ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Design and Evaluation of Clustered Processor Architectures

Ζέρβα Βασιλεία

Επιβλέπων : Διονύσιος Πνευματικάτος
Καθηγητής ΕΜΠ

Αθήνα, Ιούνιος 2025

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Design and Evaluation of Clustered Processor Architectures

Βασιλεία Ζέρβα

Επιβλέπων : Διονύσιος Πνευματικάτος
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 27 Ιουνίου 2025.

Διονύσιος Πνευματικάτος	Γεώργιος Γκούμας	Νεκτάριος Κοζύρης
(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Καθηγητής ΕΜΠ	Αναπληρωτής Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Ιούνιος 2025

(Υπογραφή)

.....

Ζέρβα Βασιλεία
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Οι αυξανόμενες υπολογιστικές απαιτήσεις των σύγχρονων εφαρμογών και η ανάγκη για την βελτιωμένη απόδοση έχουν οδηγήσει τους σχεδιαστές να επικεντρωθούν στην αύξηση του αριθμού των πυρήνων σε ένα μόνο chip. Τα συστήματα με πολλαπλούς πυρήνες είναι ιδιαίτερα αποτελεσματικά σε πολλές εφαρμογές, καθώς αυξάνουν τον παραλληλισμό σε επίπεδο νημάτων (Thread-Level Parallelism), βελτιώνοντας το συνολικό ρυθμό διεκπεραίωσης. Ωστόσο, η ενίσχυση της απόδοσης ενός μόνο νήματος παραμένει κρίσιμη, ειδικά για εφαρμογές με περιορισμένο παραλληλισμό. Σε τέτοιες περιπτώσεις, ο παραλληλισμός σε επίπεδο εντολών (Instruction-Level Parallelism) μπορεί να αποτελέσει σημαντικό εμπόδιο στην βελτίωση της απόδοσης. Για την αντιμετώπιση αυτού του περιορισμού, οι σχεδιαστές ακολούθησαν πιο ευρείες αρχιτεκτονικές με μεγαλύτερα παράθυρα εντολών και μεγαλύτερο πλάτος. Ωστόσο, αυτή η προσέγγιση συνοδεύεται από σημαντικούς συμβιβασμούς: απαιτεί πιο σύνθετη λογική ελέγχου, αυξάνει την κατανάλωση ισχύος και περιορίζει τη συχνότητα ρολογιού, λόγω της αυξημένης πολυπλοκότητας και του μεγέθους των δομικών μονάδων.

Μία εναλλακτική λύση που προτάθηκε για την αποφυγή αυτών των προβλημάτων είναι η τεχνική του "Clustering". Αυτή η τεχνική, περιλαμβάνει τη διαίρεση των πόρων σε μικρότερες, ανεξάρτητες μονάδες, όπου κάθε μονάδα διαχειρίζεται ένα υποσύνολο εντολών με δικούς της πόρους. Με αυτόν τον τρόπο, μειώνονται οι καθυστερήσεις στις συνδέσεις και διατηρείται υψηλή συχνότητα ρολογιού ακόμα και αν το σύστημα κλιμακώνεται.

Σε αυτή τη διπλωματική, αναλύουμε τους περιορισμούς του Clustering, εξετάζουμε υπάρχουσες τεχνικές κατανομής των εντολών στα clusters, και υλοποιούμε τέσσερις διαφορετικές μεθόδους κατανομής —Round-Robin, Dependency, Dependency-Load και Loadcut— οι οποίες, έχουν προσαρμοστεί ώστε να ανταποκρίνονται στους στόχους και στους περιορισμούς της αρχιτεκτονικής μας. Στη συνέχεια, προσομοιώνουμε και αξιολογούμε τις μεθόδους αυτές χρησιμοποιώντας τον προσομοιωτή gem5 και αναλύουμε την απόδοσή τους βάσει διαφόρων μετρικών.

Λέξεις Κλειδιά: Clustering, Instruction-Level Parallelism, Κατανομή Εντολών, συχνότητα ρολογιού, απόδοση, gem5

Abstract

Growing computational demands of modern workloads and the need for better application performance, has led designers to focus on increasing the number of cores into a single chip. This multi-core design is highly effective in many applications because it increases thread-level parallelism, improving the overall throughput. However, improving single-thread performance remains crucial, especially for applications with limited parallelism. Since in many scenarios, instruction-level parallelism becomes a bottleneck designers, tried to implement wider designs, with wider instructions windows and widths. This of course came with an important trade-off: wider windows and issue widths typically require more complex control logic, more power consumption, and significantly impact the clock frequency due to the increased complexity and size of the structures. Another approach was introduced in order to avoid the hazards mentioned before, called "clustering".

Clustering, means dividing resources into smaller, independent groups, each handling a subset of instructions with its own set of resources. Using this method could significantly reduce wire delays, helping to preserve high clock frequencies even as the overall system scales.

In this thesis, we analyze the limitations of clustering and review steering techniques that have been proposed in previous work. Then, we design and implement four instruction steering methods —Round-Robin, Dependency, Dependency-Load, and Loadcut— which, carefully adapted to the specific goals and architecture of our system. We then evaluate these methods through simulation using the gem5 simulator and analyze their performance.

Keywords— Clustering, instruction steering, Instruction-Level Parallelism, clock frequency, gem5, performance

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον επιβλέποντά μου, Καθηγητή ΕΜΠ Διονύσιο Πνευματικάτο που μου έδωσε την ευκαιρία να εκπονήσω την διπλωματική μου εργασία στο Εργαστήριο Υπολογιστικών Συστημάτων στο ΕΜΠ, καθώς και τον Κωνσταντίνο Παπαδόπουλο για την καθοδήγηση του. Τέλος θα ήθελα να ευχαριστήσω την οικογένειά μου και τους φίλους μου για την στήριξη τους.

Contents

List of Figures

List of Tables

Chapter 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

Για την επίτευξη υψηλότερων επιδόσεων, οι επεξεργαστές συχνά βασίζονται σε ευρύτερα παράθυρα εντολών και πλάτη εκκίνησης με στόχο τον παραλληλισμό σε επίπεδο εντολών (ILP). Ωστόσο, οι τεχνικές αυτές εισάγουν σημαντική πολυπλοκότητα, περιορίζοντας τελικά τη συχνότητα του ρολογιού. Για να μετριαστεί αυτό, έχει αναδειχθεί το clustering: χωρίζει τους πόρους του επεξεργαστή σε ανεξάρτητες ομάδες εκτέλεσης, κάθε μία από τις οποίες είναι υπεύθυνη για τον χειρισμό ενός υποσυνόλου εντολών. Αυτός ο διαχωρισμός επιτρέπει υψηλότερες συχνότητες ρολογιού λόγω μικρότερων μονοπατιών δεδομένων και απλούστερης λογικής ελέγχου. Ωστόσο, το clustering εισάγει επίσης προκλήσεις, ιδίως την επιβάρυνση λόγω της επικοινωνίας μεταξύ διαφορετικών clusters, η οποία είναι πιο χρονοβόρα και μπορεί να υποβαθμίσει σημαντικά τις επιδόσεις.

Στην παρούσα εργασία, διερευνούμε τις επιδόσεις που μπορεί να επιτύχουν διαφορετικές μεθόδοι κατανομής των εντολών στα clusters (σε εντολές-ανά-κύκλο). Αρχικά, αξιολογούμε απλούς μηχανισμούς -όπως το round-robin- που δεν λαμβάνουν υπόψη τις εξαρτήσεις των εντολών. Στη συνέχεια, διερευνούμε τεχνικές που βασίζονται στις εξαρτήσεις εντολών, που είναι ζωτικής σημασίας για την ελαχιστοποίηση της επικοινωνίας μεταξύ διαφορετικών clusters. Δεδομένου ότι η μη ισορροπημένη κατανομή εντολών μπορεί να δημιουργήσει συμφόρηση, υλοποιούμε μεθόδους που λαμβάνουν υπόψη το φορτίο των clusters την δεδομένη στιγμή.

Η ανάλυσή μας επεκτείνεται σε διάφορα πλάτη πυρήνων, συμπεριλαμβανομένων πιο μικρών (4-way και 8-way) και μεγάλων (16-way) πυρήνων, εμπνευσμένων από εμπορικά σχέδια. Συγκρίνουμε τις επιδόσεις όλων των μεθόδων κατανομής των εντολών με βασικά μοντέλα σχεδιασμού. Αυτή η αξιολόγηση παρέχει πληροφορίες σχετικά με τον τρόπο με τον οποίο το clustering αλληλεπιδρά με το πλάτος των πυρήνων, τις μεθόδους κατανομής και τα χαρακτηριστικά των εφαρμογών που χρησιμοποιούμε. Ειδικότερα, η μελέτη μας για μεγαλύτερα πλάτη αναδεικνύει ότι το clustering μπορεί ευνοήσει τη συχνότητα σε μελλοντικούς επεξεργαστές αλλά ταυτόχρονα να πετύχει και υψηλές επιδόσεις.

1.2 Out-of-order επεξεργαστές

Η εκτέλεση εκτός σειράς (out-of-order) είναι βασικό χαρακτηριστικό των σύγχρονων υπολογιστών υψηλής απόδοσης. Σε αντίθεση με τις αρχιτεκτονικές με εκτέλεση σε σειρά, που οι εντολές εκτελούνται με βάση την σειρά του προγράμματος, στην περίπτωση των out-of-order, αυτό γίνεται δυναμικά με βάση την διαθεσιμότητα των τελεστών. Αυτό επιτρέπει σε ανεξάρτητες εντολές να εκτελούνται χωρίς να περιμένουν προηγούμενες εντολές να ολοκληρωθούν. Αυτό βοηθάει στο παραλληλισμό σε επίπεδο εντολών (ILP).

Οι σύγχρονοι επεξεργαστές που υποστηρίζουν εκτέλεση εκτός σειράς βασίζονται σε πολύπλοκες υλοποιήσεις για να διασφαλίσουν την ορθότητα και την αποδοτικότητα. Αυτές περιλαμβάνουν την μετονομασία καταχωρητών (register renaming) για την εξάλειψη ψευδο-εξαρτήσεων, τους σταθμούς κράτησης (reservation stations) και τις ουρές εκκίνησης (issue queues) για την αποθήκευση έτοιμων εντολών, την προσωρινή μνήμη αναδιαταξης (ROB) για τη διατήρηση της σωστής σειράς προγράμματος κατά την φάση υποβολής, καθώς και πολλαπλές λειτουργικές μονάδες για παράλληλη εκτέλεση εντολών. Όλα αυτά συμβάλλουν ώστε ο επεξεργαστής να εκτελεί τις εντολές πιο γρήγορα και να διαχειρίζεται σωστά τις εξαρτήσεις αποφεύγοντας κινδύνους.

Ένα βασικό πλεονέκτημα της εκτέλεσης εκτός σειράς είναι η ικανότητά της να μετριάζει τις παύσεις στην διοχέτευση. Οι εντολές που περιμένουν τελεστές μπορούν να παραμένουν στους σταθμούς κράτησης χωρίς να εμποδίζουν την διοχέτευση, ενώ ανεξάρτητες εντολές μπορούν να συνεχίσουν την εκτέλεση. Αυτό μειώνει τους χρόνους αδράνειας και αυξάνει τη ροή. Επιπλέον, οι περισσότεροι σύγχρονοι επεξεργαστές αυτού του τύπου είναι υπερβαθμωτοί (superscalar), δηλαδή μπορούν να εκδίδουν πολλαπλές εντολές ανά κύκλο. Ο συνδυασμός υπερβαθμωτού σχεδιασμού και εκτέλεσης εκτός σειράς είναι κρίσιμος για την επίτευξη υψηλών επιδόσεων σε εφαρμογές ενός νήματος.

1.2.1 Απόδοση

Προκειμένου να βελτιωθούν οι επιδόσεις στις σύγχρονες εφαρμογές με σημαντικές υπολογιστικές ανάγκες, οι σχεδιαστές υλικού επιχείρησαν να ενσωματώσουν σε ένα chip πολλαπλούς πυρήνες, ειδικά για εφαρμογές ικανές να πετύχουν παραλληλισμό σε επίπεδο νημάτων (TLP). Ωστόσο, κρίνεται επιτακτικής σημασίας, η βελτίωση της απόδοσης κάθε ενός νήματος. Προκειμένου να επιτευχθεί αυτό αλλά και μεγαλύτερος παραλληλισμός σε επίπεδο εντολών, οι σχεδιαστές επιχείρησαν να αυξήσουν τα παράθυρα εντολών και το πλάτος εκκίνησης (issue width), το οποίο όπως είναι αναμενόμενο έχει αρνητικό αντίκτυπο στην πολυπλοκότητα του σχεδιασμού, την κατανάλωση ενέργειας και την συχνότητα του ρολογιού.

Οι Palacharla et. al. ανέπτυξαν τεχνικές για την ανάλυση της πολυπλοκότη-

τας και των καθυστερήσεων κάθε σταδίου της διοχέτευσης και διαπίστωσαν ότι το πλάτος εκκίνησης και το συνολικό μέγεθος του παραθύρου εντολών παίζουν πολύ σημαντικό ρόλο στην πολυπλοκότητα και οι καθυστερήσεις των καλωδίων στην συνολική καθυστέρηση του συστήματος. Μια τεχνική προκειμένου να αποφευχθούν τα αρνητικά της μεγέθυνσης του σχεδιασμού, είναι το "clustering", όπως θα αναλύσουμε στην συνέχεια.

1.2.2 Ανάλυση διοχέτευσης υπερβαθμωτού επεξεργαστή

Για να κατανοήσουμε καλύτερα τους περιορισμούς της κλιμάκωσης ενός επεξεργαστή, θα περιγράψουμε τα στάδια που προηγούνται της εκτέλεσης μιας εντολής. Οι εντολές εκδίδονται παράλληλα, και όταν συναντάται διακλάδωση, εφαρμόζεται κάποιος μηχανισμός πρόβλεψης: ο μετρητής προγράμματος ενημερώνεται βάσει της πρόβλεψης. Οι εντολές στη συνέχεια περνούν στο στάδιο της αποκωδικοποίησης και τη μετονομασία καταχωρητών και έπειτα εισέρχονται στη μνήμη αναδιάταξης και στην ουρά εκκίνησης.

Στο στάδιο εκκίνησης, οι εντολές περιμένουν μέχρι να γίνουν διαθέσιμοι οι τελεστές τους, είτε από το αρχείο καταχωρητών είτε μέσω προώθησης από εκτελεσμένες εντολές. Η λογική επιλογής διαλέγει πολλές έτοιμες εντολές κάθε κύκλο (ανάλογα με το πλάτος εκκίνησης του επεξεργαστή) και τις στέλνει στις αντίστοιχες λειτουργικές μονάδες. Αυτός ο μηχανισμός επιτρέπει στους υπερβαθμωτούς επεξεργαστές να εκμεταλλεύονται το ILP και να εκτελούν πολλαπλές εντολές ανά κύκλο.

1.2.3 Πολυπλοκότητα σχεδίασης

Η πολυπλοκότητα ενός σχεδίου μπορεί να μετρηθεί με διάφορους τρόπους, όπως τον αριθμό τρανζίστορ, την επιφάνεια του chip, και την κατανάλωση ισχύος. Στην παρούσα εργασία θα χρησιμοποιήσουμε την προσέγγιση των Palacharla et. al., οι οποίοι ορίζουν την πολυπλοκότητα ως την καθυστέρηση στο κρίσιμο μονοπάτι μιας λογικής μονάδας. Συγκεκριμένα, διαπίστωσαν ότι το window wakeup, που ξυπνά εντολές που περιμένουν τελεστές, και τη λογική επιλογής (selection logic), που επιλέγει εντολές προς εκτέλεση, ως τα πιο κρίσιμα σημεία από άποψη καθυστέρησης.

Ιδιαίτερα, η λογική επιλογής γίνεται ολοένα πιο περίπλοκη όσο μεγαλώνει η συλλογή των εντολών. Έτσι, με βάση τα ευρήματα αυτών των μελετών και την εξέταση των σταδίων της διοχέτευσης, γίνεται σαφές πως οι παραδοσιακοί μονολιθικοί σχεδιασμοί παρουσιάζουν σημαντικούς περιορισμούς στην κλιμάκωση. Αυτό ενισχύει την ανάγκη για εναλλακτικές, όπως το clusterin, οι οποίες να μοιράζουν την πολυπλοκότητα και να μετριάσουν τα bottlenecks διατηρώντας παράλληλα υψηλό ILP.

1.2.4 Το Clustering ως λύση

Ως εναλλακτική λύση στη διεύρυνση των παραθύρων έκδοσης και στην απλή κλιμάκωση του σχεδιασμού, προτείνουμε τη μέθοδο του Clustering, ώστε να προσεγγίσουμε τα πλεονεκτήματα των ευρύτερων παραθύρων έκδοσης και αριθμών εντολών, διατηρώντας ταυτόχρονα τις συχνότητες που επιτυγχάνονται με μικρότερες δομές.

Το Clustering ως τεχνική, βασίζεται στην διαίρεση του συστήματος σε μικρότερα ανεξάρτητα clusters που κάθε ενα διαχειρίζεται ενα υποσύνολο των εντολών. Συγκεκριμένα, κάθε ένα έχει δική του ουρά εκκίνησης και λειτουργικές μονάδες. Με αυτή την τεχνική, επιτυγχάνονται πλάτη από πιο ευρείες αρχιτεκτονικές, αλλά χωρίς την χρήση μεγάλων δομών –με αντίστοιχα αυξημένες καθυστερήσεις καλωδιώσεων και πολυπλοκότητα– και την μειωμένη συχνότητα ρολογιού που αυτό συνεπάγεται.

Επιπλέον, το Clustering καθιστά δυνατή την παραλληλία σε επίπεδο εντολών (ILP), χωρίς την ανάγκη για μεγαλύτερες, μονολιθικές δομές. Η λογική χρονοπρογραμματισμού επίσης γίνεται απλούστερη και ταχύτερη, καθώς κάθε cluster διαχειρίζεται μόνο ένα μικρότερο σύνολο εντολών και πόρων

1.2.5 Περιορισμοί του Clustering

Σε μια αρχιτεκτονική που υποστηρίζει εκτέλεση εκτός σειράς, όταν μια εντολή παράγει ένα αποτέλεσμα, το προωθεί σε εξαρτώμενες εντολές με την χρήση μηχανισμών προώθησης, ώστε να προχωρήσουν σε εκτέλεση χωρίς να περιμένουν να γραφτεί αυτό στον φάκελο καταχωρητών πρώτα αλλά να εκτελεστούν όσο το δυνατόν πιο γρήγορα.

Αυτές οι προωθήσεις, όταν πραγματοποιούνται μεταξύ διαφορετικών clusters, παίρνουν περισσότερο χρόνο λόγω μεγαλύτερων καλωδιώσεων -θα τις αποκαλούμε inter-cluster προωθήσεις. Για την επίτευξη ουσιαστικών αποτελεσμάτων, οι εντολές πρέπει να κατανέμονται στα clusters με τρόπο που να ελαχιστοποιούνται όσο το δυνατόν περισσότερο οι καθυστερήσεις που οφείλονται στο Clustering. Αυτές τις μεθόδους κατανομής θα τις αναφέρουμε ως μεθόδους οδήγησης (ή steering methods).

Έπειτα από ενδελεχή μελέτη των υπαρχουσών προσεγγίσεων, ξεκινήσαμε με την υλοποίηση των πιο απλών μεθόδων κατανομής. Στη συνέχεια, προχωρήσαμε στην ανάπτυξη πιο εξελιγμένων μεθόδων που λαμβάνουν υπόψη τις εξαρτήσεις μεταξύ εντολών κατά την εκτέλεση του προγράμματος. Οι μέθοδοι αυτές στοχεύουν στη συγκέντρωση εξαρτώμενων εντολών στο ίδιο cluster όποτε αυτό είναι δυνατό, ελαχιστοποιώντας έτσι την επικοινωνία μεταξύ διαφορετικών clusters και αποφεύγοντας περιττές καθυστερήσεις.

Ενώ ο μέγιστος στόχος είναι η επίδοση, λαμβάνεται επίσης υπόψη το κόστος και η πολυπλοκότητα των μεθόδων κατανομής. Αντί να βασιστούμε σε ιδιαίτερα

περίπλοκες δομές υλικού ή σε υπολογιστικά δαπανηρές δυναμικές μεθόδους εκτέλεσης, επικεντρωθήκαμε σε προσεγγίσεις που επιτυγχάνουν αποτελεσματική κατανομή εντολών με χαμηλό υλικοτεχνικό κόστος και απλή λογική.

1.3 Προηγούμενη Μελέτη

Η τεχνική του Clustering προτάθηκε από τους Palacharla et. al. ως λύση για τη μείωση του κύκλου ρολογιού στους υπερβαθμωτούς επεξεργαστές. Μετά από ανάλυση των καθυστερήσεων στην διοχέτευση, διαπίστωσαν ότι η λογική έκδοσης εντολών και η λογική προώθησης αποτελούν τους κυρίαρχους παράγοντες καθυστέρησης. Αντικατέστησαν το κλασικό παράθυρο εντολών με παράλληλες δομές FIFO και δρομολόγησαν τις εντολές βάσει των εξαρτήσεών τους. Το μοντέλο τους χρησιμοποιεί το clustering με τέσσερίς FIFO ανά cluster και διαχωρισμό των λειτουργικών μονάδων. Οι προσομοιώσεις έδειξαν μικρή μείωση του IPC αλλά σημαντική βελτίωση της συχνότητας λειτουργίας.

Οι Baniasadi και Moshovos σύγκριναν διάφορες πολιτικές οδήγησης εντολών πέραν των εξαρτήσεων, με στόχο τη μείωση των καθυστερήσεων λόγω περιορισμών εύρους ζώνης και επικοινωνίας μεταξύ clusters. Υλοποίησαν τέσσερα clusters με ξεχωριστό προγραμματιστή και λειτουργικές μονάδες. Εξέτασαν τόσο προσαρμοστικές όσο και μη προσαρμοστικές μεθόδους –με βάση το αν αλλάζει η λογική τους κατά το run-time–, με την απλή μέθοδο MOD-3 να έχει την καλύτερη απόδοση από τις μη προσαρμοστικές. Επίσης, εισήγαγαν τη μέθοδο "slice" που διασφαλίζει ότι όλες οι γονικές εντολές ανατίθενται στο ίδιο cluster.

Οι Tune, Liang, Tullsen και Calder μελέτησαν ένα μοντέλο με δύο και τέσσερα clusters. Εισήγαγαν έναν προβλεπτή κρίσιμου μονοπατιού που λαμβάνει υπόψη πόσο κρίσιμη είναι μια εντολή για την εκτέλεση. Στην προσέγγισή τους, όταν μια εντολή έχει δύο πηγαίους τελεστές από διαφορετικά clusters, χρησιμοποιείται ο προβλεπτής για να αποφασιστεί η ανάθεση.

Οι Salverda et. al. ανέλυσαν το κρίσιμο μονοπάτι για να εντοπίσουν τις αιτίες απώλειας απόδοσης και πρόσθεσαν νέα κριτήρια στις υπάρχουσες μεθόδους οδήγησης. Χρησιμοποίησαν μετρητές κρισιμότητας για τον προσδιορισμό κρίσιμων γονικών εντολών, λαμβάνοντας επίσης υπόψη την εξισορρόπηση του φορτίου των clusters.

Οι Michaud et. al. εισήγαγαν μια αρχιτεκτονική ευρείας εκκίνησης με δύο clusters, με στόχο την αύξηση του IPC διατηρώντας σταθερή τη συχνότητα. Αντίθετα με προηγούμενες εργασίες που στόχευαν στη βελτίωση της απόδοσης μέσω αύξησης της συχνότητας, επικεντρώθηκαν στο διπλασιασμό του εύρους εκκίνησης για υψηλότερο IPC.

Οι Ranganathan και Franklin ανέπτυξαν το μοντέλο PEWs (Parallel Execution Windows) με σκοπό να αντιμετωπίσουν τους περιορισμούς της κλιμάκωσης των συμβατικών επεξεργαστών. Το μοντέλο αυτο αντί να χρησιμοποιεί ένα κεν-

τρικό παράθυρο εντολών, κατανέμει τις εντολές σε πολλαπλά παράλληλα παράθυρα εκτέλεσης βάσει εξαρτήσεων μεταξύ καταχωρητών.

Οι Aggarwal και Franklin διερεύνησαν μεθόδους δυναμικής αναπαραγωγής εντολών -βασίζονται στη δημιουργία αντιγράφων εντολών σε συγκεκριμένα PEWs– για τη μείωση της επικοινωνίας μεταξύ τους. Οι τεχνικές τους οδήγησαν σε αύξηση του IPC κατά 5-10% σε σύγκριση με έναν απλό αλγόριθμο εξισορρόπησης του φορτίου των PEWs αλλά και σε αύξηση των καθυστερήσεων λόγω επικοινωνίας μεταξύ διαφορετικών παραθύρων.

Οι Canal et. al. επικεντρώθηκαν στα οφέλη του clustering για επιτάχυνση, διαχωρίζοντας τον επεξεργαστή σε ένα cluster INT και ένα FP. Εφάρμοσαν δυναμικές τεχνικές κατά την εκτέλεση, που παρά την χειροτέρευση του IPC, είχαν ως αποτέλεσμα συνολική επιτάχυνση λόγω καλύτερης αξιοποίησης των clusters και μειωμένου ανταγωνισμού για πόρους. Οι δυναμικές μέθοδοι ωστόσο απαιτούν προηγμένο υλικό και προγραμματισμό, που είναι πέρα από το πεδίο μελέτης μας.

Η παρούσα εργασία ξεκινά με μικρά εύρη εκκίνησης (2-way και 4-way clusters) και απλές τεχνικές οδήγησης των εντολών σε αυτά, που δεν απαιτούν πολύπλοκο υλικό, όπως αυτό που απαιτεί η ανάλυση κρίσιμου μονοπατιού. Για την παρακολούθηση εξαρτήσεων, χρησιμοποιούνται απλοί πίνακες που παρακολουθούν κάποιες πληροφορίες των καταχωρητών. Στη συνέχεια, η μεθοδολογία επεκτείνεται σε ευρύτερους επεξεργαστές, συγκεκριμένα ενός υπερβαθμωτού επεξεργαστή με πλάτος εκκίνησης 16 και αντίστοιχα ενισχυμένους πόρους, παρέχοντας χρήσιμες πληροφορίες για μελλοντικά συστήματα υψηλών επιδόσεων.

1.4 Μεθοδολογία

1.4.1 Προσομοιωτής

Χρησιμοποιήσαμε τον προσομοιωτή gem5 για να υλοποιήσουμε τους σχεδιασμούς και τις μεθόδους οδήγησης, μεταγλωτισμένο σε αρχιτεκτονική RISC-V.

Η RISC-V είναι μία σύγχρονη, ανοιχτού τύπου αρχιτεκτονική συνόλου εντολών (Instruction Set Architecture), η οποία έχει κερδίσει αναγνώριση τόσο στον ακαδημαϊκό όσο και στον βιομηχανικό χώρο. Σχεδιάστηκε ώστε να είναι απλή, αποδοτική και εύκολα επεκτάσιμη, χωρίς να απαιτεί τη διατήρηση παρωχημένων ή περίπλοκων χαρακτηριστικών από παλαιότερες αρχιτεκτονικές. Αυτή η ευελιξία, σε συνδυασμό με την υποστήριξη του RISC-V από το gem5, το κατέστησαν ιδανική επιλογή για την υλοποίηση και αξιολόγηση των προτεινόμενων μικροαρχιτεκτονικών μας μοντέλων.

Χρησιμοποιήσαμε το μοντέλο "O3CPU" που αναπαριστά έναν υπερβαθμωτό επεξεργαστή που υποστηρίζει την εκτέλεση εκτός σειράς, το οποίο είναι παραμετροποιήσιμο όσον αφορά την πρόβλεψη διακλάδωσης, τα πλάτη των σταδίων της διοχέτευσης, τις ουρές εντολών, την προσωρινή μνήμη αναδιάταξης και τις load/s-

tore ουρές (LSQ). Ενώ η τυπική υλοποίηση του RISC-V που φαίνεται στο Σχήμα 6.1 χρησιμοποιεί έναν απλούστερη διοχέτευση τεσσάρων σταδίων, η εργασία μας βασίζεται στο πιο σύνθετο μοντέλο επτά σταδίων του gem5 για την υλοποίηση της αρχιτεκτονικής μας, επειδή προσφέρει το απαραίτητο πλαίσιο που απαιτεί η έρευνά μας.

Στη συνέχεια, αξιολογήσαμε τις μεθόδους μας από πλευράς απόδοσης, χρησιμοποιώντας τα benchmarks SPEC2017, και πιο συγκεκριμένα τις SPECspeed2017 Integer και Floating point σουίτες, τις οποίες μεταγλωττίσαμε για την αρχιτεκτονική RISC-V χρησιμοποιώντας εργαλεία cross-compiling. Τα benchmarks και οι τομείς εφαρμογής τους παρουσιάζονται ενδεικτικά στον πίνακα 7.1.

1.4.2 Μέθοδοι οδήγησης

Διαχωρίζουμε τις διαθέσιμες λειτουργικές μονάδες του αντίστοιχου βασικού μοντέλου αναφοράς στα clusters, καθώς και την ουρά εντολών –που κρατάει τις εντολές μέχρι οι τελεστές της να είναι διαθέσιμοι– ώστε κάθε cluster να έχει δικό του. Όταν η αντίστοιχη ουρά του cluster που θέλουμε να στείλουμε μια εντολή είναι γεμάτη, τότε την τοποθετούμε σε κάποια άλλη διαθέσιμη και στη συνέχεια στο αντίστοιχο cluster. Οι αλλαγές που πραγματοποιήσαμε στον πηγαίο κώδικα του προσομοιωτή παρουσιάζονται συνοπτικά στο κεφάλαιο 6.3. Όλες οι μέθοδοι που αναπτύξαμε μπορούν εύκολα να υλοποιηθούν και σε επίπεδο υλικού και συνοψίζονται ως εξής:

- Round Robin: Οι εντολές στέλνονται στα clusters κυκλικά, δηλαδή με τρόπο round-robin.
- Dependence-based: Αυτή η μέθοδος κάνει χρήση των εξαρτήσεων μεταξύ των εντολών, ώστε να τοποθετούνται αυτές μαζί και να περιορίζονται οι inter-clusters προωθήσεις. Προκειμένου να υλοποιηθεί αυτό χρησιμοποιήθηκαν πληροφορίες σχετικές με τους καταχωρητές, και συγκεκριμένα αναγνωριστικά που υποδεικνύουν σε ποιό cluster έχουν παραχθεί οι τελεστές μιας εντολής ώστε αυτή να σταλεί εκεί. Αν δεν υπάρχουν εξαρτήσεις μεταξύ της εκάστοτε εντολής και κάποιας άλλης ή οι εξαρτήσεις δεν είναι έγκυρες λόγω υποβολής της εντολής-γονέα, τότε αυτή στέλνεται σε τυχαίο cluster.
- Dependence-load-based: Ακολουθείται η παραπάνω μεθοδολογία με την διαφορά ότι αν δεν υπάρχουν εξαρτήσεις μεταξύ της τρέχουσας εντολής και κάποιας άλλης, τότε η επιλογή του cluster γίνεται με βάση το σχετικό φορτίο των clusters.
- Loadcut: Αυτή η μέθοδος, αξιοποιεί την σειρά εξαρτήσεων που δημιουργείται από μια εντολή load, επομένως γίνεται αλλαγή του cluster κάθε φορά που συναντάται αυτού του τύπου εντολή (εκτος αν έχω διαδοχικά load, εκεί δεν αλλάζω στο κάθε ένα).

1.4.3 Βασικό Μοντέλο Αναφοράς

Το βασικό μοντέλο αναφοράς που χρησιμοποιούμε αφορά έναν υπερβαθμωτό επεξεργαστή που υποστηρίζει εκτέλεση εκτός σειράς, ικανό να εκδίδει πολλαπλές εντολές ανά κύκλο. Συγκεκριμένα, προσομοιώνονται δύο πυρήνες με διαφορετικό πλάτος εκκίνησης: ο ένας εκδίδει έως 4 εντολές ανά κύκλο και ο άλλος έως 8 –θα τους αποκαλούμε 4-way και 8-way αντίστοιχα. Για να προσεγγίσουμε καλύτερα αρχιτεκτονικές σύγχρονων επεξεργαστών, αντλούμε έμπνευση από γνωστά εμπορικά σχέδια. Ο 4-way επεξεργαστής βασίστηκε στον ARM Cortex-A76, ενώ ο 8-way επεξεργαστής προσομοιώνει στοιχεία της μικροαρχιτεκτονικής Intel Golden Cove. Οι ενισχυμένοι πόροι στον 8-way πυρήνα διασφαλίζουν ότι το επιπλέον εύρος ζώνης στο στάδιο εκκίνησης αξιοποιείται πλήρως, χωρίς να δημιουργούνται σημεία συμφόρησης σε άλλα τμήματα του pipeline. Τέλος, στον 8-way σχεδιασμό δίνεται διπλάσιος αριθμός λειτουργικών μονάδων (ή μονάδων εκτέλεσης) σε σχέση με τον 4-way, ώστε να μπορεί να υποστηριχθεί ο αυξημένος ρυθμός εντολών λόγω του μεγαλύτερου πλάτους εκκίνησης. Οι αναλυτικές μικροαρχιτεκτονικές παράμετροι που χρησιμοποιήθηκαν παρατίθενται στον πίνακα 6.1.

1.4.4 Clustered Σχεδιασμοί

Αρχικά υλοποιήσαμε δυο clustered σχεδιασμούς, 2x2-way και 2x4-way, που σημαίνει ότι διαθέτουμε δύο clusters με πλάτη εκκίνησης 2 και 4 αντίστοιχα. Ο 2x2-way σχεδιασμός προσομοιώνει το πλάτος ενός 4-way, ενώ ο 2x4-way ενός 8-way. Κάθε cluster διαθέτει ουρές εντολών με μισή χωρητικότητα σε σχέση με το αντίστοιχο μοντέλο αναφοράς, καθώς και τις μισές λειτουργικές μονάδες.

Διαχωρίζουμε τις προωθήσεις δεδομένων σε δύο κατηγορίες: τοπικές, που δεδομένα μεταφέρονται ανάμεσα σε λειτουργικές μονάδες εντός του ίδιου cluster και διαρκούν έναν κύκλο του ρολογιού, και inter-cluster, που δεδομένα μεταφέρονται ανάμεσα σε λειτουργικές μονάδες διαφορετικών cluster και διαρκούν δύο κύκλους. Η διοχέτευση των clustered σχεδιασμών μας περιγράφεται στην εικόνα 6.2 και η λογική με την οποία οι εντολές κατανέμονται στις ουρές εντολών και τα clusters στην εικόνα 6.3.

1.5 Πειράματα

Μεταγλωττίσαμε τα SPEC2017 benchmarks, και συγκεκριμένα τις SPECspeed2017 Integer και Floating Point σουίτες για την αρχιτεκτονική RISC-V. Στη συνέχεια τα τρέξαμε για 1 δισεκατομμύριο εντολές και 5 εκατομμύρια εντολές για warm-up ώστε τα προγράμματα να έχουν μπει στην φάση εκτέλεσης για να εκτιμήσουμε πραγματικά την απόδοση.

1.5.1 Απόδοση των μεθόδων οδήγησης για αρχιτεκτονικές με δύο clusters

Συγκρίναμε την απόδοση των τεσσάρων μεθόδων οδήγησης με κριτήριο τις εντολές-ανα-κύκλο (IPC). Για όλα τα benchmark, παρατηρούμε μια μικρή χειροτέρευση στο IPC για όλες τις μεθόδους σε σχέση με το βασικό μοντέλο του αντίστοιχου πλάτους (τόσο 4-way όσο και 8-way), γεγονός που εξηγείται από τις αργές inter-cluster προωθήσεις δύο κύκλων που εισάγουν οι clustered σχεδιασμοί.

Καλύτερη απόδοση παρουσιάζει η μέθοδος "dependency-load-steering" με χειροτέρευση του IPC περίπου κατά 22% και για τα δύο πλάτη. Για τις υπόλοιπες μεθόδους, στα 4-way μοντέλα, παρατηρήθηκε μείωση απόδοσης περίπου κατά 41% για τη μέθοδο "round-robin", 23% για τη μέθοδο "dependency-steering" και 28% για τη μέθοδο "loadcut". Στα 8-way μοντέλα, οι αντίστοιχες μειώσεις απόδοσης ήταν 49% για τη μέθοδο "round-robin", 26% για τη μέθοδο "dependency-steering" και 38% για τη μέθοδο "loadcut".

Οι απλούστερες μέθοδοι όπως η "round-robin" παρουσιάζουν μεγαλύτερες απώλειες απόδοσης επειδή δε λαμβάνουν υπόψη τις εξαρτήσεις μεταξύ των εντολών, με αποτέλεσμα να προκαλούνται συχνές και περιττές επικοινωνίες μεταξύ διαφορετικών clusters. Η μέθοδος "loadcut", αν και αποδίδει σημαντικά καλύτερα από τη "round-robin", δεν ξεπερνά σε απόδοση τη μέθοδο "dependence-load-steering". Αυτό είναι αναμενόμενο καθώς, παρόλο που οι εντολές load συχνά δημιουργούν αλυσίδες εξαρτήσεων, άλλες εντολές μπορούν επίσης να εισαγάγουν εξαρτήσεις. Η μέθοδος "dependence-load-steering" λαμβάνει υπόψη όλα τα είδη εξαρτήσεων και επιχειρεί να ομαδοποιήσει τις εξαρτώμενες εντολές στον μέγιστο δυνατό βαθμό για να μειώσει τις inter-cluster προωθήσεις. Τα αποτελέσματα της απόδοσης με βάση το IPC των μεθόδων μας παρουσιάζονται στις εικόνες 7.1 και 7.2.

1.5.2 Σχεδιασμοί με τέσσερα clusters

Για την καλύτερη μέθοδο οδήγησης (dependency-load-steering), πραγματοποιήθηκε σύγκριση της απόδοσης (από άποψη IPC) μεταξύ σχεδιασμών με δύο και τέσσερα clusters. Στους σχεδιασμούς τεσσάρων clusters ακολουθήθηκε η ίδια μεθοδολογία, διαχωρίζοντας περαιτέρω τις ουρές εντολών και τις λειτουργικές μονάδες. Τα μοντέλα τεσσάρων clusters ονομάστηκαν 4x1-way και 4x2-way, καθώς αποτελούνται από τέσσερα clusters 1-way και τέσσερα clusters 2-way αντίστοιχα.

Παρατηρήθηκε περαιτέρω μείωση του IPC, κάτι που ήταν αναμενόμενο, καθώς περισσότερα clusters οδηγούν σε μικρότερες δομές που γεμίζουν ταχύτερα (όπως οι ουρές εντολών), και οι τελεστές καταλήγουν συχνότερα σε διαφορετικά clusters από τις εξαρτώμενες εντολές τους. Υπολογίστηκε μείωση του IPC περίπου κατά 32% για το 4x1-way σε σύγκριση με το βασικό μοντέλο 4-way, και μείωση 36% για το 4x2-way σε σύγκριση με το βασικό μοντέλο 8-way.

Είναι σημαντικό να σημειωθεί ότι, παρόλο που το IPC μειώνεται, τα μικρότερα

clusters είναι πιθανό να επιτύχουν υψηλότερες ταχύτητες ρολογιού λόγω μικρότερων καθυστερήσεων καλωδίωσης (wire delays) και μειωμένου μήκους κρίσιμου μονοπατιού. Αυτή η αύξηση στην ταχύτητα του ρολογιού θα μπορούσε να αντισταθμίσει την απώλεια IPC, με πιθανότητα καλύτερης συνολικής απόδοσης για τον σχεδιασμό από μονολιθικούς επεξεργαστές. Τα αποτελέσματα της απόδοσης με βάση το IPC των σχεδιασμών με δύο και τέσσερα clusters παρουσιάζονται στις εικόνες 7.3 και 7.4.

1.5.3 Σχεδιασμοί με τέσσερα clusters, δύο κύκλοι καθυστέρησης

Για τους παραπάνω σχεδιασμούς, προσομοιώνουμε την επικοινωνία μεταξύ διαφορετικών clusters με δύο κύκλους καθυστέρησης αντί για έναν. Η αυξημένη αυτή καθυστέρηση, όπως είναι λογικό, επιφέρει μεγαλύτερη χειροτέρευση, συγκεκριμένα περίπου 55 και 59% για σχεδιασμό δύο και τεσσάρων cluster αντίστοιχα για τον 4-way, και 59% και 64% για σχεδιασμό δύο και τεσσάρων cluster αντίστοιχα για τον 8-way. Αυτό οφείλεται στο ότι οι εντολές καθυστερούν να λάβουν τις τιμές που αναμένουν, και παραμένουν στις δομές όπως την μνήμη αναδιάταξης και τις ουρές εντολών για περισσότερο, καθυστερώντας την εκτέλεση τους. Τα αποτελέσματα της απόδοσης με βάση το IPC των σχεδιασμών με δύο και τέσσερα clusters παρουσιάζονται στις εικόνες 7.5 και 7.6.

Αυτά τα αποτελέσματα είναι ενδεικτικά της ευαισθησίας που παρουσιάζουν οι σχεδιασμοί μας στις καθυστερήσεις επικοινωνίας, εφόσον η προσθήκη ενός μόνο επιπλέον κύκλου οδηγεί σε αρκετά σημαντική χειροτέρευση τη απόδοσης.

1.5.4 Στατιστικά σχετικά με inter-cluster προωθήσεις

Για να αναλύσουμε καλύτερα τον αρνητικό αντίκτυπο των καθυστερήσεων μεταξύ clusters στην εκτέλεση των εντολών, εισάγαμε οκτώ διακριτές στατιστικές κατηγορίες ώστε να καταγράψουμε διαφορετικά σενάρια κατανομής τελεστών. Αυτές οι κατηγορίες μετρούν πόσους πηγαίους τελεστές έχει μια εντολή και πόσοι από αυτούς βρίσκονται σε διαφορετικό cluster από την ίδια την εντολή, οδηγώντας σε inter-cluster προώθηση μεταξύ τους. Για παράδειγμα, για έναν σχεδιασμό με τέσσερα clusters, η "Κατηγορία 2" σημαίνει ότι η τρέχουσα εντολή, η οποία έχει δύο πηγαίους τελεστές, βρίσκεται στο cluster 0, αλλά οι τελεστές της βρίσκονται σε διαφορετικά clusters, όπως το cluster 1 και το cluster 2 αντίστοιχα.

Καταγράφοντας αυτές τις περιπτώσεις, μπορούμε να κατανοήσουμε καλύτερα πώς οι εξαρτήσεις δεδομένων μεταξύ clusters επηρεάζουν την απόδοση.

Πιο συγκεκριμένα:

- Κατηγορία 0: 1 τελεστής, 0 σε άλλο cluster
- Κατηγορία 1: 1 τελεστής, 1 σε άλλο cluster

- Κατηγορία 2: 2 τελεστές, 0 σε άλλο cluster
- Κατηγορία 3: 2 τελεστές, 1 σε άλλο cluster
- Κατηγορία 4: 2 τελεστές, 2 σε άλλο cluster
- Κατηγορία 5: 3 τελεστές, 0 σε άλλο cluster
- Κατηγορία 6: 3 τελεστές, 1 σε άλλο cluster
- Κατηγορία 7: 3 τελεστές, 2 σε άλλο cluster
- Κατηγορία 8: 3 τελεστές, 3 σε άλλο cluster

Τα Σχήματα 7.7 και 7.8 παρουσιάζουν το μέσο ποσοστό κάθε στατιστικής κατηγορίας για κάθε μέθοδο κατανομής.

Οι πιο συχνές κατηγορίες που σχετίζονται με inter-cluster προωθήσεις είναι οι κατηγορίες 1, 3 και 4. Οι κατηγορίες 0, 2 και 5 θεωρούνται καλές περιπτώσεις στον σχεδιασμό μας, γιατί σημαίνει ότι δεν υπήρξε καθόλου προώθηση μεταξύ διαφορετικών clusters. Με μια πρώτη ματιά, βλέπουμε ότι αυτές οι τελευταίες κατηγορίες σπανίζουν στην περίπτωση της μεθόδου "round-robin", κάτι που αποδεικνύει και τη χειρότερη συνολική απόδοση της μεθόδου αυτής.

Παρατηρούμε την εμφάνιση της κατηγορίας 1 για τις μεθόδους "round-robin" και "loadcut", κάτι που βγάζει νόημα, καθώς οι άλλες μέθοδοι που βασίζονται στις εξαρτήσεις λειτουργούν πολύ καλά σε εντολές με έναν τελεστή, επειδή η εξάρτηση αντιμετωπίζεται σωστά. Στις λίγες περιπτώσεις που παρατηρείται αυτή η κατηγορία και στις άλλες μεθόδους που βασίζονται στις εξαρτήσεις, είναι πιθανό να οφείλεται στο ότι οι ουρές εκκίνησης του επιθυμητού cluster ήταν γεμάτες, και έτσι η εντολή στάλθηκε αλλού.

Μπορούμε επίσης να δούμε την κατηγορία 3 να εμφανίζεται και στις μεθόδους που βασίζονται στις εξαρτήσεις, καθώς και στις "round-robin" και "loadcut". Αυτό ήταν επίσης αναμενόμενο, καθώς στις μεθόδους που λαμβάνουν υπόψη τις εξαρτήσεις και μια εντολή έχει δύο τελεστές, είναι πιθανό η εντολή να σταλεί στο cluster του πρώτου τελεστή, ενώ ο άλλος να βρίσκεται σε διαφορετικό cluster. Αυτό αποτελεί σημαντικό περιορισμό αυτής της μεθόδου, καθώς δεν εγγυάται ότι οι γονικές εντολές θα καταλήξουν στο ίδιο cluster. Για να αποφευχθεί αυτό, θα μπορούσε να εφαρμοστεί μια μέθοδος που βασίζεται σε "slices", όπως έχει προταθεί και σε προηγούμενες μελέτες.

1.5.5 Καθυστερήσεις λόγω πόρων

Για να αναλύσουμε περαιτέρω τα στοιχεία που περιορίζουν την απόδοση των σχεδιασμών μας, χρησιμοποιήσαμε τις τέσσερις στατιστικές κατηγορίες του gem5 που αναφέρονται παρακάτω, για να καταγράψουμε διαφορετικούς τύπους παύσεων που

σχετίζονται με πόρους. Αυτές οι παύσεις εμφανίζονται όταν κρίσιμοι υλικοί πόροι δεν είναι διαθέσιμοι, καθυστερώντας την εκτέλεση εντολών. Με την ξεχωριστή παρακολούθηση αυτών των περιπτώσεων, μπορούμε να κατανοήσουμε καλύτερα ποιοι περιορισμοί υλικού προκλήθηκαν λόγω του clustering.

Συγκεκριμένα, ορίζουμε τις εξής τέσσερις κατηγορίες:

- Κατηγορία 0: Απασχολημένη λειτουργική μονάδα (εκτέλεσης)
- Κατηγορία 1: Πλήρης ουρά load/store
- Κατηγορία 2: Πλήρης φάκελος καταχωρητών
- Κατηγορία 3: Πλήρης ουρά εντολών

Κατά τις προσομοιώσεις, αξιοποιήσαμε τα ενσωματωμένα στατιστικά του gem5 σχετικά με τις παύσεις λόγω πόρων (stalls), που καταγράφουν περιπτώσεις όπου οι εντολές καθυστερούν επειδή μονάδες εκτέλεσης, ουρές έκδοσης εντολών ή load/store ουρές είναι γεμάτες. Για παράδειγμα, η παύση τύπου "πλήρης ουρά εντολών" για τα συστήματα με clusters είναι ένας μετρητής που ελέγχει πόσες φορές όλες οι ουρές εντολών όλων των clusters είναι γεμάτες. Όπως έχουμε αναφέρει, όταν μια εντολή πρόκειται να σταλεί σε ένα cluster αλλά η ουρά του είναι γεμάτη, τότε η εντολή δρομολογείται σε άλλο διαθέσιμο cluster. Αν υπάρχει διαθέσιμη, δεν προκύπτει παύση. Αν όχι, τότε προκύπτει παύση λόγω πόρων.

Όπως φαίνεται στα Σχήματα 7.9 και 7.10 για τις σχεδιάσεις 4-way και 8-way αντίστοιχα, αυτό που αξίζει να σημειωθεί είναι το γεγονός ότι παρατηρούμε περισσότερες παύσεις τύπου “απασχολημένη μονάδα εκτέλεσης” σε σύγκριση με το βασικό μοντέλο. Αυτό συμβαίνει επειδή, όταν μια εντολή ανατίθεται σε ένα cluster, είναι διαθέσιμο μόνο ένα υποσύνολο των συνολικών μονάδων εκτέλεσης — δηλαδή, μόλις το μισό των μονάδων του βασικού μοντέλου.

Μια άλλη αξιοσημείωτη παρατήρηση είναι η μείωση των παύσεων τύπου “πλήρης ουρά load/store” στο μοντέλο 2x2-way με “dependency-load-steering” σε σχέση με το baseline 4-way. Είναι σημαντικό να σημειωθεί ότι οι εντολές μνήμης εισέρχονται στις ουρές load/store μόνο αφού έχουν εκδοθεί από την ουρά εντολών (IQ). Άρα αυτή η μείωση ενδέχεται να οφείλεται στο γεγονός ότι το μοντέλο με clusters διαχωρίζει τους πόρους εκτέλεσης μεταξύ δύο clusters, κάτι που περιορίζει πόσες εντολές μπορούν να εισέλθουν ταυτόχρονα στο στάδιο εκτέλεσης άρα και στις ουρές load/store.

1.6 Μεγαλύτεροι σχεδιασμοί

1.6.1 Μελλοντικοί Επεξεργαστές

Τα τελευταία χρόνια, οι σύγχρονοι επεξεργαστές έχουν αυξήσει σημαντικά το εύρος τους, τόσο στην αποκωδικοποίηση όσο και στην ταυτόχρονη εκτέλεση εν-

τολών. Νεότεροι πυρήνες, όπως οι Apple M4 και ARM Cortex-X925 (2024), υποστηρίζουν αποκωδικοποίηση έως 10 εντολές και εκτέλεση έως 19–23 εντολές ανά κύκλο. Στο πλαίσιο αυτό, η παρούσα διπλωματική θα παρουσιάσει την έννοια του clustering και θα εξετάσει την απόδοση των μεθόδων κατανομής εντολών που αναλύθηκαν προηγουμένως και σε ευρύτερους επεξεργαστές με πλάτος εκκίνησης 16 αλλά μετονομασίας/αποκωδικοποίησης 10, καθώς αυτό εξακολουθεί να αποτελεί bottleneck στους σύγχρονους επεξεργαστές.

1.6.2 Πειραματικό κομμάτι

Το βασικό μοντέλο πυρήνα που χρησιμοποιείται διαθέτει ενισχυμένους πόρους, αλλα και ενισχυμένη ιεραρχία μνήμης, σε σχέση με τις προηγούμενες αρχιτεκτονικές ώστε να υποστηρίζονται οι μεγαλύτερες απαιτήσεις σε εύρος ζώνης. Τα παραπάνω χαρακτηριστικά του πυρήνα παρουσιάζονται αναλυτικά στον πίνακα 8.1.

Τρέχουμε τα ίδια πειράματα με τα προηγούμενα κεφάλαια για καθυστερήσεις inter-cluster προωθήσεων ενός και δύο κύκλων του ρολογιού για clustered σχεδιασμούς 2x8 και 4x4 . Στα διαγράμματα 8.2 και 8.3 παρουσιάζονται τα αποτελέσματα των πειραμάτων σχετικά με το IPC για τις δυο παραπάνω καθυστερήσεις που αναφέρθηκαν. Παρατηρούμε μια χειροτέρευση στο IPC περίπου 15% και 27% για τους σχεδιασμούς 2x8 και 4x4 αντίστοιχα, για καθυστέρηση ενός κύκλου και περίπου 57% και 60% για τους σχεδιασμούς 2x8 και 4x4 αντίστοιχα, για καθυστέρηση δύο κύκλων.

Αξιοσημείωτο είναι ότι οι επιδόσεις στους 2×8 και 4×4 clustered σχεδιασμούς υποβαθμίζονται λιγότερο σε σύγκριση με τις αρχιτεκτονικές 4-way και 8-way, ιδίως όταν υπάρχει καθυστέρηση ενός κύκλου στην επικοινωνία μεταξύ clusters. Η μικρότερη αυτή επίπτωση οφείλεται κυρίως στο μεγαλύτερο πλάτος εκκίνησης αυτών των clustered σχεδιασμών, που επιτρέπει την ταχύτερη προώθηση ανεξάρτητων εντολών ακόμα και όταν κάποιες καθυστερούν λόγω εξαρτήσεων. Η αυξημένη ροή εντολών και οι ενισχυμένοι πόροι αυξάνουν την πιθανότητα να υπάρχουν εναλλακτικές εντολές για εκτέλεση, με αποτέλεσμα η επίδραση των καθυστερήσεων στο IPC να είναι πιο ήπια σε σχέση με πιο στενές αρχιτεκτονικές.

1.7 Συμπεράσματα

1.7.1 Συμπέρασμα

Η διπλωματική αυτή αναλύει βασικές πηγές καθυστέρησης σε επεξεργαστές με μεγάλο εύρος, εστιάζοντας στη λογική έκδοσης εντολών και στην προώθηση δεδομένων. Αυτά τα υποσυστήματα αναμένεται να αποτελέσουν ακόμη πιο σημαντικά σημεία συμφόρησης στο μέλλον, καθώς το πλάτος εκκίνησης και τα παράθυρα εκκίνησης συνεχίζουν να αυξάνονται. Για την αντιμετώπισή των περιορισμών

αυτών, εξετάζεται η τεχνική του clustering, με στόχο τη διατήρηση υψηλής απόδοσης και υψηλών συχνοτήτων ρολογιού μέσω μικρότερων και λιγότερο πολύπλοκων δομών.

Υλοποιήθηκαν διάφορες clustered αρχιτεκτονικές με διαφορετικά πλάτη εκκίνησης και διαφορετικές μεθόδους κατανομής των εντολών στα clusters και αξιολογήθηκε η απόδοσή τους σχετικά με τις Εντολές-ανά-Κύκλο (Instructions per Cycle - IPC). Η καλύτερη επίδοση επιτεύχθηκε με τη μέθοδο "Dependency-Load-Steering", που λαμβάνει υπόψη εξαρτήσεις μεταξύ των εντολών αλλά και τον φόρτο των clusters, σε αντίθεση με τις άλλες μεθόδους που αναπτύξαμε που οι inter-cluster πρωθήσεις χειροτερεύουν σημαντικά το IPC.

Παρά τη μείωση στο IPC σε σχέση με τα βασικά μονολιθικά μοντέλα, το clustering έχει προοπτικές για υπεροχή στη συνολική απόδοση χάρη στα πιθανά οφέλη σε συχνότητα λειτουργίας.

1.7.2 Μελλοντική μελέτη

Η διαχείριση της επικοινωνίας μεταξύ clusters αποτελεί βασική πρόκληση. Οι μέθοδοι που αξιοποιούν τις εξαρτήσεις μεταξύ εντολών αποδείχθηκαν πιο αποτελεσματικές από απλούστερες προσεγγίσεις όπως Round-Robin ή Loadcut, αλλά υπάρχει ακόμη περιθώριο για βελτίωση των καθυστερήσεων επικοινωνίας. Μελλοντικές έρευνες θα μπορούσαν να εστιάσουν στη βελτιστοποίηση των δυναμικών τεχνικών καθοδήγησης με απλούστερους μηχανισμούς υλικού και ακριβέστερες ευρετικές μεθόδους πρόβλεψης.

Επιπλέον, θα μπορούσαν να διερευνηθούν εναλλακτικές διαμορφώσεις clusters, όπως ετερογενή clusters, για συγκεκριμένους τύπους εντολών το κάθε ένα, αντί των ομοιογενών που χρησιμοποιήθηκαν στην παρούσα εργασία. Η δυναμική προσαρμογή των διαμορφώσεων κατά την εκτέλεση αποτελεί επίσης μια υποσχόμενη κατεύθυνση.

Τέλος, καθώς η τεχνολογία εξελίσσεται, οι καθυστερήσεις καλωδίωσης κυριαρχούν στο κόστος επικοινωνίας. Μελλοντικές έρευνες θα μπορούσαν να εξετάσουν τεχνικές όπως η βέλτιστη διάταξη των clusters και η χρήση ταχύτερων τεχνολογιών διασύνδεσης.

Chapter 2

Introduction

In pursuit of higher performance, processors often rely on wider instruction windows and issue widths to fully exploit instruction-level parallelism (ILP). However, these techniques introduce significant complexity, ultimately limiting the achievable clock frequencies. To mitigate this, clustering has emerged as a strategy: it partitions processor resources into independent execution groups, each responsible for handling a subset of instructions. This division enables higher clock frequencies and improved scalability by shortening data paths and simplifying control logic. Nevertheless, clustering also introduces challenges, particularly the overhead of communication between clusters, which can become time-consuming and degrade performance.

In this work, we explore different distribution methods for the instructions into the clusters –called steering methods. Initially, we evaluate simple mechanisms –such as round-robin– that do not consider instruction dependencies. Next, we investigate steering techniques based on instruction dependencies and register information, which are crucial for minimizing communication penalties. Since unbalanced instruction distribution can create bottlenecks, we incorporate load-balancing methods that dynamically assign instructions based on current cluster utilization.

Our analysis extends across various core widths, including narrow (4-way and 8-way) and wide (16-way) cores, inspired by commercial designs. We compare the performance of these steering methods and configurations against baseline, non-clustered models. This evaluation provides insights into how clustering interacts with core width, steering strategies, and workload characteristics. Notably, our study of wide cores highlights how clustering can help mitigate frequency degradation in future high-performance processors, while still maintaining high performance. Evaluating such a large scale design allows us to observe how our steering strategies and dependency tracking mechanisms perform under heavy instruction throughput and intense resource pressure. We aim to provide meaningful insights for the next-generation processors, where balancing complexity, performance, and scalability becomes increasingly criti-

cal.

The rest of this thesis is organized as follows: Chapter 3 is an introduction on Out-of-Order Processors, Chapter 4 introduces clustering as a solution to frequency limitation issues, Chapter 5 describes our work's methodology, Chapter 6 outlines our experimental work and performance evaluation and Chapter 7 provides insights on clustering in wider cores.

Chapter 3

Out-of-Order Processors

3.1 Out-of-order Processors

Out-of-order (OoO) execution is a key element of modern high-performance processor design. Unlike simpler in-order pipelines, where instructions are executed strictly in the sequence they appear in the program, out-of-order processors dynamically schedule instructions based on the availability of their operands. This allows independent instructions to be executed as soon as resources and data are available, rather than waiting for previous instructions to complete. In this way, OoO execution effectively exploits instruction-level parallelism (ILP) and improves resource utilization.

Modern OoO processors rely on sophisticated hardware structures to ensure correctness and efficiency. These include register renaming to eliminate false dependencies, reservation stations and issue queues to buffer ready instructions, a reorder buffer (ROB) to preserve program order at commit time, and multiple functional units to execute instructions in parallel. These features work together to let the processor run instructions faster while still handling dependencies and avoiding hazards.

A key advantage of OoO execution is its ability to mitigate pipeline stalls. Instructions waiting for operands can remain in reservation stations without blocking the pipeline, while independent instructions proceed through execution. This reduces idle time in pipeline stages and increases throughput. Furthermore, most modern OoO processors are superscalar, meaning they are capable of fetching, decoding, issuing, and executing multiple instructions per cycle. The combination of superscalar design and out-of-order execution is crucial for achieving high single-threaded performance in today's general-purpose CPUs.

3.2 Need for increasing performance

To enhance application performance and meet the growing computational demands of modern workloads, processor designers have increasingly focused on

integrating a larger number of cores onto a single chip [3]. This multi-core design has proven highly effective, particularly for applications that can be parallelized across multiple threads or processes. By enabling thread-level parallelism (TLP), multi-core processors can execute several threads simultaneously, thereby significantly improving overall throughput and reducing the time required to complete operations. This parallelism allows cloud providers to efficiently manage and allocate resources, supporting high-demand applications and workloads that can now be broken down and distributed across cores efficiently.

However, improving single-thread performance remains crucial, especially for applications with limited parallelism. In many real-world applications, performance bottlenecks arise from single-thread execution paths, and thus the efficiency of each individual core remains crucial [4].

One strategy to achieve this is maximizing instruction-level parallelism (ILP) by implementing wider instruction windows and widths. This increases the likelihood of finding independent instructions that can be executed simultaneously, improving overall performance. However, such approaches come with trade-offs: wider windows and issue widths typically require more complex control logic, more power consumption, and significantly impact the clock frequency due to the increased complexity and size of the structures. In order to achieve high single-thread performance simply scaling traditional out-of-order structures, would definitely not be the way to do so. Larger instruction windows and wider issue widths, while capable of extracting more parallelism, face great limitations, and motivated researchers to explore alternative approaches –one of these being "clustering"– that can approximate the benefits of wider structures without their associated drawbacks, as we will analyze later.

3.3 Superscalar pipeline explained

To better understand the limitations of conventional scaling, we will consider a typical superscalar pipeline and describe the stages that precede execution of an instruction. Instructions are fetched in parallel, and when a branch is encountered, speculative execution occurs: the program counter is updated based on a prediction to avoid stalling "fetch". These fetched instructions proceed through decoding and register renaming, then enter the reorder buffer (ROB) and the issue queue.

In the issue stage, instructions wait until their operands become available, either from the register file or by bypassing from executed instructions. The selection logic selects multiple ready instructions each cycle (according to the processor's issue width) and dispatches them to the appropriate functional units. This mechanism allows superscalar processors to exploit ILP and execute mul-

tiple instructions per cycle.

3.4 Superscalar Processor Challenges

As we already mentioned, in recent years, superscalar processor design faces significant challenges as higher levels of instruction-level parallelism are needed. The growth in many hardware components implies increases in power consumption and complexity, which further decreases clock speed and adds extra wire delays, which was proven by various studies. Palacharla et. al. for example, developed analytical models to estimate the complexity of the issue logic, bypass logic, rename logic and register file. Their approach, defines complexity as the delay through the critical path of a piece of logic. Notably, Palacharla et al. identified the window wakeup that wakes instructions up waiting for their operands, and selection logic that selects instructions for execution, as some of the most critical components in terms of delay. In particular, the selection logic becomes increasingly complex to scale efficiently as the pool of instructions becomes larger. They came to the conclusion that the complexity of issue logic scales quadratically with the issue width and window size, and the complexity of the bypass logic scales linearly with issue width. They also concluded that wire delays contribute significantly to the overall system latency [5].

So using the insights by the complexity analysis of previous studies and closely examining the pipeline stages, it becomes clear how conventional monolithic designs face practical scalability limitations. This motivates the exploration of other alternatives –such as clustered microarchitectures– that aim to distribute complexity and mitigate these bottlenecks while preserving high ILP.

Chapter 4

Clustered Microarchitectures

4.1 Clustering as a solution

As an alternative to wide issue windows and just scaling-up the design, clustering was proposed to approximate wider issue widths and instruction windows while maintaining the clock rates achieved by smaller structures. Clustering in general terms, means dividing resources into smaller, independent clusters, each handling a subset of instructions with its own set of resources –more specifically a separate issue queue and functional unit pool. Using this method could significantly reduce wire delays, helping to preserve high clock frequencies even as the overall system scales. Additionally, clustering enhances scalability by allowing more instruction-level parallelism to be supported without requiring larger, monolithic structures. The scheduling logic also becomes simpler and faster, since each cluster only needs to manage a smaller pool of instructions and resources. Overall, clustering is an effective way to approximate the performance of wide-issue architectures while maintaining the speed, efficiency, and simplicity of narrower designs.

4.2 Clustering limitations

In an out-of-order processor, when an instruction produces a result, it bypasses that result to its dependent instructions that are ready to be woken up, allowing them to execute as soon as possible instead of waiting for the value to be written back to the register file. More specifically, forwarding paths are hardware mechanisms that allow the result of an instruction to be sent directly from the functional unit (FU) where it was produced to another FU processing an instruction that depends on it. When having clusters, those bypasses, if they are between instructions of different clusters, take more time due to longer wires, and that is why we have some extra delays. In order to achieve valuable results the instructions need to be distributed to the clusters in a way that cluster-induced delays are minimized as much as possible –we will refer to

those methods as “steering methods”.

While maximum performance is the main goal, the cost and complexity of the steering methods is also taken into consideration. Instead of relying on highly complex hardware structures or computationally expensive run-time methods, we focused on approaches that achieve effective instruction distribution with low hardware overhead and simple scheduling logic.

Chapter 5

Related Work

Clustering was proposed by Palacharla et al. as a solution that reduced the clock cycle for superscalar processors. After analyzing the pipeline delays, they realised that issue logic and bypass logic dominate the overall delay. So they started by replacing the classic instruction window, where the instructions wait for their operands to become available, with parallel FIFO buffers that are simpler structures and that facilitate faster clock. They proposed a method that steers instructions into the FIFOs based on their dependencies. In order to further fasten they clock the initiated clustering. They used four FIFOs for each cluster, eight at total, and separated the functional units for the two clusters. They also implemented the same design for associative issue windows instead of FIFOs, and found no significant difference in their results. They ran some simulations and came to the conclusion that in terms of IPC, this new microarchitecture was nearly as effective as the window-based model with a small IPC degradation. After the critical path analysis, they realized that a 2x4-way clustered microarchitecture would achieve at least the same clock speed as a 4-way window-based microarchitecture, with a possible total performance improvement.[5]

Baniasadi and Moshovos compared several different steering policies in addition to the dependence-related one, in order to reduce stalls due to the cluster bandwidth and the inter-cluster bypasses. They implemented four clusters with a separate scheduler and functional units. The methods they adapted were both non-adaptive and adaptive (based on whether they change during run-time or not). Their adaptive methods were the simple first-fit (steers instruction into one cluster until it fills-up), modulo-N ones (something like a round-robin but with N instructions at each cluster each time), the dependence-based one, like previously, methods that change the cluster when branches or loads occur –like the method we used– and the "slice" which is worth mentioning, since it expands the simple dependence-based advantages. More specifically, as we already discussed, the dependence-based methods steer the instructions into a cluster based on where their operands reside. Moshovos et. al. implemented this

method that uses register renaming information and steers instructions to the same cluster as its parents, and more specifically the parent with the youngest program order –which is similar to our logic. They noted that there was no significant difference with other similar alternatives. Using this method though, sometimes leads to parents residing in different clusters. That is why the "slice" method was introduced, to make sure all the parent instructions are assigned to the same cluster, to further reduce inter-cluster bypass stalls via a PC-indexed table; however, this improvement that requires an additional table, adds to the hardware cost. A simple modulo-N and specifically MOD-3 is the one that performs best from the non-adaptive methods, because apparently it does a good job balancing the clusters' load and bypass-induced stalls. While there is a performance (IPC) degradation here, clustering could eventually be beneficial due the higher frequencies achieved. The adaptive methods use voting-based methods that aim to improve instruction distribution by identifying problematic cluster assignments. They use cluster prediction tables to track the assignments of one the non-adaptive methods, adapting the steering based on past behavior of the program. While effective, these adaptive methods are beyond the scope of this work.[6]

E. Tune, D. Liang, D.M. Tullsen, and B. Calder studied a clustered model with two and four clusters and replicated register files. They introduced a critical-path predictor and suggested improvements to two dependency-based cluster assignment methods that also consider how critical an instruction is. In their approach, when an instruction has two source operands coming from different clusters, the critical-path predictor is used to decide between the two options -the instruction is then assigned to the cluster of its more critical predecessor.[7]

Salverda et. al. also analyzed the critical path, to uncover the causes of performance loss and added a new criterion to the existing steering methods, the load of the clusters. As we have already mentioned, the only way to increase the IPC is by increasing the issue width and window, but with a cost to the frequency. That is why they introduced clustering, (two, four and eight clusters) and presented several steering methods to avoid as much as possible both bypass-related and contention-related stalls. So throughout their work, they used criticality counters, and metrics to define critical parent instructions and to steer instructions based on those, while taking into account balancing out the clusters too. All their methods performed relatively close to the baseline, with some expected IPC degradation, and added complexity in both design and hardware due to the criticality analysis.[8]

While all the previous works focused on small issue widths and higher frequencies, Michaud et. al. introduced a wide-issue, MOD-64, dual clustered microarchitecture -similar to steering methods mentioned before- with the goal

of increasing the IPC this time, on a constant frequency. This could only happen with clustering. With this steering method, owing to data-locality, much fewer inter-cluster bypasses occur (of about three cycles) and by increasing the issue width and window and doubling the parameters of their cores, the IPC would significantly increase. In contrast to previous work, which aimed to improve performance by increasing frequency while keeping the issue width the same, Michaud et al. focused on maintaining the same frequency while doubling the issue width to achieve higher IPC. Doubling issue width and window size though, may lead to higher IPC, but it comes with several challenges. Various structures must be enlarged to support the wider pipeline, making this approach less efficient in terms of hardware size and power consumption.[4]

Another notable contribution to the space of decentralized microarchitectures is the PEWs (Parallel Execution Windows) model studied by Ranganathan and Franklin. PEWs address the scalability limitations of conventional superscalar processors by decentralizing instruction execution. Instead of relying on a single, centralized issue window, the microarchitecture distributes instructions across multiple parallel execution windows (PEWs) based on register data dependencies. This ensures that dependent instructions are assigned to the same pew, reducing inter-window communication delays. In the PEWs architecture, the steering scheme is similar to the dependence-related cluster assignment mechanism analyzed before: it assigns an instruction to the cluster where the source operand is to be produced, except when it has two operands that are produced in different clusters, in which case the algorithm tries to minimize inter-PEW distance by creating a ring-like network.[9]

Regarding to PEWs and reducing their communication delays, A. Aggarwal and M. Franklin explored several methods to dynamically generate instruction replicas -various heuristics selectively replicate instructions in the clusters where their results are needed- to reduce inter-cluster communications. They proposed and investigated two replication schemes, and found that instruction replication gives a notable IPC increase over a simple balanced instruction distribution algorithm that focuses both on load balancing and reduced inter-cluster communication. Finally, their replication techniques were extended and found that when the techniques are used in combination, the IPC increases by about 5-10% over the balanced algorithm and is very close to the maximum possible IPC with a zero-cycle inter-cluster communication hypothetical processor.[10]

Another approach was introduced by Canal et al., who focused on the benefits of clustering for speedup rather than directly improving the clock cycle. They achieved this by splitting the processor into one integer and one floating-point (FP) cluster, and applying run-time dynamic scheduling techniques. These methods involve load-balancing and slicing load and branch in-

structions, which, despite leading to a lower IPC, result in overall speedup due to better utilization of the clusters and reduced resource contention. In contrast, Canal et al.'s approach takes this further by integrating dynamic scheduling and load balancing that require real-time decision-making during execution. This method, however, relies on advanced hardware and complex scheduling logic to effectively assign instructions to the clusters, which goes beyond the scope of our current work.[11]

That is why in our work, we start by keeping the issue widths small, with 2-way and 4-way clusters, and simple steering techniques that avoid complex hardware like critical path analysis that was mentioned before. To track dependencies, we could use simple dependency tables that monitor register information, as we will later describe. This would allow us to track operand dependencies with minimal hardware overhead, avoiding the need for complex resources. By using such an approach, we aim to steer instructions efficiently while keeping the system simple and hardware-friendly.

Chapter 6

Methodology

6.1 Simulator and Benchmarks

Before we describe various steering methods, we describe our methodology. We used gem5 simulator to implement our clustered design, as well as the steering methods.[12]

To support our clustered architecture and run the chosen benchmarks, we compiled gem5 with a RISC-V target ISA. RISC-V is a modern, open-standard Instruction Set Architecture that has rapidly gained recognition in both academia and industry. It was designed to be simple, efficient, and easy to extend, without having to carry over any outdated or complicated features from older architectures. It incorporates a modular design, which allows users to include only the instruction set features they need. This flexibility, combined with gem5's support for RISC-V, made it an ideal choice for implementing and evaluating our proposed microarchitectural changes.[13]

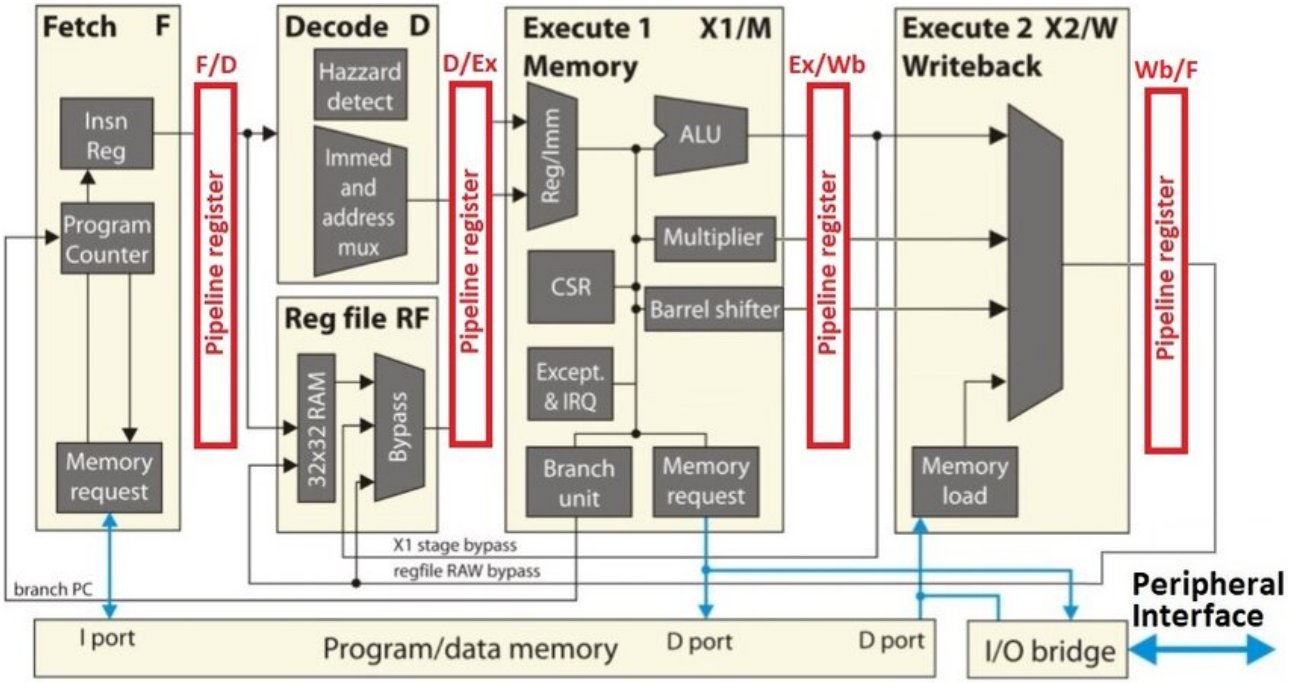

Figure 6.1: RISC-V Processor Block Diagram showing the four-stage pipeline architecture.[1]

More specifically, we used gem5's out of order CPU model "O3CPU" but then modified it to suit our needs. This model is highly timing accurate, and actually executes instructions at the execute stage of the pipeline. Most simulator models execute instructions either at the beginning or end of the pipeline -such as SimpleScalar and old CPU models used in gem5.

The "O3CPU" model simulates a generic out-of-order pipeline based on physical-register-file architectures, like the DEC Alpha design. Seven pipeline stages are modeled: fetch, decode, rename, issue, execute, writeback and commit –later we will see that issue/execute/writeback is considered as one stage. This model is highly configurable, so parameters related to the branch predictor, the pipeline stage widths, the instruction queue entries (IQ), the reorder buffer (ROB) entries and load/store queues sizes (LSQ). Also, the functional units number is configurable.[14] While the standard RISC-V implementation shown in Figure 6.1 uses a simpler four-stage pipeline, our work is based on gem5's more complex seven-stage pipeline model to implement our clustered architecture, because it offers the detailed microarchitectural framework needed for our research. The base configuration, as well as our clustered designs will be described in detail later on. Then we evaluated, in terms of performance, our methods by using the SPEC2017 benchmarks, and more specifically the SPECspeed2017 Integer and Floating point suites that we compiled for the RISC-V architecture using cross-compiling tools.

6.2 Steering Methods

We implement a clustered design, where we separate the functional units of the baseline design into clusters, by creating two functional unit pools, as well as the instruction queue, so that each cluster has its own. Instruction queue is the buffer that holds the instructions until their operands are ready.

When the instruction queue of the cluster we intend to assign an instruction is full, we simply place it to another one and assign it to the corresponding cluster, to avoid stalls. Otherwise, if all instruction queues are full, a resource-induced stall occurs. Then, we introduce four different steering methods, that steer the instructions into the clusters with a different criterion:

- Round Robin: Instructions are assigned to a cluster in a round-robin fashion. For example, for a two-cluster design, one instruction is assigned to cluster 0, the next into cluster 1, then the next to cluster 0 and so on. This is implemented at "fetch" stage, making no significant difference, if we place the logic elsewhere (for example decode).

- Dependence-based: This method leverages the data dependencies of instructions, in order to reduce inter-cluster bypasses and the induced de-

lays. This is done by using register-related information, that we retrieve at the rename stage. More specifically, when an instruction is being renamed, we find where the parent instructions were processed, based on source registers information (identifiers that are assigned to the destination registers, based on where they were processed). This way, we try to assign the instruction into the same cluster as the parent instruction of the first operand. If an instruction has no dependencies, or the dependencies are no longer valid due to write-backs of the parent instructions, it is sent in a random way to the instruction queues, and hence the clusters.

- Dependency-load-based: This follows the exact methodology of the previous one along with a new parameter, which is the relative-load of the clusters. More specifically, for the instructions that have no dependencies, the cluster with fewer instructions processed by its functional units is chosen.

- Loadcut: This method takes advantage of the fact that when a load instruction occurs, a dependency chain is usually created for the following instructions. More specifically, there is a higher possibility for those instructions to be dependent, so if they are placed in the same cluster, inter-cluster bypasses are less likely to happen. That is why we assign all the instructions to a cluster until a load instruction occurs, so then we switch clusters, if there are no multiple adjacent loads (in this case we don't switch clusters). This happens at the "dispatch" stage, right when an instruction is placed in the instruction queue of a cluster.

6.3 Implementation on Gem5

To simulate a clustered architecture in gem5, we first introduce new simulation parameters that define the total issue width, the number of clusters, and additional core configuration options. These parameters are defined in the configuration scripts (e.g., BaseO3CPU.py) and are used to scale architectural parameters accordingly. We configure distinct pipeline widths, instruction queue sizes, and instantiate separate functional unit pools—one for each cluster. We create multiple FUPool instances (e.g., fuPool1, fuPool2, according to the number of clusters), each initialized with identical sets of functional units, but treated independently. These pools are passed down into the pipeline logic.

In the execution stage (inst_queue.cc), when an instruction is selected for issue, we use the instruction's cluster_id –an additional field we introduced in the DynInst class– to determine which functional unit pool is allowed to be used.

The FU selection logic is modified so that only the functional units in the corresponding pool are considered valid. This ensures strict separation of resources between clusters. Instruction steering is implemented earlier in the pipeline, during the fetch and rename stages (depending on the method), where we assign cluster_id values. For example in simple methods like round-robin, the assignment happens in "fetch" but in methods that register information is needed, the assignment happens in "rename" –a cluster_id field is also added to physical registers to track dependencies between instructions. For instance, when an instruction is being renamed, its operands are closely observed –by their id's–, in order for the instruction to be assigned to the cluster where one of its parents resides. We make sure that when a parent instruction is committed –so there is not a possibility for a bypass, the value is retrieved by the register file– the dependence is not taken into consideration. Then the other parent instructions are examined for the cluster assignment. In the dependence-related methods, if the dependencies are not valid like in the cases described before, other techniques are used.

6.4 Hardware implementation

All four steering mechanisms can be implemented with relatively simple hardware support. The "round-robin" method is straightforward and can be achieved using a simple counter. The "dependence-based" steering relies on register dependency information already available at the rename stage, so the additional logic mainly involves tracking the cluster id's of previous destination registers –which can be done through small tables or tags. The "dependency-load" method extends this with a basic counter to monitor each cluster's FU activity, which again is lightweight. Finally, the "loadcut" mechanism requires detection of load instructions at dispatch and a control logic for cluster switching. Overall, these methods achieve hardware simplicity, making them practical for real implementations.

6.5 Our baseline

Our baseline configuration is an out-of-order superscalar processor designed to issue multiple instructions per cycle to its functional units. Specifically, we model two core designs with different issue widths: one capable of issuing 4 instructions per cycle and another capable of issuing 8 instructions per cycle. Each design is simulated with a five-stage pipeline structure that models the seven pipeline stages mentioned before: Fetch, Decode, Rename, Issue/Execute/Writeback, and Commit.

To better model real-world architectures, we drew inspiration from known

commercial designs. The 4-way issue core is designed to resemble ARM's Cortex-A76 in terms of microarchitecture characteristics, while the 8-way issue core is similar to Intel's Golden Cove microarchitecture. Also, the 8-way core features larger structures to support its higher instruction throughput. Specifically, it has a larger instruction queue (IQ) of 80 entries compared to 48 in the 4-way core. The load and store queues (LQ/SQ) are also expanded to 32/48 entries in the 8-way core, whereas the 4-way core has 16/16 entries. The reorder buffer (ROB) size is significantly increased from 128 in the 4-way core to 512 in the 8-way core, allowing for deeper out-of-order execution. Additionally, the number of integer and floating-point functional units (INT/FP) is increased from 128/192 in the 4-way core to 280/332 in the 8-way core to match the wider issue width. Finally, both cores have the same cache configurations: the L1 data and instruction caches are 32 KB, 2-way set-associative, and the L2 cache is 2 MB, 16-way set-associative. [15, 16] These differences in the 8-way core are necessary to ensure that the additional issue bandwidth is utilized without being bottlenecked by other pipeline components. Finally the 8-way features double the number of functional units compared to the 4-way. This increase in functional units is necessary to accommodate the higher instruction throughput of the wider issue width.

Baseline Configuration		
	4-way	**8-way**
IQ	48	80
LQ/SQ	16/16	32/48
ROB	128	512
INT/FP	128/192	280/332
L1-D	32 KB, 2-way	32 KB, 2-way
L1-I	32 KB, 2-way	32 KB, 2-way
L2	2 MB, 16-way	2 MB, 16-way

Table 6.1: Baseline configuration.

6.6 Our clustered designs

At first, we implemented both a 2x2-way design which is a clustered microarchitecture of two 2-way clusters, in terms of issue width, and a 2x4-way design which is a clustered microarchitecture of two 4-way clusters, with each cluster featuring instruction queues at half capacity and half the number of functional units compared to the baseline with the corresponding issue width. For example, for the 2x2-way design we will now have two separate issue queues of 24 entries each, while for the 2x4-way design we will have two issue queues of 40 entries.

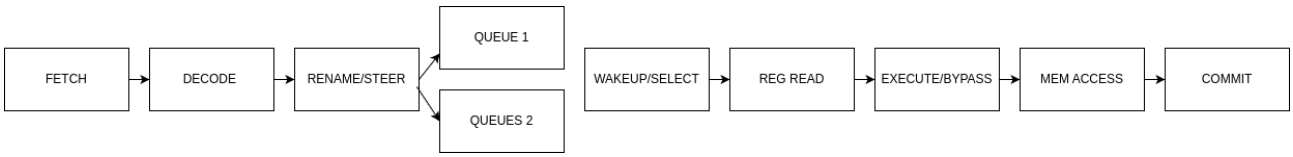

Figure 6.2: Clustered Out-of-Order Pipeline Architecture.

The clustered out-of-order pipeline architecture, shown in Figure 6.2, organizes instruction execution into two independent clusters. Instructions pass through the front-end stages — Fetch, Decode, and Rename —before being distributed into two separate instruction queues (Queue 1 and Queue 2). Each queue feeds its respective cluster, which contains its own pipeline stages: Wake-up/Select, Register Read, Execute/Bypass, Memory Access, and Commit. This design enables parallel instruction processing across clusters, improving throughput and allowing for more efficient resource utilization.

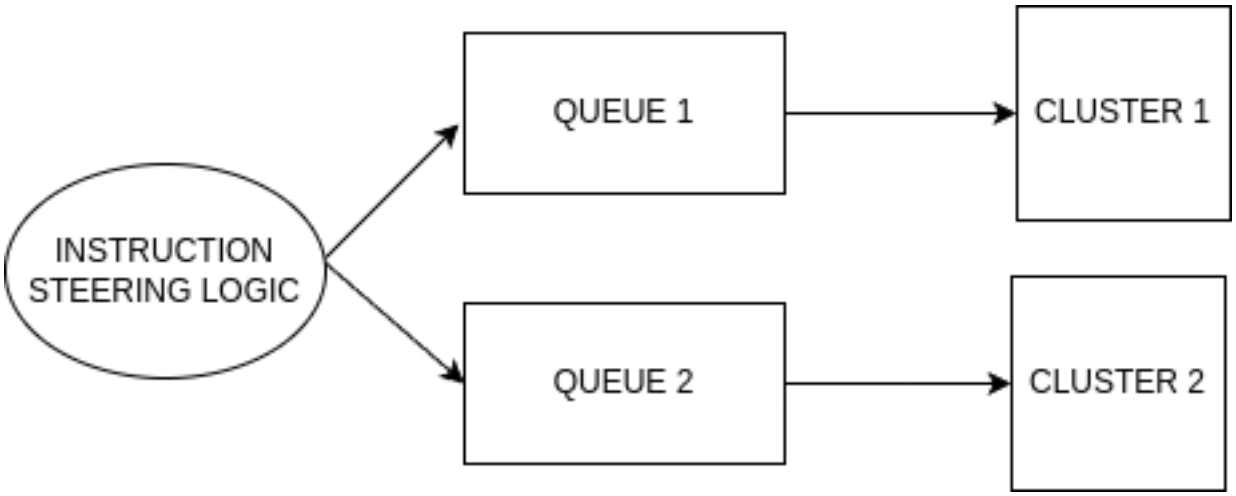

Figure 6.3: Instruction Steering Mechanism.

The instruction steering, illustrated in Figure 6.3, is responsible for dispatching instructions to either Queue 1 or Queue 2, depending on the number of clusters and the steering policy used (e.g., round-robin, dependency-based). Each queue is associated with its own execution cluster, enabling distributed and parallel instruction execution across the system.

We categorize data bypasses into two types which we will model for our simulations: local bypasses and inter-cluster bypasses. Local bypasses are responsible for bypassing results from one functional unit to another, within the same cluster. Inter-cluster bypasses are responsible for bypassing results between functional units residing in different clusters. We consider local bypasses to be accomplished in a single cycle, while inter-cluster bypasses in two cycles, so one cycle of an "inter-cluster delay" is induced. To understand the impact of this delay on performance, we will later conduct simulations with an inter-cluster delay of two extra cycles instead of one. By analyzing this variation, we

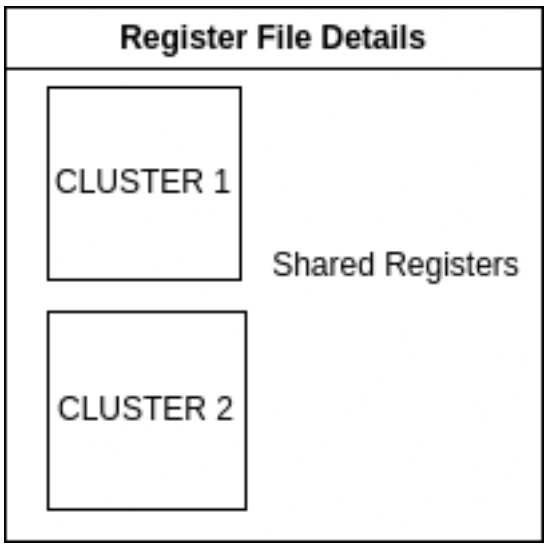

Figure 6.4: **Clustered Register File Organization.** This figure presents the organization of the register file in a clustered microarchitecture.

can evaluate how sensitive the system is to inter-cluster communication overhead -that is, how much the additional cycle affects the performance of our design.

Finally, in Figure 6.4 the organization of the register file in a clustered microarchitecture is presented. Each cluster has access to the register file, while a subset of registers is shared across clusters to enable communication and data consistency.

Chapter 7

Evaluation

In this study, we compiled and simulated the SPEC2017 benchmarks, focusing specifically on the SPECspeed2017 Integer and Floating Point suites, targeting the RISC-V architecture. The benchmarks were executed with a maximum instruction limit of 1 billion, preceded by a warm-up phase of 5 million instructions to stabilize the system's performance. This ensures that before taking results, the system has reached the execution phase, so performance can really be measured.

The following table lists the SPEC2017 benchmarks utilized in the simulations, along with their respective types and application domains.[17]

Benchmark	**Type**	**Application Area**
600.perlbench_s	INT	Perl interpreter
602.gcc_s	INT	GNU C compiler
605.mcf_s	INT	Route planning
620.omnetpp_s	INT	Discrete Event simulation - computer network
623.xalancbmk_s	INT	XML to HTML conversion via XSLT
625.x264_s	INT	Video compression
631.deepsjeng_s	INT	Artificial Intelligence: alpha-beta tree search (Chess)
641.leela_s	INT	Artificial Intelligence: Monte Carlo tree search (Go)
648.exchange2_s	INT	Artificial Intelligence: recursive solution generator (Sudoku)
657.xz_s	INT	General data compression
603.bwaves_s	FP	Explosion modeling
607.cactuBSSN_s	FP	Physics: relativity
619.lbm_s	FP	Fluid dynamics
638.imagick_s	FP	Image manipulation
644.nab_s	FP	Molecular dynamics
649.fotonik3d_s	FP	Computational Electromagnetics

Table 7.1: List of SPEC2017 Benchmarks Used in the Simulations, Including Their Types and Associated Application Areas.

7.1 Performance results of two-cluster microarchitectures

Figures 7.1 and 7.2 compare performance, in terms of instructions per cycle (IPC), of the four steering methods we described, for the clustered microarchitectures (both 4-way and 8-way) against that of the baseline design. For most of the benchmarks, we see a small performance degradation for all the steering methods, which we expected due to the slower inter-cluster communication. We find that, from all the steering methods, the smallest degradation is by "dependence-load-steering" model with average IPC reduction of approximately 22% and for the both 2x2-way and the 2x4-way designs respectively. In addition, for fourteen of all sixteen benchmarks the IPC degradation is approximately 16% for both widths, while for benchmarks like "x264","mcf" there is a dramatic IPC decrease of approximately 50-60%. For the rest of the steering methods for the 4-way we observed a degradation of performance of 41% for "round-robin", 23% for "dependency-steering", 28% for "loadcut". For the 8-way designs, for the rest of the steering methods we observed a degradation of performance of 49% for "round-robin", 26% for "dependency-steering", 38% for "loadcut".

As we can see, simpler methods like "round-robin" suffer from higher performance losses because they are not aware of the dependencies between instructions, so frequent and unnecessary inter-cluster communications occur, as dependent instructions enter different clusters. While "loadcut" performs significantly better than "round-robin", it doesn't outperform "dependence-load-steering". This is expected because, although loads often create dependency chains, other instructions –such as ALU operations or branches– can also introduce dependencies. On the other hand, "dependence-load-steering" considers all kinds of dependencies, and tries grouping dependent instructions as much as possible to reduce inter-cluster delays. Also, '"loadcut" does not even guarantee that all dependent instructions stay within the same cluster. If an instruction depends on another one before the load that triggered the cluster switch, the communication overhead will persist.

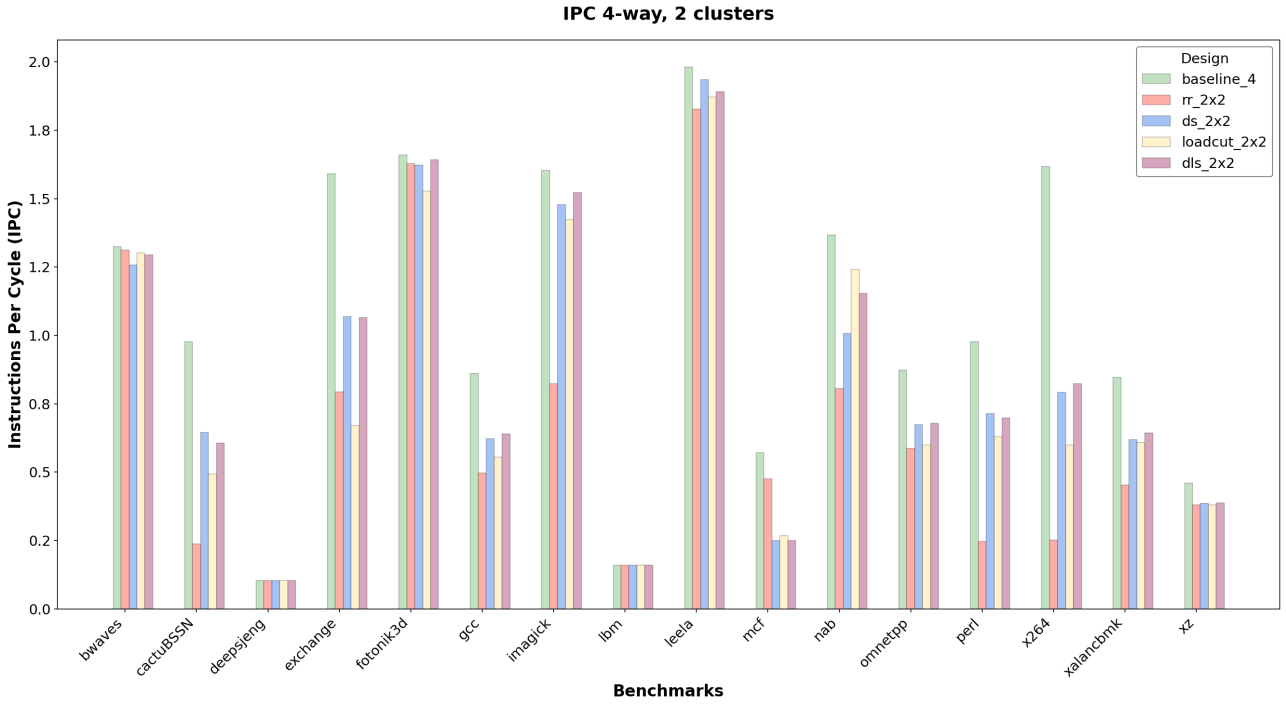

Figure 7.1: Performance comparison for the 2x2-way clustered microarchitecture.

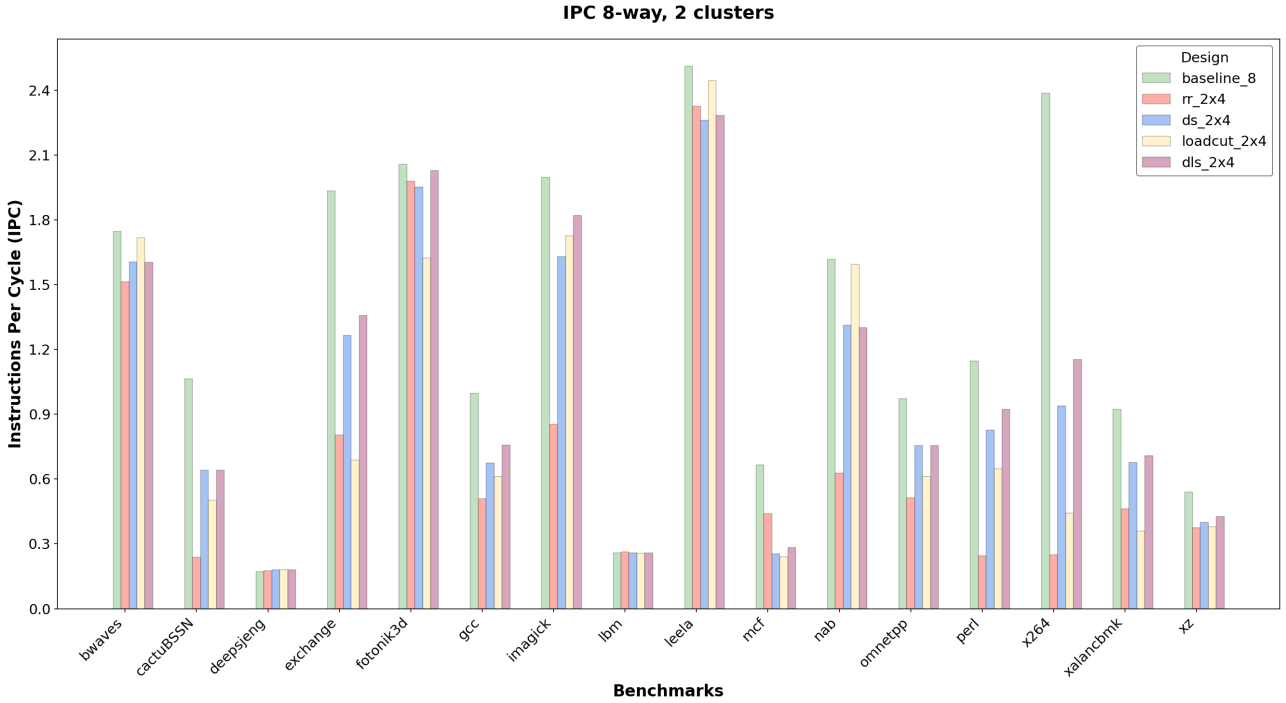

Figure 7.2: Performance comparison for the 2x4-way clustered microarchitecture.

7.2 Performance comparison of two and four cluster microarchitectures for the best steering

For the best performing steering method ("dependency-load-steering") we then proceed to compare the performance (in terms of IPC) between two and four cluster designs. The smaller structures used in the cases of four clusters (like the instruction queues of each cluster), typically result in shorter wire delays and potentially higher clock speeds, which can help mitigate some of the performance losses caused by inter-cluster communication.

For the 4-cluster designs we follow the same methodology, which is splitting once more the instruction queues and the functional units. We will name the 4-cluster models 4x1-way and 4x2-way, because now we have four 1-way clusters and four 2-way clusters respectively.

As we showed before, the best so far steering method was "dependency-load-steering". Figures 4 and 5 examine performance degradation in terms of IPC for two and four clusters for both 4-way and 8-way models. We observe a further reduction of the IPC, which we expected, because more clusters lead to smaller structures, like the instruction queues, that fill-up faster, and operands end up in different clusters than their dependent instructions more frequently. In addition, as we increase the number of clusters in the system, the probability that an instruction's operands are located in a different cluster than the instruction that needs them also increases. As a result, more inter-cluster delays occur, hence the IPC degradation. We calculate a 32% reduction of the IPC for the 4x1-way compared to the baseline 4-way model, and 36% reduction for the 4x2-way compared to the baseline 8-way model.

However, it is important to note that while IPC is reduced, the smaller cluster structures are likely to achieve higher clock speeds due to shorter wire delays and reduced critical path lengths. This increase in clock speed could help balance the IPC loss, with a possibility of better overall performance for our design.

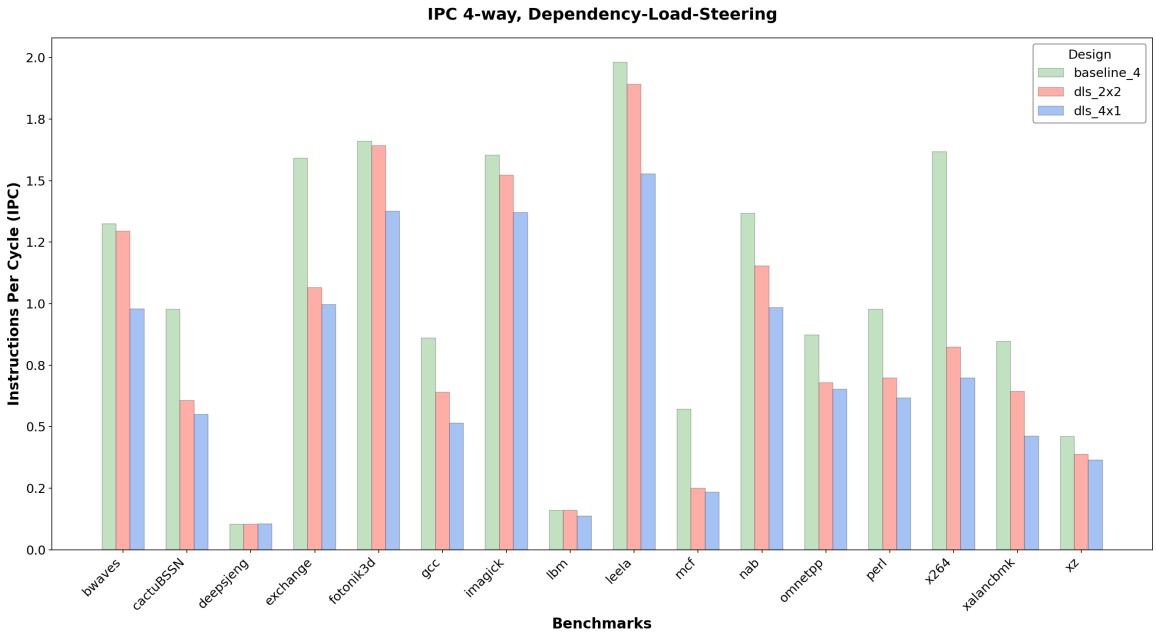

Figure 7.3: Performance (IPC) of "dependency-load-steering" for two and four cluster microarchitectures (4-way).

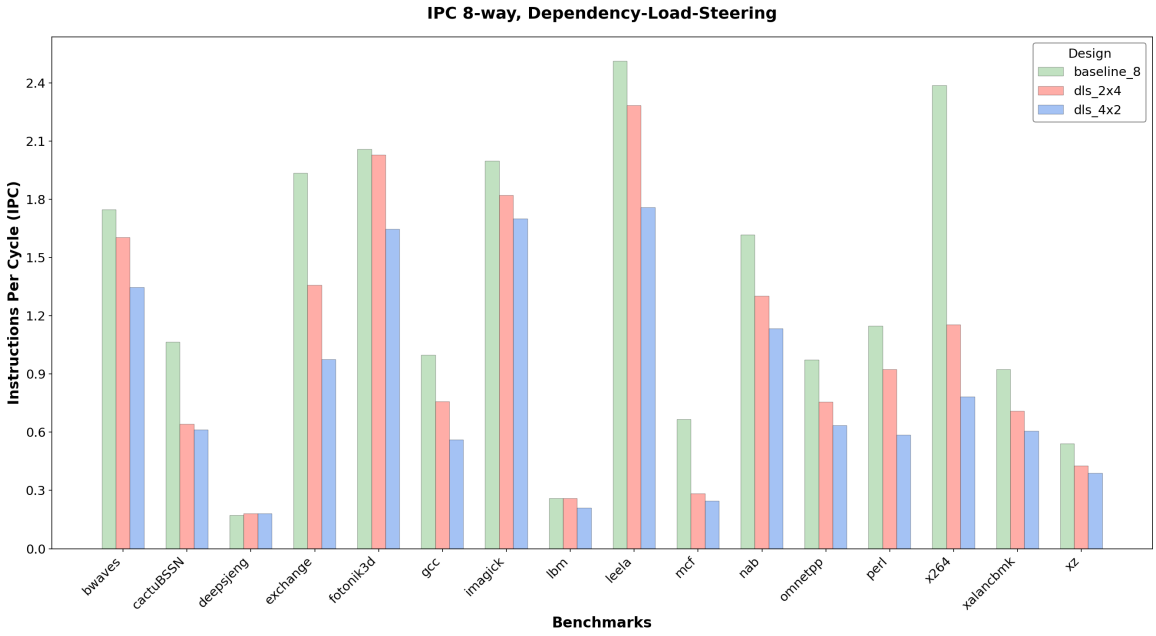

Figure 7.4: Performance (IPC) of "dependency-load-steering" for two and four cluster microarchitectures (8-way).

7.3 Performance comparison of two and four cluster microarchitectures for the best steering and two-cycle inter-cluster delay

Figures 7.5 and 7.6 depict the exact same designs as before, but with a two-cycle inter-cluster delay. In our methodology, we monitor during simulation whether the dependent instruction, which is awaiting the operand, gets "squashed" —meaning invalidated before execution— following the 2-cycle inter-cluster delay.

As expected, this leads to an even higher IPC degradation of approximately 55-59% for the two and four cluster design for an issue width of 4, and 59-64% for an issue width of 8, compared to the corresponding baseline model.

Benchmarks like cactusBSSN, gcc, Perlbench, nab have a dramatic decrease in IPC. A possible explanation for the observed IPC decrease in those benchmarks is their frequent need for operand bypassing. These applications likely rely strongly on continuous data processing for computations and data handling, making them more susceptible to the added delay of the two-cycle inter-cluster communication, compared to other benchmarks' IPC that is less affected by this additional delay.[18]

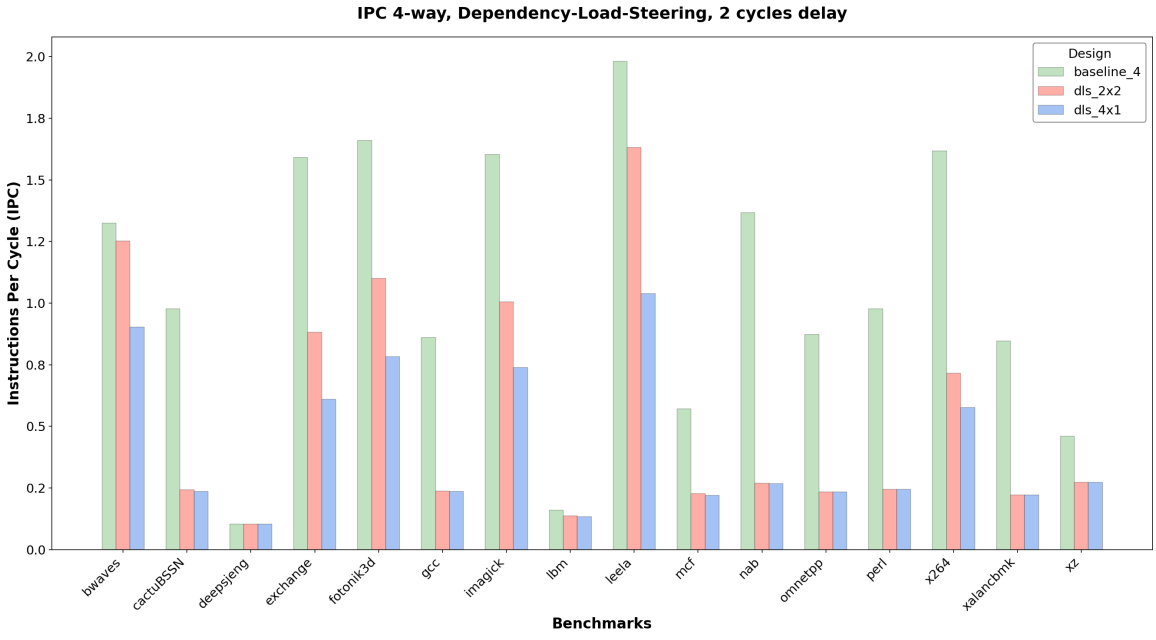

Figure 7.5: Performance (IPC) of "dependency-load-steering" for two and four cluster microarchitectures (4-way) and two cycles of inter-cluster delay.

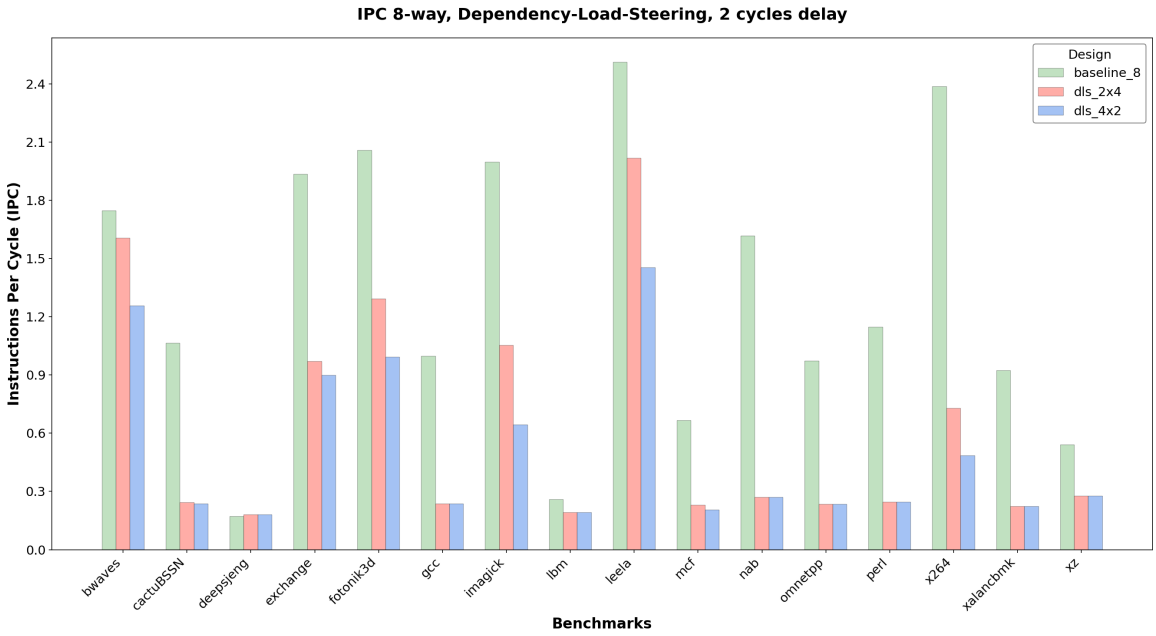

Figure 7.6: Performance (IPC) of "dependency-load-steering" for two and four cluster microarchitectures (8-way) and two cycles of inter-cluster delay.

These results demonstrate that our steering method is highly sensitive to inter-cluster communication delays, meaning that even small increases in delay –from one to two cycles– can lead to significant performance degradation, hence IPC decrease. This happens because instructions spend longer waiting in pipeline structures like the reorder buffer (ROB) and issue queue for operands from other clusters, slowing down overall throughput. As a result, the processor wastes more cycles waiting rather than executing new instructions. The longer delay also increases the chance that more speculative instructions will be squashed, further reducing performance. Since speculative instructions depend on getting results from earlier instructions, any extra delay keeps the execution path uncertain for longer, causing more instructions to be thrown away due to mispredictions.

The effectiveness of this steering method depends strongly on minimizing communication delays because its ability to exploit instruction-level parallelism is limited by how slow dependent instructions can access their source operands. As a result, having as few as possible inter-cluster bypasses is crucial.

So, the introduction of a two-cycle inter-cluster delay induces stalls and possibly more instruction squashing, indicating that a more efficient steering mechanism, potentially including dynamic cluster assignment –choice of cluster at runtime based on various of factors such as current dependencies, current load of each cluster or estimated communication cost– could help mitigate performance loss when we have higher inter-cluster delays, but this is beyond our work.

7.4 Inter-cluster bypass related stats

To better analyze the negative impact of inter-cluster delays on instruction execution, we introduced eight distinct statistical categories to capture different operand distribution scenarios. These categories count the number of source operands an instruction has and how many of those operands are located in a different cluster from the instruction, leading to an inter-cluster bypass. For example, for a 4-cluster design "Category 2" means that the current instruction, that has two source operands, is residing in cluster 0 but its operands are located in a different cluster, like cluster 1 and cluster 2 respectively. By tracking these cases, we can better understand how data dependencies across clusters affect performance.

More specifically:

- Category 0: 1 operand, 0 in other cluster
- Category 1: 1 operand, 1 in other cluster
- Category 2: 2 operands, 0 in other cluster
- Category 3: 2 operands, 1 in other cluster
- Category 4: 2 operands, 2 in other cluster
- Category 5: 3 operands, 0 in other cluster
- Category 6: 3 operands, 1 in other cluster
- Category 7: 3 operands, 2 in other cluster
- Category 8: 3 operands, 3 in other cluster

Figures 7.7 and 7.8 present the average percentage of each statistical category for each steering method. To produce Figures 8 and 9, we first collected statistics for each benchmark, categorizing every instruction based on the number of its source operands and how many of them originated from the opposite cluster (Categories 0–8). We make sure that when an operand is counted as residing in a different cluster, the bypass truly happens, and the desired value is not retrieved by the register file because the parent instruction is committed. So we keep track if a register still holds a value or if it is added back to the list of free registers to be reused by new instructions –which would mean that the parent instruction is committed. For each steering method, we then computed the average percentage of instructions falling into each category across all benchmarks, by summing the counts per category and dividing by the total number of instructions, to normalize the data and ensure a fair comparison

across benchmarks. This approach highlights how different steering strategies affect operand distribution and inter-cluster communication.

Most frequent categories related to inter-cluster bypass are categories 1, 3, 4. Categories 0, 2, 5 are good outcomes in our designs, because it means that no inter-cluster bypass occurred. At first sight we can see that those last categories are barely noticed in the case of "round-robin" and this proves the worse overall performance. We observe the appearance of category 1 for the steering methods "round-robin" and "loadcut", which makes sense since the other dependence-related methods work really well with one operand instructions, because the dependence is properly handled. In the few cases where we observe this behavior in the dependence-related methods too, it is likely due to the issue queues of the intended cluster being full, so the instruction is sent elsewhere. We can also see category 3 appearing at the dependence-related steering methods, as well as "round-robin" and "loadcut". This was also expected since when it comes to the dependence-related methods and an instruction has two operands, it is possible that the instruction is steered to the cluster of the first one, while the other one resides in a different cluster. This is a major limitation of this method, that it doesn't guarantee that the parent instructions will end up in the same cluster. To avoid this, a "slice" related method could be implemented, as previous works suggested. In this approach, instructions that are related through data would be grouped together and sent to the same cluster, mitigating as much as possible the appearance of such cases.

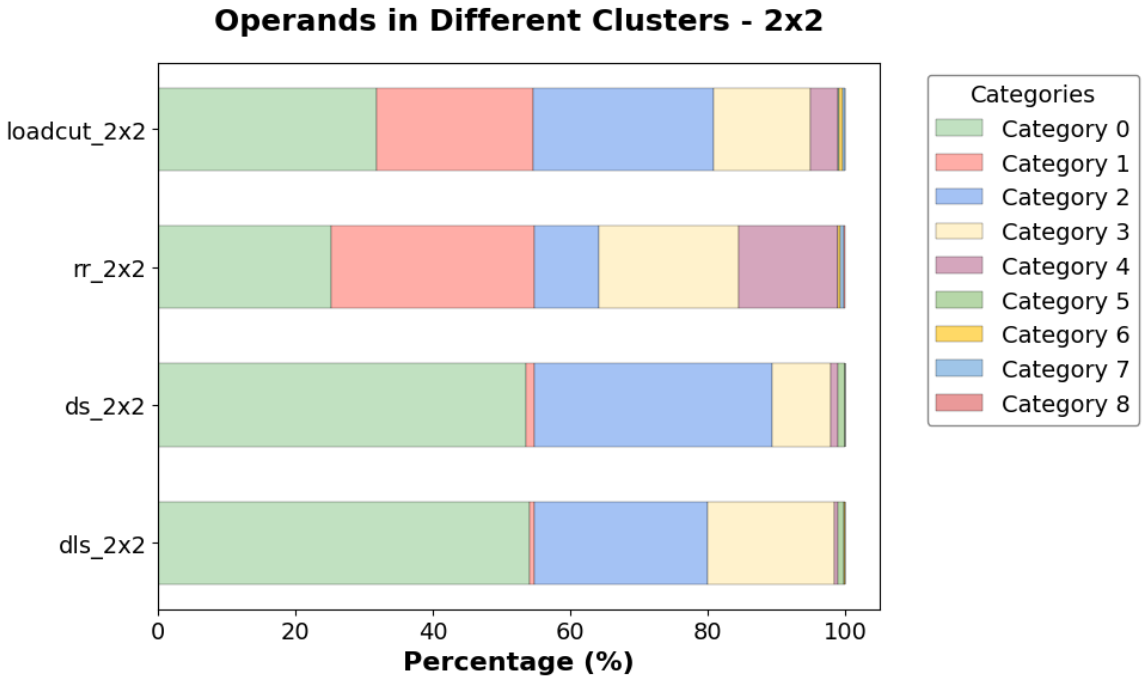

Figure 7.7: Average of each of the statistical categories specified, for 2-cluster designs (4-way).

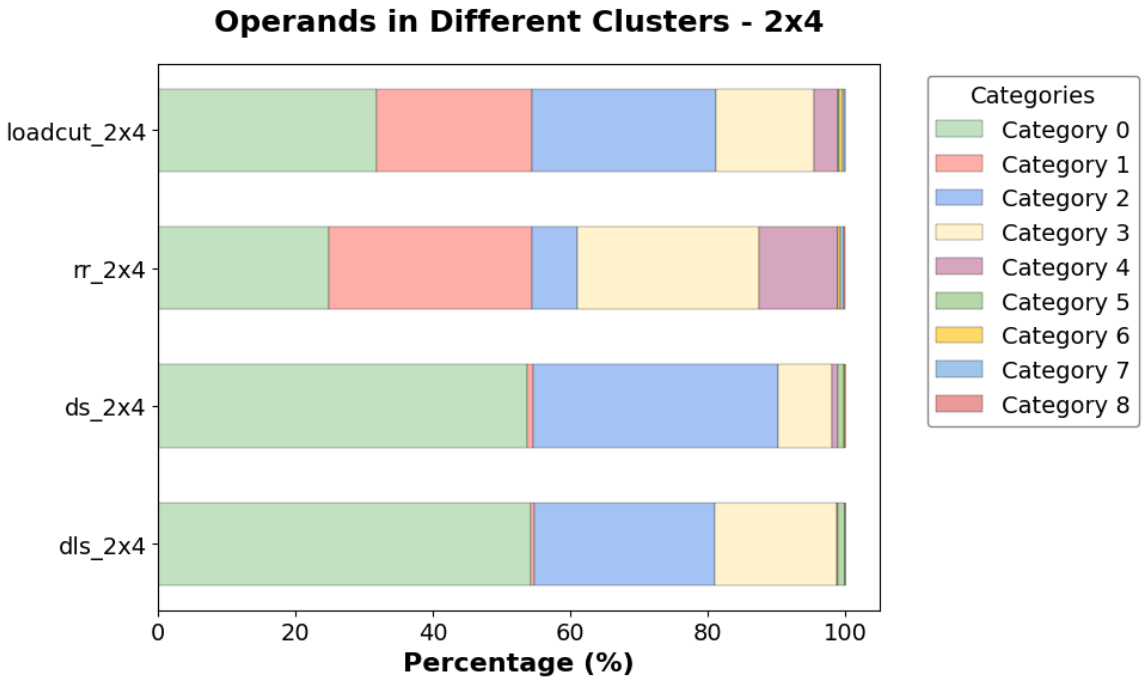

Figure 7.8: Average of each of the statistical categories specified, for 2-cluster designs (8-way).

7.5 Resource related stalls

To further analyze performance bottlenecks, we used gem5's four statistical categories mentioned below, to capture different types of resource-related stalls. These stalls occur when key hardware resources are unavailable, delaying instruction execution. By tracking these cases separately, we can better understand which hardware limitations we introduced due to clustering. So we specify the four different categories:

- Category 0: Busy functional unit
- Category 1: Full instruction queue
- Category 2: Full register file
- Category 3: Full load-store queue

During the simulations, we utilized gem5's built-in statistics related to resource stalls, which capture events where instructions are delayed due to limited hardware resources such as functional units, issue queue entries, or load/store buffers. For example, "full instruction queue" stall for the clustered designs is a counter that examines how many times both the instruction queues are full, because as we already mentioned before, when an instruction is to be sent to a cluster but its instruction queue is full, it is then steered to another available one. In this case, if there is an available one, no stalls occur. But if there isn't, we do have a resource-induced delay.

As we can see in Figures 7.9 and 7.10 for a 4-way and 8-way design respectively, what is worth mentioning is the fact that we have more "busy functional unit" type stalls, compared to the baseline model. This happens because when an instruction is assigned to a cluster, only a subset of the total functional units are available for use, which is half of the functional units of the baseline model.

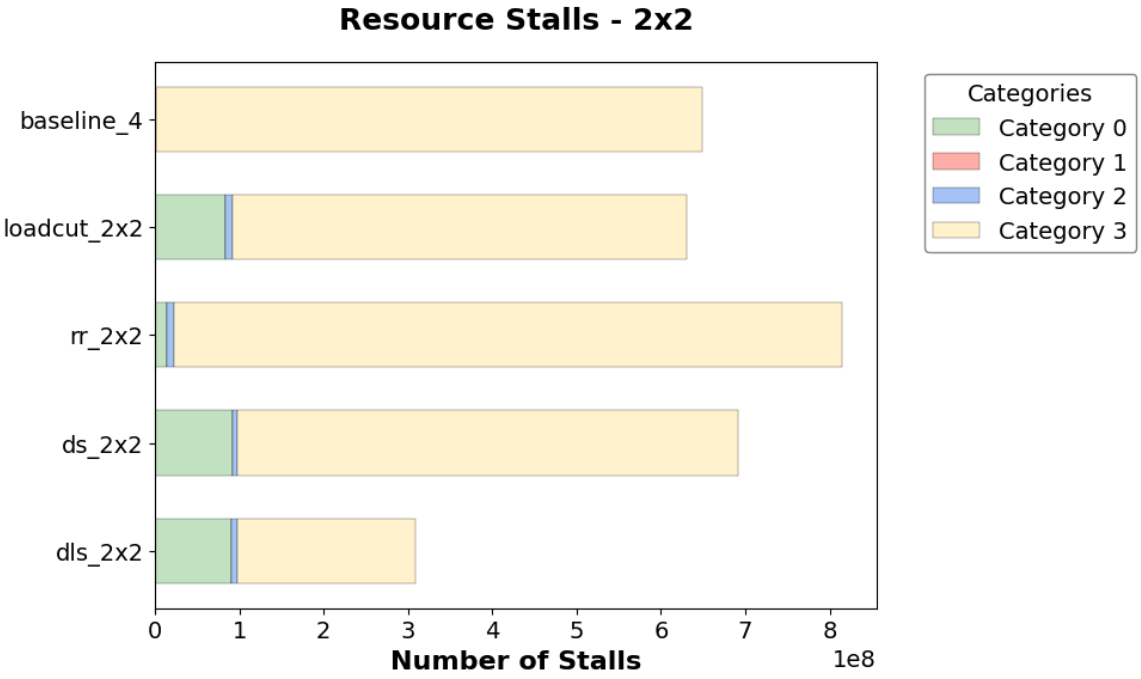

Figure 7.9: Average of each of the statistical categories specified, for 2-cluster designs (4-way).

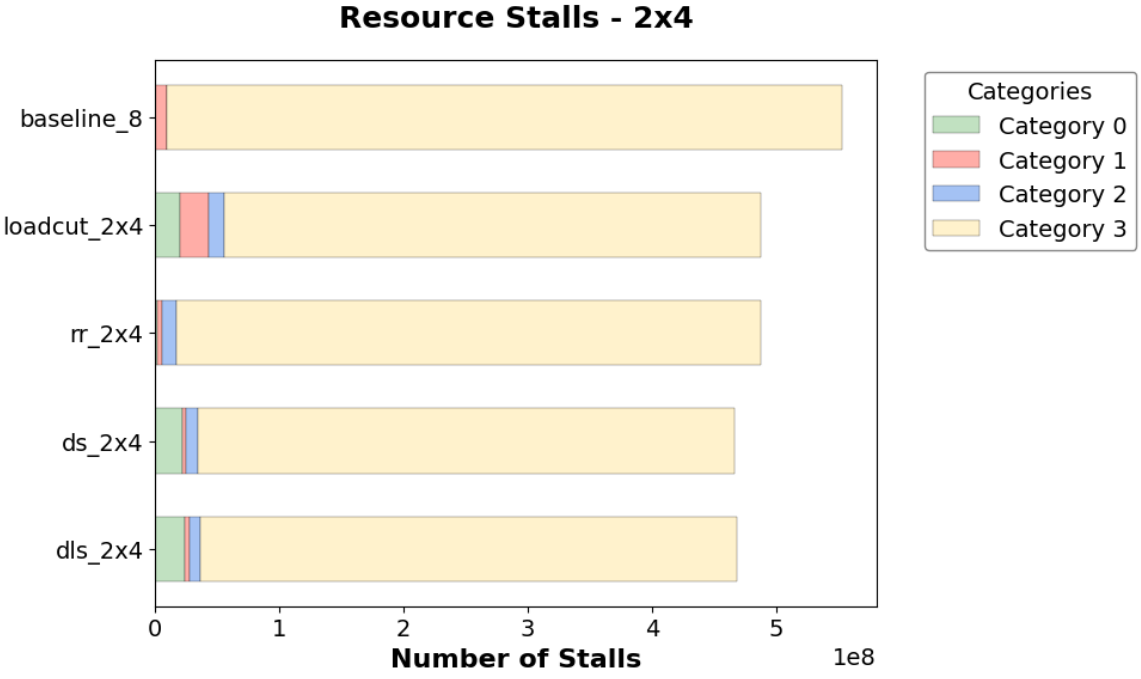

Figure 7.10: Average of each of the statistical categories specified, for 2-cluster designs (8-way).

Chapter 8

Wider Cores

Over the past decade, CPU cores have grown significantly wider, both in terms of the number of instructions they can decode per cycle and the number they can execute simultaneously. A decade ago, high-performance CPUs typically decoded up to four instructions and executed up to eight in parallel. In contrast, the latest processors have nearly doubled these capabilities. For instance, Apple's M4 processor, released in 2024, can decode up to 10 instructions and execute as many as 19 simultaneously. Similarly, ARM's Cortex-X925, also launched in 2024, achieves a decode width of 10 and an execution width of 23. These processors are also capable of massively scheduling , between about 700 and 900 instructions out-of-order, pushing the boundaries of what modern microarchitectures can manage.[19]

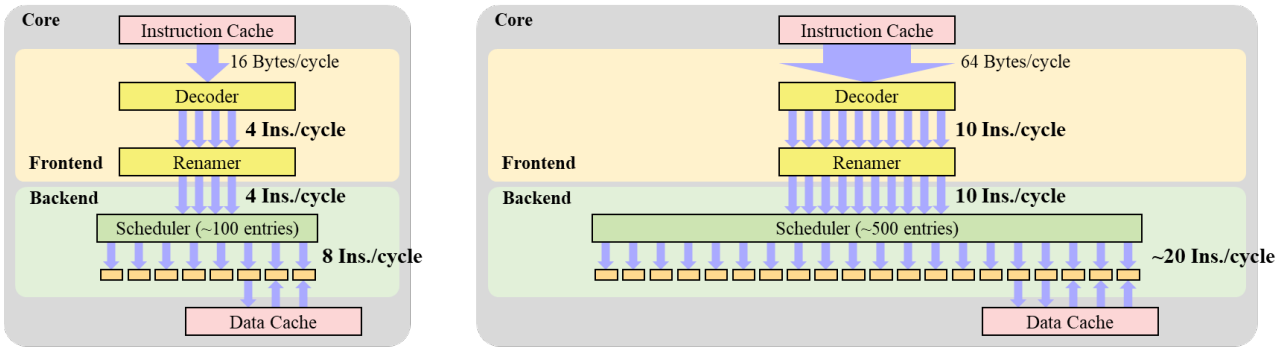

Figure 8.1: Widest CPUs.[2]

In this chapter, we focus on the implications and design challenges of greater issue widths. As CPUs continue to scale toward 16-way and beyond, we examine how such widths can be supported effectively through clustered microarchitectures and the steering mechanisms analyzed before.

Table 8.1 describes the baseline 16-way core we used for our simulations. For the clustered designs, similarly to before, we split the instruction queues into two or four, one for each cluster, as well as the functional units. Compared to our previous cores, the 16-way issue design is scaled up to explore the behavior

of extremely wide-issue architectures. Although the design parameters are not necessarily reflective of commercial products, they are an applicable estimation for evaluating future high-performance CPU trends. Following the same methodology as before, we doubled the core resources relative to the 8-way issue design. Specifically, the instruction queue (IQ) is expanded to 160 entries, while the load and store queues (LQ/SQ) are increased to 64 and 96 entries respectively. The reorder buffer (ROB) is scaled to 1024 entries, allowing the processor to manage a very deep out-of-order execution window. The number of functional units is also doubled to accommodate the larger issue width and to ensure that the increased bandwidth is not affected by limited resources. Similar to designs like Apple's M4 and ARM's Cortex-X925, as discussed by Koizumi et al., rename and decode stages often become critical bottlenecks in wide-issue architectures. To address this, our design maintains a rename and decode width of 10, while allowing the execution width to scale up to 16.

In contrast to the earlier cores, the memory hierarchy in the 16-way design is also enhanced, as we can see in Table 8.1, to keep pace with the higher demand for data bandwidth and to minimize stalling due to memory latency. The L1 instruction and data caches are each increased to 64 KB and made 4-way set-associative. The L2 cache is 4 MB with an associativity of 32-way but in addition with an L3 cache of 16 MB, shared across clusters, to reduce pressure on main memory and accommodate larger working sets. The increased cache capacity and associativity aim to reduce memory access latency and reduce the likelihood of performance bottlenecks in memory-bound workloads.

Baseline 16-way Configuration	
Component	**Specification**
IQ	160
LQ/SQ	64/96
ROB	1024
INT/FP	560/664
L1-D	64 KB, 4-way
L1-I	64 KB, 4-way
L2	2 MB, 16-way
L3	16 MB, 16-way

Table 8.1: Baseline 16-way configuration.

8.1 Performance of clustered 16-way designs

We will start by running the same SPEC2017 benchmarks, and more specifically the SPECspeed2017 Integer and Floating point suites as before for 1 billion instructions with a 5 million warm-up. Figures 8.2 and 8.3 show the performance results in terms of IPC for the 2x8-way –that is two 8-way clusters that approximate a 16-way centralized microarchitecture– and for the 4x4-way –that is four 4-way clusters that also approximate a 16-way centralized microarchitecture. In these simulations, we introduce both 1 and 2-cycle inter-cluster delays respectively to model the latency of forwarding operands between instructions residing in different clusters.

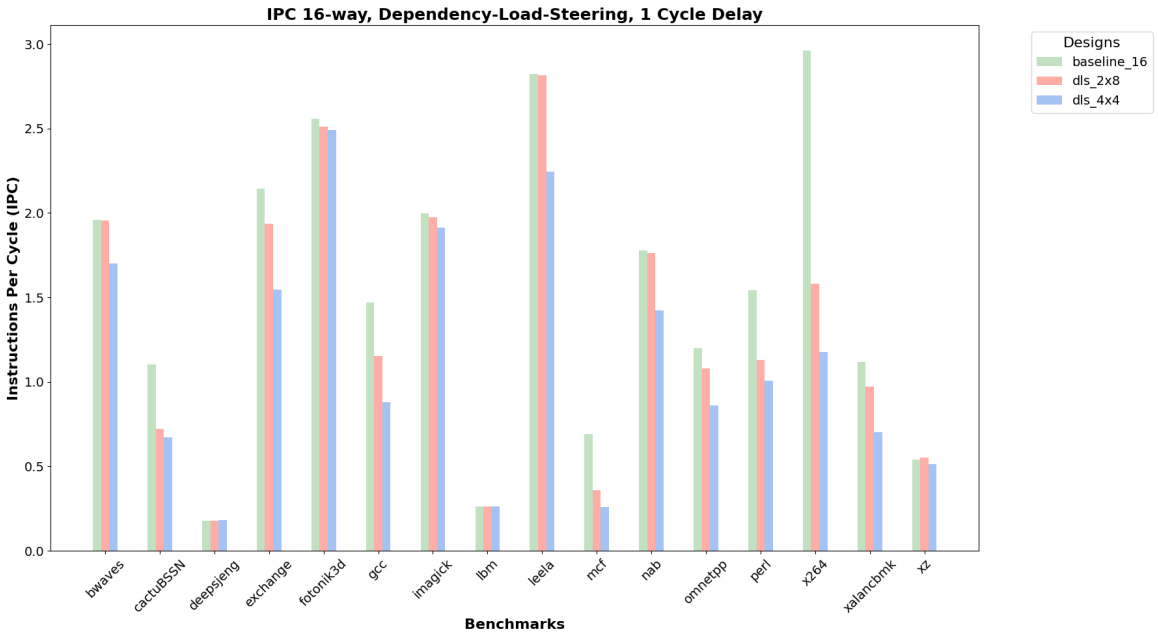

Figure 8.2: Performance (IPC) of "dependency-load-steering" for two and four cluster microarchitectures (16-way) and one cycle of inter-cluster delay.

In Figure 8.2 we observe a performance degradation in terms of IPC across both clustered designs due to the additional one cycle inter-cluster communication delay and we observe an IPC reduction of approximately 15% and 27% for the 2x8 and 4x4 configurations respectively. Also for fourteen out of the sixteen benchmarks, IPC is reduced by 9% and 20% for the 2x8 and 4x4 configurations respectively, except from "x264" and "mcf" that a dramatic degradation over 40% is observed.

In Figure 8.3 we observe a performance degradation in terms of IPC across both clustered designs due to the additional two-cycle inter-cluster communication delay, consistent with the expected impact of increased operand latency. More specifically, when comparing the clustered configurations to their respective baselines we calculate an average IPC degradation of approximately 57% for the 2×8 and 60% for the 4×4 configurations.

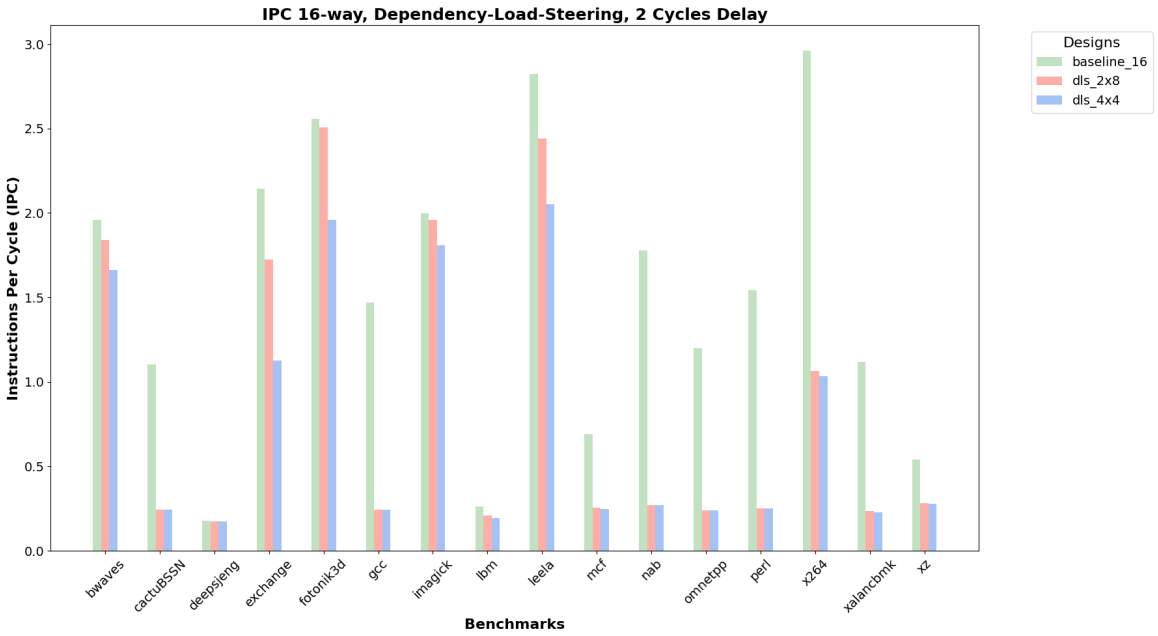

Figure 8.3: Performance (IPC) of "dependency-load-steering" for two and four cluster microarchitectures (16-way) and two cycles of inter-cluster delay.

8.2 Observations

Interestingly, the IPC degradations observed in the clustered 2×8 and 4×4 configurations are somewhat more moderate than those seen in the corresponding 4-way and 8-way clustered designs, which suffered more severe performance declines, especially when we compare IPC degradations with a 1-cycle inter-cluster delay.

This more moderate degradation in the wider clustered configurations can be largely attributed to their greater issue width, which provides a broader window for instruction scheduling and execution. A wider issue width effectively enhances the processor's ability to identify and dispatch independent instructions, even when certain instructions are temporarily stalled waiting for operands due to inter-cluster data transfers. As a result, functional units are less likely to just wait instead of executing ready instructions.

Moreover, the increased instruction throughput in wider designs means that while pipeline resources such as issue queue and reorder buffer (ROB) are also enlarged, the likelihood that alternative independent instructions are available for execution when dependent instructions are stalled, is increased. This helps to smooth out the stalls caused by inter-cluster communication delays, leading to relatively smaller drops in IPC compared to narrower configurations.

Nevertheless, despite the ability of wider clustered architectures to better tolerate communication delays thanks to their larger issue widths and greater instruction-level parallelism, the performance degradation remains a key concern, especially as the inter-cluster delay grows from one to two cycles. The

impact of such delays becomes detrimental in workloads with frequent data dependencies.

These findings highlight the critical role of inter-cluster communication to the overall performance, as well as the importance of designing efficient communication mechanisms.

Chapter 9

Conclusion and Future Work

9.1 Summary

In this thesis, we analyzed the critical sources of delay in wide-issue processors, focusing specifically on the challenges introduced by issue logic and data bypass logic. These components are expected to become even more significant bottlenecks in the future as issue widths and issue windows continue to scale. To address these challenges, we introduced clustering as a solution, aiming to maintain an effective issue width like that of a wide monolithic processor, while benefiting from smaller, simpler structures that help preserve high clock frequencies.

Several clustered designs were implemented with varying issue widths, different number of clusters, and multiple distribution (steering) methods of the instructions and their performance was evaluated in terms of Instructions per Cycle (IPC). We found that methods like Round-Robin, that alternate instructions between clusters, and Loadcut, that switch clusters every time a load instruction occurs, suffer greatly from inter-cluster bypass delays –that is, bypasses between different clusters that take longer to complete.

Our best performing method was Dependency-Load-Steering, that takes into consideration the dependencies between instructions and the relative load of the clusters. By aiming to group together dependent instructions the bypasses were relatively less compared to other techniques. The IPC was decreased compared to the corresponding baselines with the same issue widths (for all 4-way, 8-way and 16-way designs).

Finally, when considering the potential clock frequency advantages of clustering, our dependency-based clustered microarchitectures demonstrate the potential to outperform the wide monolithic designs in overall performance.

9.2 Future Work

For a clustered microarchitecture, managing inter-cluster communication is the major challenge. By implementing various steering methods, each based on a different criterion, we found that while the ones that track dependencies between instructions work better than other simpler ones like Round-Robin, or methods like Loadcut, there is still a place for further investigation of how to reduce communication latencies. In particular, dynamic steering techniques proposed in previous studies -where instructions are not statically steered based only on simple dependency checks, but the cluster is decided at runtime based on factors like current dependency information, current load of each cluster, and other heuristics- could be further optimized by developing more accurate prediction heuristics for instruction dependencies. Such improvements could help to minimize the performance penalties associated with inter-cluster communication, but would probably add to the steering method's complexity due to the real-time decision-making mechanisms that will be used. In order for those methods to be accurate, additional hardware structures like predictors, load trackers, or even advanced bypass networks will be needed. Future work could explore lightweight, low-overhead hardware, to ensure a total performance gain.

Another promising direction for future exploration is the configuration of the clusters. In our work, we consider homogeneous clusters, with the exact same functional unit configurations. More specifically, we use two or four identical clusters across all designs and issue widths. However, other configurations could be explored, possibly with heterogeneous clusters, where each cluster is specialized with a different mix of functional units specialized for particular types of instructions (e.g., clusters optimized for integer, floating-point, or memory-intensive operations). Such specialization could potentially improve performance by reducing inter-cluster delays and improving resource utilization. Additionally, varying the size, number, or specialization of clusters dynamically at runtime based on workload characteristics could be another promising research direction.

Finally, the impact of wire delays between clusters could be also optimized. As technology scales down and designs become faster, wire delays dominate overall communication costs. Future work could investigate techniques such as placing clusters physically closer together, especially those that frequently communicate, and also using faster interconnect technologies.

Bibliography

[1] A. Mahmoud, Y. Ismail, and M. Dessouky, "Riscv processor block diagram," 2021, accessed: May 6, 2025. [Online]. Available: https://www.researchgate.net/figure/RISCV-Processor-Block-Diagram-17_fig1_348273666

[2] S. Sadowski and M. Lipasti, "Distance-based isa for efficient register management," https://www.sigarch.org/distance-based-isa-for-efficient-register-management/, 2023.

[3] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 302–313. [Online]. Available: https://doi.org/10.1145/1815961.1816000

[4] P. Michaud, A. Mondelli, and A. Seznec, "Revisiting clustered microarchitecture for future superscalar cores: A case for wide issue clusters," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 3, p. 22, 2015.

[5] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, 1997, pp. 206–218.

[6] A. Baniasadi and A. Moshovos, "Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-33)*. IEEE, 2000, pp. 337–347.

[7] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, "Dynamic prediction of critical path instructions," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2001, pp. 185–195.

[8] P. Salverda and C. Zilles, "A criticality analysis of clustering in superscalar processors," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, Barcelona, Spain, 2005, pp. 12–66.

[9] N. Ranganathan and M. Franklin, "The pews microarchitecture: Reducing complexity through data-dependence based decentralization," *Microprocessors and Microsystems*, vol. 22, no. 6, pp. 333–343, 1998.

[10] A. Aggarwal and M. Franklin, "Instruction replication for reducing delays due to inter-pe communication latency," *IEEE Transactions on Computers*, vol. 54, no. 12, pp. 1496–1507, 2005.

[11] R. Canal, J.-M. Parcerisa, and A. González, "Dynamic cluster assignment mechanisms," in *Proceedings Sixth International Symposium on High-Performance Computer Architecture (HPCA-6)*. IEEE, 2000, pp. 133–142.

[12] The gem5 Team, "The gem5 simulator," 2024. [Online]. Available: https://www.gem5.org/

[13] RISC-V International, "Risc-v: The free and open isa," 2024. [Online]. Available: https://riscv.org/

[14] F. A. Endo, D. Couroussé, and H.-P. Charles, "Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5," in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, Agios Konstantinos, Greece, 2014, pp. 266–273.

[15] Arm Ltd., "Arm cortex-a76 processor," 2025. [Online]. Available: https://developer.arm.com/Processors/Cortex-A76

[16] Intel Corporation, "Intel core processors," 2025. [Online]. Available: https://www.intel.com/content/www/us/en/ark/products/series/122139/intel-core-processors.html

[17] "Overview of spec cpu2017 benchmarks," SPEC, accessed from https://www.spec.org/cpu2017/Docs/overview.html#benchmarks.

[18] S. Singh and M. Awasthi, "Memory centric characterization and analysis of spec cpu2017 suite," 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1910.00651

[19] R. Matsuo, T. Koizumi, H. Irie, S. Sakai, and R. Shioya, "Enhancing gpu performance through complexity-effective out-of-order execution using distance-based isa," *IEICE Transactions on Information and Systems*, 2024.