, ανάλογα με το πόσα αιτήματα θέλουμε να υποβάλλουμε. Αφού εκτελέσουν όλα τα αιτήματά τους, βγαίνουν από το βρόχο και μπορούν να τερματίσουν.

Από την άλλη ο εξυπηρετητής εισέρχεται σε έναν βρόχο ο οποίος επαναλαμβάνεται συνεχώς ελέγχοντας μόνο μια συνθήκη. Η συνθήκη αυτή ικανοποιείται όταν όλοι οι πελάτες έχουν ολοκληρώσει τα αιτήματά τους, τότε ο εξυπηρετητής σταματάει να επαναλαμβάνει τον βρόχο, προχωράει σε επόμενο κομμάτι όπου τυπώνει τα αποτελέσματα, τον χρόνο δηλαδή που έτρεξε το πρόγραμμα και μετά τερματίζει.

Μέσα στον βρόχο τώρα, υπάρχουν δύο εμφωλευμένοι βρόχοι. Στον πρώτο ο εξυπηρετητής ελέγχει με τη σειρά δύο πίνακες που περιέχουν σημαίες. Οι σημαίες είναι μια προγραμματιστική δομή που μας παρέχει η διεπαφή RCCE. Θα μπορούσαμε να πούμε ότι μοιάζει με μία μεταβλητή τύπου boolean αφού μπορεί να πάρει μόνο δύο τιμές. Η δομή αυτή υλοποιείται σε μνήμη καταχωρητών ανταλλαγής μνήμης, οπότε με τις συναρτήσεις που επίσης παρέχονται από την διεπαφή RCCE ένας πυρήνας μπορεί να διαβάσει οποιαδήποτε σημαία άλλου πυρήνα. Έτσι έχουμε για τις εφαρμογές μας έναν εύκολο τρόπο συγχρονισμού και ανταλλαγής μηνυμάτων. Συγκεκριμένα κάθε πελάτης έχει δύο σημαίες, μια για κάθε είδος αιτήματος που μπορεί να κάνει, εισαγωγή ή εξαγωγή στοιχείου. Ανάλογα με το τι θέλει να κάνει ο πελάτης θέτει την

κατάλληλη σημαία και στην συνέχεια ο εξυπηρετητής ελέγχοντας συνεχώς όλες τις σημαίες μπορεί να δει αυτή τη θέση της σημαίας και έτσι να ενημερωθεί για το αίτημα και να προχωρήσει στην εξυπηρέτησή του. Αφού σε κάθε επανάληψη του εξωτερικού βρόχου ελέγξει όλες τις σημαίες, όλων των πελατών και για τους δύο τύπους αιτημάτων, προχωράει στον επόμενο βρόχο, στον οποίο ελέγχει τις σημαίες τερματισμού. Με τις σημαίες αυτές κάθε πελάτης ενημερώνει πότε έχει ολοκληρώσει τα αιτήματα του. Έτσι μπορεί και ο εξυπηρετητής να ξέρει πότε δεν υπάρχουν άλλα αιτήματα προς εξυπηρέτηση και μπορεί να σταματήσει να ελέγχει τις σημαίες αιτημάτων. Ο εξυπηρετητής λοιπόν ελέγχει κάθε φορά με τη σειρά τις σημαίες τερματισμού ώσπου να βρει κάποια που δεν είναι σε θέση. Αυτό σημαίνει ότι τουλάχιστον ένας πελάτης έχει αιτήματα που περιμένουν, άρα επαναλαμβάνει τον εξωτερικό βρόχο. Αν όμως όλες οι σημαίες τερματισμού είναι σε θέση, τότε μπορεί και ο εξυπηρετητής να τερματίσει.

Με λίγα λόγια αυτή είναι η λειτουργία του μοντέλου πελάτη εξυπηρετητή. Αυτό που κάναμε στην υλοποίηση με την εξάλειψη είναι ότι κρατήσαμε όλο το προηγούμενο πρόγραμμα ίδιο αλλά προσθέσαμε έναν επιπλέον έλεγχο. Αυτό που γίνεται με την εξάλειψη είναι ότι όταν ο εξυπηρετητής χειρίζεται ένα αίτημα προσθήκης στοιχείου στη δομή, ελέγχει το αμέσως επόμενο αίτημα αν είναι για αφαίρεση στοιχείου από τη δομή. Αν αυτό συμβεί τότε δεν υπάρχει λόγος να πειράξει την στοίβα, αντιγράφει απλά το στοιχείο που θα προσέθετε στον καταχωρητή του πελάτη που έκανε αίτημα αφαίρεσης στοιχείου. Έτσι γλιτώνουμε προσβάσεις στην κύρια μνήμη, με το κόστος όμως ενός επιπλέον ελέγχου. Αργότερα στα αποτελέσματα θα δούμε πώς απέδωσε αυτή η τεχνική. Δηλαδή η αλλαγή σε σχέση με το προηγούμενο μοντέλο είναι ο επιπλέον έλεγχος του επόμενου αιτήματος στην περίπτωση προσθήκης στοιχείου και η κατάλληλη μετακίνηση του στοιχείου αυτού.

Ας μιλήσουμε τώρα για το μοντέλο με το κλείδωμα. Το μοντέλο αυτό διαφέρει σημαντικά με το προηγούμενο, η δομή δεδομένων βρίσκεται σε μοιραζόμενη μνήμη εκτός του chip και έχουν πρόσβαση σε αυτή όλοι οι πυρήνες. Σε αυτό το μοντέλο δεν υπάρχει κάποιος διαμεσολαβητής ή κάποιος που αναλαμβάνει να χειριστεί αιτήματα. Κάθε πυρήνας διεκδικεί πρόσβαση στη δομή και εκτελεί την εργασία του. Η πρόσβαση στην δομή γίνεται μέσω ενός κεντρικού κλειδώματος. Για να υλοποιήσουμε αυτό το κλείδωμα χρησιμοποιήσαμε τον καταχωρητή *test&set* που υπάρχει στο υλικό κάθε πυρήνα. Συγκεκριμένα χρησιμοποιήσαμε πάντα τον καταχωρητή του πυρήνα με το νούμερο #0. Οπότε κάθε πυρήνας που θέλει να αλληλεπιδράσει με την στοίβα, διεκδικεί το κλειδί που υλοποιείται από τον καταχωρητή που αναφέραμε. Το κλειδί αυτό είναι busy waiting και θα δούμε αργότερα πώς αυτό επηρεάζει την επίδοση των προγραμμάτων.

Στις εφαρμογές αυτού του μοντέλου αρχικά γίνονται οι δεσμεύσεις μνήμης από τους πυρήνες και στη συνέχεια αρχικοποιείται η μνήμη, όπως κάναμε και στο μοντέλο πελάτη-εξυπηρετητή. Έπειτα οι πυρήνες μπαίνουν σε ένα βρόχο του οποίου οι επαναλήψεις εξαρτώνται από το πόσα αιτήματα θέλουμε να υλοποιήσει κάθε πυρήνας. Μέσα σε αυτό τον βρόχο, ανάλογα με το σενάριο που θέλουμε να εκτελέσουν οι πυρήνες, κάνουν είτε προσθήκες είτε αφαιρέσεις στοιχείων από την στοίβα. Αφού κάποιος πυρήνας τελειώσει με τα αιτήματά του τότε βγαίνει από το βρόχο και μπλοκάρει σε ένα φράγμα (barrier) ώσπου όλοι οι πυρήνες να τελειώσουν. Τότε, αφού δηλαδή όλοι οι πυρήνες φθάσουν στο φράγμα, ένας πυρήνας

αναλαμβάνει να τυπώσει τα δεδομένα που μετρήσαμε και έπειτα όλοι τερματίζουν.

Όπως έχουμε αναφέρει και πιο πάνω, με την χρήση της μοιραζόμενης μνήμης πρέπει να φροντίζουμε για την σωστή εκδοχή των δεδομένων στις κρυφές μνήμες των πυρήνων. Οπότε σε αυτό το μοντέλο, με κάθε προσθήκη ή αφαίρεση στοιχείου από κάποιον πυρήνα αναγκαζόμαστε να κάνουμε συχνά εκκαθάριση της κρυφής μνήμης από τα μοιραζόμενα δεδομένα ώστε να είναι η τελευταία εκδοχή τους εμφανής σε όλους τους πυρήνες.

Προχωρώντας τώρα στις υλοποιήσεις της FIFO ουράς, τα μοντέλα πελάτη-εξυπηρετητή και ενός κλειδώματος δεν διαφέρουν ιδιαίτερα από τις υλοποιήσεις της στοίβας. Το μοντέλο πελάτη-εξυπηρετητή δεν αλλάζει σημαντικά αν εξαιρέσει κανείς ότι οι συναρτήσεις για τις πράξεις με την δομή δεδομένων είναι προφανώς διαφορετικές από αυτές για την στοίβα. Το μοντέλο με το κλείδωμα έχει και εδώ ένα κλειδί που υλοποιείται με τον καταχωρητή *test&set* του πυρήνα #0 και κλειδώνει όλη τη δομή, και τα δύο άκρα.

Από την άλλη μεριά το μοντέλο με τα δύο κλειδώματα εμφανίζεται σαν υλοποίηση μόνο στην δομή της ουράς. Εδώ χρησιμοποιήσαμε δύο κλειδιά, ένα στον πυρήνα #0 και ένα στον πυρήνα #1. Το ένα χρησιμοποιείται για την πρόσβαση στο άκρο που προσθέτονται στοιχεία και το άλλο για το άκρο που αφαιρούνται στοιχεία. Στα προγράμματά μας φυσικά κάνουμε τους κατάλληλους ελέγχους ώστε αν αδειάσει κάποια στιγμή η δομή να μην μπορεί ο δείκτης του ενός άκρου να ξεπεράσει τον άλλο. Οι πυρήνες διεκδικούν το αντίστοιχο κλειδί ανάλογα με την δουλειά που θέλουν να κάνουν.

Τέλος για την δομή του σωρού, είχαμε δύο υλοποιήσεις οι οποίες επίσης δεν διαφέρουν ιδιαίτερα από αυτές ας πούμε της στοίβας. Η διαφορά στις υλοποιήσεις του σωρού, εκτός από τις διαφορετικές συναρτήσεις για την αφαίρεση μεγίστου και προσθήκη στοιχείου στην δομή, είναι ότι κατά την αρχικοποίηση της δομής, και στα δύο μοντέλα, εισάγουμε τα μισά στοιχεία από αυτά που θέλουμε για την αρχικοποίηση, μετά καλούμε την συνάρτηση που δημιουργεί τον σωρό και τα υπόλοιπα εισάγονται τηρώντας τις ιδιότητες του σωρού. Αυτό γίνεται για να εξοικονομήσουμε λίγο χρόνο στην αρχικοποίηση.

1.3 Αξιολόγηση δομών δεδομένων

Σε αυτό το σημείο ας αναφερθούμε με περισσότερες λεπτομέρειες στην δομή των σεναρίων που εκτελούσαν τα προγράμματα και στα μεγέθη τα οποία μετρήσαμε και αξιολογήσαμε. Συνολικά χρησιμοποιήσαμε τέσσερα διαφορετικά σενάρια για να αξιολογήσουμε τις δομές μας. Τρία από αυτά είχαν ίσο αριθμό αφαιρέσεων και προσθηκών στοιχείων στη δομή ενώ χρησιμοποιήσαμε και ένα σενάριο με τυχαίο αριθμό από τα είδη των αιτημάτων, ο τυχαίος αυτός αριθμός δημιουργήθηκε με την συνάρτηση rand της γλώσσας C.

Το πρώτο από τα σενάρια με ίσο αριθμό προσθηκών και αφαιρέσεων είχε τυχαία κατανεμημένο τον τύπο των διεργασιών με την δομή. Δηλαδή κατασκευάσαμε έναν πίνακα 200 θέσεων, που περιείχε τους αριθμούς 0 και 1. Κάθε πυρήνας ξεκίναγε να διαβάζει τον πίνακα από την θέση που αντιστοιχούσε στο ID του. Κάθε φορά κινούνταν κατά μια θέση μέσα στον πίνακα, αν τα αιτήματα ήταν πάνω από 200 ή έφτανε στο τέλος του πίνακα απλά ξεκίναγε πάλι από τη θέση 0. Ανάλογα με τον αριθμό που θα διάβαζε εκτελούσε την αντίστοιχη εργασία, είτε

προσθήκη είτε αφαίρεση.

Το δεύτερο σενάριο που είχε συνολικά ίσο αριθμό προσθηκών και αφαιρέσεων στοιχείων βασίζονταν στην παραδοχή ότι το πρώτο μισό των πυρήνων θα έκαναν μόνο προσθήκες και το δεύτερο μισό μόνο αφαιρέσεις στοιχείων. Για παράδειγμα σε ένα μοντέλο οποιασδήποτε δομής με ένα κλείδωμα, αν τρέχαμε την εφαρμογή μας με 12 πυρήνες, οι 6 με τα μικρότερα αναγνωριστικά (0, 1, 2, 3, 4, 5) θα έκανα συνέχεια προσθήκες , ενώ οι υπόλοιποι 6 πυρήνες μόνο αφαιρέσεις στοιχείων από τη δομή. Συνολικά όμως θα γίνονταν ίσο πλήθος αφαιρέσεων και προσθηκών.

Το τρίτο και τελευταίο σενάριο που είχε μοιρασμένα αιτήματα προσθηκών και αφαιρέσεων στοιχείων βασίζονταν σε διαφορετική κατανομή των αιτημάτων σε σχέση με τα δύο προηγούμενα. Και εδώ οι μισοί πυρήνες έκαναν μόνο προσθήκες και οι υπόλοιποι μόνο αφαιρέσεις αλλά αυτή τη φορά άλλαζε η κατανομή τους στο chip. Οι πυρήνες με περιττό αναγνωριστικό (ID) θα κάνουν μόνο προσθήκες, ενώ οι πυρήνες με άρτιο αναγνωριστικό θα αφαιρούν στοιχεία. Δηλαδή αν τρέχουμε ένα πρόγραμμα με 10 πυρήνες και υλοποίηση με ένα κλείδωμα, οι πυρήνες με αναγνωριστικά 1, 3, 5, 7, 9 θα έκαναν μόνο προσθήκες στοιχείων στην δομή, ενώ οι υπόλοιπη θα αφαιρούσαν στοιχεία. Έτσι σε κάθε ψηφίδα θα είχαμε έναν πυρήνα να προσθέτει στοιχεία και έναν πυρήνα μόνο να αφαιρεί.

Τέλος πήραμε μετρήσεις και με ένα σενάριο που είχε τυχαίο πλήθος αφαιρέσεων στοιχείων και προσθηκών. Αποτελούνταν και αυτό από έναν πίνακα 200 θέσεων με 0 και 1 τον οποίο επαναλάμβανε κάθε πυρήνας μέχρι να ολοκληρώσει όλα του τα αιτήματα. Ξεκίναγε την ανάγνωση του πίνακα και εδώ ανάλογα με το αναγνωριστικό του (ID) και κάθε φορά προχώραγε μια θέση.

Ένα άλλο χαρακτηριστικό της πλειοψηφίας των πειραμάτων που πραγματοποιήσαμε είναι ότι δεσμεύαμε τους πυρήνες με τη σειρά και όχι διασκορπισμένους. Δηλαδή αν θέλαμε να τρέξουμε ένα πρόγραμμα χρησιμοποιώντας 20 πυρήνες τότε δεσμεύαμε τους πυρήνες από #0 έως #19. Στα περισσότερα πειράματα ακολουθήσαμε αυτή την τακτική, αλλά πραγματοποιήσαμε και μερικά πειράματα για να δούμε πόσο επηρεάζει η θέση των πυρήνων την επίδοση.

Ας μιλήσουμε τώρα για τα μεγέθη που μετρήσαμε και αξιολογήσαμε, οι μετρήσεις φαίνονται στο κύριο μέρος της εργασίας. Αρχικά μετρήσαμε τις χρονικές επιδόσεις των δομών. Συγκρίναμε για κάθε δομή δεδομένων πώς συμπεριφέρονται χρονικά οι διαφορετικές υλοποιήσεις και τα διαφορετικά μοντέλα συγχρονισμού. Επίσης φτιάξαμε και διαγράμματα για την απόδοση (throughput) των δομών δεδομένων. Για τις μετρήσεις αυτές χρησιμοποιήσαμε τα σενάρια που παρουσιάσαμε παραπάνω.

Έπειτα μετρήσαμε την κατανάλωση ενέργειας των προγραμμάτων. Το σύστημα SCC παρέχει την δυνατότητα να παίρνουμε μετρήσεις της τάσης και του ρεύματος ανά τακτά χρονικά διαστήματα. Έτσι με την επεξεργασία αυτών των μετρήσεων μπορούμε να υπολογίσουμε την ενέργεια που καταναλώνεται από το σύστημα όταν εκτελείται μια εφαρμογή. Και εδώ πήραμε μετρήσεις για τα σενάρια που αναφέραμε νωρίτερα.

Ακόμη αξιολογήσαμε κατά πόσο παίζει ρόλο η κατανομή των πυρήνων στο chip και συγκρίναμε τις χρονικές επιδόσεις μιας διασκορπισμένης κατανομής σε σχέση με την συνεχή δέσμευση των πυρήνων. Είδαμε επίσης αν παίζε ρόλο η θέση του πυρήνα εξυπηρετητή στα

μοντέλα πελάτη-εξυπηρετητή κάθε δομής δεδομένων. Συγκρίναμε δύο τακτικές, μία ο εξυπηρετητής να είναι πάντα ο πρώτος πυρήνας, δηλαδή αυτός με αναγνωριστικό #0, και μια ο εξυπηρετητής να είναι κάθε φορά ο μεσαίος σε σχέση με αυτούς που έχουμε δεσμεύσει.

Κλείνοντας, μετρήσαμε κατά πόσο είναι δίκαιες οι μέθοδοι συγχρονισμού. Δηλαδή πόσο δίκαιο είναι το μοντέλο πελάτη-εξυπηρετητή και πόσο δίκαια είναι τα κλειδώματα. Και είδαμε ακόμη πώς θα συμπεριφέρονταν το σύστημα και οι δομές δεδομένων αν μεσολαβούσα κάποια καθυστέρηση μεταξύ των αιτημάτων κάθε πυρήνα. Δοκιμάσαμε με διάφορα μεγέθη καθυστέρησης και ξεκινήσαμε από μικρή καθυστέρηση αυξάνοντάς την σταδιακά. Η καθυστέρηση αυτή θα μπορούσε να προσομοιώσει ένα πραγματικό σύστημα στο οποίο μεταξύ των αιτημάτων θα μεσολαβούσαν κάποιοι υπολογισμοί από κάθε πυρήνα ή θα ασχολούνταν οι πυρήνες και με άλλες εργασίες.

1.4 Συμπεράσματα και προεκτάσεις

Καταλήγοντας στα συμπεράσματα αυτής της εργασίας, τα οποία προέκυψαν από τις μετρήσεις που πήραμε παραθέτουμε τα παρακάτω σχόλια.

Όσον αφορά την χρονική επίδοση, για τις δομές της στοίβας και της ουράς, βλέπουμε ότι μετά από έναν αριθμό πυρήνων, το μοντέλο πελάτη-εξυπηρετητή έχει καλύτερη επίδοση από αυτό με το ένα κλείδωμα. Ωστόσο για λίγους πυρήνες έχουν ίδια ή μερικές φορές το μοντέλο ενός κλειδώματος έχει καλύτερη επίδοση. Η υλοποίηση με δύο κλειδώματα για την ουρά βλέπουμε ότι παρουσιάζει την καλύτερη επίδοση και είναι πάντα καλύτερη από το μοντέλο πελάτη-εξυπηρετητή. Από την άλλη η υλοποίηση της στοίβας με εξάλειψη παρατηρούμε ότι δεν πέτυχε καλύτερες επιδόσεις από τις άλλες δύο υλοποιήσεις. Για τον σωρό βέβαια βλέπουμε την σημαντικά καλύτερη επίδοση του μοντέλου πελάτη-εξυπηρετητή από αυτό με το κλείδωμα και αυτό κυρίως οφείλεται στην κακή αξιοποίηση της κρυφής μνήμης στο μοντέλο με το κλείδωμα, λόγω μοιραζόμενης μνήμης.

Όσον αφορά την κατανάλωση ενέργειας, βλέπουμε ότι εξαρτάται αρκετά από τον χρόνο που χρειάζεται κάθε πρόγραμμα, οπότε προκύπτουν ανάλογα αποτελέσματα με αυτά που είδαμε για τον χρόνο.

Η θέση των πυρήνων και η διασπορά τους βλέπουμε ότι επηρεάζει την χρονική επίδοση σε κάποια περιοχή των πυρήνων, μεταξύ 28 και 47 πυρήνων. Από την άλλη δεν βλέπουμε σημαντική διαφορά αλλάζοντας την θέση του πυρήνα εξυπηρετητή, πολλές φορές έχουμε ελάχιστα καλύτερη επίδοση με τον εξυπηρετητή στην θέση ‘0.

Τέλος το μοντέλο πελάτη-εξυπηρετητή είναι ένα σαφώς πιο δίκαιο μοντέλο από τα κλειδώματα τα οποία έχουν αρκετές διακυμάνσεις στο πόσο δίκαια δίνουν πρόσβαση στη δομή στους διάφορους πυρήνες. Ακόμη, με την εισαγωγή καθυστέρησης έχουμε ενδιαφέροντα αποτελέσματα που δείχνουν ότι θα μπορούσαμε να έχουμε βελτίωση στην απόδοση των προγραμμάτων αν εισάγαμε καθυστέρηση μεταξύ των αιτημάτων κάθε πυρήνα.

Ως προεκτάσεις αυτής της εργασίας η μελλοντική έρευνα θα μπορούσαμε να προτείνουμε την εξέταση και άλλων δομών δεδομένων. Ένα στοιχείο επίσης που δεν εξετάσαμε είναι αν οι πυρήνες όσο περίμεναν θα μπορούσαν να μπαίνουν σε μία κατάσταση αναμονής αντί να

διεκδικούν ενεργητικά τα κλειδώματα συνεχώς, μέχρι να τα λάβουνε.

Μια άλλη δυνατότητα που μας δίνει το σύστημα SCC αλλά δεν χρησιμοποιήσαμε είναι η ρύθμιση της συχνότητας και της τάσης κάθε πυρήνα. Ρυθμίζοντας τις παραμέτρους αυτές όταν οι πυρήνες είναι σε αναμονή θα μπορούσαμε να πετύχουμε εξοικονόμηση ενέργειας και πιθανώς να μεταβάλλονταν και άλλα χαρακτηριστικά της επίδοσης.

Κλείνοντας, ένα μοντέλο που θα μπορούσαμε να δοκιμάσουμε για να δούμε πώς αποδίδει θα ήταν ένα υβριδικό ανάμεσα στο μοντέλο με κλειδώματα και στο μοντέλο πελάτη-εξυπηρετητή. Δηλαδή θα μπορούσαν ανά ομάδες οι πυρήνες να έχουν ένα εξυπηρετητή, δηλαδή να υπάρχουν περισσότερη του ενός εξυπηρετητές οι οποίοι όμως θα εξυπηρετούσαν λιγότερους πυρήνες από ότι δοκιμάσαμε στην εργασία μας. Αυτοί οι εξυπηρετητές θα έχουν μικρές δομές σε ιδιωτική τους μνήμη. Όταν αυτές οι δομές γεμίσουν τότε θα μπορούσε να υπάρχει η κύρια δομή σε μοιραζόμενη μνήμη και οι εξυπηρετητές να συγχρονίζονται μεταξύ τους μέσω κλειδωμάτων για να αντιγράφουν τις τοπικές τους δομές στην κύρια δομή και μετά να ελευθερώνουν την ιδιωτική τους μνήμη και να συνεχίζουν να δέχονται αιτήματα από τους πελάτες τους.

Contents

List of Figures

Chapter 1

Introduction

1.1 Objective

The objective of this thesis is to evaluate concurrent data structures on the Intel Single-chip Cloud Computer and compare different mechanisms of synchronization methods for those data structures. As multicore systems occupy more and more space in the computing machines domain and the software must adapt and utilize multicore systems and parallelism as effectively as possible, there is an open space to research and examine the power and the abilities that these systems can offer. Also there is need to evaluate the existing programming techniques for multicore architectures and if possible design new techniques or primitives that could make a breakthrough in the the multicore and parallel computing society.

The Single-chip Cloud Computer is an experimental platform by Intel to help and encourage the computer science society to research further manycore systems. As a 48 core system it poses a new challenge for researchers to evaluate and understand how easy and effective it is to program for a platform like this. Also how one can effectively utilize this computer. The demand for concurrent data structures and the further understanding of their efficiency in manycore systems like the SCC was a major objective for us to experiment with the SCC and try to offer and add some more knowledge on the usage and the aspects of this computer system.

Based on the already existing research and material on the Single-chip Cloud Computer ([10], [11] and [15]), our goal was to further investigate the concurrent data structures on this machine and add some knowledge on

the SCC and multicore community. The interesting part about the Single-chip Cloud Computer is the special memory architecture that exists and this gave us an incentive to research how using different kinds of this memory can affect performance, power consumption and other aspects of the concurrent data structures. In our experiments we tried to make clear if a trade-off between faster memory but not so popular synchronization methods and slower memory with easier and at times more effective methods is worth.

To add to the mentioned above, another objective was to have some data about the performance and the consumption of concurrent data structures on a manycore general purpose computer, like the Single-chip Cloud Computer, that someone can use to compare with multicore or manycore embedded systems.

1.2 Tools and programs

In this section we are going to mention the tools and programming suites that helped during the work for this thesis, the development of programs and the editing of this text. We tried to base our work on free software when that was possible but of course to complete this thesis proprietary software was also used.

To begin with, the editors used for the development of the data structures libraries, the programs simulating and running the concurrent data structures on the Single-chip Cloud Computer and the scripts that helped automate the running of the programmes, the gathering of the results and the manipulation and usage of these results were *Vim* and *gedit*. *Vim* is a cross-platform editor which is free and open source software. It was originally written by Bram Moolenaar and released publicly in 1991. It supports both a command-line interface and a graphical user interface application. On the other hand, *gedit* was released in 1999 and was the work of 7 developers. It is also a cross-platform editor, based on a simple graphical user interface and released as free and open-source software.

For the designing of all the graphs present in this thesis, the *gnuplot* application was used. *Gnuplot* was released in 1986 and is a cross-platform command-line program that is used to plot functions and draw diagrams. It is a piece of free software and nowadays there are third-party programs available that use *gnuplot* as an engine but offer a graphical user interface.

Another piece of software that proved really useful for the work needed to be done during this thesis was *GNU Octave*. It is designed by John Eaton and others and released in 1988 but not in the nowadays form. *GNU Octave* is a free software high-level programming language originally developed to support and ease numerical computations. It provides a command-line interface but also a graphical user interface and it shows compatibility with MATLAB, an non-free equivalent software, maybe the most popular for this kind of use. *GNU Octave* can be used to solve equations, plot figures and also programming algorithms.

Finally, for the developing of the text of this thesis and the typesetting, LaTeXwas used. LaTeXwas released initially in 1985 from Leslie Lamport and it is based on Donald Knuth's TeX. It is a word processor and a document mark-up language that differs from famous word processors as the result is not displayed immediately but the user writes in plain text and the input has to be compiled to see the output. LaTeXis a free software and is considered a good option for high quality typing products.

1.3 Thesis structure

At this point we are going to present the structure of the thesis and talk about the chapters that are going to follow. This thesis consists of 6 chapters and we are going to describe the content of each one, except of the first.

To begin with, the next chapter,chapter number 2, has the title "Concurrent data structures". In that chapter we are going to present synchronization primitives, both basic ones and some more advanced. We are going to have an idea of the widely used primitives for synchronization on concurrent data structures but also present new ideas that exist in research in the last years.

Next in chapter number 3, with the title "Single-chip Cloud Computer", we are going to present the main computer that we worked on and tested our concurrent data structures, Intel's SCC. We are going to talk about hardware aspects of this system, as well as programming interface. In addition we are also going to present with details our implementations for the data structures.

In chapter number 4 we are going to explain the method that we followed when conducting our measures and the properties we evaluated and measured. The title of this chapter is "Evaluation of concurrent data structures in SCC".

The 5th chapter, "Experimental Results", is where all our measurements and outcomes are presented. We provide the graphs from all the different scenarios we examined and simulated. In this chapter someone can find all the material that is the result of our experiments.

Finally, in chapter number 6, "Conclusion", we summarize the remarks we made throughout this work and these experiments. We give some conclusions that resulted from the simulations and the evaluation of the concurrent data structures on the Single-chip Cloud Computer. Also we propose some future work and other aspects that can be researched.

Chapter 2

Concurrent data structures

2.1 Introduction

With multi-core systems being present more and more in the domain of computer systems and parallelism being the solution to overcome practical computer science problems and the path that will lead beyond present efficiency thresholds, concurrent data structures is one of the key factors that need to be examined and researched to establish good performance and reliability of parallel systems.

As concurrent data structures, we define the data structures that give us the ability to store and handle data by multiple threads or processing cores. In many occasions a concurrent data structure is nothing more than a usual data structure, e.g. a stack, a queue, a heap, enriched with protocols of usage or mechanisms to ensure that important properties of the structure and its function are secured and guaranteed. Nevertheless there is a lot of research interest and work done towards more complex and advanced structures, to optimize transactions with the data structures with many threads or cores taking part. Those advanced structures are not useful for single core – one thread systems.

A concurrent data structure faces many risks and dangers if certain steps and measures are not taken. One must ensure that memory transactions are made with the same order that the requests are placed, memory state and the data saved need to be in a consistent state and the values of the variables and generally the memory locations need to be the expected ones. Problems for example could occur if access is given to more than one thread, to the same

memory address simultaneously, the compiler rearranges instructions so the executed code is not what we were expecting or the compiler or memory drivers rearrange the sequence of two or more memory accesses of different cores, so the data read or written are not the expected one. These situations could lead in problems such as unexpected behaviour, if data other than the expected is read or written, program, thread or core crush, or even security holes. These situations are more than common and as an example we have the first versions of Intel processors Haswell and Broadwell that had a buggy implementation of their transactional memory implementation, TSX, that could lead to unpredictable system behaviour. We are going to present the transactional memory technique in the following paragraphs.

Having said that, one of the key principals that secure the correct and accurate function of concurrent data structures is synchronization. Synchronization guarantees that memory accesses will be secure, they will happen in the same order that it was requested, if that is needed, and the cores or threads will interact with the data structure in a way that ensures consistency and expected results. There are many ways and ideas to implement synchronization, from naive ones to much more complex. Before implementing or choosing to use a synchronization method, one should weight the implementation difficulty, the efficiency that the method provides, the software and the hardware available and the functions that both support.

Following, we are going to analyse synchronization primitives, starting from simple concepts and moving on to more complex ones.

2.2 Synchronization primitives

2.2.1 Locks

Locks are maybe the simplest way to achieve synchronization and many times, the first that comes to mind as a concept. The principle is that a thread or a core has to acquire the lock to access the memory. Only the holder of the lock can access the memory at that moment and no one else. The lock is an abstract entity, it could be a variable declared by the programmer, a structure provided by the language implementation or provided by a library.

A lock refers to a memory location, thus a naive approach is to lock with one lock the whole data structure. Later we will see that this is called a

coarse-grain lock. This approach is simple to implement but as we can imagine limits concurrency, as at any given time, only one thread can interact with the memory. Of course there is a lot of research around locks so there are many more lock patterns and even, as we will see later, lock-less concurrent data structures. The thread or core that has acquired the lock can do any transaction with the memory, whereas, any other thread or core that wanted to access memory should wait, either postpone its job if possible, or continue asking for the lock until it is released and available.

Before someone uses locks there are some aspects that we need to have clear. An important issue with locks is efficiency. Efficiency depends on many factors, such as the implementation of the lock, the hardware available on a system, the number of locks that we use for our data structure and the synchronization tactic of the lock. Depending on the problem we want to solve or what we implement with our program a use of a lock could give acceptable efficiency or could be totally inefficient. We also need to keep in mind, that many lock implementations that are based on reading and writing to a memory location, to update or check the condition of the lock, e.g. if it is available, are bus based implementation. Which has serious impact on bus usage and contention, thus consuming more power and creating delay. Also we need to consider what tactic we follow when a thread or core finds a lock unavailable. There are techniques such as polling or sleeping and again weight decisions such as implementation simplicity and energy consumption or delay.

The naive implementation of locks lead to a blocking synchronization method. This is also a parameter we need to consider when we decide about the synchronization method we are going to use. Although research has provided non-blocking techniques. Locks are also a mechanism that enters competition among the threads or cores. So depending on many factors, such as the place of the lock and the kind of the memory, e.g. if it is NUMA memory, some threads can have easier or faster access to the locks thus creating inequalities in the lock possession, a situation that is measured by fairness of the lock.

If this competition is not well thought and during the implementation we do not design well our lock protocol, problems such as deadlocks or starvation can arise. Deadlock is a situation that more than one thread claim the lock, the lock is available but due to erroneous design, there is

no progress as no one can acquire the lock. On the other hand, starvation is happening when a lock is not fair and we encounter the situation where some threads or cores progress excessively whereas other threads or core have little or no job done.

Following we are going to mention some synchronization techniques for locks.

Coarse-grain synchronization

Coarse-grain synchronization for concurrent data structures means that there will be one lock for the whole data structure. To interact with the structure one must acquire access and during his transactions with the memory, no one else (core or thread) is allowed to have access to the memory. It is a simple concept that has the advantage of a really simple implementation for most data structures but many times, limits parallelism. Simply because only one core or thread at a time can have access to the data structure, so others that may want to access memory must stall or generally wait, thus losing potential computing time.

Fine-grain synchronization

On the other hand, a fine-grain synchronization tactic means that there will be more than one locks for our structure. Depending on the memory area that a core or thread wants to access, it tries to acquire the appropriate lock. With fine-grain synchronization we achieve better parallelism, as more than one threads or cores can interact with the data structure simultaneously, if they want to access different parts of the structure. The drawbacks of this approach are that it is generally more difficult to implement and we need caution to avoid problems such as deadlocks while threads are competing for the locks.

Optimistic Synchronization

With this technique, we make a basic assumption that the majority of the times, nothing bad or unwanted will happen. As a consequence, when we want to insert or delete an element in our data structure, we first search if it exists, without acquiring a lock. After the search if we need to proceed with

the insertion or the removal, we acquire a lock, we look again if the element is present or not, and do the function we want. This technique also has a bigger implementation difficulty and depending on the situation we use it, it does not guarantee more efficiency.

Lazy synchronization

The lazy synchronization tactic is a tactic divided into two parts, a light one and a heavy one. The light one is done with synchronization and immediately, for example removing logically a node by updating a tag value. Later follows the heavy part during which there is no need for synchronization and for example the logically removed node is removed permanently by freeing the memory it occupied. Again, when using this tactic we need to consider the implementation cost and the the efficiency depending on the exact data structure that we will use.

2.2.2 Atomic operations

Moving beyond locks, we have to discuss the primitive of atomic operations or atomic transactions. Atomicity guarantees isolation from other processes that run concurrently and the atomic primitive makes transactions appear as they were made instantaneously, with small pauses between each other and there was no overlapping. Atomic operations have an other characteristic, they either succeed in changing the machine's state or make no change in the state or the memory if they fail.

Atomic operations show a lot of interest because they can offer waiting-free and block-free implementations of concurrent data structures. That is because when using atomic primitives, either hardware or software implemented, we can omit mutual exclusion primitives such as locks. A property that may not come to mind but is present when using atomic operations, is retrying. With that, we want to say that if an atomic operation fails, which is not uncommon, we have to retry, maybe more than once until this operation succeeds.

Atomic operations are based on two implementations, hardware ones, with cache coherency protocols provided by the hardware designers or with special *test&set* registers. But also atomicity can be achieved with software implementations such as checking the state of variables or checking time

stamps of operations by cores or threads. The main principle of atomicity in many-core - multi-thread systems is that every operation is made independently without thinking about the other cores or threads, but before the results are accepted-saved, the thread or core checks a value. If that value has changed, which means somebody else has already changed the data before this core - thread, then it cannot save the changes, the operation fails and it has to retry or move on, depending on our program's functionality and design. If during the check the value does not appear changed then all is good and the operation is successful.

The main mechanisms of atomicity are *cache coherency* protocols, *test&set* registers and *compare&swap* registers.

Atomic operations, as we mentioned before, can help construct blockless and wait-free concurrent data structures but one should be careful and examine well their usage as atomic primitives do not always guarantee a more efficient program or data structure.

2.2.3 Transactional memory

Moving beyond locks and atomic operations, we need to examine transactional memory. The primitives mentioned above have certain limits and when using them we face certain problems.

To be more precise, coarse-grain locks, although they are easy to implement and use, have poor efficiency as they take little advantage of parallelism and concurrency. On the other hand, fine-grain locks have much better efficiency but are difficult to implement, need a lot of attention when using them to avoid dead locks and are hard to manage if the number of the locks is big. In addition to these, locks can reveal a convoying effect. In a preemptive processor or operating system, a thread that has acquired the lock but takes a lot of time, may be forced to leave the execution unit, while holding the lock, thus leading to great time loses as other threads that want the lock have to keep waiting although no one is working in a critical part. Also locks have the problem of uncertain behaviour and very possible crash, if a core holding the lock crashes.

Atomic primitives, the other choice for synchronization, face the problem that work on only a single word which leads to complex algorithms, hard to implement and with high overhead sometimes.

With these limitations, concurrent programs being more and more used and the solution of hiring high skilled programmers every time someone needed to overcome a concurrency developing obstacle being expensive and not practical, research has offered a solution. That is transactional memory, inspired by database transactions mechanisms. The motivation for transactional memory is giving the ease to the programmer, without interfering with locks or atomic operations and facing the danger of erroneous implementations, with a declarative style, to let him define the transactional parts. So transactional memory is a higher level implementation of synchronization, where the weight of synchronization is moved from the developer of the application to the hardware designer or the developer of the run time system or language system.

Within this transactional parts, the transactional memory system must guarantee atomicity, consistency and isolation. Atomicity means that either all the commands in the transactional part will be successful and committed or none. Consistency means that all data changes should be made by allowed ways and isolation means that no other than the transactional part can see the changes made until they are committed.

There are two approaches for transactional memory, the hardware implemented and the software one. Although there are also some hybrid approaches that combine elements of the previous two implementation methods. The hardware implementations are based on cache coherency protocols. In many-core systems it is common that every core has his own cache but data is shared among cores. To achieve transactional memory a bit can be added to every cache line. So if a line is marked as transactional it cannot be evicted or shared until the transactional operation is finished. This limits transactional operations to the size of a cache. Although this is a limitation, it is a progress when we compare with atomic operations that are generally limited to a word. The software approach is based on data versioning and spotting conflicts and solving them. Usually a data conflict will lead to a transaction abort and retry for some core or cores. When comparing software implementations with the hardware ones, it is common that software implementations come with a performance penalty.

Although transactional memory is a very promising and convenient synchronization primitive, we need to keep in mind that there must be more research in that direction and overcome certain problems. The example of the

problematic IBM Blue Gene/Q processor that TSX transactional mechanism led to unexpected behaviour should remind us that transactional memory must be well designed before used excessively.

2.2.4 Monitors and conditional variables

Finally, to conclude our reference to some of the simple synchronization primitives we will present monitors. Of course the list of synchronization primitives and mechanisms is not exhausted by our references and readers that want to have a complete view of the subject are urged to study the relevant bibliography.

For concurrent data structures, a monitor is a synchronization construct that combines data, methods and synchronization in a single modular package like classes combine data and methods. Monitors offer mutual exclusion together with the ability to block until a certain condition is met. The existence of monitors is based on the support from hardware, for example the existence of hardware that offers atomic operations or disabling interrupts during critical paths.

Monitors are a construction that was aimed to address several problems that locks had. With the monitor construction, threads and cores are guaranteed safe execution of critical parts and the problem of deadlocks or the problem with threads holding the lock when they are not progressing is surpassed. The problem that is addressed, is that a thread could acquire the lock but a certain needed condition is not met, for example a thread wants to read from an empty queue. This thread is going to keep the lock forever, if the synchronization design says that it should block, thus depriving other threads, of the lock, that may want to write to the queue. As a result this program will not progress, although there are threads that want to write to the empty queue.

This problem is solved with monitors as they come with conditional variables. This means that we could implement synchronization without spinning, so when a condition is met, the threads that are interested in getting the lock are signalled and start to compete for the lock. With this protocol no thread holds the lock without progressing. This signalling can be implemented with semaphores and the threads that have pending conditions could sleep or generally not being busy waiting.

Nevertheless there are also potential problems when using monitors, such as the lost wakeup problem. In this situation a thread may miss the signal that a condition has changed which may lead to this thread never waking up. Fortunately there are ways to overcome this problem that may arise, either by setting a waiting time-out or always signalling all threads on a condition, not just one. Other potential drawback of monitors and semaphores is the fairness again, among the threads that compete when there is only one lock available. So we need to evaluate again the implementation of the monitors on the system and the language we are using and estimate the performance we can get and if it is acceptable.

2.3 More complex mechanisms

Moving on, we are going to see some more complex synchronization mechanisms that are recent to the computer science society and present some research and practical interest.

2.3.1 Embedded Transactional Memory

Having presented the transactional memory primitive, we want to mention here the embedded transactional memory, a hardware implementation of transactional memory that was firstly designed for embedded systems but now with works like Sutirtha Sanyal et al [12] we try to introduce it to large scale multi-processor systems.

Embedded transactional memory is oriented to consume less energy and, by the resent research, even to provide a speed-up. The idea is to spare energy from speculative executions that would be aborted. As we have seen from the classic transactional memory, there are transactions that due to conflicts are going to be aborted and retried. Thus researchers try with a protocol and hardware support, to minimize the loss of energy by improving speculation and stopping paths that are going to be aborted, as soon as possible.

The idea is to introduce gated clocks to the processing cores. When the system finds out that a conflict has arisen and thus a roll back in a check point is needed by one or more cores, then those cores should be cut off by the gated clock. This will lead to eliminate dynamic power consumption.

With the gated clocks we will save power from operations that would be dumped due to conflicts but also we could get a speed-up, as by cutting off cores until they begin again from the last check point, we can minimize competition for the commit. Minimizing competition means minimizing contention on the bus and fewer memory accesses. This leads to a better contention management thus an overall speed-up

Although this mechanism is new and not used extensively in practise, it is a promising technique not only for embedded systems but also for large-scale systems that could lead to more efficient execution.

2.3.2 C-Lock

The C-Lock mechanism is a hardware based synchronization mechanism, designed for embedded multicore systems by Seung Hun Kim et al [5]. Embedded systems have specific constrains and demand different attention than large scale computing systems. This constrains are many times conflicting, for example we demand very good energy efficiency by embedded systems but sometimes throughput and responsiveness are also as important as energy consumption, thus having conflicting demands.

C-Lock combines lock based ideas with transactional memory ideas to provide a hybrid approach. It is an idea that by mixing strong points of each approach presents a mechanism that achieves less energy consumption and better efficiency from the other two mechanisms. The implementation is based on an added hardware piece called *C-Lock Manager*.

The circuit of the C-Lock manager is the intermediate between cores and memory accesses. Before a core accesses the memory, the operations pass from the C-Lock manager and the memory addresses are checked. The mechanism provides transactional memory function, meaning that it allows maximum parallelism and there are no locks for the shared memory but with the help of C-Lock manager minimizes the cost of speculative executions. When the manager finds out that a conflict will occur with the memory interactions, it alters the way the cores access memory, and the cores that want to access conflicting parts of the memory begin to synchronize on a lock based scheme.

Thus when memory conflicts are found the cores do not waste energy and time on operations that will be aborted later. Also, the C-Lock mechanism

achieves better power efficiency by adding gated clocks. The cores that want to access conflicting memory parts and need to wait, are cut off with the gated clock and so consume less energy.

This mechanism is very promising for embedded multicore systems as it brings together advantages of two synchronization techniques and achieves interesting results. Furthermore it is not hard to implement but needs a hardware addition. It also offers advantages such as easier programming as the weight of the synchronization is moved to the hardware design and the system implementation. The programmer just needs to mention which part of the program should be considered as synchronization dependant.

Chapter 3

Single-chip Cloud Computer

In this chapter we are going to describe the Single-chip Cloud Computer made by Intel, see it's characteristics and architecture and have a closer look on the implementations that were used during the experiment. If the reader is not covered with the characteristics described in the following sections or wants more details, we propose the study of [13], [4] and [7].

3.1 SCC description

In an era when multicore computer systems are the mainstream choice to achieve high performance and single core systems are facing certain limits thus being questioned if they can continue providing solutions for computer systems, processor manufacturers are always trying and experimenting with new approaches towards multicore systems. One of these initiatives is the Intel Single-chip Cloud Computer a many core system consisting of 24 tiles with two cores per tile.

The SCC was released in the middle of 2010 by Intel, following a previous processor, the Teraflop Research Chip, that had 80 non-IA cores. The SCC has full IA P54C cores which means it can support compilers and operating systems which is a step from Intel to provide a system that can be easier programmed and used for broader applications by more people that it would if it required programming in specific languages or using limited software tools. In the figure 3.1 we can see the top level architecture of an SCC system.

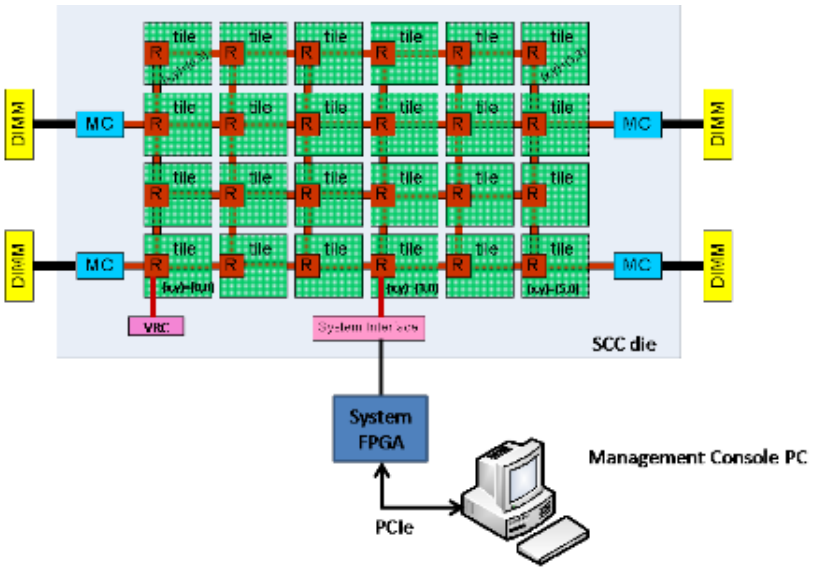

Figure 3.1: SCC Top-Level Architecture (Source: [14])

3.1.1 Tile overview

We are going to describe now the SCC tiles with more detail. As we have mentioned the SCC consists of 24 tiles with two cores on each tile. The tiles are organized in a 6×4 mesh. Each tile has a router and there are four memory controllers to provide access to off die but on board DDR3 memory. We are going to refer to the memory with details in the next section.

The SCC is connected and communicates with the management console PC, which is generally a 64-bit PC running a GNU/Linux distribution or Windows operating system. As we mentioned before, the cores can also boot a GNU/Linux distribution but it is up to the user or the research that someone is conduction if a general purpose operating system is going to be used or an experimental operating system designed for multicore systems.

Providing some more information on the die and the cores, the tile area is around $18mm^2$ and the SCC die area is around $567mm^2$. The core is made with the technology of $45nm$ high K metal gate CMOS and on the chip there are 1.3 billion transistors, 48 million on every tile. In the figure 3.2 we can see the SCC die with more detail and all of its components.

Analysing now the router that allows the cores on each tile to communicate with other cores outside the tile and the off chip memory, the router

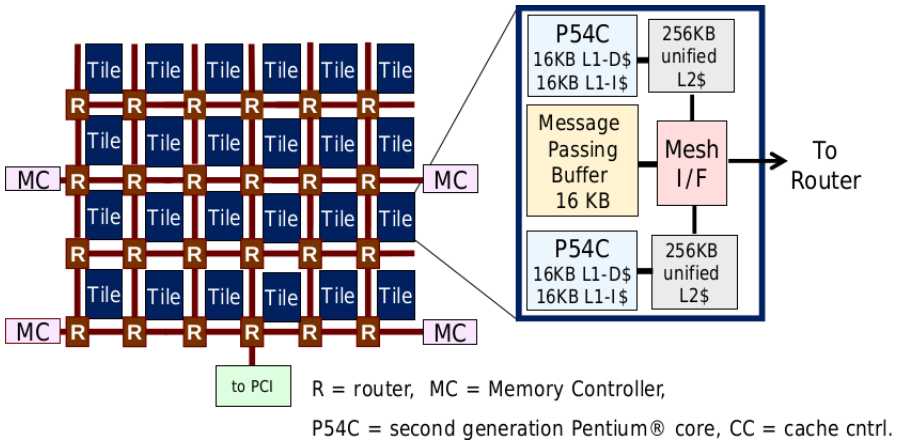

Figure 3.2: SCC die and tile overview (Source: [7])

supports a 2D network with fixed and pre-computed X-Y routes. The bisection bandwidth is $2TB/s$ at $2GHz$ and $1.1V$. It has a latency of 4 cycles, link width $16Bytes$ and bandwidth $64GB/s$ per link. Furthermore it has 2 message classes, 8 virtual channels and consumes $500mW$ at a temperature of 50 °C

The SCC board also gives us some more tools so we can experiment and research more aspects of the multicore platform. We can control and alter the voltage level and the frequency that the cores function. To be more specific it provides 7 voltage domains and the tiles can be controlled in groups of 4, so we have 6 groups that can have different voltage plus one domain for the on-die network. Also we have 24 frequency dividers so each tile can have a specific frequency chosen by the programmer. The power consumption for the full chip ranges between 25 and 125 watts.

3.1.2 Memory overview

Moving on now to see the memory overview of the SCC and learn more details about the structure and the kinds of memory that the SCC has.

To begin with, as we have already seen in figure 3.2, each tile has L1 cache, L2 cache and the message passing buffer. There is a total 32 KB L1 cache in each core, 16 KB for data cache and 16 KB for instruction cache. The L2 cache which is 256 KB in total for every core, is off the core but on the tile and it is a unified cache. In addition to the caches, on every tile there is 16 KB of message passing buffer memory, with 24 tiles total that

makes 384 KB SRAM of message passing buffer memory totally in the SCC platform.

The off chip memory is DRAM and can be from 16 GB up to 64 GB. Everyone of the 4 memory controllers can address from 4 GB up to 16 GB and depending on the part of the memory that a core wants to access, it communicates with the appropriate memory controller. This off chip memory can be private or shared among the cores. The default behaviour of the system is to give evenly as much memory as possible among the booted cores, by the user, as private and keep the rest as shared. But the user can alter this ratio of private versus shared memory either when booting the cores or by using certain commands in the program that is running on the cores.

Now that we have an image of the amount and the types of memories that exist on the SCC, we can see some details on how the memories interact. Starting from the caches, there is no cache coherency protocol between the cores' caches so the system does not provide snooping, snarfing or other cache coherency protocols. The programmer is responsible for the data exchanged between cores. The good news is that if the programmer is using the RCCE interface, which we will present later, he does not need to worry about data exchanges.

The data from the off chip memory are cashed among L1 and L2 caches according to the rules of the P54C processor. On the other hand, the data from message passing buffers which are shared, are not safe to be read from cache so SCC provides instructions and tags to mark data coming from the message passing buffers inside L1 cache as invalid. This reassures that the cores will read the data from the buffers and thus they will be sure that they read the correct shared data, as some other core might have changed them since the last time.

As it may be revealed from the previous paragraph, the message passing buffer memory is a shared memory. To be more specific it is an on-chip shared memory, distributed in every tile. Every message passing buffer, despite the tile it is on, is accessible by every core. That means every core can read or write on any message passing buffer. It is this memory that inspired us to work with the SCC platform and base our principle idea, the idea that we wanted to examine. This on-chip, very fast but also small shared memory is a foundation to research different models of communication between cores

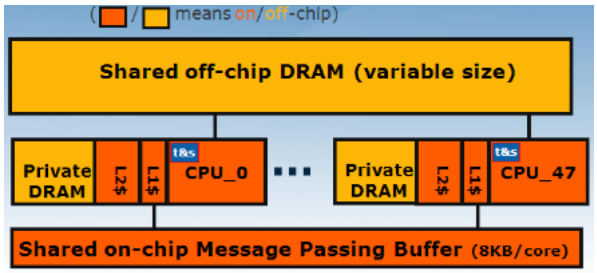

Figure 3.3: Programmer's view of the SCC memory (Source: [13])

and the exchange of data. Message passing buffer memory is the reason we implemented a server-client model of communication, something that we are going to explain later in this chapter.

Examining again the off-chip memory, we have not mentioned before that every core can see up to 4 GB of off-chip memory. This is due to the size that the look up table has. As we have mentioned before, the off-chip memory can either been seen as private parts of every core or shared memory. We also need to mention that in every core there is a test and set register that can lock the core and help us implement atomic operations. This register will also prove a key part of our implementations as it helps us implement locks.

Following we provide a picture, 3.3, of the whole memory of the SCC to help the reader visualize the organization of all the parts we have described in this section.

3.1.3 Programming interfaces

The programmer of the SCC has two main options, either to load a GNU/Linux operating system on the cores, a special distribution modified for the SCC platform, or use the baremetal option which means programming without any operating system loaded on the cores. Of course the option with the operating system simplifies many things and allows the programmer to focus on the application that he is designing but the baremetal option can also be useful if someone wants to have access to the lowest possible programming level or research the design of an operating system for the multicore platform of the SCC. The I/O system calls are routed to the FPGA.

The SCC community has offered the RCCE library, which is an application programming interface to develop programs on the SCC, based on a

message passing multicore programming style. RCCE offers many features that ease the work of the programmer and allow him to manage things on a higher level. Of course this is not always needed, when the developer wants to have low level interaction with the system, and for this occasion the developer can program without using the RCCE library.

An emulator is also available that is based on OpenMP. This emulator can work on any computer with Windows or a GNU/Linux operating system. We can develop and run SCC programs on the emulator and then just transfer them to a real SCC platform and take measurements there. This can prove useful if we have limited access to a real SCC system. The fact that needs attention is that the emulator does not provide accurate measurements both for time and power consumption.

The compilers used in the SCC platform are for the C programming language icc-8.1.038 and for the Fortran language ifort-8.1.034. Both are designed by Intel. Someone can also use the Intel's Math Kernel Library which can improve the performance of RCCE math applications. The MKL available for the SCC is mkl-8.1.1.004.

For our implementations we always used the RCCE API because we wanted to focus on a higher level of programming. But within the RCCE API there are three interfaces, the basic interface which is higher level and suitable for typical applications, the gory interface which is lower level and mostly addressed toward expert programmers and a power management API to support SCC research on power-aware applications. We used for our applications the gory interface of RCCE because we wanted to have control and define some details over the communication of the cores via the message passing buffers and also implement locks with the test&set register.

Finally we provide an image, 3.4, to visualize all the layers of software that we described above.

3.2 SCC implementations

Now that we have seen the hardware structure, the different kinds of memory that exist and information over the programming interface, we can move on to describe our implementations that were used in the various scenarios to take measurements.

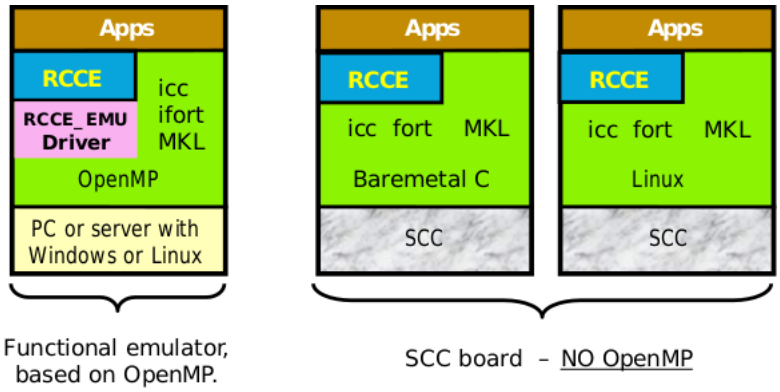

Figure 3.4: The SCC platform overview (Source: [6])

We implemented and took measurements for three data structures, a stack, a first-in first-out queue and a binary max heap. As the SCC platform is a multicore system those structures were concurrent so we needed to use synchronization methods. For the stack we had three implementations, a simple client-server model of communication that assured synchronization, a client-server model with elimination, we are going to explain that later and a coarse grain implementation with one lock for the whole structure. For the FIFO queue we also had three implementations, a simple client-server model, a coarse grain lock, one lock for the whole structure and a more fine grain lock, an implementation with two locks for the structure, one lock for every end. Finally for the binary heap we had two implementations, one with the simple client-server model and a coarse grain lock, one lock for the whole structure.

Following we are going to explain and present with more details each implementation. The reader can see an example of the actual code used for our experiments in the code appendix A

3.2.1 Stack

The stack that we implemented supports two functions, insert and extract. As expected the extract and the insert is done from the same end of the data structure, so it is a last-in–first-out structure, the known function of a stack.

Client-server model

With this implementation our goal was to take advantage of the small but fast on-die memory, the message passing buffers. In this implementation we allocate N cores from which $N-1$ are clients and there is one server. The server's job is to receive requests from the clients and grant them, only the server has access to the data structure which exists in the server's private memory. The clients send request, for insertions or extractions and wait until the server processes the request and insert the element that the client sent or returns the element that was extracted. So the server does not interact with the structure for his requests. He just handles the requests of the clients and implements them.

As we have mentioned before, for our implementation we use the *g*ory interface of the RCCE API which gives as the ability to program at a lower level and have more control over the message passing buffers. With the flags that the *g*ory interface provides we can implement non blocking communication among the cores, something that would not be possible if we didn't had this low level access.

Continuing, now let's see more details about the program. With the execution command we give as an input the number of the cores that we want to allocate and the number of requests that we want every client to make. After we allocate the needed memory for our structure, for the message buffers and we allocate and initialize the flags that we are going to use, we initialize the data structure. We decided to allocate for the structure space equal to 1.5 times the number of the total requests and initialize half of the requests' total number so the cores will never encounter an empty stack, unless more than half of the total requests are extract requests, a scenario that we will not simulate. Despite that, our implementations are designed to handle the situation of an empty stack by returning an appropriate number that indicates that situation.

To clarify what may not be clear until now, the stack works with 32-bit integers as data. So for the initialization we produce random integers with the `int rand(void)` function of the C language. Although we are interested in 32-bit integers as our data, every position of the stack is an array of 8 32-bit integers. This is because the provided functions that manipulate data to and from the message passing buffers and the memory allocation functions for the message passing buffers work with products of 32 B. That is the

reason we need to work with arrays of 8 positions, although we are only interested in a 32-bit integer every time-

Once the data structure is initialized the cores are divided in two groups, according to their ID and follow a different code path. Their ID is given by the system. One path is executed by only one core, the server. The other path is where all the other cores, the clients, enter.

The server enters a loop from which never leaves until either all the requests are granted or the server crushes or the system crushes. Inside this loop there is another nested loop, inside which the server checks the flags of every client. We explain here that if we have N clients then there are $2 \cdot N$ flags for the requests. Every client core has one flag for insert requests and one for extract requests. So in every iteration of the nested loop, the server checks both flags of every client and if he finds one of them set, he calls the appropriate function to handle this request.

After the end of this nested loop the server enters another nested loop where he checks another set of N flags. Those flags are used to notify the server when a core has fulfilled his requests. If all this flags are set then the server stops checking for new requests but if at least one of these flags is unset the server exits this nested loop and continues the iterations of the main loop.

On the other hand, the clients enter a part of the code where they submit their requests. The kind of the requests and their total number is determined by the user, depending on what scenario we want the clients to execute. We are going to present the scenarios we executed on the next chapter. Continuing with the code that the clients execute, every client blocks after he submits his request and until the server implements it. The blocking is implemented with the use of flags, the ones we mentioned in the previous paragraphs. The clients submit their request and set their flag (status `RCCE_FLAG_SET`) and wait until the flag is unset (status `RCCE_FLAG_UNSET`), which happens when the server finishes with the processing of the request.

When all the requests of a client are granted, the client sets his flag that marks that he has finished, so the server can later know when he can also finish. When a client has set his flag, his work has finished and can terminate. When every client has finished with his requests, the server exits from his main loop and prints the total time and the time that the communication took. After that the program terminates.

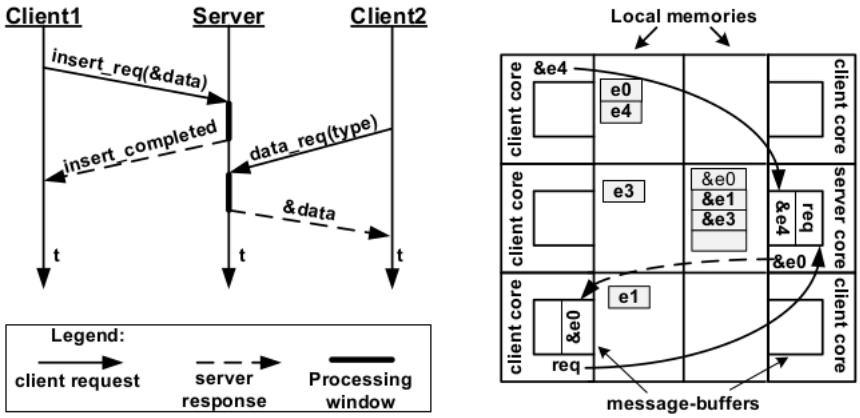

Figure 3.5: Client-server model visualization

At this point we need to explain more the use of the message passing buffers and how the flags function. Beginning with the flags, we use the provided *g*ory interface functions to work with them. These flags are implemented within message passing buffer memory. So every client writes and checks, when needed, his local buffer for the flags. The server, when checking or writing to these flags, interacts with remote buffers of other cores and tiles. With this design, as it will also be shown later by the measurements, we achieve very good efficiency because the bus is not congested with a lot of memory access requests. Actually only the server goes to the bus to read remote buffers as all the clients write and read only their buffer which is within their tile.

Finally to clarify an information that might not be stated above, the data structure as an entity is saved on private, off-chip memory, belonging to the server. That means that only the server can access the data structure.

Client-server model with elimination

This model is slightly different to the simple client-server model explained above. The main idea is that if the server receives a removing request after an inserting request, there is no need to access the off-chip memory, which means higher time cost, but can simply copy the data needed to be inserted from the one client's memory to the memory of the client that had submitted a removing request.

To be more precise on how we implemented this model, the client part is exactly the same with the above simple client-server model. Nothing

changes, concerning the clients, as only the work that the server does is modified to implement the model we want.

In the server part, an extra flag check is added, within the first nested loop. At the time that the server checks the flags of the clients, if he finds a flag set for an input request, before moving towards reading that element and storing to the data structure, the server checks only the removing flag of the next client. If he finds that flag set, he copies the element from the one message buffer to the other, without accessing the main memory to insert an element and then remove it.

This is a technique that could possibly lead to saving time as we gain time from avoiding main memory accesses. On the other hand we lose time as we do an extra check after every insert request instead of handling immediately the request. We evaluated this method for the stack on the SCC and we are going to see the results on the next chapter.

Lock model

Moving now to the model with the different approach, the one that uses a lock. In this model every core executes requests unlike the client-server model where there was one core responsible to execute the requests that he received, but did not had request on his own. The data structure in this model exists in shared, off-chip memory so all the cores have access to it. That is our goal as in this model we do not want a core acting as a link between the cores and the memory, instead all the cores should be able to implement their requests on their own.

To guarantee the correct function of the stack and the completion of the requests of every core, which means avoiding undefined conditions that can occur from concurrent accesses to the data structure, we use a lock for the whole structure. This coarse grain approach limits parallelism but is very easy to implement due to the functions that the RCCE interface offers to handle the test&set register on every core.

Every time that a core wants to add or extract an element he claims the lock. If he receives it, he completes his job and frees it. This model combined with the functions that claim the lock, provided by the RCCE interface, and the atomicity that the test&set registers provide, ensures that there will be no deadlocks. Unless of course a core that holds the lock crushes, in which

case we will have a general problem with the execution. In our program we chose that as a lock we will use the test&set register of the core with ID #0.

In this model to achieve similarity that will allow us to compare the three models, we chose to allocate the same amount of memory as in the client-server model and do the same initialization. So we allocate memory equal to 1.5 times the total number of the requests and we initialize the struct with elements equal to half of the number of the total requests. So either if all the requests are for insertion or for removal we will not encounter a problem with a full or empty struct. Although, as mentioned before, we have specific messages when such situations arise and the program can handle them.

A very important fact about the lock model is the issue with cache coherency in the SCC system. SCC does not support cache coherency so it is the programmers responsibility to ensure the correct version of the data among the different cores' caches. As in the lock model we use the shared memory for our data structure, whenever a core makes a change to an element, either adding one or removing one from the structure, we must be sure that all the other cores can see this change. For that to happen the updated data should always be written back to the shared memory as soon as possible. To achieve this we need to flush the cache of every core after he makes a change to the data structure. This can be accomplished by the RCCE function `RCCE_shflush()` which does exactly that, evicts shared data from a cache and it forces the data to be saved in the shared memory. By this way we can be sure that every change that is made in the stack can be seen by every core. If we think about it we will see that this causes a time penalty but we are going to see in the next chapters how that cache flush affects performance.

So the cores after we initialize our stack implement their requests competing every time for the lock. When all cores have finished their requests, with a barrier that the RCCE interface provides us, all the cores synchronize. Core with ID #0 is responsible to print the total time and the time that communication took. After that the program terminates.

3.2.2 FIFO queue

The basic idea of this data structure might be clear to most people with a background on computer science but we need to mention some more information to make clear what exactly we implemented. This data structure

supports two functions, insertion and removal of elements. These two operations happen in different edges of the structure, as we want the first element being inserted to be the first that that will be removed.

Client-server model

In this model we follow the basic principles and structure with the stack simple client-server model. One core is the server and implements the requests that the clients submit. The server does not make requests of his own. The communication and notification between the clients and the server is again based on the flags provided by the RCCE interface and use message passing buffer memory.

The elements to be added and the extracted elements from the structure are exchanged between the clients and the server again via message passing buffers. Also the data structure is saved in private off-chip memory of the server. We keep the same code structure with the stack client-server model, the server loops and has nested loops where he checks the flags for requests and the flags for finished cores. The clients execute another branch of code where they have a loop and submit their requests.

For this data structure too, we initialize the queue with some elements and allocate space enough so we never run out of space. We have to keep in mind that with the FIFO queue, unlike the stack, the two functions occur on different edges so when removing an element the structure does not get back a used place for a new element as the insertions are made from the other edge of the queue. The queue is implemented as an array in C language.

Lock model

Again, for the lock model of the FIFO queue we followed the same principles and design with the stack lock model. We use one lock for the whole structure and this is implemented by using the test&set register of the core with ID #0. The data structure is saved in off-chip shared memory and the cores compete for the lock to have access to the queue.

We allocate the same memory as we did with the stack and initialize the same amount of elements. Combining with the scenarios we run to take our measurements, we have distributed and mixed types of requests so that we do not run out of space in our queue. In this model the queue is also implemented as an array.

Two locks model

With the FIFO queue data structure we had the ability to check the performance on the SCC system of a more fine grain lock model, a FIFO queue with two locks. We took advantage of the fact that the operations on the structure are done in different edges, so we can have one lock for every edge.

We kept the structure and the code exactly the same with the implementation of the one lock model but we used one lock for insertion and a different lock for element removals. When a core wanted to insert an element to the queue it competed for the lock implemented with the core's test&set register with ID #0 whereas to remove an element a core competed for for the lock implemented in the core with ID #1.

Of course we had taken into account and prepared for conditions that could make the two ends of the queue meet or even the one end pass the other end, a fact that would have as an effect an erroneous function of our data structure. We have avoided this situations from happening by controlling the total number of requests and the distribution of their type, enqueue or dequeue. Also by making checks on the position of each edge.

3.2.3 Binary max heap

The last data structure we implemented is the binary max heap. This is a structure with partial ordering, which means there is some ordering of elements in the structure but it is not so deterministic where an element will always be in a specific position given same elements in the data structure, as in totally ordered arrays or trees.

The maximum element is in the top of the structure and as we move further down the elements start to decrease in value, decrease as integers. No element has a predecessor with lower value. We implemented the heap as an array and the functions that our data structure supports are extract the maximum element and insert an element.

For our heap to be functional and correct we also implemented a combine function, that does the rearranging of the elements when an element is inserted. Also we had into the code parts that made the initialization of a heap, that means when we had an array of elements it rearranged the elements to make the array have the heap properties.

Client-server model

To begin with, the main structure of the client-server model of the heap shares many things with the previous data structures. The roles of the cores, the code structure and the memory used is the same with the previous data structures. The thing that changes is the initialization process of the heap.

To be more specific, the part of the code that the clients and the server execute is the same as with the previous data structures except that now we use different functions to insert and remove data, the appropriate functions for the heap structure. The difference lays in the initialization of the heap. Because the heap is a partially ordered structure we need to apply some functions when initializing it to get the heap properties. We used a method of initialization that allows us to save some time. This method is to insert the half of the elements first, without checking something about their value. Because it is a binary max heap, half of the element are going to be leafs. Then, for every element that is being inserted we use the combine function to make sure that the inner parts of the heap are added by ensuring the heap properties.

Lock model

The lock model for the heap, is also very similar to the lock model of the other data structures but has the same difference that we mentioned above in the client-server model.

Getting into more detail, the lock is again implemented with the test&set register of the core with ID #0 and the data structure is saved in off-chip shared memory. There is one lock for the whole data structure which may seem inefficient, as when an element is inserted or extracted we need to pass from many nodes to rearrange the elements to keep the heap property. Although there are nowadays ways that can make a heap a more parallelizable data structure and there are even non-blocking concurrent heaps. But we wanted to focus on other aspects in this thesis so we opted for the simple to implement choice. The reader can refer to [1] for more information on concurrent heaps.

Focusing on the differences with the other data structures, as we mentioned above, the initialization procedure is different because a heap should

have certain properties. Like the client-server model, half of the initializing elements are simply entered in the structure and after them, for every element inserted we call the combine function to ensure the needed partial order of the elements in the heap.

Chapter 4

Evaluation of concurrent data structures in SCC

In this chapter we are going to explain the method that we followed to evaluate the data structures we implemented, the scenarios that our measurements were based on and which properties of the data structures we measured.

4.1 Analysis of method and scenarios

With our measurements we wanted to evaluate the scalability and the efficiency of our concurrent data structures. We conducted measurements with various methods to evaluate how the distribution and the nature of the requests by the cores affected various aspects of the execution.

One of the decisions for the client server model we took was that the server will be an extra core. By that we mean that comparing with the lock models, when we chose to have for example 12 cores running on the lock model, on the clients server we had 13 cores, 12 clients and the server. Although this inserts an inequality in the tests of the two models, we made sure that the total requests conducted are equal, so every core had the same amount of requests to implement, both in the lock model and in the clients server model. Of course as we have mentioned the server does not do requests, only handles the requests of the clients. Summing up, on the experiments we conducted, the cores running on the lock model match the

number of the clients and not the total cores running on the client server model.

For our measurements we had four scenarios, the three of them had equal amounts of insert and remove requests and the other one had a number, generated randomly, of insert and remove requests. To analyse more those scenarios, the first one had equal number of both requests but distributed randomly for every core. We generated once a sequence of 200 numbers, zeros and ones, each number indicated the kind of the request to be submitted. We had this sequence coded and compiled with our program. Every core started to read the sequence depending on his ID. So every core started from the place of an array that was equal to his ID number. Depending on the number he read, he did the appropriate action. After the request was completed the core read the next number. If the requests to be done by a core were more than 200 then the core just started reading again the numbers from the beginning of this 200-position array. Every core read this array enough times to complete the amount of requests that we asked to be done.

In the scenario we presented above, every core did mixed types of requests. We also designed scenarios that had a more dedicated role for the cores and we are going to present these scenarios in the following paragraphs. The first of these scenarios that we are going to present was "Half insert Half delete A". In this benchmark some cores did only insertions in the data structure and the rest of the cores only removed elements. To be more specific, the first half of the cores did only insertion requests and the other half did only requests of removal. For example in a client-server model of a data structure, running with total 13 cores, lets say that the core with ID #0 was the server, then cores from #1 to #6 only did insertion requests and cores from #7 to #12 did removal requests. If the total number of the cores was even, then it depended on which core was the server to determine what group of cores would have one more core than the other. Whereas in the lock models, if we had an odd number of cores then the removing group of cores would be bigger by one core, as the division of the cores is done by the integer division with the number 2 and as first half of the cores we consider the cores that have an ID smaller than the quotient. Thus the group of cores with an ID greater or equal than the quotient is bigger by one in case of an odd number.

We named the other scenario that gave particular tasks to cores "Half insert half delete B". The idea behind this benchmark is similar to the previous one. Half of the cores do only insert requests and the rest of the cores do removal requests. As we mentioned above there needs to be a small disclaimer, some times it is not actually two equal groups of cores but one of the groups might have a core more than the other. Furthermore, the difference between A and B is that in B, unlike the rule that we had in A, cores with an odd ID number will do insertions of elements and cores with an even ID will do only removal petitions. Consequently in every tile we will have one core always wanting to insert elements and one core always wanting to remove. This rule has two exceptions, firstly if the total cores are an odd number, so there will be a tile with only one core operating and secondly in the client-server model, the tile that hosted the server had only one client, thus depending on the servers position, his pair core might either had been only adding elements or removing.

Finally we had another random scenario that we used for some measurements, but this particular scenario, unlike the three mentioned above, does not have equal amounts of insertions and removals of elements. In the first random scenario presented, a core had a set of 200 requests that even though the type of the request was randomly distributed in this set, there were 100 insert codes and 100 removal codes. In both "half insert half delete" scenarios, approximately half of the cores did insertions and half removals therefore we also had a balanced mixture of requests. However, in the last random scenario we used, we produced again a sequence of 200 integers, zeros and ones, but this time we let the amount of both request codes to be random. We did not designate that the two sets of codes should be divided in half. The 200 integer sequence was produced with the help of the C language function `int rand(void)`. These zeros and ones were saved in an array, coded in our programmes and the cores read from that array, beginning on the position based on their ID, the type of request that they should submit. This scenario shared many similarities with the first one, this means that the cores read the various entries of the array to see what request they should do and each time moved by one position. If the total requests asked to a core were more than 200, then the core simply started from the beginning of the array after reaching the end of it, until all the requests asked to the core were submitted and implemented.

In all of the above scenarios we silently implied that we allocated contin-

uously the cores. For example if we wanted to allocate 5 cores we allocated cores with ID #0, #1, #2, #3 and #4. This is true for most of our measurements, we followed a strategy of continuous allocation of cores. Nevertheless we also conducted experiments with cores distributively allocated to compare how that effects results. We are going to present all these data in the next chapter of this thesis.

4.2 Properties measured

In this point we are going to present and discuss the properties that we measured with our experiments. The results of these experiments, the analysis and the comments are part of the next chapters.

To begin with we measured the throughput of every model, client-server, client-server with elimination, one lock and two locks, respectively in the data structures that these models existed. We measured the throughput for the scenarios we presented and analysed above. We used different number of cores to see how scalable each data structure was and how the type and the distribution of the requests affected performance. The number of the requests that we wanted the cores to implement was specified and that number was given with the execution command. In reference to the time measurements, many times we took two measurements, we called the first communication time and the second total time. Total time referred, as the name implies, to the total time that the program run and to be more precise was the time from the moment that the cores learnt their ID, through a RCCE interface command, until all the cores finished their communication and were ready to stop execution. The other measurement, communication time, was the time interval from the point that the cores started communicating, which means after memory allocations and initialization of the data structures, the point that the cores started submitting requests to the server or competing for the lock, until again the cores have finished all their requests and were ready to stop execution.

Secondly we measured the fairness of the different data structures and the different synchronization methods. Fairness is many times an important variable when somebody wants to choose a synchronization mechanism for a concurrent data structure because in an application the similarity in progress among the cores and how often they get the lock might affect performance

or the response of the application. We measured how fair the locks were and how fair the client-server model. To achieve the measurement we let the program run for some seconds and then saw how many requests each core had submitted and have been implemented.

In addition to the properties mentioned above, we measured power consumption of our data structures. Power consumption is a very important aspect and design constrain for data centres, hand-held devices, embedded systems but even general use computers. We wanted to have an image on how each data structure performed in means of power consumption and also how the synchronization techniques affected consumption. The measurement of power was made able by the system information that the SCC system provided us. We could have a measure of voltage and current so by combining these we could calculate the total power consumed in a period of time. As we have mentioned in previous chapter, the SCC platform also gives us the opportunity to control the voltage and the frequency of a tile, nevertheless during this work we did not experimented on the result that controlling and changing the voltage or the frequency could have.

Furthermore, we benchmarked different versions of our data structure programs to evaluate the role of the server's position. Of course we are talking about the client server implementations because there was no server in the lock versions. So we tried some different approaches for the server's position to see how that affected throughput. As well as this, we have mentioned in previous paragraphs of the chapter that we also tried and checked how the position of the allocated cores in general affected performance.

Finally, we had another idea to measure and see how the SCC platform will perform. The idea was to add a delay between consecutive requests and see how this affected fairness and other aspects. This concept has a reality base because in real systems it is common that cores will have to do some calculations or other work before they submit a request or interact with the data structure. By inserting this delay, that was implemented by a for loop of variable number of iterations, we tried to simulate this behaviour. By increasing the number of the loop iterations we increased the amount of time that theoretically another task, that the core dealt with, would consume.

After explaining the methods we followed and the properties we measured and evaluated, we can now move on to see the results that we acquired by our experiments. In the next chapter we are going to see the graphs with

the data and the results. Also we are going to comment and explain what the experiments show about each data structure and about the different synchronization methods.

Chapter 5

Experimental Results

In this chapter we are going to present the results we got from our measurements and provide comments to describe the performance of the SCC system and the various concurrent data structures.

5.1 Throughput measurements

To begin with, we provide the throughput measurements. With figures 5.1, 5.3 and 5.5 we see the performance in time for the three data structures comparing three scenarios each time, viewing the communication time, which is the time during which the cores implemented requests. With figures 5.2, 5.4 and 5.6 we see the performance in time but for the total time that each program run.

Next with figures from 5.7 to 5.24 we can compare time performance when we allocated continuously the cores and when the cores were allocated distributed.

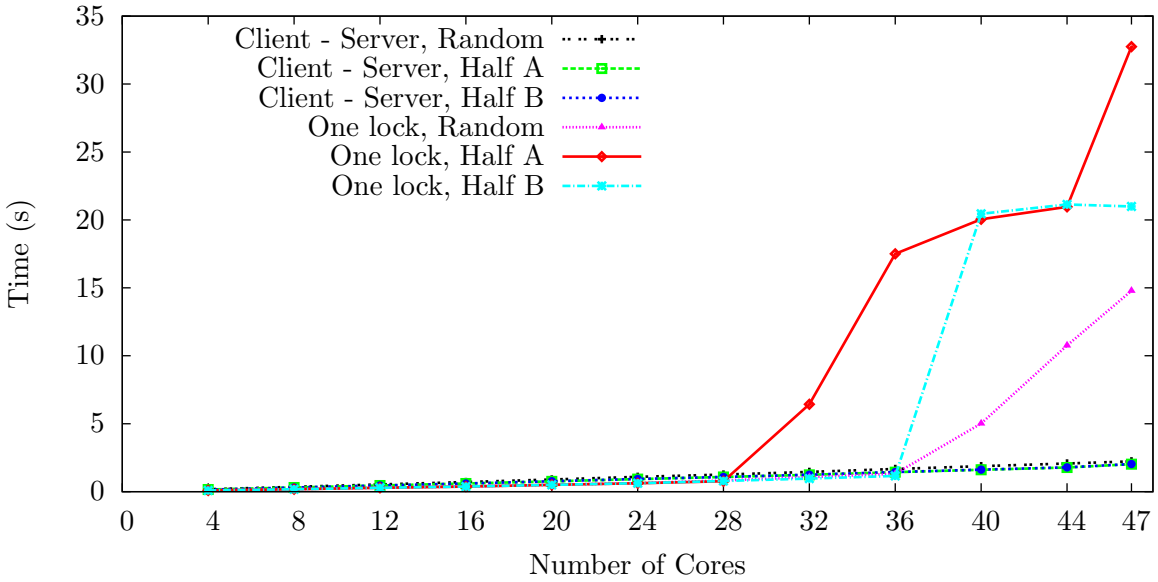

Figure 5.1: Communication time measurements for the stack, comparing three scenarios

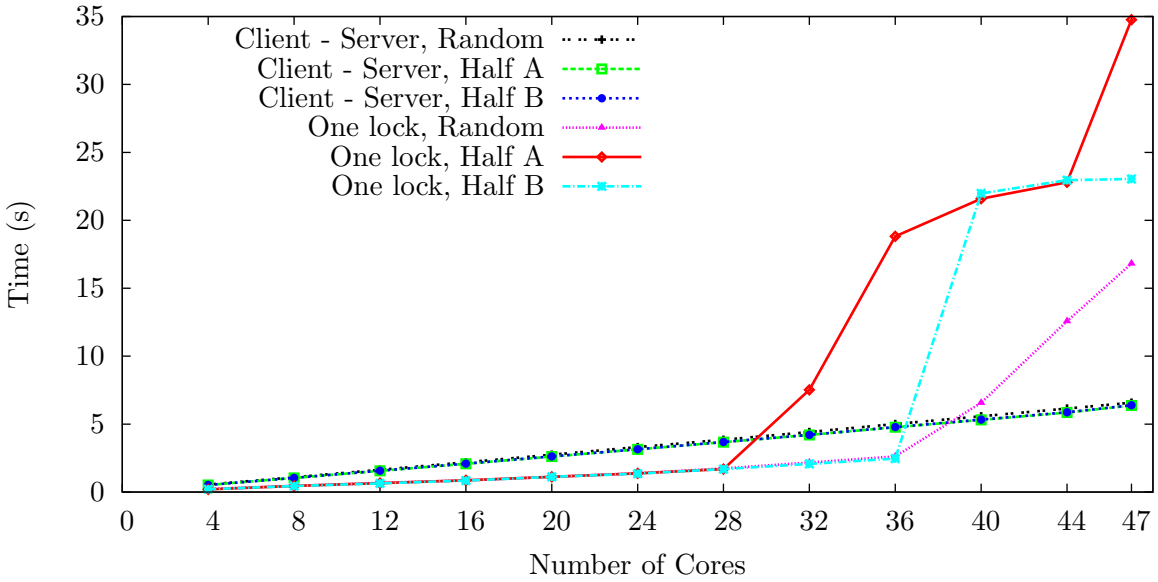

Figure 5.2: Total time measurements for the stack, comparing three scenarios

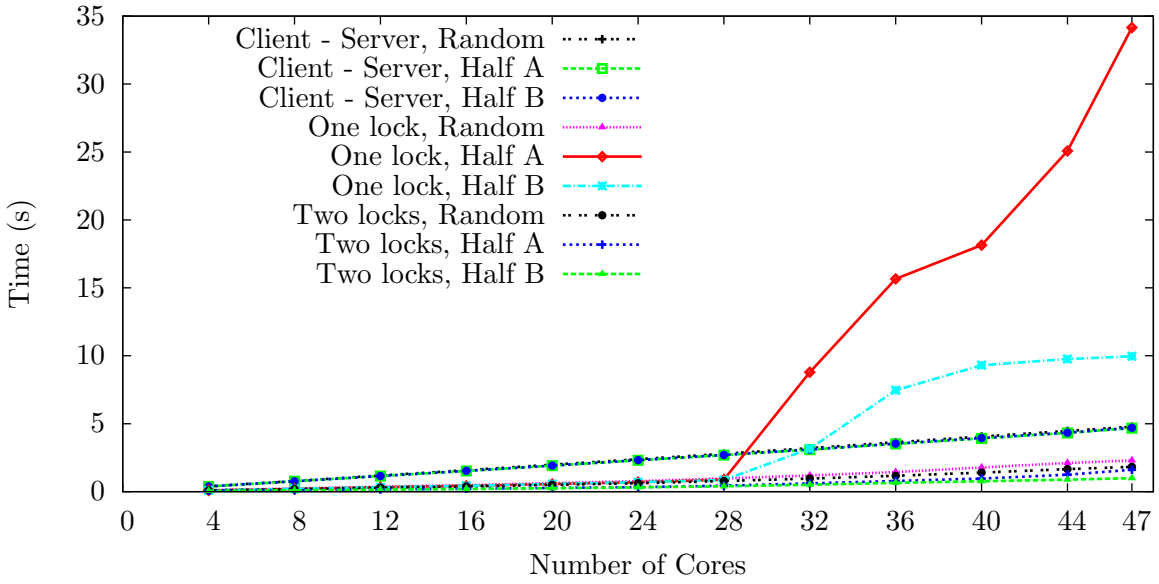

Figure 5.3: Communication time measurements for the fifo queue, comparing three scenarios

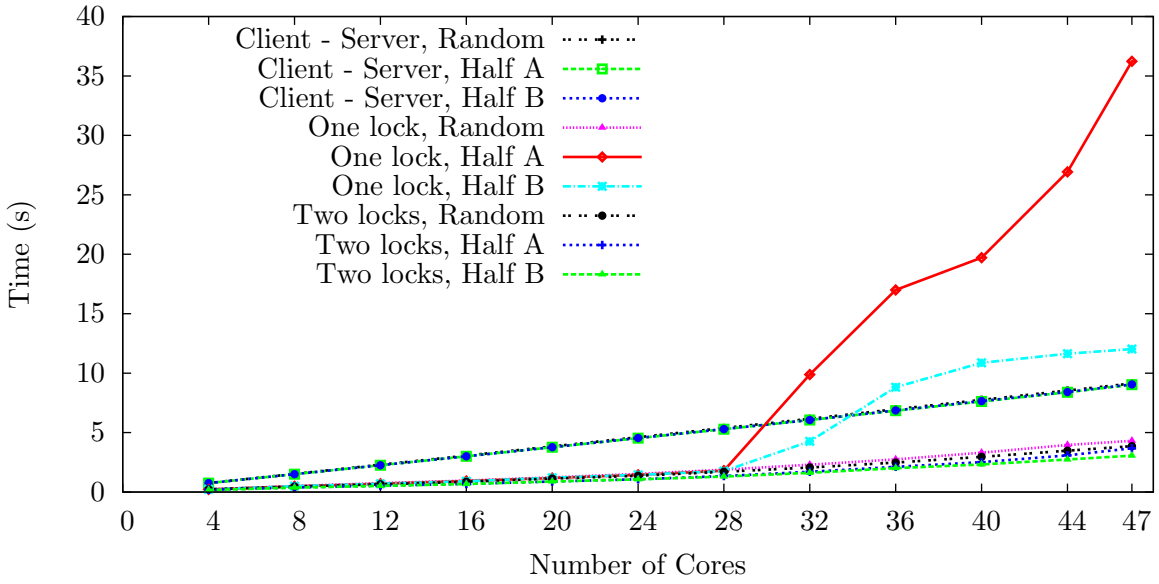

Figure 5.4: Total time measurements for the fifo queue, comparing three scenarios

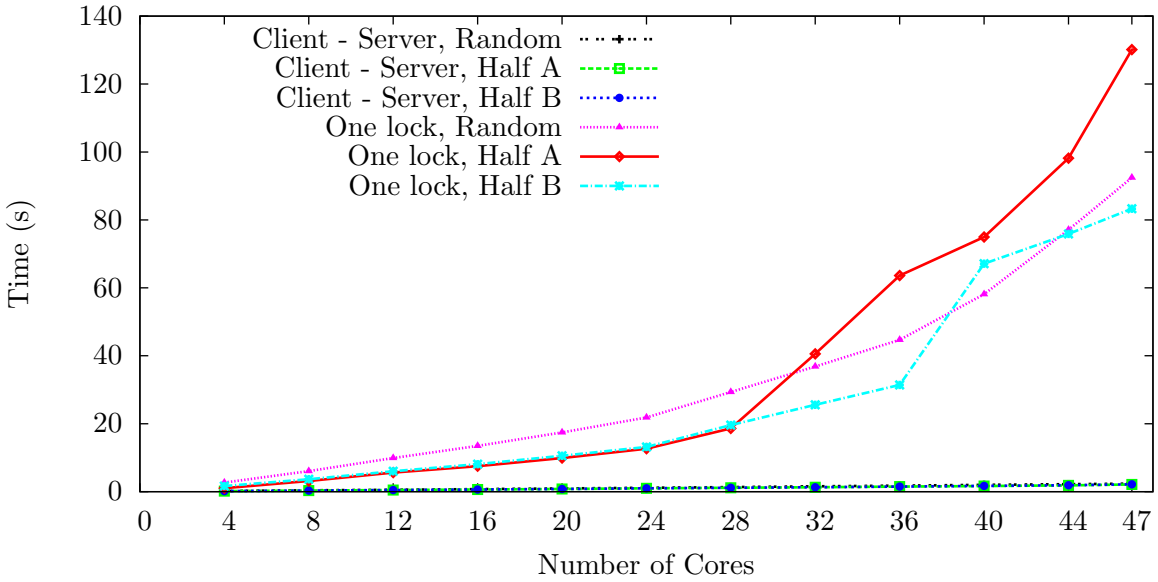

Figure 5.5: Communication time measurements for the binary max heap, comparing three scenarios

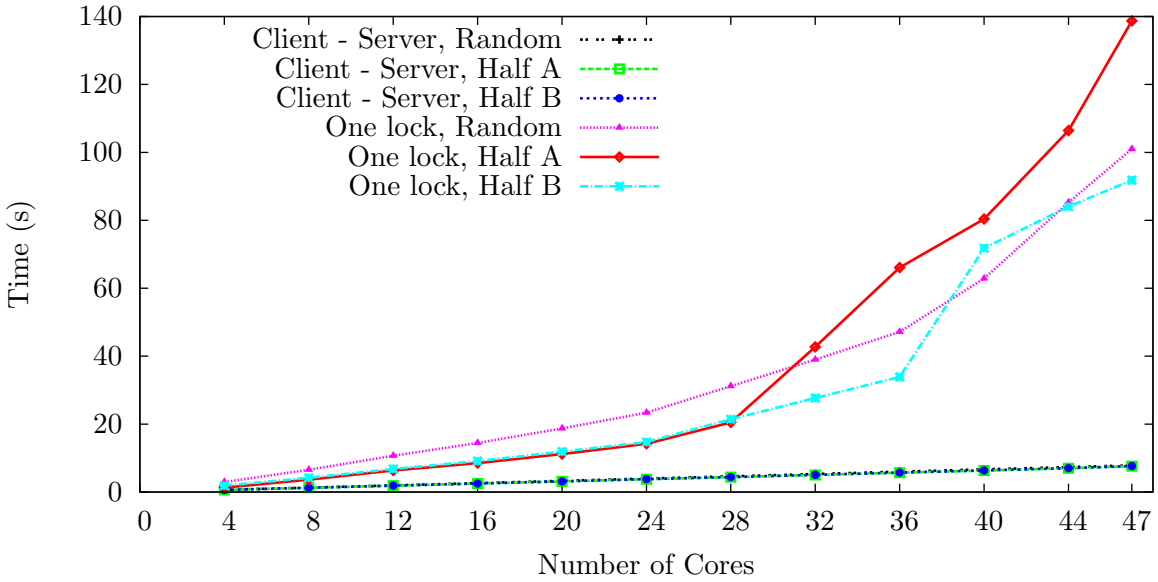

Figure 5.6: Total time measurements for the binary max heap, comparing three scenarios

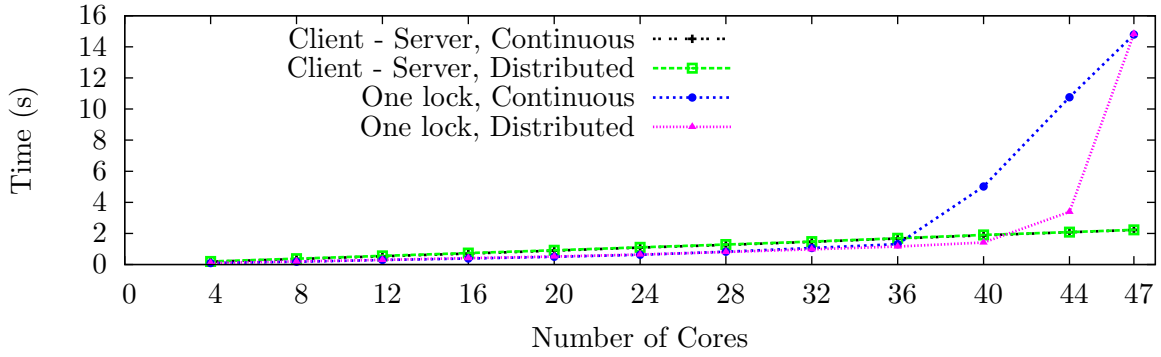

Figure 5.7: Communication time measurements for the stack, random scenario, comparing continuous and distributed cores

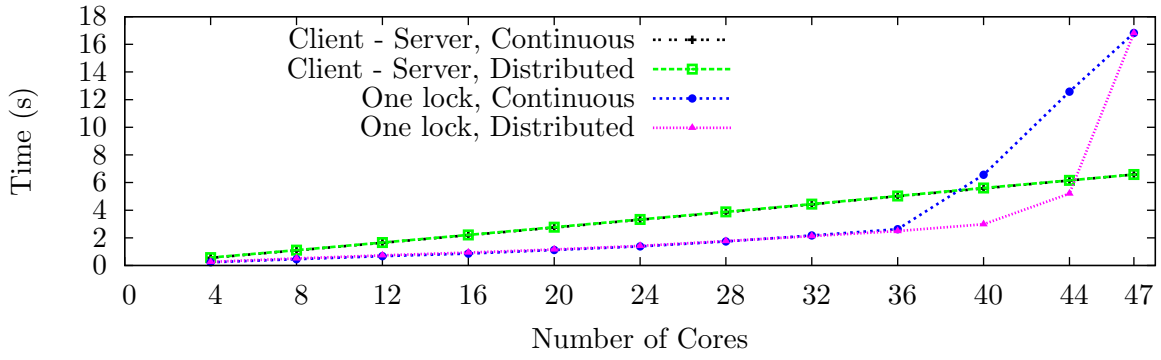

Figure 5.8: Total time measurements for the stack, random scenario, comparing continuous and distributed cores

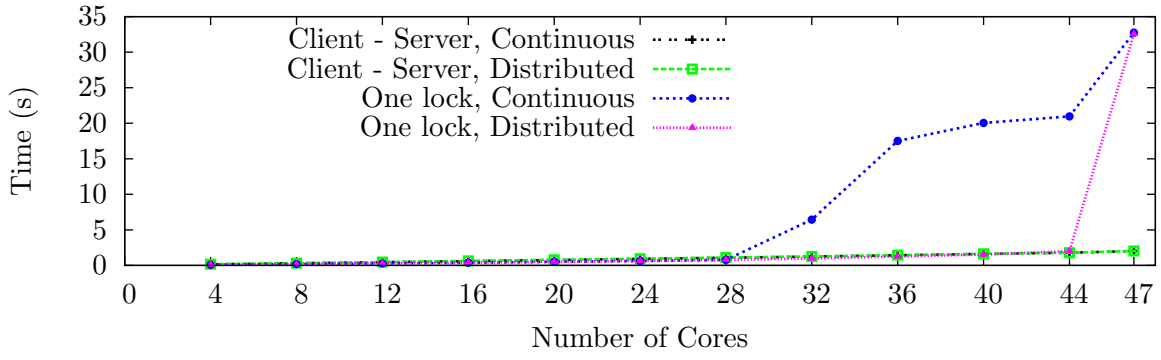

Figure 5.9: Communication time measurements for the stack, half A scenario, comparing continuous and distributed cores

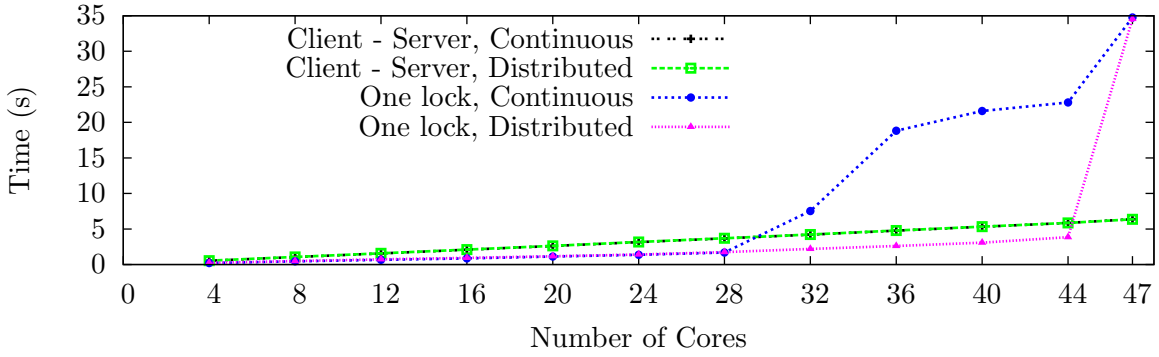

Figure 5.10: Total time measurements for the stack, half A scenario, comparing continuous and distributed cores

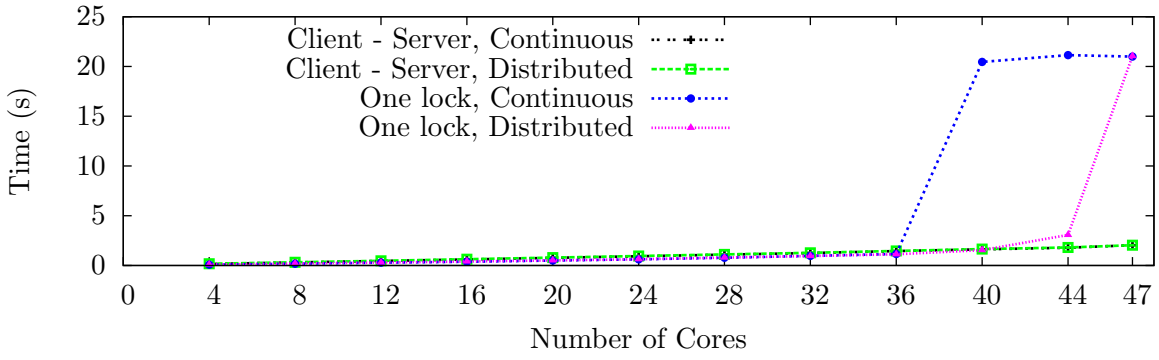

Figure 5.11: Communication time measurements for the stack, half B scenario, comparing continuous and distributed cores

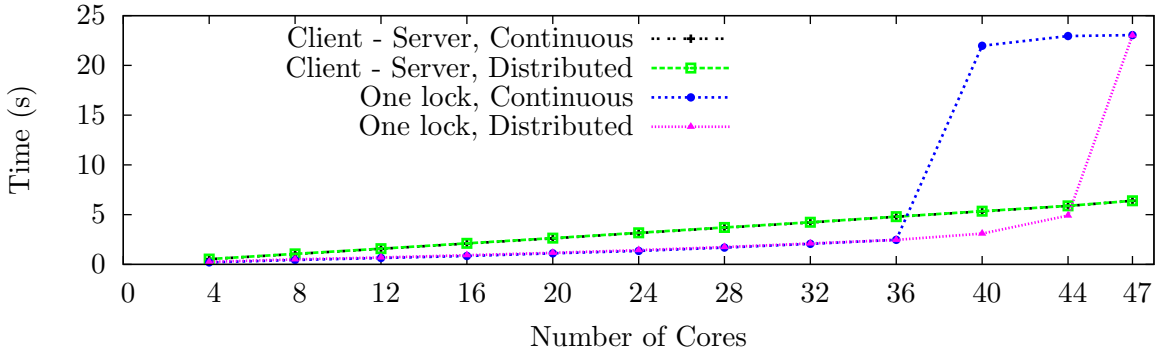

Figure 5.12: Total time measurements for the stack, half B scenario, comparing continuous and distributed cores

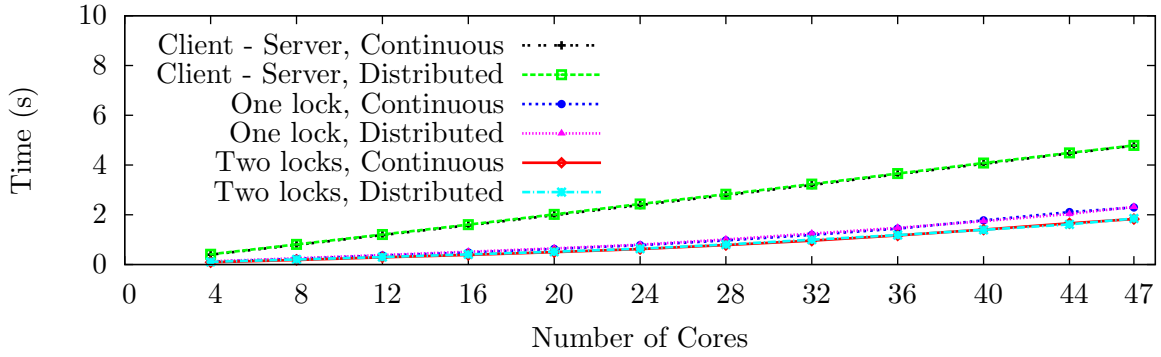

Figure 5.13: Communication time measurements for the fifo queue, random scenario, comparing continuous and distributed cores

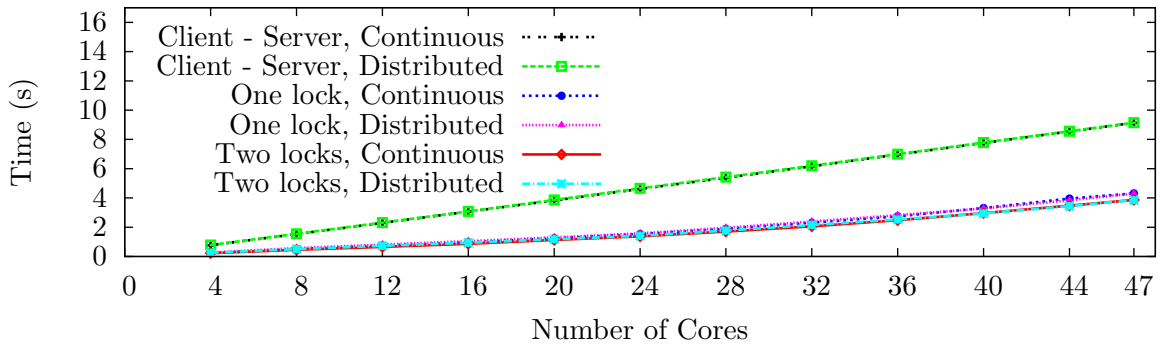

Figure 5.14: Total time measurements for the fifo queue, random scenario, comparing continuous and distributed cores

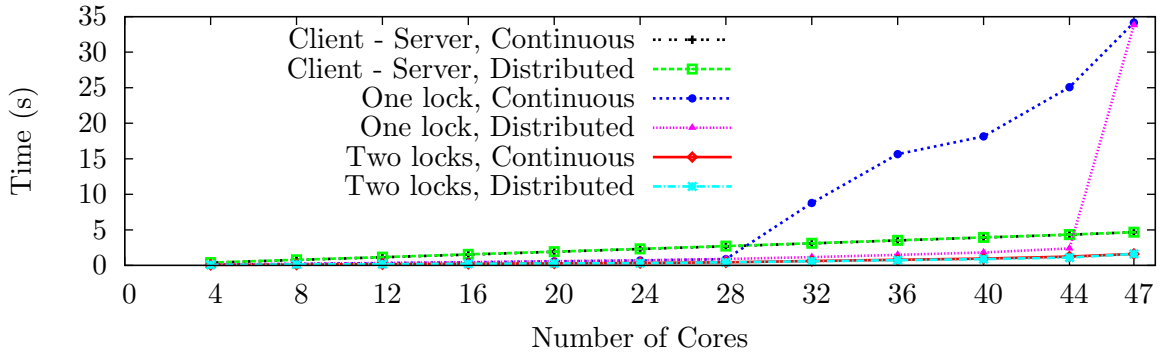

Figure 5.15: Communication time measurements for the fifo queue, half A scenario, comparing continuous and distributed cores

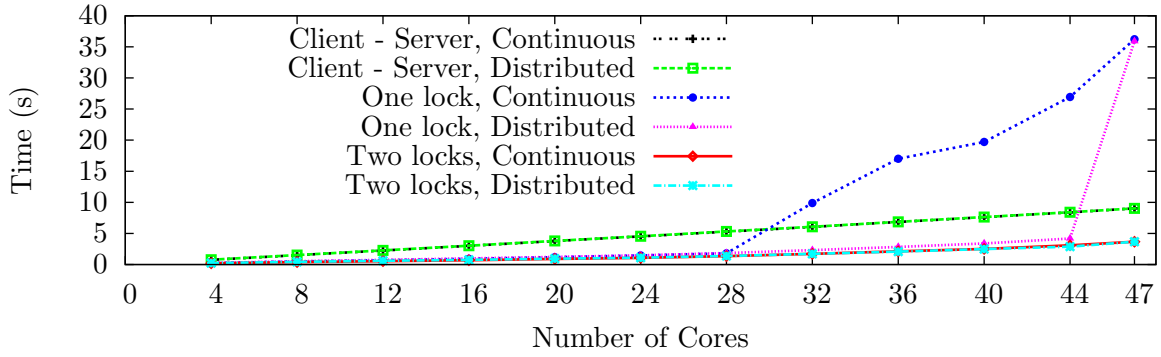

Figure 5.16: Total time measurements for the fifo queue, half A scenario, comparing continuous and distributed cores

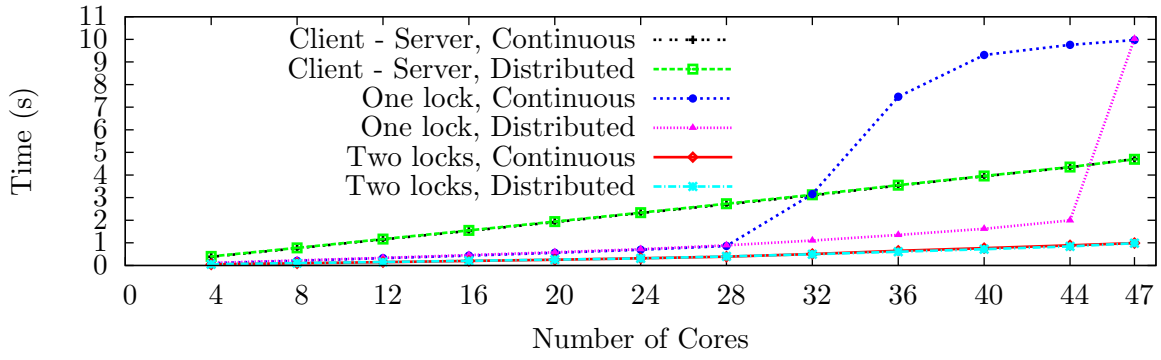

Figure 5.17: Communication time measurements for the fifo queue, half B scenario, comparing continuous and distributed cores

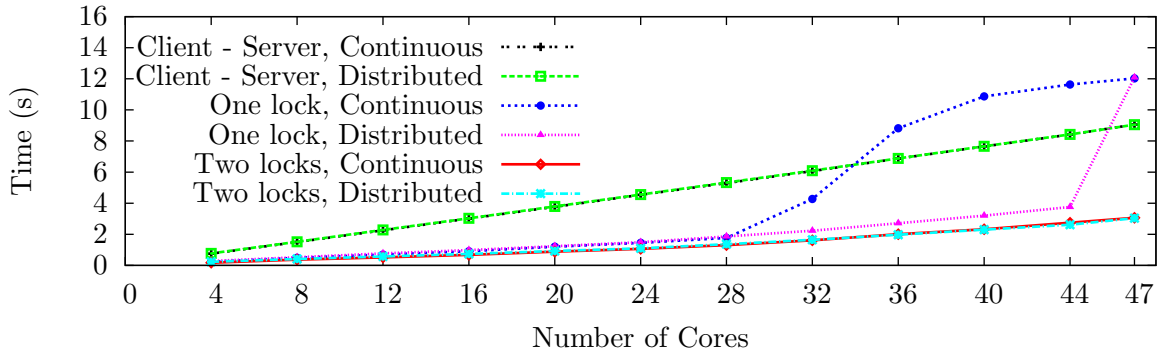

Figure 5.18: Total time measurements for the fifo queue, half B scenario, comparing continuous and distributed cores

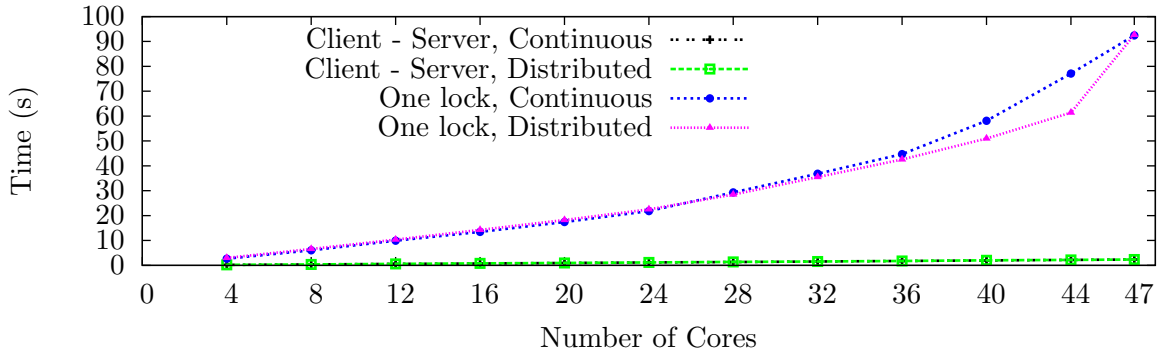

Figure 5.19: Communication time measurements for the binary max heap, random scenario, comparing continuous and distributed cores

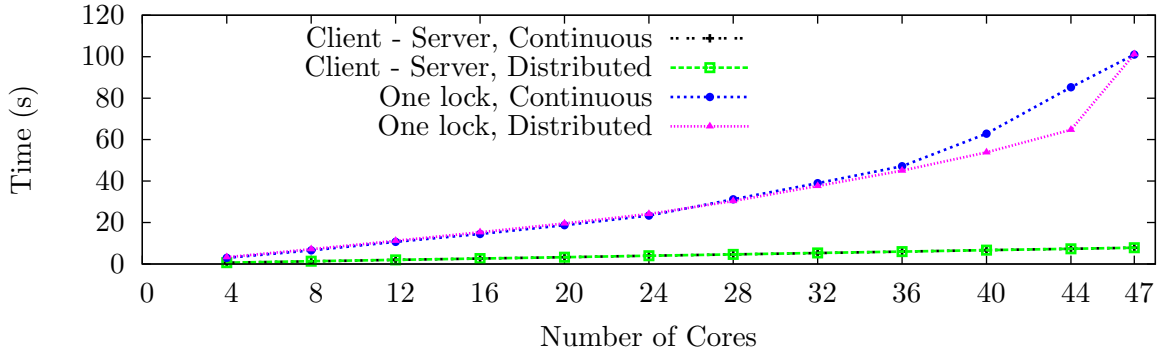

Figure 5.20: Total time measurements for the binary max heap, random scenario, comparing continuous and distributed cores

Finally in figures 5.25, 5.26 and 5.27 we provide our measurements for the random scenario which had unequal amount of insertion and removal functions.

5.2 Power measurements

In this section we are going to present the graphs that show our measurements for the power consumption. We conducted experiments for the three data structures, for all the implementations of every data structure and for the four scenarios mentioned in the previous chapter. The power consumption prices that are shown in the graph are an average price of three measurements every time.

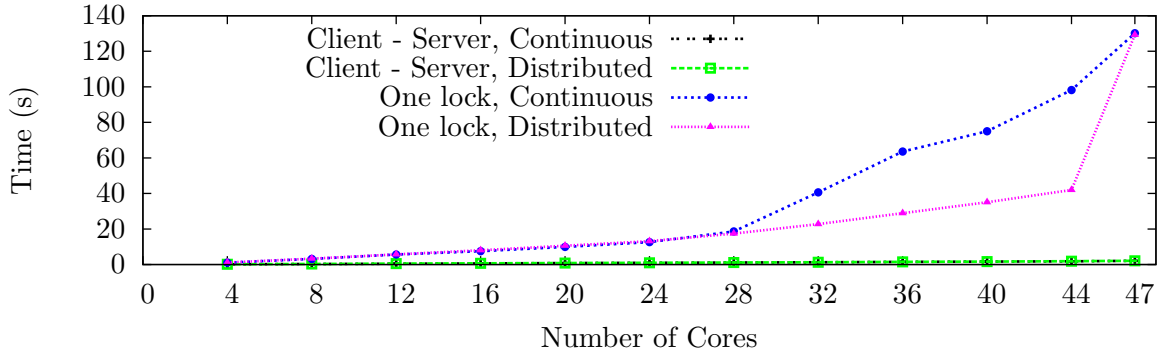

Figure 5.21: Communication time measurements for the binary max heap, half A scenario, comparing continuous and distributed cores

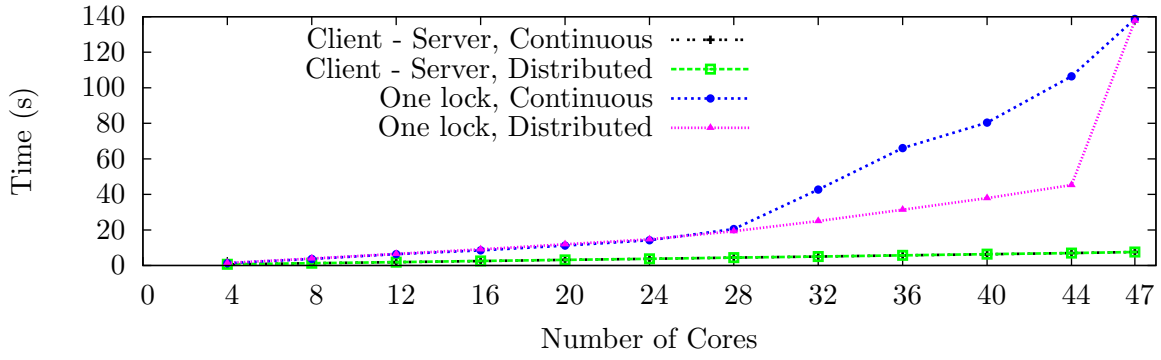

Figure 5.22: Total time measurements for the binary max heap, half A scenario, comparing continuous and distributed cores

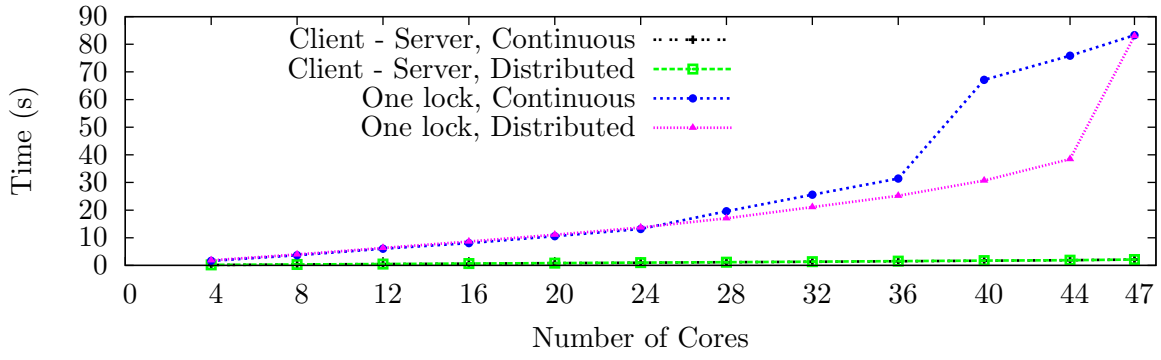

Figure 5.23: Communication time measurements for the binary max heap, half B scenario, comparing continuous and distributed cores

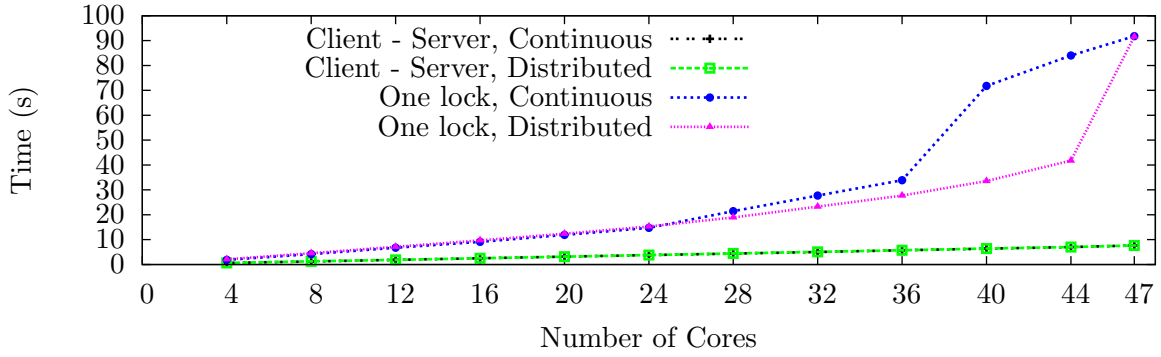

Figure 5.24: Total time measurements for the binary max heap, half B scenario, comparing continuous and distributed cores

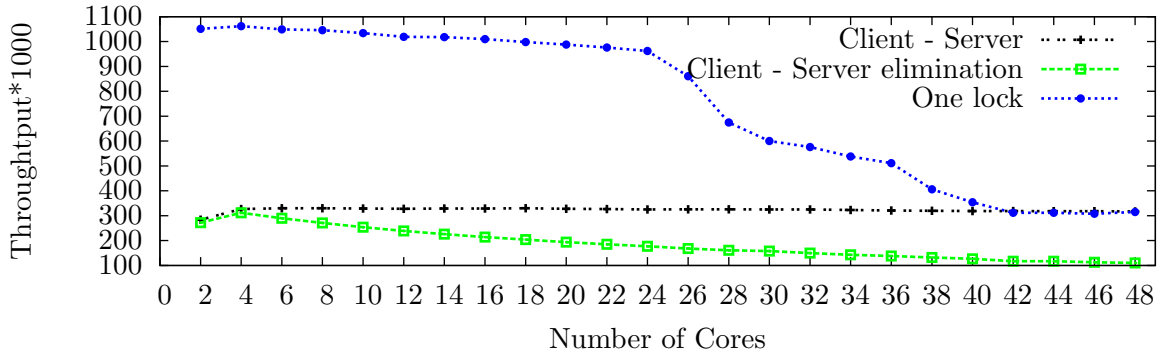

Figure 5.25: Throughput measurements for the stack, random scenario with unequal insertions and removals

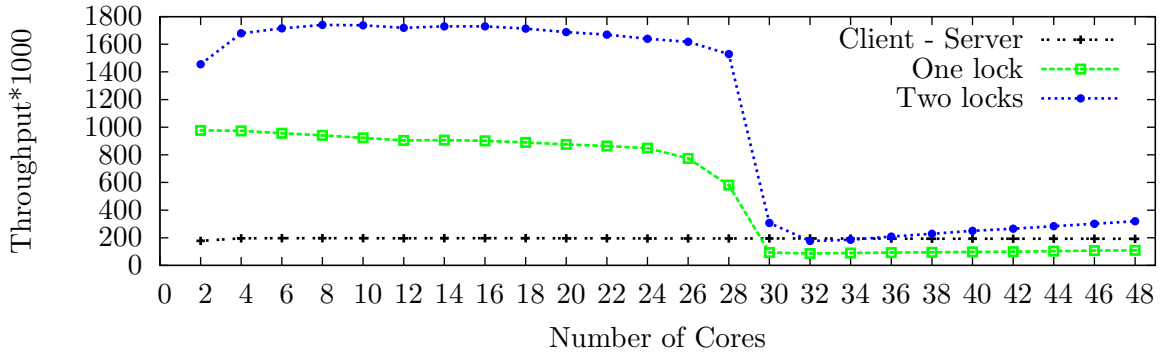

Figure 5.26: Throughput measurements for the fifo queue, random scenario with unequal insertions and removals

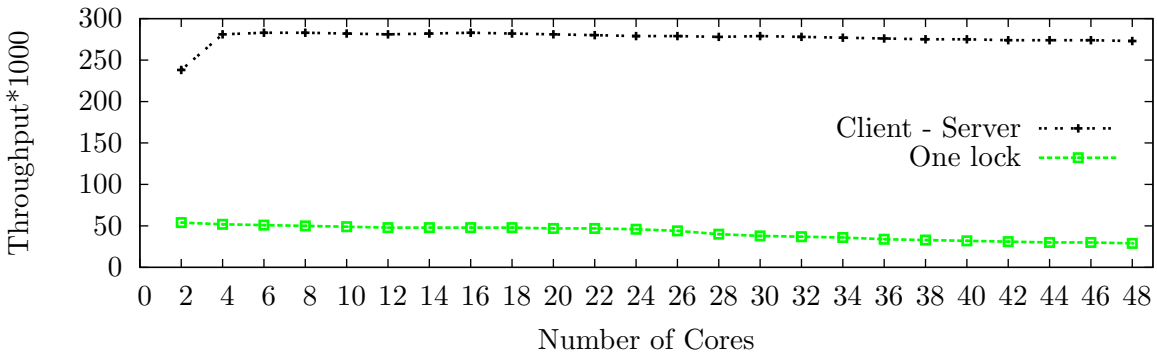

Figure 5.27: Throughput measurements for the binary max heap, random scenario with unequal insertions and removals

The power was measured for the total time that the programs ran and not just for the communication time. Cores were allocated continuously and every core had to complete 28.000 transactions before it was able to exit.

In the graphs 5.28, 5.29 and 5.30 we can see the results for the three data structures comparing the three scenarios with equal amounts of insertions and removals. In these graphs we have to keep in mind that in the client-server models the server core is considered an extra core, so for example when we get the consumption of 40 running cores, in the client-server model we have 40 clients and 1 core as the server.

Furthermore, in the graphs 5.31, 5.32 and 5.33 we present the power measurements for the random scenario that has unequal amount of the two kinds of operations. Here though, we had the same number of cores in the various implementations, the server is not an extra core on the client-server model.

5.3 Fairness measurements

In this section we present the results of the experiments concerning the fairness of each implementation of every data structure in the graphs 5.34, 5.35 and 5.36. We measured how fair the lock and the client-server implementations were because this is also an important criteria when selecting a synchronization method for a data structure.

In the graphs mentioned above, lower number in the results means more fair mechanism and when the number raises the mechanism becomes more

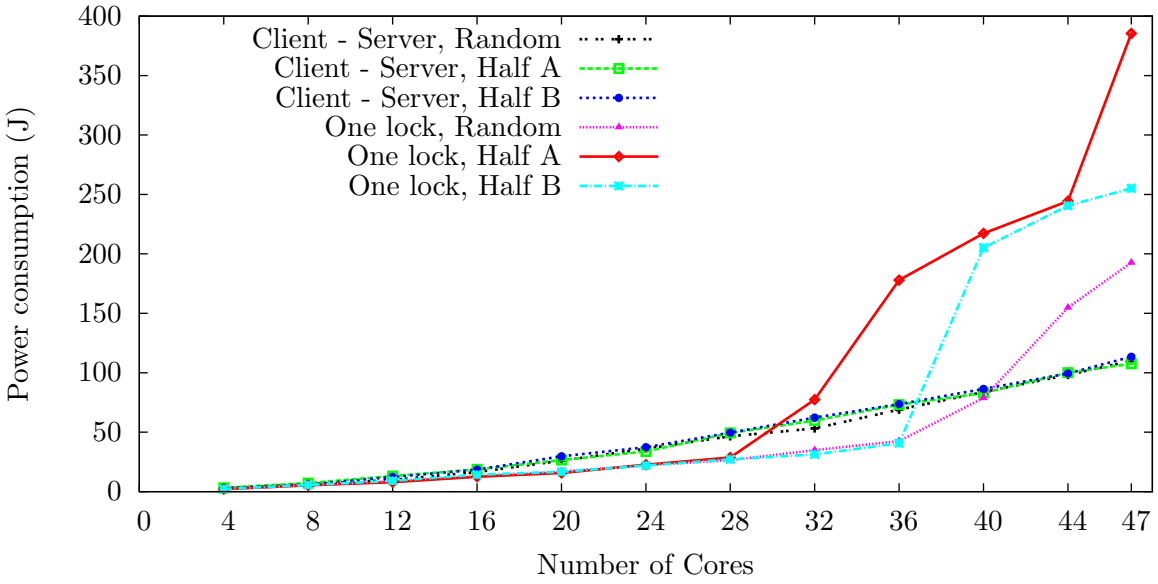

Figure 5.28: Power consumption measurements for the stack, comparing the three scenarios

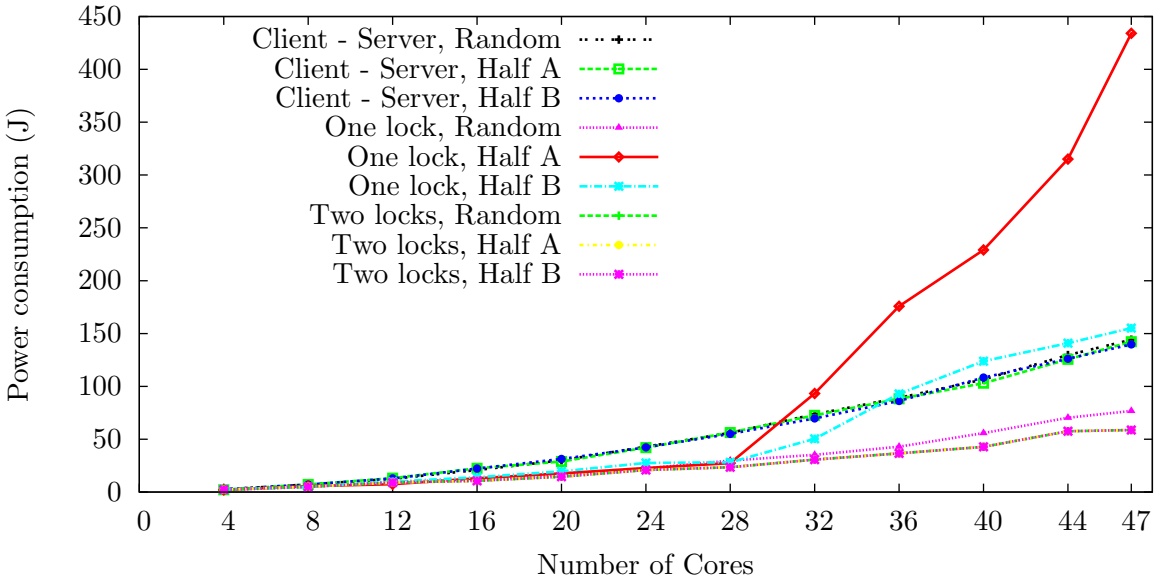

Figure 5.29: Power consumption measurements for the fifo queue, comparing the three scenarios

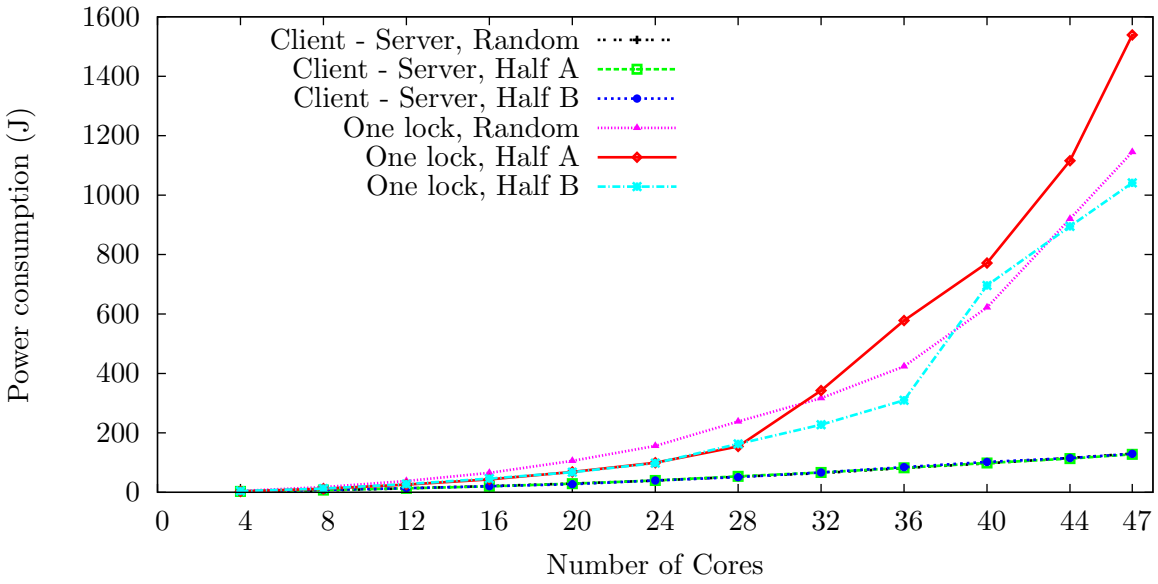

Figure 5.30: Power consumption measurements for the binary max heap, comparing the three scenarios

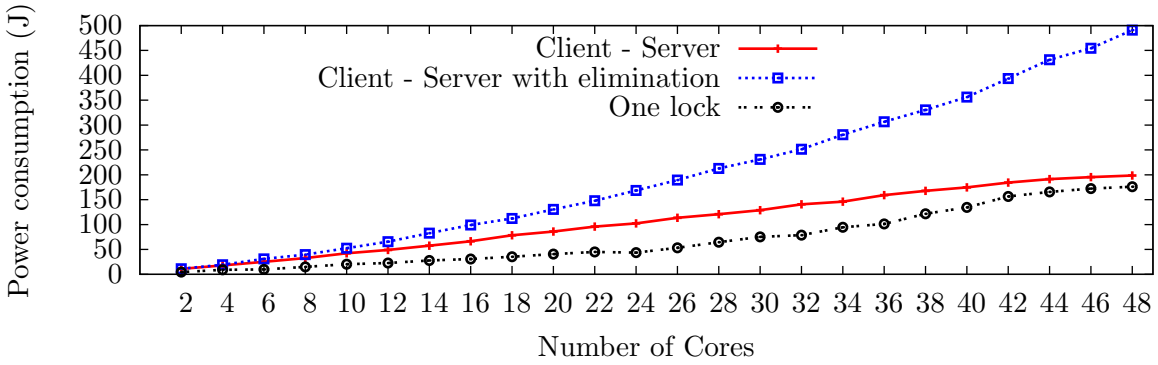

Figure 5.31: Power consumption measurements for the stack, unequal insertions and removals random scenario

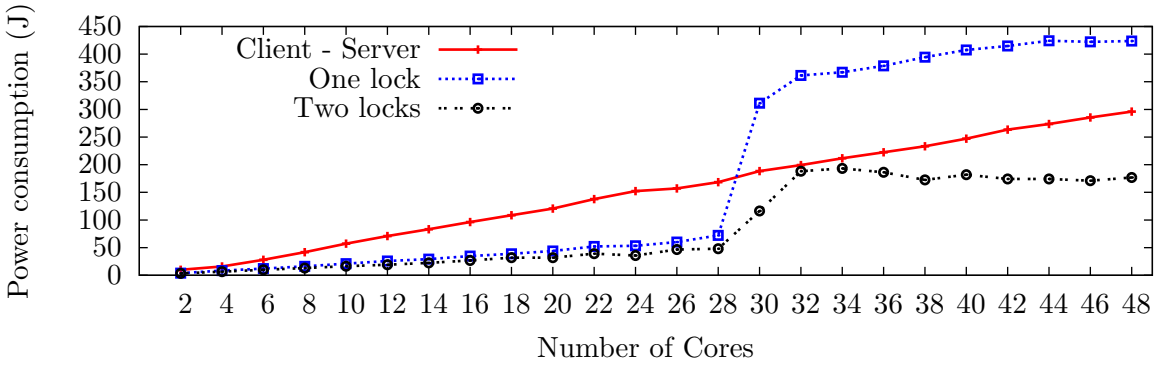

Figure 5.32: Power consumption measurements for the fifo queue, unequal insertions and removals random scenario

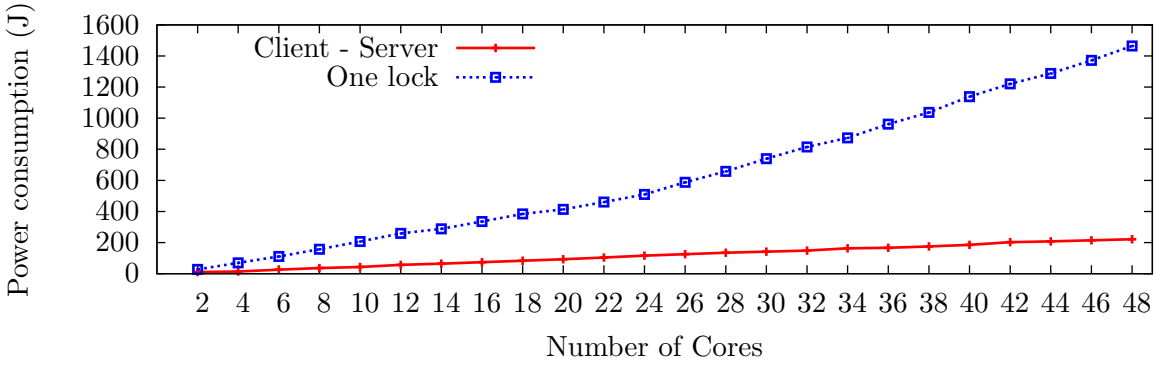

Figure 5.33: Power consumption measurements for the binary max heap, unequal insertions and removals random scenario

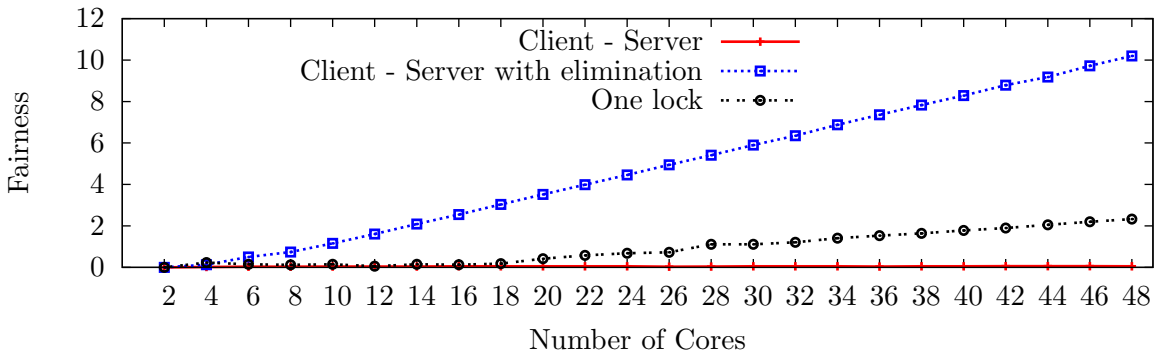

Figure 5.34: Fairness measurements for the stack, unequal insertions and removals random scenario

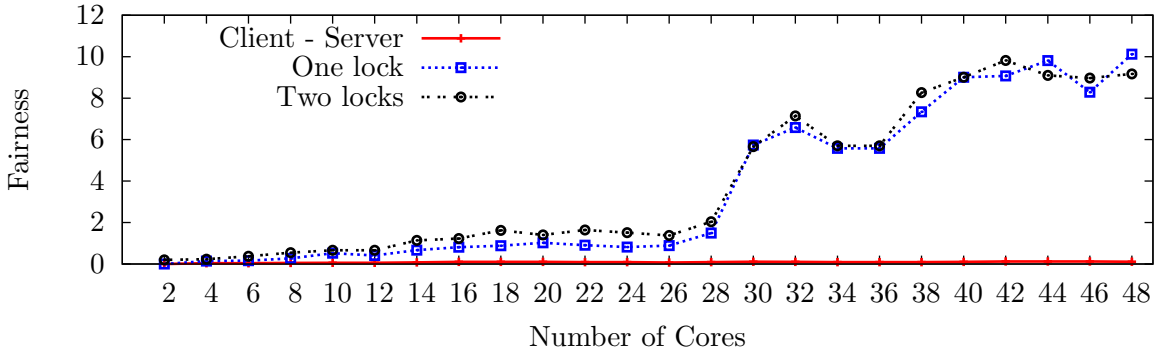

Figure 5.35: Fairness measurements for the fifo queue, unequal insertions and removals random scenario

unfair. We calculated this number of fairness by finding the average amount of requests that all the cores accomplished in a certain period of time. Then we found the number with the biggest numerical distance from the average, that was the core with the most or the fewest transactions. The result that shows the property of fairness in a data structure is the fraction of the greatest difference from the average price by the average price.

5.4 Server position evaluation

A further experiment we conducted was how the position of the server influences the performance of the client-server model of the data structures. We had two cases, one where the server was always core with ID #0 and the

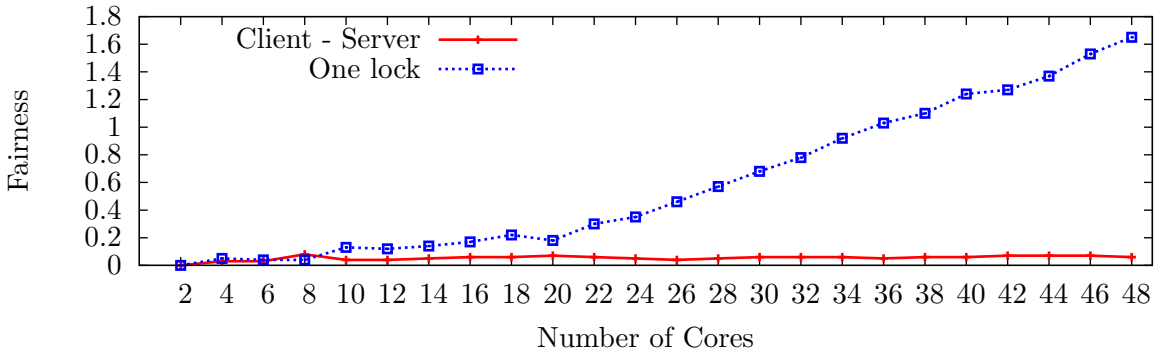

Figure 5.36: Fairness measurements for the binary max heap, unequal insertions and removals random scenario

other where the core was the median of the allocated cores. For example if we had 16 cores operating and they were allocated continuously, the server was the core with ID #7. We present the graphs that show the performance of the client-server implementations of every data structure when we tried out the two approaches. On graphs 5.37, 5.39 and 5.41 we can see the comparison for communication time for the three data structures respectively and on graphs 5.38, 5.40 and 5.42 we can see the total time comparison for the three data structures.

5.5 Evaluation with delay inserted

The last experiment we are going to present is this of the throughput in combination with delay inserted between every request of a core. With this experiment we wanted to evaluate the performance of the data structures in a scenario where every core should do some work or other task between every request. This task was simulated by a delay we implemented. The delay was a for loop and we tested various numbers of delay, so the cores began with a small delay and moved on to larger ones.

On the following graphs, 5.43, 5.44 and 5.45, the cycles of delay are increased as we move on to the right of the xx' axes. For these measures we used 48 cores and the random scenario with unequal amount insertions and removals.

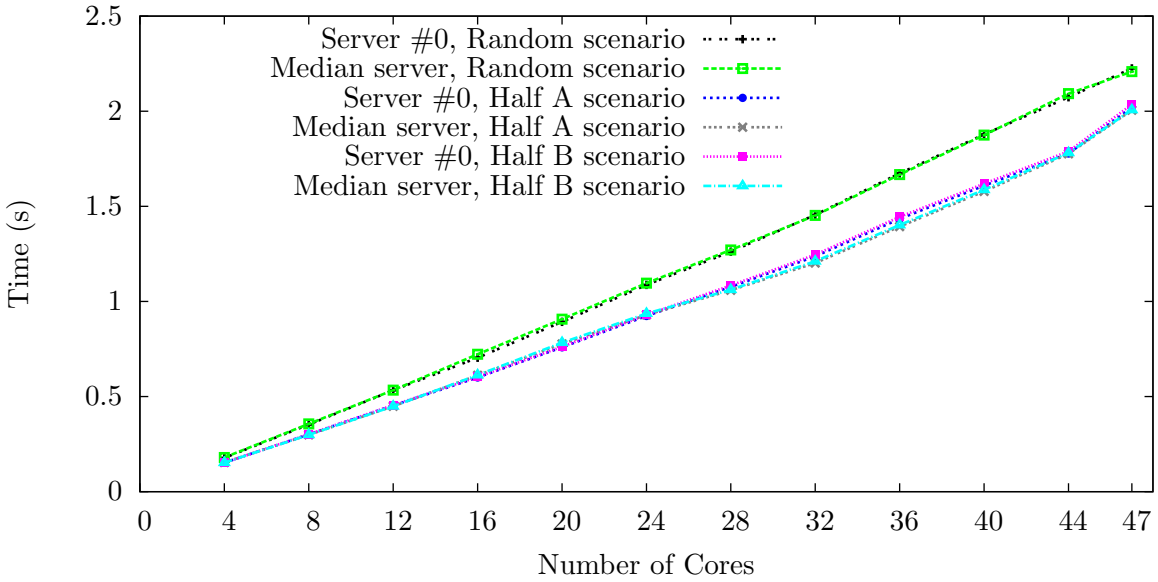

Figure 5.37: Evaluation of server's position for the stack, communication time, comparing 3 scenarios

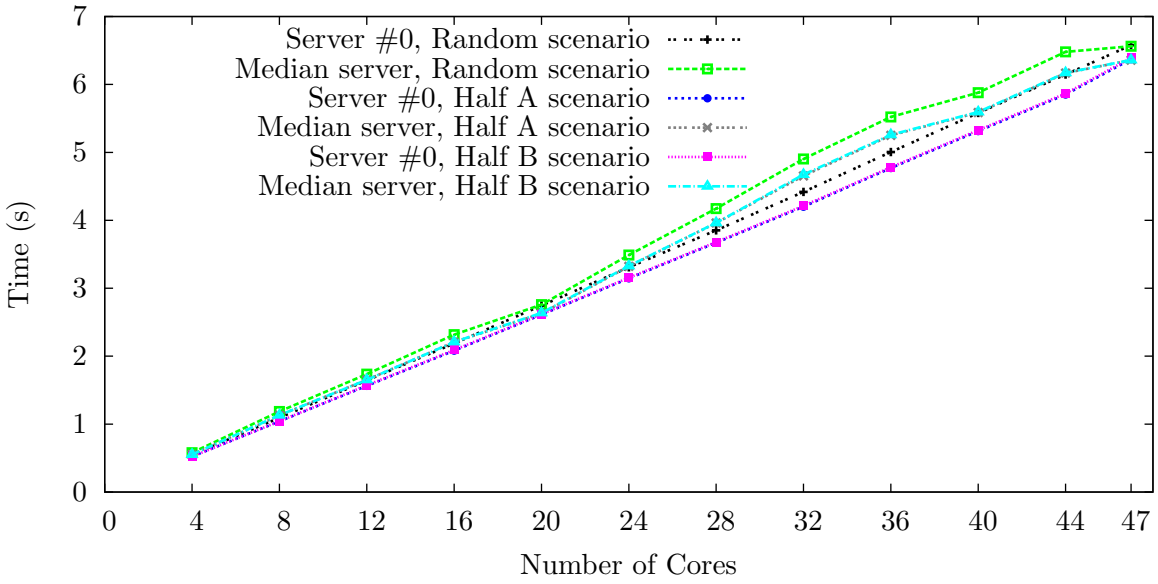

Figure 5.38: Evaluation of server's position for the stack, total time, comparing 3 scenarios

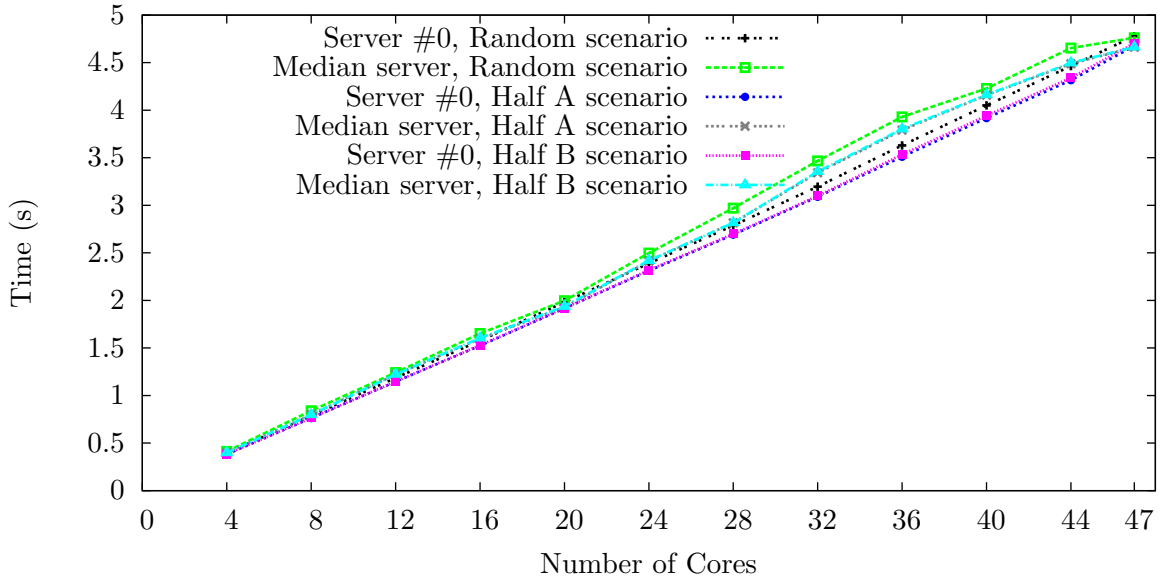

Figure 5.39: Evaluation of server's position for the fifo queue, communication time, comparing 3 scenarios

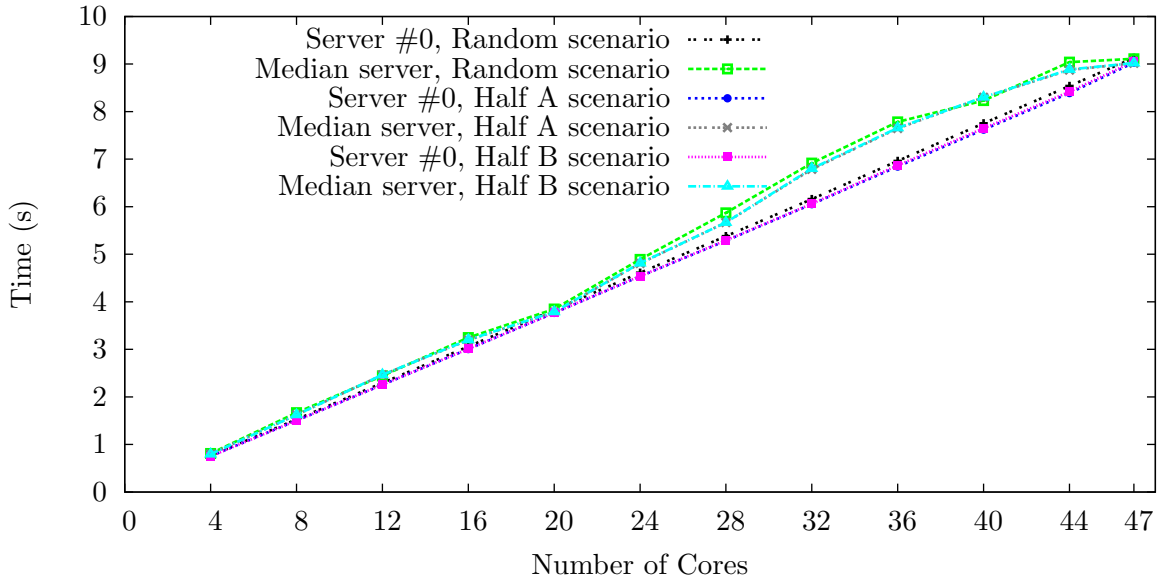

Figure 5.40: Evaluation of server's position for the fifo queue, total time, comparing 3 scenarios

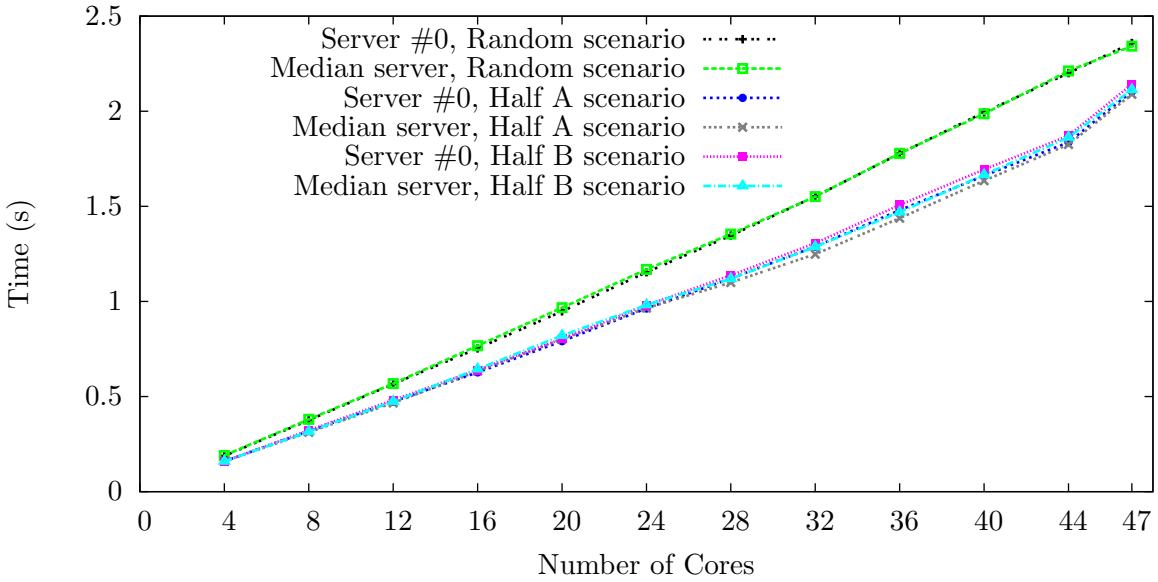

Figure 5.41: Evaluation of server's position for the binary max heap, communication time, comparing 3 scenarios

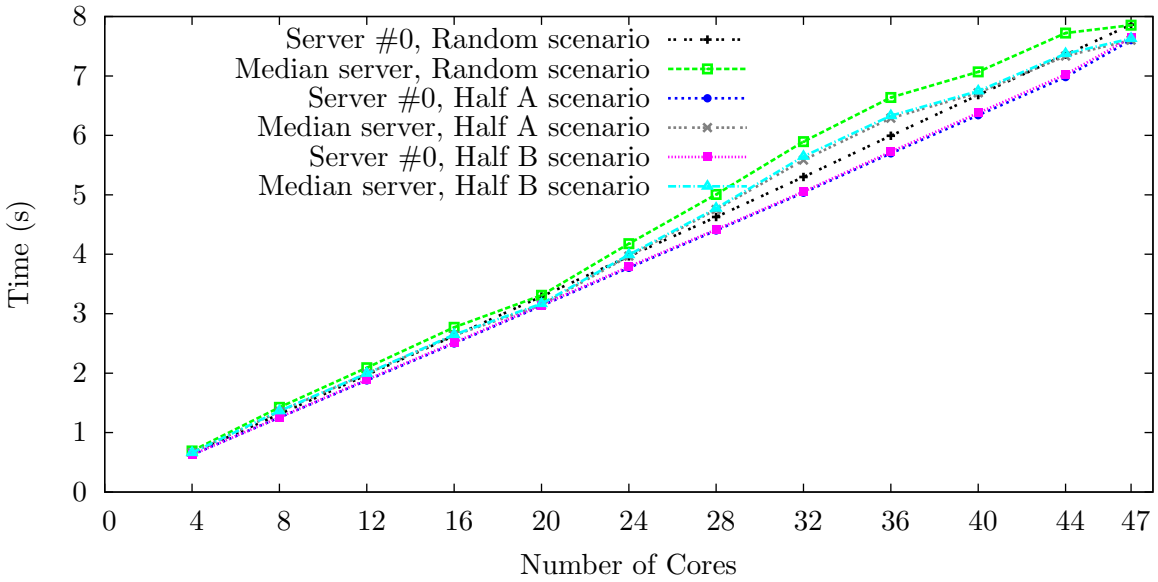

Figure 5.42: Evaluation of server's position for the binary max heap, total time, comparing 3 scenarios

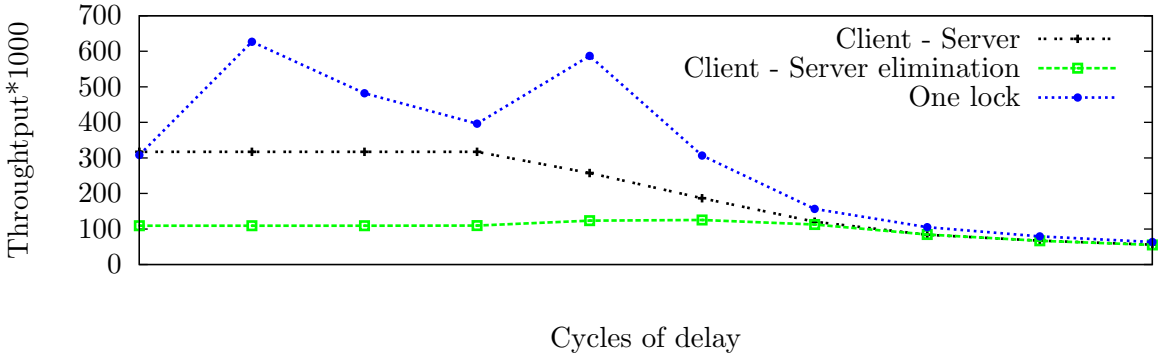

Figure 5.43: Throughput with delay between requests, stack, random scenario with unequal insertions and removals

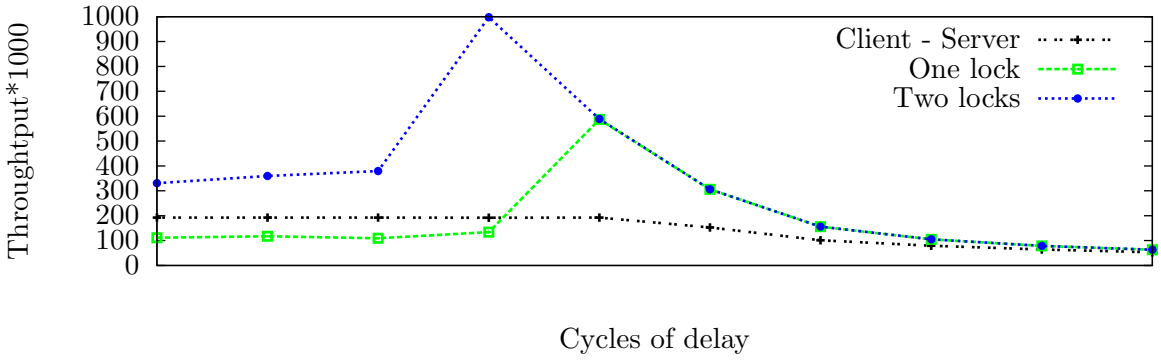

Figure 5.44: Throughput with delay between requests, fifo queue, random scenario with unequal insertions and removals

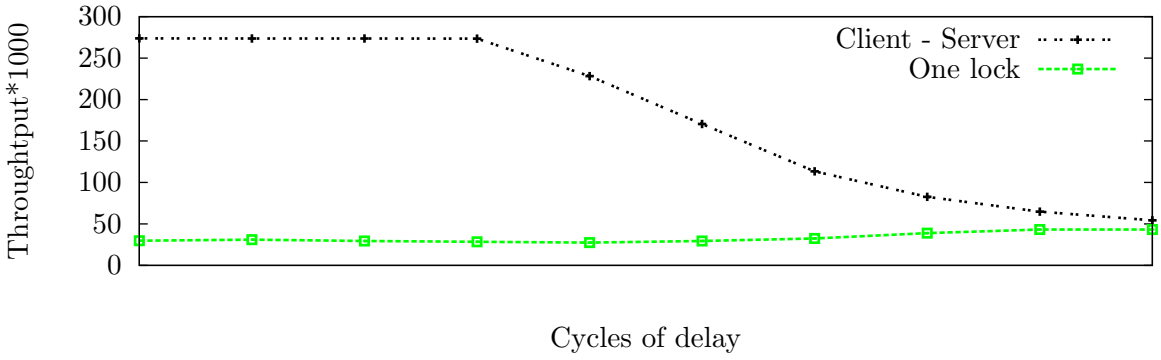

Figure 5.45: Throughput with delay between requests, binary max heap, random scenario with unequal insertions and removals

Chapter 6

Conclusion

In the last chapter of this thesis we are going to provide some remarks and summarize some results gathered form the results presented in the previous chapter. The aim is to provide feedback and suggestions to potential future users of the Intel's Single-chip Cloud Computer or people interested of getting a quick opinion over this computer and the concurrent data structures use.

We are going also to suggest some future work that could be done to research further data structure performance or the use of different kinds of memory types. Also we are going to propose other aspects of the concurrent data structures or the Single-chip Cloud Computer that could be investigated.

6.1 General Remarks

After finishing the experiments and our research, we are in a position that we can comment on some general remarks over the platform, the development and the results we collected.

To begin with the development and the working environment that SCC provides, we can say that there are no big hinders or obstacles. The fact that there is the option to use the PC that the SCC board is connected gives us the opportunity to have a GNU\Linux or Windows PC available, where it is easy to develop programs and applications and have available tools for these operating systems. So the development and testing procedure is not very hard but we have to keep in mind that cores need some times

rebooting or clearing the registers or porting again the Linux image on the cores. This happens because it is common from time to time and especially if an application does not terminate successfully that some cores may get stuck or have saved values on their registers or be in an irregular state. So, many times before we can run and try our applications we need to be sure of the state of the cores and of course that all the cores we want to use are reachable.

Furthermore we need to be careful when using the SCC platform because it does not support multiple users. In case the platform is used by a group of people we need to be cautious and maybe establish and follow a programme of usage. Although many users can be connected simultaneously to the PC server that supports the SCC board, only one application at a time can run on the cores. So if two users try to execute an application at the same time one might over through the other or there might be unexpected or incorrect results.

About the cores, we need to comment that the fact that Intel gave the opportunity to boot a Linux image on the cores gives programmers many tools. We can use C and Fortran compilers, so programming applications becomes easier. Also the Linux presence on the cores allows programmers to use many libraries and functions, thus making development easier and increasing available tools to work with the platform.

We will now comment on the results we got from our experiments. The most important and distinguishing element of the results comes from the binary max heap implementations. As we have mentioned in previous chapters, SCC lacks a coherency protocol for the caches, so it is the programmers responsibility to check the data versions between the cores. The issue arises when we use shared memory. As we have explained, the lock based implementations use shared memory to save the data structure, so every core must read and write to that memory. To be sure of the version of the data, we need to do frequent cache flushes to make sure that the data we read and write come directly from the shared memory and are in the newest version. In addition to the frequent cache flushes, the remove and insert operations on the heap have both a complexity of $\mathcal{O}(\log n)$. This means that unlike the other data structures that have a complexity of insertions and removals equal to $\mathcal{O}(1)$, with the binary heap we need to read many more elements before completing our task, which translates to more memory reads and

writes. But the fact we evacuate the cache after every request means that we can not benefit from previous memory accesses by the same core. This leads to greater advantage in the performance of the client-server model, as in that model the data structure is saved in private memory of the server so the cache can be utilizes. This great efficiency in the performance leads also to great power saves because a program running for less time means that it will consume less energy.

Another more obvious fact is that for the fifo queue, the two lock implementation performs always better than the client-server model and of course the one lock model. Also the two locks model always consumes less power. On the other hand, we see that the client-server model is always the most fair, in all three data structures.

We can also notice that our attempt with the client-server implementation with elimination does not work better, is not more efficient or more fair, than the simple client-server model. Furthermore, we see that the position that the cores are allocated play an important role on some implementations on the throughput. This can by explained either by the change in the congestion of the mesh network that the cores use to communicate with each other and with the off-chip memory, but also by the change in the congestion in the memory controllers that are used to access the off-chip memory, as different cores use different memory controllers depending on the part of the memory they want to have access to.

In addition to this, we see that the server position in the client-server model does not play a very significant role in the throughput. Although some time we get a slightly better performance with the server at the core #0, in some scenarios and mostly when looking at the communication time we get almost the same performance, no matter the position of the server.

Finally, with the experiments that have some delay inserted, we saw that for every data structure there is a period of time that if the cores where occupied with another task for that time, the overall throughput can be better, mostly for the lock-based implementations. This means that in an application where the cores have to do a calculation, or some other task, between the requests or the lock acquiring try, there might be an improvement in throughput. The only part that may not be well depicted is for the binary max heap lock implementation, we probably had to examine larger delays that could improve throughput for this program.

6.2 Future Work

Based on the already existing work on data structures for the SCC platform, and the work done for this thesis, someone can continue and investigate more aspects both of the SCC and the concurrent data structures.

For example, a future work could be to experiment with the ability the SCC platform gives us to control and alternate voltage and frequency of a core or a group of cores. If a core competes for the lock and does not succeeds in acquiring it, the core can lower its frequency or the voltage, thus the power it consumes, for some time, until it retries to acquire the lock. Or in the client serve model, until a request is implemented by the server, the client that has submitted the request can lower its frequency until notified that the request is implemented and it can move on.

Another suggestion for further investigation would be a hybrid model for all three data structures, based on the client-server model and the lock model. We could have more than one servers, with a group of clients dedicated to them and each server would have a small private data structure. Then this servers would share a common data structure which will be the important one, located in shared memory. The servers could synchronize through a lock or more locks to access this central data structure. When a "local" private data structure is full, the server has to copy the elements in the central shared data structure and free the space of his "local" structure. Then he can continue to receive requests from clients.

Appendix A

Code

In this section we are going to present some of the code that run as the data structure implementations, the headers we used and the C files with the functions implementations.

To begin with, the header file to use the functions for every data structure.

```
1  /* A header file to ease the use of functions and
2   * structures for the programs developed during my
3   * thesis. All the programs will be developed for the
4   * Intel SCC computer using the RCCE platform.
5   *
6   * Tasoulas, Zois Gerasimos
7   * 31st October 2015
8   * Microlab ECE NTUA
9   * Athens, Greece
10  */
11
12 #ifndef DSTRUCTSLIB_H
13 #define DSTRUCTSLIB_H
14
15 /********************** STRUCTURES **********************/
16
17 /************************ Stack ************************/
18 struct stackNode {
19         uint32_t num[8];
20         //struct stackNode *next;
21 } typedef stacknode;
22
23 /****************** FIFO and Sorted List ****************/
24 struct listNode {
```

```
25          uint32_t num[8];
26          //struct listNode *next;
27          //struct listNode *prev;
28  } typedef listnode;
29
30  /********************** Binary Heap **********************/
31  struct heapNode {
32          uint32_t num[8];
33  } typedef heapnode;
34
35
36  /********************** FUNCTIONS **********************/
37
38  /*************** Stack (Client - Server) ****************/
39  void client_push(int, uint32_t *,  uint32_t *, RCCE_FLAG *);
40  uint32_t client_pop(int, uint32_t *, RCCE_FLAG *);
41  void server_push(int, uint32_t *, stacknode *, int *, RCCE_FLAG *, int);
42  void server_pop(int, uint32_t *, stacknode *, int *, RCCE_FLAG *);
43
44  /******************** Stack (Locks) *********************/
45  void locked_push(int, uint32_t, int *, stacknode *, int);
46  uint32_t locked_pop(int, int *, stacknode *);
47
48  /************** FIFO List (Client - Server) **************/
49  void client_enqueue(int, uint32_t *, uint32_t *, RCCE_FLAG *);
50  uint32_t client_dequeue(int, uint32_t *, RCCE_FLAG *);
51  void server_enqueue(int, uint32_t *, listnode *, int *, RCCE_FLAG *, int);
52  void server_dequeue(int, uint32_t *, listnode *, int, int *, RCCE_FLAG *);
53
54  /****************** FIFO List (Locks) *******************/
55  void locked_enqueue(int, uint32_t, int *, listnode *, int);
56  uint32_t locked_dequeue(int, int *, int *, listnode *);
57  uint32_t locked2_dequeue(int, int *, listnode *);
58
59  /***************** Binary Heap (General) ****************/
60  void combine(int, heapnode *, int);
61
62  /************** Binary Heap (Client - Server) ************/
63  void client_insert (int, uint32_t *, uint32_t *, RCCE_FLAG *);
64  uint32_t client_extract (int, uint32_t *, RCCE_FLAG *);
65  void server_insert (int, uint32_t *, heapnode *, int *, RCCE_FLAG *, int);
66  void server_extract (int, uint32_t *, heapnode *, int *, RCCE_FLAG *);
67
68  /****************** Binary Heap (Locks) *****************/
69  void locked_insert (int, uint32_t, int *, heapnode *, int);
```

```
70  uint32_t locked_extract(int, int *, heapnode *);
71
72  #endif
```

The library for stack structure functions.

```
1   #include <stdlib.h>
2   #include <stdint.h>
3   #include "RCCE.h"
4   #include "dstructslib.h"
5
6   void client_push(int ID, uint32_t *num, uint32_t *buffer, RCCE_FLAG *flag_push)
7   {
8           RCCE_flag_write(flag_push, RCCE_FLAG_SET, ID);
9           RCCE_put((t_vcharp)buffer, (t_vcharp)num, 8*sizeof(uint32_t), ID);
10          //printf("I sent %d\n", num[0]);
11  }
12
13  uint32_t client_pop(int ID,  uint32_t *buffer, RCCE_FLAG *flag_pop)
14  {
15          uint32_t num[8];
16          RCCE_flag_write(flag_pop, RCCE_FLAG_SET, ID);
17          RCCE_wait_until(*flag_pop, RCCE_FLAG_UNSET);
18          RCCE_get((t_vcharp)num, (t_vcharp)buffer, 8*sizeof(uint32_t), ID);
19          //printf("I received %d\n", num[0]);
20          return num[0];
21  }
22
23  void server_push(int ID, uint32_t *buffer, stacknode *stack_array, int *head, RCCE_FLAG *flag_push, int SIZE)
24  {
25          if (*head == (SIZE + (SIZE / 2) - 1)) {
26                  printf("Stack_out_of_space");
27                  return;
28          }
29          *head += 1;
30          RCCE_get((t_vcharp)stack_array[*head].num, (t_vcharp)buffer, 8*sizeof(uint32_t), ID);
31  //          printf("Pushed %d\n", stack_array[*head].num[0]);
32          RCCE_flag_write(flag_push, RCCE_FLAG_UNSET, ID);
33          return;
34  }
35
36  void server_pop(int ID, uint32_t *buffer, stacknode *stack_array, int *head, RCCE_FLAG *flag_pop)
37  {
38          if (*head == -1)
39                  stack_array[++(*head)].num[0] = -1;    //Write -1 on stack_array[0]
40          RCCE_put((t_vcharp)buffer, (t_vcharp)stack_array[*head].num, 8*sizeof(uint32_t), ID);
```

```
41  //      printf("Popped %d\n", stack_array[*head].num[0]);
42          RCCE_flag_write(flag_pop, RCCE_FLAG_UNSET, ID);
43          (*head) -= 1;           //If head was -1, then we restore it, else we just move the index one positio
44          return;
45  }
46
47  void locked_push(int ID, uint32_t nm, int *head, stacknode *array, int SIZE)
48  {
49          RCCE_shflush();         //Flush needed to make sure that we read the updated value of head
50          if (*head == (SIZE + (SIZE / 2) - 1)) {
51                  printf("Full_stack !!!");
52                  return;
53          }
54          (*head) += 1;
55          array[*head].num[0] = nm;
56          RCCE_shflush();         //To be sure that head value and array is updated on all cores
57          return;
58  }
59
60  uint32_t locked_pop(int ID, int *head, stacknode *array)
61  {
62          uint32_t element;
63
64          RCCE_shflush();         //Flush needed to make sure that we read the updated value of head
65          if (*head == -1) {
66                  return -1;
67          } else {
68                  element = array[(*head)--].num[0];
69                  RCCE_shflush();         //To be sure that head value and array is updated on all cores
70                  return element;
71          }
72  }
```

The library for fifo queue structure functions.

```
1   #include <stdlib.h>
2   #include <stdint.h>
3   #include "RCCE.h"
4   #include "dstructslib.h"
5
6   void client_enqueue(int ID, uint32_t *num, uint32_t *buffer, RCCE_FLAG *flag_enqueue)
7   {
8           RCCE_flag_write(flag_enqueue, RCCE_FLAG_SET, ID);
9           RCCE_put((t_vcharp)buffer, (t_vcharp)num, 8*sizeof(uint32_t), ID);
10  //      printf("I sent %d\n", num[0]);
11  }
```

```
12
13 uint32_t client_dequeue(int ID, uint32_t *buffer, RCCE_FLAG *flag_dequeue)
14 {
15         uint32_t num[8];
16         RCCE_flag_write(flag_dequeue, RCCE_FLAG_SET, ID);
17         RCCE_wait_until(*flag_dequeue, RCCE_FLAG_UNSET);
18         RCCE_get((t_vcharp)num, (t_vcharp)buffer, 8*sizeof(uint32_t), ID);
19 //      printf("I received %d\n", num[0]);
20         return num[0];
21 }
22
23 void server_enqueue(int ID, uint32_t *buffer, listnode * list_array , int *head, RCCE_FLAG *flag_enqueue, int SIZE)
24 {
25         if (*head == (SIZE + (SIZE / 2) - 1)) {
26                 printf("List_out_of_space.\n");
27                 return;
28         }
29         *head += 1;
30         RCCE_get((t_vcharp)list_array[*head].num, (t_vcharp)buffer, 8*sizeof(uint32_t), ID);
31 //      printf("Inserted %d\n", list_array[*head].num[0]);
32         RCCE_flag_write(flag_enqueue, RCCE_FLAG_UNSET, ID);
33         return;
34 }
35
36 void server_dequeue(int ID, uint32_t *buffer, listnode * list_array , int head, int *tail, RCCE_FLAG *flag_dequeue)
37 {
38         if (*tail > head) {
39                 uint32_t num[8];
40                 num[0] = -1;
41                 printf("Empty_list!\n");
42                 RCCE_put((t_vcharp)buffer, (t_vcharp)num, 8*sizeof(uint32_t), ID);
43         } else {
44                 RCCE_put((t_vcharp)buffer, (t_vcharp)list_array[*tail].num, 8*sizeof(uint32_t), ID);
45 //              printf("Extracted %d\n", list_array[*tail].num[0]);
46                 *tail += 1;
47         }
48         RCCE_flag_write(flag_dequeue, RCCE_FLAG_UNSET, ID);
49         return;
50 }
51
52 void locked_enqueue(int ID, uint32_t num, int *head, listnode *array, int SIZE)
53 {
54         RCCE_shflush();          //To ensure we read the updated value of head
55         if (*head == (SIZE + (SIZE / 2) - 1)i) {
56                 printf("List_out_of_space.\n");
```

```
57                  return;
58          }
59          *head += 1;
60          array[*head].num[0] = num;
61          RCCE_shflush();         //To make sure new head value and array are seen by other cores
62          return;
63  }
64
65  uint32_t locked_dequeue(int ID, int *head, int *tail, listnode *array)
66  {
67          uint32_t num;
68
69          RCCE_shflush();         //To read the updated values
70          if (*tail > *head)
71                  return -1;
72          num = array[(*tail)++].num[0];
73          RCCE_shflush();         //To make sure all cores see the updated prices
74          return num;
75  }
76
77  /* To use the following function you have to be sure that
78   * #dequeue <= #enqueue and when the first enqueue happens
79   * then the number of continous dequeues is not greater than
80   * the total enqueues that have been commited until that
81   * moment
82   */
83  uint32_t locked2_dequeue(int ID, int *tail, listnode *array)
84  {
85          uint32_t num;
86
87          RCCE_shflush();         //To read the updated values
88          if (*tail == -1)
89                  return -1;
90          num = array[(*tail)++].num[0];
91          RCCE_shflush();         //To make sure all cores see the updated prices
92          return num;
93  }
```

The library for binary max heap structure functions.

```
1  #include <stdlib.h>
2  #include <stdint.h>
3  #include "RCCE.h"
4  #include "dstructslib.h"
5
6  void combine(int x, heapnode *heap_array, int tail)
```

```
7  {
8          int l, r, mp;
9          uint32_t swap;
10
11         l = 2 * x;
12         r = (2 * x) + 1;
13         mp = x;
14         if ((l <= tail) && (heap_array[l].num[0] > heap_array[mp].num[0]))
15                 mp = l;
16         if ((r <= tail) && (heap_array[r].num[0] > heap_array[mp].num[0]))
17                 mp = r;
18         if (mp != x) {
19                 swap = heap_array[x].num[0];
20                 heap_array[x].num[0] = heap_array[mp].num[0];
21                 heap_array[mp].num[0] = swap;
22                 combine(mp, heap_array, tail);
23         }
24 }
25
26 void client_insert (int ID, uint32_t *num, uint32_t *buffer, RCCE_FLAG *flag_insert)
27 {
28         RCCE_flag_write(flag_insert, RCCE_FLAG_SET, ID);
29         RCCE_put((t_vcharp)buffer, (t_vcharp)num, 8*sizeof(uint32_t), ID);
30 //          printf("I sent %d\n", num[0]);
31 }
32
33 uint32_t client_extract (int ID, uint32_t *buffer, RCCE_FLAG *flag_extract)
34 {
35         uint32_t num[8];
36         RCCE_flag_write(flag_extract, RCCE_FLAG_SET, ID);
37         RCCE_wait_until(*flag_extract, RCCE_FLAG_UNSET);
38         RCCE_get((t_vcharp)num, (t_vcharp)buffer, 8*sizeof(uint32_t), ID);
39 //        printf("I received %d\n", num[0]);
40         return num[0];
41 }
42
43 void server_insert (int ID, uint32_t *buffer, heapnode *heap_array, int *tail, RCCE_FLAG *flag_insert, int SIZE)
44 {
45         int i, p;
46         uint32_t swap;
47         if (*tail == (SIZE + (SIZE / 2) - 1)) {
48                 printf("Heap_out_of_space.\n");
49                 return;
50         }
51         *tail += 1;
```

```
52          RCCE_get((t_vcharp)heap_array[*tail].num, (t_vcharp)buffer, 8*sizeof(uint32_t), ID);
53  //          printf("Inserted %d\n", heap_array[*tail].num[0]);
54          i = *tail;
55          p = ((*tail) / 2);
56          while ((i > 1) && (heap_array[i].num[0] > heap_array[p].num[0])) {
57                  swap = heap_array[p].num[0];
58                  heap_array[p].num[0] = heap_array[i].num[0];
59                  heap_array[i].num[0] = swap;
60                  i = p;
61                  p = i / 2;
62          }
63          RCCE_flag_write(flag_insert, RCCE_FLAG_UNSET, ID);
64          return;
65  }
66
67  void server_extract(int ID, uint32_t *buffer, heapnode *heap_array, int *tail, RCCE_FLAG *flag_extract)
68  {
69          if (*tail == -1) {
70                  uint32_t num[8];
71                  num[0] = -1;
72                  printf("Empty_list!\n");
73                  RCCE_put((t_vcharp)buffer, (t_vcharp)num, 8*sizeof(uint32_t), ID);
74          } else {
75                  RCCE_put((t_vcharp)buffer, (t_vcharp)heap_array[1].num, 8*sizeof(uint32_t), ID);
76  //                printf("Extracted %d\n", heap_array[1].num[0]);
77                  heap_array[1].num[0] == heap_array[*tail].num[0];
78                  (*tail) -= 1;
79                  combine(1, heap_array, *tail);
80          }
81          RCCE_flag_write(flag_extract, RCCE_FLAG_UNSET, ID);
82          return;
83  }
84
85  void locked_insert(int ID, uint32_t num, int *tail, heapnode *array, int SIZE)
86  {
87          int i, j;
88          uint32_t swap;
89          RCCE_shflush();          //To ensure we read the updated value of tail
90          if (*tail == (SIZE + (SIZE / 2) -1)) {
91                  printf("List_out_of_space.\n");
92                  return;
93          }
94          *tail += 1;
95          array[*tail].num[0] = num;
96          j = *tail;
```

```
97          i = j / 2;
98          while ((i > 1) && (array[j].num[0] > array[i].num[0])) {
99                  swap = array[i].num[0];
100                 array[i].num[0] = array[j].num[0];
101                 array[j].num[0] = swap;
102                 j = i;
103                 i = j / 2;
104         }
105         RCCE_shflush();         //To make sure new head value and array are seen by other cores
106         return;
107 }
108
109 uint32_t locked_extract(int ID, int *tail, heapnode *array)
110 {
111         uint32_t num;
112
113         RCCE_shflush();         //To read the updated values
114         if (*tail < 0)
115                 return -1;
116         num = array[1].num[0];
117         array[1].num[0] = array[(*tail)--].num[0];      //Move last element to the top, reduce tail index
118         combine(1, array, *tail);
119         RCCE_shflush();         //To make sure all cores see the updated prices
120         return num;
121 }
```

Following we will give one example for every data structure. We are going to present a client-server implementation for the stack, a lock implementation for the binary max heap and a two lock implementation for the fifo queue.

The code for the stack.

```
1  /* A client server model executing stack operations
2   * (insert and delete).
3   *
4   * Core with ID 0 is the server and the others are the
5   * clients
6   *
7   * Tasoulas, Zois Gerasimos
8   * Microlab, ECE, NTUA
9   * 16th September, 2015
10  */
11
12 #include <stdint.h>
13 #include <stdlib.h>
```

```
14 #include <errno.h>
15 #include "RCCE.h"
16 #include "dstructslib.h"
17
18 int RCCE_APP(int argc, char *argv[])
19 {
20         int i, ID, op, totalUEs, error, checkvar = 0, iterations, SIZE, *head;
21         int data[200] = {1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0,
22         0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0,
23         0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1,
24         0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1,
25         1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1,
26         0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1,
27         0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0};
28         uint32_t num[8], myvar, *buffer;
29         double stime1, stime2, ftime;
30         time_t t;
31         stacknode *stack_array;
32         RCCE_FLAG flag_push, flag_pop, flag_finished;
33         RCCE_FLAG_STATUS status;
34
35         RCCE_init(&argc, &argv);
36         stime2 = RCCE_wtime();
37         ID = RCCE_ue();
38         if (argc != 2) {
39                 if (ID == 0)
40                         printf("SIZE needed in the input\n");
41                 return(1);
42         }
43         SIZE = atoi(argv[1]);
44         totalUEs = RCCE_num_ues();
45         iterations = (SIZE / (totalUEs - 1));
46         buffer = (uint32_t *) RCCE_malloc(8*sizeof(uint32_t));
47
48         if (error = RCCE_flag_alloc(&flag_push))
49                 printf("Mark 01: Could not allocate flag_push on %d, error=%d\n", ID, error);
50         if (error = RCCE_flag_alloc(&flag_pop))
51                 printf("Mark 02: Could not allocate flag_pop on %d, error=%d\n", ID, error);
52         if (error = RCCE_flag_alloc(&flag_finished))
53                 printf("Mark 03: Could not allocate flag_finished on %d, error=%d\n", ID, error);
54         //The following 3 if statements are not necessary, flags initialized automatically as UNSET
55         if (error = RCCE_flag_write(&flag_push, RCCE_FLAG_UNSET, ID))
56                 printf("Mark 04: Could not initialize flag_push on %d, error= %d\n", ID, error);
57         if (error = RCCE_flag_write(&flag_pop, RCCE_FLAG_UNSET, ID))
58                 printf("Mark 05: Could not initialize flag_pop on %d, error=%d\n", ID, error);
```

```
59          if (error = RCCE_flag_write(&flag_finished, RCCE_FLAG_UNSET, ID))
60                  printf("Mark 06: Could not initialize flag_finished on %d, error=%d\n", ID, error);
61
62          srand((unsigned) time(&t));
63          if (ID == 0) {
64                  stack_array = (stacknode *) malloc((SIZE + (SIZE / 2)) * sizeof(stacknode));
65                  head = (int *) malloc(sizeof(int));       //Allocating space for index
66                  *head = -1;                               // Initialize index
67                  for(i = 0; i < (SIZE / 2); i++) {         //Size inserts
68                          *head = i;
69                          stack_array[*head].num[0] = (rand() % 100001);
70                  }
71          }
72          RCCE_barrier(&RCCE_COMM_WORLD);
73          if (ID == 0) {
74                  stime1 = RCCE_wtime(); //Get starting time
75                  while (checkvar == 0) {
76                          for (i = 1; i < totalUEs; i++){
77                                  if (error = RCCE_flag_read(flag_push, &status, i))
78                                          printf("Mark 07: Could not read flag_push on %d, error=%d\n",
79                                                  i, error);
80                                  if (status == RCCE_FLAG_SET) {
81                                          //printf("Got a push message from %d\n", i);
82                                          server_push(i, buffer, stack_array, head, &flag_push, SIZE);
83                                  }
84                                  if (error = RCCE_flag_read(flag_pop, &status, i))
85                                          printf("Mark 08: Could not read flag_pop on %d, error=%d\n",
86                                          i, error);
87                                  if (status == RCCE_FLAG_SET) {
88                                          //printf("Got a pop message from %d\n", i);
89                                          server_pop(i, buffer, stack_array, head, &flag_pop);
90                                  }
91                          }
92                          for (i = 1; i < totalUEs; i++) {
93                                  if (error = RCCE_flag_read(flag_finished, &status, i))
94                                          printf("Mark 09: Could not read flag_finished on %d, error=%d
95  ____________________________________________________\n", i, error);
96                                  if (status == RCCE_FLAG_UNSET)
97                                          break;
98                                  if (i == totalUEs - 1)
99                                          checkvar = 1;
100                         }
101                 }
102         } else {
103                 op = ID;
```

```
104                 for (i = 0; i < iterations; i++) {
105                         if (data[op] == 1) {
106                                 num[0] = (rand() % 10000001) + ID;
107                                 client_push(ID, num, buffer, &flag_push);
108                                 RCCE_wait_until(flag_push, RCCE_FLAG_UNSET);
109                                 //Push is "non-blocking" we have to be sure it was processed by the
110                                 //server
111                         } else {
112                                 myvar = client_pop(ID, buffer, &flag_pop);
113                         }
114                         op += 1;
115                         if (op > 199)
116                                 op = 0;
117                 }
118                 if (error = RCCE_flag_write(&flag_finished, RCCE_FLAG_SET, ID))
119                         printf("Mark 10: Could not write flag_finished on %d, error=%d\n", ID, error);
120         }
121         ftime = RCCE_wtime();           //Get ending time
122         RCCE_free((t_vcharp) buffer);
123         if (ID == 0) {          //Printing and freeing stack and execution time
124                 printf("\nTotal time\t\t %f secs\nCommunication took\t %f secs\n", ftime - stime2,
125                         ftime - stime1);
126                 if (*head == -1)
127                         printf("Queue is empty\n");
128                 free(head);
129         }
130         RCCE_finalize();
131         return(0);
132 }
```

The code for the binary max heap.

```
1  /* Implementing a shared binary max heap, protected with lock from
2   * simultaneous accesses. Executing binary max heap operations
3   * (insert and delete max).
4   *
5   * We use core's 0 lock as a global lock for our data
6   * structure. To do any operation with the data you need to
7   * acquire this lock before proceeding.
8   *
9   * Tasoulas, Zois Gerasimos
10  * Microlab, ECE, NTUA
11  * 12th December, 2015
12  */
13
14  #include <stdint.h>
```

```
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <errno.h>
18 #include "RCCE.h"
19 #include "dstructslib.h"
20
21 int RCCE_APP(int argc, char *argv[])
22 {
23         int data[200] = {1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0,
24         0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1,
25         0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1,
26         1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1,
27         0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0,
28         1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0,
29         0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0};
30         int ID, i, j, totalUEs, error, op, *tail, iterations, SIZE, *data;
31         uint32_t num, *buffer;
32         double stime1, stime2, ftime;
33         time_t t;
34         heapnode *heap_array;
35
36         RCCE_init(&argc, &argv);
37         stime2 = RCCE_wtime();          //Get starting time
38         ID = RCCE_ue();
39         if (argc != 2) {
40                 if (ID == 0)
41                         printf("Size needed upon input\n");
42                 return(1);
43         }
44         SIZE = atoi(argv[1]);
45         totalUEs = RCCE_num_ues();
46         iterations = SIZE / totalUEs;
47         //8 integers because memory allocation should be product of 32, we just need one integer
48         tail = (int *) RCCE_shmalloc(8*sizeof(int));//Tail variable will hold the tail cell of the heap
49         if (tail == NULL){
50                 if (ID == 0)
51                         printf("01: Problem with allocating shared memory\n");
52                 return(1);
53         }
54         //Allocate heap, heapnode = 32B
55         heap_array = (heapnode *) RCCE_shmalloc((SIZE + (SIZE / 2)) * sizeof(heapnode));
56         if (heap_array == NULL){
57                 if (ID == 0)
58                         printf("02: Problem with allocating shared memory\n");
59                 return(1);
```

```
60          }
61          srand((unsigned) time(&t));
62          if (ID == 0) {                  // Initializing tail index
63                  *tail = ((SIZE / 2) - 1);
64                  RCCE_shflush();
65                  for (i = ((SIZE / 2) - 1); i >= (SIZE / 4); i--) {
66                          num = (rand() % 100001);        //Doing #SIZE/4 inserts to initialize the heap
67                          heap_array[i].num[0] = num;
68                  }
69                  for (i = ((SIZE / 4) - 1) ; i > 0; i--) {
70                          num = (rand() % 100001);
71                          heap_array[i].num[0] = num;
72                          combine(i, heap_array, *tail);
73                  }
74          }
75          RCCE_shflush();
76          op = ID;
77
78          RCCE_barrier(&RCCE_COMM_WORLD);
79          stime1 = RCCE_wtime();          //Get starting time
80          for (i = 0; i < iterations; i++) {
81                  if (data[op] == 1) {
82                          num = (rand() % 100001) + ID;
83                          //printf("Insert element %d, core %d\n", num, ID);
84                          RCCE_acquire_lock(0);  //Everybody uses lock(0)
85                          locked_insert(ID, num, tail, heap_array, SIZE);
86                          RCCE_release_lock(0);
87                  } else {
88                          RCCE_acquire_lock(0);
89                          num = locked_extract(ID, tail, heap_array);
90                          //printf("Extract element %d, core %d\n", num, ID);
91                          RCCE_release_lock(0);
92                  }
93                  op += 1;
94                  if (op > 199)
95                          op = 0;
96          }
97          RCCE_barrier(&RCCE_COMM_WORLD);
98          ftime = RCCE_wtime();           //Get ending time
99          if (ID == 0) {          //Freeing heap and printing execution time
100                 printf("\nTotal_time\t\t_%f_secs\nCommunication_took\t_%f_secs\n", ftime - stime2,
101                         ftime - stime1);
102                 RCCE_shflush();         //To be sure we read the updated value of head
103                 if (*tail < 0)
104                         printf("Heap_is_empty.\n");
```

```
105         }
106         RCCE_shfree((t_vcharp) tail);
107         RCCE_shfree((t_vcharp) heap_array);
108         RCCE_finalize();
109         return(0);
110 }
```

The code for the fifo queue.

```
1  /* Implementing a shared data structure protected with locks from simultaneous
2   * accesses. Executing fifo list operations (insert and delete).
3   *
4   * We use core's 0 lock to access the end point of the structure, so to do
5   * insertions and core's 1 lock to access the starting point, so to do removals.
6   * To do any operation with the data you need to acquire one of these locks before
7   * proceeding.
8   *
9   * Tasoulas, Zois Gerasimos
10  * Microlab, ECE, NTUA
11  * 11th December, 2015
12  */
13
14 #include <stdint.h>
15 #include <stdlib.h>
16 #include <stdio.h>
17 #include <errno.h>
18 #include "RCCE.h"
19 #include "dstructslib.h"
20
21 int RCCE_APP(int argc, char *argv[])
22 {
23         int data[200] = {1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0,
24         0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1,
25         0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1,
26         1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1,
27         0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0,
28         1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0,
29         0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0};
30         int ID, i, totalUEs, error, op, *head, *tail, iterations, SIZE;
31         uint32_t num, *buffer;
32         double stime1, stime2, ftime;
33         time_t t;
34         listnode *list_array;
35
36         RCCE_init(&argc, &argv);
37         stime2 = RCCE_wtime();          //Get starting time
```

```
38          ID = RCCE_ue();
39          if (argc != 2) {
40                  if (ID == 0)
41                          printf("SIZE needed upon input\n");
42                  return(1);
43          }
44          SIZE = atoi(argv[1]);
45          totalUEs = RCCE_num_ues();
46          iterations = SIZE / totalUEs;
47          head = (int *) RCCE_shmalloc(8*sizeof(int));//Head variable will hold the head cell of the list
48                  //8 integers because memory allocation should be product of 32, we just need one integer
49          tail = (int *) RCCE_shmalloc(8*sizeof(int));//Tail variable will hold the tail cell of the list
50          if (head == NULL){
51                  if (ID == 0)
52                          printf("01: Problem with allocating shared memory\n");
53                  return(1);
54          }
55          if (tail == NULL){
56                  if (ID == 0)
57                          printf("02: Problem with allocating shared memory\n");
58                  return(1);
59          }
60          list_array = (listnode *) RCCE_shmalloc((SIZE + (SIZE / 2)) * sizeof(listnode));
61          if (list_array == NULL){
62                  if (ID == 0)
63                          printf("03: Problem with allocating shared memory\n");
64                  return(1);
65          }
66          srand((unsigned) time(&t));
67  // Initializing head, tail index. Initializing queue only by core #0, to be similar to CS implemantation
68          if (ID == 0) {
69                  *head = -1;
70                  *tail = 0;
71                  RCCE_shflush();
72                  for (i = 0; i < (SIZE / 2); i++) {
73                          num = (rand() % 100001) + ID;          //Doing totalUEs enqueues to
74                          list_array[i].num[0] = num;
75                  }
76                  *head = (SIZE / 2) - 1;
77                  RCCE_shflush();
78          }
79          RCCE_barrier(&RCCE_COMM_WORLD);  //To ensure all cores have finished their operations
80          stime1 = RCCE_wtime();          //Get starting time
81          op = ID;
82          for (i = 0; i < iterations; i++) {
```

```
83                  if ((data[op] % 2) == 1) {
84                          num = (rand() % 100001) + ID;
85                  //      printf("Push element %d, core %d\n", num, ID);
86                          RCCE_acquire_lock(0);  //lock(0) for insert
87                          locked_enqueue(ID, num, head, list_array, SIZE);
88                          RCCE_release_lock(0);
89                  } else {
90                          RCCE_acquire_lock(1);  //lock(1) for remove
91                          num = locked2_dequeue(ID, tail, list_array );
92                  //      printf("Pop element %d, core %d\n", num, ID);
93                          RCCE_release_lock(1);
94                  }
95                  op += 1;
96                  if (op > 199)
97                          op = 0;
98          }
99          RCCE_barrier(&RCCE_COMM_WORLD);  //To ensure all cores have finished their operations
100         ftime = RCCE_wtime();           //Get ending time
101         if (ID == 0) {          //Printing and freeing queue and execution time
102                 printf("\nTotal_time\t\t %f_secs\nCommunication_took\t %f_secs\n", ftime - stime2,
103                         ftime - stime1);
104                 RCCE_shflush();         //To be sure we read the updated value of head
105                 if (*head < *tail || *head == -1)
106                         printf("List is empty.\n");
107 /*              while (*head >= *tail) {
108                         printf("Element is %d\n", list_array[*head].num[0]);
109                         (*head) -= 1;
110                 }
111 */
112         }
113         RCCE_shfree((t_vcharp) head);
114         RCCE_shfree((t_vcharp) tail);
115         RCCE_shfree((t_vcharp) list_array);
116         RCCE_finalize();
117         return(0);
118 }
```

Bibliography

[1] M. Acharya. *Non-blocking Concurrent Operations on Heap*, UNLV theses, University of Nevada, Las vegas, USA. 2012.

[2] R. Arpaci-Dusseau and A. Arpaci-Dusseau. Locked Data Structures. *Operating Systems: Three Easy Pieces, Arpaci-Dusseau Books*, Chapter 29, pp. 29-1 – 29-13, 2015.

[3] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Morgan Kaufmann Publishers*, 2008.

[4] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Wijngaart and T. Mattson. A 48-core IA-32 Message-Passing Processor with DVFS in 45 nm CMOS.*ISSCC 2010, Session 5, Processors, 5.7*, pp. 1-3, 2010.

[5] S. Kim, S. Lee, M. Jun, B. Lee, W. Ro, E. Chung, J. Gaudiot. C-Lock: Energy Efficient Synchronization for Embedded Multicore Systems. *Computers, IEEE Transactions on*, vol. 63, no. 8, pp. 1962-1974, 2014.

[6] T. Mattson *Using Intel's Single-Chip Cloud Computer (SCC)*, pp. 1-49.

[7] T. Mattson, R. Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl and S. Dighe. The 48-core SCC processor: the programmer's view. *IEEE, SC10 November 2010*, pp. 1-11, 2010.

[8] M. Moir and N. Shavit. Concurrent Data Structures. *Handbook of Data Structures and Applications, D. Metha and S.Sahni Editors, Chapman and Hall/CRC Press* , Chapter 47, pp. 47-1 – 47-17, 2004.

[9] *Parallel Processing Systems, Computing Systems Laboratory, School of Electrical and Computer Engineering, National Technical University of Athens* , http://www.cslab.ece.ntua.gr/courses/pps/files/fall2015/pps-notes-Fall2015.pdf, 2015.

[10] R. Rotta. On Efficient Message Passing on the Intel SCC. *3rd Many-core Applications Research Community (MARC) Symposium.* Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, pp.1-6, 2011.

[11] R. Rotta, T. Prescher, J. Traue and J. Nolte. In-Memory Communication Mechanisms for Many-Cores - Experiences with the Intel SCC. pp. 1-6, 2012.

[12] S. Sutirtha, S. Roy, A. Cristal, O. Unsal and M. Valero. Clock Gate on Abort: Towards Energy-Efficient Hardware Transactional Memory. *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.* IEEE, pp. 1-8, 2009 .

[13] *The SCC Platform Overview*, Revision 0.80, Intel Labs, pp. 1-23, 2012.

[14] R. Wijngaart and T. Mattson. RCCE A small library for many-core communication. *Intel Labs Single-chip Cloud Computer Symposium*, pp. 1-51, 2010.

[15] I. Walulya, Y. Nikolakopoulos, M. Papatriantafilou and P. Tsigas. Concurrent Data Structures in Architectures with Limited Shared Memory Support. *Parallel Processing Workshops - Euro-Par 2014 International Workshops*, Porto, Portugal, pp.1-12, 2014.