ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Τεχνικές χρονοδρομολόγησης εφαρμογών για δίκαιη κατανομή πόρων σε πολυπύρηνες αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ
του

Θεόδωρου Ι. Μαρινάκη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Τεχνικές χρονοδρομολόγησης εφαρμογών για δίκαιη κατανομή πόρων σε πολυπύρηνες αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ
του

Θεόδωρου Ι. Μαρινάκη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 1η Ιουλίου 2016.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Λέκτορας Ε.Μ.Π.	Δημήτριος Τσουμάκος Επίκουρος Καθηγητής Ιονίου Πανεπιστημίου

Αθήνα, Ιούλιος 2016

(Υπογραφή)

.....
ΜΑΡΙΝΑΚΗΣ ΘΕΟΔΩΡΟΣ
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η χρονοδρομολόγηση είναι μια διαδικασία, η οποία αφορά το διαμοιρασμό πόρων σε εργασίες ανά τακτά χρονικά διαστήματα, με σκοπό τη βελτιστοποίηση ορισμένων στόχων. Ένας τομέας, στον οποίο χρησιμοποιείται κυρίως, είναι τα υπολογιστικά συστήματα. Όσον αφορά μονοπύρηνες αρχιτεκτονικές έχουν αναπτυχθεί τεχνικές χρονοδρομολόγησης, οι οποίες ανταπεξέρχονται αποδοτικά στο διαμοιρασμό του χρόνου του πυρήνα ανάμεσα στις διεργασίες του συστήματος. Η ανάγκη όμως για εξυπηρέτηση μεγαλύτερου φόρτου εργασίας οδήγησε τους κατασκευαστές στη δημιουργία πολυπύρηνων αρχιτεκτονικών ενσωματώνοντας πολλούς πυρήνες μέσα σε ένα ολοκληρωμένο κύκλωμα (CMP). Αντίθετα με τα συμμετρικά πολυπύρηνα συστήματα (SMP), οι πυρήνες των CMPs δεν θεωρούνται ανεξάρτητες μονάδες εφόσον μοιράζονται στοιχεία της αρχιτεκτονικής, όπως το τελευταίο επίπεδο κρυφής μνήνης και το δίαυλο μνήμης.

Στα πολυπύρηνα συστήματα αυτά (CMP) ενσωματώθηκαν, αυτούσιες ή με λίγες αλλαγές, οι τεχνικές χρονοδρομολόγησης, που είχαν υλοποιηθεί στις συμμετρικές (SMP) αρχιτεκτονικές. Εκτελώντας προγράμματα παράλληλα σε πυρήνες που μοιράζονται στοιχεία υπάρχει πιθανότητα να παρατηρήσουμε μείωση στην απόδοσή τους λόγω ανταγωνισμού μεταξύ τους. Υιοθετώντας τεχνικές που δε λαμβάνουν υπόψιν τους τον ανταγωνισμό αυτόν, θεωρώντας τους πυρήνες απομονωμένες μονάδες που δεν μοιράζονται στοιχεία, οδηγούμαστε σε αποτελέσματα που δεν πληρούν τις αρχές της χρονοδρομολόγησης. Μερικά προβλήματα που δημιουργούνται είναι τα εξής: πολύ χαμηλή απόδοση των εφαρμογών, άνισος διαμοιρασμός των πυρήνων μεταξύ τους και συνεπώς απρόβλεπτη και ασταθής συμπεριφορά του συστήματος.

Ο σκοπός αυτής της διπλωματικής είναι να παρουσιάσουμε τεχνικές που βελτιώνουν την άνιση κατανομή των πόρων, λαμβάνοντας υπόψιν την διαμάχη των διεργασιών για τα μοιραζόμενα στοιχεία του τσιπ και τις καταστροφικές συνέπειες που αυτή αποφέρει στην αξιοπιστία του συστήματος. Επικεντρωνόμαστε σε τεχνικές που προσφέρουν δικαιοσύνη στις εφαρμογές με τους παρακάτω τρόπους. Στην πρώτη προσέγγιση σχεδιάζουμε έναν χρονοδρομολογητή που προταιρεότητα του είναι να αποφύγει τον ανταγωνισμό των διεργασιών, βασισμένοι σε ένα σχήμα ταξινόμησης και σε ένα μοντέλο που προβλέπει πως οι διάφορες κατηγορίες διεργασιών αλληλεπιδρούν. Η επόμενη μέθοδος προσπαθεί να διαχειριστεί τον ανταγωνισμό των διεργασιών. Λαμβάνει υπόψιν τη μείωση της απόδοσης των διεργασιών και τις ευνοεί αναλόγως. Αυτο το επιτυγχάνει δίνοντας ευκαιρία συχνότερης εκτέλεσης σε εκείνες που υποφέρουν περισσότερο, σε βάρος εκείνων που εκτελούνται με υψηλότερη απόδοση. Τέλος επεκτείναμε την δεύτερη τεχνική μας, έτσι ώστε να αποφεύγει μέρος του ανταγωνισμού και να διαχειρίζεται το υπόλοιπο.

Δοκιμάζοντας τις τεχνικές μας σε μια ποικιλία πειραμάτων και συγκρίνοντας τα αποτελέσματα με χρονοδρομολογητή που χρησιμοποιείται σε εμπορικά πολυπύρηνα συστήματα (Linux scheduler), καταλήξαμε στο εξής συμπέρασμα. Οι προτεινόμενες εναλλακτικές είναι ικανές να βελτιώσουν σε μεγάλο βαθμό την άνιση κατανομή των πόρων και να προσφέρουν αξιοπιστία στην απόδοση του συστήματος.

Λέξεις Κλειδιά: Χρονοδρομολόγηση, ανταγωνισμός, μοιραζόμενοι πόροι, πολυπύρηνες αρχιτεκτονικές, δικαιοσύνη

Abstract

Scheduling is a decision-making process that deals with the assignment of resources to tasks over given periods, aiming to optimize one or more objectives. Responsible for efficient distribution of the CPU time among the processes, scheduler has become an essential part of computer systems. OS schedulers for single processor architectures have become so optimized that need for further improvements dramatically subsided. The scheduling problem was considered solved until the arrival of chip multiprocessors (CMP). Driven by the critical problems of transistor shrinking, heat generation, power consumption and poor performance improvement, manufacturers abandoned single core architectures and turned to chips with multiple cooler-running, energy-efficient processing cores. In order to provide a cost-effective solution, they integrated the cores into a single circuit die, sharing architectural components, such as the last level cache and the memory bus.

Scheduling policies developed for symmetric multiprocessors (SMP) have been integrated without modifications into CMPs. While applications run on neighboring cores of a CMP, they contend with each other for the shared resources. This contention can result in great performance degradation for the applications that are concurrently executed. For this reason, treating the cores of a CMP as isolated and independent units is a very optimistic abstraction and can cause great problems to the objectives a scheduler tries to optimize. First and foremost, it cannot be assured that applications would make adequate progress. In addition, it is observed that resources cannot be fairly distributed among the applications of the system. As a result, poor fairness enforcement can lead to unstable and unpredictable performance of the system.

In this paper, we develop 3 scheduling techniques aiming to improve the unequal sharing of the resources, taking into consideration the destructive effects of applications interference, when contending for the shared resources. On our first approach, we attempt to avoid pairing applications that interfere destructively. We build a contention-aware scheduler, based on a classification scheme and an interference prediction model. On the next approach, we manage the results of the interference, boosting the performance of applications based on their IPC reduction. Applications that are heavily impacted have the opportunity to increase their running time at the expense of the well-performed. Finally, we extend our previous technique, aiming to avoid a part of the contention and manage the rest of it.

Evaluating our proposed scheduling policies on a variety of workloads and comparing them with the Linux scheduler, we come to the following conclusions. All of them manage to overcome the problem of unfair distribution, creating an environment where Quality of Service (QoS) guarantees are possible to be provided and Service Level of Agreements (SLAs) can be enforced.

Key words: Scheduling, contention, shared resources, chip multiprocessor, fairness

Ευχαριστίες

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Ν. Κοζύρη που μου έδωσε την ευκαιρία να ασχοληθώ με αυτό το θέμα. Επίσης θα ήθελα να ευχαριστήσω το κ. Γ. Γκούμα για τις συμβουλές του και τον Αλέξανδρο Χαριτάτο, που με βοήθησε στην περάτωση της εργασίας.

Τέλος θα ήθελα να ευχαριστήσω τον πατέρα και την μητέρα μου που με υποστήριξαν σε όλη μου τη προσπάθεια και με ενθάρρυναν διαρκώς με την αγάπη τους. Ευχαριστώ ιδιαιτέρως τη Φλώρα που στάθηκε στο πλευρό μου τις πιο δύσκολες στιγμές μου και τους φίλους μου που με άκουγαν υπομονετικά.

Κεφάλαιο 1
Εισαγωγή

1.1 Χρονοδρομολόγηση και Εφαρμογές

Η χρονοδρομολόγηση είναι μια διαδικασία, κατά την οποία λαμβάνονται αποφάσεις για το διαμοιρασμό πόρων σε εργασίες, με τέτοιο τρόπο ώστε να επιτυγχάνονται συγκεκριμένοι στόχοι. Παίζει πολύ σημαντικό ρόλο σε διάφορους τομείς και έχει εφαρμογές σε εργοστάστια παραγωγής, σε υπηρεσίες μεταφορών και σε υπολογιστικά συστήματα. Σε κάθε τομέα οι πόροι, οι εργασίες και οι προς εκπλήρωση στόχοι παίρνουν ποικίλες μορφές.

Ας εξετάσουμε, αρχικά, ένα εργοστάσιο που παράγει χάρτινες συσκευασίες για σκυλοτροφές, κάρβουνα και τσιμέντο. Η διαδικασία παραγωγής αποτελείται απο διάφορα στάδια, όπως τύπωμα λογοτύπου ή ράψιμο των πλευρών της συσκευασίας. Σε κάθε στάδιο τα μηχανήματα διαφέρουν ως προς τα χαρακτηριστικά τους και τις λειτουργίες που εξυπηρετούν. Τα μηχανήματα αυτά αποτελούν τους πόρους του εργοστασίου. Ανάλογα με το είδος της παραγγελίας, κάποιες εργασίες πρέπει να πραγματοποιηθούν στα μηχανήματα. Οι παραγγελίες δεν αφορούν μόνο ένα προϊόν, άρα και οι λειτουργίες των μηχανημάτων διαφέρουν απο παραγγελία σε παραγγελία. Ένας στόχος θα μπορούσε να είναι η ελαχιστοποίηση του χρόνου παραγωγής μιας συγκεκριμένης παραγγελίας, αν υποθέσουμε ότι ένας υψίστης σημασίας πελάτης απαιτεί γρήγορη παράδοση. Ένας άλλος στόχος μπορεί να είναι η αποδοτική λειτουργία των μηχανημάτων για εξοικονόμηση ενέργειας και κόστους. Σε κάθε περίπτωση βλέπουμε ότι είναι αναγκαία η εφαρμογή μίας πολιτικής, που θα μοιράζει τους πόρους στις εργασίες με τέτοιο τρόπο, ώστε συγκεκριμένοι στόχοι να εξυπηρετούνται.

Η χρονοδρομολόγηση συναντάται, επίσης, στα υπολογιστικά συστήματα. Αποτελεί αναπόσπαστο κομμάτι του λειτουργικού συστήματος και η κύρια αρμοδιότητα της είναι να παρέχει ένα περιβάλλον, στο οποίο πολλά προγράμματα μπορούν να τρέχουν παράλληλα. Μια διεργασία δεν μπορεί να κρατάει τον επεξεργαστή απασχολημένο όλη την ώρα. Το λειτουργικό σύστημα οφείλει να προσφέρει τμήμα του χρόνου του επεξεργαστή σε όλες τις διεργασίες και να διαβεβαιώνει ότι αυτές κάνουν πρόοδο. Ο δρομολογητής αποτελεί απαραίτητο κομμάτι του λειτουργικού, διότι εκτός από το να προσφέρει ένα πολυπρογραμματιστικό περιβάλλον, παίζει βαρίνουσας σημασίας ρόλο στη συνολική απόδοση του συστήματος, καθώς οι στόχοι που προσπαθεί να ικανοποιήσει σχετίζονται με την απόδοσή του. Μερικοί απο τους στόχους είναι οι εξής: να κρατάμε τον επεξεργαστή όσο περισσότερο απασχολημένο γίνεται, να εκτελούμε όσο το δυνατόν περισσότερες διεργασίες ανα μονάδα χρόνου ή να ελαχιστοποιούμε το χρόνο των διεργασιών που μένουν εκτός επεξεργαστή. Ανεξαρτήτως στόχου, ο δρομολογητής εφαρμόζει δύο πολιτικές. Στην μία μοιράζει τον επεξεργαστή χρονικά και στην άλλη τοπικά, δηλαδή πως σε μία χρονική στιγμή θα μοιραστούν οι πυρήνες στις διεργασίες. Όταν πρόκειται για μονοπύρηνες αρχιτεκτονικές η μόνη πολιτική που μπορεί να εφαρμοστεί είναι η κατανομή του χρόνου, ενώ σε πολυπύρηνες αρχιτεκτονικές μπορούν να εφαρμοστούν και οι δύο.

1.2 Πολυπύρηνα Συστήματα (CMPs)

Οι κατασκευαστές για χρόνια επιδίωκαν την αύξηση της απόδοσης του επεξεργαστή, αυξάνωντας τον αριθμό των τρανζίστορ και συρρικνώνοντας τα ώστε να μειώσουν το συνολικό μέγεθος του τσιπ. Αυτή η τάση, όμως, δεν μπορεί να συνεχιστεί για πάντα, καθώς οι δυνατότητες συρρίκνωσης των τρανζίστορ περιορίζονται και προβήματα κατανάλωσης και θερμότητας συνεχώς αυξάνονται. Έτσι η βελτίωση της απόδοσης του επεξεργαστή άρχισε να μειώνεται. Στη δεκαετία του 90 η απόδοση αυξανόταν κατά 60 % κάθε χρόνο,ενώ απο το 2000 έως το 2004 η αύξηση έπεσε στο 40% κάθε χρόνο. Η κατασκευή ενός ελάχιστα, κατά 20% γρηγορότερου τσιπ με διπλάσιο μέγεθος δεν ανταποκρίνεται στις προσδοκίες μας για βελτιωμένη απόδοση, αποδοτική κατανάλωση και μειωμένο κόστος.

Για αυτούς τους λόγους, στραφήκαμε στην κατασκευή πολυπύρηνων συστημάτων. Σε αντίθεση με τις μονοπύρηνες αρχιτεκτονικές, ενσωματώνουμε στο ίδιο τσιπ πολλούς πυρήνες, οι οποίοι λειτουργούν σε ελαφρώς χαμηλότερες ταχύτητες και μοιράζονται στοιχεία αρχιτεκτονικής (CMPs). Ο διαμοιρασμός του φόρτου σε πολλές μονάδες επεξεργασίας, οι οποίες μπορούν να εκτελούν προγράμματα παράλληλα αποφέρει σημαντική αύξηση στην συνολική απόδοση. Από την άλλη, το κόστος για την κατασκευή και τη λειτουργία τους (κατανάλωση) είναι χαμηλότερο συγκριτικά με τα συμμετρικά πολυπύρηνα συστήματα, καθώς οι επεξεργαστές μοιράζονται στοιχεία και η επικοινωνία τους γίνεται γρηγορότερα.

Λόγω των πλεονεκτημάτων που προσφέρουν, αυτές οι αρχιτεκτονικές έχουν κυρίαρχη θέση στα υπολογιστικά συστήματα. Χρησιμοποιούνται ευρέως σε πολλούς τομείς, απο ενσωματωμένα σε συστήματα επεξεργασίας ψηφιακών σημάτων. Όσον αφορά την βιομηχανία, η AMD, η Fujitsu, η IBM, η Intel και η Sun Microsystems κατασκευάζουν ήδη πολυπύρηνα συστήματα και σκοπεύουν να σχεδιάσουν νέα μοντέλα στο μέλλον.

1.3 Ο χρονοδρομολογητής του λειτουργικού συστήματος Linux

Όπως αναφέραμε νωρίτερα, ο ρόλος των δρομολογητών στα CMPs είναι να μοιράζει τους επεξεργαστές όχι μόνο χρονικά αλλά και τοπικά. Δηλαδή να αποφασίζουν σε κάθε χρονική στιγμή ποιες διεργασίες θα ανατεθούν στους πυρήνες. Ο κύριος σκοπός τους είναι να μοιράζουν το φόρτο εργασίας ίσα μεταξύ των πυρήνων. Ας εξετάσουμε έναν δρομολογητή που χρησιμοποιείται στο λειτουργικό σύστημα του Linux, ώστε να καταλάβουμε καλύτερα πως το σύνολο των διεργασιών διαχειρίζεται στα CMPs.

Η χρονοδρομολόγηση γίνεται σε δύο επίπεδα. Στο πρώτο επίπεδο χρησιμοποιούνται ξεχωριστές ουρές διεργασιών για κάθε πυρήνα και πολιτικές για την διαχείριση τους. Στο δεύτερο επίπεδο υπάρχει ο load balancer που μοιράζει τις διεργασίες στους πυρήνες. Με αυτόν τον τρόπο επιτυγχάνουμε κατανομή στο χρόνο (πρώτο επίπεδο) και στο χώρο (δεύτερο επίπεδο).

Η ουρά διεργασιών σε κάθε πυρήνα αντιπροσωπεύει το σύνολο των προγραμμάτων που έχει αναλάβει να εκτελέσει αυτός ο πυρήνας. Η πολιτική που χρησιμοποιείται για την διαχείρηση κάθε ουράς βασίζεται στην εξής αρχή. Η διεργασία που έχει την μεγαλύτερη ανάγκη για υπολογιστικό χρόνο θα είναι η επόμενη που θα δρομολογηθεί. Σκοπός αυτής της πολιτκής είναι να προσφέρει δίκαιη κατανομή του χρόνου μεταξύ των διεργασιών και για αυτό το λόγο ονομάζεται CFS (Completely Fair Scheduler). Για να το πετύχει αυτό, λαμβάνει

υπ’όψιν του το χρόνο που οι διεργασίες βρίσκονται εκτός πυρήνα και επιλέγει κάθε φορά αυτήν με τον μεγαλύτερο. Με αυτόν τον τρόπο διαβεβαιώνει ότι καμία διεργασία δεν θα μένει εκτός πυρήνα για περισσότερο χρόνο, μοιράζοντας τους πόρους δίκαια μεταξύ των. Η ουρά διεργασιών υλοποιείται με ένα time-ordered red black tree για κάθε πυρήνα

Στις CMP αρχιτεκτονικές κάθε πυρήνας έχει τη δικιά του ουρά διεργασιών. Εφ’όσον οι χρόνοι εκτέλεσης των διεργασιών διαφέρουν, είναι πιθανό μερικοί επεξεργαστές να διαχειρίζονται λιγότερο αριθμό διεργασιών. Αυτήν την άνιση κατανομή φόρτου προσπαθεί να μετριάσει ο load balancer, ο οποίος εφαρμόζεται περιοδικά και σκοπός του είναι να ισορροπεί το μήκος των ουρών σε όλους τους πυρήνες. Αυτό το επιτυγχάνει μετακινόντας διεργασίες απο το πιο απασχολημένο πυρήνα σε εκείνο με το λιγότερο φόρτο. Η εξισορρόπηση του φόρτου γίνεται ιεραρχικά, ξεκινώντας απο επίπεδο που αφορά διαφορετικά NUMA domains καταλήγοντας στο επίπεδο των SMT contexts. Η συχνότητα και ο αριθμός των μετακινήσεων διαφέρει απο επίπεδο σε επίπεδο.

1.4 Περιγραφή Κεφαλαίων

Στο δέυτερο κεφάλαιο παρουσιάζουμε την επίδραση των εφαρμογών στην απόδοση τους,όταν εκείνες ανταγωνίζονται για τους μοιραζόμενους πόρους. Επιπλέον μελετάμε πως η αντιμετώπιση των πόρων απο σύγχρονο χρονοδρομολογητή, επηρεάζει την απόδοση του συστήματος.

Στο τρίτο κεφάλαιο προτείνουμε 3 τεχνικές χρονοδρομολόγησης, με σκοπό να βελτιώσουμε τη σταθερότητα του συστήματος και να προσφέρουμε δικαιη κατανομή των πόρων

Στο τέταρτο κεφάλαιο, ελέγχουμε τις τεχνικές μας σε μία ποικιλία φόρτων εργασίας και συγκρίνουμε τα αποτελέσματα μας με το Linux scheduler.

Στο πέμπτο κεφάλαιο, παρουσιάζουμε τα συμπεράσματα μας και πιθανές ιδέες για βελτίωση, ενώ στο τελευταίο κεφάλαιο αναφέρουμε διάφορες προσεγγίσεις που έχουν γίνει από την επιστημονική κοινότητα.

Κεφάλαιο 2
Ορισμός του Προβλήματος και Κίνητρο

2.1 Δημιουργική και καταστροφική συμπεριφορά

Όπως αναφέραμε προηγουμένως, οι πυρήνες των CMP είναι ενσωματομένοι σε ένα τσιπ και μοιράζονται στοιχεία της αρχιτεκτονικής, όπως την κρυφή μνήμη τελευταίου επιπέδου και το δίαυλο μνήμης. Όταν εκτελούνται εφαρμογές παράλληλα στους πυρήνες οι μοιραζόμενοι πόροι μπορεί να έχουν είτε θετικό είτε αρνητικό ανίκτυπο στην απόδοσή τους.

Στην περίπτωση που συνεισφέρουν δημιουργικά, οι εφαρμογές μπορούν να γνωρίσουν τρομερή βελτίωση στην απόδοση τους. Αυτό μπορεί να επιτευχθεί όταν νήματα μιας εφαρμογής κάνουν αναφορά στα ίδια ή σε τοπικά δεδομένα ταυτόχρονα (fine-grained sharing). Με αυτό το τρόπο μερικά νήματα φέρνουν δεδομένα στην κρυφή μνήμη και εξυπηρετούν και τα γειτονικά νήματα που χρησιμοποιούν τα ίδια, χωρίς να χρειάζεται να κάνουν εκ νέου

αναφορά στην μνήμη.

Από την άλλη μεριά, οι μοιραζόμενοι πόροι μπορεί να έχουν καταστροφικές συνέπειες για τις εφαρμογές. Αυτό συμβαίνει όταν τρέχουν παράλληλα νήματα διαφορετικών διεργασιών, που δεν έχουν καμία επικοινωνία μεταξύ τους. Επίσης νήματα των ίδιων διεργασιών είναι δυνατό να έχουν ένα μοτίβο αναφοράς στα δεδομένα που να μην εξυπηρετούν τα γειτονικά τους. Για παράδειγμα μπορεί ένα νήμα να αναφέρεται σε ένα τμήμα δεδομένων για κάποιο χρονικό διάστημα και το γειτονικό του να αναφερθεί σε αυτό αφού τελειώσει την αναφορά το αρχικό(coarse-grained sharing). Συνεπώς δεν μπορεί το ένα να βοηθήσει το άλλο και συμπεριφέρονται σαν νήματα ξεχωριστών διεργασιών. Σε όλες τις περιπτώσεις τα νήματα προσπαθούν να ικανοποιήσουν τις απαιτήσεις τους, χρησιμοποιώντας τους μοιραζόμενους πόρους για δικό τους λογαριασμό αποκλειστικά και ανταγωνίζονται μεταξύ τους για την ικανοποίηση των αναγκών τους. Αυτή η διαμάχη για τη χρήση των μοιραζόμενων πόρων έχει σημαντικές αρνητικές συνέπειες στην απόδοσή των εφαρμογών, καθώς παρατηρείται καθυστέρηση στην εκτέλεση τους συγκριτικά με τον όταν τρέχουν μόνες τους.

Μελετάμε παρακάτω την σύγκρουση σε δύο μοιραζόμενους πόρους, τη LLC και το Memory Bandwidth

2.2 Ανταγωνισμός για τη κρυφή μνήμη

Όταν οι διεργασίες τρέχουν παράλληλα ο πρώτος πόρος, για τον οποίο έρχονται σε σύγκρουση είναι η κρυφή μνήμη. Διεργασίες φέρνουν δεδομένα στη κρυφή μνήμη με σκοπό να ικανοποιήσουν τις απαιτήσεις τους. Στην περίπτωση που τρέχει μία διεργγασία η κρυφή μνήμη περιέχει τα δεδομένα της. Όταν όμως τρέξει παράλληλα μία άλλη στον γειτονικό πυρήνα, φέρνει τα δικά της δεδομένα, διώχνοντας απο τη κρυφή μνήμη τα δεδομένα της άλλης. Αυτό έχει σαν αποτέλεσμα την αύξηση των αποτυχιών και συνεπώς μεγαλύτερες καθυστερήσεις στην εκτέλεση της διεργασίας. Τρέχοντας σε ένα τετραπύρηνο σύστημα με κοινή LLC (θα αναλύσουμε λεπτομερώς παρακάτω) 4 εφαρμογές παρατηρούμε τις καταστροφικές συνέπειες που έχει αυτή η διαμάχη στην απόδοση τους. (Figure1.1-1)

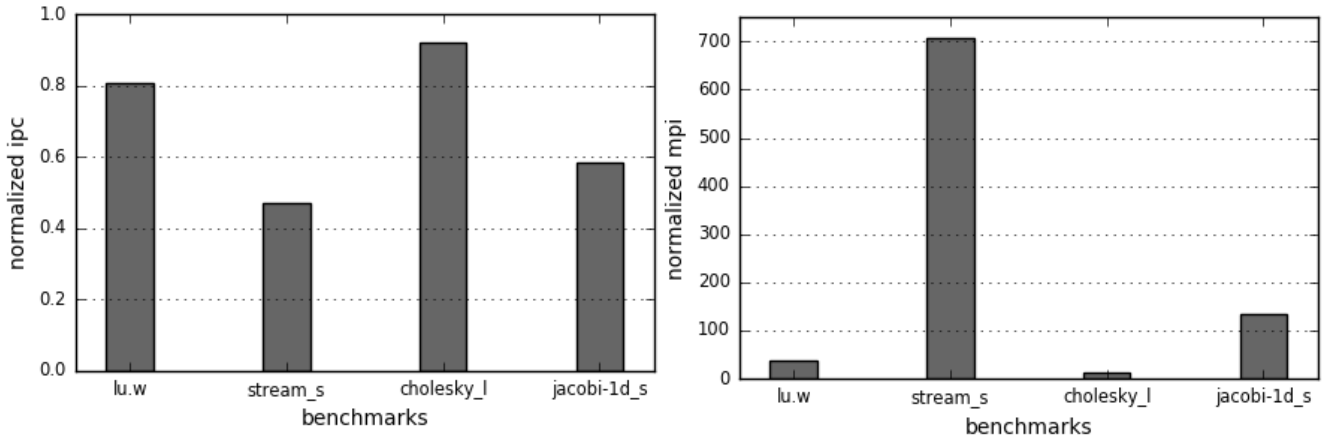


Figure 1.1-1: Επίπτωση της αύξησης των MPI στο IPC 4 διεργασιών που τρέχουν παράλληλα

Παρατηρούμε, λοιπόν, ότι ο ανταγωνισμός για την κρυφή μνήμη προκαλεί αύξηση στα

MPI των εφαρμογών. Βλέπουμε, επίσης, ότι η εφαρμογές δεν επηρεάζονται το ίδιο, μερικές επωφελούνται περισσότερο από τη κρυφή μνήμη ενώ άλλες όχι.

2.2 Ανταγωνισμός για το Memory Bandwidth

Ένα άλλο σημείο σύγκρουσης είναι ο δίαυλος της μνήμης. Ο ρυθμός, με τον οποίο τα δεδομένα διαβάζονται ή γράφονται από και προς τη μνήμη, ονομάζεται Memory Bandwidth. Όταν τρέχουν παράλληλα διεργασίες, μολύνουν το δίαυλο μνήμης με δεδομένα. Το σύστημα της μνήμης, όπως είναι φυσικό, έχει ένα όριο στο ρυθμό εξυπηρέτησης των δεδομένων. Αυτό σημαίνει ότι υπάρχει ένα σημείο στο οποίο το Bandwidth δεν μπορεί να ικανοποιήσει όλες τις διεργασίες. Σε αυτή τη περίπτωση το Bandwidth μοιράζεται μεταξύ των διεργασιών και κάθε μία παίρνει ένα τμήμα του αρχικού εύρους που θα έπαιρνε, όταν έτρεχε μόνη της. Χρησιμοποιώντας streaming εφαρμογές, δηλαδή εφαρμογές που χρησιμοποιούν δεδομένα από την μνήμη έως τον επεξεργαστή χωρίς να επωφελούνται απο τις κρυφές μνήμες, παρατηρούμε πώς η διαμάχη για το Memory Bandwidth επηρεάζει την απόδοσή τους.

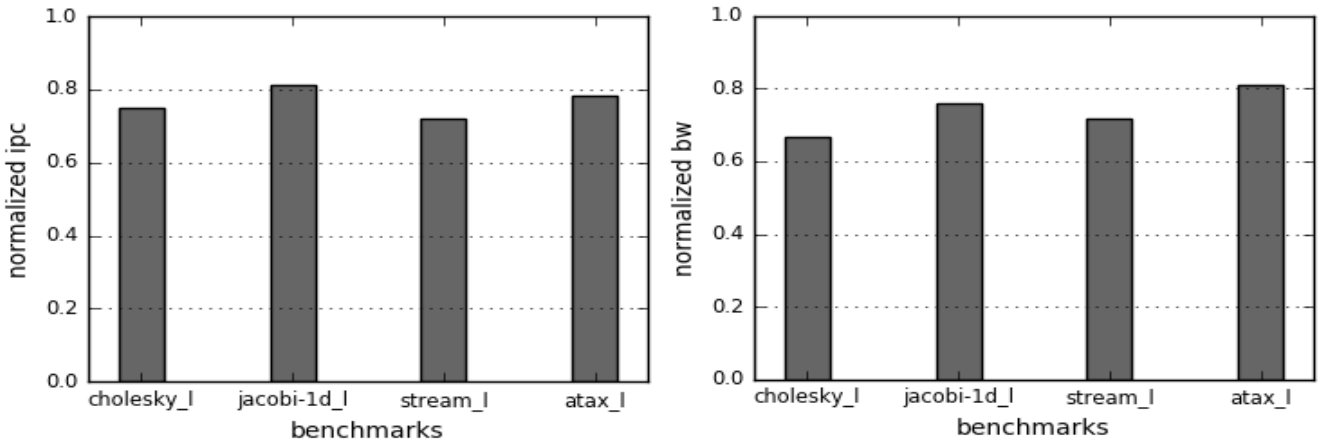


Figure 1.1-2: Η μείωση του Bandwidth έχει άμεσο αντίκτυπο στο IPC 4 streaming εφαρμογών

Παρατηρούμε ότι η σύγκρουση στο δίαυλο της μνήμης προκαλεί μείωση του Bandwidth και μία ανάλογη μείωση στο IPC. Από αυτό καταλαβαίνουμε ότι το Bandwidth είναι ένας σημαντικός παράγοντας που επηρεάζει την απόδοση των διεργασιών.

2.3 Κίνητρο

Με την άφιξη των CMPs οι περισσότερες τεχνικές υλικού και λογισμικού που είχαν υλοποιηθεί στις μονοπύρηνες αρχιτεκτονικές ή στα SMPs εφαρμόσθηκαν χωρίς μετατροπές. Οι χρονοδρομολογητές θεωρούν ότι οι πυρήνες είναι ανεξάρτητοι και απομονωμένοι. Αυτό, όμως, δεν ισχύει, καθώς μοιράζονται στοιχεία για τα οποία υπάρχει διαμάχη μεταξύ των εφαρμογών που τρέχουν μαζί. Η αντιμέτωπιση αυτή όμως δημιουργεί πολλά προβλήματα στους στόχους που ένας χρονοδρομολογητής προσπαθεί να ικανοποιήσει.

Τρέχοντας 4 εφαργμογές στο σύστημα που αναφέραμε προηγουμένως για 5 λεπτά με τον Linux scheduler (Figure 1.1-3) παρατηρούμε πως ο ρόλος του χρονοδρομολογητή

υποβαθμίζεται.

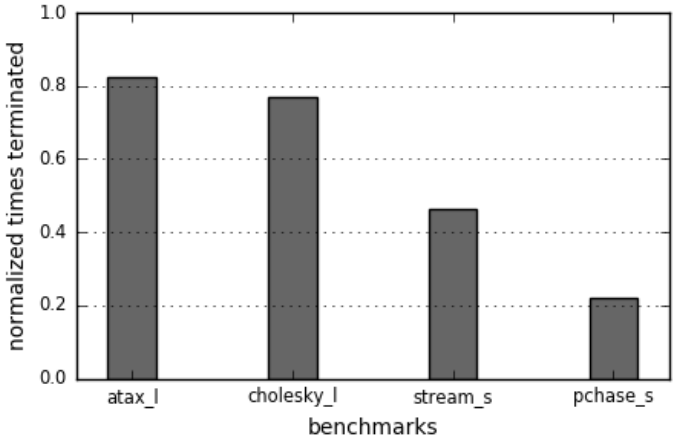


Figure 1.1-3: Φορές τερματισμού 4 διεργασιών κανονικοποιημένες στην περίπτωση που εκτελούνται μόνες τους

Αρχικά είναι προφανές ότι ο πρωταρχικός σκοπός του να προσφέρει ένα περιβάλλον, στο οποίο όλες οι διεργασίες κάνουν πρόοδο δεν μπορεί να επιτευχθεί, καθώς υπάρχουν εφαρμογές σαν την pchase_s οι οποίες εκτελούνται με πολύ χαμηλή απόδοση.

Το γεγονός αυτό σε ευρύτερη κλίμακα έχει μεγάλη επίπτωση στον αριθμό των διεργασιών που το σύστημα μπορεί να εκτελέσει ανά μονάδα χρόνου (throughput). Σε χρονικό διάστημα 5 λεπτών οι διεργασίες stream_s και pchase_s έχουν εκτελεστεί 50% και 80% λιγότερες φορές.

Επίσης είναι αδύνατο να εφαρμοσθεί μία πολιτική με προτεραίοτητες. Δίνοντας σε μία διεργασία υψηλότερη προτεραιότητα σημαίνει ότι επιθυμούμε αυτή να κάνει μεγαλύτερη πρόοδο. Αυτό στο παράδειγμά μας δεν μπορεί να επιτευχθεί βλέποντας την πολύ χαμηλή απόδοση της pchase_s.

Ένας απο τους κυριότερους ρόλους του χρονοδρομολογητή είναι να κατανέμει δίκαια τους πόρους μεταξύ των διεργασιών. Αυτό πλέον δεν μπορεί να γίνει εφικτό, καθώς βλέπουμε ότι μερικές διεργασίες έχουν υψηλή και άλλες πολύ χαμηλή απόδοση. Είναι εμφανές ότι οι επιπτώσεις της σύγκρουσης δεν μπορούν να μοιραστούν ομοιόμορφα στις εφαρμογές και συνεπώς δικαιοσύνη δεν μπορεί να επέλθει.

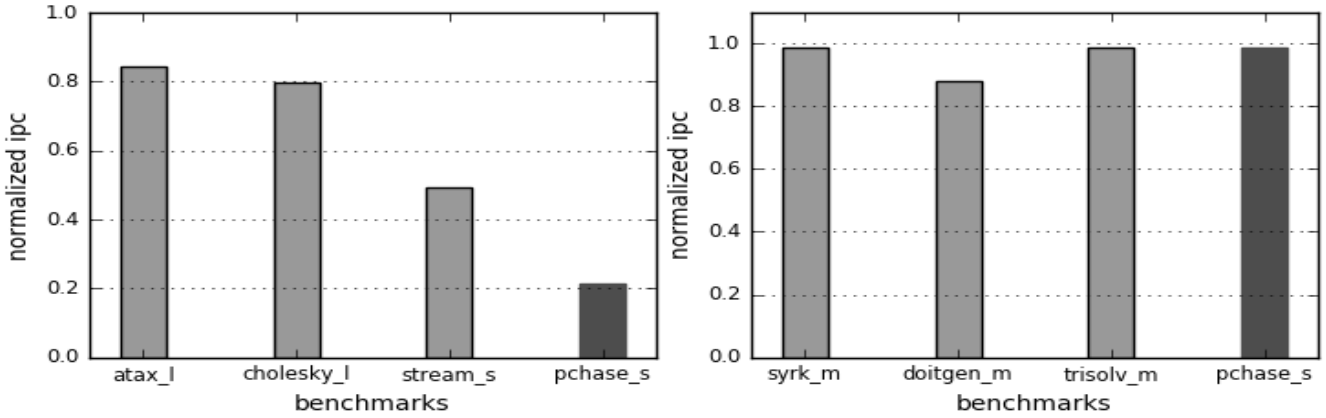


Figure 1.1-4: Μεταβλητή απόδοση της εφαρμογής pchase_s

Τέλος παρατηρώντας το σχήμα 1.1-4 καταλήγουμε στο εξής συμπέρασμα. Η απόδοση μιας εφαρμογής εξαρτάται απο τις εφαρμογές που θα τρέξουν στους γειτουνικούς πυρήνες. Στη μία τετράδα η απόδοσή της είναι πολύ χαμηλή ενώ στην άλλη πολύ υψηλή. Αυτό σημαίνει ότι η απόδοση του συστήματος είναι ασταθής και απρόβλεπτη και ο χρονοδρομολογητής δεν μπορεί να εγγυηθεί αξιοπιστία στην εκτέλεση των διεργασιών.

Κεφάλαιο 3
Προτεινόμενες τεχνικές χρονοδρομολόγησης

Σε αυτό το κεφάλαιο παρουσιάζουμε τρεις τεχνικές χρονοδρομολόγησης, βάσει των οποίων επιδιώκουμε να ξεπεράσουμε τα προβλήματα σταθερότητας και αξιοπιστίας που δημιουργούν οι χρονοδρομολογητές που δεν λαμβάνουν υπ'όψιν τους την σύγκρουση στους μοιραζόμενου πόρους.

Στην πρώτη προσέγγισή μας, σχεδιάζουμε έναν χρονοδρομολογητή, ο οποίος προσπαθεί να αποφύγει τη σύγκρουση των διεργασιών. Το επιτυγχάνει χωρίζοντας τις διεργασίες σε κατηγορίες και εξετάζοντας πως αυτές οι κατηγορίες αλληλεπιδρούν μεταξύ τους όταν τρέχουν παράλληλα. Εφ'όσον υπάρχει μία πρόβλεψη για το πως αυτές οι κατηγορίες συγκρούονται, είναι έτοιμος να πάρει αποφάσεις για το πως θα τις διαχωρίσει με σκοπό να πετύχει δίκαιη κατανομή των πόρων.

Στην δεύτερη προσέγγιση, διαχειριζόμαστε εκ των υστέρων τα αποτελέσματα της σύγκρουσης των διεργασιών. Ευνοούμε τις διεργασίες που έχουν πληγεί περισσότερο, δίνοντας τους την ευκαιρία να τρέξουν για περισσότερο χρόνο και να αυξήσουν την πρόοδο τους σε βάρος αυτών που τρέχουν με καλύτερη απόδοση. Με αυτό τον τρόπο θέλουμε να εξισορροπήσουμε την απόδοση τους.

Η τρίτη προσέγγιση είναι επέκταση της δεύτερης. Ξέροντας ότι μερικά ζευγάρια όταν τρέξουν μαζί, προκαλούν καταστροφικές συνέπειες. Για αυτό το λόγο αποφεύγουμε την εκτέλεσή τους και εξακολουθούμε να εφαρμόζουμε την πολιτική της δεύτερης τεχνικής, αποφεύγοντας έτσι ένα μέρος της σύγκρουσης και διαχειρίζοντας το υπόλοιπο. Με αυτήν την τεχνική θέλουμε να πετύχουμε μεγαλύτερη βελτίωση στη πρόοδο των διεργασιών και συνεπώς στο throughput.

3.1 Πρώτη Προσέγγιση (Αποφεύγοντας τη σύγκρουση)

Κύριο χαρακτηριστικό αυτής της τεχνικής είναι η εύρεση του κατάλληλου συνδιασμού διεργασιών η οποία οδηγεί στην επιθυμητή απόδοση. Όπως παρατηρήσαμε προηγουμένως, οι διεργασίες δεν αποδίδουν το ίδιο όταν εκτελούνται παράλληλα με άλλες στους γειτονικούς πυρήνες. Μερικοί συνδιασμοί συγκρούονται σε πολύ χαμηλό βαθμό και αποδίδουν αρκετά καλά, ενώ άλλοι σε έντονο και επηρεάζονται καταστροφικά. Όμως δεν είναι εφικτό να δοκιμάσουμε δυναμικά όλους τους συνδιασμούς ώστε να καταλήξουμε στον πιο αποδοτικό. Για αυτό το λόγο η χρονοδρομολόγησή μας γίνεται σε τρία επίπεδα. Στο πρώτο *ταξινομούμε* τις εφαρμογές σε κατηγορίες. Στο δεύτερο βρίσκουμε ένα μοντέλο που *προβλέπει* πως αλληλεπιδρούν οι κατηγορίες αυτές μεταξύ τους. Τέλος παίρνουμε την *απόφαση* να

τοποθετήσουμε τις εφαρμογές στους πυρήνες με τέτοιο τρόπο ώστε να αποφύγουμε τη σύγκρουσή τους και να πετύχουμε τα επιθυμητά αποτελέσματα. Πριν αναλύσουμε τα στάδια, θα αναφέρουμε την πλατφόρμα στην οποία θα εκτελέσουμε τα πειράματά μας και τις δυνατότητες που μας προσφέρει.

3.1.1 Πλατφόρμα Εκτέλεσης

Στην εργασία αυτή, για την αξιολόγηση των χρονοδρομολογητών μας, χρησιμοποιήσαμε τον επεξεργαστή Intel® Xeon® Processor X5560 Nehalem αρχιτεκτονικής. Αποτελείται απο δύο τσιπ (sockets) τα οποία αναφέρονται σε ξεχωριστό μέρος μνήμης (NUMA) και αποτελούνται απο τα εξής.

Τεσσερις πυρήνες 2,8 GHz συχνότητας, L1 ιδιωτική κρυφή μνήμη, που χωρίζεται σε 32 KB δεδομένων και 32 KB εντολών, L2 ιδιωτική κρυφή μνήμη 256 KB και L3 κοινή κρυφή μνήμη 8 MB. Κάθε τσιπ επικοινωνεί με την μνήμη μέσω τριών 8-byte καναλιών που λειτουργούν στο 1,333 GT/s. Δηλαδή το μέγιστο θεωρητικό Bandwidth μπορεί να φτάσει την τιμή 31,992 GB/s. Η επικοινωνία μεταξύ των 2 τσιπ επιτυγχάνεται μέσω του QPI συνδέσμου που φτάνει Bandwidth 12,8 GB/s. Επίσης κάθε τσιπ υποστηρίζει Hardware Prefetching Logic, για να φέρνει δεδομένα που τα νήματα θα χρησιμοποιήσουν στο κοντινό μέλλον, Simultaneous Multi-Threading" (SMT), επιτρέποντας παραπάνω νήματα να εκτελούνται στον ίδιο πυρήνα και Turbo Boost Technology για εξοικονόμηση ενέργειας.

Για τα πειράματα μας χρησιμοποιήσαμε μόνο το ένα τσιπ με ενεργοποιημένη τη λειτουργία του hardware prefetching logic και απενεργοποιημένες τις SMT και TBT.

Η αρχιτεκτονική διαδραματίζει πολύ σημαντικό ρόλο στην τοποθέτηση των διεργασιών στους πυρήνες και καθορίζει τις διαφορετικές ομάδες που μπορούν να δημιουργηθούν συνδιάζοντας τις διεργασίες. Παίρνοντας για παράδειγμα 4 εφαρμογές Α, Β, Γ, Δ, στην δική μας αρχιτεκτονική παράγεται ένας μοναδικός συνδιασμός εκτέλεσης τους στους πυρήνες. Αν είχαμε μια αρχιτεκτονική που οι πυρήνες ανα 2 μοιραζόντουσαν μια LLC, τότε τα πράγματα θα ήταν διαφορετικά. Σε αυτή την περίπτωση οι πιθανές τετράδες εκτέλεσης αυξάνονται στις 3. Παρατηρούμε λοιπόν ότι η δυνατότητα αποφυγής της σύγκρουσης αυξάνεται καθώς μειώνεται ο αριθμός των πυρήνων που μοιράζονται τη LLC. Αυτό συμβαίνει, επειδή μας δίνεται επιπλέον η δυνατότητα να μοιράσουμε τις διεργασίες στο χώρο (space-sharing). Στη δική μας περίπτωση, ο μόνος τρόπος να μοιράσουμε τις εφαρμογές είναι χρονικά (time-sharing) και αυτό περιορίζει τις επιλογές μας για αποφυγή της σύγκρουσης.

3.1.2 Μέθοδος Ταξινόμησης

Ξεκινάμε με το πρώτο στάδιο της ταξινόμησης. Βασισμένοι στο σχήμα που προτάθηκε απο τη δουλειά του Haritatos et al [1] κατηγοριοποιούμε τις εφαρμογές σε τέσσερις κλάσεις, την L, LC, C και N.

Στην L κατηγορία ανήκουν εκείνες, οι οποίες παρουσιάζουν μία σημαντική και σταθερή ροή δεδομένων από την μνήμη έως το πυρήνα. Αυτές οι εφαρμογές δεν επωφελούνται απο τη LLC και μολύνουν με μεγάλο αριθμό δεδομένων το δίαυλο μνήμης κάνοντας χρήση μεγάλου ποσοστού του Memory Bandwidth.

Στην LC κατηγορία ταξινομούνται εφαρμογές, οι οποίες χρησιμοποιούν σε μέτριο βαθμό το δίαυλο της μνήμης και επωφελούνται ελαφρά απο τη LLC, επαναχρησιμοποιώντας την σε μικρό βαθμό.

Στην C κλάση κατατάσσονται οι εφαρμογές με τα εξής χαρακτηριστικά. Χρησιμοποιούν σε πολύ μικρό βαθμό το δίαυλο μνήμης, καταναλώνοντας μικρό ποσοστό του Bandwidth, αλλά παρουσιάζουν μεγάλη εξάρτηση απο τη LLC. Αυτό σημαίνει ότι φέρνουν δεδομένα σε αυτή και πραγματοποιούν αναφορές σε αυτά για αρκετό χρονικό διάστημα πριν φέρουν τα επόμενα. Με αυτό τον τρόπο περιορίζονται μόνο σε αυτό το μοιραζόμενο πόρο, πραγματοποιώντας πολύ μεγάλη επαναχρησιμοποίηση.

Τέλος η N κατηγορία αποτελείται απο εφαρμογές οι οποίες δεν δείχνουν σημαντική δραστηριότητα στους μοιραζόμενους πόρους, αλλά περιορίζονται στις ιδιωτικές κρυφές μνήμες και στον επεξεργαστή. Οι κατηγορίες παρουσιάζονται στο σχήμα 1.1-5.

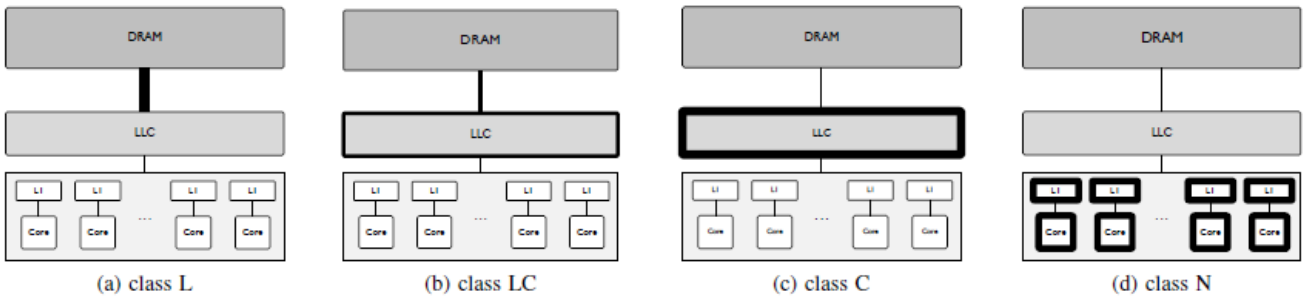


Figure 1.1-5: Δραστηριότητα στις 4 κατηγορίες

Για να ταξινομήσουμε τις εφαρμογές στις παραπάνω κατηγορίες, είναι απαραίτητο να καταγράψουμε τη συμπεριφορά τους. Αυτό το επιτυγχάνουμε χρησιμοποιώντας κάποιες απλές μετρικές, τις οποίες μπορούμε να αποκτήσουμε κατά το χρόνο εκτέλεσης απο τους hardware performance counters που παρέχουν οι σύγχρονοι επεξεργαστές.

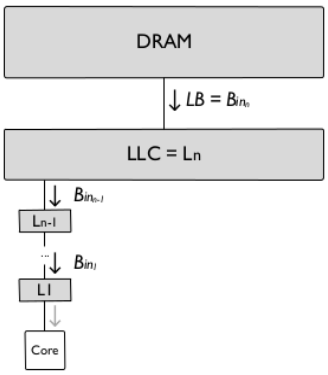


Figure 1.1-6: Ροή δεδομένων

Για κάθε εφαρμογή είναι απαραίτητο να μετρήσουμε τη ροή των δεδομένων απο τή κύρια μνήμη έως τον επεξεργαστή, χρησιμοποιώντας το Bandwidth σαν μετρική και να βρούμε σε ποιό μέρος της ιεραρχίας της μνήμης υπάρχει μεγάλη δραστηριότητα, χρησιμοποιώντας το

$CR_i = \frac{Bin_{i-1}}{Bin_i}$ (σχήμα 1.1-6).

Αν το LB (Memory Bandwidth) είναι μεγάλο και το $CR_n = 1$ τότε η εφαρμογή είναι L. Αν η δραστηριότητα μεταξύ μνήμης και LLC είναι μέτρια και $CR_n > 1$, τότε είναι LC. Διαφορετικά εφαρμογές με πολύ χαμηλο LB δραστηριοποιούνται στο υπόλοιπο τμήμα της ιεραρχίας.
Αν υπάρχει μεγάλη επαναχρησιμοποίηση της LLC αναφερόμαστε σε C αλλιώς σε N.

Η μέθοδος ταξινόμησης μας παρουσιάζεται στο δέντρο του σχήματος 1.1-7. Σε κάθε πλατφόρμα πρέπει να θέσουμε 5 όρια, α, β, γ, δ και ε. Σε αυτή τη πλατφόρμα έχουμε $\alpha = 0.12 \times B_{max}$, $\beta = 0.045 \times B_{max}$, $\gamma = 0.068 \times B_{max}$, $\delta = 0.25$, $\varepsilon = 0.25 \times IPC_{max}$. Η μέγιστη τιμή του Memory Bandwidth όπως μετρήθηκε απο τη stream εφαρμογή [2] είναι $B_{max} = 13.20\, GB/sec$ και $IPC_{max} = 4$

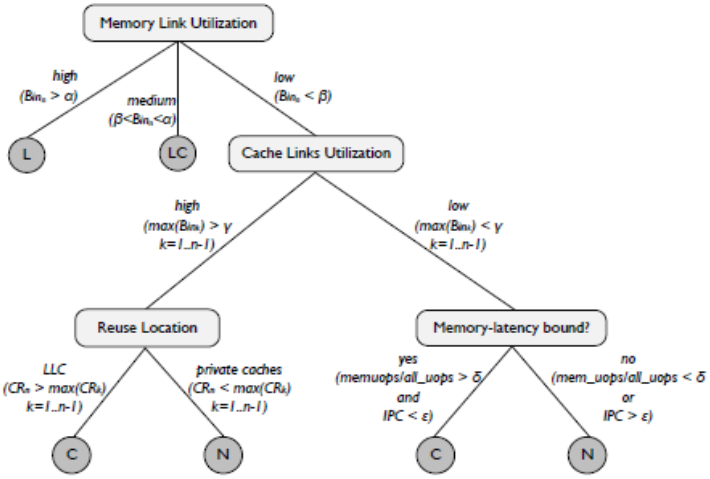


Figure 1.1-7: Δέντρο ταξινόμησης

Διαχωρισμός της C κατηγορίας

Οι εφαρμογές μπορεί να παρουσιάσουν διαφορετικά χαρακτηριστικά ανάλογα το μέγεθος του data set τους. Εξετάζοντας δύο εφαρμογές με διάφορα μεγέθη για το data set τους παρατηρούμε τα εξής .

Αρχικά βλέπουμε ότι όταν το σύνολο των δεδομένων είναι αρκετά μικρό ώστε να ικανοποιείται απο τα χαμηλότερα τμήματα της μνήμης (ιδιωτικές κρυφές μνήμες) η δραστηριότητα περιορίζεται σε αυτό το κομμάτι και οι εφαρμογές παρουσιάζουν συμπεριφορά Ν κλάσης.
Αυξάνουμε το data set και παρατηρούμε ότι αυτό μπορεί να ικανοποιηθεί πλέον απο τη LLC, οπότε και οι εφαρμογές γίνονται C.

Αξίζει να σημειώσουμε, ότι η μετάβαση απο τη C στη LC κατηγορία πραγματοποιείται σε διαφορετικό data set για κάθε εφαρμογή. Στην pchase γίνεται στο σημείο στο οποίο αρχίζουν να μην χωράνε τα δεδομένα στη LLC, ενώ στην mvt πολύ αργότερα. Αυτό σημαίνει

ότι η πρώτη εφαρμογή όταν βρίσκεται στα όρια της LLC ευνοείται από όλο το χώρο που αυτή προσφέρει και όταν το data set ξεπερνάει το μέγεθος της αρχίζει να κάνει συχνή αναφορά στην κύρια μνήμη, δείχνοντας μηδενική ανοχή. Από την άλλη, η δεύτερη εφαρμογή δείχνει να μην επηρεάζεται απο τον περιορισμό και να εξακολουθεί να εξυπηρετείται απο τη LLC ακόμα και για 14 MB data set.

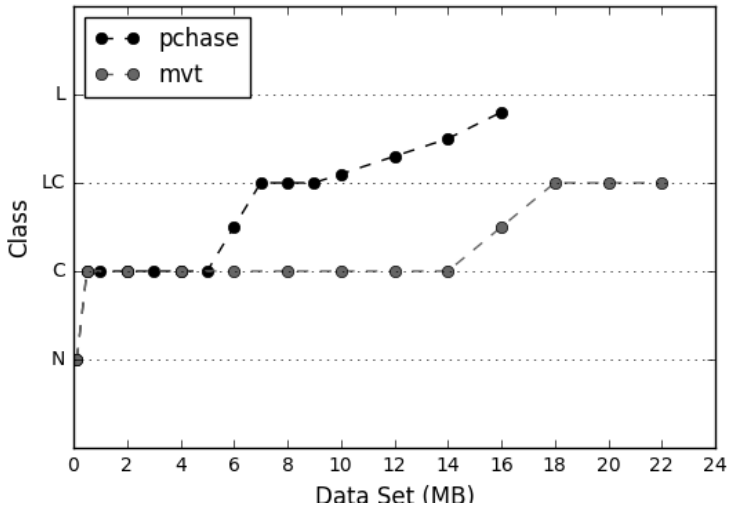


Figure 1.1-8: Συμπεριφορά δύο εφαρμογών για διάφορα data sets

Βασισμένοι στην προηγούμενη παρατήρησή μας, μπορούμε να συμπεράνουμε τα εξής. Εάν στη πλατφόρμα μας (8 MB LLC, 4 πυρήνες) τρέξουμε παράλληλα δύο εφαρμογές με 6 MB data set, που παρουσιάζουν ίδια συμπεριφορά με τη pchase, η LLC δεν μπορεί να εξυπηρετήσει και τις δύο. Θα μοιράσει το χώρο της και αναγκαστικά θα πάρουν ένα υποσύνολο του data set τους. Ο περιορισμός αυτός θα τους οδηγήσει σε αύξηση των misses, συχνότερη αναφορά στη μνήμη (αυξημένο Bandwidth) και μείωση στην απόδοση τους. Αν οι εφαρμογές είχαν την συμπεριφορά της mvt θα ήταν ανεκτικές στον περιορισμό και δεν θα παρουσίαζαν προβλήματα στην απόδοσή τους.

Κρίνεται αναγκαίο λοιπόν να διαχωρίσουμε περαιτέρω την C κατηγορία στις υποκατηγορίες SC και BC.

Στη SC ανήκουν εκείνες με working set μικρότερο των 2 MB, καθώς στην περίπτωση 4 πυρήνων μπορούν να συνυπάρχουν στη 8 MB LLC. Πιάνουν μικρό χώρο και δεν επηρεάζονται σημαντικά απο την παρεμβολή.

Στη BC κατηγορία κατατάσσονται εκείνες με working set μεγαλύτερο των 2 MB, οι οποίες χρειάζονται αρκετό χώρο της LLC για να τρέξουν αποδοτικά και όταν περιορίζονται παρουσιάζουν μεγάλες καθυστερήσεις στην εκτέλεσή τους.

Ο τρόπος διαχωρισμού της C κατηγορίας γίνεται στατικά. Για να πραγματοποιηθεί δυναμικά χρειάζονται πολύπλοκες τεχνικές (cache partitioning) που είναι εκτός του πεδίου μελέτης αυτής της εργασίας. Ενδεικτικά παρουριάζουμε στον παρακάτω πίνακα μερικές εφαρμογές (single-threaded and single execution phase) και σε ποιες κατηγορίες ταξινομούνται βάσει του δέντρου που περιγράψαμε.

Name	Source	DataSet (MB)	$B_{in3=LLC}(MB/s)$	CR3	CR2	IPC	Class
stream	[2]	366	6618.432	0.994	1.003	0.721	L
atax	polybench	72	4401.339	1.029	1.759	0.380	
gemver	polybench	125	1173.694	4.299	1.315	0.402	LC
mvt	polybench	125	996.640	5.597	1.171	0.279	
pchase	[3]	6	491.220	9.920	1.000	0.081	BC
stream	[2]	5	99.691	51.732	1.000	0.921	
correlation	polybench	2	0.706	1275.893	15.874	1.334	SC
covariance	polybench	2	0.714	1255.327	15.960	1.332	
3mm	polybench	0.064	0.760	1.983	72.710	2.250	N
bicg	polybench	0.064	0.670	2.194	504.134	1.180	

Table 1.1-1: Ταξινόμηση εφαρμογών στις κατηγορίες

3.1.3 Μοντέλο Πρόβλεψης

Περνάμε στο επόμενο στάδιο μετά την ταξινόμηση, την πρόβλεψη της σύγκρουσης μεταξύ των κατηγοριών. Έχοντας μία καλή εκτίμηση για το πως οι κατηγορίες αλληλεπιδρούν μεταξύ τους όταν τρέχουν παράλληλα, μπορούμε να πάρουμε μία έγκυρη απόφαση για το πως θα τοποθετήσουμε τις εφαρμογές ώστε να πάρουμε το επιθυμητό αποτέλεσμα. Παραθέτουμε παρακάτω πως περιμένουμε να επηρεαστούν οι κατηγορίες, όταν τρέξουν παράλληλα με τις υπόλοιπες.

Η N κατηγορία περιορίζεται στα χαμηλότερα επίπεδα της ιεραρχίας της μνήμης (ιδιωτικές κρυφές μνήμες) και δεν αλληλεπιδρά με καμία άλλη κατηγορία.

Η L κατηγορία καταναλώνει μεγάλο μέρος του Memory Bandwidth και εξαρτάται απο αυτό. Όταν πολλές L εφαρμογές τρέξουν παράλληλα το συνολικό τους Bandwidth μπορεί να ξεπεράσει αυτό που το σύστημα μπορεί να ικανοποιήσει. Αυτή η σύγκρουση προκαλεί μείωση στην απόδοσή τους που είναι ανάλογη της μείωσης του Bandwidth. Άρα αυτή η κατηγορία επηρεάζεται από τον εαυτό της.

Για την LC κλάση περιμένουμε μικρές καθυστερήσεις όταν εκτελούνται μαζί με τις L. Αυτό μπορεί να συμβεί λόγω της διαμάχης στο Memory Bandwidth και της μόλυνσης της LLC με δεδομένα των L.

Όσον αφορά την SC ομάδα, οι εφαρμογές της επωφελούνται απο τη LLC καθώς φέρνουν σε αυτή ένα μικρό όγκο δεδομένων και κάνουν έντονη επαναχρησιμοποίηση σε αυτά. Περιμένουμε να επηρεάζονται σε ικανοποιητικό βαθμό απο την L κατηγορία, διότι οι εφαρμογές της μολύνουν διαρκώς με δεδομένα τη LLC προκαλώντας καταστροφική παρεμβολή.

Τέλος εξετάζουμε τη BC κλάση, η οποία είναι η πιο ευαίσθητη απο όλες. Λόγω του ότι χρειάζεται το μεγαλύτερο μέρος της LLC για να τρέξει με καλή απόδοση, οποιαδήποτε παρεμβολή σε αυτόν τον πόρο θα προκαλέσει πολύ σοβαρή ζημιά. Επομένως προβλέπεται ότι η L και η LC κατηγορία, οι οποίες μολύνουν με μεγάλη συχνότητα την LLC, να επηρεάζουν την ομαλή εκτέλεση της BC. Επίσης οι εφαρμογές της ίδιας κατηγορίας περιμένουμε να προκαλέσουν μεγάλη διαμάχη στη LLC.

	L	LC	SC	BC	N
L	1.3014	1.1273	1.6744	2.4278	1.0821
LC	1.1184	1.0714	1.1192	2.2216	1.0549
SC	1.0564	1.0436	1.0783	1.4609	1.0384
BC	1.1438	1.0761	1.1608	2.3253	1.0093
N	1.0189	1.0063	1.0008	1.0644	1.0017

Figure 1.1-9: Μέση καθυστέρηση των εφαρμογών όταν εκτελούνται παράλληλα σε επίπεδο κλάσεων

Στο σχήμα 1.1-9 παρουσιάζουμε τη μέση καθυστέρηση που υποφέρει κάθε κατηγορία όταν εκτελείται με μία άλλη. Στον άξονα χ παρουσιάζουμε την καθυστέρηση που επιβάλουν οι κλάσεις, ενώ στον y την καθυστέρηση που υποφέρουν.

Τα αποτελέσματα δείχνουν ότι οι προβλέψεις μας είναι έγκυρες. Η πιο ευάλωτη κλάση είναι η BC, που επηρεάζεται από τις L, LC και BC. Η L και SC βλέπουμε ότι επηρεάζονται σε μικρότερο βαθμό απο την L. Η N και η LC δεν φαίνονται να παρουσιάζουν καθυστερήσεις, όταν τρέχουν με τις άλλες κατηγορίες.

3.1.4 Απόφαση

Στο τελευταίο αυτό στάδιο εφαρμόζουμε την πολιτική μας, αποφασίζοντας με ποιο τρόπο θα συνδιάσουμε τις εφαρμογές, ώστε να αποφύγουμε τη σύγκρουση για τους μοιραζόμενους πόρους. Με αυτό το τρόπο επιδιώκουμε να προσφέρουμε ένα περιβάλλον, στο οποίο οι διεργασίες θα εκτελούνται με υψηλή απόδοση (κοντά στην απόδοση που βιώνουν όταν εκτελούνται μόνες τους) και κατ'επέκταση η κατανομή των πόρων θα γινεται δίκαια.

Ο χρονοδρομολογητής μας λαμβάνει υπ'όψιν του τη διαμάχη για τη LLC και το Memory Bandwidth και ονομάζεται CMB (Cache and Memory Bandwidth contention-aware scheduler). Υπενθυμίζουμε ότι επειδή δουλεύουμε στο ένα τσιπ, στο οποίο 4 πυρήνες μοιράζονται μια LLC, έχουμε τη δυνατότητα να χωρίσουμε τις εφαρμογές στο χρόνο μόνο (time-sharing). Έτσι συνδιάζουμε τις εφαρμογές σε τετράδες που εκτελούνται σε διαφορετικά κβάντα χρόνου, με σκοπό να αποφύγουμε τη μεταξύ τους σύγκρουση.

Βασισμένοι στο μοντέλο πρόβλεψης και στα αποτέλεσμα της συνεκτέλεσης των κατηγοριών, δημιουργούμε μία πολιτική με προτεραιότητες. Η πρώτη μας προτεραιότητα είναι να απομονώσουμε την BC κατηγορία που είναι πιο ευάλωτη. Επίσης θέλουμε να προστατέψουμε τη SC κατηγορία και τέλος να μετριάσουμε τη διαμάχη των L για το Memory Bandwidth. Για αυτό το λόγο δημιουργούμε τρεις πιθανές τετράδες.

Στην μία (BC) επιτρέπεται να είναι μαζί BC με SC με N εφαρμογές, καθώς βλέπουμε απο τον παραπάνω πίνακα ότι σε αυτήν την περίπτωση δεν επιβάλλονται μεγάλες

καθυστερήσεις. Το ιδανικό θα ήταν αυτή η τετράδα απο μόνο μία BC εφαρμογή καθώς όπως ξέρουμε η συνεκτέλεση τους προκαλεί καταστροφική παρεμβολή, αλλά αυτό είναι εφικτό όταν ο αριθμός των BC είναι μικρός και μπορούν να απομονωθούν πλήρως στις τετράδες.

Στην δεύτερη τετράδα (SC) μπορούν να βρίσκονται πολλές SC μαζί με N εφαρμογές. Με αυτό τον τρόπο επιτυγχάνουμε αποφυγή σύγκρουσης SC με L.

Η τρίτη τετράδα (L) αποτελείται απο L, LC ή N εφαρμογές.

Τα βήματα που ακολουθεί ο χρονοδρομολογητής για τον σχηματισμό αυτών των τετράδων παρουσίαζονται στον παρακάτω πίνακα 1.1-2. Παράλληλα παραθέτουμε και το σκοπό που κάθε βήμα εξυπηρετεί.

Βήματα	*Στόχοι*
1) μεγιστοποίησε της BC τετράδες και άπλωσε τις BC εφαρμογές	Απέφυγε τη παρεμβολή των BC με τις BC και τις L
2)συγκέντρωσε τις SC εφαρμογές ώστε να ελαχιστοποιήσεις τις SC τετράδες	Απέφυγε τη παρεμβολή των SC με τις L
3) Ταξινόμησε τις L εφαρμογές ως προς το Bandwidth και μοίρασε τες στις τετράδες που απομένουν	Ισορρόπησε τη χρήση του Memory Bandwidth

Table 1.1-2: Βήματα και στόχοι του CMB χρονοδρομολογητή

Ο χρονοδρομολογητής μας υλοποιείται σε 5 συναρτήσεις, init(), qex(), thaw(), freeze(), schedule(). Η qex() καλείται μετά το τέλος του κβάντου χρόνου και καλεί με τη συγκεκριμένη σειρά τις freeze(), schedule() και thaw(). Στη freeze() σταματάμε την τετράδα που έτρεχε, στη schedule() επιλέγουμε την επόμενη τετράδα, στην περίπτωση μας οι τετράδες εκτελούνται κυκλικά, και στη thaw() ξεκινάμε την εκτέλεση της τετράδας που επιλέχθηκε προηγουμένως. Στην init() γίνεται ο σχηματισμός των τετράδων, όπως περιγράφηκε.

Οι τετράδες και οι κλάσεις υλοποιούνται σε λίστες. Οι εφαρμογές έχουν ταξινομηθεί offline. Ο αλγόριθμος έχει πολυπλοκότητα O(n + L log L), O(n) για τον διαχωρισμό των εφαρμογών στις διαφορετικές ομάδες και O(L log L) για την ταξινόμηση των L εφαρμογών (code 1.1-1), και O(1) για την απόφαση. Μετά το σχηματισμό των τετράδων οι τετράδες εκτελούνται κυκλικά σε κάθε κβάντο χρόνου.

```
void create_gangs (gangs_nr, apps_l){
        list_t      gangs[gangs_nr];
        for_each_app_in_list(apps_l){
                remove_from_top (apps_l);
                add_to_tail (gangs[i++]);
        }
}
void form_gangs(apps){
        move_to_lists(apps);
        gangs_nr = (apps_nr / cores_nr);
        minimum_L = (L_nr / cores_nr);
        BC_gangs_nr = 0;
        SC_gangs_nr = 0;

        if (BC_list != []){
                maximum_BC = (gangs_nr – minimum_L);
                BC_gangs_nr = ((BC_nr < maximum_BC) ?
BC_nr : maximum_BC);
                create_gangs(BC_gangs_nr, [BC, N, SC]);
        }

        if (SC_list != []){
                SC_gangs_nr = (SC_nr / cores_nr);
                create_gangs(SC_gangs_nr, [SC, LC, N]);
        }

        remaining_gangs_nr = gangs_nr – (BC_gangs_nr +
SC_gangs_nr);
        if (L_list == []) create_gangs(remaining_gangs_nr, [LC, N]);
        else{
                quicksort(L_list);
                create_gangs(remaining_gangs_nr, [L, LC, N]);
        }
}
```

Code 1.1-1: Αλγοριθμος του CMB χρονοδρομολογητή

3.2 Δεύτερη Προσέγγιση (Διαχειρίζοντας τη σύγκρουση)

Σε αυτή τη τεχνική σκοπός μας είναι να διαχειριστούμε τα αποτελέσματα της σύγκρουσης των εφαρμογών. Όπως είδαμε σε προηγούμενο κεφάλαιο οι διεργασίες δεν επηρεάζονται το ίδιο απο τη μεταξύ τους και η επίπτωση στην πρόοδο τους είναι διαφορετική. Αυτό έχει ως αποτέλεσμα την άνιση κατανομή των πόρων σε αυτές. Τρέχουμε 12 εφαρμογές με το Linux scheduler για 320 δευτερόλεπτα και παρατηρούμε ότι δίνει ίδια ευκαιρία στις εφαρμογές να εκτελεστούν, αλλά δεν κατανέμει την πρόοδο τους δίκαια. Η πρόοδος υπολογίζεται ως το γινόμενο των φορών που εκτελέστηκε η κάθε εφαρμογή επί το χρόνο που χρειάζεται για να εκτελεστεί όταν τρέχει μόνη της.

Αυτό που επιδιώκουμε εμείς είναι οι διεργασίες να κάνουν την ίδια πρόοδο στη χρονική περίοδο που τους δίνεται να τρέξουν. Λαμβάνοντας υπ'όψιν ότι η πρόοδος τους καθορίζεται απο το κλάσμα $IPC_{co-running}/IPC_{alone}$ και από το χρόνο που τρέχουν στον πυρήνα, μια

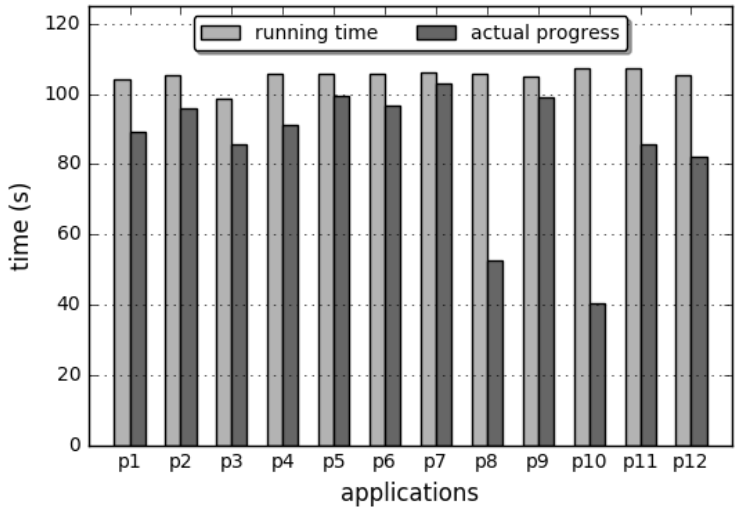


Figure 1.1-10: Δικαιη κατανομή χρόνου αλλά όχι προόδου

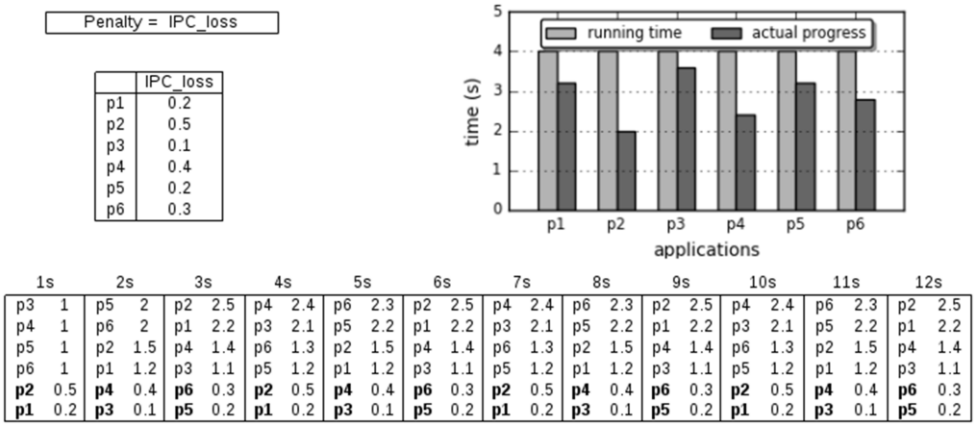
εφαρμογή Α με χαμηλότερο IPC από αυτό μιάς άλλης Β πρέπει να τρέξει περισσότερο χρόνο απο την Β ώστε οι πραγματικοί χρόνοι που αυτές έτρεξαν να έρθουν στα ίδια επίπεδα.

Οδηγούμενοι απο αυτήν την παρατήρηση, σχεδιάζουμε έναν χρονοδρομολογητή, ο οποίος λαμβάνει υπ'όψιν του την μείωση του IPC που κάθε διεργασία έχει υποστεί μετά το τέλος της εκτέλεσής της και τις ευνοεί αναλόγως. Επιδιώκει να αυξήσει το χρόνο εκτέλεσης αυτών που έχουν μεγάλη μείωση στο IPC, με σκοπό να αυξήσει την πραγματική τους πρόοδο, μειώνοντας το χρόνο και συνεπώς πρόοδο εκείνων που τρέχουν με μεγαλύτερο IPC. Με αυτόν τον τρόπο μπορεί να κατανέμουμε άνισα το χρόνο του επεξεργαστή ανάμεσα στις διεργασίες, διαβεβαιώνουμε όμως ότι όλες θα κάνουν την ίδια πρόοδο.

Στην υλοποίησή μας χρησιμοποιούμε μία ουρά διεργασιών για όλους τους πυρήνες. Μετά από κάθε κβάντο χρόνου η ουρά αυτή ταξινομείται σε φθίνουσα σειρά με βάση ένα κριτήριο, που έχουμε επιλέξει και θα εξηγήσουμε παρακάτω, και οι τέσσερις πρώτες διεργασίες επιλέγονται για να τρέξουν στους πυρήνες.

Η βασική ιδέα είναι να ευνοούμε τις διεργασίες ανάλογα με το πόσο IPC έχασαν την τελευταία φορά που ήταν στον πυρήνα. Αυτό το επιτυγχάνουμε προωθώντας αυτές τις διεργασίες ψηλότερα στην ουρά, έτσι ώστε να έχουν την ευκαιρία να εκτελεστούν συχνότερα, αυξάνοντας την πρόοδο τους. Για αυτό το λόγο ταξινομούμε τις διεργασίες με βάση το παρακάτω κριτήριο $criterion = waiting\ time + factor * IPC\_loss$.Στην περίπτωση που είχαμε μόνο το waiting time οι διεργασίες θα επιλέγονταν κυκλικά και θα είχαν ίση κατανομή χρόνου. Τώρα όμως αυτές που μόλις έτρεξαν με μεγάλο IPC_loss μπορούν να ξεπεράσουν μερικές άλλες στη ουρά και να τοποθετηθούν στον επεξεργαστή νωρίτερα, καθυστερώντας εκείνες που έτρεξαν με καλύτερο IPC. Το IPC_loss υπολογίζεται πολύ εύκολα, μετά το τέλος του κβάντου χρόνου για τις διεργασίες που ήταν στον πυρήνα, από τη σχέση $\text{IPC\_loss} = \left(1 - \frac{\text{IPC}_{\text{co-running}}}{\text{IPC}_{\text{alone}}}\right)$. Το IPC_{alone} το υπολογίζουμε offline για κάθε διεργασία, ενώ το $IPC_{co-running}$ το αποκτούμε μετά από κάθε κβάντο χρόνου από τους hardware performance

counters που παρέχει ο επεξεργαστής μας.



	IPC_loss
p1	0.2
p2	0.5
p3	0.1
p4	0.4
p5	0.2
p6	0.3

1s		2s		3s		4s		5s		6s		7s		8s		9s		10s		11s		12s	
p3	1	p5	2	p2	2.5	p4	2.4	p6	2.3	p2	2.5	p4	2.4	p6	2.3	p2	2.5	p4	2.4	p6	2.3	p2	2.5
p4	1	p6	2	p1	2.2	p3	2.1	p5	2.2	p1	2.2	p3	2.1	p5	2.2	p1	2.2	p3	2.1	p5	2.2	p1	2.2
p5	1	p2	1.5	p4	1.4	p6	1.3	p2	1.5	p4	1.4	p6	1.3	p2	1.5	p4	1.4	p6	1.3	p2	1.5	p4	1.4
p6	1	p1	1.2	p3	1.1	p5	1.2	p1	1.2	p3	1.1	p5	1.2	p1	1.2	p3	1.1	p5	1.2	p1	1.2	p3	1.1
p2	0.5	**p4**	0.4	**p6**	0.3	**p2**	0.5	**p4**	0.4	**p6**	0.3	**p2**	0.5	**p4**	0.4	**p6**	0.3	**p2**	0.5	**p4**	0.4	**p6**	0.3
p1	0.2	**p3**	0.1	**p5**	0.2	**p1**	0.2	**p3**	0.1	**p5**	0.2	**p1**	0.2	**p3**	0.1	**p5**	0.2	**p1**	0.2	**p3**	0.1	**p5**	0.2

Figure 1.1-11: Εκτέλεση 12 δευτερολεπτων χωρίς factor σε 2-πύρηνο σύστημα

Το IPC_loss από μόνο του δεν είναι ικανό να προωθήσει σωστά τις διεργασίες. Για αυτό το λόγο χρειάζεται να πολλαπλασιαστεί με τον παράγοντα factor. Αυτό γίνεται εμφανές στο παράδειγμα του σχήματος 1.1-11. Παρατηρούμε ότι καθώς το IPC_loss κυμαίνεται μεταξύ 0 και 1, είναι αδύνατο να ξεπεράσει τις διεργασίες που περιμένουν στην ουρά με waiting time 1 και 2 δευτερόλεπτα (κβάντο χρόνου = 1 δευτερόλεπτο).

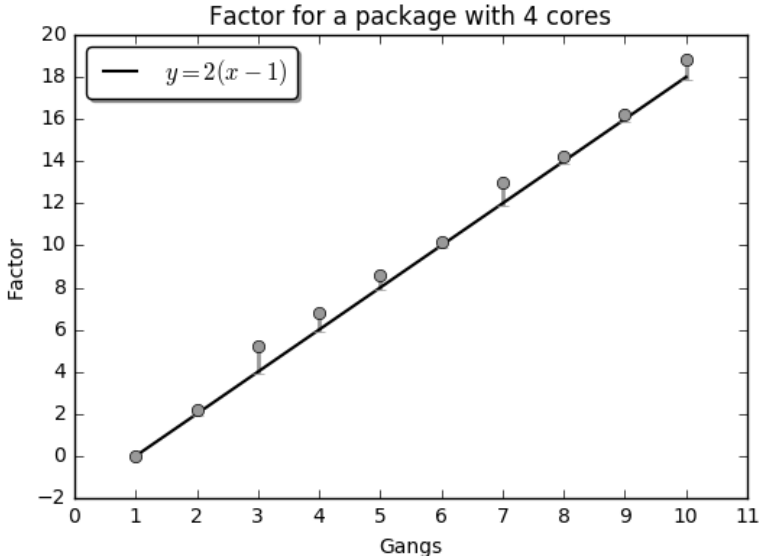


Figure 1.1-12: Σχέση μεταξύ παράγοντα και αριθμό τετράδων

Ο παράγοντας factor είναι ένας ακέραιος που εξαρτάται απο το κβάντο χρόνου και τις ομάδες των διεργασιών και καθορίζει τη σωστή προώθηση των διεργασιών στην ουρά. Στο παράδειγμά μας αν ο παράγοντας 4 ικανοποιούσε τις απαιτήσεις μας και αλλάζαμε τον φόρτο εργασίας προσθέτοντας μερικές διεργασίες τότε οι ομάδες που θα περίμεναν να τρέξουν στους πυρήνες θα αυξάνονταν. Αυτό σημαίνει ότι και το waiting time θα αυξανόταν συνολικά. Σε

αυτή την περίπτωση η προηγούμενη τιμή δεν θα ήταν αρκετή για να μας ανεβάσει τις διεργασίες στην ουρά. Το ίδιο ισχύει και για μια πιθανή αλλαγή του κβάντου χρόνου

Σκοπός μας είναι να βγάλουμε μία σχέση με την οποία θα καθορίσουμε τον factor μια για πάντα για οποιοδήποτε σύστημα. Με κβάντο χρόνο ίσο με 1 δευτερόλεπτο βρίσκουμε για διάφορες ομάδες διεργασιών τον παράγοντα που οδηγεί στην καλύτερη απόδοση, δηλαδή ίση κατανομή της προόδου. Επειδή είναι πολυ χρονοβόρο και σχεδόν ανέφικτο να κάνουμε δοκιμές για όλες τις πιθανές ομάδες σε πραγματικό σύστημα, προσωμοιόνουμε την πολιτική μας. Δημιουργούμε όλες τις πιθανές ομάδες για τα συστήματα δύο, τριών και τεσσάρων πυρήνων και για κάθε ομάδα ελέγχουμε επαναληπτικά ένα εύρος από παράγοντες. Κρατάμε εκείνον που οδηγεί στην μικρότερη τυπική απόκλιση της προόδου των διεργασιών. Ενδεικτικά παραθέτουμε τα αποτελέσματα τις προσομοίωσης για ένα 4-πύρηνο σύστημα για το διαφορετικό αριθμό τετράδων. Η ίδια εικόνα παρουσιάζεται και για τα υπόλοιπα συστήματα που εξετάσαμε (δύο, τριών πυρήνων).

Βλέπουμε λοιπόν ότι η σχέση μεταξύ παράγοντα και αριθμού ομάδων με κβάντο χρόνου ίσο με 1 δευτερόλεπτο είναι η εξής :

$$Factor = 2 \cdot (gangs\_nr - 1)$$
$$gangs\_nr = \frac{progs\_nr}{cores\_nr}$$

Είμαστε έτοιμοι λοιπόν να υλοποιήσουμε το χρονοδρομολογητή μας, σκοπός του οποίου είναι να προσφέρει δίκαιη κατανομή τής προόδου μεταξύ των διεργασιών (FOP-Fairness over Progress scheduler).

Όπως στην προηγούμενη προσέγγιση μας, ο χρονοδρομολογητής υλοποιείται σε 5 συναρτήσεις όπως φαίνεται στον κώδικα 1.1-2.

Στην init() ενεργοποιούμε τους performance counters, λαμβάνουμε το IPCalone για κάθε εφαρμογή, υπολογίζουμε το σύνολο των τετράδων που θα τρέξουν και εν συνεχεία τον παράγοντα. Η quantum_expired() καλείται μετά το τέλος του κβάντου και καλεί με τη σειρά τις freeze(), schedule() και thaw(). Στη freeze σταματάμε την τετράδα που ήταν στους πυρήνες, λαμβάνουμε το IPCco-running, βρίσκουμε το IPCloss και υπολογίζουμε το penalty. Ανανεώνουμε επίσης το waiting time και συνεπώς το criterion για όλες τις εφαρμογές. Στη schedule() ταξινομούμε τις εφαρμογές σε φθίνουσα σειρά με βάση το criterion και επιλέγουμε τις 4 πρώτες προς εκτέλεση. Στη thaw() εκτελούμε τις επιλεγμένες εφαρμογές. Η πολυπλοκότητα του FOP scheduler για την απόφαση είναι O(n log n) (πολυπλοκότητα της quicksort), η ουρά υλοποιείται σαν λίστα.

```
void init (){
        gangs_nr = progs_nr/cores_nr;
        factor = 2 (gangs_nr – 1);
        counters = perf_counters_init(selected_events);

        for_each_application_in_list(progs_all_list){
                application->criterion = 0;
                application->IPC_alone = get_IPC_parameter();
        }
}
void schedule(){
        quicksort(progs_all_list, criterion, DESCENDING);
        while (number(progs_schedule_list) < cores_nr){
                remove_from_top(progs_all_list);
                add_to_tail(progs_schedule_list);
        }
}
void thaw(){
        for_each_application_in_list(progs_schedule_list){
                start_running(application);
        }
        perf_counters_zero(counters);
        perf_counters_start(counters);
}
void freeze(){
        perf_counters_stop(counters);
        for_each_application_in_list(progs_schedule_list){
                stop_running(application);
                value = perf_counters_read(counters);
                IPC_co-running = get_IPC(value);
                IPC_loss = (1 – IPC_co-running/application->IPC_alone);
                application->waiting_time = 0;
                application->penalty = factor * IPC_loss;
                remove_from_top(progs_schedule_list);
                add_to_tail(progs_all_list);
        }
for_each_application_in_list(progs_all_list){
                update_waiting_time(application);
                application->criterion = waiting_time + penalty;
        }
}
void quantum_expired(){
        freeze();
        if (current_tics < RUN_TICS){
                schedule();
                thaw();
        }else{
                stop_execution();
                print_results();
        }
```

Code 1.1-2: Αλγόριθμος του FOP scheduler

3.2 Τρίτη Προσέγγιση (Αποφεύγοντας και διαχειρίζοντας τη σύγκρουση)

Σε αυτή τη προσέγγιση μας επεκτείνουμε τον προηγούμενο χρονοδρομολογητή έτσι ώστε να αποφεύγει τη σύγκρουση ορισμένων εφαρμογών. Από το μοντέλο πρόβλεψης της σύγκρουσης των διαφορετικών κατηγοριών βλέπουμε ότι η απόδοση των C (SC, BC) εφαρμογών μειώνεται σημαντικά όταν εκτελούνται μαζί με L (x1.67 καθυστέρηση για τη SC, x2.43 καθυστέρηση για τη BC). Θέλοντας να παρέχουμε ένα περιβάλλον, στο οποίο οι διεργασίες θα εκτελούνται πιο αποδοτικά, μετασχηματίζουμε την προηγούμενη τεχνική ώστε να αποφεύγει την εκτέλεση των L με τις C. Με αυτό τον τρόπο οι C έχουν τη δυνατότητα να κάνουν μεγαλύτερη πρόοδο. Σε ευρύτερη κλίμακα αυτό σημαίνει βελτιωμένο throughput σε σχέση με τον FOP scheduler. Τους ‘κακούς’ συνδιασμούς που δεν μπορούμε να αποφύγουμε τους διαχειριζόμαστε, εφαρμόζωντας την ίδια πολιτική με τον FOP.

Ο FOP-LCI (Fairness over Progress with L-C Isolation) scheduler μετά την ταξινόμηση της ουράς διαμοιράζει τις εφαρμογές όπως φαίνεται στο σχήμα 1.1-13. Όπως παρατηρούμε είναι πιθανό μερικές εφαρμογές που δεν ήταν η σειρά τους να τρέξουν, να τοποθετηθούν στον πυρήνα νωρίτερα, ‘κλέβοντας’ τη σειρά αυτών που έπρεπε να τρέξουν. Εφ’όσον έμειναν εκτός πυρήνα για άλλο ένα κβάντο χρόνου το waiting time τους θα αυξηθεί και επομένως το κριτήριο τους. Θα φτάσουν, λοιπόν, να είναι στην αρχή της ουράς και να εκτελεστούν με τις επιθυμητές εφαρμογές. Με αυτό τον τρόπο αποφεύγουμε τη λιμοκτονία.

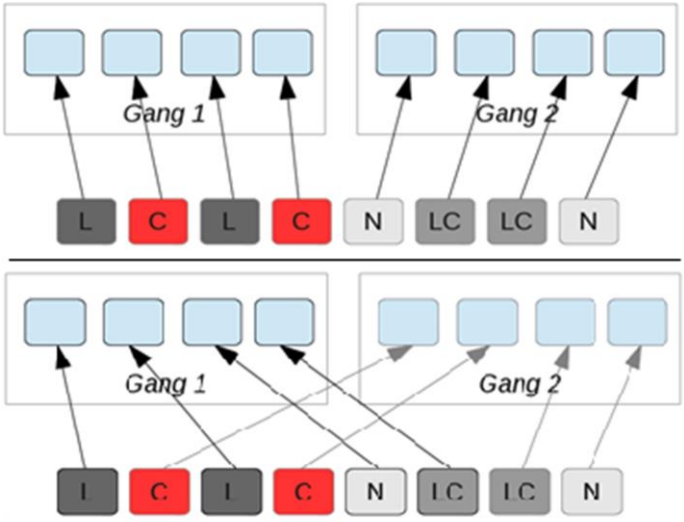


Figure 1.1-13: Επιλογή τετράδων με τον FOP(πάνω)
και με τον FOP-LCI (κάτω)

Ο FOP-LCI διαφοροποιείται σε σχέση με τον FOP μόνο στη συνάρτηση schedule(). Ο κώδικάς της φαίνεται δίπλα (Code 1.1-3). Οι εφαρμογές ταξινομούνται με βάση το σχήμα του πρώτου CMB scheduler. Η πολυπλοκότητα για την απόφαση είναι όπως και πριν O(n log n) (πολυπλοκότητα της quicksort).

```
void schedule()
        L_ON = 0;
        C_ON = 0;
        allowed = 1;
        quicksort(progs_all_list, criterion, DESCENDING);
        for_each_app_in_list(progs_all_list)
                switch (app->class)
                        case L_CLASS:
                                if (C_ON) allowed = 0;
                                else L_ON = 1;
                        case C_CLASS:
                                if (L_ON) allowed = 0;
                                else C_ON = 1;
        if (allowed )
                if (number(progs_schedule_list) != cores_nr)
                        remove_app_from_list(progs_all_list);
                        add_to_tail(progs_schedule_list);
```

Code 1.1-3: Schedule() συνάρτηση του FOP-LCI scheduler

Κεφάλαιο 4
Αξιολόγηση

Για την αξιολόγηση των πολιτικών μας δημιουργούμε 3 ομάδες πειραμάτων, με 5 φόρτους εργασιών σε κάθε ομάδα. Στην πρώτη ομάδα δημιουργούμε φόρτους εργασιών συνδιάζοντας όλες τις κατηγορίες εφαρμογών εκτός της BC. Η δεύτερη αποτελείται απο εφαρμογές όλων των κατηγοριών και από αριθμό BC εφαρμογών, οι οποίες μπορούν να απομονωθούν πλήρως σε διαφορετικές τετράδες. Στην τελευταία ομάδα ο αριθμός των BC εφαρμογών είναι τέτοιος και η πλήρης απομόνωση τους δεν είναι εφικτή. Κάθε φόρτος εργασιών αποτελείται από 16 εφαρμογές. Οι εφαρμογές έχουν ταξινομηθεί offline. Σε κάθε εκτέλεση επιθυμούμε ο φόρτος διεργασιών να είναι σταθερός, έτσι επανεκτελούμε κάθε εφαρμογή που τερματίζει. Η πλατφόρμα που χρησιμοποιήσαμε είναι αυτή που περιγράψαμε σε προηγούμενο κεφάλαιο και ασχοληθήκαμε με το ένα τσιπ, ώστε να δοκιμάσουμε τις τεχνικές μας σε περιβάλλον που μας δίνει τη δυνατότητα να μοιράσουμε τις εφαρμογές μόνο χρονικά.

Ενδιαφερόμαστε να μετρήσουμε το κατά πόσο δίκαια κατανέμονται οι πόροι (fairness) και πόσες διεργασίες εκτελούνται ανα μονάδα χρόνου (throughput). Γιαυτό το λόγο χρησιμοποιούμε δύο μετρικές, την τυπική απόκλιση (2) και τον μέσο (1). Τις μετρικές αυτές τις εφαρμόζουμε στο σύνολο των εφαρμογών, στο κλάσμα των φορών που τερματίστηκαν με την κάθε τεχνική προς τον ιδανικό αριθμό που θα τερματιζόντουσαν αν δεν υπήρχε σύγκρουση (3).

$$T(s) = average\big(N(s,i)\big), \quad for\ all\ applications\ i \quad (1)$$

$$F(s) = \sigma\big(N(s,i)\big), \qquad for\ all\ applications\ i \quad (2)$$

$$N(s,i) = \frac{times\_terminated\,(s,i)}{ideal\_times\_terminated\,(i)}, \qquad s = scheduler, i = application \quad (3)$$

$$ideal\_times\_terminated\,(i) = \frac{execution\_time}{time\,(i)_{alone-execution}}, \qquad i = application \quad (4)$$

$$execution\_time = total\_execution\_time \cdot \frac{cores\_nr}{apps\_nr} \quad (5)$$

Παραθέτουμε μία ενδεικτική μέτρηση για κάθε ομάδα πειραμάτων. Χρησιμοποιούμε boxplots για να δείξουμε την γενική εικόνα μεταξύ των τεχνικών μας και barcharts για να δείξουμε πιο αναλυτικά την απόδοση κάθε εφαρμογής. Σε κάθε ομάδα συγκρίνουμε τις τεχνικές μας με το Linux scheduler όσον αφορά το throughput και το fairness.

Πρώτη Ομάδα Πειραμάτων

Παραθέτουμε παρακάτω ένα φόρτο διεργασιών με 8 L, 6 SC και 2 N. (σχήμα 1.1-14)

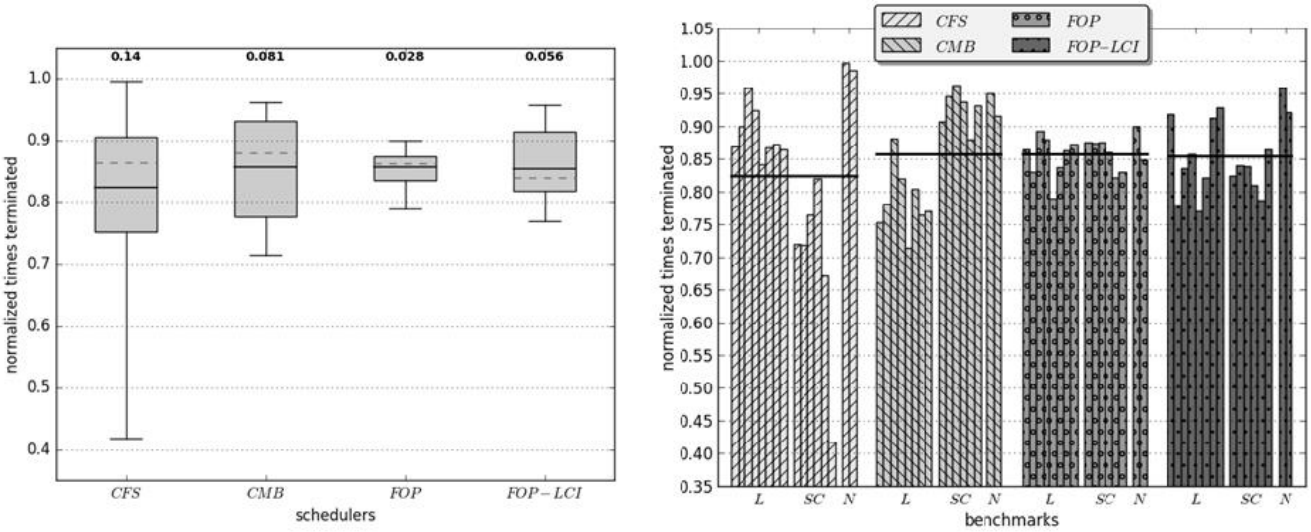


Figure 1.1-14 : Φόρτος διεργασιών με 8 L, 6 SC και 2 N διεργασίες

Παρατηρούμε ότι όλες οι τεχνικές που προτείναμε υπερτερούν και σε throughput και σε fairness τον Linux scheduler. Την καλύτερη απόδοση όσον αφορά το throughput επιδεικνύει ο FOP, φέρνοντας την πρόοδο των διεργασιών σχεδόν στην ίδια ευθεία. Ο CMB παρουσιάζει το μεγαλύτερο throughput, δίνοντας μία σημαντική ώθηση στις SC εφαρμογές και μειώνοντας ελαφρά τις L (λόγω πακεταρίσματος). Για το FOP-LCI, παρατηρούμε παρόμοια συμπεριφορά με το FOP. Βλέπουμε ότι ευνοεί τις N εφαρμογές αντι των SC και αυτό δικαιολογείται απο το γεγονός ότι οι N είναι ουδέτερες, δηλαδή μπορούν να τρέξουν και με τις L και με τις C.

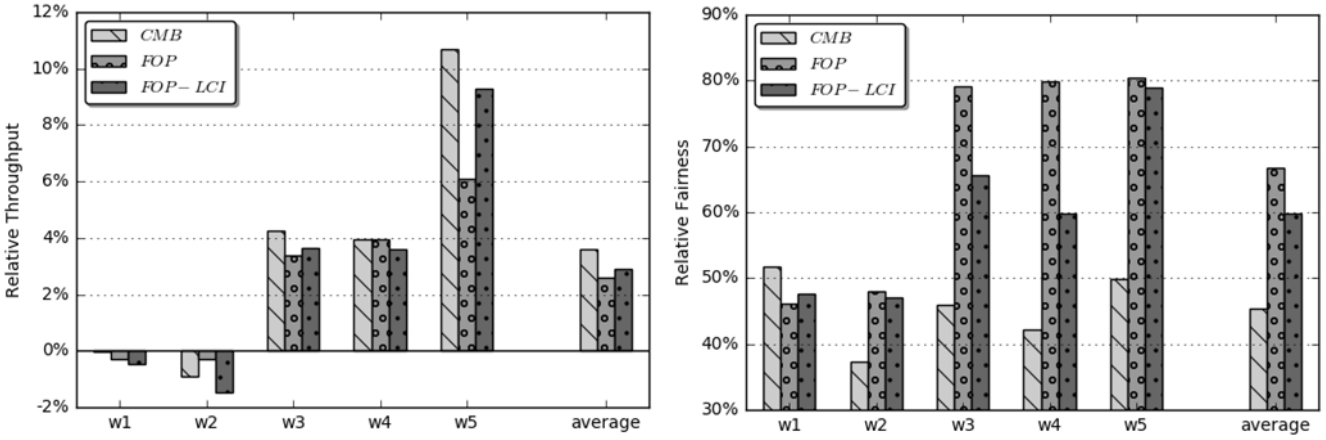


Figure 1.1-15: Βελτίωση throughput και fairness σχετικά με το Linux

Στο σχήμα 1.1-15 παρουσιάζουμε τα αποτελέσματα σύγκρισης με το Linux scheduler των φόρτων της πρώτης ομάδας. Όσον αφορα το throughput, ο CMB δείχνει τα καλύτερα αποτελέσματα, ενώ σχετικά με το fairness, πρώτος έρχεται ο FOP.

Δεύτερη Ομάδα Πειραμάτων

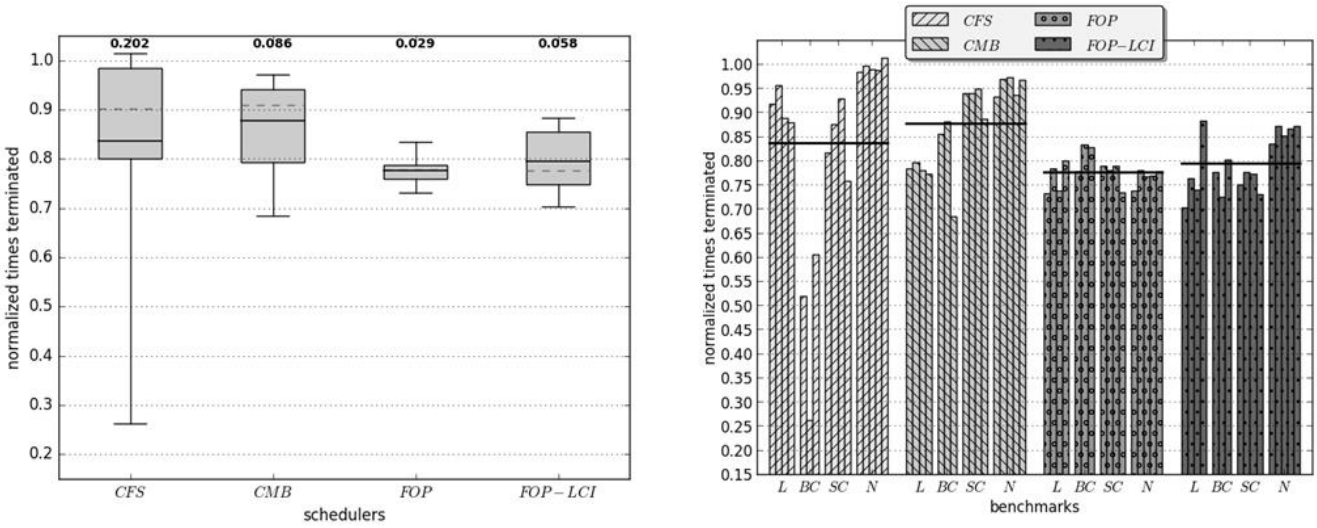


Figure 1.1-16: Φόρτος διεργασιών με 4 L, 3 BC, 4 SC και 5 N διεργασίες

Σε αυτό το φόρτο παρατηρούμε τα εξής. Ο CMB πετυχαίνει το καλύτερο throughput, καθώς καταφέρνει να απομονώσει πλήρως τις BC εφαρμογές στις τετράδες και να αυξήσει σημαντικά την απόδοσή τους, όπως επίσης κάνει και με τις SC. Μειώνει την πρόοδο των L όπως πριν, καθώς τις πακετάρει μαζί. Ο FOP, όπως πριν, έχει την καλύτερη απόδοση ως προς το fairness, ισορροπώντας σχεδόν τέλεια τις εφαρμογές, αλλά μειώνει το throughput, διότι τρέχει συχνότερα εφαρμογές που δεν κάνουν πρόοδο. Ο FOP-LCI φαίνεται, πάλι, να ευνοεί τις N εφαρμογές αντί των SC.

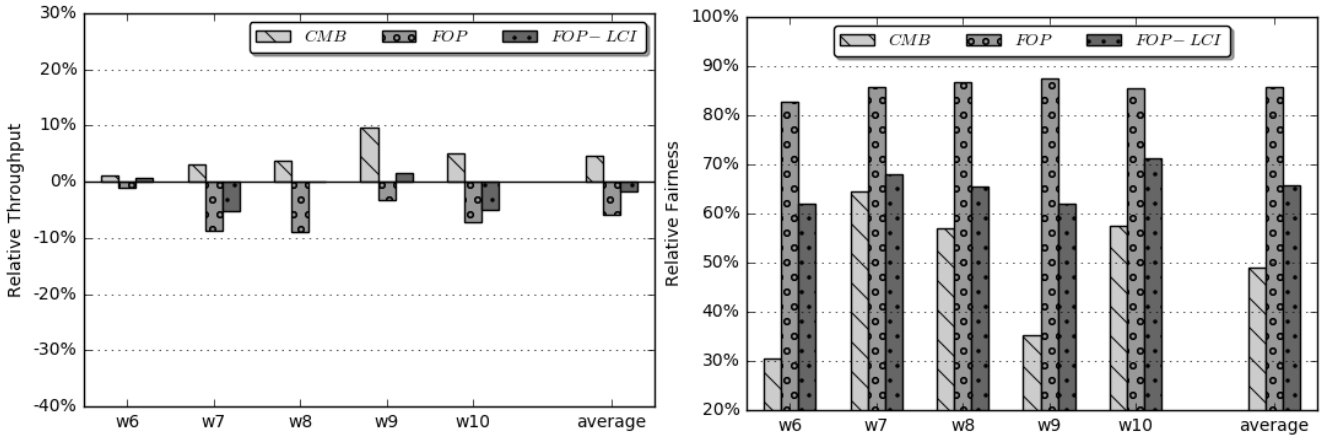


Figure 1.1-17: Βελτίωση throughput και fairness σχετικά με το Linux

Στο σχήμα 1.1-17 συγκρίνουμε τις τεχνικές μας με το Linux για τους φόρτους της δεύτερης ομάδας. Καταλήγουμε στα εξής. Ο CMB είναι ικανός να μοιράσει τις BC εφαρμογές και να προσφέρει το καλύτερο throughput. Από την άλλη ο FOP μειώνει ελαφρώς το throughput, αλλά πετυχαίνει το καλύτερο fairness. Ο FOP-LCI βρίσκεται σε μια ενδιάμεση κατάσταση

Τρίτη Ομάδα Πειραμάτων

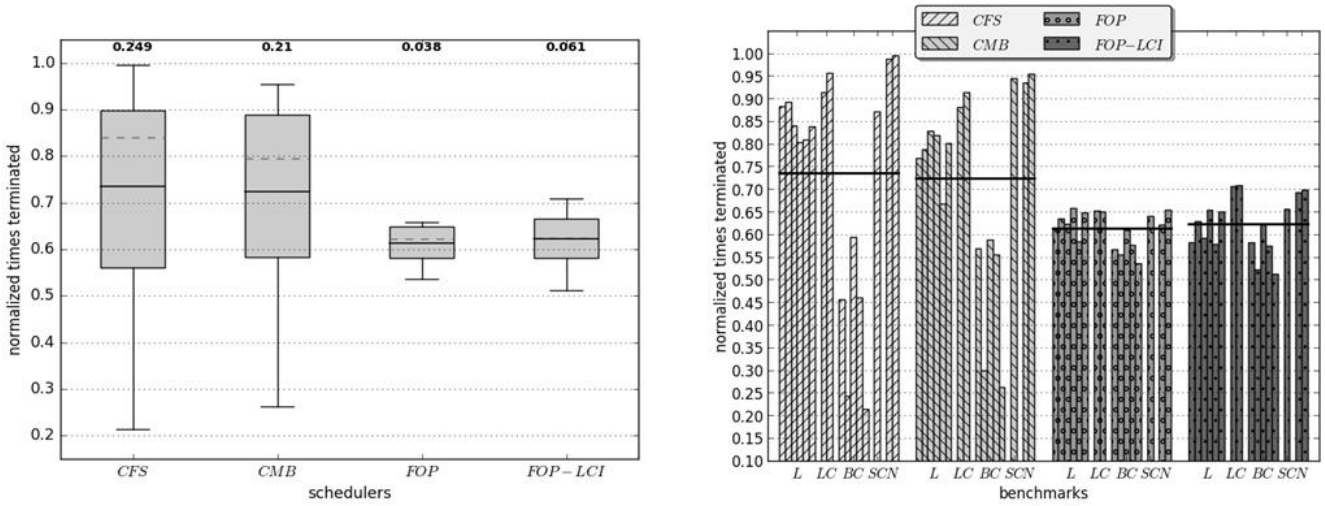


Figure 1.1-18: Φόρτος διεργασιών με 6 L, 2 LC, 5 BC, 1 SC και 2 N διεργασίες

Σε αυτή την περίπτωση (σχήμα 1.1-18) καταλαβαίνουμε ότι ο CMB δεν μπορεί να απομονώσει τις BC εφαρμογές και αναγκαστικά τις τρέχει μαζί. Λαμβάνοντας υπόψιν ότι πακετάρει και τις L, αυτό οδηγεί σε ελαφρώς μειωμένη απόδοση συγκριτικά με το Linux. Γενικά ακολουθεί τη συμπεριφορά του Linux. Απο την άλλη, οι άλλες δυο τεχνικές αποδίδουν όπως περιμέναμε, με χαμηλότερο throughput σχετικά με τις άλλες ομάδες πειραμάτων, διότι ο αριθμός των εφαρμογών που δεν εκτελούνται αποδοτικά (BC) είναι αυξημένος.

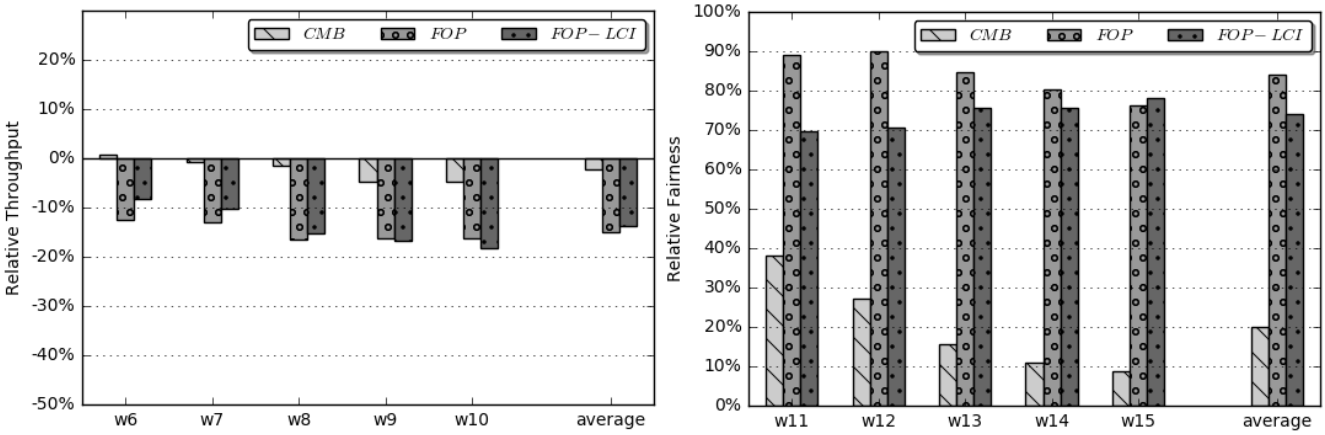


Figure 1.1-19: Βελτίωση throughput και fairness σχετικά με το Linux

Συγκεντρωτικά για όλους τους φόρτους διεργασιών έχουμε τα εξής. Ο CMB παρουσίαζει την μικρότερη μείωση στο throughput και την μικρότερη αύξηση στο fairness. Οι άλλες δύο τεχνικές έχουν παρόμοια συμπεριφορά, με τον FOP να είναι ελαφρώς χειρότερος στο throughput και ελαφρώς καλύτερος στο fairness από τον FOP-LCI.

Κεφάλαιο 4
Συμπεράσματα

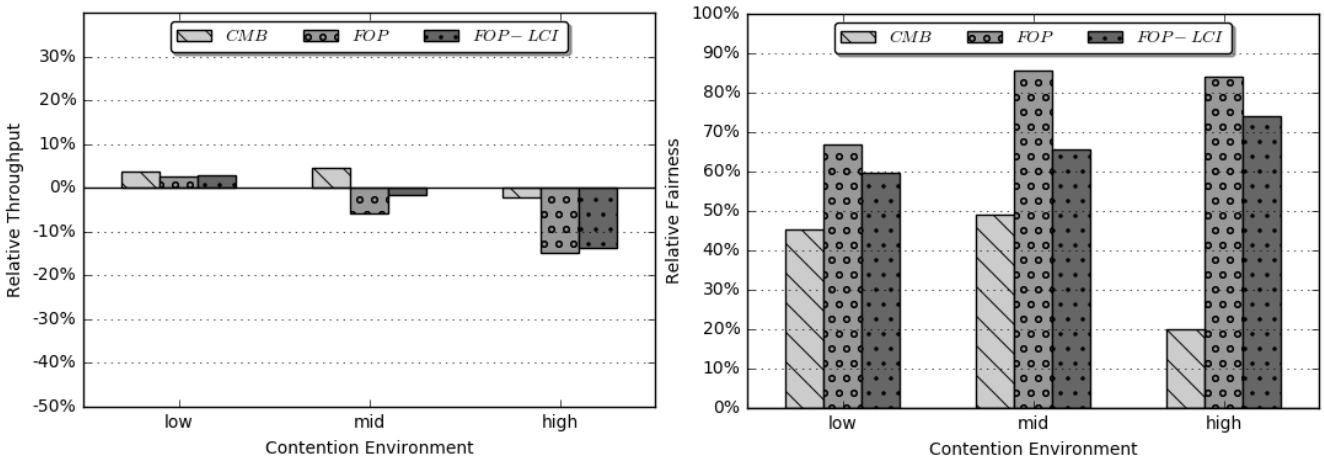


Figure 1.1-20: Βελτίωση throughput και fairness για κάθε ομάδα σχετικά με το Linux

Βλέποντας τα αποτελέσματα της σύγκρισης καταλήγουμε στο εξής. Όλες οι τεχνικές που προτείναμε κατανέμουν πιο δίκαια τους πόρους του συστήματος στις διεργασίες. Εκπληκτικά αποτελέσματα παρουσιάζει η δεύτερη και η τρίτη προσέγγιση καθώς φέρνει σε σχεδόν απόλυτη ισορροπία ην πρόοδο τους. Στην πρώτη προσέγγιση έχουμε τον περιορισμό ότι ο διαχωρισμός των εφαρμογών που υποφέρουν περισσότερο μπορεί να γίνει μόνο χρονικά. Αυτό σημαίνει ότι όταν ο αριθμός τους ξεπεράσει τις πιθανές τετράδες, τότε ο χρονοδρομολογητής μας δεν μπορεί να ανταποκριθεί σε αποδοτική αύξηση throughput και fairness.

Αναφορικά με την μείωση του throughput για τους FOP και FOP-LCI, αυτό συμβαίνει επειδή τον επεξεργαστή κρατάνε απασχολημένο κυρίως διεργασίες που δεν κάνουν πρόοδο. Παρ'όλα αυτά είναι σπουδαίο το αποτέλεσμα που προσφέρουν και αξίζει να θυσιάσουμε throughput ώστε να πετύχουμε σχεδόν την απόλυτη ισορροπία.

Σχετικά με τον FOP-LCI, επιδιώκαμε να αυξήσουμε την πρόοδο των C με την απομόνωση τους. Καταλαβαίνουμε όμως ότι με αυτό το τρόπο, συσσωρεύαμε τις L ή τρέχαμε τις BC μαζί, κάτι που δεν οδηγεί σε καλύτερα αποτελέσματα. Αυτό που παρατηρήσαμε είναι αύξηση του throughput απο την ευνοική εκτέλεση των 'ουδέτερων' εφαρμογών.

Συνοψίζοντας, βλέπουμε ότι οι σύγχρονοι χρονοδρομολογητές δεν λαμβάνουν υπόψιν τους τις ιδιαιτερότητες των πολυπήρνων συστημάτων και δεν παρέχουν σταθερότητα. Οι τεχνικές που προτείναμε μπορούν να εγγυηθούν αξιοπιστία και δικαιοσύνη στην κατανομή των πόρων.

Table of Contents

Chapter 1
Introduction

1.1 Definition of scheduling

Scheduling is a process concerning the decision of allocating resources to tasks. It occurs over given time periods and is responsible for optimizing several objectives [4].The resources, tasks and objectives vary between different organizations. In the examples below we illustrate the possible forms they may take.

Taking industry for example, we examine a factory that produces paper bags used for dog food or cement. Regardless of the type of bags we want to produce, the process remains the same and follows three stages, the printing, the gluing and the sewing. In each stage, there are machines of different characteristics, which execute different operations at various speeds. So we recognize here that the resources and the tasks coincide with the machines and the clients' orders respectively. The size of the order affects the completion time. In addition, the setup of a machine should be changed when a switch from one type of bag to another occurs. Another thing that should be taken into consideration is the impact a late delivery would have on the relationship between the factory and the clients. For these reasons, it is necessary to schedule the operation of the machines aiming to satisfy some of the following objectives, minimize the penalties a late delivery would impose or minimize the time wasted on different setups.

As far as transportation services are concerned, we examine a terminal at an airport. Each day hundreds of planes arrive to and depart from numerous gates. Some gates have plenty of space and are easily accessible by large planes, while others are in a location where it is difficult for a plane to reach in. Planes follow a certain schedule for their arrival and their departure. However unforeseen circumstances, like bad weather conditions, require changes to the main schedule. When a plane arrives to the airport, it occupies its gate. The arriving passengers disembark, the plane remains at the gate in order to be serviced and the departing passengers are boarded. However the flight could be postponed, because the destination airport could be busy enough to accommodate another plane. On this occasion, the plane may have to remain at the gate for a long period, preventing others from using it. In this example, the gates are the resources and the servicing of planes are the tasks. It is made clear that a scheduling policy should be adopted in order to optimize several objectives. In the first place, we need to assure that the arriving planes would be assigned to an unoccupied gate. Another objective may be the minimization of personnel's work or the minimization of delays.

Scheduling is widely used in information processing environments. In such environments, we recognize the CPUs (Central Processing Units) as the resources and the executable programs (processes) as the tasks. Computing systems provide with the ability of multiprogramming, assigning, in other words, numerous processes in the CPUs in a given time period. The scheduler is responsible for this work. It slices the CPU time into pieces and devotes them to different processes. In that way it assures that all of the processes would take a fraction of the CPU time and that the CPU would not be kept busy by only a few. The

optimization of specific objectives has a great impact on the overall performance of the system. For example, keeping the CPUs busy all the time leads to maximization of throughput or minimizing the waiting time of applications provides with fair distribution of the resources. For these reasons, schedulers are an indispensable part of computing systems and it is essential to make efforts to improve them.

1.2 Chip Multiprocessor

Manufacturers adopt a simple rule to improve computer performance. They increase the number of transistors a CPU contain and decrease in parallel their size. In accordance with Moore's law this has caused speeds to climb and prices to fall. The computing industry followed this trend for years. However, it is impossible for transistors to continuously shrink. Despite the fact that transistors grow thinner, manufacturers have to face two critical problems, power usage and heat generation. Even approaches for performance enhancement, like running multiple instructions per thread (ILP) have reached a plateau.

For these reasons, potentials for improvement of the processor performance have been seriously restricted. Chip performance experienced a 60% increase per year in 90s but declined to 40% per year from 2000 to 2004. Apparently designing a chip with 20% speed increase, costing twice the die area would not be ideal for meeting our expectations for performance boost, energy efficiency and cost effectiveness [5].

In response, manufacturers are turning from single-core to multi-core architectures. Instead of one increasingly powerful core, they are building chips with multiple more energy-efficient processing cores. These cores run in lower speeds as compared to the single-core systems but they improve overall performance by executing more processes in parallel. Taking for example a dual-core chip running multiple processes simultaneously, we conclude that it is about 1.5 faster than a chip with just one core.

When a single-core chip runs multiple programs, it assigns a time slice to work on one program and then assigns different time slices for others. This can cause conflicts, errors, or slowdowns when the processor must perform multiple tasks simultaneously. When considering multi-core chips, on the opposite, it is feasible to execute multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing. So if you have multiple tasks that all have to run at the same time, you will see a boost of performance with multi-core processors.

The improvement in performance gained by the use of a multi-core processor depends very much on the software algorithms used and their implementation. In particular, possible gains are limited by the fraction of the software that can be run in parallel simultaneously on multiple cores. Programmers must find good places to break up the applications, divide the work into roughly equal pieces that can run at the same time, and determine the best times for the threads to communicate with one another (thread-level parallelism (TLP)), a work that makes the parallelization of software a significant ongoing topic of research.

Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package. Because the chips' cores are on the same die, they can share architectural components, such as memory

elements and memory management. They thus have fewer components and lower costs than systems running multiple chips (SMP). Also, the signaling between cores can be faster and use less electricity than on multichip systems.

The advantages a multi-core processor can offer, mentioned above, made them the dominant in the area of computing systems. Multi-core processors are widely used across many application domains including general-purpose, embedded, network, digital signal processing (DSP), and graphics. Multi-core technology, with its promise of improved power efficiency and increased hardware utilization, has been embraced by the industry: AMD, Fujitsu, IBM, Intel and Sun Microsystems are shipping multicore systems and have announced plans to release future models. Having become mainstream in both server and desktop processors, we expect to see processors with tens and even hundreds of cores on a chip, over the next decade.

1.3 Operating System Services

An operating system is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.

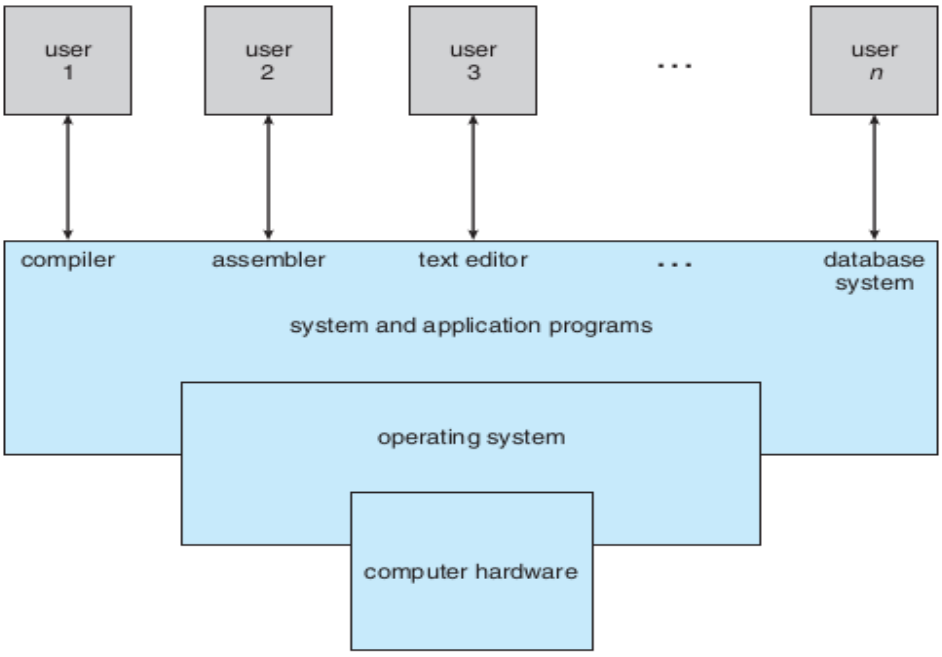


Figure 1.3-1: Abstract view of the components of a computer system

A computer system can be divided roughly into four components: the *hardware*, the *operating system*, the *application programs*, and the *users (Figure 1.3-1).* The *hardware* consists of the central processing unit (CPU), the memory and the input/output (I/O) devices

and provides the resources for the system. The *application programs*, for example word processors, spreadsheets and browsers, define the ways in which these resources are used to solve *users*' computing problems. The *operating system* controls the hardware and coordinates its use among the various application programs for the various users [6].

The operating system is responsible for process, memory and storage management. Regarding process management, it schedules processes and threads on the CPUs, creates and deletes both user and system processes, suspends and resumes processes and provides mechanisms for process synchronization and communication.

Concerning the memory management, it keeps track of which parts of memory are currently being used and who is using them, decides which processes and data to move into and out of memory and (de)allocate memory space as needed.

Operating system abstracts the physical properties of the storage devices and creates a logical, uniform storage unit, the file. It accesses the storage devices via the files and takes charge of specific operations for managing the file-system. It creates and deletes files or directories, supports mechanisms for their manipulations, and maps files onto storage devices. I

In the case a program needs to communicate with a file or a device, the operating system provides I/O operations. The communication can be achieved via special functions that utilize the devices properly and offer efficiency and protection

Communication between users and their programs should be helpful and convenient. The operating system provides the user interface (UI), in order to make computer interactive. There are many form an interface could take. It may be a command line interface (CLI), which uses text commands. The other and most common form is the graphical user interface (GUI). It consists of a window system with a pointing device and a keyboard.

Protection and *security* are two aspects that an operating system should take into consideration. User and processes are not allowed to have access to the resources without regulation. Operating systems enforce some controlling mechanism in order to protect the execution of the processes. They ensure that files, memory and CPUs can be utilized by the processes that have gained proper authorization.

Operating systems should offer not only protection but also security. This means that they should be able to defend the system from external and internal attacks. Such attacks may include viruses, identity theft, denial-of-service and theft of service.

1.4 Process Scheduling

A process is considered as the task that should be completed in a computing system. A system consists of numerous processes, some of them are operating-system processes, meaning that they execute system code, and the others are user processes, meaning that they execute user code.

CPUs are considered as the main resources of a system. Operating systems assign these resources to the tasks aiming to optimize some objectives. Scheduler is the indispensable part of the operating systems that holds this responsibility. Its main role is to provide with the ability of multiprogramming. CPU cannot be kept busy by a single process all the time. It should be ensured that all processes take a slice of the CPU’s time and make progress

However, multiprogramming is not the only reason that makes the scheduler so important. The way it manages the processes has a great impact on the system performance. The extent at which the various objectives are satisfied is reflected on the system, as these objectives consist the criteria for the performance measurement.

The main objectives that a scheduler tries to optimize are the following. First of all we want to maximize the *CPU utilization*, meaning that we try to keep the CPU as busy as possible. In addition we want to maximize the *throughput*, which is the measure of work done. It indicates the processes that are completed per time unit. Furthermore it is desirable to minimize the *turnaround time* of applications. This refers to the interval from the time of submission to the time of completion of a process. *Waiting time* is another criterion we try to minimize. It is the total time that an application spent waiting in the ready queue. We do not want waiting time to be gathered to one application only, but to be equally distributed among them. Another thing we should take into account is the *response time*, which is the time from the submission of a request until the first response is produced. In an interactive system, it is preferable to minimize response time

Regardless of the various scheduling algorithms proposed in order to achieve the optimization of the desirable objectives, the main abstract purpose of the scheduler is analyzed in the meaning of *time-sharing* and *space-sharing* the CPUs. *Time-sharing* the CPU means multiplexing a single processor in time and devotes every time-slice to different process, so that CPU executes multiple processes in a time interval. *Space-sharing* the CPUs is about deciding on which CPU each process chosen to run at a given time interval will be assigned to run.

For single processor architectures the scheduler enforces obviously time-sharing only. OS schedulers for these architectures had become so optimized that need for further improvements dramatically subsided, thereby diminishing interest in this research topic. In the late 90s, the scheduling problem was considered solved; at least it appeared that way until the advent and subsequent near ubiquitous proliferation of chip multiprocessors (CMP).

CMP architectures, consisting of multiple processing cores on a single die added new dimensions to the scheduler's role. In addition to time-sharing, it became necessary to also space-share cores among the processes. In the state-of-the-art schedulers the cores are treated as isolated and independent processors, just like SMPs, and the strategy for placing processes on cores is load balancing. The OS scheduler tries to balance the runnable threads across the available resources to ensure fair distribution of CPU time and minimize the idling of cores.

1.5 The Linux Scheduler

To comprehend how the time and space-sharing are applied and how scheduling strategies for optimization of some desirable objectives are implemented in practice, we now present the Linux Scheduler.

Linux contemporary multiprocessor operating system uses a two-level scheduling approach for efficient resource distribution. In the first level, it assigns queues in each core and adapts fair policies to manage each queue. In the second level, it redistributes the tasks across the cores using the load balancer. The first level reflects the time-sharing policy, while the second level reflects the space-sharing policy. In the paragraphs that follow we discuss each

level independently.

Run Queue Management

The Linux scheduler is called CFS (Completely Fair Scheduler). It is different from the classical schedulers, as it does not embrace the idea of time slices. It takes into account the waiting time (the time an application is in the run queue and is ready to be executed) of the applications only, rather than computing time slices for each one and run it until their time slice is used. In this level, a per core run queue is created and represents the set of programs assigned on this core. CFS schedules the tasks of each core, deciding which application from the queue will run next, via its run queue management.

Inspecting deeper the general principle of CFS we understand why it is called completely fair. It provides maximum fairness to tasks in terms of power, scheduling the task with the gravest need for CPU time. Associating waiting time with unfairness, it ensures that no heavy unfairness will gathered in tasks, as it picks the application with the highest waiting time and assigns it to the core. In that way, it shares the unfairness equally among all the tasks in the system.

Run queues of the CFS are implemented through a time ordered red-black tree.

CFS enforces priority indirectly using a delay factor for the time a task is allowed to be executed. Lower-priority processes have higher delay factors, while higher-priority processes have lower factors. This means that the time a task is permitted to run passes more quickly for a lower-priority application than for a higher-priority.

The Load Balancer

In the CMP platforms, each core has its own run queue, as described previously. Completion times of tasks across the run queues are not the same. For this reason, some cores may execute all the tasks of their run queues and become idle, while other still have many applications in the queue waiting to be executed. This phenomenon is called load imbalance. The load balancer is designed to solve this problem. It is called periodically migrating tasks from the busiest CPU to the less-loaded. In that way, it attempts to balance the number of tasks across all the run queues of the system.

The selection of potential tasks to be migrated occurs based on meeting some restrictions. For example it needs to be ensured that the candidate application is not on a CPU at the moment or that it is allowed to run on the destination core.

Topology and Locality Awareness

The Linux scheduler organizes the run queues hierarchically into different scheduling domains, in such a way that it is reflected how the hardware resources are shared. It balances the queues progressing up the following domains of the hierarchy: Simultaneous Multithreading (SMT) contexts, last-level caches, and NUMA domains. At each level the load balancer decides how many processes need to be migrated between two groups in order to balance the number of tasks in those groups. If the groups are already in balance, no action is

taken. The load balancer is invoked in a frequency specified by both the scheduling domain kernel settings and the instantaneous load.

As the level in the hierarchy increases, the frequency of invoking the balancer and the number of migrations decreases. This is based on the concept that migrations between domains located higher in the hierarchy are more costly than those between domains located lower. This is valid if we consider migrations between NUMA banks and between SMT contexts.

1.6 Chapter Description

In Chapter 2 we present the problem of co-executing applications that contend for the shared resources of a CMP. In addition, we study the destructive consequences that the agnostic treatment of threads has on the objectives of a scheduler.

Motivated by the observations made on the previous chapter, in Chapter 3 we propose 3 scheduling policies that attempt to mitigate the phenomenon of unfair resource sharing. In addition, we describe the execution platform used for our experiments, and the tools and mechanisms used for the implementation of our techniques.

In Chapter 4, we form 15 workloads and group then in 3 categories. We evaluate our proposed schedulers and compare them with the Linux scheduler, describing in detail the results for each category.

Conclusions and ideas for further improvements are summarized in Chapter 5, while approaches made by the research community are presented in Chapter 6.

Chapter 2
Problem Definition and Motivation

2.1 Resource sharing utilization

As mentioned before, CMP's cores managed to overcome the problems of transistor shrinking, power consumption and heat generation that a simple core had to face. The need for fast signaling between cores, less electricity usage and in consequence cost effectiveness lead the manufacturers to integrate the cores into a single circuit die and thus share architectural components such as memory elements and memory management. The main components (on- and off-chip) shared among the cores of a CMP architecture are the LLC (Last Level Cache) the memory bus or interconnects, DRAM controllers and pre-fetchers (Figure 2.1-1).

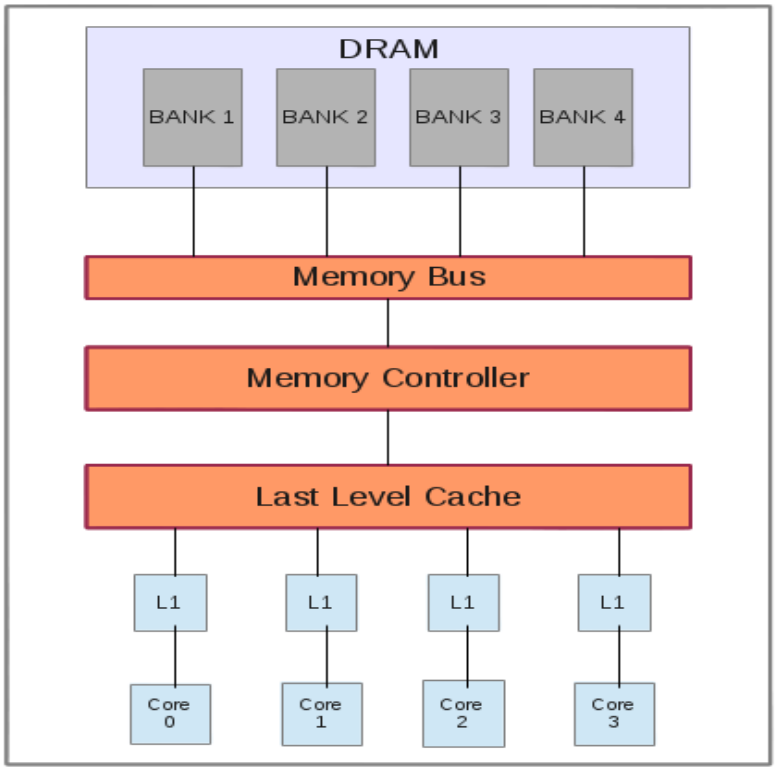


Figure 2.1-1: Shared resources of a CMP platform

While CMP platforms seem to be promising for performance improvement, sharing the workload to parallel cores, their design would not have such a beneficial impact on the various threads running simultaneously. Workloads of both client and server domain consist of a variety of applications with a big range of different characteristics and behavior. When they are executed concurrently in the neighboring cores of a CMP they can use its resources either constructively or destructively.

2.1.1 Constructive behavior of resource sharing

CMP platforms can offer tremendous speed up for applications that exploit the thread level parallelism (TLP). When multiple cores share resources, threads of an application running concurrently on those cores can constructively use these resources in a number of ways.

Threads of an application share data that they access in different patterns. A memory access pattern that can lead to constructive use of shared resources is the one of fine-grain sharing. This kind of sharing is achieved when threads ping-pong data back and forth between private caches and main memory [7]. Such behavior express threads that update the same or coterminous group of data either concurrently or one after another, like the producer-consumer example, where one thread writes some data and another thread reads it.

When co-executing threads, presenting the above behavior of memory accesses, into the cores of a CMP, the sharing of LLC, memory controllers, memory bus and hardware prefetching helps the threads to productively cooperate. Last Level Cache in this case is the main component playing the dominant role for constructive behavior. It holds blocks of data that are accessed from all the cores with low latency at the same period. This means that only one copy of the data is shared rather than multiple copies spread out across private caches, reducing coherence traffic [8]. It also means that since data are concurrently used by all cores they are not ping-ponged back and forth from cache to main memory, reducing the latency of a costly transfer, contaminating less frequent the memory bus, dram controllers and prefetchers. Furthermore data are being brought to cache by one thread which exploiting the prefetching logic brings a coterminous group of data that can be used by the other threads. This reduction in data redundancy also leads to larger effective capacity on chip.

Even threads not sharing a cache can interfere constructively. If caches are located in the same socket, coherence traffic can avoid transfer of messages in a heavily contended system bus. Of course, coherence traffic between caches sharing a bus is less costly than between caches located to different sockets.

2.1.2 Destructive behavior of resource sharing

Considering the workloads to be executed in CMP platforms both in desktop and server environment, they consist of a variety of applications multithreaded or not that need to run concurrently on the neighboring cores. Threads of different applications, usually, do not communicate with each other or share data, thus they don't help each other when running simultaneously, using constructively the shared resources of a CMP. Even threads of the same application running together will not benefit, if they exhibit coarse-grained sharing [7]. Opposite to fine-grained sharing, they do not ping-pong data back and forth, but there is a long period where one thread accesses the data followed by a long period where another thread accesses the same data. In that way they do not bring data for each other and act like being threads of different applications that exhibit different memory access properties.

Such applications are competing for the shared resources, as their needs for utilization of elements of the memory hierarchy try to be satisfied. The performance of the applications participated in this conflict for resources can be negatively impacted, being slowed down by

hundreds of percent relative to running alone, because of the destructive interference they cause to each other. In this paper we will examine only the destructive behavior between threads, when they contend for shared resources.

Below we present how the contention in each level of resource sharing (Last Level Cache, DRAM Controllers) impacts destructively on performance of applications.

2.2 Last Level Cache Contention

When threads run concurrently in cores, the first point of contention is Last Level Cache they share. Threads bring data to cache without regulation, following the cache replacement policy being implemented on the CMP platform, in order to save the heavy latency the memory impose. This means that a thread evicts the data of a neighboring thread or its data are evicted by another (Table 2.2-1).

Processor	Cache Line X	Action	Cause	Result
P1	Access Data A	Compulsory miss	Cache empty	-
P1	Access Data A	Hit	In cache	-
P2	Access Data B	Capacity miss	Not in cache	Evicts P1 Data
P1	Access Data A	Conflict miss	Evicted by P2	Evicts P2 Data
P2	Access Data B	Conflict miss	Evicted by P1	Evicts P1 Data
P1	Access Data A	Conflict miss	Evicted by P2	Evicts P2 Data
P1	Access Data C	Capacity miss	Not in cache	Evicts P2 Data

Table 2.2-1: Impact of interference in the Last Level Cache

As it is depicted in the above example, the process P1 is subjected to pay the cost of two extra misses and P2 of one extra miss, forced to wait more time for the data to come from memory and consequently having a severe degradation to their performance.

Research showed that the 3C classification of misses (Compulsory, Conflict, Capacity) is inadequate to analyze the exact cause of misses and cannot model the contention existing in the shared cache. They provide a new cache miss classification, CII (Compulsory, Intra processor, Inter processor), that is able to model the interactions between transactions of multiple processors at the level of shared cache [9]. Intra-processor miss is the one caused by the same process, opposite to Inter-processor which is caused due to a conflict by a neighboring process. In the above example, P1 has two inter-processor and one intra-processor miss and P2 has one inter-processor and one intra-processor miss.

Using a set of benchmarks they measured the distribution of misses of their classification scheme to an 8 processor CMP architecture with L1 private caches and L2 shared unified Last Level Cache. On an average, 40.3% of the misses are Inter-processor misses, 24.6% of the misses are Intra-processor misses and the remaining 35.1% are compulsory misses.

Several studies have shown how the contention for the Last Level Cache negatively

affect the performance of the applications co-scheduled [10] [11]. They showed that the reduction in IPC is corresponded to the increase of cache misses.

Working on a CMP platform with Nehalem architecture, using the one of two packages consisting of four processors with up to L2 private caches, one fully inclusive and fully shared L3 Last Level Cache and an integrated memory controller, we show how the conflict for the LLC affect the co-running applications. All bars are normalized to the case where applications were running alone.

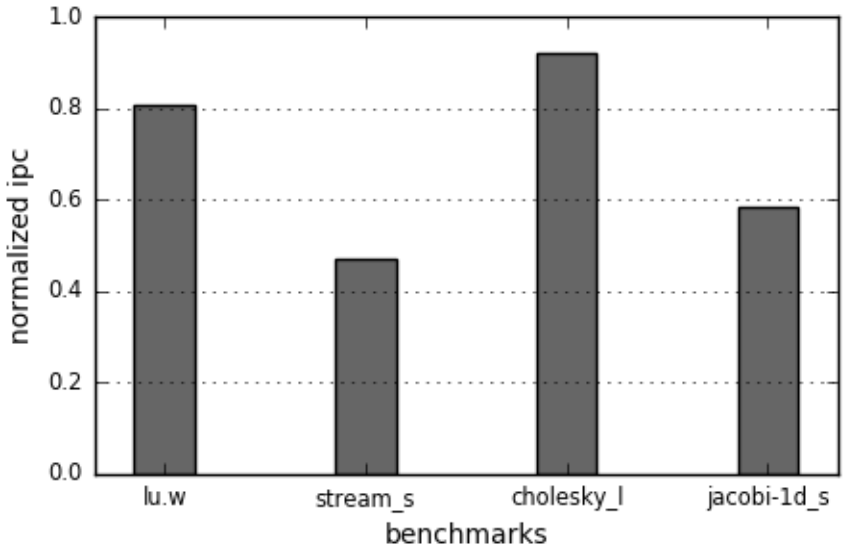


Figure 2.2-1: Destructive effects of LLC contention on IPC of 4 co-running applications

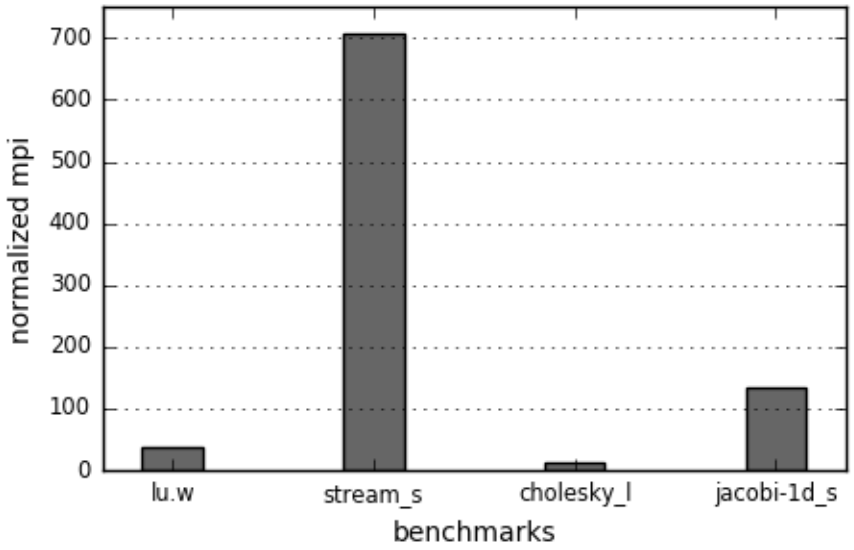


Figure 2.2-2: Impact of LLC contention on MPI

Examining the two above figures we come to the following conclusions. When applications are executed concurrently, they cause increase of their Misses Per Instruction (MPI), because of their destructive interference in the shared LLC. This increase reflects to a

corresponding decrease of their IPC, which occurs at a different rate for each application. Furthermore, the increase of MPI is not uniformly distributed, meaning that the LLC contention is not treated equally between threads and thus present unfair slowdown.

Summarizing, the above figures, assured our previous statements, that when applications compete for the shared LLC, they do cause destructive interference that leads to severe performance degradation (x1.23, x2.12, x1.08, x1.71 times of slowdown).

2.3 Memory Bandwidth Contention

The other crucial point of contention is the memory bus, which is the mediator between the Last Level Cache and the main memory. Memory Bandwidth is referred as the rate that data are written to or loaded from main memory through the memory data bus. Its theoretical maximum rate depends on the source and timing constraints of the memory system, such as memory clock frequency, bus width, number of data transfers per clock. The actual sustainable rate, though, is implied by the memory access pattern of the application running and the scheduling algorithm imposed to those arriving requests by the Dram controller.

When multiple threads are running concurrently, they contaminate the memory bus with their data without regulation. As long as the sustainable rate of transactions is limited, their bandwidth would be reduced, in order to obey the source constraints.

Working on the same CMP platform described before, we show how the contention on the memory bandwidth affects destructively the performance of the applications running together.

To avoid contribution to the performance degradation from the shared LLC contention, we choose applications that have streaming behavior. This means that data transferred from main memory to LLC are further descending to the private caches without gaining any benefit from their preservation on the LLC. Every second another amount of data arrives on the LLC, evicting the previously brought, and moves towards to the processor with the same rate. This direct flow of data characterizes these applications as streaming and do not take advantage of the LLC existence. Such applications when running concurrently, they do not cause interference to each other because of the shared LLC. This happens because each application have a flow of data which passes through the LLC and is directed deeper to the processors, evicting each other’s data, but without causing extra LLC misses, as they wouldn't refer to them in future time. They only refer to the flow that comes every second. Usually this kind of applications is memory intensive, meaning that they access the main memory at long and frequent time periods.

The Figure 2.3-1 shows how bus bandwidth contention affects the performance of the co-executed applications. The bars describing the impact on IPC, memory bandwidth and misses per instruction are all normalized to the case where these applications are running alone. Examining the results, we come to the following conclusions. On the one hand, our previous statement that working with streaming applications sharing the LLC we avoid the contention coming from this source comes true. Misses per Instruction have remained unharmed. On the other hand, all applications are subjected to lower memory bandwidth rate, causing a decrease

on IPC (instructions per cycle) that follows the same reduction pattern. Time is increased inversely proportional to the IPC reduction, as expected.

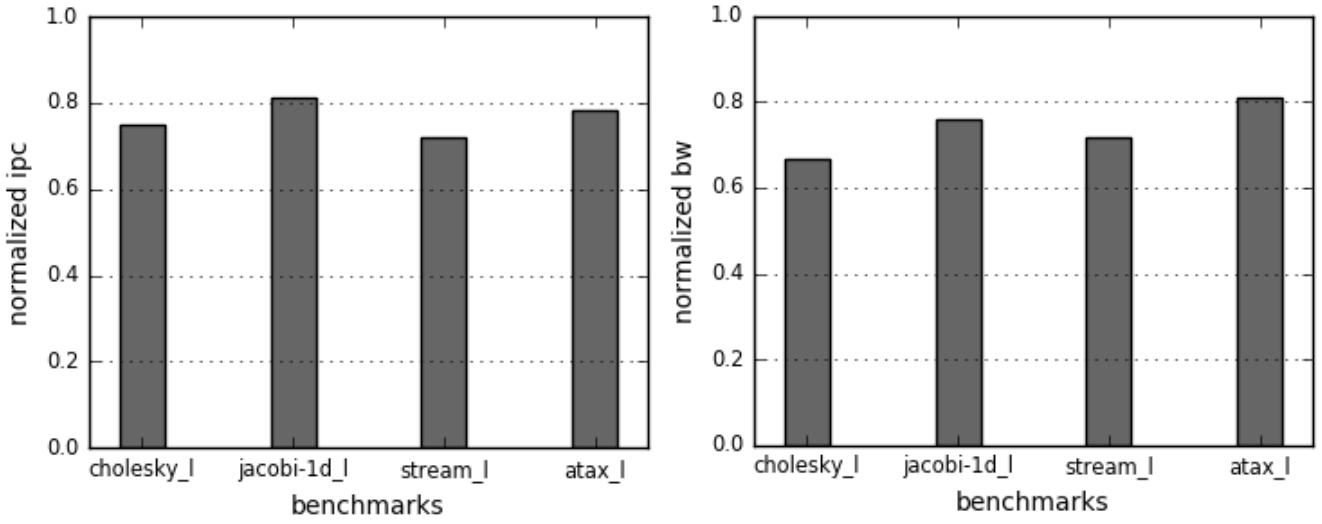


Figure 2.3-1: Bandwidth and IPC distribution of 4 co-running streaming applications subjected to bandwidth contention

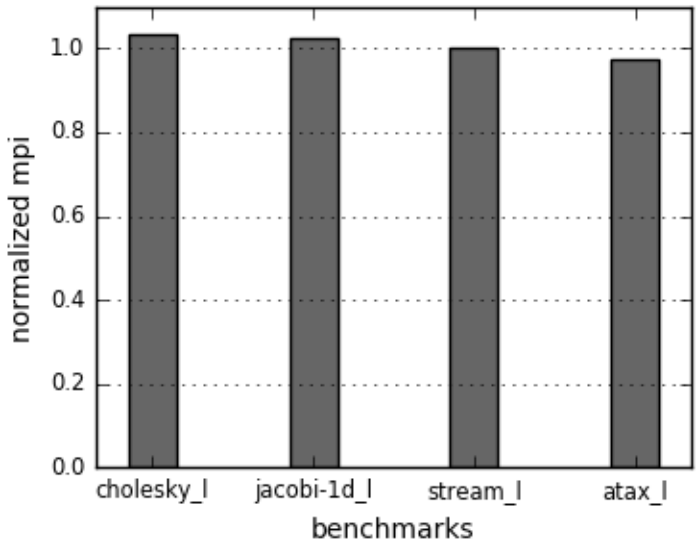


Figure 2.3-2: MPI behavior for the set of streaming application executed concurrently

The total sustainable bandwidth the memory controller can perform is $B_{actual-peak} = 13.21\,GB/s$, as measured form the co-execution of the streaming applications. The sum of the sustainable bandwidth of each application, when running alone, is $B_{total-peak} = 18.5\,GB/s$. The percentage of bandwidth, that the dram controller satisfies, is $\frac{B_{actual-peak}}{B_{total-peak}} = 0.7319$. This

means that when running together, the memory controller contention cause each application to reach a 73.19% proportion of their initial bandwidth, on the ideal case where the reduction was uniformly distributed. As we see the ratio of bandwidth of each application is with a small deviation around this price, that is the memory controller shares its actual peak bandwidth quite fairly between the conflicted applications. The bars of IPC are decreased proportionally to those of bandwidth by a factor of 0.914 (~1) on average. This behavior shows how important the bandwidth is for the streaming applications. As mentioned above, the flow of data, translated to bandwidth here, determines the performance of such applications, so a decrease on bandwidth leads to same decrease on IPC.

2.4 NUMA Architecture

In the last years, processor manufacturers have adopted a new way for accessing the main memory system, the NUMA (Non Uniform Memory Access) architecture. Traditionally, processors access data through an external bi-directional data bus called front-side bus (FSB). This bus is connected to the memory controller hub that receives requests from all the processors, directed to the main memory. This uniform memory access (UMA) through a common FSB and memory controller is replaced by a new system architecture (Intel Quick Path for example) that integrates a memory controller into each microprocessor, dedicates specific areas of memory to each processor, and connects processors and other components with a new high-speed interconnect (Figure 2.4.1).

The main benefit, that NUMA architectures offer, is that the contention for bus and its bandwidth present a significant reduction. Processors do not anymore need to compete with each other to reach memory system, as they have their own dedicated memory banks accessed through an integrated memory controller. In case, that one processor needs to access other processor's memory, this happens through a high speed interconnect that links all the processors.

Despite the fact that, now processors are integrated into packages and access specific memory areas, through separate memory channels and memory controllers, avoiding the contention of a unique shared bus, contention cannot be eliminated completely. Even processors of the same package would suffer from memory controller and bus contention, as these components are still shared among them. We showed previously that contention does exist, when co-running four applications in a Nehalem quad core package (NUMA architecture).

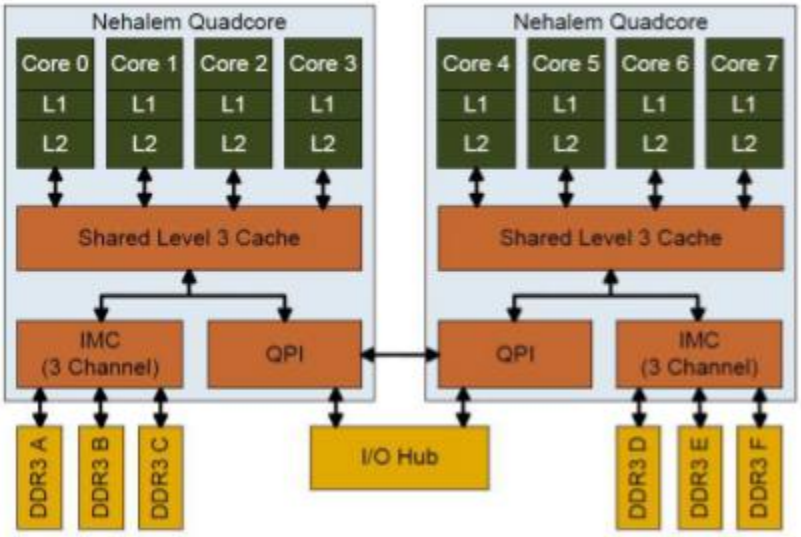


Figure 2.4-1: Nehalem, NUMA architecture

2.5 Motivation

On the arrival of the CMP architecture, most of hardware and software techniques implemented on single-core or SMP processors are adopted unmodified. Cache and memory controller scheduling policies, optimized for isolated processors, do not satisfy the needs of threads that run concurrently on neighboring cores and share resources. LRU cache replacement policy treats the misses from multiple threads uniformly, allocating space based on their rate demand, causing destructive interference. Memory scheduling policies, designed to maximize overall data throughput, prioritize requests of threads with specific memory patterns over others. Hardware, in general, do not support contention or thread aware mechanisms, in order to mitigate or avoid destructive interference that causes severe performance degradation.

Not only hardware but also software implementations proposed for utilization of hardware sources on single-core or SMP architectures are destined for CMP platforms. State-of-the-art schedulers, running without modifications on CMPs, create the illusion that the processors, they handle, are isolated and independent units, without taking into account the sharing components and their occurring contention. This flaw causes a serious impact on the optimization of the scheduler objectives.

2.5.1 **Low progress**

Before inspecting deeper how the contention agnostic treatment influences the characteristics of a scheduler, we describe what alone execution means for an application. Alone execution time of an application is very useful for comparing with the time needed to

finish when contending with others. It is measured when application is running completely alone, with the neighboring cores remaining idle, and the shared sources are exclusively occupied only by this application.

A scheduling algorithm should provide with maximum utilization of CPU time, ensuring that all the threads it manages are assigned frequently on the different cores and make progress. But when threads competing for the shared resources, are placed together without regulation, their progress is subjected to the consequences of contention.

For the co-running group below, every application make a fraction of the ideal progress achieved if no contention occurs (same as running alone). For example, if an application on every time slice the scheduler assigns, achieves the 70 % of the IPC measured in the alone execution, the actual progress made on this time slice is equal to the 70 % of the ideal progress, where no contention exists.

So the actual time of an application running on contention is given by the form:

$$time_{actual} = \frac{IPC_{co-running}}{IPC_{alone}} \cdot time_{co-running} \quad (form\ 2.5.1)$$

and is interpreted as the time passed if this application was running alone. When the application is finished, the actual time is the same as the alone execution time and is equal to the measured co-execution time multiplied by the factor that implies how much progress was done (form 2.5.1). The slowdown imposed on this application is inversely proportional to the IPC ratio:

$$Slowdown = \frac{time_{co-execution}}{time_{alone-execution}} \overset{(2.5.1)}{=} \frac{IPC_{alone-execution}}{IPC_{co-execution}} \quad (form\ 2.5.2)$$

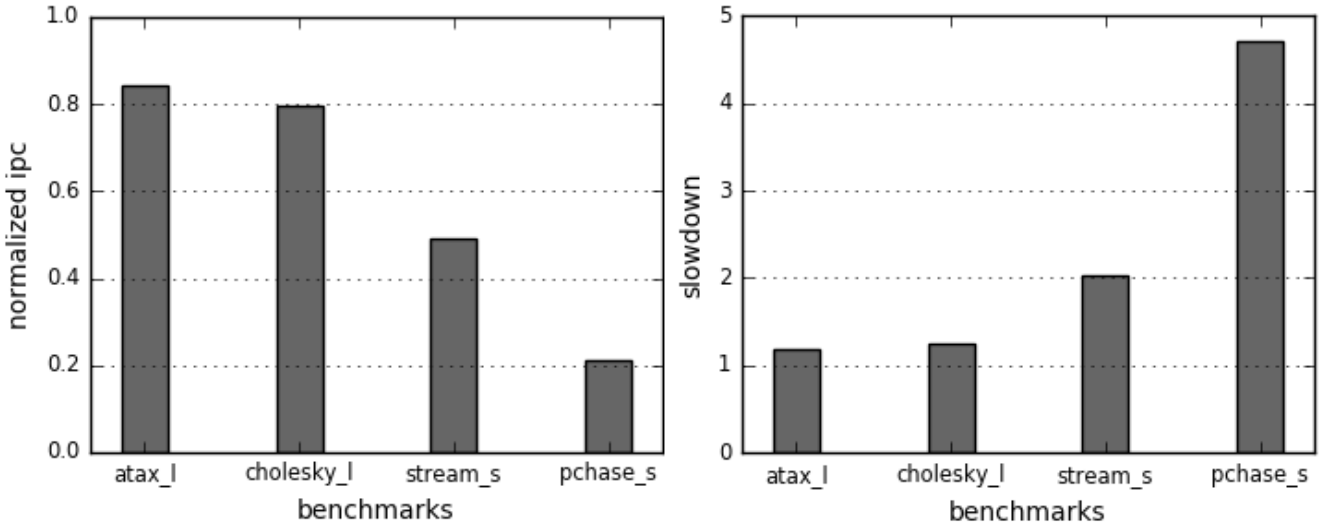


Figure 2.5-1: Relationship between IPC and slowdown

As showed in the figure the relation between IPC and Slowdown (form 2.5.2) is satisfied

$$Slowdown_{pchase} = \frac{IPC_{alone-execution}}{IPC_{co-execution}} = \frac{1}{0.208} = 4.808$$

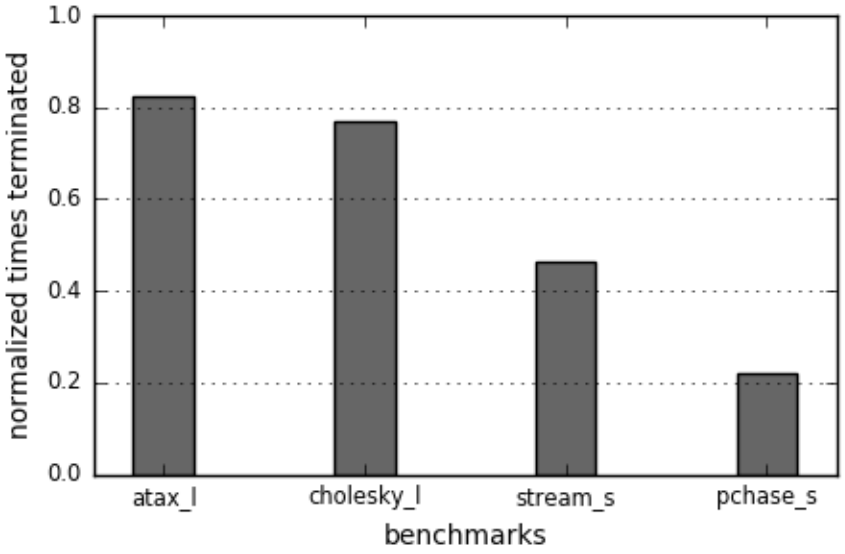


Figure 2.5-2: Normalized times terminated over standalone execution of 4 applications running on a time window of 5 minutes

This slowdown has a further negative impact on the objective of the scheduler to maximize overall throughput, as in a 5 minute time period (Figure 2.5-2) the application suffering from contention is finished far fewer times than expected if running alone (x 0.208). Furthermore the ensuring that threads do make adequate progress is no longer reliable, leading in cases of great suffering to thread starvation.

2.5.2 **Poor fairness**

When applications suffer from contention, their performance is not affected uniformly. This means that they experience contention in variable ways. Some of them may present no suffering; others may present average or severe degradation. On the example given, it is made clear that the contention resulted in ununiformed IPC reduction for the four co-executed applications in a 5 minute time period, giving unfair advantages to some at the expense of others (Figure 2.5-2). This means, that the main principal of the scheduler to share the resources equally among threads is trespassed.

The poor fairness enforcement, lead to another major problem. Thread priority policies are unable to be adapted by the scheduler and if so, they lead to undesirable results. Giving higher priority to an application, results in greater actual progress for this one. But when treated

unfairly this progress is impeded by the progress of lower priority threads, taking for example pchase as a high priority application and stream a lower one, leading to priority inversion.

2.5.3 **Co-runner Dependent Performance**

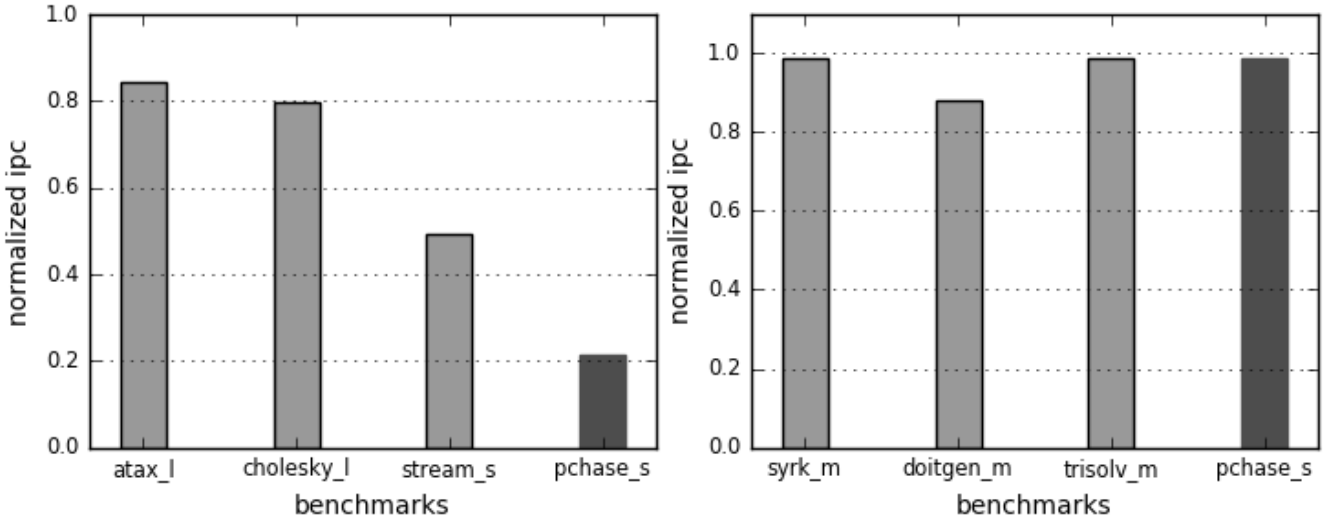


Figure 2.5-3: Variable performance of pchase_s application between different co-running groups

The level of contention generated depends on the co-running threads that compete for the shared resources. As illustrated on the above figure, the application pchase_s is subjected to high level of contention, resulting to high IPC reduction. When placing together the same application with 3 different co-runners, we see an important increase on the IPC as a result of lower occurring contention (Figure).

This means that the performance of an application is highly variable between different co-running sets. This unstable and co-runner dependent behavior of applications leads to unpredictable and workload-dependent overall performance of a CMP platform. Consequently, Quality of Service (QoS) guarantees cannot be provided to threads and Service Level Agreements (SLAs) are very difficult to enforce.

Chapter 3
Proposed Scheduling Policies

Shared resources of a CMP system are managed exclusively in hardware, meaning that they treat the requests from different threads running simultaneously on neighboring cores as they were requests from one single source. In addition, state-of-the-art process scheduling algorithms enforced initially on single-core or SMT architectures and later implemented without modifications on CMP platforms, consider the cores as isolated units and do not take into account the contention occurring when different threads compete for the shared resources. This thread and contention unaware behavior leads to poor system throughput, unfairness and unpredictable/workload dependent performance.

In this chapter we describe 3 scheduling policies, aiming to mitigate the unpredictable and unstable performance the state-of-the-art schedulers, like CFS, dictate. Driven by the unfair treatment these thread-unaware schedulers enforce, we focus on implementation of techniques that attempt to share the resources evenly among the application of the system. On the first hand, we propose a scheduler that picks combinations of applications that do not interfere with each other, in order to provide fairness via contention avoidance. On the second approach, we try to increase running time of heavily suffering applications at the expense of the well-performed applications, in order to increase their progress and overall equalize the workload performance. Finally, we extend the second solution, building a scheduler that takes into account the bad consequences of some pairs of applications, trying to avoid a part of contention and manage the rest of it, providing in that way a more efficient solution concerning throughput.

3.1 First approach (Avoiding contention)

A promising solution for addressing the contention of shared resources and its destructive effects on the co-running applications is the contention-aware thread-level scheduling. As mentioned before, performance of threads is subjected to the level of contention generated when running together. Different combinations of threads compete for shared resources to different extents and as such suffer different levels of performance loss. Some combinations compete less aggressive than others. In this approach, we attempt to mitigate the phenomenon of unfair distribution of the resources via contention avoidance. Our purpose is to build a contention-aware thread scheduling policy which would determine which threads should be placed together and which are placed far apart in such a way to minimize the effects of resource contention. Thus applications would execute in a less contended environment, experiencing higher performance and fairer resource sharing.

Mitigation of contention effects through thread-to-core mappings is a very attractive solution, as the mechanism enforcement requires no changes to the hardware and minimal changes to the operating system. Modern operating systems allow binding of threads to cores from user space via system calls with no modification to the kernel.

Taking into consideration the vast number of different thread-to-core mappings, it is impossible to perform online all the execution combinations in order to decide which leads to the best result. Contention-aware schedulers are composed in three stages. On the first stage they observe the activity of programs and *classify* them based on a particular performance metric. On the next level they *predict* how programs from different classes interfere with each other when they run together. On the last stage, they are ready to enforce their *decision-making* policy, choosing the thread combination which leads to the desired performance according to the prediction model.

3.1.1 **CMP architecture and thread placement**

Before describing the building stages of our contention-aware scheduler, it is worth mentioning the significance of the different CMP architectures on the thread placement combinations. Most contention-aware schedulers proposed on research attempt to address the problem of contention on multicore systems with components shared among a small set of cores, usually two ([12], [13], [14]) . Cores are placed into different sockets and each socket is dedicated to a specific memory area managed by an integrated memory controller (NUMA). Cores of the same socket share pairwise a Last Level Cache. Opposite to this kind of sharing, modern architectures have all of their cores (4 or 8) sharing the same LLC in the socket (Figure 3.1-1).

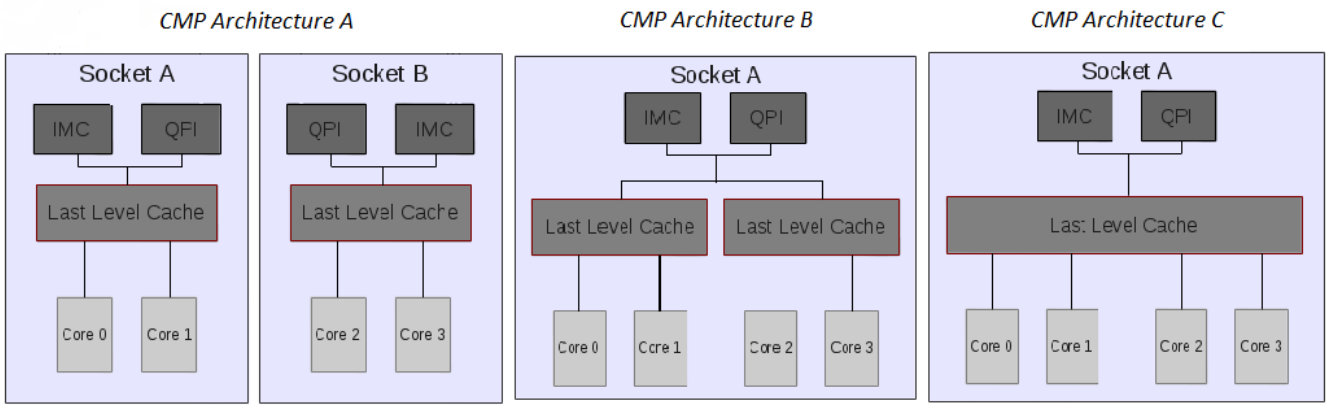


Figure 3.1-1: Structure of 3 different CMP architectures

Supposing we have a workload of 4 applications (A, B, C, D), we have to test all the different combinations produced by the architectures, in order to find the appropriate thread-to-core mapping that leads to the best possible performance. Although architectures A and B, having different structure, they have the same subset of cores sharing a component and thus generate the same combinations. The order of placement of threads on the pair cores does not matter, as (A to core 0, B to core 1) results in the same execution with (B to core 0, A to core 1). In addition, the pair of threads can be placed to any pair of cores resulting in the same execution. The mapping (C to core 0, D to core1, A to core2, B to core3) is redundant as it is the same with this mapping (A to core0, B to core1, C to core2, D to core3). (If we utilize two

sockets including many pairs of cores sharing a cache, this mapping has to be taken into account, as the contention for DRAM controller and bus is different. For example, having two sockets with two LLC each and 2 cores on each cache, the (AB CD) (EF GH) and (AB EF) (CD GH) could cause different behavior on the execution of this workload). With these restrictions the different mappings of these 4 applications to cores on Architectures A and B are 3. On the other hand, on Architecture C there is only one unique mapping of this workload, as all cores share the same component and different placement would lead to the same execution.

In general all the possible unique mappings of n threads (same to the set of cores) to cores that per k share a cache (in the same socket) is given by the following form:

$$Map(n,k) = \frac{\prod_{i=0}^{i<n/k} C(n-k\cdot i,k)}{\left(\frac{n}{k}\right)!}, \frac{n}{k} \geq 1, C(n,k) = Binomial\ Coefficient\ (form\ 3.1.1)$$

Taking another example of 8 applications on 8-core architectures that per 2, per 4 and all cores share a LLC in a socket, the number of different combinations for executing this workload is respectively $Map(8,2) = 105$, $Map(8,4) = 35$, $Map(8,8) = 1$. It is obvious, that increasing the number of cores sharing a component, the search space for finding the best co-running combination is significantly reduced. With multiple shared LLCs we have the ability to multiplex the applications in space (space-sharing), deciding how they should be distributed in a time interval: which threads will be scheduled to neighboring cores and which wil be scheduled to distant cores. That is the reason why we have a high number of generated combinations. On the other hand, the ability of space sharing the CPU is weakened, when fewer LLCs are shared between cores, reaching zero space sharing for 1 LLC. This means that threads causing high levels of contention can be placed apart only via time-sharing, executing them to different time intervals, in order to avoid contention and bad performance. And with workloads containing a normal number of contending threads, there is no way of avoiding completely their co-execution only by time-sharing. Taking for example a workload of 8 applications half of them contending more aggressively when co-running (**A**, B, **C**, D, **E**, F, **G**, H). We execute this workload to Architecture B and C. Architecture B offers the opportunity for time and space sharing, opposite to Architecture C which offers only time sharing. A possible co-execution on these different architectures could be the following:

	Time slice 1	Time slice 2
Architecture B	(**A**B) (**C**D)	(**E**F) (**G**H)
Architecture C	(**A**B**C**D)	(**E**F**G**H)

Table 3.1-1: Pairing of applications between different architectures

We come to the conclusion that Architecture B makes it possible to avoid bad co-runners, comparing to Architecture C which is forced to place them together and is obliged to generate combination of high levels of contention.

Summarizing, working with CMP architectures that all of their cores are sharing the

same component (LLC or DRAM controller), the choices we have for finding the best thread-to-core mapping are severely limited. Without space-sharing, we are obliged to co-schedule threads in a highly contended environment by avoiding contention only via time-sharing. Executing threads on such conditions, the potential for further improvement is extremely restricted. We evaluate our scheduling policies on a CMP platform that shares a LLC on 4 cores in each package, desiring to inspect how fairness can be enforced to architectures that provide with restricted thread placement options.

3.1.2 **Experimental CMP Platform**

In this paper, we are testing our scheduling techniques on an Intel® Xeon® Processor X5560 Nehalem architecture, consisted of two processor chips (sockets), each of them including several functional parts within a single silicon die. A Nehalem chip consists of the following components [15]. The core domain:

- Four identical compute cores of 2.8 GHz processor base frequency,
- L2 Cache: Each core has a private, non-inclusive, 256KiB, 8-way set associative unified level 2 (L2) cache,
- L1 Cache: At level 1 each core has separate instruction and data caches. They are private, non-inclusive and unified. Each of them has a size of 32 KiB. The instruction and data caches have 4-way and 8-way set associativity organization, respectively.

The un-core domain:

- L3 Cache: A unified, inclusive, 16-way set associative, 8 MiB cache shared by all four cores of the chip
- UIU: Un-Core Interface Unit (switch connecting the 4 cores to the 4 L3 cache segments, the IMC and QPI ports),
- L3: level-3 cache controller and data block memory,
- IMC: 1 integrated memory controller with 3 DDR3 memory channels,
- QPI: 2 Quick-Path Interconnect ports,
- Auxiliary circuitry for cache-coherence, power control, system management and performance monitoring logic (both core and un-core domain).

The L3 is inclusive (unlike L1 and L2), meaning that a cache line that exists in either L1 data or instruction, or the L2 unified caches, also exists in L3. The L3 is designed to use the inclusive nature to minimize “snoop” traffic between processor cores and processor sockets.

A Nehalem chip is divided into two broad domains, namely, the “core” and the “un-core”. Components in the core domain operate with the same clock frequency as that of the actual computation core. The un-core domain operates under a different clock frequency. This modular organization reflects one of Nehalem’s objectives of being able to consistently implement chips with different levels of computation abilities and power consumption profiles.

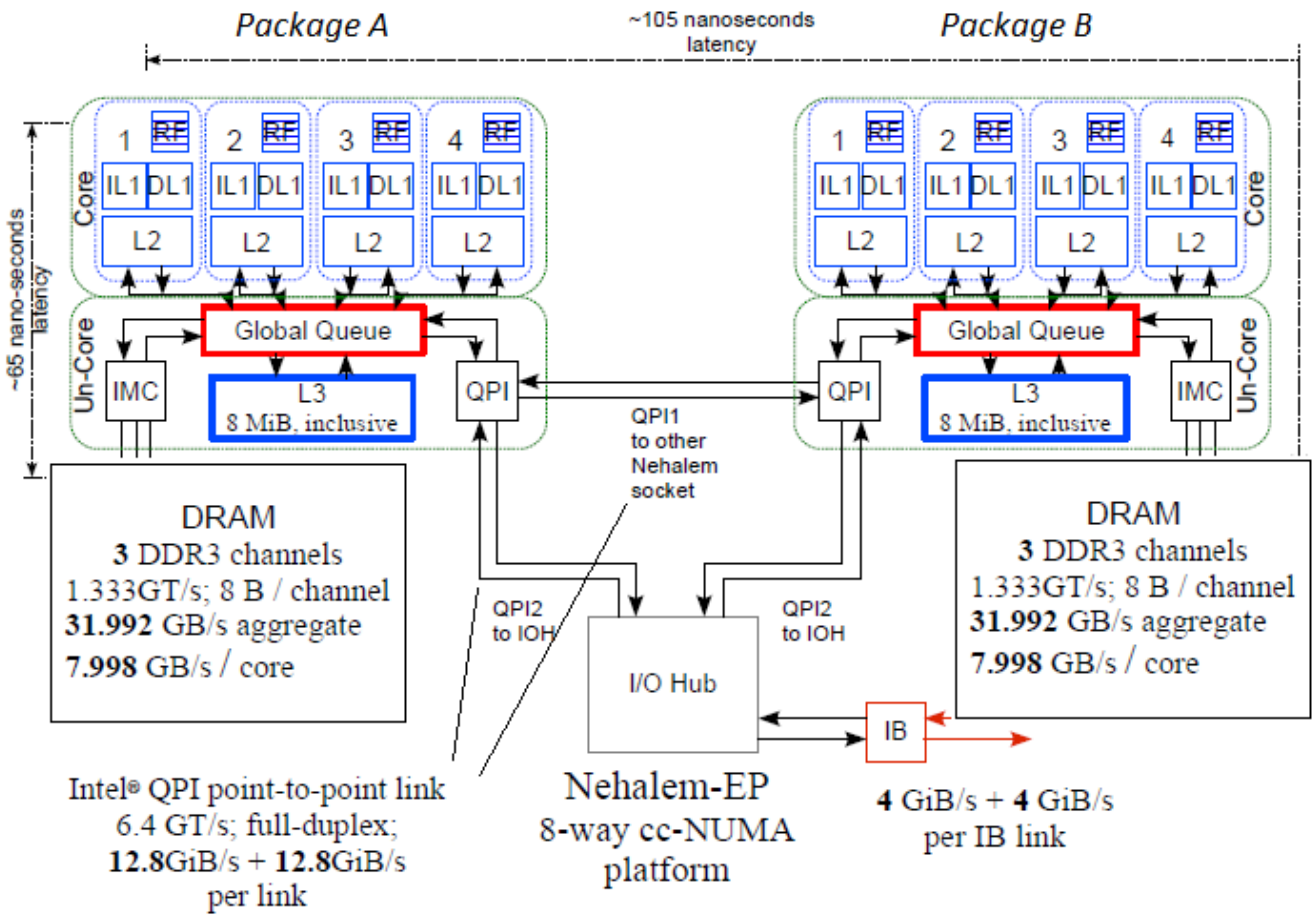


Figure 3.1-2: Nehalem architecture details

Each chip is dedicated to specific memory areas, accessible via the IMC, which supports three 8-byte channels of DDR3 memory operating at up to 1.333 GT/s. Maximum theoretical bandwidth between DRAM and the IMC in the uncore domain of the chip is 31.992 GB/s.

Communication between the two chips is achieved via the QPI link with available bandwidth of 12.8 GB/s. Memory latency for a remote memory access is higher, since the memory request and response must go through this QPI link. The latency to access the local memory is, approximately, 65 nanoseconds. The latency to access the remote memory is, approximately, 105 nanoseconds.

A Nehalem chip supports:

- Hardware Prefetching Logic, for requesting data which threads will use in the near future.
- "Simultaneous Multi-Threading" (SMT). SMT is an implementation which allows more than one hardware threads (two threads for Nehalem) to execute simultaneously within each core,
- Power saving features ("Turbo Boost Technology") for dynamically turning off unused processor cores and increasing the clock speed of the cores in use (when thermal and electrical requirements are still met.

Our experimental setup includes one of the Nehalem chips with the four cores sharing a LLC, aiming to test our scheduling techniques on highly contended condition. The main memory shared between the two chips is a 12 GB DDR3 1333 MHz's The hardware prefetching logic is enabled, whereas the SMT and TBT features are disabled. The platform runs Ubuntu 12.04.2 LTS.

3.1.3 **The Classification Method**

A contention-aware scheduler is responsible for determining, given a workload, on which time slice the applications should run (time-sharing) and which cores they should occupy at this time interval (space-sharing). In our case space-sharing is not a possible option. Even with time-sharing only, the different possible schedules of a 12-applications workload executed on our platform are 5.775 (form 3.1.1). This vast number of combinations makes dynamic trial and error exploration infeasible. As such, it becomes necessary for a contention-aware scheduler to be able to predict the performance of different mappings without actually trying them.

In order to establish a prediction model, firstly we have to record the behavior (which resource they utilize and at what extent) of the applications when executing alone. Applications with common characteristics can be grouped together forming a class. Following a classification scheme, applications can be divided into different classes, according to their characteristics. In this paper we categorize applications following the classification method proposed by Haritatos et al [1]. Necessary information for the programs' performance is acquired via the performance monitoring facilities available on the Nehalem processor. In the next paragraph, we briefly describe the performance monitoring, before presenting the classification method.

The step after the classification of applications to different classes is to examine how they interact with each other. Testing all the different combinations, we determine how applications of a class behave when executing with those of another class, the level of occurring contention and at what extent each class is impacted by this contention. Now that we know how applications of different classes contend with each other and what influence this contention has on their performance, it is possible for a prediction model to be adopted.

Following a model that can predict the performance of the different mappings, the scheduler is able to decide which applications should co-execute every time interval for achieving the desired objective.

Performance Monitoring

As explained previously, in order to classify the applications to different classes, we need to acquire information about their profile. In addition, it is very useful to collect performance data from their co-execution with applications of other classes, in order to understand how contention changes their behavior. Modern processors, like the one described before, provides hardware performance counters on both core and un-core domain for recording the activity of the cores and the memory system [16].

Once deciding which data we want to collect, selecting the proper events, we initialize the corresponding counters. Then, we start counting at the moment that the desired application is running for a specific time interval. After this interval, we read the value stored in the counters. Finally the requested data are available and with the proper interpretation we can use them to reach to great conclusions.

Counters, that provide us with useful information for classifying the applications and verifying the proposed prediction model, are listed on the table below accompanied with necessary information used for their initialization.

Name	Number	Mask	Type	Description
EVENT_UNHLT_CORE_CYCLES	0x3C	0x00	PMC	Clock Cycles of Unhalted Core
EVENT_INSTR_RETIRED	0xC0	0x00	PMC	Instructions Retired
EVENT_LLC_MISSES	0x2E	0x41	PMC	Last Level Cache Misses
EVENT_L1D_REPL	0x51	0x01	PMC	Cachelines allocated in the L1
EVENT_L1D_M_EVICT	0x51	0x04	PMC10	Modified Cachelines evicted from the L1
EVENT_L2_LINES_IN	0xF1	0x07	PMC	Cachelines allocated in the L2
EVENT_L2_LINES_OUT_DEMAND_DIRTY	0xF2	0x02	PMC	Modified Cachelines evicted from the L2
EVENT_UNC_L3_LINES_IN	0x0A	0x0F	UNCORE	Cachelines allocated in L3
EVENT_UNC_L3_LINES_OUT	0x0B	0x1F	UNCORE	Modified Cachelines evicted from the L3

Table 3.1-2: Performance monitoring events

Application Classes

Our classification method separates applications into 4 different categories [1]. The criteria chosen for their characterization are based on which of the resources they utilize and at what extent. Using this model, we our able to accurately locate which part of the shared resources, both memory bus bandwidth (memory link) and LLC, is stressed the most and thus is prone to contention. Our approach uses simple metrics that can be easily and quickly collected at runtime from the existing monitoring facilities of modern processors. Concluding, this classification scheme provides us with the possibility of adopting a highly reliable

prediction model, while the classes represent in a very clear way the behavior of applications (specific resource utilization) and how it is affected when interfering with each other.

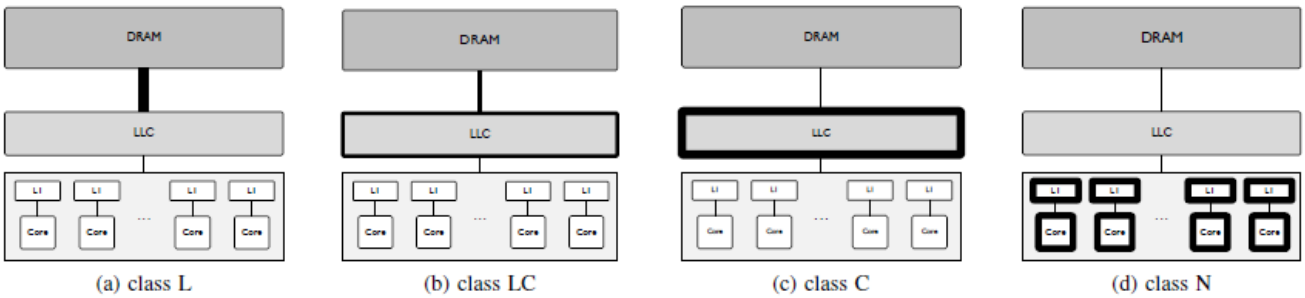


Figure 3.1-3: Activity in application classes

L Class

Applications with streaming nature tend to have a stable data flow throughout the entire memory hierarchy. They fetch data from the main memory directly to cores at a high rate, contaminating the memory bus (memory link), all the caches and their connections (cache links). Responsible for this behavior are their memory accesses, which occur to data sets that largely exceed the size of LLC. They bring data to LLC but the reference on those data is either not occurring at all or is happening with large reuse distances. So, data are descending down to the cores at the same rate they are brought to LLC, resulting in zero benefit from the existence of this shared resource. But more importantly, the continuing pollution of the LLC with their data cause great destructive interference to co-running threads that benefit from the reservation of their data in this shared resource. Providing that the determining factor for their performance is the memory bandwidth utilization, contention to this shared resource can cause them severe degradation.

LC Class

On this class belong applications with modest pressure on the memory link and modest to low activity on the shared LLC. They present common characteristics with the L class imitating their streaming nature, as they fetch data from the main memory at a satisfying rate, but the flow of data is not the same on the rest path. They take advantage of their reservation in the LLC, reusing them for a while before bringing the next group of data. This kind of applications cause contention to the memory bandwidth and the LLC, affecting negatively the applications with important activity on this shared resources. As long as they take advantage from both memory bandwidth and LLC utilization, their performance gets negatively affected by applications that contaminate these shared resources.

C Class

On this class belong applications with high activity on the shared LLC. They bring data to cache at a really low rate and perform heavy reuse on the cached data. This class includes applications with varying characteristics, such as those with data sets not exceeding the size of the LLC or with memory access patterns optimized for taking advantage of the LLC and latency-bound applications that make irregular data accesses and benefit a lot from LLC hits.

N CLASS

This class consists of applications that take advantage exclusively of the lower parts of the memory hierarchy, such as private caches and cores. Applications, which perform heavy computations, have working sets that are small enough to fit the private (L1 or L2) caches or have optimized memory access patterns that can be serviced by the private caches, do not extend their activity further on the shared resources. This means that they do not pollute the LLC or the memory bus with their data, causing destructive interference to threads running on neighboring cores. Except that, their performance is not affected by the contention caused by their co-runners. Concluding, applications, that neither suffer nor cause suffering from/to others, form the N class. Figure 1.6-3 indicates the activity spot of each class.

Classification scheme

Once the application classes are defined, we need to adopt a method to perform classification using runtime statistics. The idea is to inspect the data path from main memory to cores [1]. We are focused on the stream flowing towards the cores, recording the occurring activity on memory and cache links. Thus, for each application it is necessary to measure the flow of data from main memory to LLC, from LLC to private cache and between private caches as depicted in figure. (Bin_i, incoming Bandwidth to i level of memory hierarchy, $0 < i \leq n, n = last\ level$). Now that we know the utilization of the links across the entire path, we have to specify on which resource the pressure is high. The ratio $CR_i = \frac{Bin_{i-1}}{Bin_i}$ is a good indicator for the pressure put on a resource, as it follows the simple rationale. If data flow out of the source with a much higher rate than they flow in, we can safely assume that high activity occurs on this resource, as it is heavily reused.

The information required for this metrics can be easily collected at runtime by the performance monitoring facilities of modern processors. Once assigning the application on the core, we choose the appropriate counters and start monitoring, as described before. In our case, for the bandwidth computation the events that count the cache lines allocated to each cache should be activated. We do not take into account the number of modified (dirty) cache lines evicted from the cache, as long as they overestimate the incoming bandwidth. Events counting allocated lines in a cache also include data transfers due to a write allocate load on a store miss in that cache. For our platform the bandwidth between the different levels of the memory hierarchy is computed by the following forms

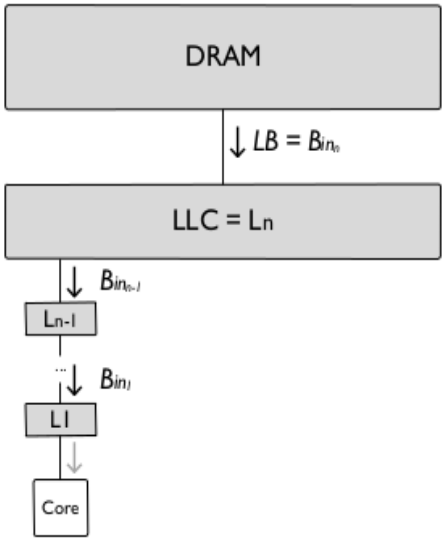


Figure 3.1-4: Inspected data flow in the memory hierarchy

The memory hierarchy of our platform consists of 3 cache levels, two private and one shared. So the Bin_3, Bin_2, Bin_1 refers to memory → LLC , LLC→L2 , L2→L1 bandwidth respectively. The ratio $CR_3$and CR_2corresponds to the LLC and L2 reuse. The block size is 64 B and the measurement unit of bandwidth is [MB/s].

$$Bin_3 = \frac{(EVENT\_UNC\_L3\_LINES\_IN) \cdot 64 \cdot 10^{-6}}{time}, \qquad CR_3 = \frac{Bin_2}{Bin_3}$$

$$Bin_2 = \frac{(EVENT\_L2\_LINES\_IN) \cdot 64 \cdot 10^{-6}}{time}, \qquad CR_2 = \frac{Bin_1}{Bin_2}$$

$$Bin_1 = \frac{(EVENT\_L1D\_REPL) \cdot 64 \cdot 10^{-6}}{time}$$

Now that we acquired all the appropriate metrics we can perform our classification. High bandwidth utilization across all links and zero cache reuse means that the application put high pressure on the links only, experiencing streaming behavior and can be classified as L. Other with satisfying activity on the memory → L3 link and higher activity on the L3 → L2 link ($CR_3 > 1$) belong to LC class. Applications with low memory bandwidth activity and significant data flow somewhere in the LLC → core part of the data path reuse heavily either the LLC or the private caches. Locating the reuse location we can classify as C and N respectively.

In case that low data flow is measured throughout the entire data path, we have identified three application patterns that may exhibit this picture [1]: a) applications that heavily reuse data on the L1 cache (high activity on the gray arrow of Figure 1.6-4), b) applications that perform computations within the core with minimal data accesses, and c) memory-latency bound applications that suffer from high LLC miss penalties (and also greatly

benefit from LLC hits). Although we may utilize various combined metrics involving cache hit ratios to illuminate this further, we have observed that inspecting the application's mem uops/all uops ratio and IPC suffices to distinguish a) and b) cases, which are N applications, from the c) case, which includes C applications. High mem uops/all uops and low IPC is a quite safe indicator for memory-latency bound applications (at the absence of significant data flow in the path).

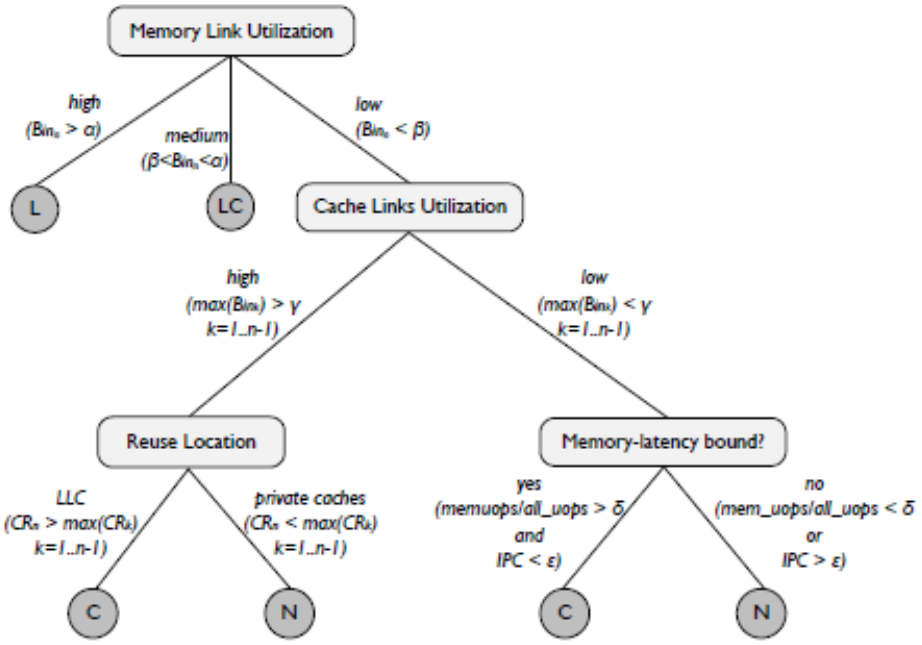


Figure 3.1-5: Decision tree for application classification

Our classification method implements the decision tree shown in Figure 3.1-5 [1]. In each execution platform we need to set five thresholds, namely α, β, γ, δ and ε that guide the classification process. In this platform we set $= 0.12 \times B_{max}$, $\beta = 0.045 \times B_{max}$, $\gamma = 0.068 \times B_{max}$, $\delta = 0.25$, $\varepsilon = 0.25 \times IPC_{max}$. The maximum memory link bandwidth as measured by the stream benchmark is $B_{max} = 13.20\, GB/sec$ and $IPC_{max} = 4$.

C class analysis

An application's data set determines its behavior. Their characteristics do not remain unharmed, when they are executed with different sizes of their working set. As depicted on the table below, the same benchmark [3] could belong to any class of our scheme, depending on its size.

With a data set half of the size of our platform's L2 private cache (256 KB), the activity

of the chase benchmark is restricted down to the core domain. The reference is occurring mainly on the L2 (CR2 >> CR3) and the application is classified as N. Increasing the size to 250 KB we notice that pressure on the L2 cease (CR2 = ~ 1) and start occurring on the LLC (high CR3), while the data set barely fits to the L2. From now on the application is satisfied by the LLC and is classified as C. Increasing the size more and more, we observe that CR3 decreases dramatically and the reference starts occurring on the main memory too (higher bandwidth and MPI). At size of 7 MB we reach the point where the data set barely fits the LLC of 8 MB size and the pressure on memory link becomes heavier, changing the behavior of this application to LC. From now on, the application will experience intense activity on memory link and zero activity on LLC and is on the edge of changing behavior to L.

Data Set (KB)	IPC	MPI	B3 (MB/s)	CR3	CR2	Class
125	0.3167	0.00000061	0.464	49.114	880.992	N
250	0.1420	0.00000079	0.676	8,384.399	1.477	
500	0.1139	0.00000109	0.474	14,144.102	1.003	
1,000	0.1141	0.00000092	0.747	9,001.215	1.000	
2,000	0.1144	0.00000164	0.813	8,260.022	1.009	C
3,000	0.1126	0.00000388	1.490	4,485.029	1.002	
4,000	0.1119	0.00005670	3.895	1,709.589	1.007	
5,000	0.0966	0.01351051	251.686	22.795	1.001	
6,000	0.0819	0.03272887	491.220	9.920	1.000	
7,000	0.0590	0.08496037	905.713	3.873	1.001	
8,000	0.0418	0.16437671	1,238.942	1.979	1.000	
9,000	0.0370	0.19934837	1,326.267	1.652	1.000	
10,000	0.0329	0.23939016	1,419.495	1.353	1.000	LC
12,000	0.0292	0.28887063	1,515.268	1.129	1.000	
14,000	0.0283	0.30526646	1,552.032	1.063	1.000	
16,000	0.0282	0.31001331	1,581.033	1.026	1.000	

Table 3.1-3: Classification of pchase across different data sets

However, not all applications change their behavior in the same way. Inspecting, for example, the mvt benchmark, we observe that the transition from C to LC class is occurring at a bigger data set compared to the pchase benchmark, as shown on the Figure 3.1-6 below.

Increasing the data set to a size bigger than the size of LLC, the mvt benchmark remains to the C class. This means that its access pattern do not take advantage of the whole data set but a part of it. While the data set exceeds the LLC, the access restricts to data that fit the LLC and the reference on the main memory is occurring at a low rate. On the other hand, pchase benchmark takes advantage of the whole data set. When it does not fit the LLC, reference on the main memory becomes really intense.

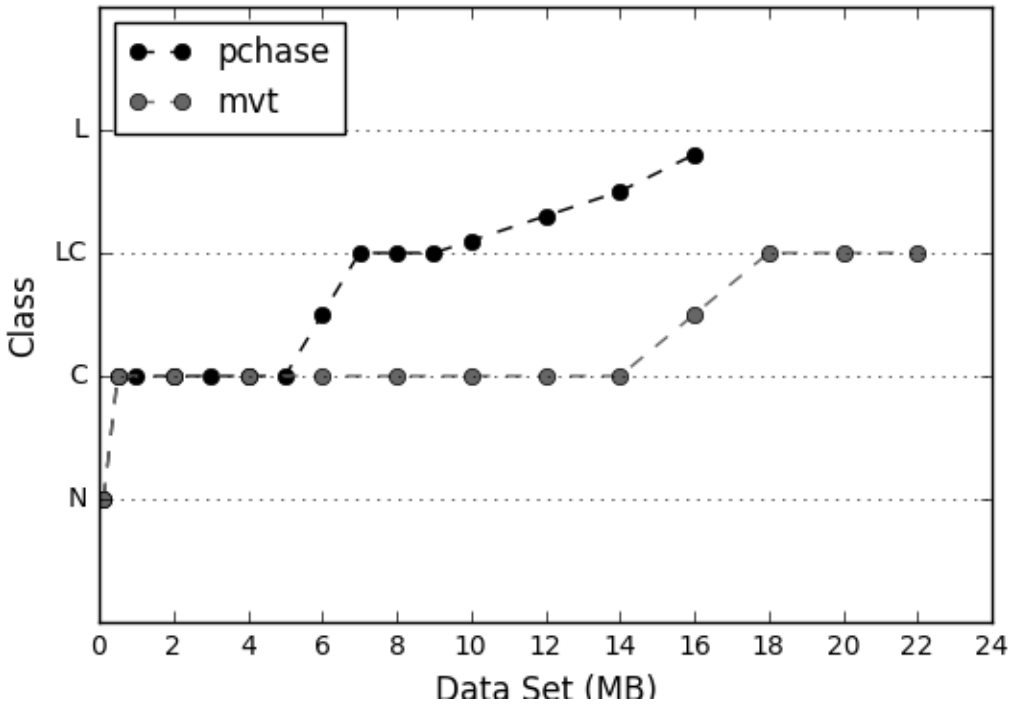


Figure 3.1-6: Behavior of pchase and mvt applications across different data sets

We come to the conclusion that applications imitating the behavior of pchase benchmark require the majority of the available shared LLC capacity to perform well. When they get restricted, meaning that they do not receive the space their data set implies, their misses present a major increase and their performance is significantly degraded.

Based on this observation, we understand that, while the data set of these applications (pchase) do not exceed the LLC size, it can be translated to working set, as it can be completely served by the LLC. The same cannot be assumed for applications imitating the mvt behavior, as their working set is a subset of their data (data set = 12 MB, working set < 8 MB).

Executing on a multi-core system, where 4 cores share an 8 MB LLC, 4 applications with working set bigger than 2 MB, it is made clear that they cannot be served efficiently by the LLC, as their total working set would exceed its size. On this case, interference in the LLC would occur, leading to destructive effects on their performance. In general, on an N core system, where the cores share an L MB size LLC, applications with working set bigger than $L/N\ MB$ are considered to interfere destructive with each other.

Taking the previous observation into consideration, we extend the classification scheme proposed by Haritatos et al. [1], distinguishing further the C class into 2 subclasses, the SC and BC.

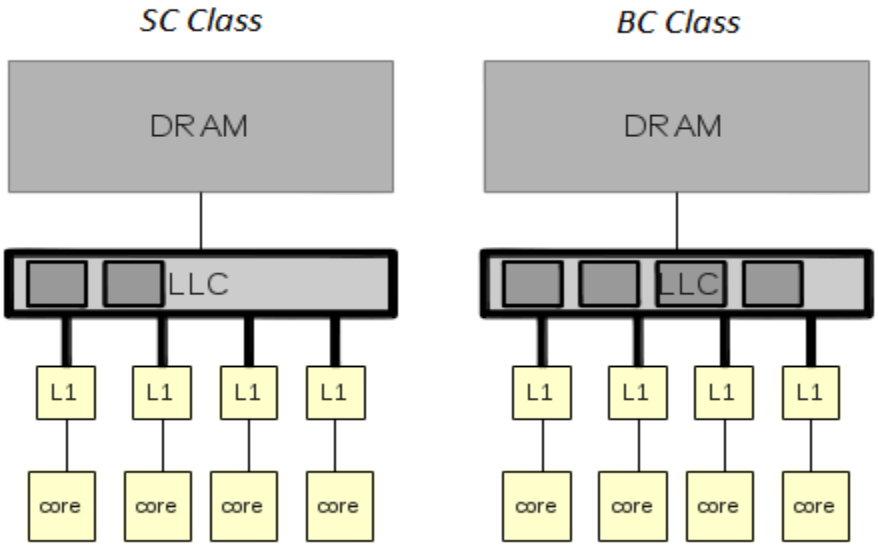


Figure 3.1-7: Activity of the C subclasses

SC Subclass

Applications operate on data with size occupying a small part of the shared LLC (< 2 MB). Such applications benefit from the available shared cache, performing heavy reuse, but their performance does not degrade significantly when they do not receive the bulk of its space. While their activity occurs on a restricted LLC area, interference on this resource generated by other thrashing applications will cause them moderate damage.

BC Subclass

Applications with data occupying the biggest or a satisfying part of the LLC space (> 2 MB), also performing heavy reuse on the cached data. As they require the majority of the available shared LLC capacity to perform well, they get severely degraded when high contention level occurs on this resource. The only way to experience good performance is, when running in isolation.

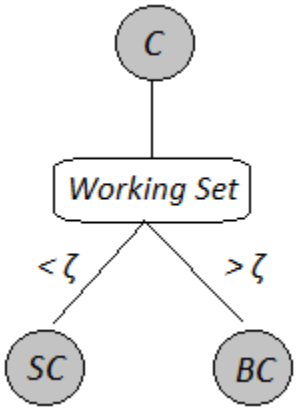


Figure 3.1-8: C class distinction in the decision tree

We modify the decision tree (Figure 3.1-8) to correspond to the C class distinction described previously; adding to the C leaves the graph shown in the Figure 1.6-8. We define the new threshold $\zeta = LLC\_size/cores\_nr$ which is equal to 2 MB for our platform (8 MB LLC size and 4 cores).

Finally we have all the information required to perform our classification. Although we have the ability to classify applications into these 4 different categories via simple metrics collected by hardware performance counters at runtime, it is difficult to go deeper into the C class and acquire information about size of data allocated in the LLC. This requires additional hardware support [17] or complicated software techniques [18]. Inevitably, the only way to distinguish applications of C class into the subclasses is by knowing beforehand their working set. In our case, we inspect the behavior of benchmarks across different data set sizes. If they imitate the pchase benchmark behavior, it is possible to extract information about their working set, as explained previously. We compile those applications with data set greater than 3 MB and smaller than 7 MB (equal to working set), in order to extract the BC applications. Applications that imitate the mvt benchmark behavior are compiled with 2 MB data set (working set < 2 MB), in order to populate the SC class.

Summarizing, we chose to adopt the classification scheme described in the LCA paper [1], in order to observe how applications of different categories interfere with each other and enforce a scheduling policy to avoid interference and improve their performance. We inspected the behavior of applications compiled with different data sets. Based on our observations, we further extended the classification of C class into two different categories, the SC and BC, according to the working set. While the classification into the 4 categories can be accomplished at runtime via simple metrics, acquiring information from the performance counters, it is impossible, following the procedure described previously, for the C applications to be deeper distinguished into the two subclasses online. Acquiring information for the working set at runtime requires complicated software techniques, something that is out of the scope of this paper. However, the scheduling policy developed for those subclasses can be easily applied with whichever method that classifies online the C applications deeper.

Benchmark Selection

The benchmarks populating our classes are selected mostly from polybench suite. We use single threaded applications with single execution phases and low I/O operations. We compile them with large data sets (far exceeding the LLC) for generating L and LC classes and with small data sets fitting the L2 for generating the N class. We inspect the behavior of each benchmark across different data sets and compile those that imitate the pchase benchmark, with data set of size between 3 and 7 MB creating the BC class. Finally the SC class is populated by benchmarks compiled with 2 MB data set.

Name	Source	DataSet (MB)	$B_{in3=LLC}(MB/s)$	CR3	CR2	IPC	Class
cholesky	polybench	128	3112.835	1.009	1.355	0.838	
jacobi-1d	polybench	48	3932.853	1.013	1.002	0.558	
2mm	polybench	66	2276.183	2.430	1.790	0.256	
fw	polybench	64	2292.729	1.011	1.043	0.913	
stream	[2]	366	6618.432	0.994	1.003	0.721	L
atax	polybench	72	4401.339	1.029	1.759	0.380	
syr2k	polybench	34	2997.791	1.022	1.184	0.881	
trmm	polybench	144	2418.043	1.002	1.459	0.859	
jacobi-2d	polybench	12	1124.768	1.112	1.071	1.057	
bt	NAS	15	1450.758	2.081	1.112	0.897	
gemver	polybench	125	1173.694	4.299	1.315	0.402	LC
mvt	polybench	125	996.640	5.597	1.171	0.279	
pchase	[3]	4	3.895	1,709.589	1.007	0.111	
pchase	[3]	5	251.686	22.795	1.001	0.096	
pchase	[3]	6	491.220	9.920	1.000	0.081	
stream	[2]	4	9.87	544.98	1.000	0.960	
stream	[2]	5	99.691	51.732	1.000	0.921	BC
jacobi-1d	polybench	5	88.765	24.154	1.000	1.101	
jacobi-1d	polybench	6	328.559	5.937	1.000	1.001	
jacobi-2d	polybench	5	113.177	16.152	1.012	1.544	
jacobi-2d	polybench	2	0.807	2378.055	1.000	1.621	
jacobi-1d	polybench	2	0.974	2284.703	1.000	1.145	
pchase	[3]	2	0.246	27344.115	1.000	0.114	
correlation	polybench	2	0.706	1275.893	15.874	1.334	
covariance	polybench	2	0.714	1255.327	15.960	1.332	SC
ludcmp	polybench	2	0.763	1869.429	1.330	1.512	
symm	polybench	2	0.820	1815.967	8.979	1.355	
stream	[2]	2	0.478	11280.28	1	964	
syrk	polybench	0.064	0.632	0.791	304.735	2.392	
doitgen	polybench	0.064	0.681	0.493	346.346	2.311	
trisolv	polybench	0.064	0.609	0.762	2021.165	0.813	
3mm	polybench	0.064	0.760	1.983	72.710	2.250	N
gesummv	polybench	0.064	0.656	1.183	1300.079	1.160	
bicg	polybench	0.064	0.670	2.194	504.134	1.180	

Table 3.1-4: Classification of benchmarks selected for the population of our workloads

3.1.4 The Prediction Model

Another crucial building block of a contention-aware scheduler is the interference prediction model. Having characterized the applications into different categories, following a classification scheme, it is important to establish a model that accurately predicts how their performance degrades as they share resources. Knowing beforehand the co-execution effects between applications of different classes, it is feasible for a scheduler to find a thread placement policy without performing trials of the sheer number of combinations generated by the applications of each workload.

Prediction and Evaluation of inter-class interference

For each category of our classification scheme we describe how performance of applications is expected to be affected when co-executing with applications of other classes. In order to validate our prediction model, for the selected benchmarks populating our classes, we generate all the possible pairs and perform their co-execution. We collect the slowdown measurements for applications of each class and extract their average.

N Class Suffering

Performance of applications belonging to this class is not affected by co-runners of other classes. While their activity is restricted to the core domain of the CMP, there is no interference with applications experiencing intense activity on the shared resources.

L Class Suffering

Applications of this class consume a high percentage of the memory bandwidth. They get negatively affected only by applications of the same class that are capable of causing destructive interference on the memory link. When L applications run simultaneously and the sum of their bandwidth exceeds the maximum sustainable bandwidth (13.20 GB/s as measured on the example below), they are forced to perform with a fraction of their initial bandwidth (form 3.1.2). Usually the maximum bandwidth is divided unequally between the competing applications. The slowdown imposed is inversely proportional to the ratio of the experienced bandwidth.

$$BW_{total} = \sum_{i=1}^{i<5} a_i \cdot BW_i, \qquad Slowdown_i = \frac{1}{a_i} \quad (form\ 3.1.2)$$

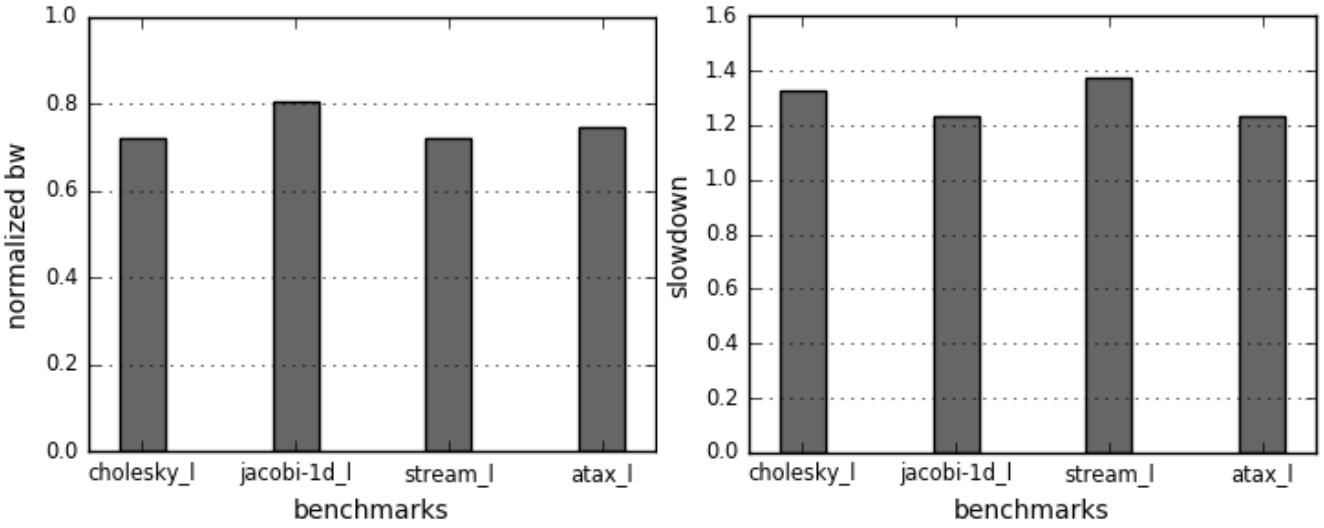


Figure 3.1-9: Slowdown imposed to L applications due to bandwidth contention

LC Class Suffering

For application with modest activity on the memory link we expect mild degradation by the L applications only. While it is possible to experience LLC reuse, interference by BC applications do not harm them. LC application trash the LLC quite often and have very low LLC reuse factor in order to get negatively affected by the BC class. In fact the opposite is happening, LC cause great pain to BC applications, as BC are greatly benefited by the residence of their data in the LLC and LC sweeps them away.

SC Class Suffering

Applications with small data set resident on the LLC perform well with all the classes except the L. Although L applications, with their heavily trashing nature, bring at a high rate data evicting SC cache lines, we do not expect devastating slowdowns, as the small amount of data being swept away can be easily and quickly retrieved. We observe heavy but not destructive damage for the SC-L pair and no slowdown for the other pairs.

BC Class Suffering

While applications of this class require the bulk of the LLC space to perform well, applications, that have LLC trashing behavior like L and LC, cause great contention and devastating damage. Severe slowdowns are expected to occur when BC applications are co-executed, as the sum of their data set exceeds the size of LLC and they force eviction of each other’s cache lines. Only the SC and N co-runners have friendly behavior, showing light and zero interference respectively.

Summarizing we present the on the figure below the average slowdown each class suffer when co-executing with the other classes. Along the x axis we show the slowdown imposed by each class, and along the y axis we show the slowdown suffered. We observe that the co-execution test validates our prediction model. L applications are affected only by themselves, as they are subjected to bandwidth contention. LC applications present a slight slowdown by the L applications only, as expected. SC applications suffer heavily by the L only, while they show LLC trashing behavior. BC applications are the most vulnerable, suffering devastating damage by L, LC and themselves. They perform the best with N and get a mild degradation with SC. N applications neither suffer nor cause suffering by/to other classes.

	L	*LC*	*SC*	*BC*	*N*
L	1.3014	1.1273	1.6744	2.4278	1.0821
LC	1.1184	1.0714	1.1192	2.2216	1.0549
SC	1.0564	1.0436	1.0783	1.4609	1.0384
BC	1.1438	1.0761	1.1608	2.3253	1.0093
N	1.0189	1.0063	1.0008	1.0644	1.0017

Figure 3.1-10: Average application slowdown due to the co-execution at the class level

3.1.5 The Decision - CMB (Cache and Memory Bandwidth contention-aware) Scheduler

Taking into consideration the prediction model described previously, we try to enforce a thread placement that avoids contention and maximizes throughput and fairness. Our primary purpose is to keep separated L class from C class (SC, BC) applications and LC class from BC class applications, as long as their co-execution would be catastrophic for the C class (x1.67 slowdown for the SC and x2.43 (L), x2.22 (LC) slowdown for the BC).

To achieve this separation, we form L and C gangs that will be executed on different time quantum. Considering that we work on one package, we have the ability of time-sharing only the CPU and the possible gangs generated are equal to $\text{progs}_{\text{nr}}/\,\text{cores\_nr}$. An L gang consists of a mix of L, LC and N applications. A C gang can be either a SC gang or a BC gang. A SC gang is filled by SC and N applications, whereas a BC gang has BC, SC and N applications. (Figure 3.1-11)

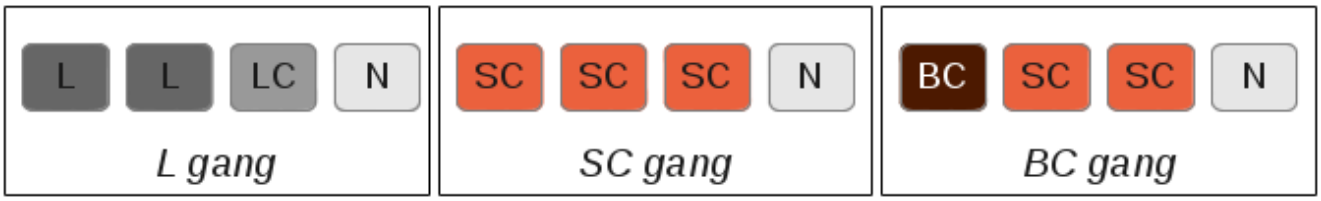


Figure 3.1-11: Desired gangs for the CMB scheduler

However isolating L from C applications is not enough. Based on the interference prediction model, the combination BC-BC causes heavy damage on the BC applications. In addition, co-executing L applications with themselves can cause severe degradation, when their total bandwidth far exceeds the maximum ($BW_{total} > 13.20\,GB/s$). Following the rules imposed by the prediction model we form the gangs in such a way, that co-execution of BC applications would be avoided and memory link pressure of the L gangs would be mitigated.

While the BC-BC pair experience higher slowdown than the L-L pair, the isolation of BC applications gets higher priority than the mitigation of the memory bandwidth.

Taking into consideration the prioritized objectives to be satisfied, our thread placement scheduling policy is described in the steps below:

1. We spread the BC applications across the gangs as much as possible in order to avoid their co-execution. We compute the maximum number of BC gangs that could be created, if interference with L applications only is avoided ($\text{total}_{\text{gangs}} - \min(\text{L\_gangs})$). If the number of BC applications exceeds the number of available gangs, we are obliged to pair them together, tolerating the destructive behavior this match causes. Now that we have isolated the BC applications, we fill their gangs firstly with N and then with SC applications (BC-SC, BC-N pairs with mild and zero interference respectively).
2. We pack together SC applications. If they cannot form whole gangs (minimum number) by themselves, we group together LC applications firstly and then N applications (SC-LC, SC-N pairs with zero interference).

3. We sort the L applications with bandwidth criterion. We add on the list the remaining LC and N applications. We fill the remaining gangs with the applications of the list.

If applications of the L, SC and BC class are completely missing or have been used in previous steps, then the step forming the respective gangs is skipped. If there are no BC applications, we skip step 1. If there are no SC applications, we skip step 2. If there are no BC and no SC applications we skip steps 1 and 2.

In the above steps we fill the gangs in the same way. We iterate over the number of selected applications and with the determined order we pack them switching the gangs on each iteration. The figure below shows this packing clearer.

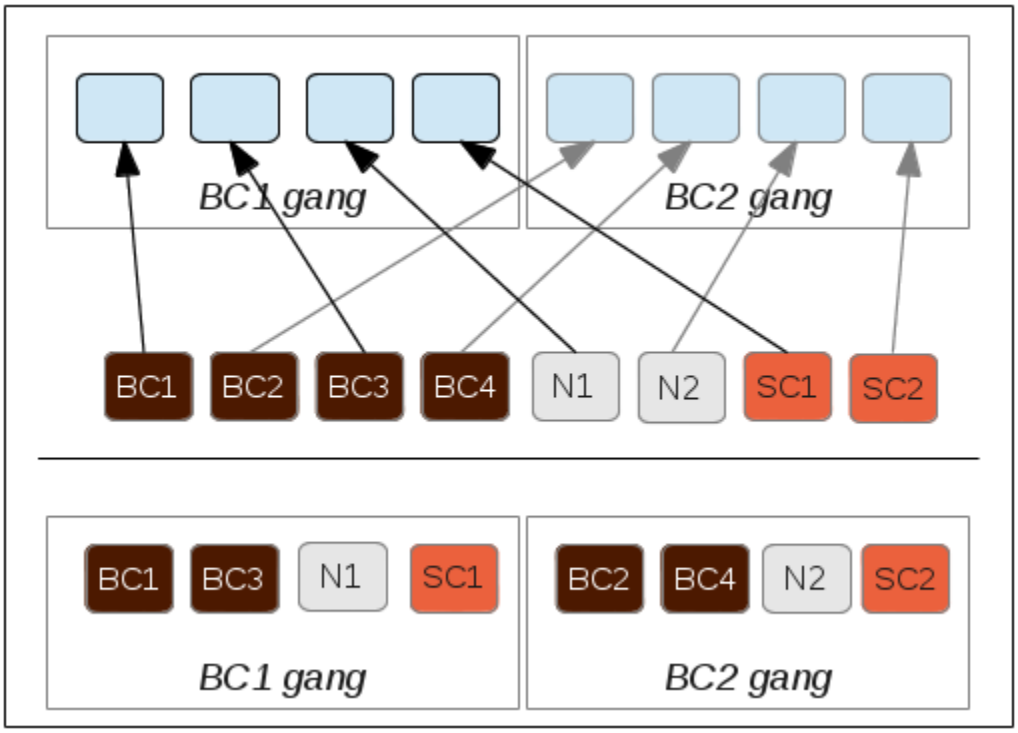


Figure 3.1-12: Formation of gangs

Following this packing, the sorted L applications are distributed equally across the gangs, as heavy L are not accumulated in one gang.

On the table below we summarize the steps we follow for the placement of our applications and the objective each step accomplish.

Steps	*Objectives*
1) maximize BC gangs and spread BC applications across them as much as possible	Avoid interference of BC with BC,LC and L applications
2) pack together SC gangs to minimize SC gangs	Avoid interference of SC with L applications
3) sort L applications and share them on the remaining L gangs	Balance the memory link utilization.

The algorithm

Our scheduler consists of 5 functions, init(), qexp(), freeze(), schedule(), thaw(). Upon selection of our scheduler the init() function is called. In this function we form the gangs. Then we are ready to start the execution. After the end of the time quantum qexp() is called. Inside this function freeze(), schedule() and thaw() are called. The gangs are scheduled in a round robin way. On our implementation, each class is represented by a list. Applications on the input are already classified. The classification can be easily carried out online, running each application alone for one time quantum. This would result in a small deviation in the execution time of N quantum comparing to Linux scheduler. In order to be as accurate as possible we perform the classification offline. The gangs are, also, implemented as lists. In each step we compute the number of gangs and fill them with the appropriate applications in a specific order. For the L gangs, the corresponding applications should be sorted. The algorithm has O(n + L log L) complexity, O(L log L) corresponds to the sorting of L applications and O(n) to the forming of gangs in a greedy way. The flow chart and pseudocode below present the implementation of our scheduling policy.

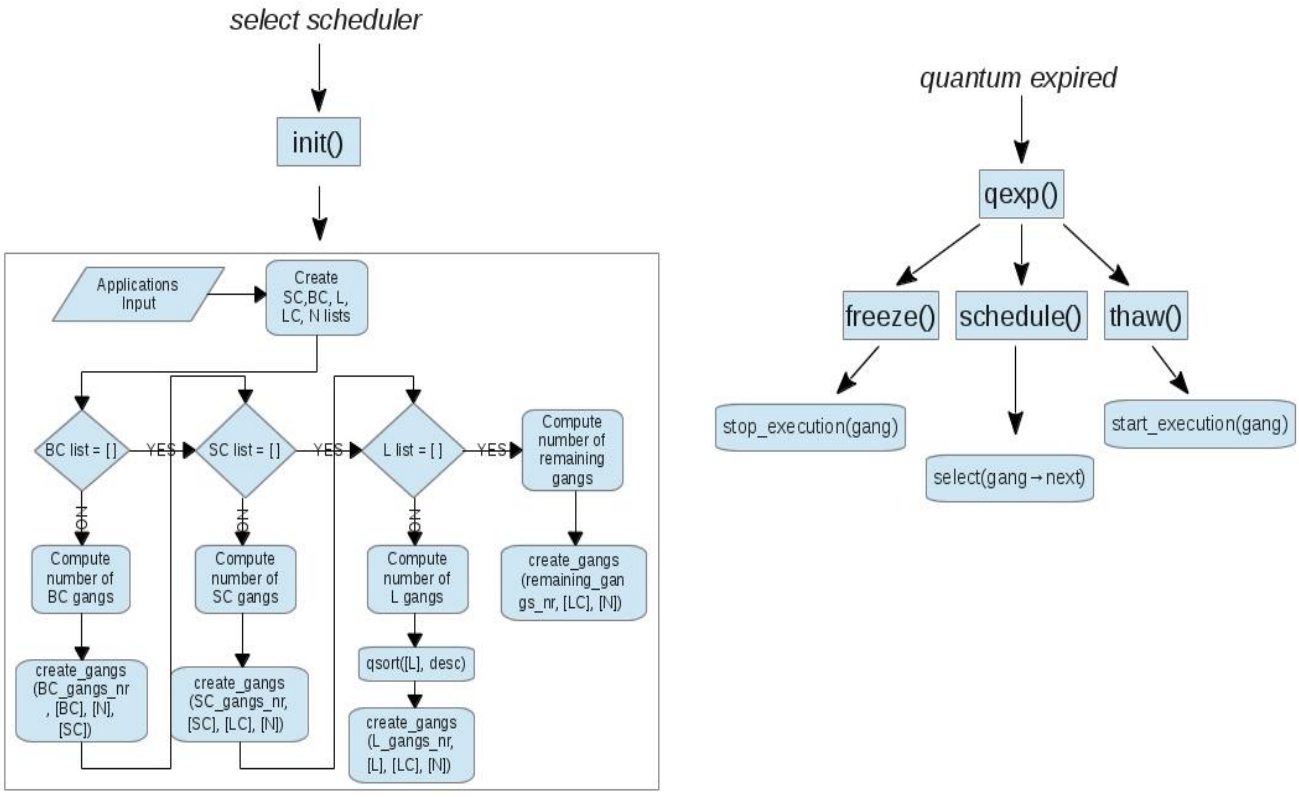


Figure 3.1-13: Flowchart of CMB scheduler

```
void create_gangs (gangs_nr, apps_l)
{
        list_t    gangs[gangs_nr];

        for_each_app_in_list(apps_l){
                remove_from_top (apps_l);
                add_to_tail (gangs[i++]);
        }
}

void form_gangs(apps)
{
        move_to_lists(apps);
        gangs_nr = (apps_nr / cores_nr);
        minimum_L = (L_nr / cores_nr);
        BC_gangs_nr = 0;
        SC_gangs_nr = 0;

        if (BC_list != [])
        {
                maximum_BC = (gangs_nr – minimum_L);
                BC_gangs_nr = ((BC_nr < maximum_BC) ? BC_nr : maximum_BC);
                create_gangs(BC_gangs_nr, [BC, N, SC]);
        }

        if (SC_list != [])
        {
                SC_gangs_nr = (SC_nr / cores_nr);
                create_gangs(SC_gangs_nr, [SC, LC, N]);
        }

        remaining_gangs_nr = gangs_nr – (BC_gangs_nr + SC_gangs_nr);
        if (L_list == []) create_gangs(remaining_gangs_nr, [LC, N]);
        else
        {
                quicksort(L_list);
                create_gangs(remaining_gangs_nr, [L, LC, N]);
        }
}
```

Code 3.1-1: CMB algorihm

3.2 Second Approach (Managing Contention)

3.2.1 **Fairness over running time**

The state-of-the-art schedulers implemented on CMPs have varying objectives they try to optimize. One of those is the equal share of resources among the applications of their workload. They multiplex the CPU in time and space with policies ensuring, that all the available cores will be utilized and each process will take the same fraction of CPU time in a time interval. CFS, the Linux scheduler, achieves fairness via assigning to the CPU processes with the gravest need (higher waiting time). This means that all processes have been waiting for the same period before they run on a core and, thus they have been treated fairly as far as the time out of the CPU is concerned. This implies that fairness is enforced for their running time too. Testing on our platform (4 cores) the Linux scheduler for a time period of 321 seconds with a workload of 12 random applications, we understand how the CPU time is distributed across those applications (Figure 3.2-1) and how fairness is enforced.

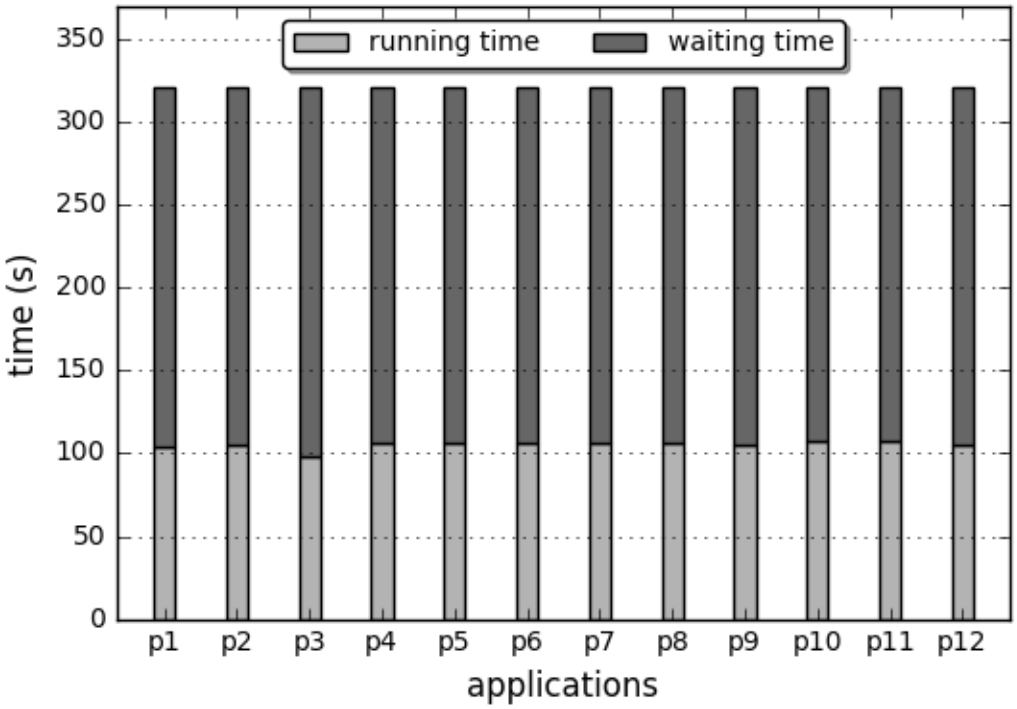


Figure 3.2-1: Running and waiting time of 12 applications scheduled with Linux

To keep the workload full in this executing interval, we respawn every process that terminates. As expected, fairness is enforced, as long as waiting (2/3 of total time) and running (1/3 of total time) time is distributed equally among processes.

On each time slice, the Linux scheduler executes 4 applications on the neighboring cores. As discussed earlier, contention occurs on the shared resources when threads run simultaneously and have negative impact on their performance. Their IPC decreases significantly (compared to alone execution) and the progress made is not reflected on their running time. Their actual progress is given by the form 2.5.1 and is a fraction ($\frac{\mathrm{IPC_{co-running}}}{\mathrm{IPC_{alone}}}$) of their running time.

For each application we know its alone execution (measured offline) and how many times it is finished when co-running. So the actual progress is simply deduced by the form:

$$actual\_progress = times\_finished \times alone\_execution \quad \textit{(form 3.2.1)}$$

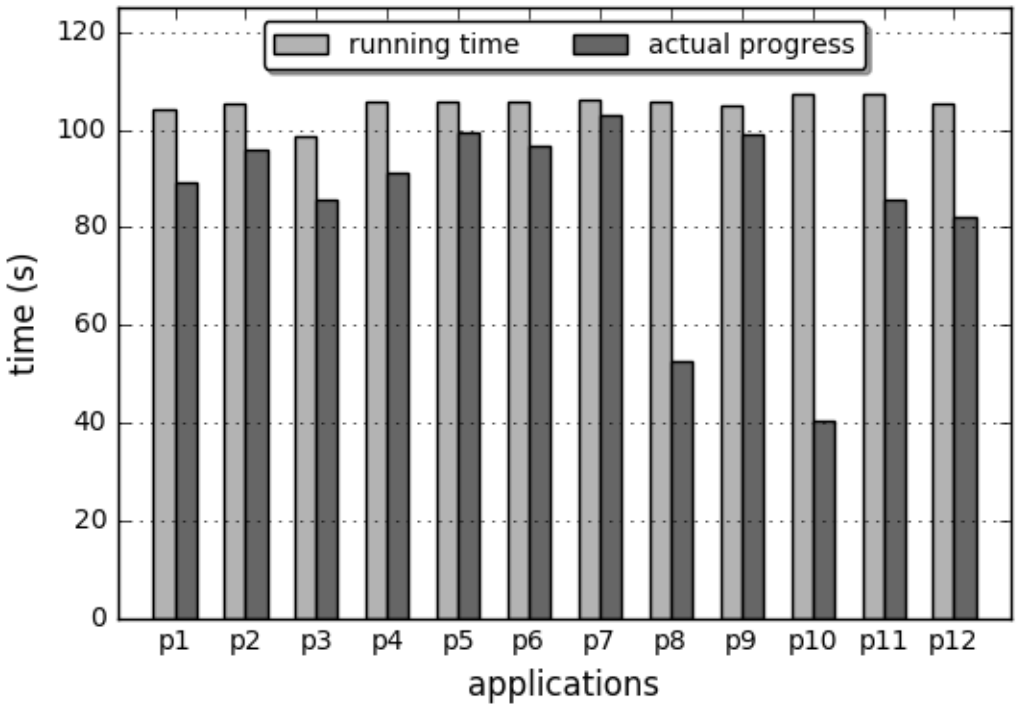


Figure 3.2-2: Actual progress compared to running time

This figure clearly depicts, that despite the effort of the Linux scheduler to be fair, giving equal running time to each process, fairness cannot be achieved. It cannot be assured that, on a given time interval, applications would make equal progress, as long as IPC loss do not occur uniformly among them.

3.2.2 **Fairness over progress**

Applications are executed on groups of 4. Our workload consists of 3 groups. Each group, also called gang, have the same 4 applications throughout the entire execution. At the end of any time quantum, a group has just left the CPU and has zero waiting time, the other is ready to be assigned to the CPU with waiting time = 2 x time quantum and the 3rd is the next group to be executed with waiting time = 1 x time quantum. Thus the contention level for each gang remains the same and the IPC loss is stable for all of the applications.

The IPC ratio is given by the form 2.5.1:

$$ratio = \frac{IPC_{co-running}}{IPC_{alone}} = \frac{time_{actual}}{time_{co-running}} \quad (form\ 3.2.2)$$

Assuming that the IPC ratio remains the same for each application, independent of the varying contention levels that may occur between different co-runners, we try to find how the running time should change in order to share the progress fairly among them.

The simple equation system below will help us find the solution:

$$ratio_i \cdot x_i = equal_{progress}, \qquad 0 < i \leq 12 \quad (form\ 3.2.3)$$

$$\sum_{i=1}^{i<13} x_i = total_{time} \cdot cores_{nr} = 1284 \quad (form\ 3.2.4)$$

Equation 3.2.3 is derived from the form 3.2.2. We want to find the running time (unknown) for each application, which multiplied by the corresponded IPC ratio would produce the equal progress (unknown). Thus the sum of their running time should be equal to the total given time that the workload is going to be executed multiplied by the number of cores. (form 3.2.4).

The results of the solution of this equation system are shown in the figures below. It seems that in order to achieve actual fairness, we need to distribute unequal the CPU time across the applications. We notice that applications suffering the most (low IPC ratio) need to be scheduled more frequent, “stealing” running time from those that make sufficient progress. This approach assure fairness over progress in a contention agnostic way. It depends only on the IPC loss and aims to proper distribution of running time between processes performing with varying IPC ratio. There is no requirement of specifying contention levels between different combinations of applications and finding a scheduling policy that could place threads in such a way that their IPC loss would distributed uniformly.

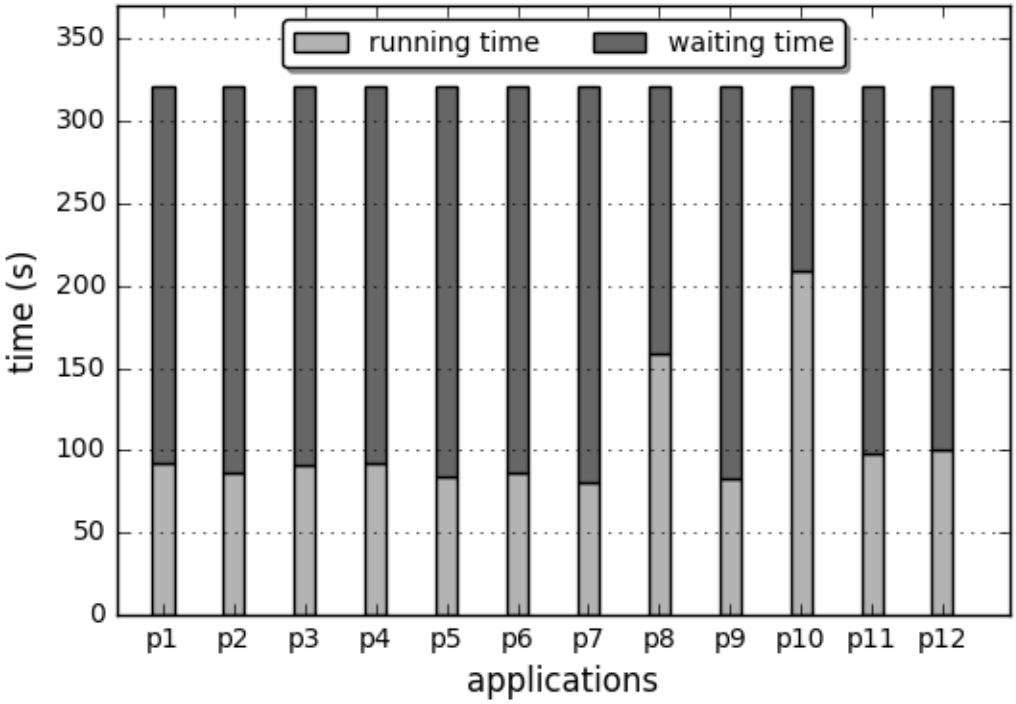


Figure 3.2-3: Waiting and running time distribution for fair progress

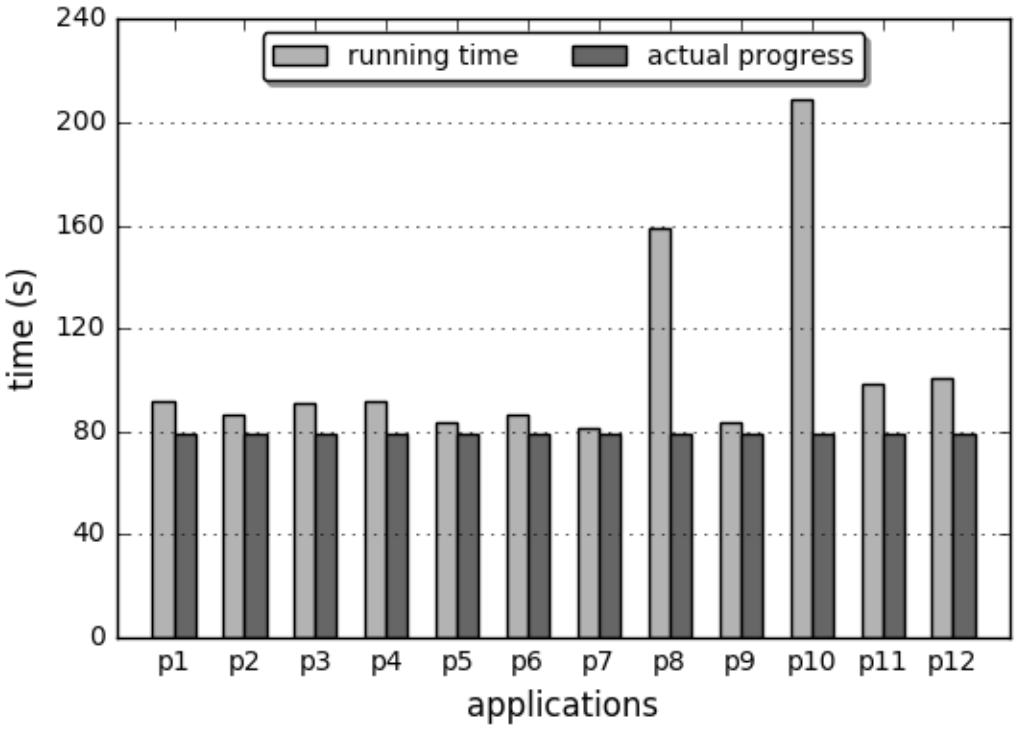


Figure 3.2-4: Unfair running time over fair progress

3.2.3 **The FOP (Fairness over Progress) Scheduler**

The Idea

Based on the observation made before, the core idea is to give the opportunity for applications that experience significant IPC reduction to be scheduled more frequent at the expense of those which perform with a high IPC ratio. Enforcing this policy, applications with poor progress are boosted, while others with satisfying progress are declined in order to create a balanced environment concerning the progress and not the running time.

The implementation

Metric

This approach does not require any preliminary work for finding the appropriate classification method, inspecting how applications of different classes interact with each other (prediction model) and deciding on a specific policy for their placement. The only thing needed is information about the IPC of each application when running alone and when running with others. This simple metric can be easily taken on runtime via the performance monitoring our CMP platform provides. The alone IPC can be acquired offline before the execution of the workload or online sacrificing one time quantum per application. The co-running IPC is gathered at the end of each time quantum where the application are exiting the running state and entering the ready state.

$$IPC = \frac{EVENT\_INSTR\_RETIRED}{EVENT\_UNHLT\_CORE\_CYCLES} \quad (form\ 3.2.5)$$

Waiting Queue

Now that we have collected all the information about the IPC loss after each time quantum, we need to find a way to boost the applications depending on that loss. Inspired by the way the Linux scheduler chooses the applications to be scheduled (trying always to minimize those with the highest waiting time), we implement a waiting queue for the applications of the workload. Before the start of each time quantum, this waiting queue gets sorted in descending way with criterion the waiting time plus a penalty (2). The k applications (k = number of cores) on the top of the queue with the higher criterion are selected for execution. This penalty represents the IPC loss of the applications multiplied by a factor (3) and is updated every time they are assigned on the CPU. Waiting time of applications remaining out of the CPU is incremented one time quantum per schedule and is zeroed when they are executed. Concluding for each application its criterion is initialized with the penalty value every time it finishes its execution and is incremented one quantum per schedule that keeps waiting out of the CPU.

$$criterion = waiting\ time + penalty \qquad (form 3.2.6)$$

$$penalty = factor * IPC\_loss \qquad (form\ 3.2.7)$$

$$\text{IPC\_loss} = \left(1 - \frac{\text{IPC}_{\text{co-running}}}{\text{IPC}_{\text{alone}}}\right), \quad \text{IPC\_loss} \in [0,1] \qquad (form\ 3.2.8)$$

Adding this penalty on the waiting time, we give the opportunity to applications, that in this specific time quantum on the CPU suffer with a significant IPC loss, to take a higher place on the waiting queue, stealing the turn of others that have previously perform on an adequate IPC ratio and have low penalty. Therefore, it is possible to be selected more frequently, increasing their running time and progress accordingly (Figure 3.2-5). However, IPC loss only is not sufficient for boosting the applications up to the queue. Its value ranges between 0 and 1 (4), practically less than 1 because it is difficult for an application to experience zero IPC ratio. If, for example, the time quantum is defined to 1 second and only the IPC_loss was considered as penalty, the criterion value of applications that just finished their execution would be less than 1 ($criterion = IPC\_loss, IPC\_loss < 1$) and they would be placed at the end of the queue. On the next position of the queue are placed the applications that were executed right before the currently finished and their criterion value is 1 second at least ($criterion = waiting\_time + IPC\_loss = one\_time\_quantum + IPC\_loss = 1 + IPC\_loss > 1$). With the group just left the CPU having criterion < 1 and the next lower group having criterion > 1, no elevation upon the waiting queue could be possible and the scheduling would result in fairness over running time as shown in the figure below (figure applications with bold are assigned on the CPU for the current quantum).

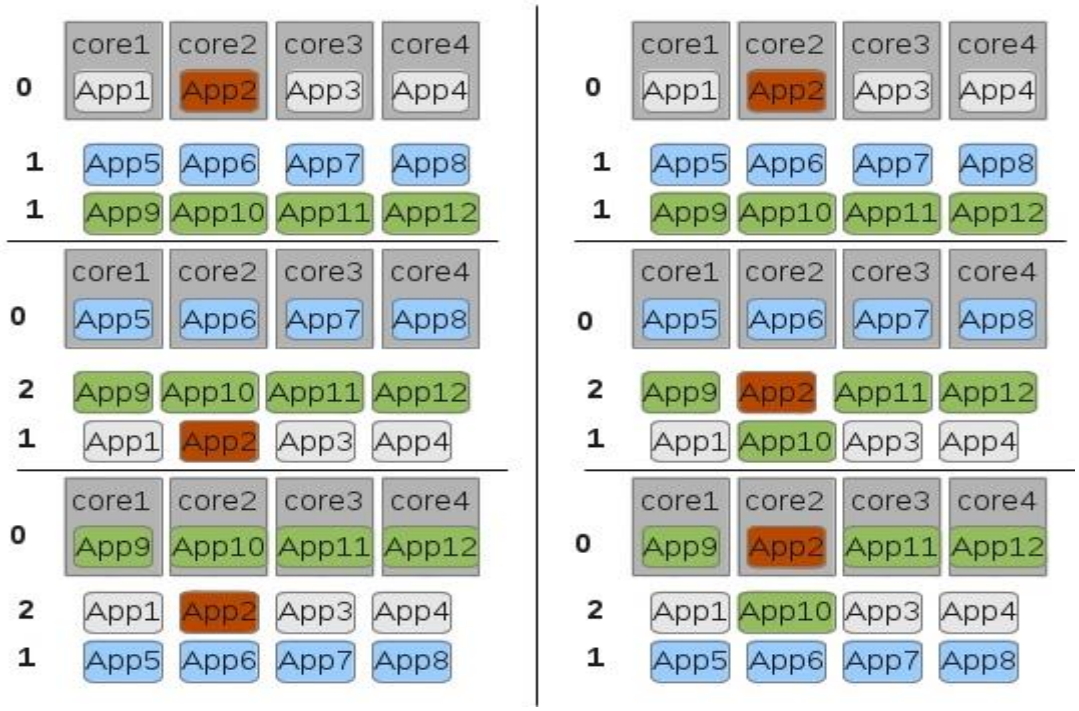
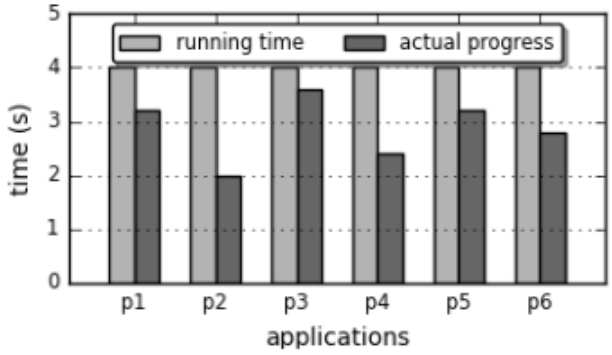


Figure 3.2-5: Elevation of App 2 higher in the waiting queue, using as criterion the waiting time plus the penalty

Penalty = IPC_loss

	IPC_loss
p1	0.2
p2	0.5
p3	0.1
p4	0.4
p5	0.2
p6	0.3

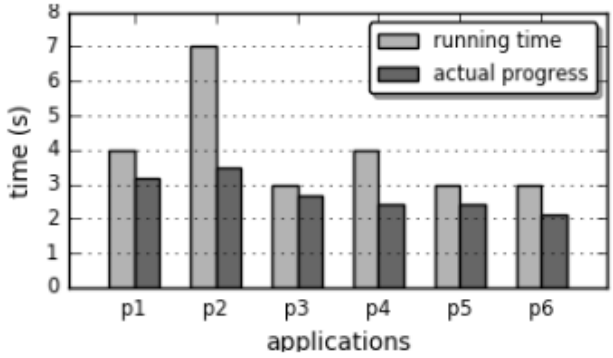


1s		2s		3s		4s		5s		6s		7s		8s		9s		10s		11s		12s	
p3	1	p5	2	p2	2.5	p4	2.4	p6	2.3	p2	2.5	p4	2.4	p6	2.3	p2	2.5	p4	2.4	p6	2.3	p2	2.5
p4	1	p6	2	p1	2.2	p3	2.1	p5	2.2	p1	2.2	p3	2.1	p5	2.2	p1	2.2	p3	2.1	p5	2.2	p1	2.2
p5	1	p2	1.5	p4	1.4	p6	1.3	p2	1.5	p4	1.4	p6	1.3	p2	1.5	p4	1.4	p6	1.3	p2	1.5	p4	1.4
p6	1	p1	1.2	p3	1.1	p5	1.2	p1	1.2	p3	1.1	p5	1.2	p1	1.2	p3	1.1	p5	1.2	p1	1.2	p3	1.1
p2	0.5	**p4**	0.4	**p6**	0.3	**p2**	0.5	**p4**	0.4	**p6**	0.3	**p2**	0.5	**p4**	0.4	**p6**	0.3	**p2**	0.5	**p4**	0.4	**p6**	0.3
p1	0.2	**p3**	0.1	**p5**	0.2	**p1**	0.2	**p3**	0.1	**p5**	0.2	**p1**	0.2	**p3**	0.1	**p5**	0.2	**p1**	0.2	**p3**	0.1	**p5**	0.2

Figure 3.2-6: 12 second execution with factor = 1

Penalty = factor * IPC_loss
Factor = 4

	IPC_loss
p1	0.2
p2	0.5
p3	0.1
p4	0.4
p5	0.2
p6	0.3



1s		2s		3s		4s		5s		6s		7s		8s		9s		10s		11s		12s	
p2	2	p4	2	p2	3	p1	3.8	p3	3.4	p5	3.8	p4	3.6	p1	3.8	p6	3.2	p5	3.8	p2	3	p3	3.4
p3	1	p5	2	p6	3	p4	2.6	p2	3	p6	3.2	p2	3	p3	2.4	p2	3	p4	3.6	p1	2.8	p6	3.2
p4	1	**p2**	2	p1	2.8	p3	2.4	p5	2.8	p4	2.6	p1	2.8	p6	2.2	p5	2.8	**p2**	2	p3	2.4	p4	2.6
p5	1	p6	2	**p4**	1.6	**p2**	2	p6	2.2	**p2**	2	p3	1.4	**p2**	2	p4	2.6	p1	1.8	p6	2.2	**p2**	2
p6	1	p1	1.8	p3	1.4	p5	1.8	**p4**	1.6	p1	1.8	**p6**	1.2	p5	1.8	**p1**	0.8	p3	1.4	**p4**	1.6	p5	1.8
p1	0.8	**p3**	0.4	**p5**	0.8	**p6**	1.2	**p1**	0.8	**p3**	0.4	**p5**	0.8	**p4**	1.6	**p3**	0.4	**p6**	1.2	**p5**	0.8	**p1**	0.8

Figure 3.2-7: 12 second execution with factor = 4

For this reason, it is necessary to form the penalty as the weighted IPC loss (multiplied by a factor). The factor should be selected in such a way that would give the proper boost to the suffering applications, taking into consideration the time quantum and the number of gangs (progs_nr/cores_nr), and create a fair over progress environment as shown in figure (figure). The proper selection of this factor is explained further on the next paragraph.

Factor Selection

We previously described the strategy followed to boost applications that suffer significant IPC loss and give them the opportunity to occupy the resources more times. We understand that IPC loss only is not an adequate characteristic to elevate properly the applications up to the waiting queue. We need to establish a weighted IPC loss deciding the factor, while we take into consideration 2 other parameters, the time quantum and the gangs (progs_nr/cores_nr). In the above given example lowering the time quantum we have to lower the factor because the increment of the criterion of the waiting applications will occur at a slower pace and the applications suffering the most would occupy the CPU much more frequently. This would create an unfair over progress environment, as the execution of applications with high IPC ratio would be repeatedly impeded by those with low. Supposedly now that in the above example we increase the number of gangs, by increasing the applications. As a result, the waiting time of the group being on the top of the queue would be much more higher, as long as it has to wait for more groups to be executed before its turn comes. Without changing the factor , applications that need to occupy the CPU more frequent would be unable to reach in a satisfying place in the queue (they have to pass more gangs). This means that their progress cannot be boosted and fairness cannot be enforced.

Concluding, once the time quantum is defined, we try to find out how the factor and the number of gangs should be related in order to achieve our objective, fairness over progress. Having defined the time quantum to 1 second, the idea is to test our scheduling policy between different combinations of gang numbers and factors. For each gang number we select the factor that leads to the best performance. Our metric for understanding the best possible performance is the standard deviation of the actual progress the applications made in the total execution time of the workload. When this metric converges to zero, the result would be similar to this one shown in figure (figure) and fairness would be achieved. So for each gang number, we select the factor that minimizes the standard deviation of progress, gather all the pairs of gang number and factor and try to find a possible relation.

However, executing all these tests in a real environment is really time costly and infeasible to be accomplished. For this reason we simulate our scheduling policy, creating a simple program that take as input the number of applications with their varying IPC losses, the number of cores and the execution time interval and produce as output the factor that results in the minimum standard deviation. Factor values between 1 and progs_nr have been tested on the given workload with iteration step 0.2. Execution time is divided to N quantum ($N = execution\_time/time\_quantum$) and one time quantum is one iteration in the for-loop. For any gang number the standard deviation follows the same behavior as a function of factor. Initially it starts with a high value and decreases as the factor increase. It reaches a minimum value and then increases as the factor continues to rise (Figure 3.2-8). This seems absolutely logical, while , adapting a high factor value, applications with great IPC loss would remain always on the top of the queue with criterion value that dominates the criterion value of those with low IPC loss. As a result those applications should remain a long time out of the CPU to increase their criterion (increasing waiting time) and reach the top in order to be executed. This situation leads to thread starvation of those with low IPC loss.

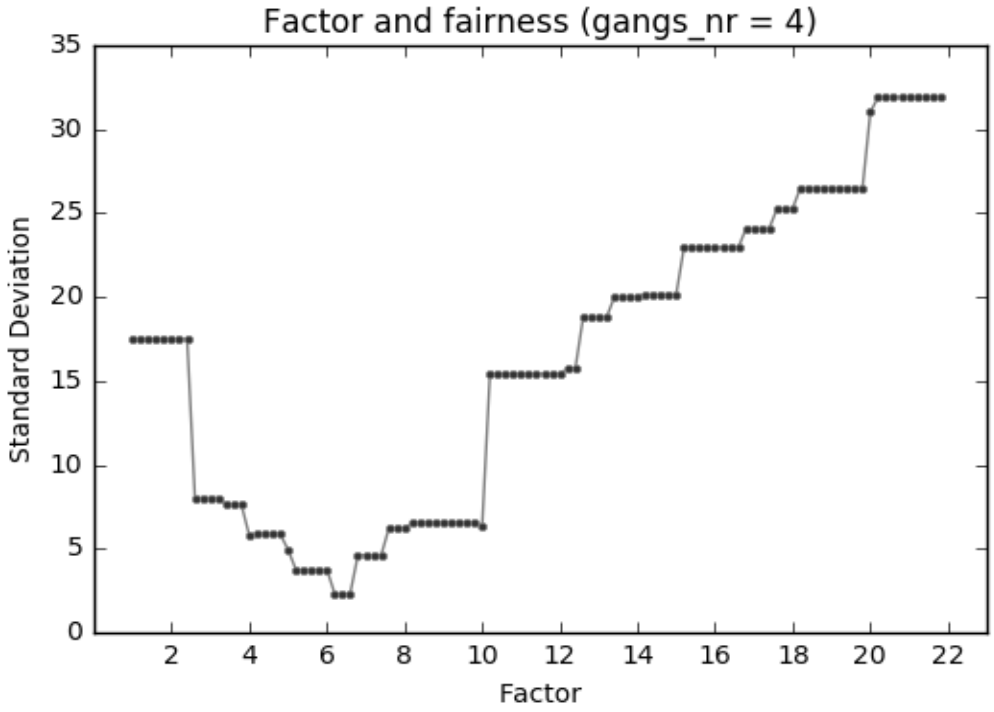


Figure 3.2-8: Minimum standard deviation for factor = 6 on a workload with 4 gangs

Finally, we take measurements for three different core numbers (2, 3, 4), in order to extract the relationship between factor and gangs number. Results are shown in the figures below.

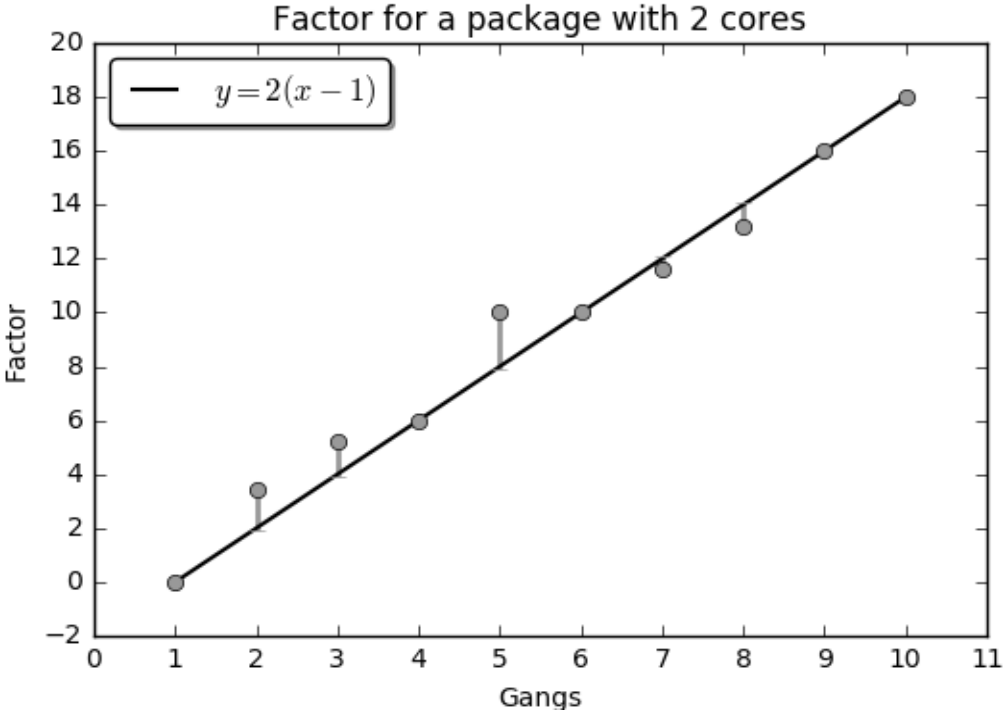


Figure 3.2-9: Relationship between factor and gangs number for a 2-core architecture

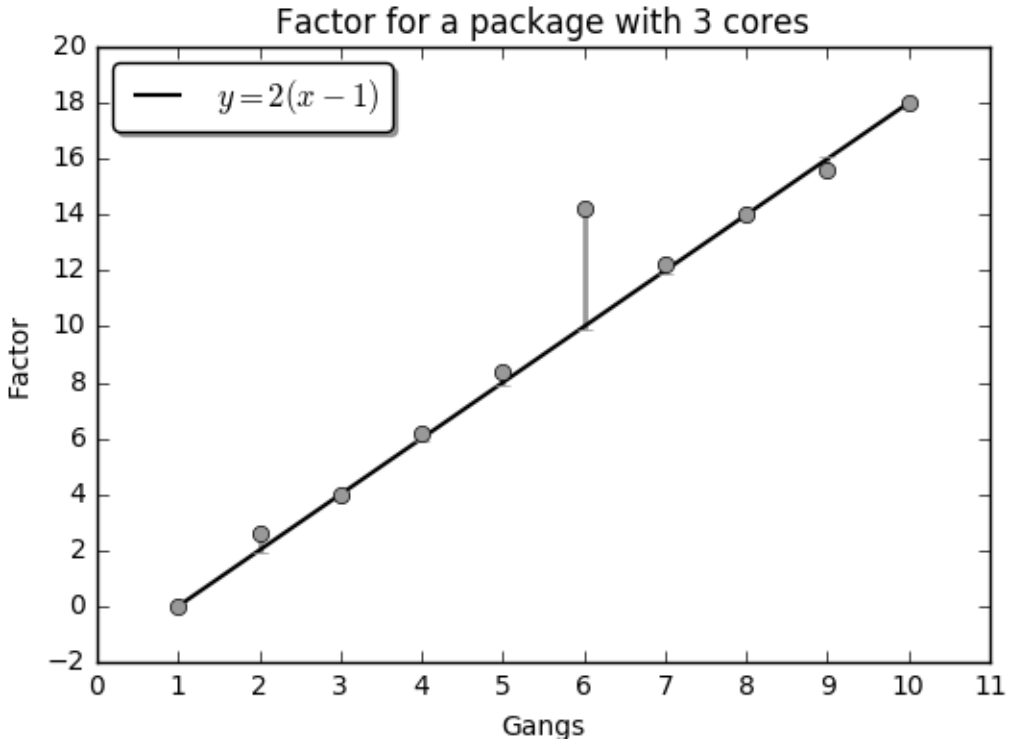


Figure 3.2-10: Relationship between factor and gangs number for a 3-core architecture

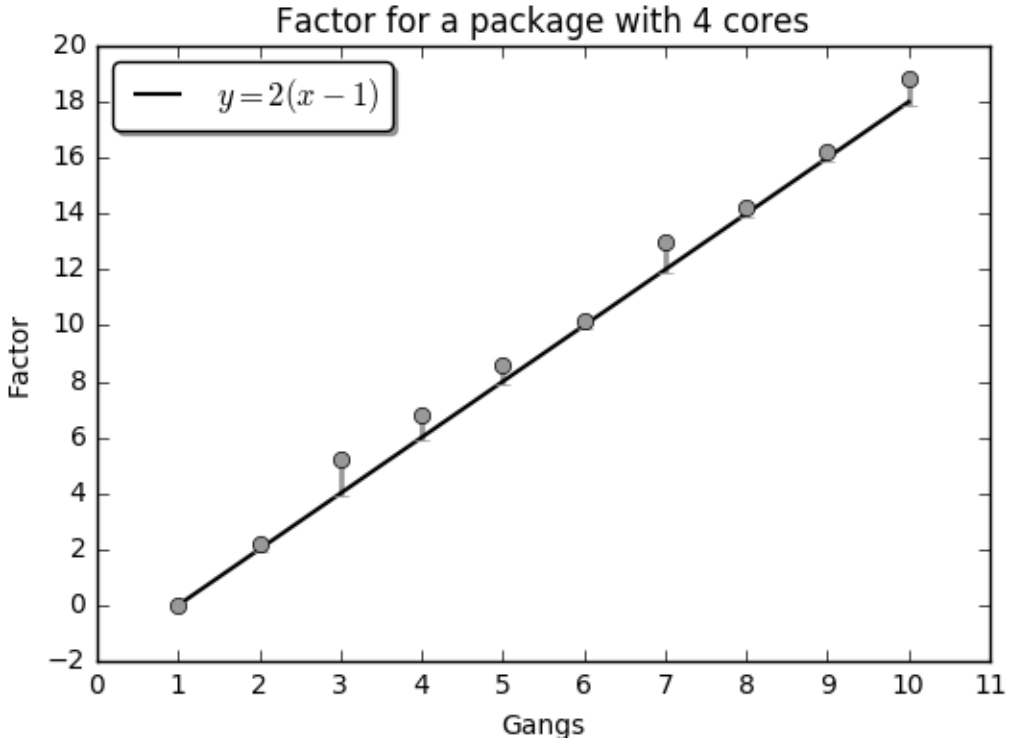


Figure 3.2-11:Relationship between factor and gangs number for a 4-core architecture

As it is made clear by the graphs, the factor is a function of gangs number (same behavior across different cores_nr) and it can be simply selected by the form:

$$Factor = 2 \cdot (gangs\_nr - 1), \quad gangs\_nr = \frac{progs\_nr}{cores\_nr} \qquad form(3.2.9)$$

IPC loss

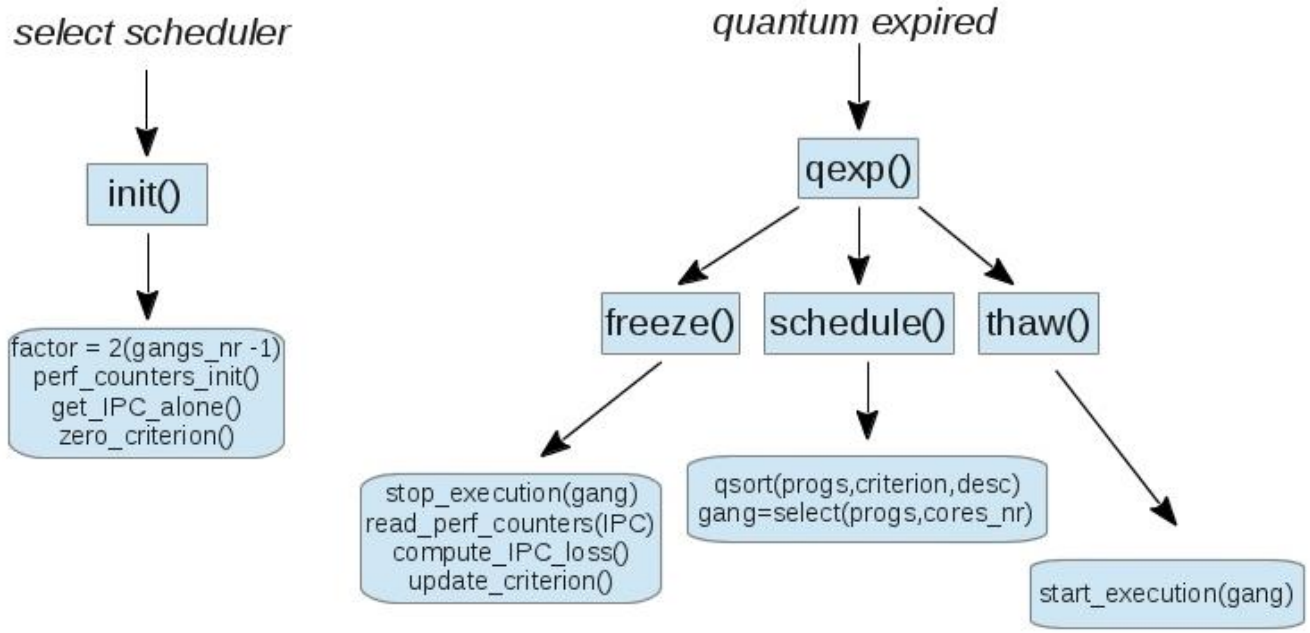
In a real execution environment, IPC loss of an application does not remain the same, as it is possible to have different co-runners and generate varying contention levels. Although all previous work was made with the assumption, that IPC loss remains stable, in order to simplify our approach, this scheduling policy is not violated when enforced to real platforms, as long as the idea to boost the suffering applications is based on the IPC loss occurring on every time quantum.

The algorithm

In our case the waiting queue is implemented as a list and gets sorted with quicksort algorithm. This means that the complexity for deciding which applications to run before each time quantum is O(n log n). Queues on Linux scheduler are implemented as red-black trees and all the actions are performed in O(log n) time complexity.

The factor is selected on the initialization of the scheduler, following the relationship we extract on the previous paragraph. The IPC of each application on alone execution has been measured beforehand and is given as a parameter. The co-running IPC (and IPC loss) is measured after the execution of the gang on each quantum.

The algorithm is presented on the flow chart and pseudocode below.



```
void init (){
        gangs_nr = progs_nr/cores_nr;
        factor = 2 (gangs_nr – 1);
        counters = perf_counters_init(selected_events);

        for_each_application_in_list(progs_all_list){
                application->criterion = 0;
                application->IPC_alone = get_IPC_parameter();
        }
}

void schedule(){
        quicksort(progs_all_list, criterion, DESCENDING);
        while (number(progs_schedule_list) < cores_nr){
                remove_from_top(progs_all_list);
                add_to_tail(progs_schedule_list);
        }
}

void thaw(){
        for_each_application_in_list(progs_schedule_list){
                start_running(application);
        }
        perf_counters_zero(counters);
        perf_counters_start(counters);
}

void freeze(){
        perf_counters_stop(counters);

        for_each_application_in_list(progs_schedule_list){
                stop_running(application);
                value = perf_counters_read(counters);
                IPC_co-running = get_IPC(value);
                IPC_loss = (1 – IPC_co-running/application-
>IPC_alone);
                application->waiting_time = 0;
                application->penalty = factor * IPC_loss;
                remove_from_top(progs_schedule_list);
                add_to_tail(progs_all_list);
        }

for_each_application_in_list(progs_all_list){
                update_waiting_time(application);
                application->criterion = waiting_time + penalty;
        }
}
```

```
void quantum_expired(){
        freeze();
        if (current_tics < RUN_TICS){
                schedule();
                thaw();
        }else{
                stop_execution();
                print_results();
        }
}
```

Code 3.2-1: FOP scheduler algorithm

3.3 Third Approach (Avoiding and Managing Contention)

3.3.1 The FOP-LCI (Fairness over Progress with L-C class Isolation) Scheduler

The idea

In the first approach, we build a scheduler that aims to improve fairness and throughput avoiding contention. In the second approach, we try to manage contention, assuring that all threads would take equal share of the resources without caring at what extent the thread is possible to affect or be affected by the co-runner. In this approach we adopt the idea of the second scheduler, taking into consideration that some applications when co-executed cause destructive interference. We extend its functionality, in order to avoid co-execution of L-C pair. As the prediction model imposes, co-scheduling of L with C (SC, BC) applications could result in heavy IPC losses for the C (x1.67 for SC, x2.43 for BC). So our idea is to avoid a part of the contention (L-C) and manage the rest of it (BC-LC, BC-BC).

The Objective

Avoiding bad co-runners for the C applications means, that they have the opportunity to run in a less contended environment and increase their actual progress. This way the total throughput is expected to be enhanced. On the other hand, suffering applications would have the change to be selected more frequent, as assured on the FOP scheduler. Concluding, our aim is to provide fairness with improved throughput comparing to FOP.

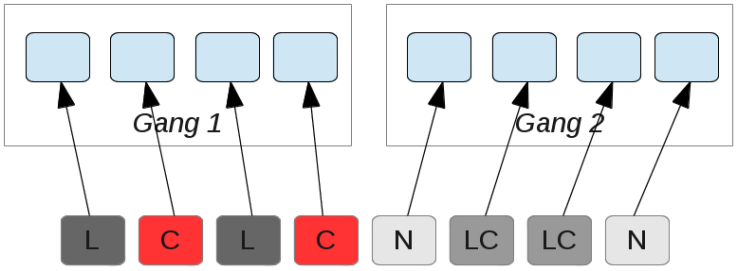


Figure 3.3-1: Selection of applications with the FOP scheduler

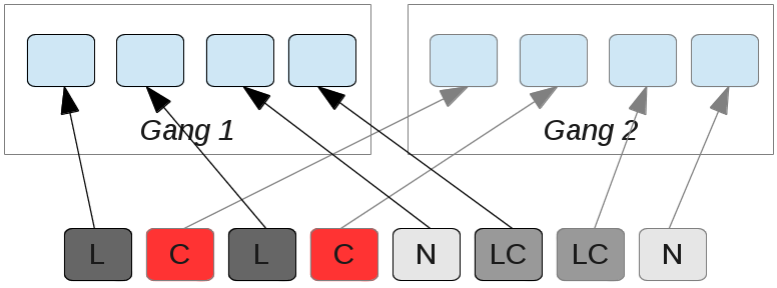
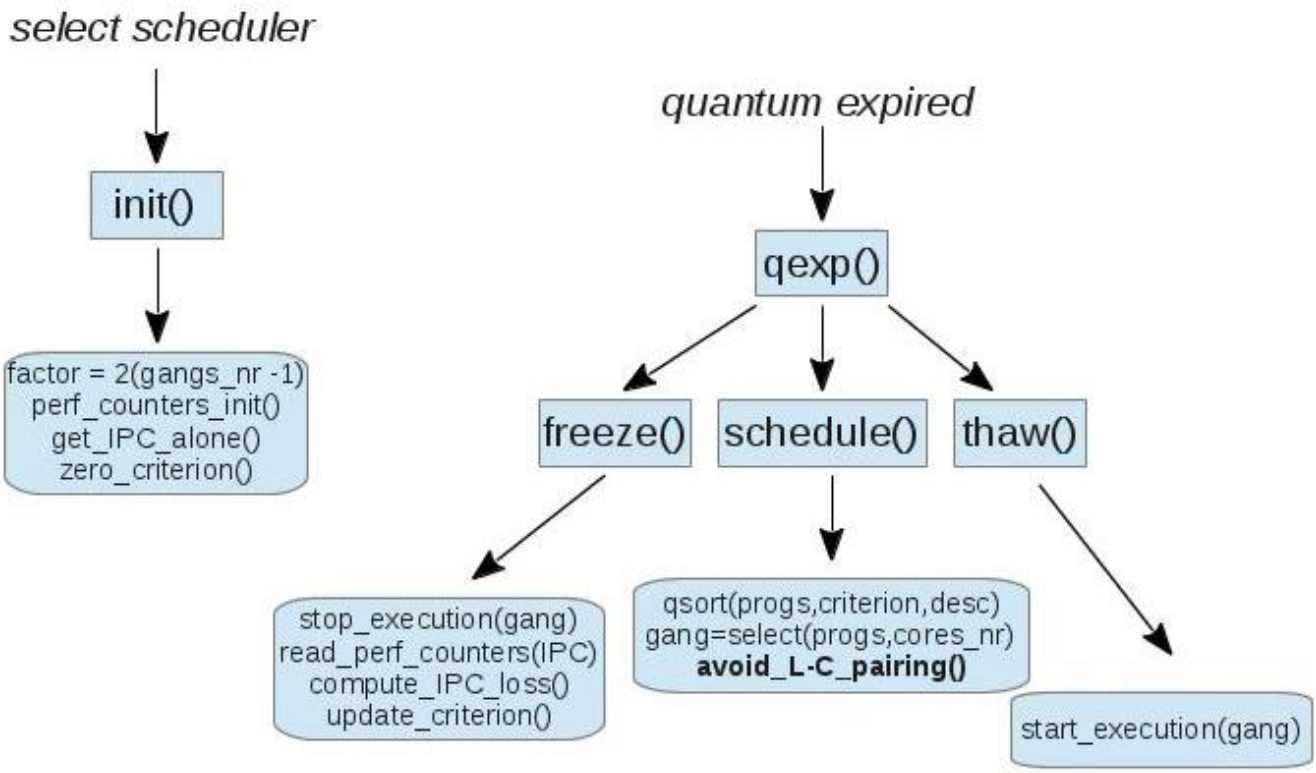


Figure 3.3-2: Selection of applications with the FOP-LCI scheduler

The Implementation

We use the same waiting queue, sorted in the way described on the previous scheduler. The difference now is that we do not schedule the N top applications of the queue (N = cores_nr). The applications are classified following the scheme we defined in the CMB scheduler. If an L (C) application is identified first, it is packed in the gang. If the gang is not filled, we iterate over the waiting queue ignoring all the C (L), packing with the LC, N applications. The ignored applications C (L), expected to run on this time quantum, lose their turn. They are kept out of the CPU for an extra quantum, increasing their waiting time. On the next quantum, they will be on the top of the queue, will be selected first and will keep out the L (C). In this way we form gangs that consist of L(C), LC, N applications. On the Figure 3.3-2 we show how the selection occurs.

To achieve the scheduling policy described, we extend the schedule () function of the B scheduler. This extension is depicted on the flow chart and the pseudocode below.



```
void schedule()
{
        L_ON = 0;
        C_ON = 0;
        allowed = 1;
        quicksort(progs_all_list, criterion, DESCENDING);
        for_each_app_in_list(progs_all_list){
                switch (app->class){
                        case L_CLASS:
                                if (C_ON) allowed = 0;
                                else L_ON = 1;
                        case C_CLASS:
                                if (L_ON) allowed = 0;
                                else C_ON = 1;
        }

        if (allowed ){
                if (number(progs_schedule_list) != cores_nr){
                        remove_app_from_list(progs_all_list);
                        add_to_tail(progs_schedule_list);
                }
        }
}
```

Code 3.3-1: Extension of schedule() function

3.4 Implementation tool

3.4.1 The scaff

All scheduling policies proposed in this chapter are implemented on the scaff infrastracture. Scaff is a runtime system that organize the execution of single or multi-threaded applications on multi-core architectures [19]. It operates at user-space, on top of Linux-based systems. Its main responsibility is to provide a communication mechanism between the scheduler implementation and the executed applications. Scaff consists of two basic subsystems, the executor and the scheduler. The executor handles execution events regarding creation or termination of applications, while the scheduler decides on which way the available resources would be distributed among the applications of the workload. In the next paragraphs, we describe in detail the operation of these subsystems.

The executor

The executor keeps information about the programs of the workload, specific events during execution and the current scheduler implementation. For each execution test, the following parameters are passed to the executor: a configuration file, an output folder, the set of CPU and the scheduler. The configuration file includes the execution path of the applications of the workload along with necessary information for their execution, such as number of required cores for each one (on this paper we work with single threaded applications), starting time if a delay is desired, the class they belong to and their IPC on alone execution (essential for the decision making of our schedulers). On the output folder, we keep track of the scheduler operations, the programs behavior (performance counters data) and error messages. The executor inspects the information given for each application and prepares them for the execution, establishing the cpuset and a shared memory area for communication between the application and the scheduler. Its main duty is to handle execution events and programs' signals and trigger corresponding scheduler functions. It is mainly interested on the events, *EVNT_NEWPROG* and *EVNT_QEXPIRED*. The first is associated with the start of the execution of a program. When a program is added, this event is allocated. Subsequently the executor make the following actions, it moves this program to the *new_p* list and calls the function *scheduler→rebalance().* The second event indicates the expiration of a time quantum. The executor, then, returns the execution flow to the scheduler calling the function *scheduler→qexpired().* The signals that scaff handles are *SIGCHLD* and *SIGSTOP. SIGCHLD* is send to executor when a program has terminated its execution normally, while *SIGSTOP* is raised when a forced termination occurs. Regarding the first signal, the executor moves the finished program to *finished_p* list and calls the scheduler function *rebalance(),* whereas the second signal implies an erroneous behavior and the executor terminates the whole execution.

The scheduler

The scheduler is the second subsystem of the scaff, responsible for the placement of the programs to the available cores. Based on a scheduling policy, it distributes the resources among the programs, taking into consideration their demands and the information for the

execution platform provided by the executor. Concerning our work, the scheduler has to make decision about when (which time quantum) and not where (which core) the programs should be co-executed, as it based on time-sharing only.

Our schedulers implement the following functions:

- *init()*: this function is called at the beginning of execution by the executor and allocates structures necessary for the management of programs. These are stored by the executor and used to subsequent callsof the scheduler.
- *schedule(app_list)*: at this point occurs the selection of the programs to be executed on the following time quantum.
- *thaw(sched_list):* the selected applications are ready to be executed on the set of the available cores. The performance counters are activated and the state of the freezer cgroup subsystem of each program is set to thaw.
- *freeze(sched_list):* the time quantum has expired and the running applications should leave the CPU. The performance counters are stopped and their value can be easily read, providing information for the performance of each program running on the current quantum. The state of the freezer cgroup subsystem of each one is set to freeze.
- *qexpired(sched_data):* this function is called on the expiration of the time quantum by the executor (*EVNT_QEXPIRED* occurs). Subsequently it calls *freeze, schedule* and *thaw* functions, making the decision for the next time quantum.
- *rebalance(sched_data):* when a program is ready to start execution, it is added in the *new_p* list (*EVNT_NEWPROG*) and this function is called. It moves this program from the *new_p* list to the *app_list* and allocates the structure that the scheduler keeps for every program in order to store important information for their execution. This function is also called when a *SIGCHLD* is raised to the executor, meaning that a program has terminated normally. In this case, the scheduler removes it from the *finished_p* list, respawns it, if this attribute is enabled and deallocates it.

3.4.2 System tools and mechanisms

The scaff infrastructure requires some tools, in order to manage the execution of programs. A scheduler determines which applications should be assigned on the available cores on a specific time slice. Taking into account that a multi-core system provides the ability of space sharing the cores, the scheduling policy should determine which program would map to the specific core. The ability of binding programs to specific cores and executing them on specific time slices is provided by a couple of Linux tools, cpusets and freezer. Both of these are part of the Control Groups (Cgroups) feature provided by Linux.

Cgroups

Control groups provide a mechanism for aggregating or partitioning sets of tasks into hierarchical groups with specialized behavior [20]. A *cgroup* is a set of task associated with a set of parameters for one or more subsystems. A *subsystem* is a module that treats the groups of tasks provided by cgroups in particular ways, applying per-cgroup limits. A *hierarchy* is a set of

cgroups arranged in a tree, such that every task is in exactly one *cgroup* and a set of *subsystems*. Each hierarchy has an instance of the cgroup virtual file system associated with it and can be easily handled from user space. User-level code can create and destroy cgroups by name in the virtual file system, specify and query to which cgroup a task is assigned, and list the task PIDs assigned to a cgroup.

On their own, the only use for cgroups is for simple job tracking. The intention is that other subsystems hook into the generic cgroup support to provide new attributes for cgroups, such as accounting/limiting the resources which processes in a cgroup can access. For example, cpusets allow you to associate a set of CPUs and a set of memory nodes with the tasks in each cgroup.

Freezer subsystem

We use the cgroup freezer subsystem, in order to implement the time sharing policy of our schedulers. It provides a convenient way to start and stop a set of tasks, by simply writing values in files of the virtual file system. Using sequences of *SIGSTOP* and *SIGCONT* signals are not always sufficient for stopping and resuming tasks in user space. Both of these signals are observable from within the tasks we wish to freeze. *SIGSTOP* cannot be caught, while *SIGCONT* can be caught by the task. Any programs designed to watch for these signals could be broken by attempting to use this way to stop and resume them [21]. In contrast, the cgroup freezer uses the kernel freezer code to prevent the freeze/unfreeze cycle from becoming visible to the tasks being frozen. The cgroup freezer is hierarchical. Freezing a cgroup freezes all the tasks associated with it and all its descendant cgroups. Each cgroup has its own state and the state inherited from the parent. These states are described in a combined way in the *freezer.state* file created by the freezer subsystem in the virtual file system. Reading this file returns the state of the cgroup – “THAWED”, “FREEZING” or “FROZEN”. FREEZING → FROZEN transition occurs when all tasks of the cgroup and its descendants become frozen. FROZEN → FREEZING transition is possible, when a new task is added to the cgroup, until it is frozen. Two values are allowed to be written in this file – “FROZEN” and “THAWED”. If frozen is written, the cgroup, if not already freezing, enters the FREEZING state until it is frozen. If THAWED is written the state of the cgroup is changed to THAWED. Therefore, it is easy to manipulate the execution of programs by writing to the *freezer.state* file THAED or FROZEN. In our implementation we create one cgroup freezer for every application of the workload.

Cpuset subsystem

Cpusets use the generic cgroup subsystem and provide a mechanism for assigning a set of CPUs and Memory Nodes to a set of tasks [22]. Cpusets extend the existing mechanisms of Linux kernel that specify in which cpus and memory nodes a process is allowed to run. Requests by a task, using the sched_affinity system call to include CPUs in its CPU affinity mask, and using the mbind and set_mempolicy system calls to include Memory Nodes in its memory policy, are both filtered through that task’s cpuset, filtering out any CPUs or Memory Nodes not in that cpuset. User space code can create and destroy cpusets by name in the cgroup virual file system, manage their attributes and specify which CPUs and Memory Nodes are

assigned to each cpuset, writing values in the files *cpuset.cpus* and *cpuset.mem* respectively. In our implementation one cpuset is created for each application of the workload.

Chapter 4
Experimental Evaluation

Contention Environment

From the benchmarks selected on the previous chapter, we create workloads of 16, that form conditions of 3 different contention levels, low, medium and high. Each contention level is populated by 5 workloads. On the low contention environment, the workloads consist of various combinations of L, LC, SC and N applications. The medium contention environment includes workloads of applications from all the classes with the BC applications being absolutely isolated. Workloads of the high contention environment have BC applications far exceeding the number of BC gangs.

Execution Details

The total 15 workload mixtures are tested with all the 3 scheduling policies (CMB, FOP, FOP-LCI) proposed on the previous section plus the state-of-the-art Linux scheduler (CFS). We set the execution time to 20 minutes and the time quantum to 1 second. In order to keep the gangs and the contention conditions stable, we keep the workload full at the whole execution, respawning every terminated application.

The execution platform, described in detail on the section, is a Nehalem architecture (one socket is utilized) with characteristics summarized on the table below

Cores	4
L1	Data cache: private, 32 KB, 8-way, 64 bytes block size Instruction cache: private, 32 KB, 4-way, 64 bytes block size
L2	private, 256 KB, 8-way, 64 bytes block size
L3	shared, 8 MB, 16-way, 64 bytes block size
Memory	12 GB, DDR3, 1333 MHz

Table 3.4-1: Execution platform

The classification of applications has accomplished offline on their alone execution. The class and other characteristics of applications, like IPC_alone, are passed to our schedulers as parameters on the configuration file. We can acquire all this information online, executing the N applications of the workload alone for N time quantums, adding this time to the total execution time, in order to be accurate with the CFS. An alternative is to reduce the time of quantum for N quantums, sacrificing a smaller fraction of the total execution time, if we keep it the same for all the schedulers. For the distinction of the C applications into SC and BC the size of the data occupying the LLC should be given as a parameter, assuming that it has been deduced in a "magical" way. We have not enforced any mechanism like cache partitioning for extracting online the size of data resident on the LLC. Our way for generating BC applications

is inspecting their behavior across different data set, as described in the previous chapter.

Performance Metrics

While our goal, from the beginning, is to implement techniques that would optimize throughput and fairness, we are interested into inspecting the proposed scheduling policies for these two characteristics. So we define some metrics that will help us compare schedulers and evaluate their characteristics.

On the first hand, for each application of the workload, we compute how many times, in the equal part of the total execution time distributed among the applications (form 4.1), it would be terminated on the ideal occasion, where the interference of the neighboring applications was null (form 4.22). This means that the application would run on an environment with conditions of zero contention and would have the same termination time as when running alone.

$$execution\_time = total\_execution\_time \cdot \frac{cores\_nr}{apps\_nr} \quad (form\ 4.1)$$

$$ideal\_times\_terminated\ (i) = \frac{execution\_time}{time\,(i)_{alone-execution}}, \qquad i = application \quad (form\ 4.2)$$

We have implemented the schedulers in such a way, that they keep track of the times each application has terminated throughout the whole execution. We compare these results to the ideal_times_terminated and we extract their normalized values (form 4.3).

$$N(s,i) = \frac{times\_terminated\,(s,i)}{ideal\_times\_terminated\,(i)}, \qquad s = scheduler, i = application \quad (form\ 4.3)$$

We define throughput and fairness of a scheduler as the average (form 4.4) and the standard deviation (form 4.5) respectively of the normalized times for all programs of the workload.

$$T(s) = average\big(N(s,i)\big), \quad for\ all\ applications\ i \quad (form\ 4.4)$$

$$F(s) = \sigma\big(N(s,i)\big), \qquad for\ all\ applications\ i \quad (form\ 4.5)$$

Information about the execution of the workload is represented by a boxplot [23] for each scheduler. A boxplot is a convenient way to graphically depict the distribution of data based on the five number summary: minimum, first quartile, median, third quartile and maximum. It consists of a central rectangle and the whiskers. The rectangle spans the first quartile to the third quartile (IQR, interquartile range). A dashed line in the box shows the median and the vertical lines (whiskers) above and below that box show the locations of the minimum and maximum. Our graph includes for each scheduler, one boxplot, an additional solid line representing the average and a text on top of y-axis annotating the standard deviation. In that way, it is possible to acquire a short and accurate description of the applications' behavior and depict the metrics of our interest essential for the schedulers' evaluation. Although the general picture of the boxplot provides with sufficient information about the deviation of the data from the average (rectangle height and whiskers range), we annotate the standard deviation on top of the y-axis for greater accuracy.

4.1 Evaluation of scheduling policies

4.1.1 Low contention environment

Workload 1

We start with a workload consisting of 4 L, 4 LC, 4 SC and 4N applications.

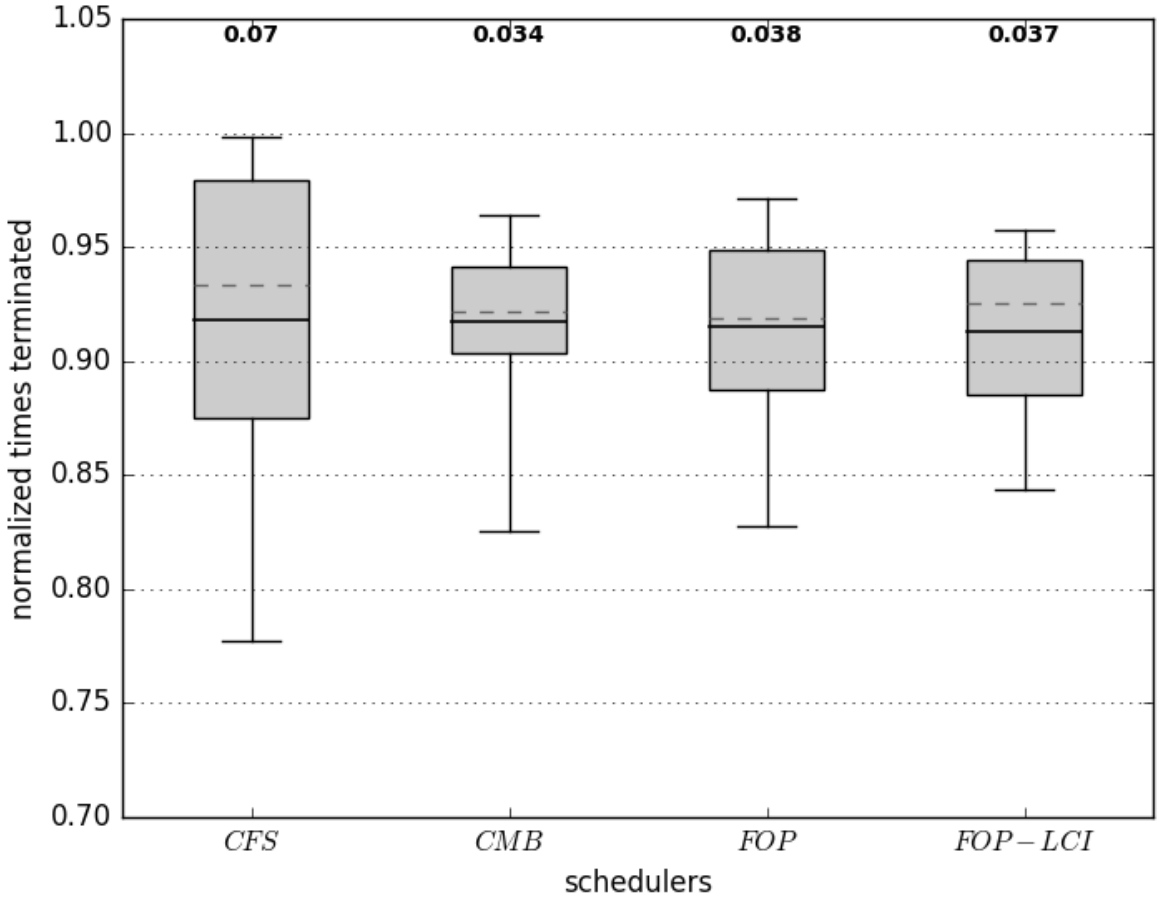


Figure 4.1-1: Workload 1 consists of 4 L, 4 LC, 4 SC and 4N applications

We observe that all schedulers are performing roughly at the same throughput level. CMB presents the best results concerning fairness. FOP and FOP-LCI present almost identical behavior.

Workload 2

We increase the number of L applications, forming a workload of 6 L, 4 LC, 4 SC and 2N applications.

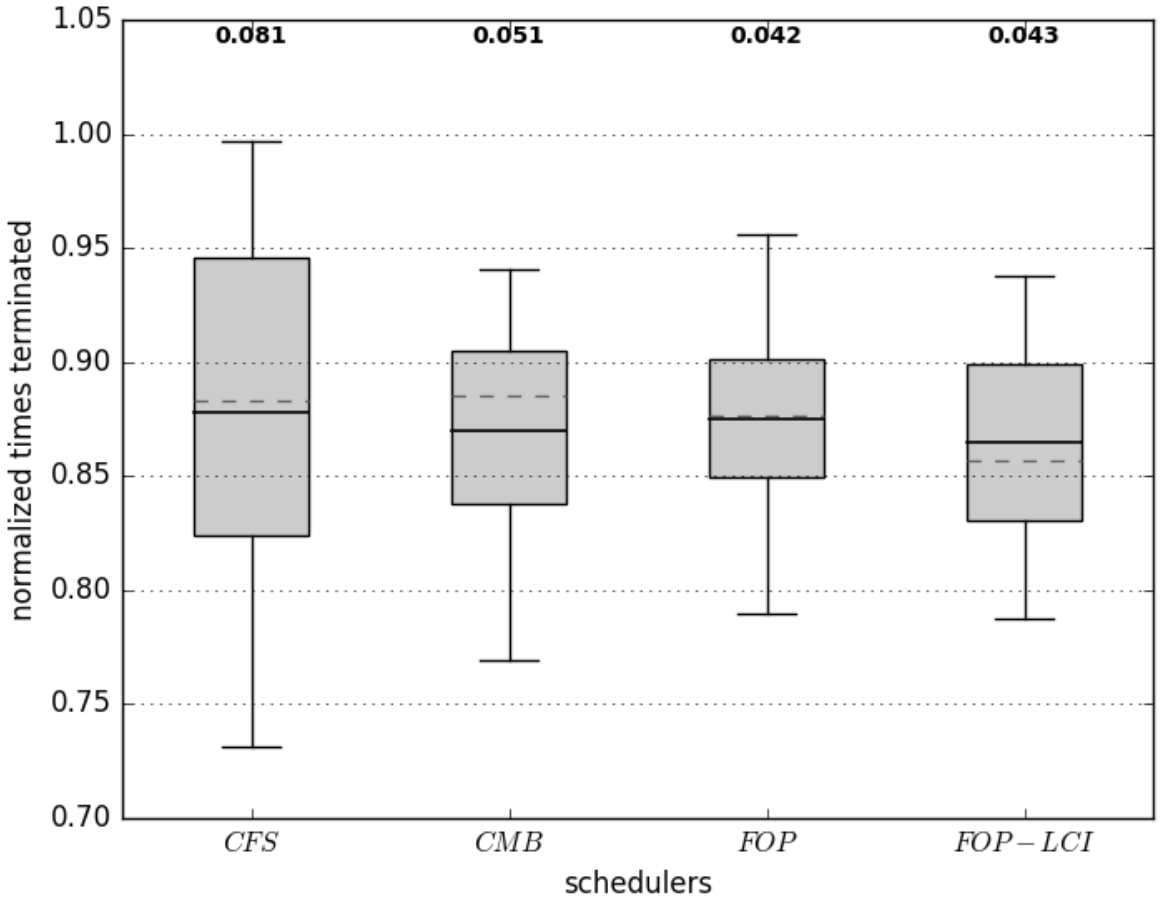


Figure 4.1-2: Workload 2 consists of 6 L, 4 LC, 4 SC and 2N applications

With a slight increase on the number of L applications, we notice that CFS widens the performance gap between the applications, as the SC applications are more likely to be executed with L, suffering their destructive interference. The throughput of our proposedschedulers is located at the same level with CFS. They all outperform Linux on the aspect of fairness, with the FOP showing the best behavior (lower standard deviation).

Workload 3

We replace 2 LC with 2 more SC applications, creating a workload of 6 L, 2 LC, 6 SC and 2N.

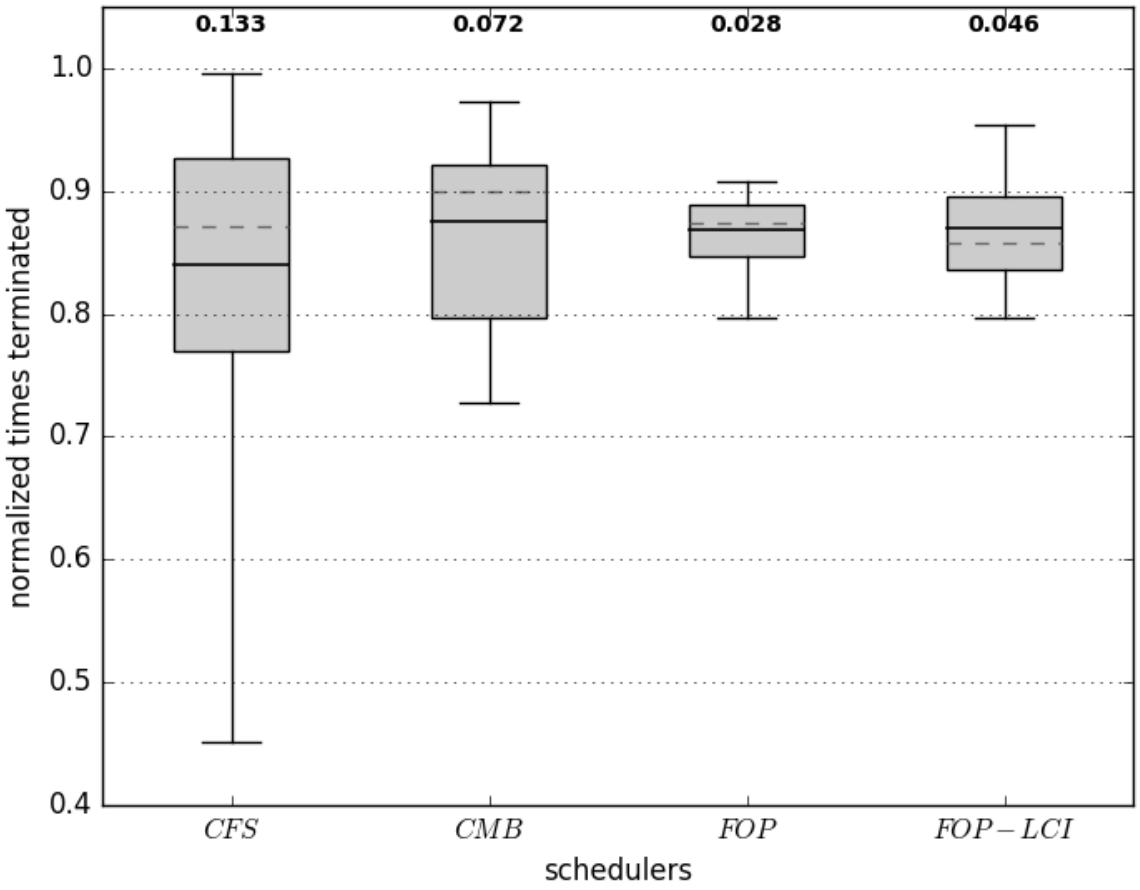


Figure 4.1-3: Workload 3 consists of 6 L, 2 LC, 6 SC and 2N applications

It is obvious that the contention agnostic treatment of Linux can lead to severe degradation of some applications (0.45 % of the ideal performance), performing overall with poor fairness and throughput comparing to our approaches. The isolation of SC applications occurring on the CMB scheduler provides a friendlier environment for their execution, boosting their performance and leading overall to the best throughput. FOP shows great results concerning fairness, as it is performing with the lower deviation. FOP-LCI performs at the same throughput with FOP and presents higher deviation.

Workload 4

In this workload we eliminate completely the LC applications, having 8 L, 6 SC and 2N.

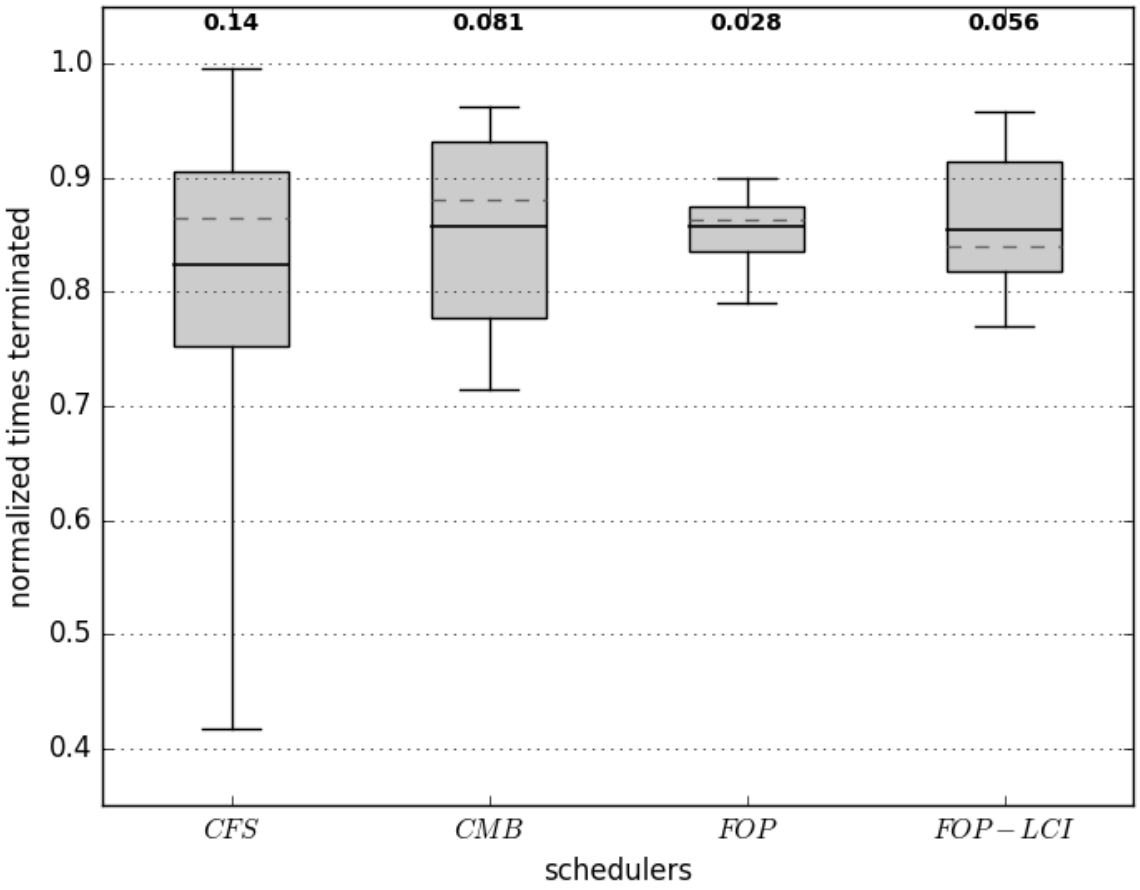


Figure 4.1-4: Workload 4 consists of 8 L, 6 SC and 2N applications

We observe that our proposed schedulers outperform Linux on both aspects of throughput and fairness. It is getting clearer that CFS fails to handle a workload with applications subjected to the destructive effects of contention (SC applications). The general picture of the results of our scheduling policies is similar to the previous workload. CMB performs at the same level with FOP in the matter of throughput. The FOP presents the most improved fairness. FOP-LCI, again, does not perform with higher throughput comparing to FOP.

Workload 5

At this point, we inspect the behavior of the schedulers, when it comes to handle applications of two classes exclusively (workload of 8 L and 8 SC). The contention level is getting higher, as the contention for the memory bandwidth and the trashing of the LLC caused by the L applications affects destructively their selves and the SC applications respectively.

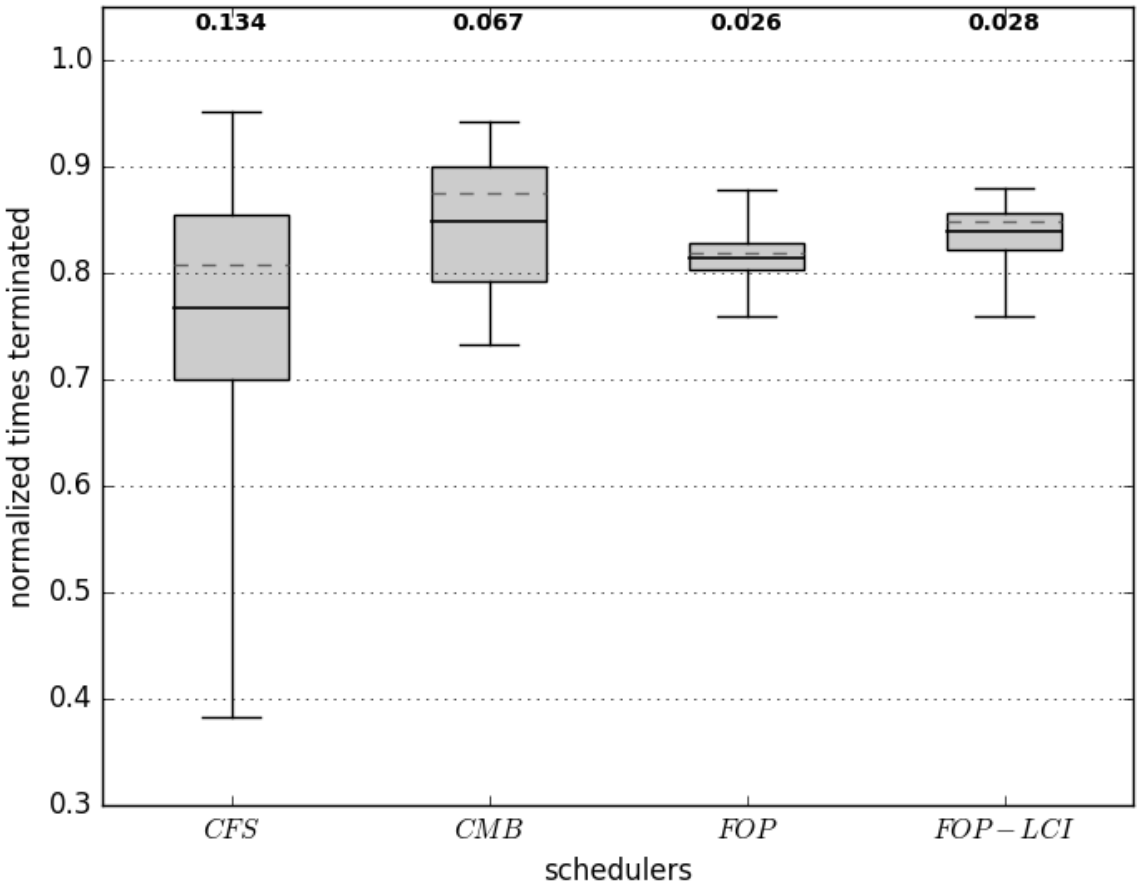


Figure 4.1-5: Workload consists of 8 L and 8 SC applications

Reaching the peak of the low contention environment, we come to the following conclusions. The isolation of the SC applications from the bad co-runners (L applications) and the balanced distribution of L applications, adapted on the contention aware approach, lead to the best results concerning throughput. The FOP scheduler manages to balance the performance of the applications near the average, boosting the suffering SC applications at the expense of the well performing L applications. The FOP-LCI lives up to our expectations for improved throughput comparing to FOP, since the SC applications are not co-executed with the unfriendly L. As a result they are not subjected to destructive interference and they utilize the CPU more efficiently (making greater progress). It is worth mentioning that FOP and FOP-LCI perform almost at the same fairness level.

Low contention environment comparison results

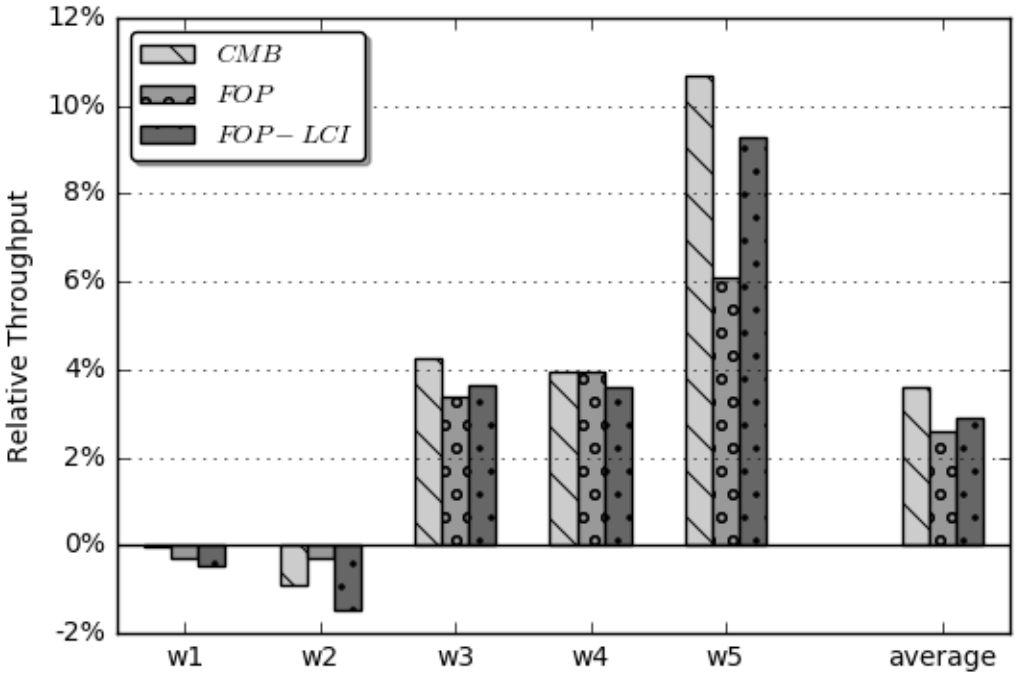


Figure 4.1-6: Relative throughput improvement comparing to Linux

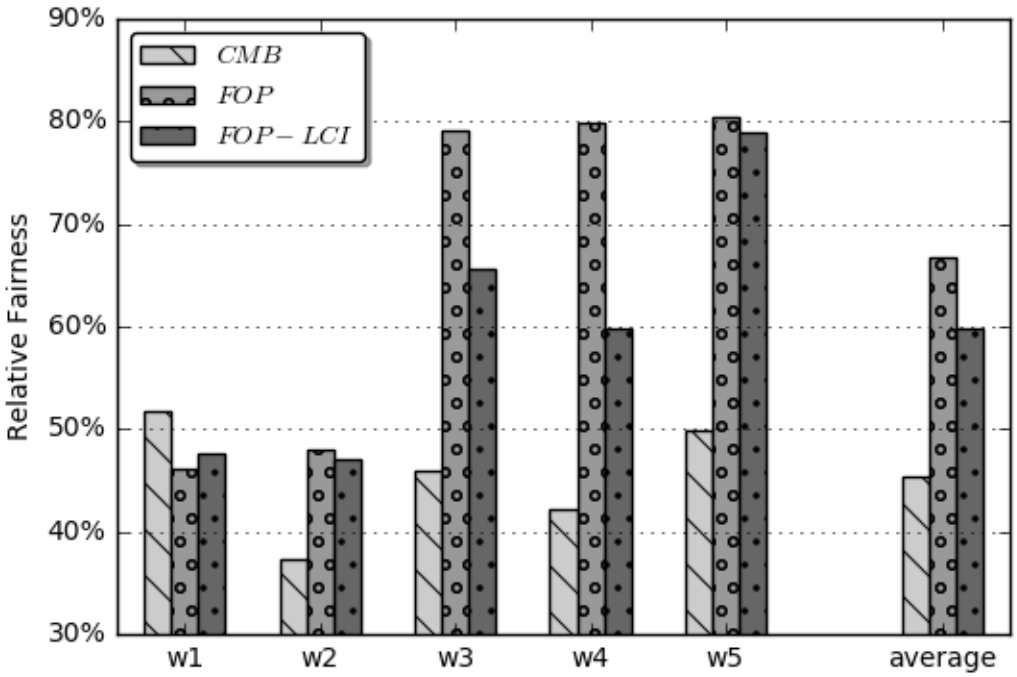


Figure 4.1-7: Relative fairness improvement comparing to Linux

The above graphs (Figures 4.1-6 and 4.1-7) depict the gains of throughput and fairness of our scheduling policies comparing to Linux scheduler for each workload and for the low contention environment (average). We observe that our techniques outweigh Linux scheduler regarding fairness on every workload of this environment. Concerning throughput CFS seems to perform better for workloads (first two) consisting of small number of contending applications (L and SC). On the other hand, when it comes to workloads with increased number of destructively interfering applications, our approaches outperform Linux respecting throughput.

Taking into account the graphs below (Figures 4.1-8 and 4.1-9), which present in detail the execution of each class on the different schedulers for two representative workloads, we analyze deeper the behavior of the proposed scheduling policies.

The CMB performs with the best throughput and the least improved fairness. The contention aware policy helps the most the SC applications. On the other hand, the L applications are slightly downgraded, because of their packing.

The FOP scheduler provides the best solution concerning fairness. Overall it performs with the lower deviation, balancing the applications near the average, as shown in the graphs below. It is worth mentioning that the frequent occupation of CPU with lower progress applications does not cause reduction in throughput, ranking FOP in the third place. This happens because the applications do not suffer from devastating interference and thus do not need significant increase on their running time. This, in turn, does not lead to great reduction of the running time of the well performed applications, keeping throughput on high levels.

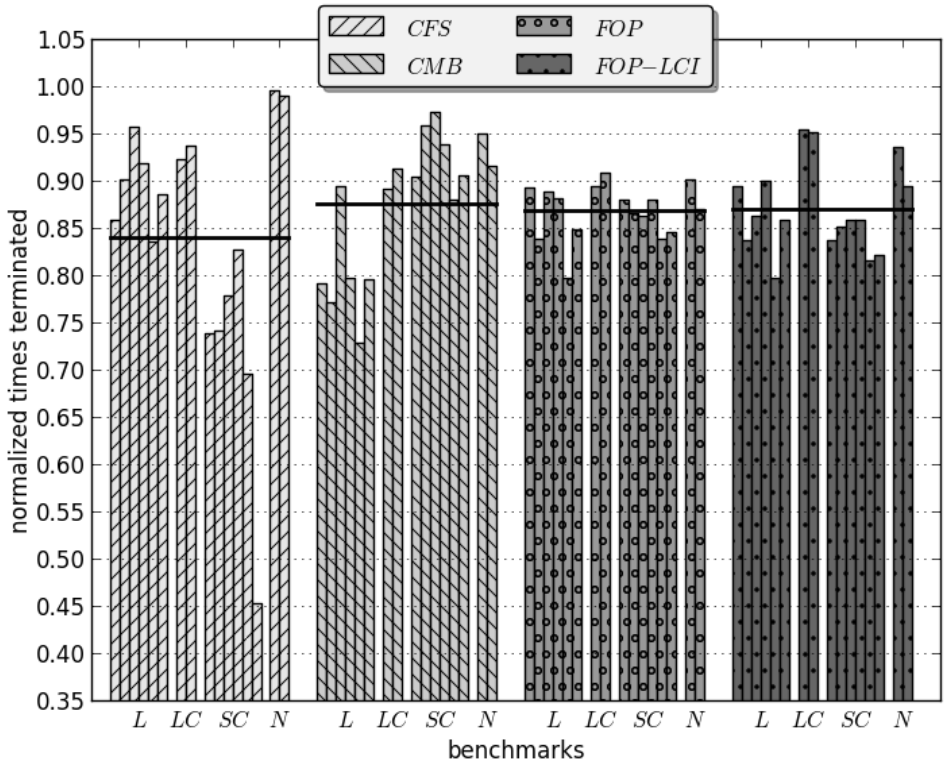


Figure 4.1-8: Detailed execution of workload 3

The FOP-LCI holds the second place among our schedulers on both aspects of fairness and throughput. Our intention is to separate the L-C pairing, in order to boost C applications performance and provide an improved throughput comparing to FOP. This is achieved only in the 5th workload. In the general case, it benefits the applications (LC and N) allowed to be executed with both L and C and keeps the C applications in the same level with FOP. Consequently the improved throughput is a result of LC and N rather than C boosting. One possible explanation is that the packing of L would result in higher IPC loss. This means that they are likely to be more frequently executed carrying with them the LC or N.

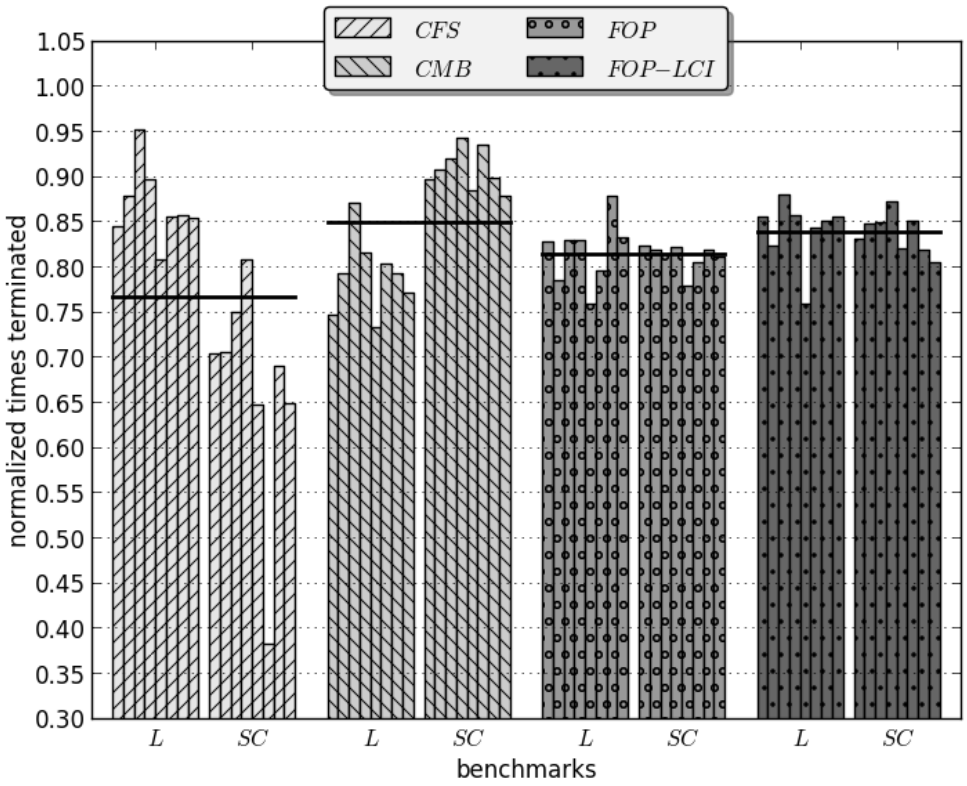


Figure 4.1-9: Detailed execution of workload 5

4.1.2 **Medium contention environment**

Workload 6

On this test group, we evaluate our schedulers when it comes to manage applications of the BC class, which suffer from heavier slowdowns but can be completely isolated via time sharing. On this workload we introduce 1 BC applications, having totally 4 L, 4 LC, 1 BC, 4 SC and 3 N.

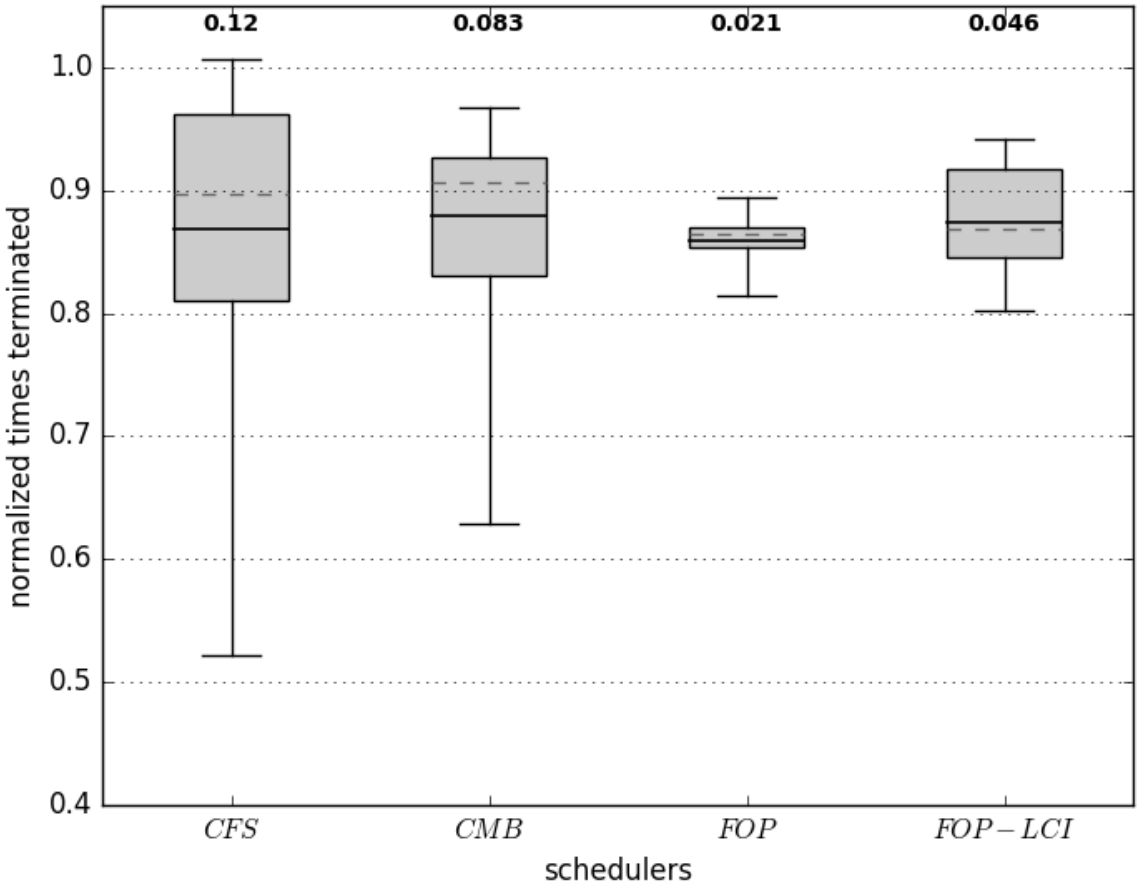


Figure 4.1-10: Workload 6 consists of 4 L, 4 LC, 1 BC, 4 SC and 3 N applications

We notice that FOP manages to distribute almost completely fair the resources, experiencing the lower deviation, with a small decrease in throughput. The CMB performs with the highest throughput and improved fairness. The FOP-LCI provides an intermediate solution for the inspecting characteristics.

Workload 7

We add another BC application, forming a workload of 4 L, 4 LC, 2 BC, 3 SC and 3N.

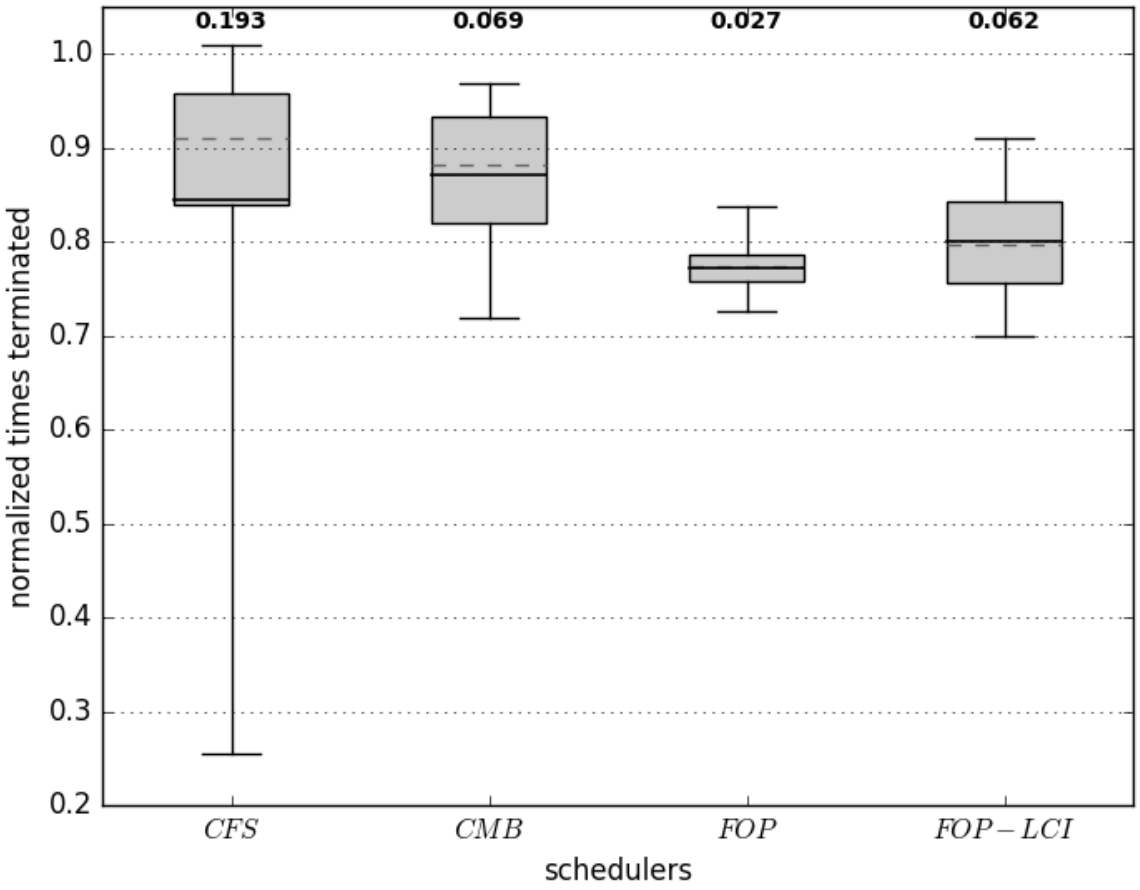


Figure 4.1-11: Workload 7 consists of 4 L, 4 LC, 2 BC, 3 SC and 3N

The CMB spreads the BC applications across the 2 available gangs. This offers a great performance boost for them and for the SC, leading overall to the highest throughput. In addition it experiences great improvement regarding fairness, as its deviation is really close to the FOP-LCI. The other approaches experience lower throughput than Linux, since they occupy the CPU mostly with applications that make low progress. It is becoming clearer that, except the unfair distribution, Linux cannot guarantee that applications would make adequate progress (application executing at fraction of 25% of its ideal performance).

Workload 8

We increase the number of L applications to 8, eliminating completely the LC. The workload now consists of 8 L, 2 BC, 3 SC and 3 N.

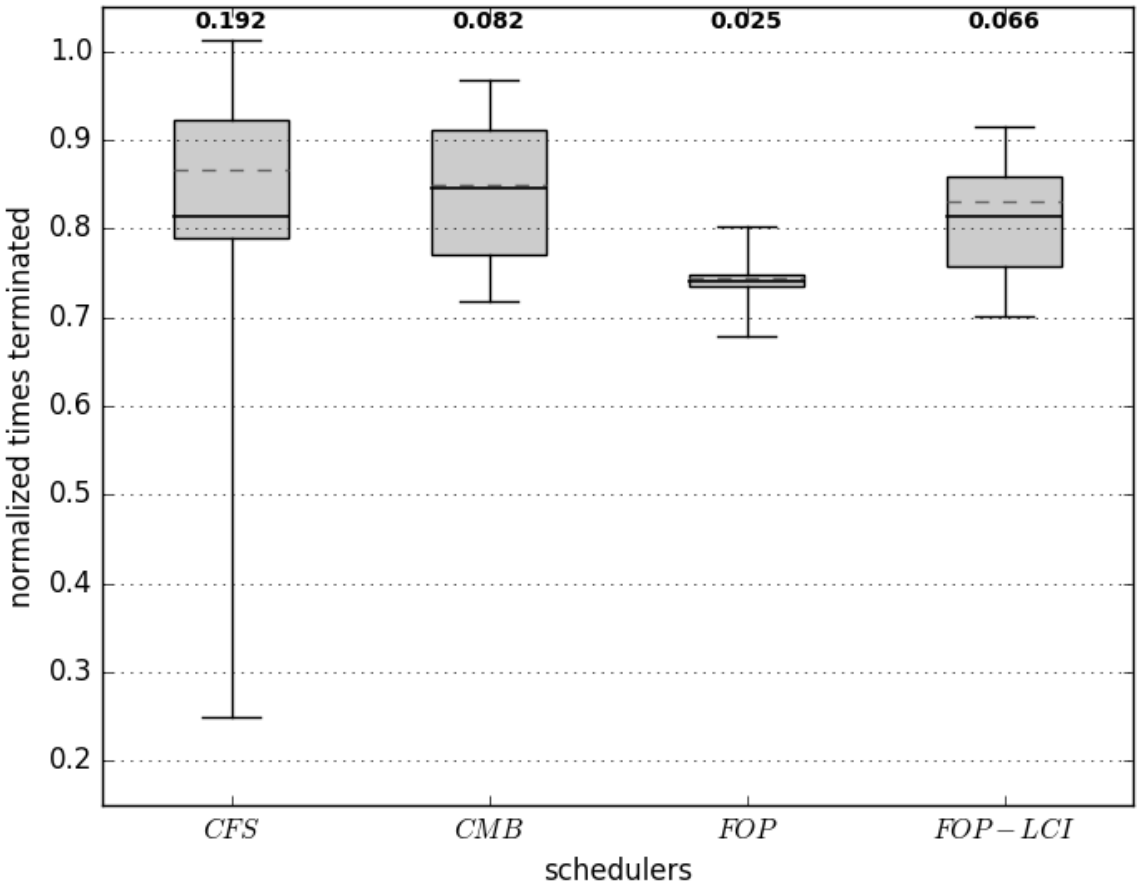


Figure 4.1-12: Workload 8 consists of 8 L, 2 BC, 3 SC and 3 N applications

The general picture presented on this graph is similar to the previous workload. Concerning the CMB scheduler, the BC applications are not affected with the increase of L, as they can be completely separated and the destructive interference can be avoided.

Workload 9

Now, we replace the N applications with SC. This workload contains 8 L, 2 BC and 6 SC.

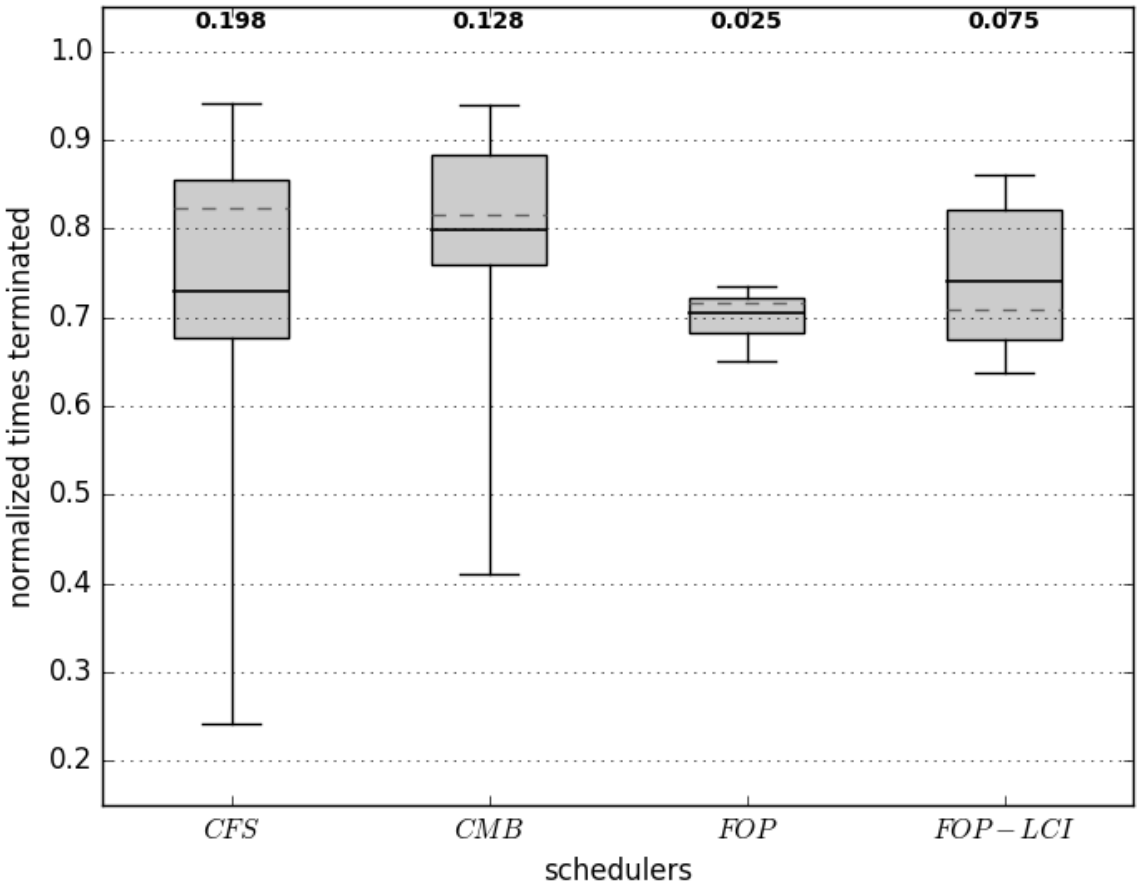


Figure 4.1-13: Workload 9 consists of 8 L, 2 BC and 6 SC applications

We observe that the replacement of N applications with SC causes greater problem for the CMB scheduler. Following its policy, it splits the BC and packs them together with the SC only. This leads to lower increase of BC applications' performance, while the slowdown imposed to them is higher when co-executing with SC applications rather than N (prediction model). Despite that fact, overall it performs with the highest throughput. The other schedulers follow similar behavior to the previous workloads.

Workload 10

This workload consists of 4 L, 3 BC, 4 SC and 4 N.

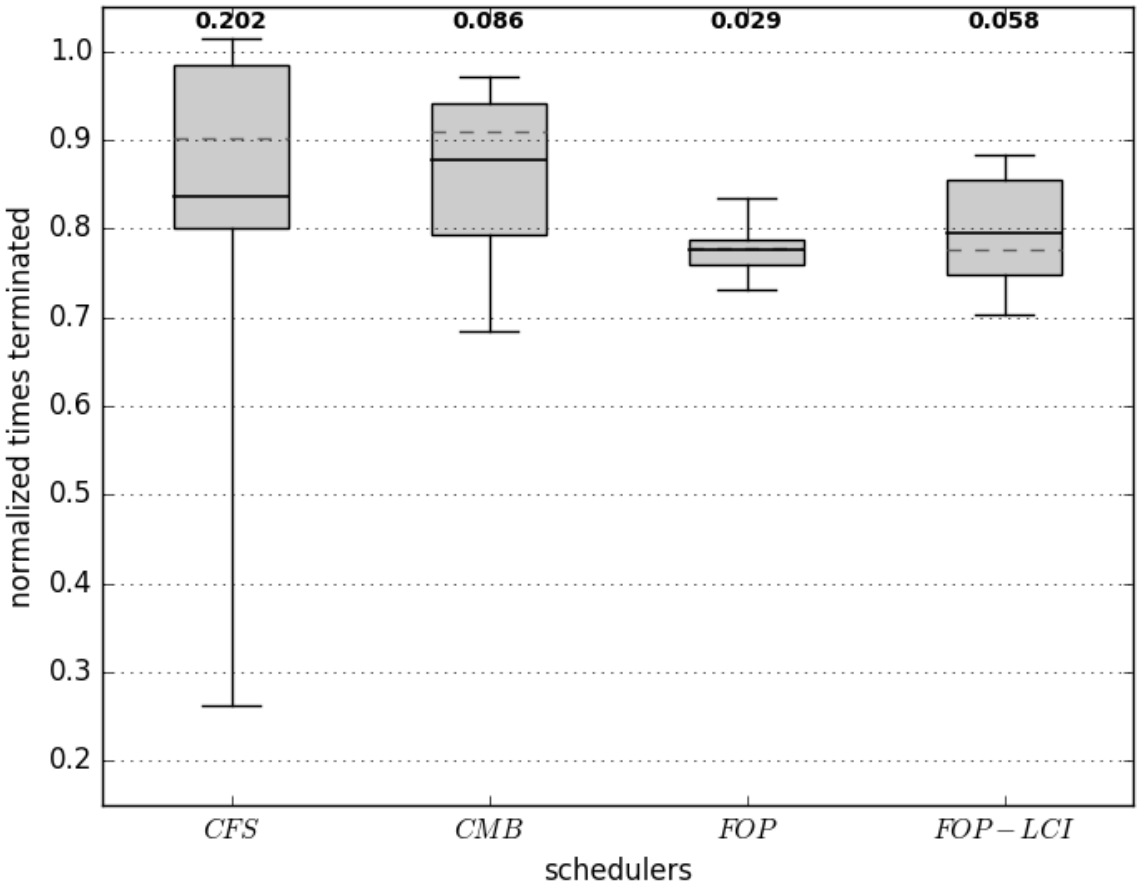


Figure 4.1-14: Workload 10 consists of 4 L, 3 BC, 4 SC and 4 N applications

As it is depicted on the graph above, our first approach manages to spread the 3 BC applications across the gangs, performing, again, with the best throughput and a satisfying improvement of fairness.

Medium contention environment comparison results

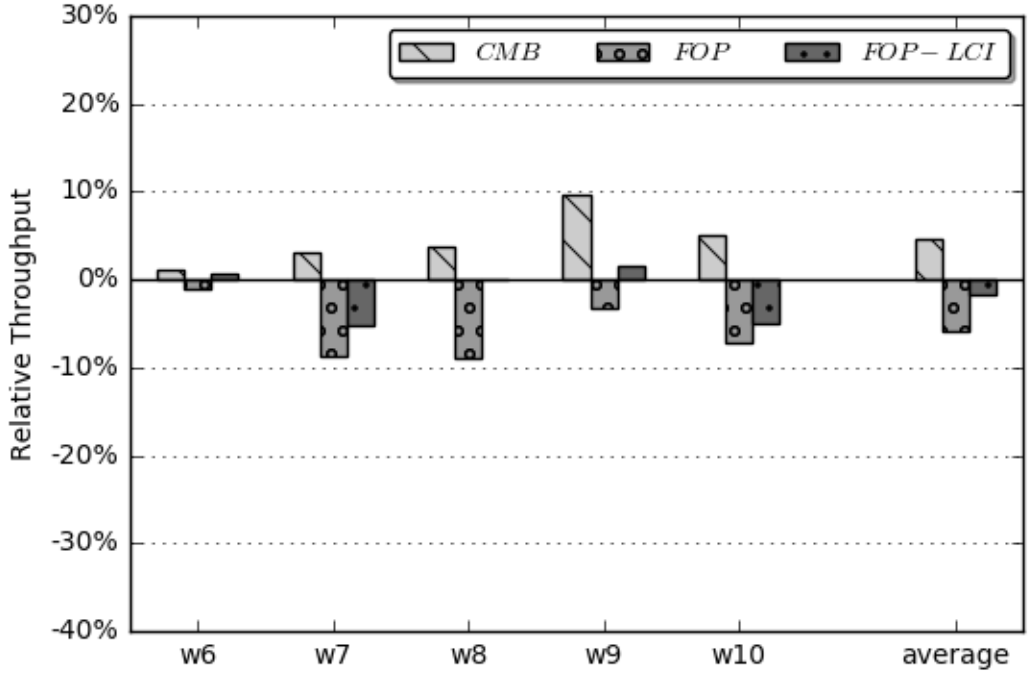


Figure 4.1-15: Relative throughput improvement comparing to Linux

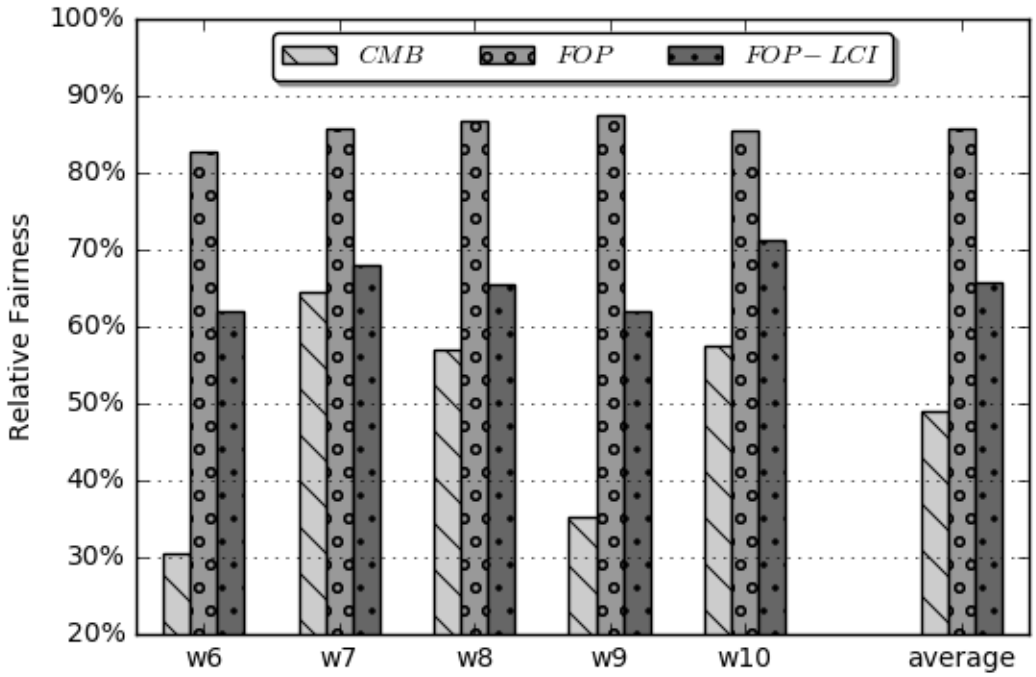


Figure 4.1-16: Relative fairness improvement comparing to Linux

Inspecting the above graphs (Figures 4.1-15 and 4.1-16), which depict the gains of throughput and fairness of our scheduling policies comparing to Linux scheduler for each workload and for the medium contention environment (average), we come to the following conclusions. All of our scheduling policies outperform Linux regarding fairness. When it comes to throughput, only CMB presents improvement across the workloads of this environment. FOP performs with lower throughput on every workload, showing a reduction of 6% on average. FOP-LCI increases throughput on two workloads only, performing on average with a 1.8% decrease.

Introducing the BC applications, it is getting clearer that Linux fails to manage the devastating effects of contention, providing an environment where those applications cannot make adequate progress and are lead to starvation (Figure 4.1-18). This results in greater unequal distribution of the resources among the applications. Comparing with the low-contention environment, our schedulers present higher improvement in the matter of fairness.

Taking into account two representative examples (Figures 4.1-17 and 4.1-18) of this environment, we are able to understand deeper the behavior of our schedulers.

The CMB manages to offer a great performance boost for the BC applications, when they can be completely isolated from the L and from each other. We observe that the improvement is dependent on the co-runners. Working with workloads consisting of N applications (Figure 4.1-17), the BC applications experience better improvement compared to workloads with SC applications only (Figure 4.1-18). This seems reasonable, since the prediction model implies zero slowdown for the execution pair BC-N and mild degradation for the pair BC-SC ($\times$ 1.46 on average (Figure 3.1-10)). As far as the L and LC applications are concerned, their performance is affected similarly to the workloads of the low contention environment. The packing of L with LC applications and with each other generates memory bandwidth contention and a higher degradation compared to Linux. Overall, it performs with the highest throughput and a satisfying fairness improvement.

When it comes to resource distribution, the FOP presents the fairest behavior. The figures below depict the tremendous results of the FOP performance. It manages to equalize the progress of the applications with a really negligible deviation around the average. This picture, also, confirms the correctness of the relationship between the factor and the gangs extracted previously. Opposite to the workloads of the low-contention environment, the throughput is suffering from a slight reduction. This phenomenon is explained by the fact that the CPU is mostly occupied by applications with low IPC ratio (BC class).

The FOP-LCI provides the second best solution for both characteristics of throughput and fairness. We notice that the avoidance of L-C pairing does not benefit the BC applications, as they are co-executed with each other and with LC, suffering from devastating damage ($\times$ 2.33 and $\times$ 2.22 slowdown respectively (Figure 3.1-10)). As a result, they get really often high in the waiting queue. Allowed to be co-executed only with SC, LC and N, the progress of L is slightly impeded regarding FOP. Similar to low-contention environment the increased throughput is due to the beneficial treatment of LC and N (executed with C and L) (Figure 4.1-17) plus the SC that perform well when isolated from L and are likely to be more frequent selected by BC (Figure 4.1-18)

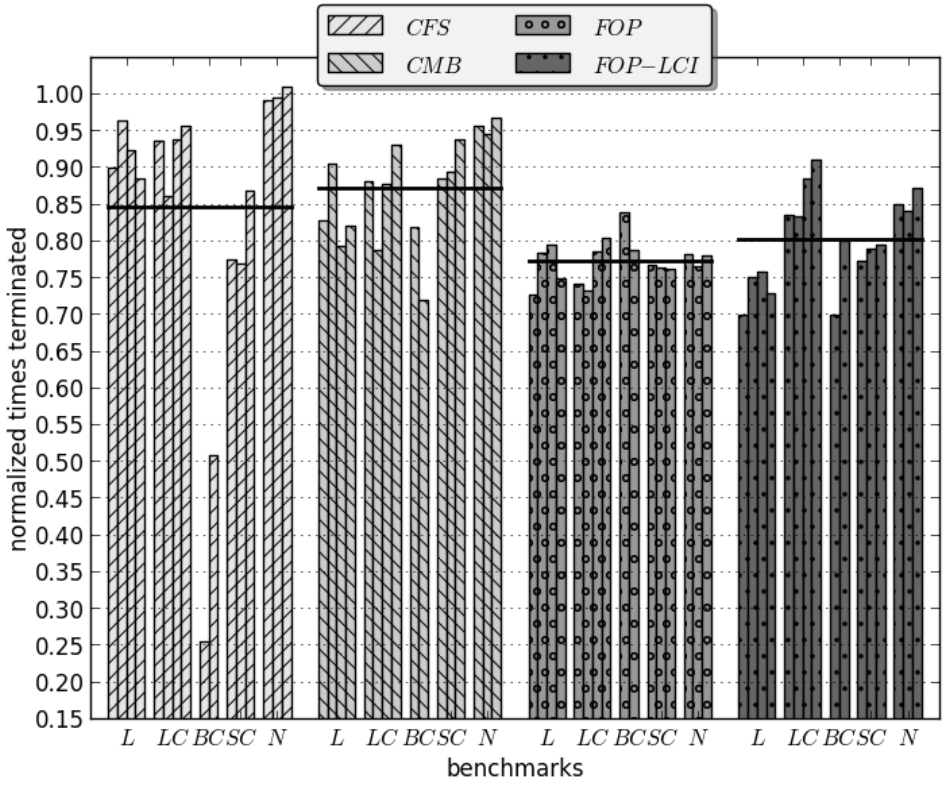


Figure 4.1-17: Detailed execution of workload 7

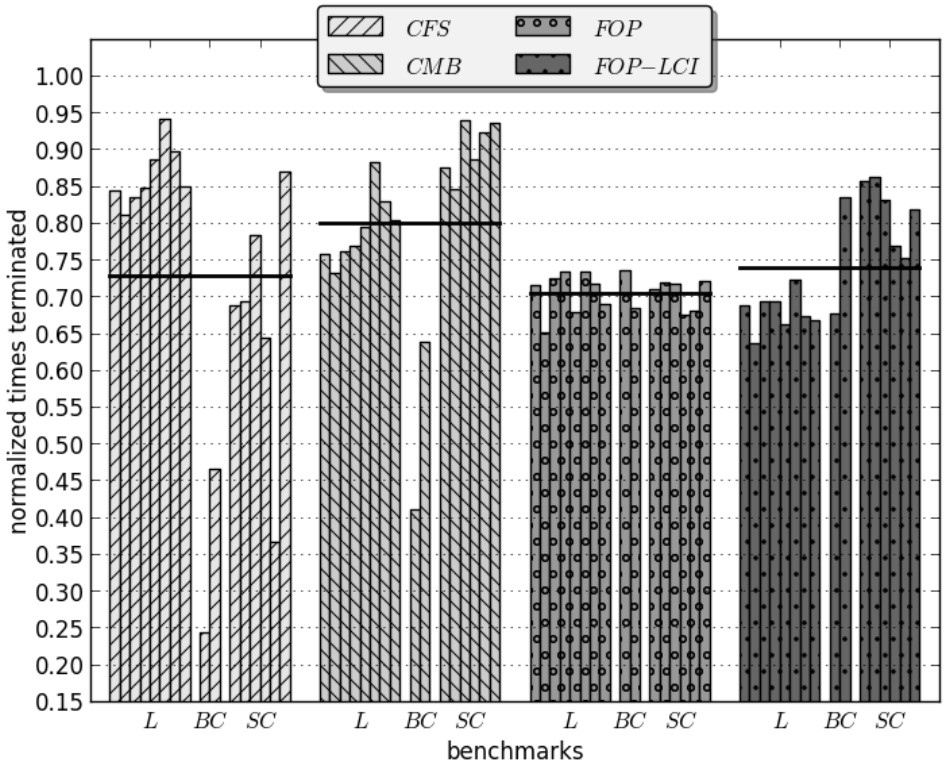


Figure 4.1-18: Detailed execution of workload 9

4.1.3 **High contention environment**

Workload 11

On this environment, we test our scheduling policies considering workloads with high number of BC applications, which cannot be isolated from each other. This means that our first approach would be forced to group them together and tolerate the heavy damage they cause. This workload consists of 4 L, 4 LC, 3 BC, 2 SC and 3N applications.

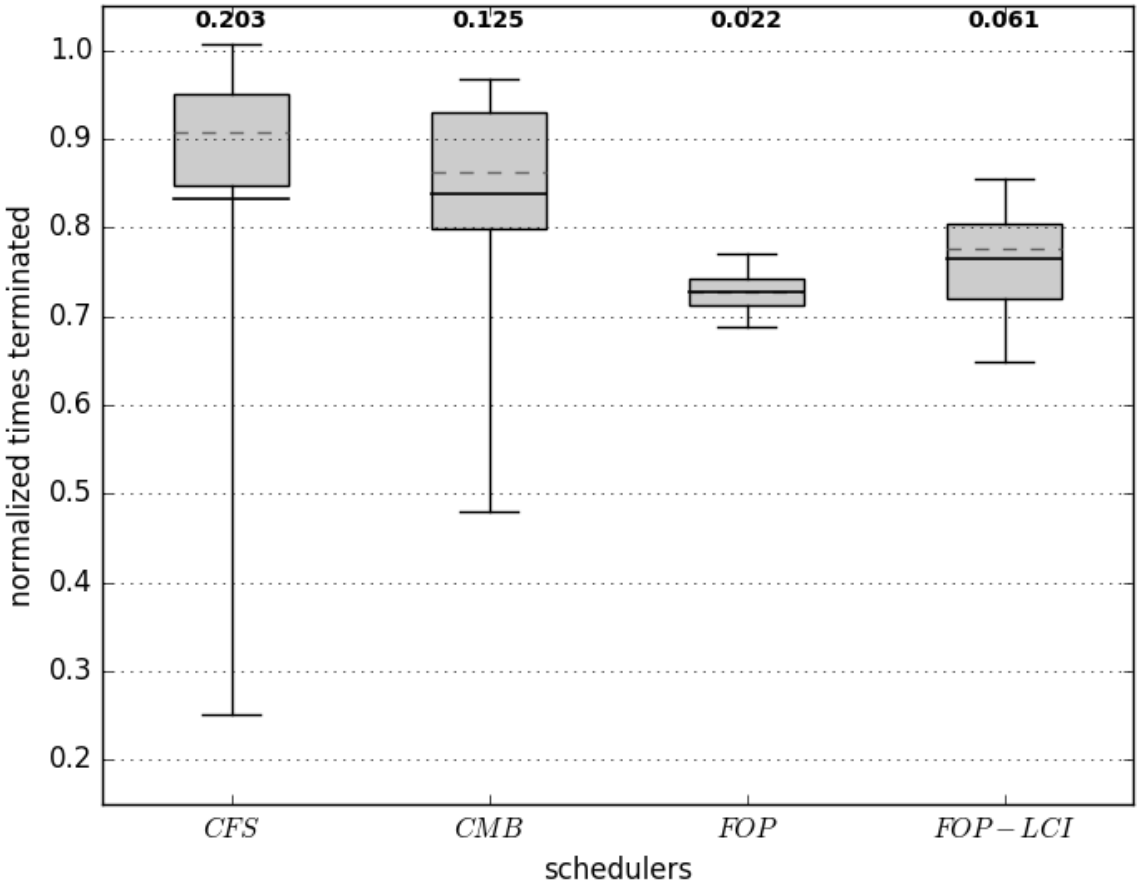


Figure 4.1-19: Workload 11 cosists of of 4 L, 4 LC, 3 BC, 2 SC and 3N applications

We observe that the effects of the BC pairing impact on the CMB performance, as it cannot provide sufficient improvement to them, in order to increase the throughput. However it is performing with lower deviation. The results of the other two schedulers are similar to previous workloads.

Workload 12

We increase the number of BC and L applications. We have 5 L, 3 LC, 4 BC, 2SC and 2N.

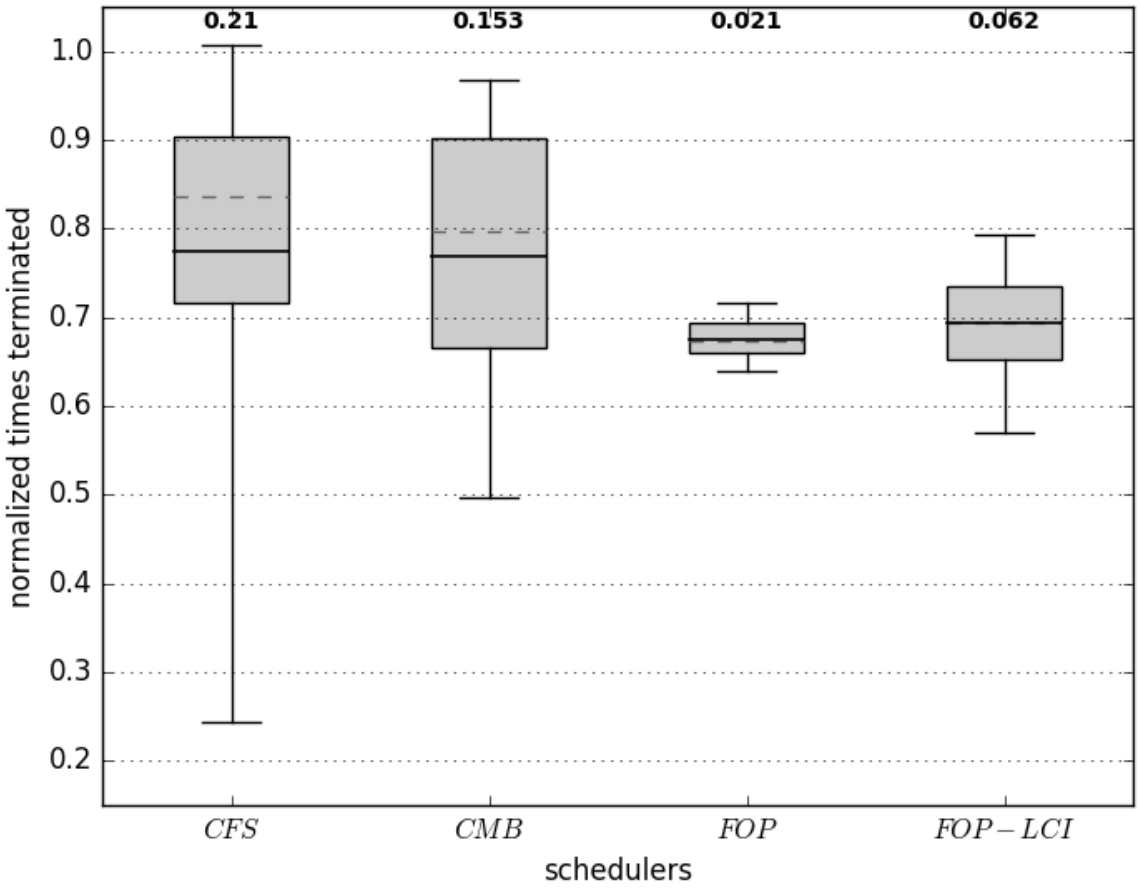


Figure 4.1-20: Workload 12 consists of 5 L, 3 LC, 4 BC, 2SC and 2N applications

It is getting clearer that CMB scheduler cannot boost BC applications' performance, remaining in the same throughput levels with Linux. Even the fairness improvement is declining. The general picture of FOP and FOP-LCI is similar to previous workloads.

Workloads 13, 14 and 15

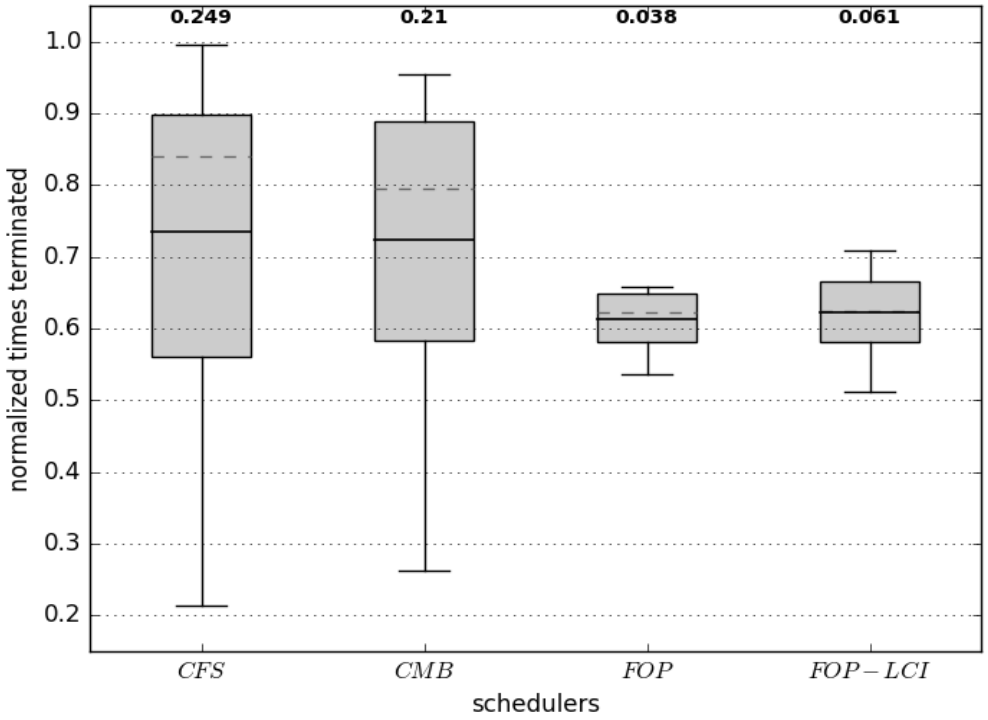


Figure 4.1-21: Workload 13 consisting of 6 L, 2 LC, 5 BC, 1 SC and 2 N applications

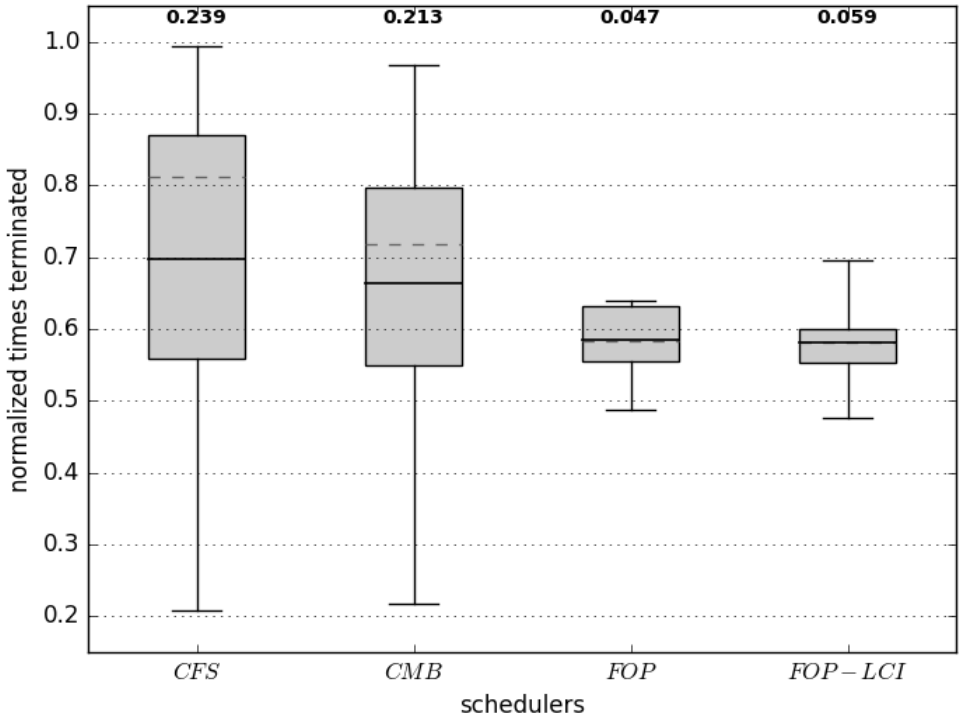


Figure 4.1-22: Workload 14 consisting of 7 L, 1 LC, 7 BC and 1 N applications

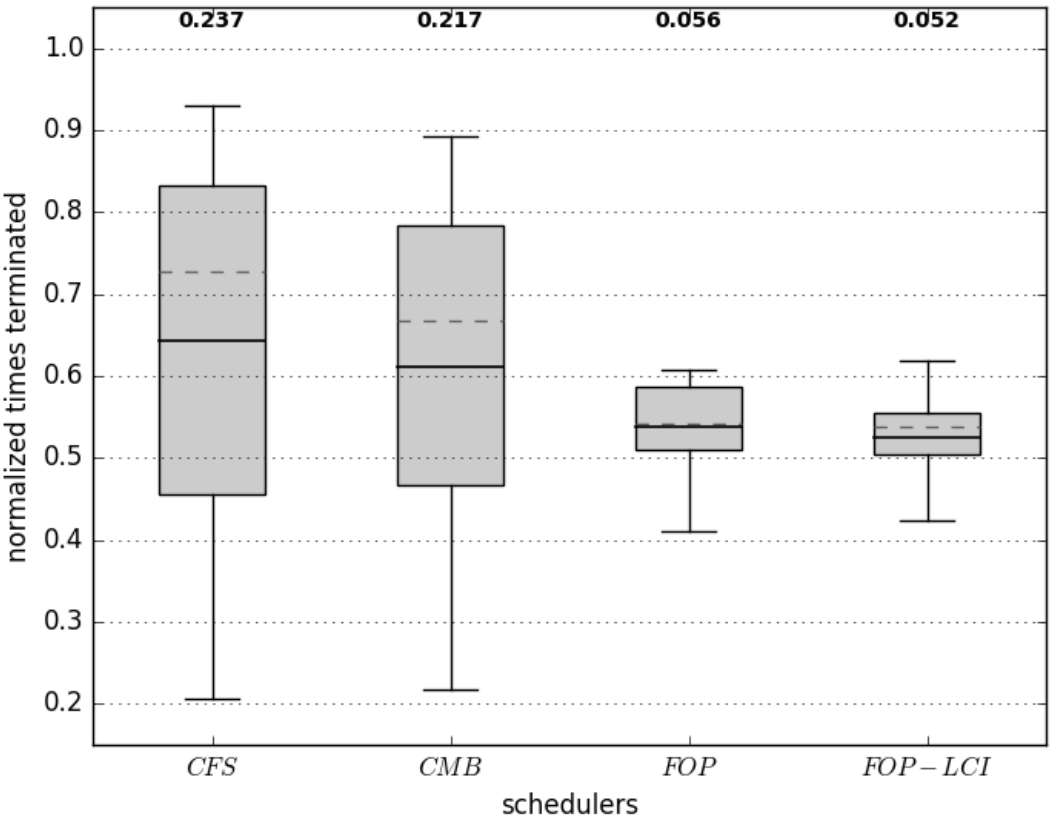


Figure 4.1-23: Workload 15 consisting of 8 L and 8 BC applications

Inspecting the above graphs, it is obvious that the policy of the CMB scheduler, packing the BC applications that cannot be spread across the gangs, harms its performance regarding both fairness and throughput. We see that the boxplots of Linux and CMB scheduler are almost identical, with the CMB providing reduced throughput and a slightly improved environment regarding fairness. In addition we notice that FOP-LCI is performing almost at the same throughput level with FOP. FOP distributes the resources in the fairest way, except the last workload, where FOP-LCI is slightly improved.

High contention environment comparison results

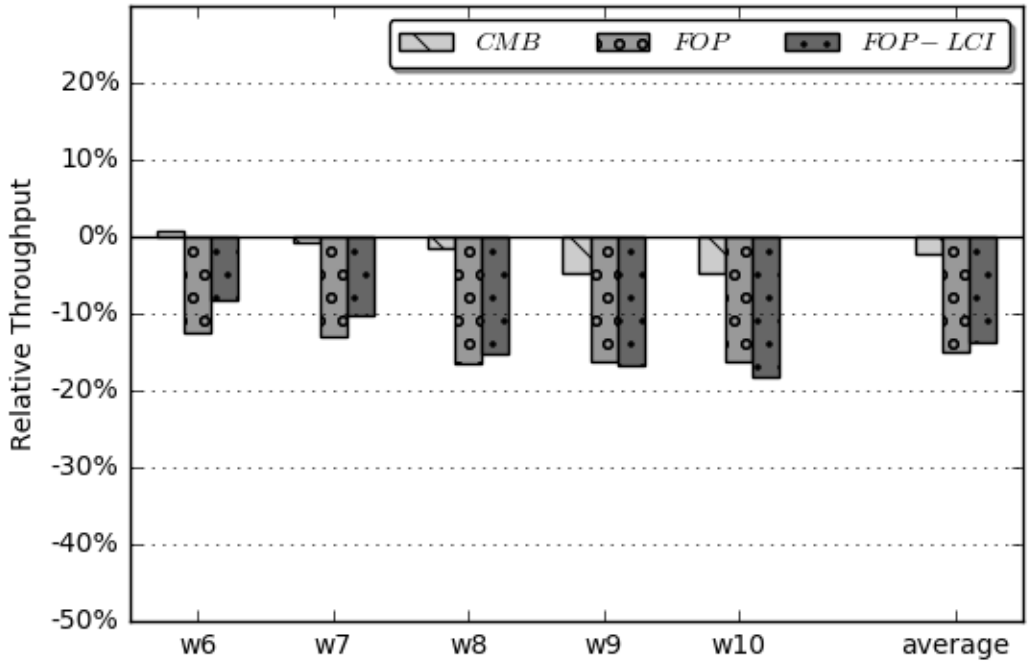


Figure 4.1-24: Relative throughput improvement comparing to Linux

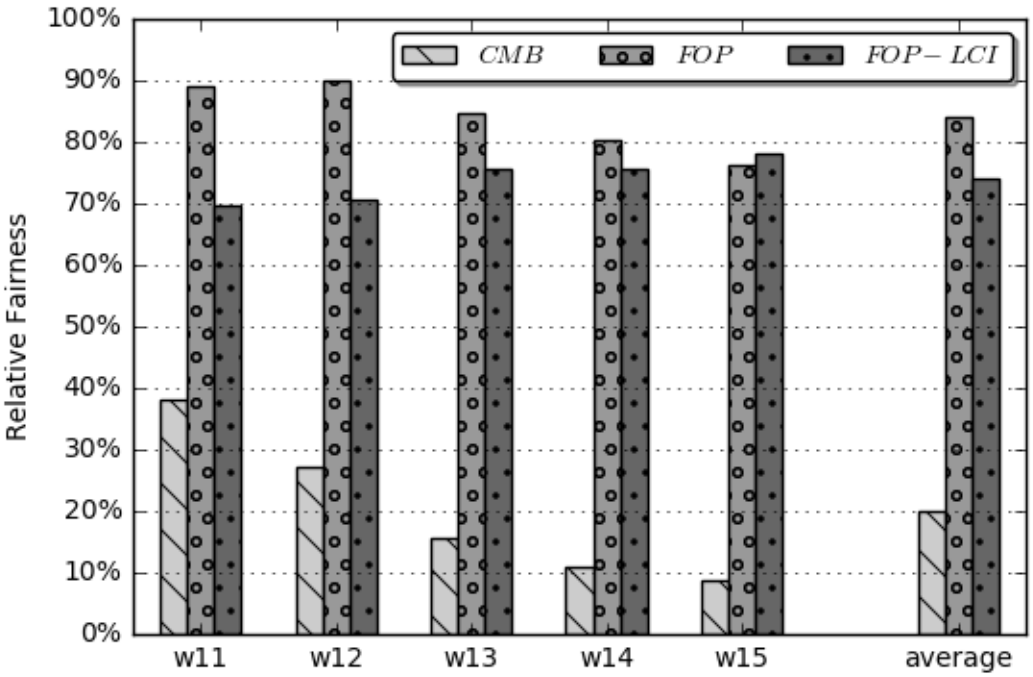


Figure 4.1-25: Relative fairness improvement comparing to Linux

Concluding, the graphs (Figures 4.1-24 and 4.1-25) present some interesting results for the heavy contention environment. Regarding throughput, it is clear that none of our scheduling policies is possible to provide improved performance, with the FOP and FOP-LCI experiencing higher decrease relative to medium-contention environment. Inspecting fairness, we understand that all of our schedulers outweigh Linux, with FOP and FOP-LCI performing at the same level and CMB with a satisfying decrease (- 30%) compared to the medium-contention environment.

It is worth mentioning how the rising of applications with devastating suffering (BC) influence the behavior of our scheduling policies. We describe in detail the impact on our schedulers.

The CMB fails to provide a friendly environment for the BC applications, as it is impossible to be completely isolated across the gangs. This proves our previous statement that, working on a package with increased number of cores sharing a cache, applications with great suffering are more likely to interfere with each other, while the distinction can be achieved only by time sharing. Our scheduler avoids execution of the BC-L packing. However when the number of BC applications far outweigh the available gangs it is obliged to group them together, tolerating the destructive effects this packing causes. Figure 4.1-26 presents clearly the general behavior of this scheduler, which imitates the Linux scheduler with lower throughput and a moderate improvement on fairness. Lower throughput is a result of the L packing, as they generate bandwidth contention. The performance degradation of L is the reason of the decreased throughput and the improved fairness. It achieves better deviation because of the lower progress of L and not the enhancement of BC applications, which was initially our purpose.

Concerning the FOP scheduler, the results, when it comes to fairness, are tremendous. It offers equal distribution of resources among the applications, improving fairness at 85% on average. Inspecting the graph below, we deduce that even applications with low progress cannot get any performance boost. This happens, because there are applications experiencing even greater decrease on their IPC getting prioritized on the waiting queue, occupying mostly the CPU time. This phenomenon leads to the satisfying reduction of 15% on throughput

Inspecting the FOP-LCI scheduler, we see that, following the policy of L-C isolation, fails to boost the performance of C, while the BC-BC pairing causes great degradation. Similar to the previous contention environment the improved throughput compared to FOP is a result of the LC and N beneficial treatment. These two classes are favored by the scheduler, as it is possible to be selected with both L and C applications (Figure 4.1-26).

Summarizing, on this environment we understand that with time sharing only applications that are mostly impacted by the contention cannot be completely isolated. Thus their performance cannot be enhanced and overall throughput cannot be improved (CMB scheduler). For the other scheduling policies we see that despite the rise of BC applications and the high contention levels, they are able to provide a fair distribution of resources across the applications.

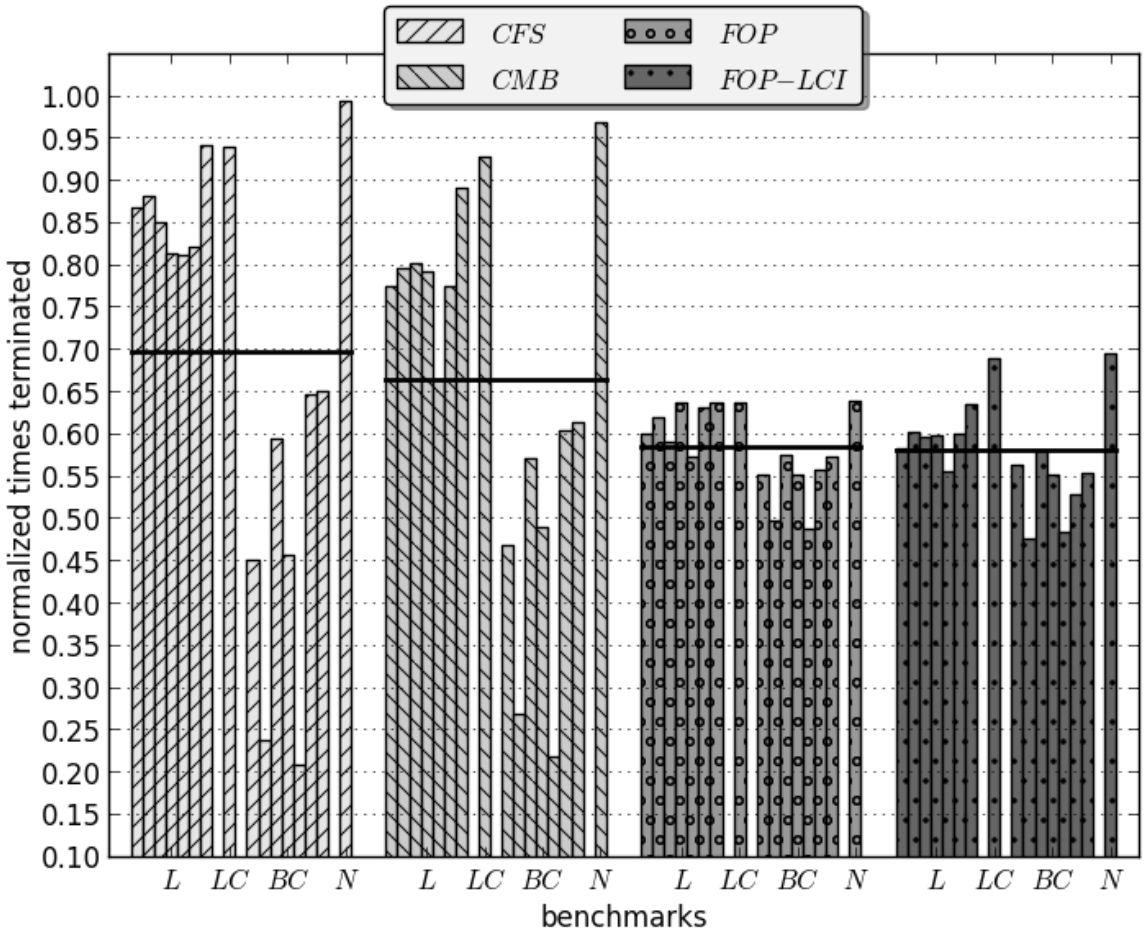


Figure 4.1-26: Detailed execution of workload 14

Chapter 5
Conclusion and Future Work

5.1 Results Evaluation

In this paper we focused on the problem of destructive interference of single-threaded applications with one execution phase when they are co-scheduled on multicore platforms that share architectural components, such as the Last Level Cache and the memory bandwidth. Inspecting the behavior of the system when applications are executed with state-of-the-art schedulers, like the CFS of Linux, we came to the following conclusions. The applications are treated in a contention agnostic way and are subjected to the effects of the destructive interference. As a result the overall performance of the system is negatively affected and the prime objectives that a scheduler tries to satisfy are totally impeded. We found that Linux scheduler is unable to share the resources of the platform equally among the applications, providing an unstable and workload dependent performance. This means that there is no guarantee that an application would make equal and stable progress with the other. Thus QoS (Quality of Service) guarantees cannot be provided to threads and SLAs (Service Level Agreements) are very difficult to enforce. We proposed three scheduling policies, aiming to mitigate the phenomenon of unfair distribution. On the first approach we built a contention-aware scheduler based on the classification scheme proposed by [1]. We try to improve fairness via contention avoidance and performance enhancement of the suffering applications, proposing a novel scheduler which take into consideration both LLC and memory bandwidth contention. On the second approach we enforce a policy that boosts the performance of suffering applications in expense of the ones that experience high IPC ratios, aiming to equalize their progress. The third approach is an extension of the previous policy, which select the threads in a contention aware manner, in order to create a friendlier environment for their execution and improve their performance. All of our proposed schedulers were tested and evaluated on a 4-core package sharing one LLC with a variety of workloads and have been compared to Linux scheduler. The results for the different contention environments are summarized on the graphs below.

Inspecting the Figure 5.1-1, it is clear that our scheduling policies achieved a tremendous improvement of the unfair distribution of resources provided by the Linux scheduler. Analyzing deeper the behavior of our schedulers, we come to the following conclusions.

Providing fairness through a thread-to-core placement policy (CMB scheduler) that avoids destructive interference is a possible solution. The suffering applications experience a great performance boost and the system throughput is improved. The main disadvantage is that our options for thread isolation are constrained, while only time-sharing is available. This leads to low throughput and a slight fairness improvement when it comes to workloads with suffering applications that cannot be completely isolated across the gangs.

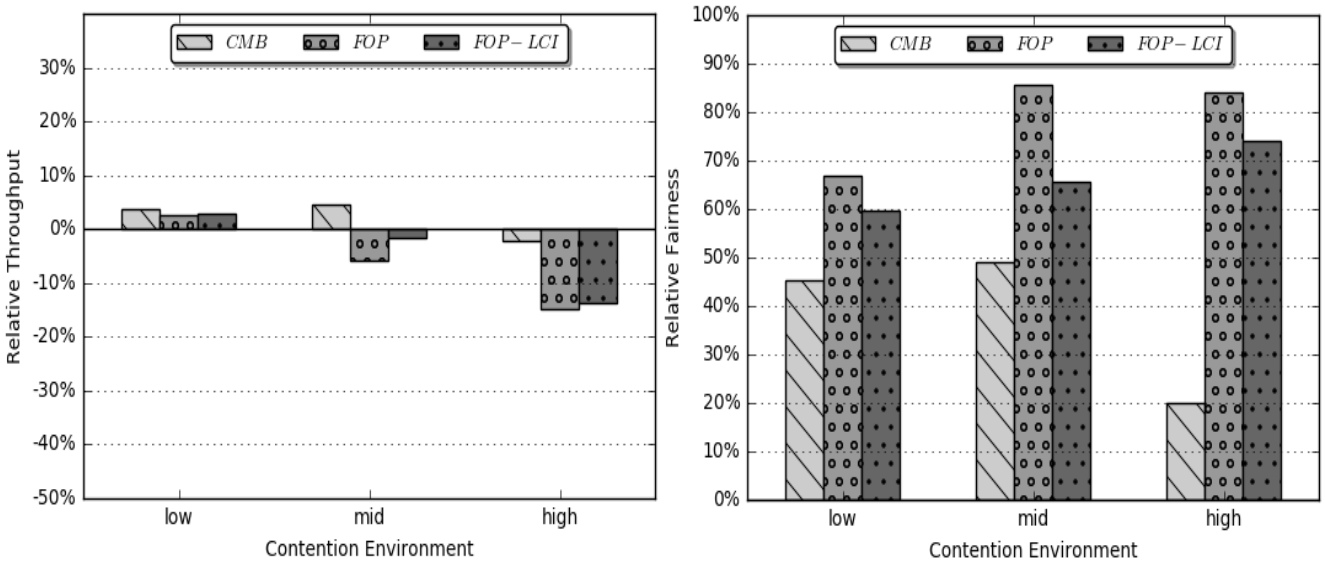


Figure 5.1-1: Throughput and fairness gains compared to Linux

Occupying the CPU with applications that have the gravest need concerning their progress and not their waiting time seems to be the best solution for equal distribution of resources. The results of the other two approaches show incredible fairness speedup. FOP presents the greatest fairness enhancement, while FOP-LCI experiences improved throughput and a slight decrease of fairness relative to FOP. Opposite to CMB, they are not affected by the rising of applications suffering from devastating interference, and manage to balance their progress across the different workloads.

Summarizing, we proposed three novel scheduling policies that take into consideration the destructive effects of resource sharing and manage to overcome the problem of unfair distribution. Balancing the performance of application, we assure that none would run at the expense of others. In addition, we prevent threads from experiencing starvation phenomena, providing an environment where all applications make adequate progress. In conclusion, all of our approaches manage to provide with QoS guarantees and make it possible for SLAs to be enforced. Concerning the first approach, a SLA could be established, assuring that all applications would share equally the CPU time and would perform well, as they do not interfere destructively with each other. Regarding the other two approaches, it is also possible for a SLA to be determined, taking into consideration that suffering applications would consume more CPU time but they could make adequate progress, equal to that of the other applications of the workload.

5.2 Future work

The work presented on this paper has prospect for expansion. While all of our metrics used are based on information acquired at run time via performance counters provided by the facilities of modern processors, the classification of C class cannot be performed online. The method we follow for the population of BC class requires investigation of the applications' behavior across different data sets. So, a technique for online identification of the working set could be developed, in order to make the CMB scheduler completely applicable to real environments. Opposite to CMB, the other two schedulers do not require additional adjustments and can be integrated as completely independent units on multicore architectures.

Additionally, our policies could be further extended, in order to handle more complex scheduling scenarios, including multi-threaded applications, applications with multiple execution phases, short-running tasks, I/O-bound applications and dynamic workloads. Finally, a very interesting scenario is to enhance our proposed schedulers, so that they could be integrated in NUMA architectures and in large scale cloud environments.

Chapter 6
Related Work

There has been significant interest in the research community in addressing shared resource contention on CMPs. The majority of work has focused on solutions which require elaborate modifications to hardware and can be distinguished into two categories: *performance aware cache modification* (commonly known as cache-partitioning) and *DRAM controller scheduling*.

Concerning the inefficient distribution of the shared LLC resources the LRU replacement policy impose, some researchers [24], [10], [17] proposed solutions to address this problem by explicitly partitioning the cache space based on how much applications would be benefited, rather than based on their demand rate. In the work on *Utility Cache Partitioning* [17], it has been developed a monitoring circuit, designed to estimate an application's number of hits and misses for all possible number of ways allocated in the cache (a technique based on stack-distance profiles (SDPs)). Then the cache is partitioned in such a way, so that the cache misses of a particular set of co-running applications would be minimized.

Tam et al. [25] addresses cache contention via software-based cache partitioning. Cache is partitioned using *page coloring*. Each application reserve a portion of the cache and the physical memory is allocated such that the application's cache lines map only into that reserved portion. The size of the allocated cache portion is determined based on the marginal utility of allocating additional cache lines for that application. Marginal utility is estimated via an application's reuse distance profile, acquired online utilizing hardware counters.

Approaches based on cache modification present some common characteristics. They typically rely on classification of the application execution behavior and a prediction model of the interference when they are executed together.

Xie and Loh [26] proposed a classification method based on *animals*. Each application may belong to one of the four classes named turtles, sheep, rabbits and devils. Applications that present low activity in the LLC are classified as turtles. Those that make use of the LLC but have zero sensitivity to the ways allocated to them belong to sheep. Rabbits are very sensitive to the ways allocated to them. Devils make heavy use of the LLC, while they experience high miss rates.

Similarly to Tam et al., Lin et al. [18] adapted page coloring and classified the applications into four colors based on their slowdown when executing on 1 MB compared to executing on 4 MB cache. Their scheme consists of two thresholds for the slowdown imposed to them. Applications showing over 20% degradation are considered red. Those suffering with slowdown between 5% and 20% are classified as yellow. Programs, degrading at a rate lower of 5%, are deeper classified as green or black according to their number of misses per thousand cycles.

Another approach by Jaleel et al. [13] categorize the applications into the following classes: Core Cache Fitting (CCF) have a dataset size that can be served by the lower levels of the memory hierarchy (private caches) and do not benefit from the shared LLC. LLC Trashing (LLCT) have a dataset larger than the size of LLC. Such applications cause great damage to

those that get highly benefited by the LLC utilization. LLC Fitting (LLCF) require the majority of the LLC capacity to perform well and get degraded when they do not receive it. LLC Friendly (LLCFR) benefit from the available shared LLC but their performance does not degrade significantly when they do not receive the bulk of its space.

Taking into consideration the DRAM memory system, researchers [27] found that the FR-FCFS policy implemented on the DRAM controllers leads to poor system throughput in CMPs and to potential starvation of threads with low row-buffer locality. Therefore, memory aware schedulers have been proposed in several studies, aiming to equalize the DRAM-related slowdown experienced by each thread because of interference. Nesbit et al. [28] proposed the *fair queuing memory* scheduler (FQM). For each thread, in each bank, FQM keeps a counter called virtual runtime. When a memory request of the thread is serviced, the scheduler increases this counter. This approach prioritizes the treads with the earliest virtual time, in order to balance the progress of each thread in each bank. Another algorithm of special interest is the *parallelism-aware batch scheduling algorithm* (PAR-BS) [29], which provides with higher memory throughput and exploits per-thread bank-level parallelism more efficiently than FQM. It attempts to minimize the average stall time by giving higher priority to requests from the thread with the shortest stall time at a given instant. This scheduler relies on the idea of batches. Regarding systems with multiple DRAM controllers, ATLAS (*Adaptive per-Thread Least-Attained-Service*) [30] manages to provide superior scalability and throughput than PAR-BS. It divides execution time into quanta, during which each controller makes scheduling decisions locally based on a system-wide thread ranking, in order to reduce the coordination between memory controllers.

It is worth mentioning that the techniques discussed previously may seem promising but require significant changes to the underlying hardware and the operating system. Thus it is possible to be evaluated only in simulation and it is very difficult to be implemented in commercial systems.

In contrast to cache partitioning and DRAM controller scheduling, a part of research community embraces a different trend to deal with the contention of resource sharing, the thread-level scheduling. According to this approach, studies focus on finding the appropriate thread-to-core mappings that generate the friendlier contention conditions for the execution of applications and thus lead to the best possible performance. They attempt to classify programs based on simple metrics that can be easily collected via the performance monitoring facilities of modern processors and predict their interference through simple heuristics.

Banikazemi et al. [31] attempted to predict the performance of a thread in a particular mapping, calculating the cache occupancy ratio, which is the ratio of the LLC access rate of a specific thread to the LLC access rate of all the threads sharing a cache in that mapping. Then they calculated the possible LLC miss rate the thread would experience, based on the currently measured LLC miss rate, the cache occupancy ratio and a rule of a heuristic. Finally, they use a linear regression model to convert the predicted LLC miss rate into a predicted CPI for the thread, which directly represents the predicted performance of a thread in that mapping.

Opposite to the complex method adapted by Banikazemi, some researchers [32], [14], [12] preferred to approximate contention with one simple heuristic: *the LLC miss rate*. Based

on this simple performance metric, they observed that applications with high LLC miss rate stress the entire memory hierarchy for which they are competing and tried to keep them separated, in order to avoid poor performance. They use concepts like cache light/heavy threads [32] and turtle/devils [14] in order to characterize low and high miss applications, rather than converting this intuition into a numerical prediction.

Once a prediction model has been established, a scheduler must decide how the threads should be combined in order to satisfy the desired objective.

Jiang et al. [33] approximated the problem in graph-theoretical form. The nodes of the graph represent the applications to be executed. The weighted edges between them represent the performance degradation occurring to these applications when they run concurrently sharing a LLC, assuming that the performance degradation of every possible pair of threads is known. They showed that the optimal scheduling solution to this problem is the *minimum weight perfect matching* of the graph.

Based on the simple LLC miss rate performance metric, Zhuravlev et al. [14] proposed their *distributed intensity (DI)* scheduling policy. According to that, all threads are sorted based on their miss rate. Then the threads are paired in the following way: the thread with the highest miss rate is co-scheduled with the one presenting the lowest miss rate, the thread with the second highest miss rate is co-scheduled with that with the second lowest miss rate, etc. In that way it achieves to balance the miss rate (intensity) across the threads.

Merkel et al. [12] developed its decision-making policy based on the *activity vectors.* A thread’s activity vector represents its usage of system resources; the memory bus, the LLC and the rest of the core. Their scheduler exploits thread migration to co-schedule threads with complementary activity vectors. Programs with high variability of activity vectors are likely to yield to high performance if co-scheduled in a complementary manner. They introduce the sorted co-scheduling, which based on a parameter of the activity vector keeps the run queue of one core sorted in descending order and the run queue of the other core in ascending. This way, they attempt to co-schedule pairs of threads with complementary activity vectors.

Bibliography

[1] Haritatos A. H. et al., "LCA: a memory link and cache-aware co-scheduling approach for CMPs," in *In Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 469-470.

[2] McCalpin J. D., "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19-25, 1995.

[3] pChase benchmark. [Online]. https://github.com/maleadt/pChase

[4] Michael L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 5th ed. New York, USA: Springer.

[5] Geer D., "Industry trends: Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11-13, 2005.

[6] Silberschatz A., Galvin P. B., and Gagne G., *Operating System Concepts*, 9th ed.: John Wiley & Sons, 2012.

[7] Zhuravlev S., Juan Carlos Saez, Blagodurov S., Fedorova A., and Prieto M., "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys*, vol. 45, no. 1, pp. 1-28, November 2012.

[8] Hsu L. R., Reinhardt S. K., Iyer R., and Makineni S., "Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT'06)*, 2006, pp. 13-22.

[9] Srikantaiah S., Kandemir M., and Irwin M. J, "Adaptive set pinning: managing shared caches in chip multiprocessors," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, 2008, pp. 135-144.

[10] Chandra D., Guo F., Kim S., and Solihin Y., "Predicting inter-thread cache contention on a chip," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, 2005, pp. 340–351.

[11] Kim S., Chandra D., and Solihin Y., "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*, 2002, pp. 111-122.

[12] Merkel A., Stoess J., and Bellosa F., "Resource-conscious scheduling for energy efficiency on multicore processors," in *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*, 2010, pp. 153-166.

[13] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, and Simon, "Cruise: Cache replacement and utility-aware scheduling," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII*, New York, USA, 2012, pp. 249-260.

[14] Zhuravlev S., Blagodurov S., and Fedorova A., "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*, 2010, pp. 129-142.

[15] Michael E. Thomadakis, The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms, 2011, Ph. D. Supercomputing Facility.

[16] Intel Performance Counter Monitor - A better way to measure CPU utilization. [Online].

http://software.intel.com/en-us/articles/intel-performance-counter-monitor

[17] Qureshi M. K. and Patt Y.N., "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*, 2006, pp. 423-432.

[18] Lin J. et al., "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'08)*, 2008, pp. 367-378.

[19] Χαλιός Χαράλαμπος, Δρομολόγηση Παράλληλων Εφαρμογών σε Πολυπύρηνα Συστήματα, 2013, Diploma thesis, School of Electrical and Computer Engineering, N.T.U.A.

[20] CGroups - The Linux Kernel Archives. [Online]. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[21] freezer. [Online]. https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt

[22] Cpusets - The Linux Kernel Archives. [Online]. https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt

[23] Box Plot: Display of Distribution. [Online]. http://www.physics.csbsju.edu/stats/box2.html

[24] Suh G. E., Rudolph L., and Devadas S., "Dynamic partitioning of shared cache memory," *J. Supercomput.28*, vol. 28, no. 1, pp. 7-26, 2004.

[25] Tam D. K., Azimi R., and Soares L. B. and Stumm M., "RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations," in *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, 2009, pp. 121-132.

[26] Xie Y. and Loh G., "Dynamic classification of program memory behaviors in CMPs," in *Proceedings of CMP-MSI, held in conjunction with ISCA-35*, 2008.

[27] Moscibroda T. and Mutlu O., "Memory performance attacks: Denial of memory service in multi-core systems," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007, pp. 18:1-18:18.

[28] Nesbit K. J., Aggarwal N., Laudon J., and Smith J. E., "Fair queuing memory systems," in *Proccedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*, 2006, pp. 208-222.

[29] Mutlu O. and Moscibroda T., "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*, 2008, pp. 63-74.

[30] Kim Y., Han D., Mutlu O., and Harchol-Balter M., "ATLAS: A scalable and high- performance scheduling algorithm for multiple memory controllers," in *Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture (HPCA'10)*, 2010, pp. 1-12.

[31] Banikazemi M., Poff D., and Abali B., "Pam: A novel performance/power aware meta- scheduler for multi-core systems," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*, 2008, pp. 1-12.

[32] Knauerhase R., Brett P., Hohlt B., Li T., and Hahn S., "Using OS observations to improve performance in multicore systems," *IEEE Micro*, vol. 28, no. 3, pp. 54-66.

[33] Jiang Y., Shen X., Chen J., and Tripathi R., "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*, 2008, pp. 220-229.

[34] The freezer subsystem. [Online]. https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt

Appendix

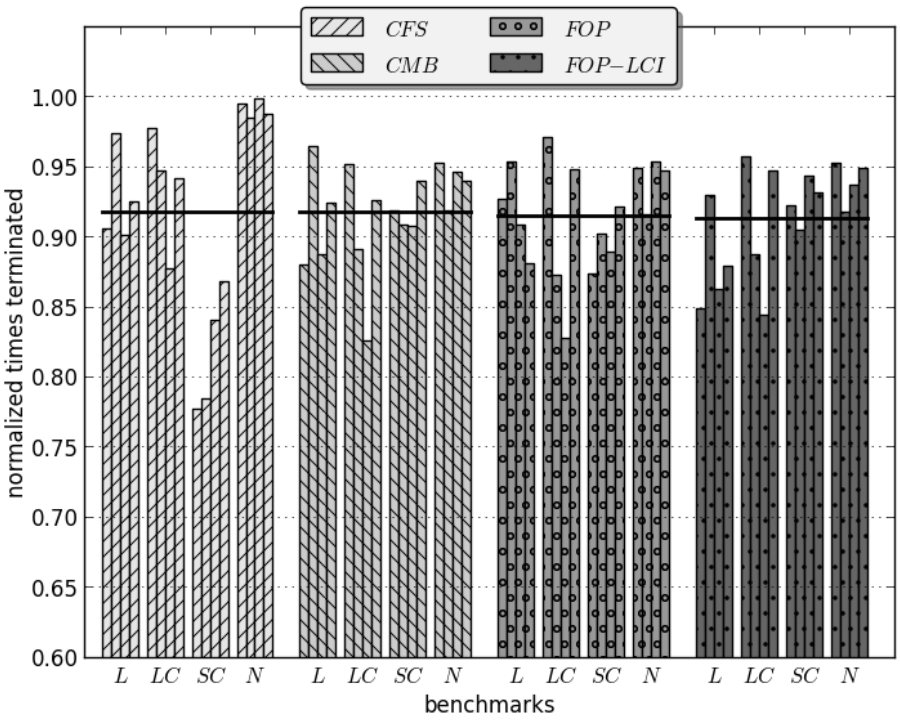


Figure A-5.2-1: Workload 1

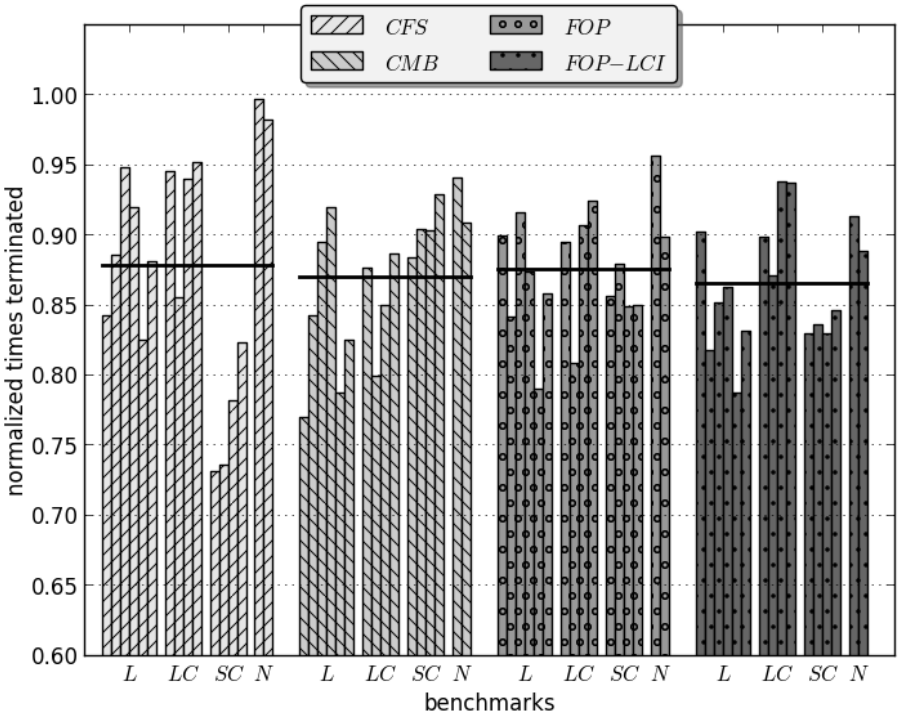


Figure A-5.2-2: Workload 2

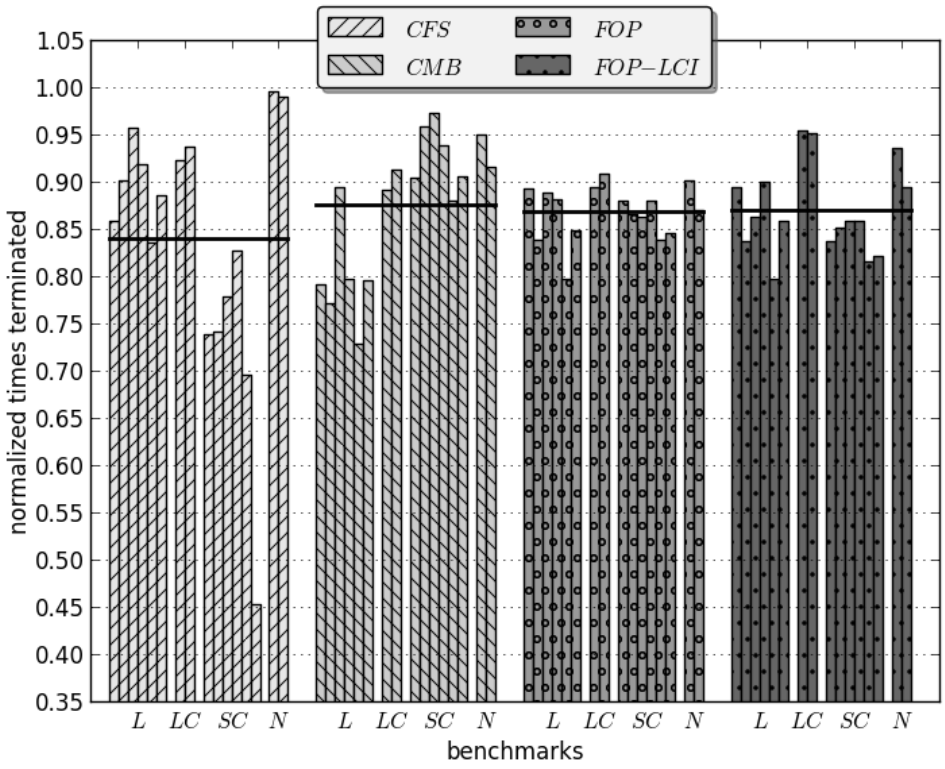


Figure A-5.2-3: Workload 3

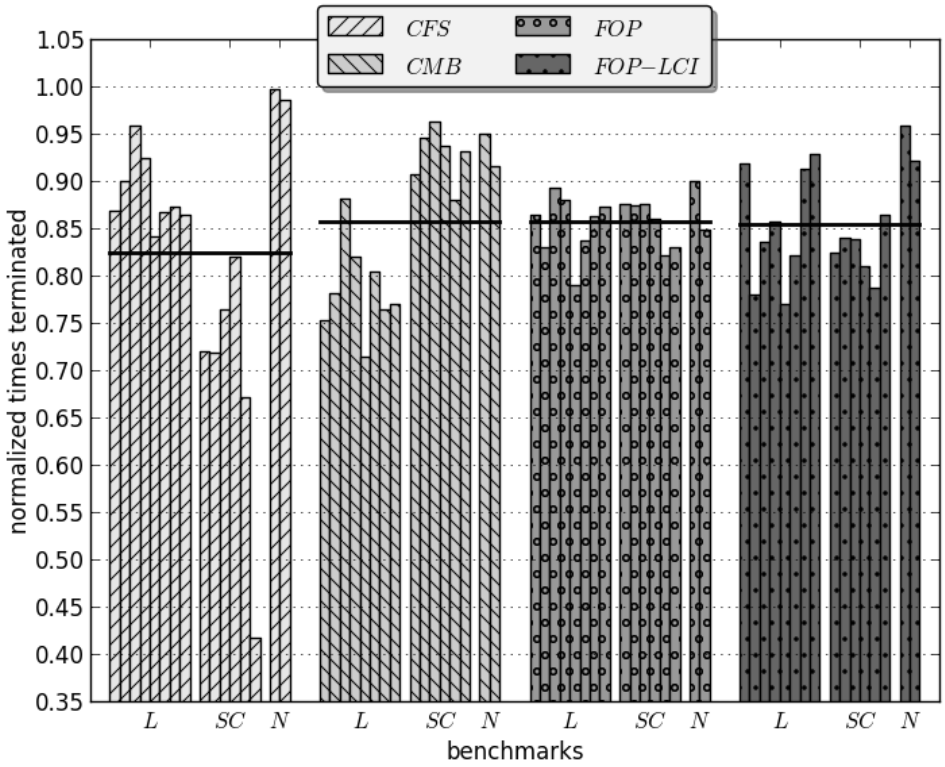


Figure A-5.2-4: Workload 4

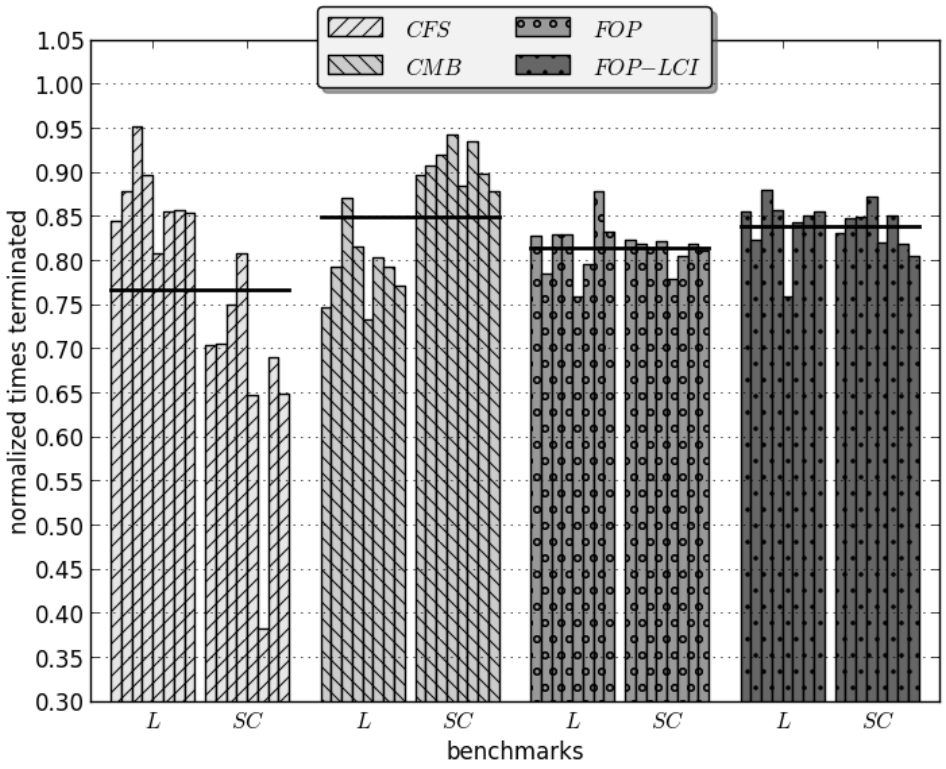


Figure A-5.2-5: Workload 5

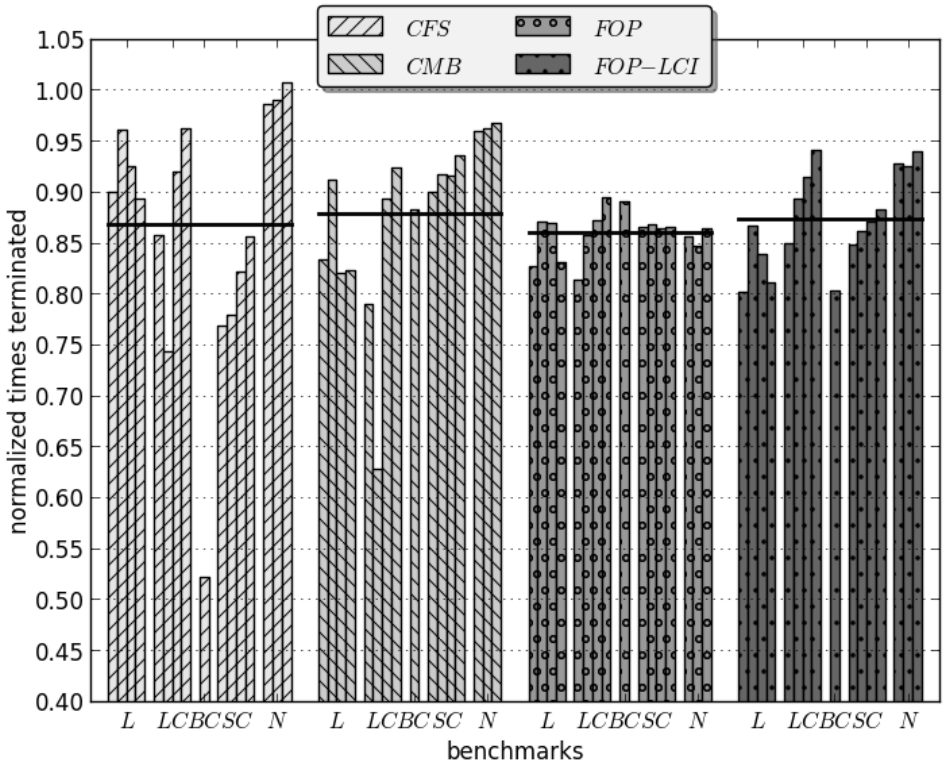


Figure A-5.2-6: Workload 6

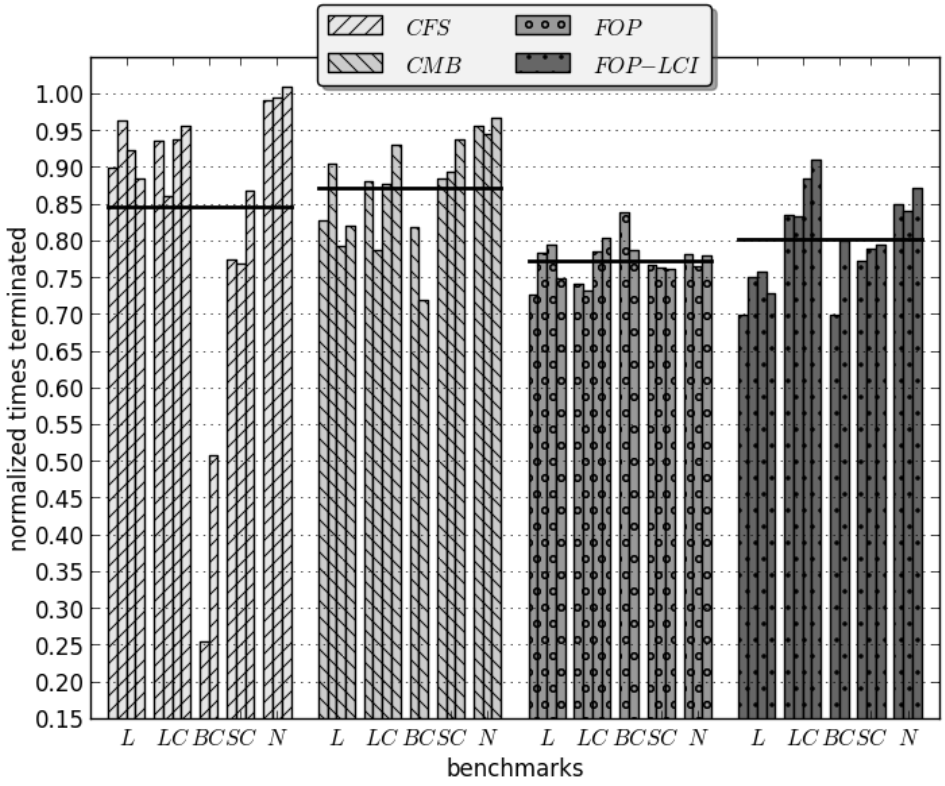


Figure A-5.2-7: Workload 7

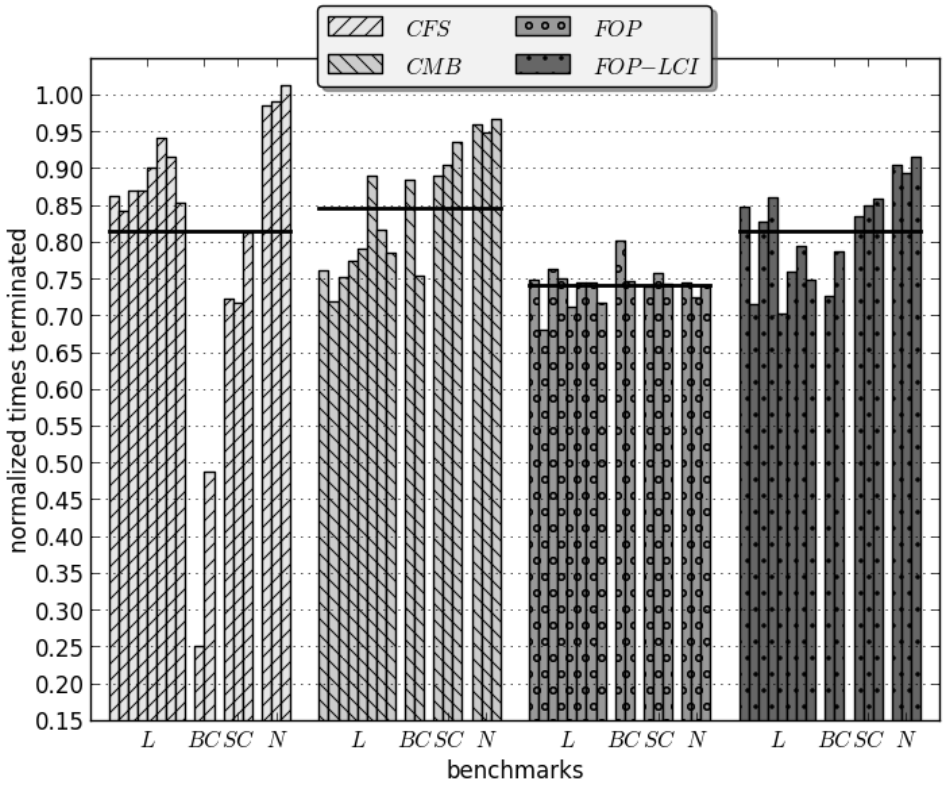


Figure A-5.2-8: Workload 8

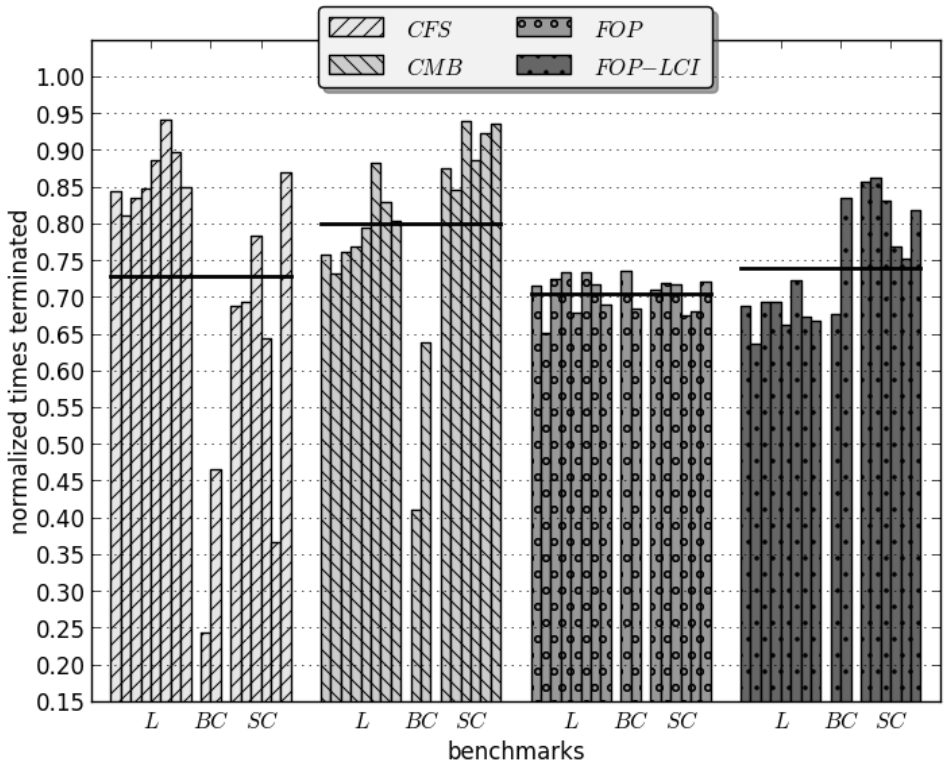


Figure A-5.2-9: Workload 9

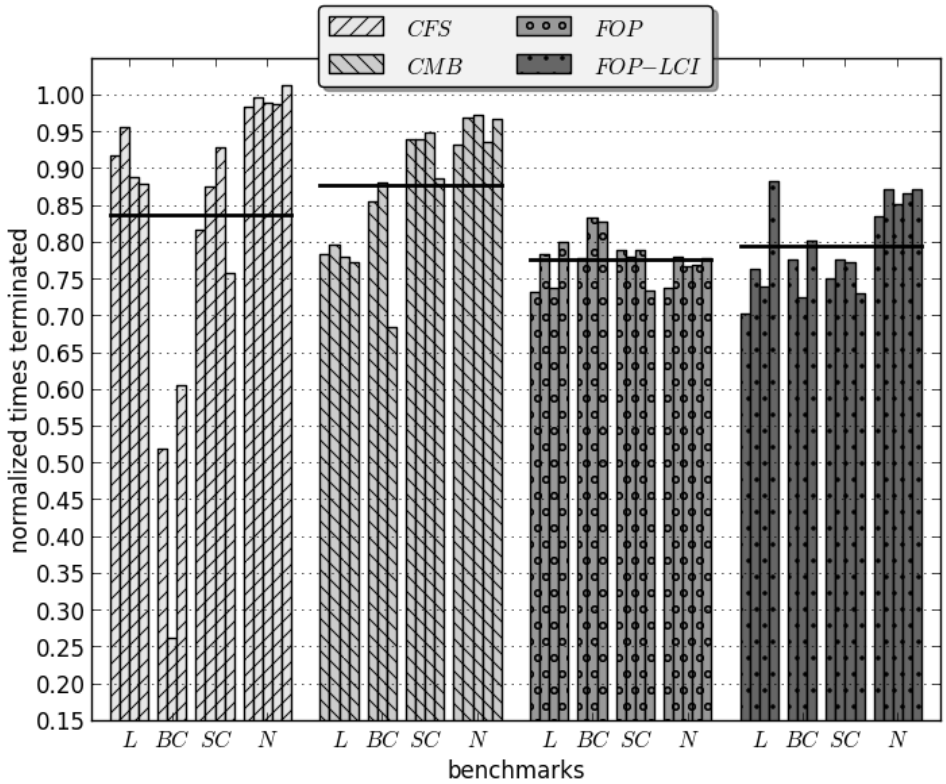


Figure A-5.2-10: Workload 10

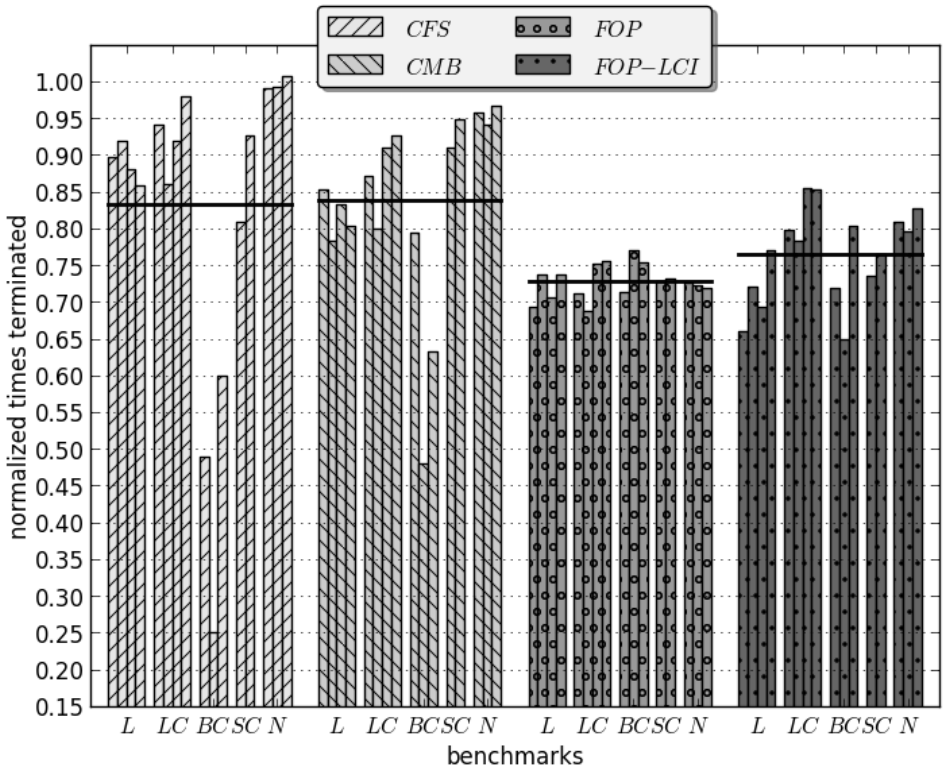


Figure A-5.2-11: Workload 11

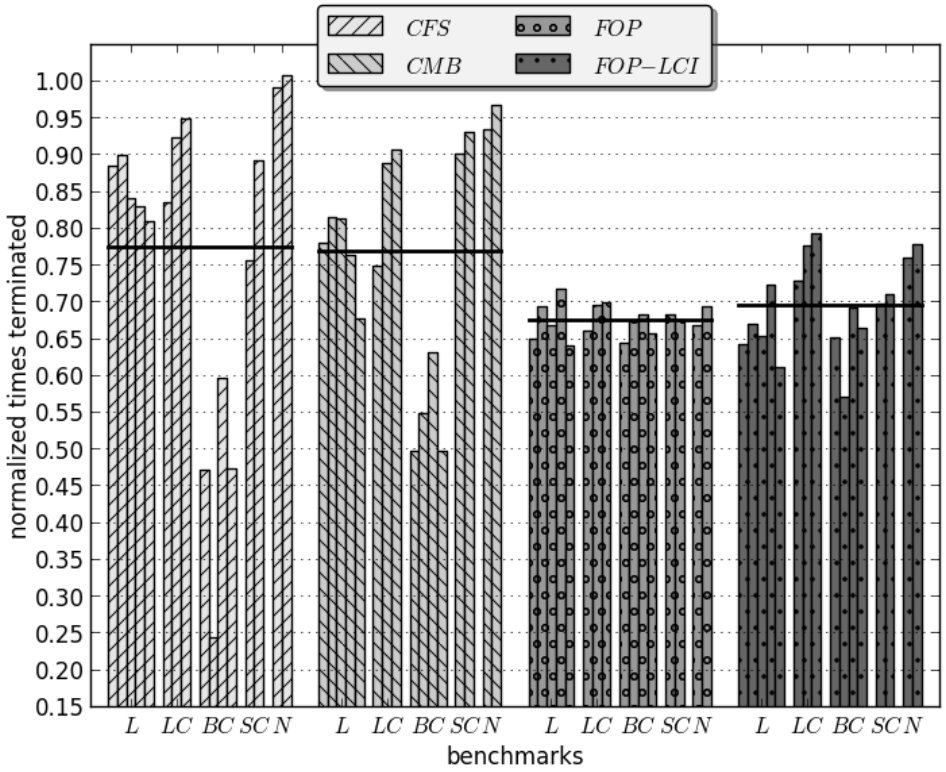


Figure A-5.2-12: Workload 12

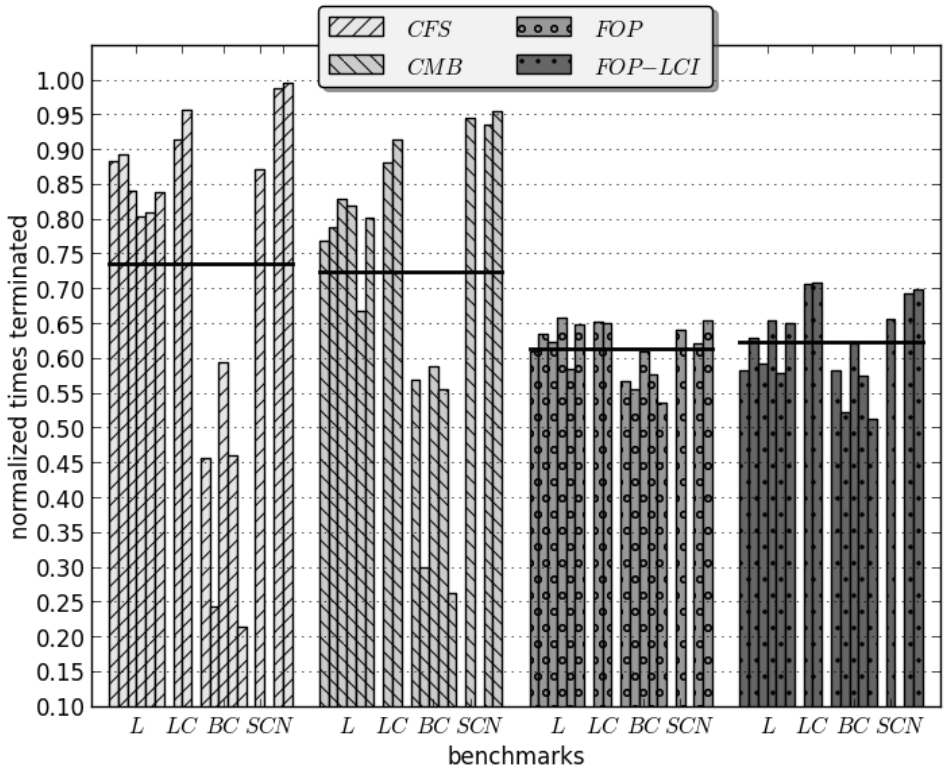


Figure A-5.2-13: Workload 13

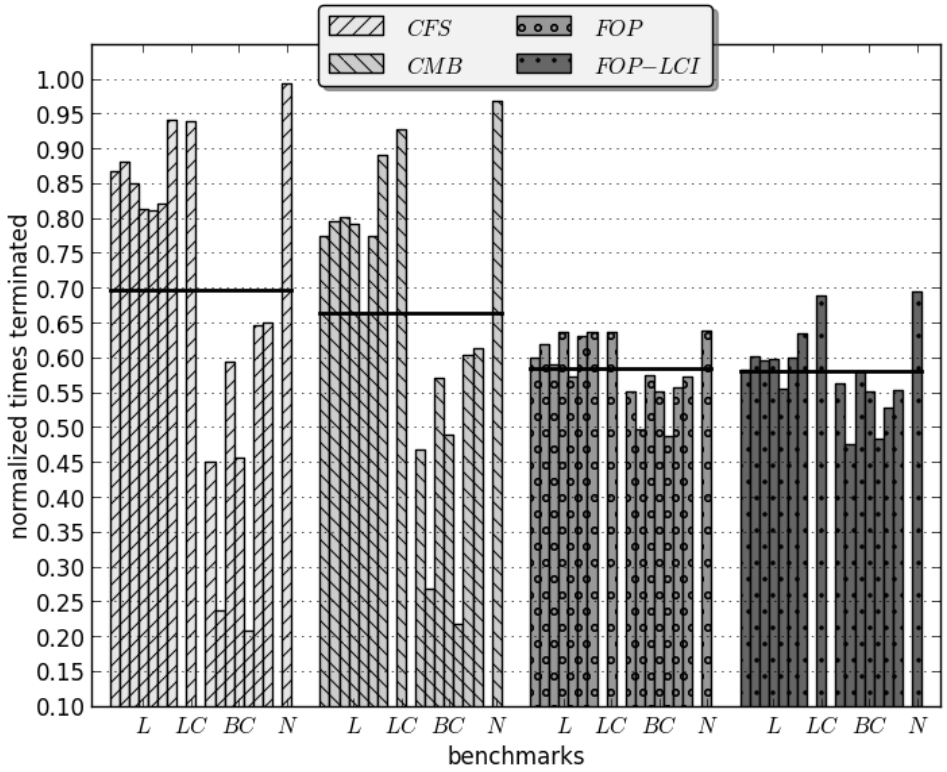


Figure A-5.2-14: Workload 14

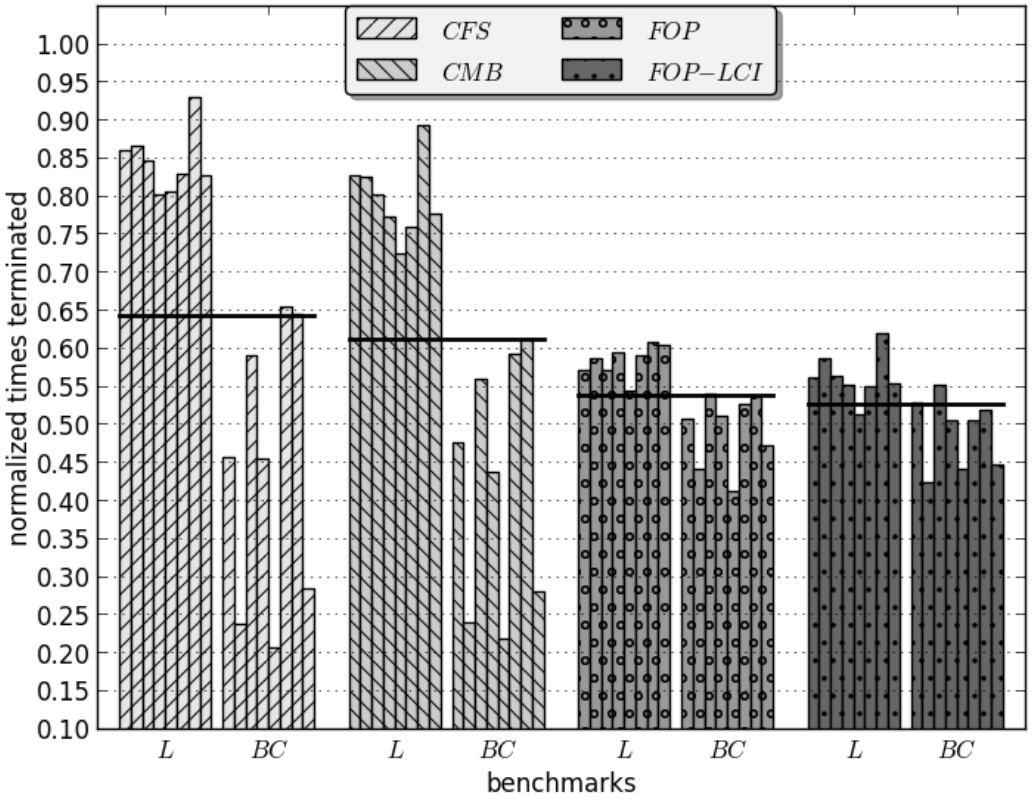


Figure A-5.2-15: Workload 15