NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

DIVISION OF COMPUTER SCIENCE
COMPUTER SYSTEMS LAB

Application classification techniques' design for interference mitigation in multiprocessor systems

DIPLOMA THESIS

Marina Vemmou

Supervisor: Georgios Goumas
Assistant Professor, N.T.U.A.

Marina Vemmou

Athens, July 2019

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE
COMPUTER SYSTEMS LAB

Application classification techniques' design for interference mitigation in multiprocessor systems

DIPLOMA THESIS

Marina Vemmou

Supervisor: Georgios Goumas
Assistant Professor, N.T.U.A.

Approved by the examining committee on July 19th, 2019.

.....	.....	.....
G.Goumas	N.kozyris	N.Papaspyrou
Assistant Professor, N.T.U.A.	Professor, N.T.U.A.	Associate Professor, N.T.U.A.

Athens, July 2019.

.....

Marina Vemmou
Electrical and Computer Engineer

Περίληψη

Οι πολυπύρηνοι επεξεργαστές είναι ο βασικός δομικός λίθος όλων των σύγχρονων υπολογιστικών συστημάτων. Παρ' όλα τα οφέλη που παρέχει η δυνατότητα ταυτόχρονης εκτέλεσης εφαρμογών, ο ανταγωνισμός που προκαλείται για κοινόχρηστους πόρους του πολυεπεξεργαστή όπως η κρυφή μνήμη τελευταίου επιπέδου (Last Level Cache) και το εύρος του διαύλου δεδομένων προς τη μνήμη είναι πολλές φορές καταστροφικός για την επίδοση των εφαρμογών. Ειδικότερα, σε περιβάλλοντα υπολογιστικού νέφους (Cloud Environments), ο πάροχος καλείται να εξασφαλίσει συγκεκριμένα και αυστηρά επίπεδα επίδοσης (Quality of Service goals) για συγκεκριμένες εφαρμογές, οδηγώντας στην ανα-γκαστική εκτέλεση των τελευταίων σε απομονωμένα περιβάλλοντα για την αποφυγή αντα-γωνισμού και την τελική υποχρησιμοποίηση του συστήματος.

Για το λόγο αυτό, το πρόβλημα του ανταγωνισμού για κοινόχρηστους πόρους σε σενάρια συνεκτελέσεων έχει απασχολήσει εκτενώς την επιστημονική κοινότητα. Η παρούσα εργασία επικεντρώνεται στην πρόβλεψη των καταστάσεων ανταγωνισμού αξιοποιώ-ντας αποκλειστικά δεδομένα από μετρικές επίδοσης υλικού (hardware performance counters) κατά την απομονωμένη εκτέλεση των εφαρμογών. Βασικό χαρακτηριστικό της πρόσεγγίσης μας είναι το ότι δεν απαιτεί τη συνεκτέλεση μιας εφαρμογής με άλλες ώστε να εντοπιστεί το εάν επηρεάζει την επίδοσή τους ή επηρεάζεται η ίδια, καθιστώντας την ιδανική για κέντρα δεδομένων όπου δεν υπάρχει η πολυτέλεια εσκεμμένης πρόκλησης καταστάσεων ανταγωνισμού.

Ο τελικός μας μηχανισμός περιλαμβάνει δύο ταξινομητές βασισμένους σε τεχνικές μηχανικής μάθησης. Ο κάθε ταξινομητής λαμβάνει ως είσοδο ένα συγκεκριμένο σύνολο μετρικών επίδοσης και κατηγοριοποιεί την εφαρμογή ως προς την ικανότητά της να επηρεάζει την εκτέλεση άλλων εφαρμογών (noise) και την ευαισθησία της επίδοσής της όταν συνεκτελείται με άλλους (sensitivity). Υποδεικνύουμε επίσης πώς μπορούν οι χαρακτηρισμοί που αποδίδουμε στις εφαρμογές μπορούν να αξιοποιηθούν από έναν χρονοδρομολογητή εφαρμογών (application scheduler) σε ένα περιβάλλον υπολογιστικού νέφους ώστε να μεγιστοποιηθεί η επίδοση εφαρμογών υψηλής προτεραιότητας.

Λέξεις-Κλειδιά: ανταγωνισμός για κοινόχρηστους πόρους επεξεργαστή, χαρακτηρισμός εφαρμογών, πρόβλεψη συμπεριφοράς, μετρικές επίδοσης υλικού, μηχανική μάθηση

Abstract

Multiprocessors are the basic building block of all modern computing systems. Despite the benefits yielded by the ability to execute applications concurrently, the rivalry between applications for the chip's shared resources, such the Last Level Cache and the memory bandwidth, can be detrimental to performance. Especially in commercial cloud environments, the provider is obliged to abide by strict performance guarantees required by certain applications (Quality of Service goals), leading to the isolated execution of the latter in dedicated servers to avoid interference, and consequently to the system's under-utilization.

As a result, extensive research has been conducted on the problem of application interference. This diploma thesis focuses on predicting cases where interference might be present by utilizing exclusively data by low-level hardware performance counters gathered during isolated application execution. The main characteristic of our approach is that it does not require executing an application with co-runners to decide whether it will suffer from or create contention, making it ideal for cloud environments, where subjecting an application to artificial interference is prohibitive.

Our final mechanisms consists of two machine learning base multiclass classifiers. Each classifier receives a s input a specific set of hardware performance counter values and classifies the application in regards to its ability to cause interference (noise) and its sensitivity to it. We also showcase how the labels we have assigned each application can then be utilized by an application scheduler in a datacenter, in order to maximize the performance of high-priority applications.

Keywords: interference, processor shared resources, application profiling, application classification, hardware performance counters, machine learning

Ευχαριστίες

Η παρούσα εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου.

Θα ήθελα να ευχαριστήσω ιδιαιτέρως τον Επιβλέποντα Καθηγητή μου κ. Γεώργιο Γκούμα για την συνεχή καθοδήγηση και στήριξη που με προθυμία μου παρείχε κατά τη διάρκεια εκπόνησης της παρούσας διπλωματικής εργασίας. Επιπλέον, θα ήθελα να ευχαριστήσω τους καθηγητές κ. Ν.Κοζύρη και κ. Ν.Παπασπύρου, καθώς και τον μεταδιδακτορικό ερευνητή κ. Κ.Νίκα, για το ενδιαφέρον που μου καλλιέργησαν μέσα από τα μαθήματά τους για την επιστήμη και αρχιτεκτονική των υπολογιστών, καθώς και για τις πολύτιμες γνώσεις που μου προσέφεραν όλα αυτά τα χρόνια.

Ο κύκλος σπουδών μου και η παρούσα εργασία δεν θα είχαν ολοκληρωθεί δίχως την παρουσία της πιο όμορφα ετερόκλητης παρέας φίλων, μιας παρέας που ποτέ δε φανταζόμουν ότι θα μου άξιζε να αποκτήσω. Σας ευχαριστώ όλους για κάθε στιγμή που μου χαρίσατε. Ιδιαίτερα θα ήθελα να ευχαριστήσω τη Φωτεινή, για την συνεχή της υπομονή και στήριξη, και το Νίκο, για κάθε στιγμή που πίστεψε στους στόχους μου πιο πολύ απ' όσο εγώ.

Το μεγαλύτερο ευχαριστώ είναι για την οικογένειά μου, για όλη τη στήριξη και εμπιστοσύνη που συνεχίζει να μου προσφέρει.

Μαρίνα Βέμμου,
Ιούλιος 2019

Contents

List of Figures

Chapter 1

Εκτεταμένη Περίληψη

1.1 Το Πρόβλημα του Ανταγωνισμού σε Πολυεπεξεργαστικά Συστήματα

Οι πολυπύρηνοι επεξεργαστές, οι επεξεργαστές δηλαδή που σε μία μόνο πλακέτα (chip) περιλαμβάνουν περισσότερους από έναν πυρήνες (CPU cores), αποτελούν τη βασική πηγή υπολογιστικής δύναμης κάθε σύγχρονου υπολογιστικού συστήματος. Κάθε πυρήνας έχει μία ιδιωτική επιπέδου 1 (Level 1, L1) και επιπέδου 2 (Level 2, L2) κρυφή μνήμη (cache), ενώ οι υπόλοιποι πόροι του chip, όπως η κρυφή μνήμη τελευταίου επιπέδου (Last Level Cache, LLC), το εύρος του διαύλου προς τη μνήμη (DRAM Bandwidth), το δίκτυο διασύνδεσης και το εύρος ζώνης του δικτύου είναι κοινοί και διαμοιραζόμενοι ανάμεσα στους πυρήνες, και κατ' επέκταση ανάμεσα στις εφαρμογές που εκτελούνται σε αυτούς.

Παρόλο που η ταυτόχρονη συνεκτέλεση πολλών εφαρμογών αποτελεί το βασικό πλεονέκτημα ενός πολυεπεξεργαστή, ο ανταγωνισμός (interference) που δημιουργείται ανάμεσα στις εφαρμογές για τους κοινόχρηστους πόρους του chip οδηγεί στην σημαντικά μειωμένη επίδοση των τελευταίων. Στην παρούσα διπλωματική εξετάζεται ο ανταγωνισμός στους εξής δύο πόρους:

- LLC: Διαφορετικές εφαρμογές προσπελαύνουν διαφορετικές θέσεις μνήμης και δεδομένα. Εφαρμογές που πραγματοποιούν συχνές προσβάσεις στην LLC ή/και χρησιμοποιούν μεγάλο κομμάτι της μπορεί να εκτοπίσουν δεδομένα άλλων εφαρμογών, ή να υποφέρουν οι ίδιες από συχνό εκτοπισμό των δεδομένων τους. Ο συνεχόμενος ανταγωνισμός για χώρο στην LLC οδηγεί σε αυξημένα miss rates, τα οποία υποβαθμίζουν την επίδοση και αυξάνουν την κατανάλωση ενέργειας.

- DRAM Bandwidth: Εφαρμογές που πραγματοποιούν πολλές και συχνές προσβάσεις στην μνήμη συναγωνίζονται μεταξύ τους για το διαθέσιμο Memory Bandwidth, κυρίως ως αποτέλεσμα μοτίβου προσβάσεων που δεν επωφελείται από την ιεραρχία κρυφών μνημών.

Σε αυτή τη διπλωματική εργασία, θεωρούμε ότι οι εφαρμογές είναι μονονηματικές (ένα νήμα εκτέλεσης), και οι όροι "εφαρμογή" και "νήμα εκτέλεσης" χρησιμοποιούνται ως ταυτόσημοι

1.2 Προσεγγίσεις στο Πρόβλημα του Ανταγωνισμού

Εξαιτίας της συνεπειών στην επίδοση που έχει ο ανταγωνισμός για τους κοινόχρηστους πόρους, έχει προταθεί ένας αριθμός μηχανισμών και λύσεων για την αντιμετώπισή του. Γενικότερα, οι μηχανισμοί αυτοί στοχεύουν σε ένα ή περισσότερα από τα κάτω όσον αφορά το interference:

1. Πρόληψη
2. Εντοπισμός και διαχωρισμός του από φυσιολογικές αυξομοιώσεις στο φόρτο εργασίας (workload) των εφαρμογών ή την εναλλαγή φάσεων εκτέλεσης
3. Ελαχιστοποίησή του και των συνεπειών του

Η γενική ακολουθία γεγονότων όταν μία εφαρμογή υποβληθεί σε ένα σύστημα είναι η εξής:

1. Άφιξη εφαρμογής
2. (Προαιρετικό) Εκτός σύνδεσης δημιουργία προφίλ εφαρμογής (offline profiling)
3. Χρονοδρομολόγηση εφαρμογής (τοποθέτηση σε server/core) και λήψη αποφάσεων για το διαμοιρασμό πόρων
4. Έναρξη εκτέλεσης
5. (Προαιρετικό) Δημιουργία προφίλ εφαρμογής ταυτόχρονα με την εκτέλεσή της (profiling concurrent to the execution)
6. (Προαιρετικό) Παρακολούθηση εκτέλεσης εφαρμογής (online monitoring) και αναπροσαρμογή των αποφάσεων χρονοδρομολόγησης ή/και διαμοιρασμού πόρων

Τα βήματα 2,3,5,6 δίνουν στον σχεδιαστή του συστήματος ένα εύρος επιλογών, από το αν θα τις συμπεριλάβει ή όχι (αν είναι σημειωμένες ως προαιρετικές) ως τις παραμέτρους τις υλοποίησής τους (π.χ. τον αλγόριθμο χρονοδρομολόγησης). Μπορούμε να διαχωρίσουμε τις μέχρι τώρα προσεγγίσεις ως προς τις τεχνικές που χρησιμοποιούν σε δύο κατηγορίες:

- **Online Monitoring** κατά την συνεκτέλεση: Οι προτάσεις που εντάσσονται σε αυτή την κατηγορία ([29], [17], [13]) δεν απαιτούν γνώση των χαρακτηριστικών της απομονωμένης ("φυσιολογικής") εκτέλεσης της εφαρμογής, και δεν περιλαμβάνουν κάποιο profiling. Ως συνέπεια, το σύστημα δεν γνωρίζει τίποτα για την εφαρμογή

πριν την τοποθέτησή της σε κάποιον production server μαζί με άλλες εφαρμογές. Οι προσεγγίσεις αυτές στοχεύουν στον εντοπισμό του interference μέσω της online συλλογής μετρικών για την επίδοση των εφαρμογών, και τη χρήση αυτών για τη λήψη αποφάσεων σε περίπτωση που δεν ικανοποιούνται οι στόχοι επίδοσης που έχουν τεθεί (QoS).

- **Profiling**: Σε αυτή την κατηγορία περιλαμβάνονται μηχανισμοί που συγκεντρώνουν πληροφορίες σχετικά είτε με την απομονωμένη εκτέλεση μιας εφαρμογής, είτε με την συνεκτέλεσή της με άλλες, συγκεκριμένες εφαρμογές, με σκοπό την πρόβλεψη του interference και την μείωση των συνεπειών του. Οι μηχανισμοί αυτοί απαιτούν ένα απομονωμένο, ελεγχόμενο περιβάλλον (isolated server), όπου μπορεί να συλλεχθεί ο απαραίτητος αριθμός μετρήσεων για σενάρια απομονωμένης εκτέλεσης ή εσκεμμένης συνεκτέλεσης. Συνήθως συνδυάζονται με τη χρήση online monitoring, ώστε οι μετρήσεις του τελευταίου να μπορούν να συγκριθούν με αυτές της απομονωμένης εκτέλεσης και να εντοπιστεί το interference, καθώς και με τεχνικές διαμοιρασμού πόρων. Οι τεχνικές profiling που έχουν προταθεί στη βιβλιογραφία είναι οι εξής:

  - **Intrusive Micro-Benchmarks**: Συνθετικά μετροπρογράμματα υποβάλλουν την εφαρμογή σε συγκεκριμένη πίεση ως προς τους κοινούς πόρους, ώστε το σύστημα να καταγράψει την αντίδραση της εφαρμογής, και να μπορεί να προβλέψει το αποτέλεσμα σε περιπτώσεις που εφαρμογές σε σενάρια συνεκτέλεσης δημιουργήσουν αντίστοιχη πίεση ([4], [5]).
  - **Isolated Profiling**: Καταγραφή της συμπεριφοράς μιας εφαρμογής όταν τρέχει απομονωμένη ([30], [21], [27], [7]).

Κάθε μία από τις παραπάνω προσεγγίσεις και μηχανισμούς έχει συγκεκριμένα πλεονεκτήματα και μειονεκτήματα. Πιο συγκεκριμένα, το μεγάλο μειονέκτημα του profiling είναι ότι απαιτεί έναν ή περισσότερους απομονωμένους servers για να πραγματοποιηθεί, με αποτέλεσμα να μειώνονται ουσιαστικά οι πόροι και η υπολογιστική δύναμη του συστήματος. Επιπλέον, εάν διενεργείται πριν την τοποθέτηση μια εφαρμογής σε έναν production server (a-priori), καθυστερεί τη χρονοδρομολόγηση της εφαρμογής, μια καθυστέρηση που αυξάνει σημαντικά το κόστος λειτουργίας όσο αυξάνεται ο αριθμός των εφαρμογών προς εκτέλεση. Στον αντίποδα, εάν ο μηχανισμός profiling σχεδιαστεί και βελτιστοποιηθεί προσεχτικά, έχει τη δυνατότητα να ισοσταθμίσει τα προαναφερθέντα κόστη που επιφέρει. Σε περιβάλλοντα όπου δεκάδες εφαρμογές συνυπάρχουν σε ένα server, είναι πολύ δύσκολο να εντοπιστεί εκείνη η οποία δημιουργεί τον ανταγωνισμό λόγω του μεγάλου αριθμού αλληλεπιδράσεων. Ειδικότερα σε εμπορικά περιβάλλοντα υπολογιστικού νέφους, όπου οι πελάτες χρεώνονται ανάλογα με τη διάρκεια των διαστημάτων όπου οι εφαρμογές τους ικανοποιούν τους QoS στόχους τους, το interference μπορεί να επηρεάσει σημαντικά το οικονομικό κέρδος του παρόχου. Σε αυτές τις περιπτώσεις, η δυνατότητα αποφυγής ή ελαχιστοποίησης του interference πριν αυτό συμβεί αποτελεί σημαντικό πλεονέκτημα.

Η βασική πρόκληση για τις προτάσεις που δε χρησιμοποιούν profiling είναι ο ίδιος ο εντοπισμός του interference, και ο διαχωρισμός του από φυσιολογικές εναλλαγές φάσης

ή workload της εφαρμογής. Δεδομένου του ότι το σύστημα δε γνωρίζει τα χαρακτηριστικά της απομονωμένης εκτέλεσης της εφαρμογής, και του ότι οι περισσότερες εφαρμογές έχουν πολλαπλές φάσεις και δυναμικά workloads, ο εντοπισμός των περιπτώσεων όπου η μείωση της επίδοσης είναι εξαιτίας ανταγωνισμού και όχι φυσιολογική κι αναμενόμενη είναι ένα ιδιαίτερα απαιτητικό πρόβλημα. Αλγόριθμοι που εντοπίζουν τις φάσεις εκτέλεσης ([3], [20], [6]) μπορούν ενδεχομένως να χρησιμοποιηθούν, αλλά συνήθως εισάγουν πολύ μεγάλο υπολογιστικό κόστος και αποφεύγονται.

Τέλος, μία άλλη παράμετρος που πρέπει ο σχεδιαστής να λάβει υπόψιν του είναι η πολυπλοκότητα του συστήματος, και το επίπεδο στο οποίο λαμβάνει αποφάσεις. Συστήματα που λειτουργούν σε "χαμηλό" επίπεδο προσφέρουν υψηλά εξειδικευμένες και εξατομικευμένες πολιτικές χρονοδρομολόγησης και διαμοιρασμού πόρων, σχεδιασμένες για κάθε συγκεκριμένο σύνολο εφαρμογών. Μια τέτοια προσέγγιση πιθανότητα θα μεγιστοποιούσε τη χρησιμοποίηση των πόρων, αλλά θα περιλάμβανε σημαντικά κόστη υλοποίησης και χρήσης, καθώς θα απαιτούσε συχνή λήψη μεγάλου αριθμού μετρήσεων, επηρεάζοντας την ίδια την επίδοση των εφαρμογών.

1.3 Κατηγοριοποίηση Εφαρμογών για την Πρόληψη Φαινομένων Ανταγωνισμού

1.3.1 Πειραματική Πλατφόρμα και Μετροπρογράμματα

Όλες οι εκτελέσεις μετροπρογραμμάτων που παρουσιάζονται στην παρούσα διπλωματική εργασία έγιναν σε έναν επεξεργαστή Intel® Xeon® E5-2630 v4, του οποίου τα χαρακτηριστικά φαίνονται στον Πίνακα 1.1.

Οικογένεια Επεξεργαστών	Broadwell
Βασική Συχνότητα Επεξεργαστή	2.20 GHz
Αριθμός Πυρήνων	10
Αριθμός Νημάτων	20
L1 (data) Cache (ανά πυρήνα)	320 KB
L2 Cache (ανά πυρήνα)	2.5 MB
Last Level Cache (κοινή)	25 MB, 20-way
DRAM Bandwidth	68.3 GB/sec

Πίνακας 1.1: Χαρακτηριστικά του Επεξεργαστή Intel® Xeon® E5-2630 v4

Η οικογένεια επεξεργαστών στην οποία ανήκει ο Intel® Xeon® E5-2630 v4 περιλαμβάνει την Intel Resource Director Technology (RDT), η οποία επιτρέπει στον χρήστη να

παρακολουθεί την εκτέλεση διάφορων εφαρμογών (Cache Monitoring Technology, CMT) μέσω μετρικών επίδοσης υλικού (Performance Monitoring Units, PMUs) και να ελέγχει το διαμοιρασμό της LLC (Cache Allocation Technology, CAT). Για την λήψη των αναγκαίων μετρήσεων, απενεργοποιήσαμε το hyperthreading στον επεξεργαστή (ώστε να μπορούν να καταγραφούν έως και 8 PMUs ταυτόχρονα, σε αντίθεση με τα 4 PMUs που επιτρεπόταν αρχικά), και τροποποιήσαμε κατάλληλα την διεπαφή που προσφέρει η Intel για την λήψη μετρήσεων (PQoS API) ώστε να μπορεί να λαμβάνει τιμές για επιπλέον PMUs (πέραν των 4 που ήδη κάλυπτε). Επίσης, λάβαμε μετρήσεις και μέσω του εργαλείου linux perf, για να επιβεβαιώσουμε ότι συμφωνούν με αυτές του PQoS και ότι μπορεί να χρησιμοποιηθεί σαν εναλλακτική του. Τα αποτελέσματα από το linux perf ήταν στο μεγαλύτερο μέρος τους παρόμοια με αυτά του PQoS, αλλά περιλάμβαναν περισσότερο θόρυβο, οδηγώντας μας στην επιλογή του PQoS ως εργαλείου συλλογής μετρήσεων.

Τα μετροπρογράμματα (benchmarks) που χρησιμοποιήσαμε προέρχονται από τη σουίτα SPEC 2017, με την προσθήκη ενός μετροπρογράμματος από τη σουίτα Polybench 3.2 (jacobi-2d benchmark), καθώς και δύο άλλων μετροπρογραμμάτων (stream, hpcg). Επιπρόσθετα, χρησιμοποιήθηκε η σουίτα Alberta Workloads ([1]), η οποία περιλαμβάνει επιπλέον inputs για κάποια από τα SPEC 2017 benchmarks. Συνολικά, χρησιμοποιήθηκαν 140 benchmarks.

1.3.2 Σενάρια Συνεκτέλεσης

Αρχικά, εκτελέσαμε κάθε εφαρμογή σε απομονωμένο περιβάλλον, ώστε να καταγράψουμε τα χαρακτηριστικά της κατά την "φυσιολογική" εκτέλεση. Εν συνεχεία, δημιουργήσαμε και εκτελέσαμε σενάρια συνεκτελέσεων 2 εφαρμογών. Κάθε εφαρμογή ήταν μονονηματική και προσκολλημένη σε έναν πυρήνα. Εάν μία εφαρμογή ολοκληρωνόταν πριν την άλλη, ξαναξεκινούσε, μέχρι να ολοκληρωθούν και οι 2 τουλάχιστον μία φορά. Ποσοτικοποιήσαμε την επίδοση κάθε εφαρμογής χρησιμοποιώντας το IPC (Instructions Per Cycle, Εντολές Ανά Κύκλο) και το συνολικό χρόνο εκτέλεσης, ενώ για την μέτρηση του interference ορίσαμε τις μετρικές *Sl* (Slowdown) and *Deg* (Degradation) ως:

$$Sl = \frac{t_{coexec}}{t_{alone}}$$

$$Deg = \frac{ipc_{alone} - ipc_{coexec}}{ipc_{alone}} * 100\%$$

όπου:
t_{alone}: συνολικός χρόνος απομονωμένης εκτέλεσης
t_{coexec}: συνολικός χρόνος εκτέλεσης στο σενάριο συνεκτέλεσης
ipc_{alone}: ipc απομονωμένης εκτέλεσης
ipc_{coexec}: ipc στο σενάριο συνεκτέλεσης

Μελετώντας διαφορετικά σενάρια συνεκτέλεσης, παρατηρήσαμε ότι συγκεκριμένες εφαρμογές εμφανίζουν πάντα την ίδια συμπεριφορά ως προς το interference που δημιουργούν ή δέχονται, ανεξάρτητα της εφαρμογής με την οποία εκτελούνται. Για παράδειγμα,

το stream δημιουργεί μείωση στην επίδοση οποιασδήποτε εφαρμογής με την οποία συνεκτελείται, ενώ το xz_r εμφανίζει μειωμένη επίδοση με οποιαδήποτε εφαρμογή κι αν συνεκτελεστεί. Σταθερές συμπεριφορές όπως οι παραπάνω αποτελούσαν ένδειξη για το ότι το κατά πόσο μια εφαρμογή θα προκαλέσει ή θα επηρεαστεί από κάποια άλλη είναι ένα εγγενές χαρακτηριστικό της εφαρμογής. Εάν αυτό το χαρακτηριστικό μπορεί να συσχετιστεί με μετρικές υλικού που έχουν καταγραφεί κατά την απομονωμένη εκτέλεση, τότε μπορεί κάποιος να ξέρει την συμπεριφορά της εφαρμογής ως προς το interference χωρίς να της δημιουργήσει εσκεμμένο interference, και να λάβει αποφάσεις σχετικά με την χρονοδρομολόγησή της και τους πόρους που θα της παραχωρηθούν.

1.3.3 Σχεδιασμός Αλγορίθμου Κατηγοριοποίησης

1.3.3.1 Σχεδιαστικές Επιλογές

Οι όροι *noise* (ή και contentiousness) και *sensitivity* χρησιμοποιούνται από τη βιβλιογραφία για να περιγράψουν το πόσο μία εφαρμογή υποφέρει από ή μπορεί να δημιουργήσει μείωση επίδοσης σε σενάρια συνεκτέλεσης. Γενικά, μια εφαρμογή χαρακτηρίζεται ως **noisy** εάν οδηγεί σε σημαντική μείωση της επίδοσης της συνεκτελούμενης εφαρμογής, και **quiet** εάν την αφήνει ανεπηρέαστη. Αντίστοιχα, χαρακτηρίζεται ως **sensitive** εάν η επίδοσή της μειώνεται ανεξάρτητα της συνεκτελούμενης εφαρμογής, και **insensitive** εάν η επίδοσή της μένει σχεδόν πάντα σταθερή. Στο παρελθόν έχουν γίνει αρκετές προσπάθειες ([16], [28], [31], [26]) κατηγοριοποίησης των εφαρμογών λαμβάνοντας υπόψιν τα παραπάνω χαρακτηριστικά, αλλά όλες εμφάνιζαν σημαντικούς περιορισμούς ως προς την ακρίβεια και την απόδοση. Η προσέγγισή μας στόχευε στη δημιουργία ενός αλγόριθμου κατηγοριοποίησης που θα είχε τα εξής χαρακτηριστικά:

- **Προληπτικός** (Preventive): Θα θέλαμε να μπορούμε να προλάβουμε τον ανταγωνισμού προτού συμβεί, όχι να τον αντιμετωπίσουμε αφού εμφανιστεί.
- **Χαμηλού Κόστους** (Lightweight): Οι τεχνικές που χρησιμοποιούμε θα πρέπει να μην επιβαρύνουν την επίδοση των εφαρμογών, ούτε να απαιτούν διατήρηση μεγάλων βάσεων δεδομένων ή μοντέλων ή να έχουν υψηλό υπολογιστικό κόστος.
- **Μη-Επεμβατικός** (Non-Intrusive): Δε θα πρέπει να απαιτείται εκτέλεση σεναρίων με εσκεμμένη πρόκληση interference.

Με βάση τα παραπάνω, επιλέξαμε μία αντίστροφη προσέγγιση. Πρώτα κατηγοριοποιήσαμε τις εφαρμογές παρατηρώντας τον τρόπο που αλληλεπιδρούσαν με άλλες στα σενάρια συνεκτέλεσης, και μετά προσπαθήσαμε να δούμε αν μετρικές που είχαν ληφθεί στις απομονωμένες εκτελέσεις μπορούσαν να χρησιμοποιηθούν για να δικαιολογήσουν τα μέλη κάθε κατηγορίας. Κάθε εφαρμογή χαρακτηρίστηκε ως προς το noise ως *noisy*, *potentially noisy* (κάποιες φορές προκαλεί interference και κάποιες όχι, ανάλογα με την εφαρμογή που συνεκτελείται) ή *quiet*, και ως προς το sensitivity ως *sensitive*,*potentially sensitive* (κάποιες φορές εμφανίζει μειωμένη απόδοση λόγω interference και κάποιες όχι, ανάλογα με την εφαρμογή που συνεκτελείται) ή *insensitive*. Η διαδικασία με την οποία χαρακτηρίστηκαν οι εφαρμογές παρουσιάζεται αναλυτικά στο 4.5.1.

1.3.3.2 Μοτίβα σε Μετρικές Επίδοσης Υλικού

Η πρώτη μας προσέγγιση ήταν να εξετάσουμε την περίπτωση κάποιες PMUs να μπορούν να υποδείξουν την κατηγορία στην οποία ανήκει μία εφαρμογή. Συγκεντρώσαμε μετρήσεις για όλες τις εφαρμογές από 25 διαφορετικά hardware events σχετικά με την ιεραρχία κρυφών μνημών και την κεντρική μνήμη, και προσπαθήσαμε να τις συσχετίσουμε με τις διάφορες κατηγορίες, λαμβάνοντας υπόψιν τις παρατηρήσεις των Molka et al. ([19]) και Subrmanian et al. ([25]), καθώς και τις δικές μας παρατηρήσεις. Πέραν από τις απευθείας μετρήσεις των PMUs, συνδυάσαμε διαφορετικές PMUs μεταξύ τους για να δημιουργήσουμε νέες μετρικές (π.χ. misses per kilo instructions).

Δύο μετρικές που φάνηκαν να μπορούν να χρησιμοποιηθούν στην κατηγοριοποίηση ήταν το LLC acpki (accesses per kilo isntructions) και το memory bandwidth. Εφαρμογές που ήταν quiet και ταυτόχρονα insensitive εμφάνιζαν LLC acpki κοντά στη μονάδα, πιθανότατα επειδή εφαρμογές που κάνουν λίγες προσβάσεις στην LLC δεν επηρεάζονται και δεν επηρεάζουν τις προσβάσεις άλλων εφαρμογών. Αντίστοιχα, noisy εφαρμογές εμφάνιζαν υψηλές τιμές memory bandwidth, γεγονός που οφείλεται στο ότι εάν μια εφαρμογή κάνει πολλές και συχνές προσβάσεις στη μνήμη, καταναλώνει όλο το διαθέσιμο bandwidth, εμποδίζοντας έτσι την εκτέλεση των υπόλοιπων εφαρμογών. Μία επιπλέον μετρική που εμφάνισε ενδιαφέρον οι είναι κύκλοι στους οποίους η εκτέλεση παύεται επειδή εκκρεμεί κάποιο αίτημα προς κάποιο επίπεδο μνήμης (cache/memory stalls). Sensitive εφαρμογές φάνηκε να έχουν γενικά περισσότερα stalls.

Αντίθετα, μετρικές που συχνά χρησιμοποιούνταν στη βιβλιογραφία, όπως το LLC mpki (misses per kilo instructions) και το LLC miss rate (misses per accesses) δεν φάνηκαν να εμφανίζουν κάποιο διακριτό μοτίβο. Ακόμα και για τις μετρικές που πιθανώς ακολουθούσαν κάποιο μοτίβο, όπως το LLC acpki, δεν μπορούσαμε να καθορίσουμε συγκεκριμένες τιμές-κατώφλια που να διαχωρίζουν επαρκώς τις εφαρμογές.

1.3.3.3 Κατηγοριοποίηση με τον Αλγόριθμο K-Means

Η επόμενη προσέγγισή μας ήταν να χρησιμοποιήσουμε τον αλγόριθμο k-means ([12]), έναν αλγόριθμο ο οποίος διαχωρίζει ένα σύνολο παρατηρήσεων σε k ομάδες (clusters) χρησιμοποιώντας Ευκλείδειες αποστάσεις, με κάθε παρατήρηση να χαρακτηρίζεται από ένα διάνυσμα τιμών (features) (όλες οι παρατηρήσεις έχουν ίδια features, αλλά καθεμία έχει διαφορετικές τιμές για το καθένα). Στην περίπτωσή μας, κάθε εφαρμογή ήταν μία παρατήρηση, και τα features ήταν διαφορετικές PMUs. Στόχος μας ήταν να βρούμε το κατάλληλο σύνολο features ώστε τα clusters που θα προέκυπταν να ήταν ίδια με αυτά που είχαμε ήδη δημιουργήσει εποπτικά. Η διαδικασία που ακολουθήθηκε περιγράφεται στο 4.5.3. Ο k-means κατάφερε σε κάποιες περιπτώσεις, όπως αυτή της κλάσης noisy, να δημιουργήσει cluster παρόμοια με τα ζητούμενα, αλλά στην πλειοψηφία των περιπτώσεων απέτυχε. Η αποτυχία του αυτή οφείλεται στο ότι υποθέτει κλάσεις μη-επικαλυπτόμενες στο χώρο και παρόμοιου μεγέθους, οι οποίες συνθήκες δεν ισχύουν στην περίπτωσή μας.

1.4 Χαρακτηρισμός Εφαρμογών με τη Χρήση Μηχανικής Μάθησης

Τέλος, προσεγγίσαμε το πρόβλημα ως ένα πρόβλημα ταξινόμησης επιβλεπόμενης μηχανικής μάθησης (supervised machine learning classification problem). Στόχος μας ήταν η δημιουργία δύο ταξινομητών (classifiers), ενός για το noise κι ενός για το sensitivity, οι οποίοι θα λάμβαναν ένα διάνυσμα από features για κάθε εφαρμογή (διαφορετικό σύνολο features για κάθε classifier) και θα την τοποθετούσαν στην αντίστοιχη κατηγορία. Το μοντέλο που επιλέξαμε για τους classifiers είναι το SVM (Support Vector Machine). Το συγκεκριμένο μοντέλο είναι εξαιρετικά κατάλληλο για μικρά datasets (140 εφαρμογές στην περίπτωσή μας), καθώς δημιουργεί τα σύνορα μεταξύ των κλάσεων χρησιμοποιώντας μόνο τα στοιχεία κοντά στο υποψήφιο σύνορο (support vectors) και όχι όλα τα στοιχεία του dataset. Για τον ίδιο λόγο είναι και αρκετά ανθεκτικό σε outliers (στοιχεία με μη ομαλή συμπεριφορά), ενώ είναι και ένας σχετικά χαμηλού κόστους μοντέλο ως προς την εκπαίδευση και τη χρήση.

Το αρχικό dataset διαχωρίστηκε δύο ξεχωριστές φορές, ώστε να δημιουργηθεί ένα σύνολο εκπαίδευσης (train set) και ένα σύνολο αξιολόγησης (test set) για τον κάθε classifier. Τα train και test sets κατασκευάστηκαν με την τεχνική του stratified sampling, ώστε κάθε set να έχει έναν αντιπροσωπευτικό αριθμό στοιχείων από κάθε κλάση (και άρα να είναι μία όσο το δυνατόν πιο πιστή μικρογραφία του αρχικού dataset). Σε μεγαλύτερα datasets συνήθως χρησιμοποιείται τυχαίο sampling για την δημιουργία των train και test sets, καθώς ο πληθυσμός είναι τόσο μεγάλος που μια τυχαία επιλογή μπορεί να δημιουργήσει αντιπροσωπευτικά δείγματα, αλλά το μέγεθος του dataset μας ήταν απαγορευτικό για να εφαρμοστεί αποτελεσματικά μια τέτοια τεχνική.

Εν συνεχεία, εφαρμόσαμε στα train sets έναν αλγόριθμο κανονικοποίησης (standardization scaling), ώστε να μην επηρεαστούν οι classifiers από τις διαφορετικές τάξεις μεγέθους των διάφορων features. Για να γίνει η επιλογή του συνόλου features για κάθε classifier, χρησιμοποιήθηκε τόσο ένας αλγόριθμος αυτόματης επιλογής features (Recursive Feature Elimination, RFE), όσο και οι παρατηρήσεις μας από τις προηγούμενες ενότητες. Παρόλο που ο αλγόριθμος RFE επέλεξε σε κάποιες περιπτώσεις αναμενόμενα features, όπως το memory bandwidth, κάποιες φορές κατέταξε ως δευτερεύοντες μετρικές που θα περιμέναμε να είναι πρωτεύοντες, όπως το LLC acpki για τον sensitivity classifier. Εκπαιδεύσαμε πολλά διαφορετικά μοντέλα, με feature sets διαφορετικού μεγέθους και με διαφορετικά features, για να επιλέξουμε τα καταλληλότερα (Πίνακας 1.2).

Μετά την εκπαίδευση, κάθε μοντέλο υποβλήθηκε σε μία διαδικασία βελτιστοποίησης. Η υλοποίηση του SVM που χρησιμοποιήσαμε (python scikit-learn framework) παρέχει δύο παραμέτρους για τον έλεγχο της μορφής των συνόρων που θα δημιουργήσει ο αλγόριθμος, τα C και gamma. Οι παράμετροι αυτοί μπορούν να καταστρατηγηθούν τόσο για τη βελτίωση της επίδοσης του classifier, όσο και για τον έλεγχο του φαινομένου του overfitting. Υπάρχουν περιπτώσεις όπου ένα μοντέλο μαθαίνει με τόση λεπτομέρεια το train set, όπου κατασκευάζει ένα εξατομικευμένο σύνορο, με αποτέλεσμα να μην μπορεί να γενικευτεί σε νέα δεδομένα. Σε αυτή την περίπτωση γίνεται λόγος για overfitted μοντέλο. Αντίθετα, όταν το σύνορο που δημιουργείται είναι υπερβολικά απλουστευμέ-

συνολικά features που εξετάστηκαν	LLC acpki, LLC mpki, LLC miss rate, DRAM Bandwidth, total L2 pending miss stalls, total L2 pending miss stalls/total cycles, store buffer stalls
τελικό noise feature set	LLC acpki, LLC mpki, LLC miss rate
τελικό sensitivity feature set	LLC mpki, LLC acpki, DRAM bandwidth, total L2 pending miss stalls/total cycles

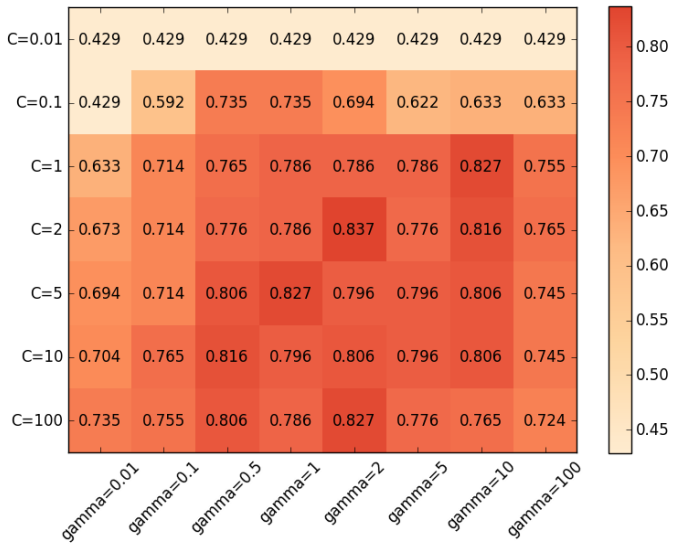
Πίνακας 1.2: Features που χρησιμοποιήθηκαν στους τελικούς classifiers

νο και δεν καταφέρνει να καταγράψει τα χαρακτηριστικά του πληθυσμού γίνεται λόγος για underfitting. Και σε αυτή την περίπτωση, το μοντέλο δεν μπορεί να γενικευτεί επαρκώς. Πέραν των C και gamma, έχουμε και τη δυνατότητα να εκπαιδεύσουμε μοντέλα με διαφορετικούς μαθηματικούς πυρήνες (η συνάρτηση με βάση την οποία υπολογίζονται τα σύνορα). Οι πυρήνες που εξετάστηκαν ήταν ο γραμμικός (linear) και ο gaussian. Για τη διερεύνηση των παραμέτρων εφαρμόσαμε τη μέθοδο 10-fold cross validation με grid search, και αξιολογήσαμε κάθε συνδυασμό με ένα σύνολο διαφορετικών μετρικών επίδοσης (recall, precision, f1 score). Το σύνηθες μέτρο επίδοσης για έναν classifier, η ακρίβεια (accuracy), δεν μπορούσε να χρησιμοποιηθεί στην περίπτωσή μας, καθώς οι εφαρμογές ήταν άνισα κατανεμημένες στις διάφορες κλάσεις.

Στο Σχήμα 1.1 παρουσιάζουμε ενδεικτικά τα scores ενός sensitivity classifier με συγκεκριμένο feature set και gaussian πυρήνα, για διάφορες τιμές των παραμέτρων C και gamma (ως μετρική χρησιμοποιείται το recall). Γενικά, χαμηλές τιμές C οδηγούν σε underfitting (χαμηλό recall), ενώ υψηλότερες τιμές για τα C και gamma μπορεί να οδηγήσουν σε overfitting. Παρατηρούμε επίσης ότι υπάρχουν συγκεκριμένοι συνδυασμοί τιμών για τις παραμέτρους που επιτυγχάνουν πολύ καλύτερη επίδοση σε σχέση με την πλειοψηφία. Σε κάθε περίπτωση, λόγω του πολύ μικρού μεγέθους του dataset μας, κάθε αποτέλεσμα αντιμετωπίστηκε με μεγάλη προσοχή.

Μετά την ολοκλήρωση της παραπάνω διαδικασίας, επιλέξαμε τους 5 καλύτερους noise και τους 5 καλύτερους sensitivity classifiers, οι οποίοι έπειτα αξιολογήθηκαν στα αντίστοιχα test sets. Τα χαρακτηριστικά των δύο τελικών classifiers που χρησιμοποιήθηκαν παρουσιάζονται στον Πίνακα 1.3, ενώ τα scores του με βάση διάφορες μετρικές παρατίθενται στον Πίνακα 1.4.

Παρατηρώντας την κατανομή των προβλέψεων των classifiers για τα test sets, συμπεράναμε ότι εμφανιζόταν ένα bias προς τις κλάσεις potentially sensitive και potentially noisy. Το φαινόμενο αυτό οφείλεται στο ότι οι κλάσεις αυτές περιέχουν περισσότερα στοιχεία από τις υπόλοιπες, και άρα οι classifiers έχουν εκπαιδευτεί με περισσότερα στιγμιότυπά τους και δίνουν μεγαλύτερη πιθανότητα σε ένα στοιχείο να ανήκει σε αυτές. Το bias αυτό μπορεί να εξαλειφθεί σε σημαντικό βαθμό χρησιμοποιώντας κάποια τεχνική oversampling στο αρχικό dataset. Πειραματιστήκαμε με τον αλγόριθμο SMOTE (Synthetic Minority Over-sampling TEchnique), ο οποίος εισάγει τεχνητά στοιχεία στις νοητές ευθείες που ενώνουν τα προϋπάρχοντα στοιχεία του dataset. Εκπαιδεύοντας ξα-



Σχήμα 1.1: Διερενηση παραμέτρων για έναν sensitivity classifier. Πολύ χαμηλές τιμές για το C οδηγον σε underfitting.

Τύπος Classifier	Feature Set	Πυρήνας	C	Gamma
noise	LLC mpki, LLC acpki, LLC miss rate	gaussian	10	1
sensitivity	LLC mpki, LLC acpki, DRAM bandwidth, total L2 pending miss stalls/total cycles	gaussian	2	1

Πίνακας 1.3: Χαρακτηριστικά τελικών noise και sensitivity classifiers

Τύπος Classifier	noise	sensitivity
Accuracy	0.8333	0.8095
Recall (macro)	0.8333	0.8095
Recall (micro)	0.8005	0.7787
F1 score (macro)	0.8322	0.8095
F1 score (micro)	0.8271	0.7902

Πίνακας 1.4: Scores των τελικών noise και sensitivity classifiers στο αντίστοιχο test set

νά τα μοντέλα που επιλέξαμε παραπάνω στα νέα train sets, και αξιολογώντας τα στα νέα test sets, παρατηρήσαμε ότι το bias είχε εξαφανιστεί, δίχως να επηρεαστούν σημαντικά τα scores των classifiers.

1.5 Συμπεράσματα και Μελλοντικές Επεκτάσεις

Τα πειράματά μας και η τελική επίδοση των classifiers αποτελούν ενδείξεις υπέρ του ότι το noise και το sensitivity είναι δύο χαρακτηριστικά που μπορούν να συσχετιστούν με μετρικές επίδοσης υλικού και να εντοπιστούν στη συμπεριφορά μίας εφαρμογής κατά την απομονωμένη της εκτέλεση. Τα αποτελέσματα αυτά πρέπει σε κάθε περίπτωση να αντιμετωπιστούν με προσοχή, κυρίως λόγω του σχετικά μικρού συνόλου εφαρμογών που μελετήθηκαν. Παραθέτουμε παρακάτω κάποιες προτάσεις για περαιτέρω επέκταση της παρούσας διπλωματικής:

- Εμπλουτισμός του dataset με μεγαλύτερο αριθμό εφαρμογών, και με εφαρμογές από διαφορετικά επιστημονικά πεδία (cloud computing, graph processing, machine learning) για να αυξηθεί η επίδοση των classifiers.
- Πειραματισμός με διαφορετικούς τύπους μοντέλων μηχανικής μάθησης.
- Δημιουργία υποκλάσεων από τις υπάρχουσες κλάσεις οι οποίες θα περιγράφουν με μεγαλύτερη ακρίβεια τη συμπεριφορά των εφαρμογών. Οι κλάσεις που ορίσαμε είναι εξειδικευμένες σε βαθμό που να εξυπηρετεί το σκοπό της παρούσας διπλωματική εργασίας, αλλά εμφανίζουν εσωτερικά κάποια ετερογένεια (ειδικά οι κλάσεις potentially noisy και potentially sensitive), η οποία θα μπορούσε να αξιοποιηθεί για τη λήψη πιο εξατομικευμένων αποφάσεων κατά τη χρονοδρομολόγηση και το διαμοιρασμό των πόρων.
- Μελέτη της επίδρασης στις κλάσεις του πολυνηματισμού (multithreading). Σε σενάρια όπου συνεκτελούνται περισσότερα του ενός αντίγραφα (νήματα) μίας εφαρμογής, τα χαρακτηριστικά δρουν ”αθροιστικά” μετατρέποντας για παράδειγμα μια potentially noisy εφαρμογή σε noisy. Θα ήταν εξαιρετικά χρήσιμο για έναν χρονοδρομολογητή να γνωρίζει σε ποιο βαθμό η συμπεριφορά των εφαρμογών κάθε κλάσης ”μεγεθύνεται” ή αλλάζει όταν συνεκτελούνται πολλά νήματα μίας εφαρμογής.

Chapter 2

Introduction

2.1 Modern Multicore Systems

Undoubtedly, the creation of multiprocessors has played a significant role in the dramatic increase of the computational power modern computers possess. Before multicore Central Processing Units (CPUs) made their appearance, a single-core system fostering more than one applications would showcase significantly decreased performance, as each application could only be scheduled after the previous one was completed. The issue of the increasing number of applications that needed to be run concurrently was tackled both on the software level, with the introduction of hyperthreading and the scheduling of multiple threads of execution in one physical core, and the hardware level, with attempts to create systems with more than one CPUs. On the downside, the first solution could lead to the system underperforming, if the added needs in resources for the threads sharing a core surpass what the core can offer, whereas the second one introduces a noteworthy overhead due to the necessary data transfer between different chips, balancing out the performance gains yielded by the presence of multiple CPUs. Thus, the creation of a single chip containing more than one processing cores was a revolution, decreasing the overhead of communication between chips and providing multiple applications with sufficient resources to function properly.

Although each core in a multiprocessor has some resources private to itself and thus the thread running on it, several aspects of the chip are still shared between different concurrent threads. A typical example of a shared component is the Last Level Cache (LLC). Caches are utilized to increase performance, as they keep frequently accessed data close to the processor, minimizing in theory the need for time and energy consuming accesses to the main, off-chip memory. Modern multicores usually have cache hierarchies consisting both of private and shared caches. In the example below (Figure 2.1), each core has a private, Level 1 (L1) and Level 2 (L2) cache, and all cores share a common Level 3 (L3 or Last Level) cache. Except for caches, cores (and thus applications) also share the bus interface that communicates with the main memory, making DRAM Bandwidth (the rate at which a processor write to / reads from memory) another crucial shared component. Other

resources that could be affected by interference are the memory controllers, the network bandwidth and the interconnection network, but this thesis focuses upon the LLC and the DRAM Bandwidth.

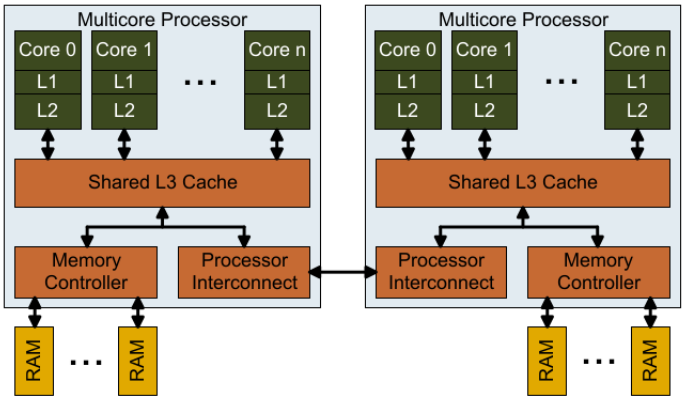


Figure 2.1: A typical multiprocessor architecture

Note: In this thesis, the words "application" and "thread" are used interchangeably, as all discussed applications are single-threaded.

2.2 The Interference Problem

Multiprocessor technology has definitely evolved since it was first introduced, and has overcome many of the problems it used to face, but as the need for resource utilization and the amount of applications and threads ran on a multiprocessor continue to increase, several issues regarding resource sharing have been detected.

Shared resources significantly suffer from application interference as past bibliography has showcased ([5],[10],[15],[24]), becoming a bottleneck for execution and affecting the overall performance of the system. More specifically, we can detect two main ways applications interfere with each other's execution:

- LLC Contention: Different applications have different workloads and access different memory locations. As a result, applications that make frequent accesses to the LLC and/or utilize a large portion of it might evict data stored there from other applications, or suffer themselves from frequently evicted data. This phenomenon is augmented in inclusive cache hierarchies, where all the data stored int the L1 and L2 caches must also be stored in the LLC. This constant rivalry for cache space leads to increased miss rates, which degrade performance and increase energy consumption.
- DRAM Bandwidth Contention: Applications that make many and frequent requests to the main memory compete with each other for the available Memory Bandwidth, usually due to access patterns than do not benefit from cache hierarchies.

Simply increasing the size of those resources to the extent of fully satisfying modern application needs is unrealistic. As a result, a vast amount of research has been conducted to resolve the interference issue, extending across the development stack. As no one solution has proven to be a panacea, researchers continue to examine all possible approaches, as the interference problem appears to need a multi-level, collaborative solution.

2.3 Resource Sharing in the Cloud

As multiprocessors are key to achieving high performance in environments with hundreds of applications, they have become the dominant processor type in cloud environments. However, such environments are distinctively different from commercial computers, as the applications they serve as well as the performance restrictions they have exhibit some unique characteristics.

Cloud applications are classified into two categories:

- Latency-Critical (LC): User-facing applications, such as social media or advertising, sensitive to the request response latency.
- Best Effort (BE): Batch applications, typically computationally intensive.

LC applications usually have strict performance requirements, which can easily be violated as a consequence of interference. In a commercial cloud that employs a pricing model based upon requested performance guarantees ([15]), stricter and higher Quality of Service (QoS) goals come with a higher cost for the costumer and profit for the provider. However, the need to abide by such goals (as costumers are charged only when they are met) may lead to system underutilization, if applications need to be isolated into servers to achieve them. As datacenters host up to thousands of applications, it becomes evident that naively scheduling LC applications to dedicated, isolated servers severely hinders the infrastructure from reaching its maximum potential. Consequently, resource sharing between applications seems to be inevitable, augmenting the aforementioned interference problem to the extreme and making the creation of efficient policies to tackle it necessary.

Chapter 3

Approaches to the Interference Problem

3.1 Overview

As we have already established, interference in shared resources is a problem that has not yet been resolved, and one that most probably requires a combination of mechanisms across the execution pipeline, from the point an application arrives to the server until its execution completes. Solutions proposed usually are consistent with this rule, proposing full mechanisms that span the pipeline and address interference in multiple stages.

Since present solutions comprise of manifold complementary methods, there is no single base upon which they can be clearly categorized. Those solutions target in one or more of the following when it comes to interference:

1. Prevention
2. Detection
3. Mitigation

In general, the pipeline followed upon the arrival of a new application can be summarized in the following steps:

1. Application arrival
2. (Optional) Offline application profiling
3. Scheduling and resource allocation decisions
4. Start of execution
5. (Optional) Application profiling concurrent to the execution

6. (Optional) Online monitoring of execution and adjustment of scheduling and/or resource allocation decisions

In a naive system, steps 2,5 and 6 would be completely omitted, and scheduling and resource allocation would be done in a random fashion. Steps 2,3,5,6 present the system designer with several choices: from whether to include them or not (if they are marked as ”Optional”) to the specifics of each step's implementation (for example, the scheduling algorithm, the performance counters used, or the allocation policy). Consequently, those steps and the decisions designers make can be used to broadly categorize research on the matter.

3.2 Online Monitoring during Co-Execution

Firstly, we will examine proposals that do not include any kind of profiling of individual applications. This means the system has no knowledge of that the characteristics of each application, such as its memory access patterns or LLC utilization, before its assignment to a production server. As a result, those proposals mainly aim to detect and moderate interference as soon as it begins, rather than prevent it. in general, they rely solely on gathering measurements (in the form of performance counters) during the co-execution of applications, which are later used to make scheduling and/or resource allocation decisions, in the case that performance goals are not met.

CPI^2 [29] is a mechanism developed by Google to monitor the performance of jobs (applications split down to multiple threads/tasks) running on their cloud clusters and manipulate the measurements to detect when a job's performance is degrading. Zhang et al. argue that the Cycles Per Instruction (CPI) metric is adequate to identify interference in a cloud environment, since it correlates highly both with request latency and transactions per second, the main performance metrics used for latency sensitive and batch jobs respectively. The CPI^2 overview is presented in Figure 3.1.

Firstly, CPI samples per job (all the threads of a job in a specific machine) are collected from all the machines for a 10 second period every minute and sent to a per-cluster CPI sample aggregator. The aggregator creates a per-job per-cpu type structure (refreshed daily) that includes the corresponding CPI mean and standard deviation from the collected samples along with historical data. Those strictures are the ”predictions” of how a specific job normally executes, and are sent to local agents running in each machine. Each agent collects one CPI sample per-task per-minute and uses the predicted values to determine if it is an outlier. If more than 3 outliers are gathered over a 5-minute window, the job's behavior is marked as *anomalous*. The harmful antagonist is identified by correlating the harmed job's CPI samples with the CPU usage of its co-runners. Possible antagonists are addressed by CPU usage throttling. As it is targeted at a real-life commercial cloud environment, CPI^2 is an example of a mechanism that tries to minimize as much as possible overheads, mainly by using only one performance metric, lightweight monitoring and simple decision-making to detect and reduce contention. However, their approach solely focuses upon compute-intensive workloads, making no reference to the ways CPI

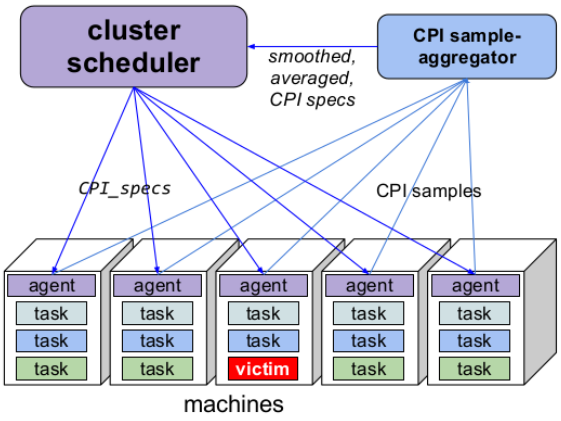


Figure 3.1: The CPI^2 pipeline

accounts for interactions with the memory subsystem, the network and other resources. Also, their contention reduction mechanism can be considered as rather coarse-grained, as it attempts to restore performance by capping all execution of the suspected antagonist, instead of only memory operations, LLC occupancy etc.

Heracles [17] is a more sophisticated approach that targets resource underutilization by focusing on the different and sometimes complementary performance needs of LC and BE applications. Lo et al. note that an interesting approach to minimizing resource underutilization would be the collocation of LC and BE applications, as the first remain idle for large periods of time, during which the latter can take over the unused resources. They go on to define an optimization problem, where the target is maximum utilization with respect to the QoS goals of LCs, and structure Heracles upon the premise that interference in a resource is considered harmful only when its utilization is so high it affects an LC application, which is always prioritized over a BE application.

The metrics monitored by Heracles are queries per second (application load) and tail latency. A top-level controller is responsible for deciding if BE application will be collocated with a LC one, depending on whether they threaten its performance. BE execution is halted when the LC workload is above 85% of its maximum in the server, and is restored when load drops beneath 80%. It is also suspended for an amount of time (before re-attempting to start BE execution) when the latency slack (the difference between the QoS target and the measured tail latency) is negative, accounting for load spikes. If BE execution is decided, the top-level controller forwards the latency lack values to three independent sub-controllers, each responsible for a shared resource: core and memory, CPU frequency and network bandwidth. Note that the first sub-controller accounts for both core count and cache and memory bandwidth portion granted to BE applications, as the authors notice a strong connection between them. Each sub-controller follows a specific algorithm that dictates how the respective resources are allocated to BE applications while always ensuring that LCs' performance targets are not violated.

Heracles was evaluated using representative workloads of containing different LC and

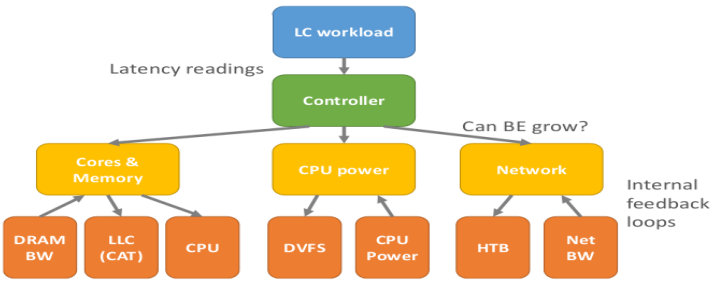


Figure 3.2: Heracles overview

BE applications, and managed to always satisfy QoS goals. Additionally, resource utilization was notably increased across all resources, with some even approaching 90%. The authors also introduced a new metric to describe combined performance, called Effective Machine Utilization (EMU), which is equal to the sum of the LC applications throughput with that of the BE applications. EMU was also shown to increase in all examined workloads, as a result of the system's consideration of more subtle interference relationships in different resources and its fine-tuned policies of reducing said interference.

Although Heracles adopts a more fine-grained approach to interference that CPI^2 by controlling the allocation of discrete shared resources, it places its focus point on maximizing resource utilization. The workloads examined consist of only two types of applications, where contentious co-runners are all considered equally responsible for performance degradation, so interference detection is more straightforward. Kannan et al. ([13]) on the other hand concentrate on workloads whose applications place stress on different resources, and attempt to pinpoint the contentious co-runner. The resources considered are: CPU, LLC, Network Bandwidth and I/O. Proctor is divided into two components: a Performance Degradation Detector (PDD) and a Performance Degradation Investigator (PDI).

The PDD is responsible for constantly monitoring application execution and informing the system when contention in a shared resource is discovered. To that end, the QoS metric (IPC for CPU/LLC sensitive applications, I/O latency and throughput for I/O applications and tail latency or network throughput for Network applications) of each running application is continuously sampled, and step detection is applied to the resulting time series. Step detection is a process of finding abrupt changes in a time series, and is implemented by Proctor using the finite difference method. The PDD performs pair wise difference of subsequent elements in the time series, and signals a possible interference issue when the finite difference of two elements spikes. The process is illustrated in Figures 3.3 and 3.4. The timestamp of the spike is saved and propagated to the PDI. To reduce noise in the time series before step detection is applied, Proctor utilizes median filtering, with a moving window that selectively discards elements that are notably higher than the window's median. This technique proves to be crucial for minimizing false positives (for example, in case of spikes) in interference detection, while simultaneously preserving the characteristics of the time series without excessively smoothing it.

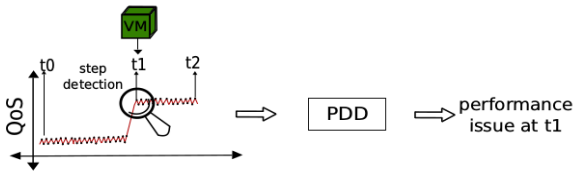


Figure 3.3: Proctor's performance degradation detection

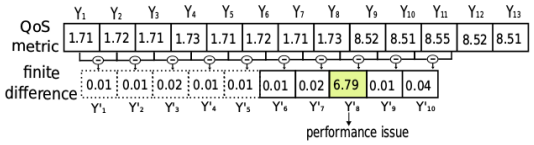


Figure 3.4: Time series step detection using the finite difference method

When the PDD flags an interference incident, the PDI is activated to investigate it. During application execution, low level metrics such as cache misses and context switches are collected, and when Proctor suspects contention for application A in a workload, it tries to correlate A's QoS metric time series with the low-level time series of its co-runners by obtaining the Pearson's Correlation Coefficient. The co-runners whose low-level metrics exhibit higher correlation with the affected application are labeled as the contentious ones, and the corresponding metrics are use to dictate the source which is more likely saturated. We assume that the timestamp sent by the PDD is so as the time series correlated include the moment that interference begun. To reduce overheads due to the volume of the data collected, real-time subsampling is performed upon the collected series using the Chi-square χ^2 test before they are correlated. To mitigate interference, Proctor simply migrates the contentious application to a different CPU/network channel/physical disk.

The speedup observed by the authors when Proctor is used is on average above 2.0 compared to a system where no interference detection and mitigation mechanism is used. Furthermore, the computational overhead of its components is rather small, and its 8% false positive rate (times that non-existing intrusion was identified) is characterized as small, even though the cost of those false positives and the migrations they lead to are not reported. In addition, the examined 5-application workloads include only one affected and one contentious application, with the contention taking place in only one resource. It is unclear whether Proctor's techniques would perform as effectively in workloads with multiple intruders or when contention spans across multiple resources.

3.3 Profiling-Based Mechanisms

This category includes mechanisms that attempt to acquire information about an application's normal (isolated) execution characteristics or its reaction to specific co-runners, in order to make decisions to avoid and/or lessen interference consequences.

One way of categorizing profiling techniques is according to when they are employed;

either before or concurrently to the application execution. In both cases they require a dedicated server that resembles a ”lab environment”, meaning an application can be heavily monitored and executed without interference, or with deliberate, known interference. It is important though to underline at this point the difference between online monitoring and what we characterize as profiling concurrently to the execution: online monitoring collects metrics about an application periodically and tries to determine its current state (whether is suffers from contention). If coupled with a profiling mechanism, online monitoring is used to compare current metrics with an established set of behaviors. When alone, online monitoring is agnostic to the ”ground truth” of each application. In this thesis, we consider as ”profiling concurrently to the execution” all actions made to discover an application's normal behavior or its interactions with specific, known co-runners that take place after its execution has started.

In the following subsections we examine mechanisms that include profiling to prevent, detect or moderate interference categorized based on the profiling method used. Most of those mechanisms include profiling as only the first step towards the end goal, and usually propose online monitoring and resource management techniques to maximize benefits.

3.3.1 Intrusive Micro-Benchmarks

Firstly, we will focus on approaches that, during profiling, apply controlled pressure upon shared resources so as to gain insight about the application's characteristics. Synthetic, tunable microbanchmarks are used to deliberately interfere with the application, and their performance effects on the latter are measured. In that way, the system not only has knowledge of the isolated execution, but also of the possible harm specific contention can create.

One of the most interesting employments of this technique is by Delimitrou et al. ([4]). Although Paragon performs profiling to also estimate the impact of hardware heterogeneity (and [5] extends it to resource scale-out (more servers) and scale-up (more resources per server)), we will discuss only the interference-related profiling. The authors suggest characterizing each application based on two criteria: its sensitivity to contention and its potential to create it. Several shared resources (sources of interference, SoI) are identified (memory bandwidth and capacity, L1/L2/L3 caches, TLBs, network and storage bandwidth) and a microbenchmark is created to target each resource. During profiling, an application is run with a specific microbenchmark, whose intensity is progressively tuned up until the application's QoS target (set to 95% of solo run performance) is violated. The microbenchmark's intensity percentage at this point is the application's ”tolerated interference” (TI) score. Similarly, the application is run with the microbenchmark as we intensify its pressure, until its performance degrades by 5% compared to its solo execution. Again, its intensity percentage at that point is the application's ”caused interference” (CI) score. A small set of applications is profiled as described for all SoIs.

Paragon classifies new applications using two collaborative filtering techniques: Singular Value Decomposition (SVD) and PQ-reconstruction (PQ) ([23]). We will not proceed into fully explaining those techniques, as this exceeds the scope of this thesis. The

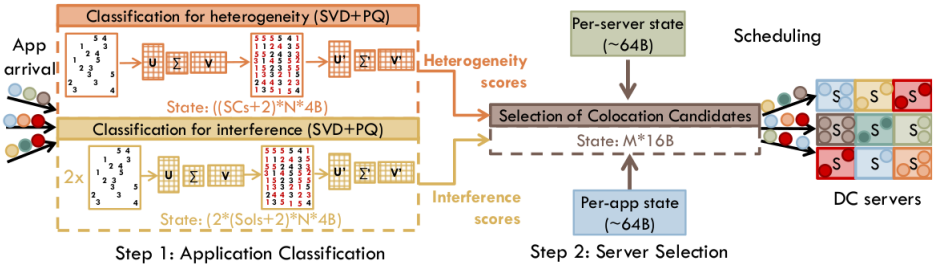


Figure 3.5: Paragon overview

process requires two utility matrices, one for TI and one for CI, that have applications as rows and SoIs as columns. When the previously presented offline profiling is completed, the matrices are populated with the collected scores, creating dense rows. In the online mode, when a new application arrives, it is profiled for 1 minute with two random microbenchmarks and its scores are added to the matrices. PQ-reconstruction and SVD are then used to fill the empty entries in the row and compute the confidence in each similarity concept. An example of a similarity concept can be ”application A and b both have a TI score above 60%”. Similarity concepts are represented by single values, and their magnitude signifies their confidence. In summary, collaborative filtering is used to classify applications in regards to their ability to tolerate and cause interference. This classification is then utilized by Paragon’s scheduler to schedule applications in servers so as to minimize interference.

As the microbenchmarks used in mechanisms like Paragon are usually in-house implementations, and are not easily created, El-Sayed et al. ([8]) introduce a different method of calculating applications’ sensitivity to interference online. DynaWay utilizes the Intel RDT technology previously mentioned to modify the available LLC capacity, creating thus artificial interference. More specifically, DynaWay periodically enters into profiling phase for a specific application. During that time, is divides the cache into two partitions: one for the profiled application and one for the rest. Then, it progressively subtracts cache ways from the first partition and adds them to the second, monitoring at the same time the effects on LLC misses, ipc and memory bandwidth. When profiling ends, the respective curves are created and used to make cache partitioning decisions. This method can potentially be used similarly to a synthetic microbenchmark, emulating contention on a shared resource so that the application’s sensitivity to interference is characterized.

3.3.2 Isolated Profiling

In this category we present mechanisms that only require the knowledge of an application’s normal, uninterfered execution. The value of such knowledge has been previously explained; it can be used to predict how the application may interfere with others and/or be compared with metrics gathered online to determine whether it suffers from contention.

A representative example of offline-profiled behavior being utilized to make perfor-

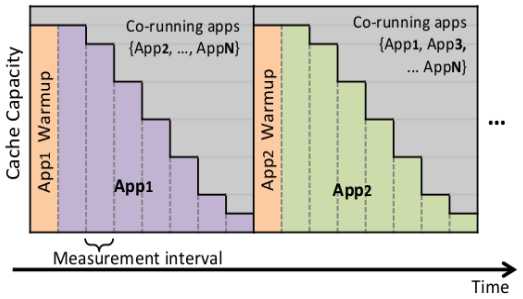


Figure 3.6: DynaWay's profiling phase

mance predictions is Dirigent ([30]). Zhu et al. again split applications into Latency Critical, LC and Best Effort, BE (in the paper different titles are used, but the definitions are the same) and try to minimize the performance variation of LC applications through fine-grained interference management. Dirigent's profiler is activated upon the arrival of a new LC application. The application is executed in an isolated environment and a series of (time, progress) pairs is periodically recorded, where progress is measured as the number of retired instructions during the past time segment.

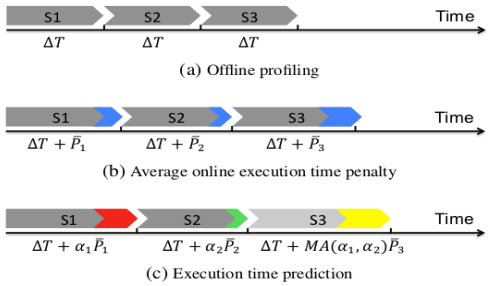


Figure 3.7: Execution time prediction phases

Then, the LC application is placed in a server along with BE applications, and its progress is monitored with the same time interval that was used during its offline profiling. For each time segment a time penalty is computed using the expected time to make the amount of progress within the offline profiled segment at the rate of progress experienced in the online monitored segment. Instead of utilizing only the time penalties computed, Dirigent attempts to increase accuracy by keeping an exponential moving average of the penalty within each segment across multiple executions of a specific LC application. When invoked, Dirigent's predictor uses the calculated penalty for each past segment along with the average penalties of the segments yet to execute and the total elapsed time since the beginning of the application's execution to estimate the total execution time of the application.

After the expected execution time of an LC application due to interference is calculated, it is used to determine whether any resource management actions need to be taken.

Since Dirigent aims to minimize performance variations (while satisfying QoS goals), and not execution time, if an LC application's expected execution time is smaller than its target time, resources will be allocated to BE applications. Similarly, if the expected time surpasses the target one, more resources will be allocated to the LC application. A fine-grained and a coarse-grained controller are responsible for managing the operating frequency of each core (and the execution suspension of BE applications) and the LLC partitioning respectively.

Another interesting approach is that of Novakovic et al. ([21]), who propose initiating application profiling in an isolated environment only after the application has started its co-execution with others in one of the production servers. The isolated execution uses real-time inputs to create a profile that can be directly compared to the interfered execution, so that performance fluctuations can be attributed to either normal phase changes or interference. DeepDive is comprised of three parts that cooperate to ensure LC applications are detrimentally affected by interference: a warning system, an interference analyzer and a placement manager (Figure 3.8).

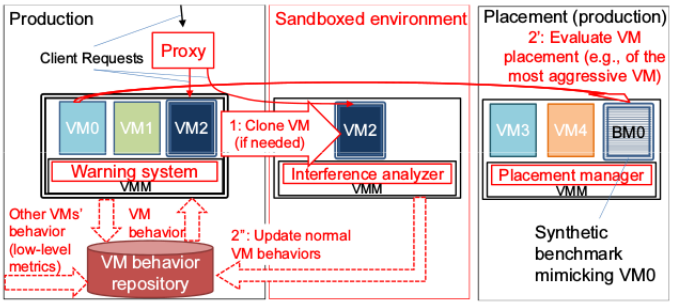


Figure 3.8: DeepDive overview

The warning system's role is to differentiate performance changes due to interference from those due to workload change. More specifically, it monitors a set of low-level metrics and compares the measurements of each LC application with a set of known behaviors it possesses. If there is no match within its archive, it searches in other servers for instances of the same application running (something that is typical for LC applications), and compares the two instances' behaviors. This utilization of global information is based on the notion that if threads executing the same code have the same performance changes, the latter probably are caused by workload changes and not interference. If no match is found with either local or global behaviors, the interference analyzer is invoked. The authors claim that invocations of the analyzer due to false positives pose minimal overheads to the overall mechanism. False negatives on the other hand are more impactful, and are handled using a vector of metric thresholds. Authors used a clustering algorithm on the acquired normal behaviors to produce said thresholds, which separate representative application performance from noise, while also properly identifying interference.

The interference analyzer used in DeepDive is based on a previously proposed technique described by Vasic et al. in DejaVu ([27]). Upon invocation, the analyzer clones

the examined application (which is being executed in a production server, as normal) in an isolated environment. It then intercepts the requests of the original application's copy, and forwards copies of them to the isolated clone. In that way, it establishes the normal, uninterfered behavior of said application under real-time workloads. The low-level measurements regarding isolated performance are sent to the warning system, along with the aforementioned vector of metric thresholds. Application degradation is computed as the ratio of retired instructions during interfered execution to that of the isolated execution, and if degradation is higher than an the operator-established threshold, the placement manager is activated. In this case, the analyzer uses the collected "isolation" metrics to compute stalls duo to contention in different shared resources, selects the resource that is introduces the most and informs the manager accordingly. The placement manager then makes scheduling decisions regarding the application that is more aggressive with the shared resource where interference is detected.

During evaluation, DeepDive required about a day of operation to capture all normal behaviors, during which the false positive rate was high. After that period, the rate dropped to nearly zero, showing that DeepDive can successfully differentiate interference from workload changes. In addition, no false negatives where detected, and the analyzer's degradation estimation is within 10% accuracy in the worst case.

As we mentioned before, the idea of concurrently running an application in the production and the isolated environment with the same workload was introduced in [27]. However, Vasic et al. use this kind of profiling not to differentiate workload fluctuations from interference, but to determine resource allocations that satisfy performance goals. They specifically target request-and-response applications, that are user-dependent and experience significant variability in their inputs, so an approach aiming to capture the behavior during different workloads is reasonable. In addition, DejaVu is among the first mechanisms that leverage machine learning techniques to solve the resource allocation and interference problem.

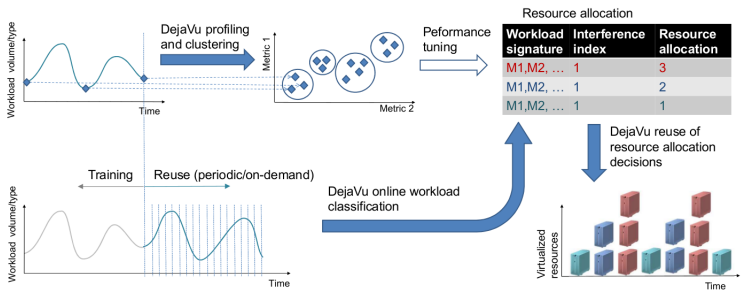


Figure 3.9: DejaVu overview

When the training phase begins, DejaVu profiles an application both in its production and its isolated environment, using a proxy that duplicates requests as described above. Profiling is performed for a certain period (e.g. a week), until the administrator decided that a representative set of workloads has been captured. N different low-level metrics are monitored during profiling, and are used to compose a per-workload signature, a N-tuple

that is representative of the workload's behavior. K-means clustering is then performed using all workload signatures of an application, and workload classes are created. The workload that is closer to each class's centroid is selected as representative, and a tuning process is triggered.

During this stage, tuning is performed for each workload class. A tuning mechanism is responsible for determining the resource allocation that is sufficient to meet QoS targets of the application under each representative workload, without being wasteful. When tuning is completed, DejaVu creates a lookup table containing (representative workload signature, interference index, optimal resource allocation) triplets for each application, where interference index is equal to 1 for all entries (its purpose is later explained). This resource allocation will be referenced as baseline.

Furthermore, a decision tree classifier is trained to place new incoming workloads during production execution into the best fitting class. Trained with all the gathered workload signatures and their corresponding classes (as denoted by k-means), the classifier can determine the class of an unknown workload based on its signature. When a new workload begins, its signature is created. Although it is not stated by the authors, we assume that the signature is produced through the isolated profiling mechanism. The classifier then assigns the new workload to a class and the corresponding baseline resource allocation is applied.

In case the QoS goal is still being violated after applying the optimal resource allocation for a workload, DejaVu assumes interference is to be held responsible. The current interference index of the workload is then computed as the ratio of production to isolated performance, and the lookup table is queried to find a matching entry. If there isn't one, tuning is triggered, and a new entry with the best resource allocation for this (representative workload signature, interference index) is added, and can later on be reused.

Dwyer et al. ([7]) also employ machine learning techniques, but this time for performance degradation estimation. Their main idea is to train a model that can predict the future performance of an application using measurements gathered online. The benefit of such approach is that, although it does require offline profiling of a significant number of scenarios explained below, this is a one-time overhead; once the prediction model is trained, no offline profiling is necessary for new application coming.

To create the training set, the authors selected a set of HPC applications created the following execution scenarios for each one of them: a solo run (the primary application runs alone), a clean run (the primary application runs with copies of itself) and several random runs (the primary application runs with other, randomly selected applications from the set), creating over 500 scenarios. The authors chose to split each scenario execution into 5-billion instructions windows called execution instances, and train the model based on instances and not on complete applications runs. The duration of the instances was selected to give the system enough time to gather all the 340 different low-level metrics available in their system. All the scenarios where then executed, measuring all the low-level metrics in each execution. For each scenario there is a primary application (the one whose performance degradation we study) and its co-runners. For each instance in each scenario the degradation of the primary application is calculated, using the duration in

clock cycles of the scenario instance and its respective solo run instance. As a result, each instance is characterized by a set of 340 attributes for each one of the applications it includes, plus the degradation value.

Before training the model, attribute selection was performed to eliminate unnecessary attributes, reducing their number from 340 to 19. In addition, for the distinction between primary and co-running application to be made, the measurements of the co-running applications are averaged. The final training set is comprised of thousands of instances, each one characterized by the measurements of its primary application, the average of the measurements of the co-runners, and the degradation value. The model selected was a regression tree, and bootstrap aggregating was also used to improve accuracy. The trained model was evaluated with cross-validation, using error rate (difference between the estimated and the actual degradation) as the accuracy metric. The average error rate is 16%, with 80% of the error rates being under 20%. To eliminate cases where the error rate is very large due to outliers, the authors create a confidence predictor. If two or more attributes of the to-be-predicted instance are more than two standard deviations away from the mean of the training measurements, the predictor outputs a null prediction, marking the instance as non-confident.

This predictor is coupled with a scheduler that tries to maximize resource utilization without violating performance goals. All cores in a server are filled in a best-fit policy and the necessary low-level metrics are constantly monitored and used to estimate degradation. If the latter exceeds an established threshold, the scheduler migrates the suffering application.

3.4 Conclusions

The proposals examined in the previous sections are only some of the ways designers have tried to tackle the interference issue. It has now become evident there probably isn't a method or a a combination of methods that is a "one-fits-all", performing optimally in every scenario, under all application types and with no overhead. All decisions bare advantages and disadvantages, and designers must carefully enorchistrate mechanisms whose overheads are balanced by the profits yielded.

A technique that has significant trade-offs is profiling. Conducting controlled profiling requires one or more isolated servers that can no longer be used to host multi-application workloads, essentially reducing the computational power of the datacenter, and as a consequence its profits. Also, a-priori profiling is time consuming, and given the large number of applications arriving at a datacenter, profiling them all would impose a major delay to their execution, again increasing costs for the provider. Although making sophisticated decisions before execution begins can prevent interference, in the cases of non-interfering applications, precious time and resources have been wasted for a workload whose performance is already satisfactory. Concurrent to execution profiling as we have described it might not require the extra time a-priori profiling does, but it still requires isolated servers, and no choices can be made before execution begins.

The big advantage of profiling is that, if carefully articulated and optimized, it can cancel the overheads it creates by the performance gains it offers. In environments where tens of applications share one server, locating the contentious ones can be very challenging due to the large amount of interactions present. Indeed, being able to completely prevent interference or take actions to control its impact before it becomes harmful can prove extremely helpful for a commercial cloud. As we have previously described, in the pricing model we examine cloud costumers are charged only when the performance goals they have set are met. Consequently, executing applications that do not satisfy their QoS goals is unprofitable.

The primary challenge with proposals that do not conduct profiling is actually detecting interference, and separating it from application phase or workloads changes. As the isolated execution measurements of the application are unknown, and most applications are comprised of more than one execution phases and have multiple or dynamic workloads, recognizing performance decreases due to contention is demanding, as it has become evident from the works previously described. Phase-detection mechanisms ([6],[3],[20]) can perhaps be utilized, but they are often computationally expensive and introduce prohibitive overheads.

Another parameter that needs to be taken into consideration is the actual complexity of the mechanism, and the granularity at which it makes decisions. Having a system that operates at a very fine granularity, offering highly customized policies in regards of scheduling and resource management, tailored to a specific combination of applications, seems very lucrative. Such an approach would probably utilize resources optimally, but have an excessive cost of operation, probably requiring a large amount of low-level measurements, profiling and frequent monitoring. As the amount of metrics needed or the frequency at which they are gathered increases, application performance is also affected.

Chapter 4

Application Classification for Interference Prevention

In the previous chapter several approaches and design trade-offs where discussed, to showcase just a portion of the design space system engineers face. In this chapter, we attempt to investigate in practice the impact of co-execution and interference, and present our approach to the problem.

4.1 System Configuration and Benchmarks

All the executions presented in this thesis where run on a Intel® Xeon® Processor E5-2630 v4, whose specifications are listed in Table 4.1.

This family of processors features the Intel Resource Director Technology (RDT), which provides the user with the ability to monitor performance metrics and manage resource allocation. The Cache Monitoring Technology (CMT) allows the user to dynamically observe a number of low-level metrics, such as the LLC misses and occupancy, at the granularity of a logical core. CAT is utilizing the four underlying registers that Intel technology offers for event counters monitoring to report up to four metrics: ipc (instructions per cycle), LLC mpki (misses per kilo instructions), LLC occupancy and DRAM bandwidth. Because in our experiments we wanted to monitor as many performance counters as possible in one run, but did not want to opt for sampling techniques, we decided to deactivate hyperthreading in our processor, which makes another four registers available for events' monitoring, adding up to eight events monitored per run. Additionally, we modified the code of the API (called PQoS) so that it can monitor all the desirable events. We also took measurements using the linux perf command to verify that they match those reported by PQoS. Because perf uses switching between monitored events and sampling, it's results contained more noisy that those of PQoS, but where for the most part identical. All the reported measurements from now on where gathered using PQoS.

Architecture Family	Broadwell
Processor Base Frequency	2.20 GHz
Number of Cores	10
Number of Threads	20
L1 (data) Cache (per core)	320 KB
L2 Cache (per core)	2.5 MB
Last Level Cache (shared)	25 MB, 20-way
DRAM Bandwidth	68.3 GB/sec

Table 4.1: Intel® Xeon® Processor E5-2630 v4 specifications

In addition to CMT, Intel RDT also offers Cache Allocation Technology (CAT), with which the user can partition the LLC into sets of ways, and assign those sets to groups of cores. Later processor model also feature Memory Bandwidth Monitoring Technology (MBM) and Memory Bandwidth Allocation Technology (MBA).

The benchmarks used in our experiments are mainly from the SPEC 2017 Suite, with one addition from the Polybench 3.2 Suite (jacobi-2d benchmark), as well as the stream and hpcg benchmarks. The SPEC 2017 benchmarks are divided into two categories, that differ mainly in the input sizes and the memory footprint: rate and speed. Some of the benchmarks have implementations in both categories, so to avoid confusion we denote the rate version with ”_r” and the speed version with ”_s”. Furthermore, the Alberta Workloads ([1]) were also used for some of the benchmarks. When the input of benchmark is one of the Alberta ones, its name is added to that of the benchmark. For example, omnetpp_r_star is the rate version of the omnetpp benchmark with the ”star” input (from the Alberta Workloads), whereas omnetpp_r is the rate version with the original SPEC 2017 input. Our final set has in total 140 benchmarks.

4.2 Co-Execution Scenarios

Firstly we run each application on each own, to capture its solo behavior. Then, we create 2-application scenarios. As we want to examine interference in resources that are shared across the chip, each application is single threaded and pinned to a specific core (to eliminate interference in core-private components such as the L1 cache). If one application finishes execution before the other, it is restarted. This continues until all applications are executed at least one time. To avoid executing all 9.730 possible pairs, for our initial analysis we choose only the rate implementations with the SPEC 2017 inputs, as well as the hpcg, stream and jacobi-2d benchmarks. As performance indicators we choose IPC

and total execution time, and to measure interference we define Sl (Slowdown and Deg (Degradation) as:

$$Sl = \frac{t_{coexec}}{t_{alone}}$$

$$Deg = \frac{ipc_{alone} - ipc_{coexec}}{ipc_{alone}} * 100\%$$

where:
t_{alone}: total execution time when run alone
t_{coexec}: total execution time in the co-execution scenario
ipc_{alone}: ipc when run alone
ipc_{coexec}: ipc in the co-execution scenario

Figure 4.1 shows a typical scenario where interference impacts performance. Omnetpp_r exhibits a slowdown of 1.23, with a 19.7% Deg, even with just one co-running thread. In Figure 4.2 we can see that lbm_r dominates omnetpp_r in the LLC (leading to almost three times higher LLC mpki).

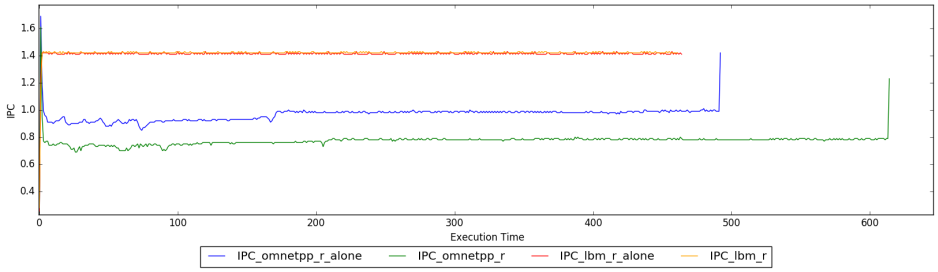


Figure 4.1: IPC, scenario: 1 omnetpp_r with 1 lbm_r

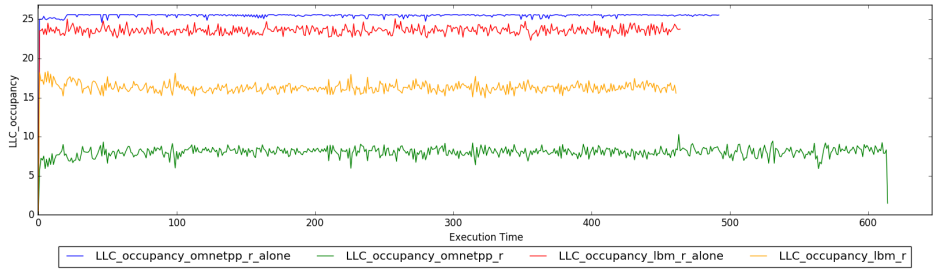
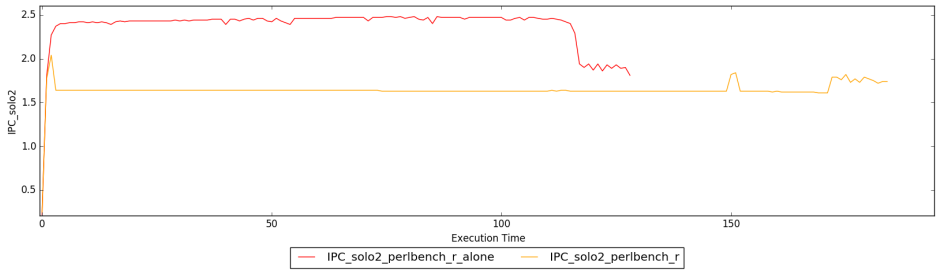
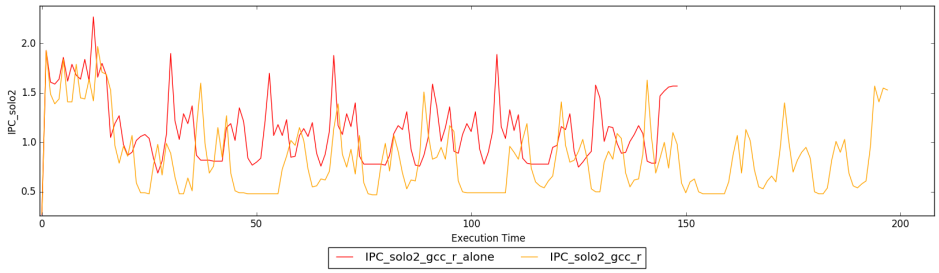


Figure 4.2: LLC occupancy, scenario: 1 omnetpp_r with 1 lbm_r

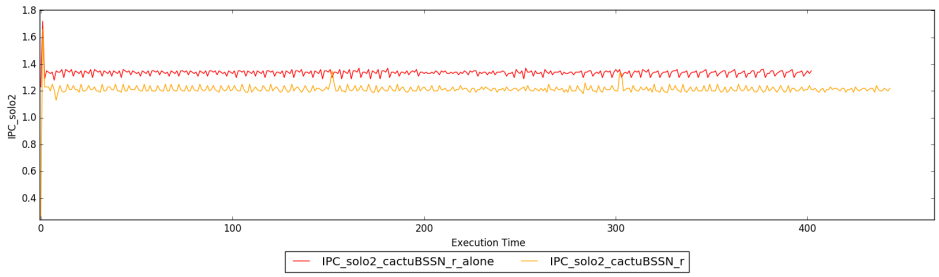
When examining closer how different applications interact when co-scheduled, one can begin to see a pattern: there are applications that exhibit the same behavior regardless of their co-runner. In Figure 4.3, we see stream executed with perlbench_r, gcc_r, cactuBSSN_r and blender_r. In all four cases, stream appears to negatively impact its co-runners performance to various degrees, while always taking over the LLC.



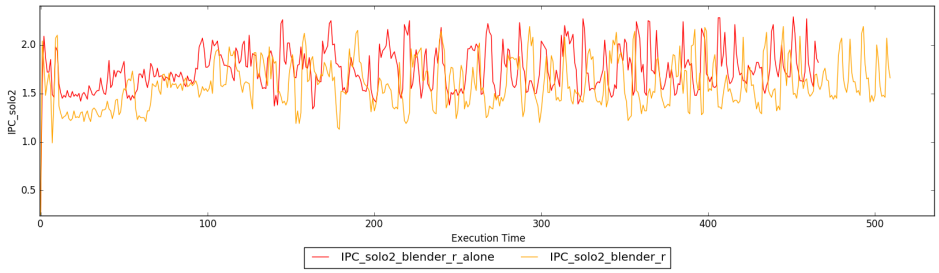
(a) 1 stream with 1 perlbench_r



(b) 1 stream with 1 gcc_r



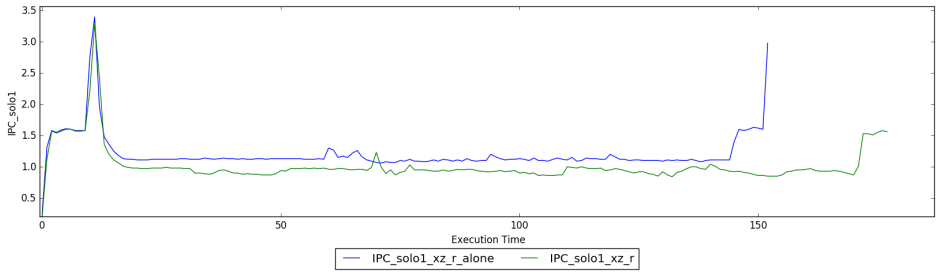
(c) 1 stream with 1 cactuBSSN_r



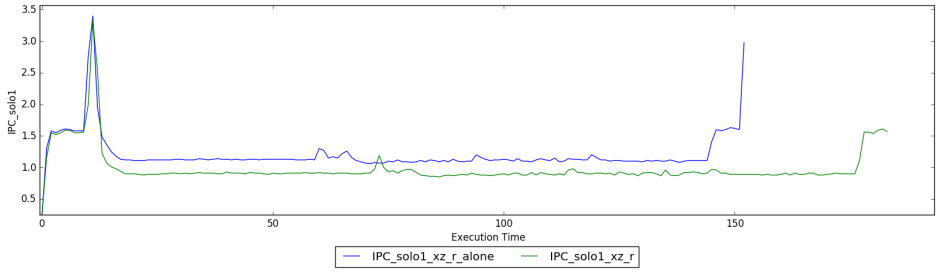
(d) 1 stream with 1 blender_r

Figure 4.3: IPC of various stream's co-runners

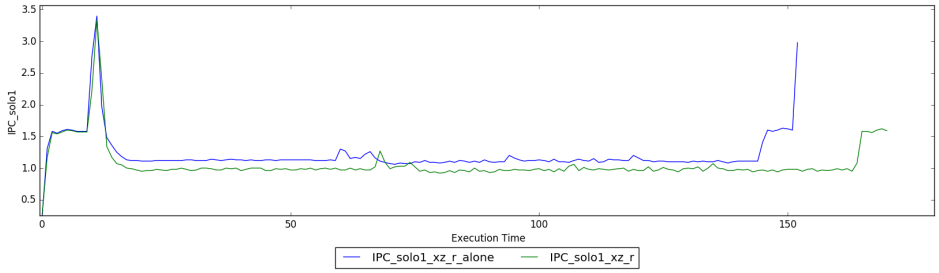
In Figure 4.4, we see xz_r's ipc when executed with mcf_r, jacobi-2d, cactuBSSN_r and cam4_r. In all cases, xz_r experiences different degrees of performance degradation.



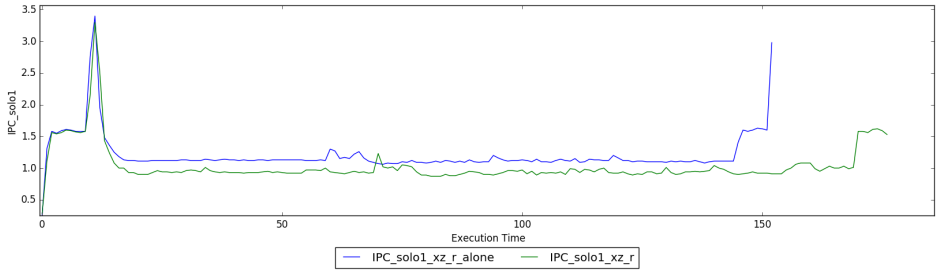
(a) 1 xz_r with 1 mcf_r



(b) 1 xz_r with 1 jacobi-2d



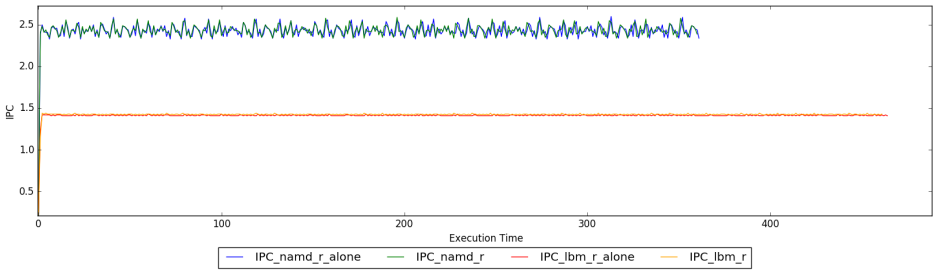
(c) xz_r with 1 cactuBSSN_r



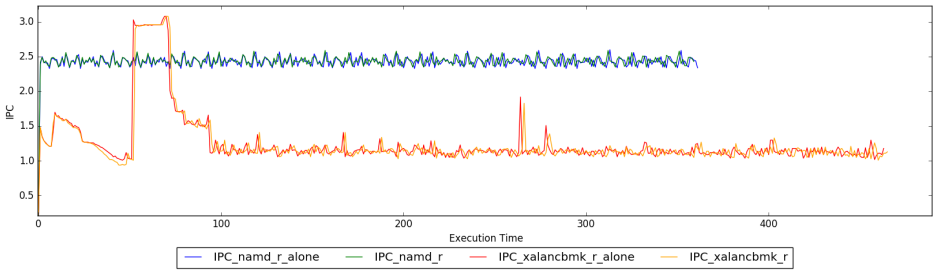
(d) 1 xz_r with 1 gcc_r

Figure 4.4: IPC of xz_r in various scenarios

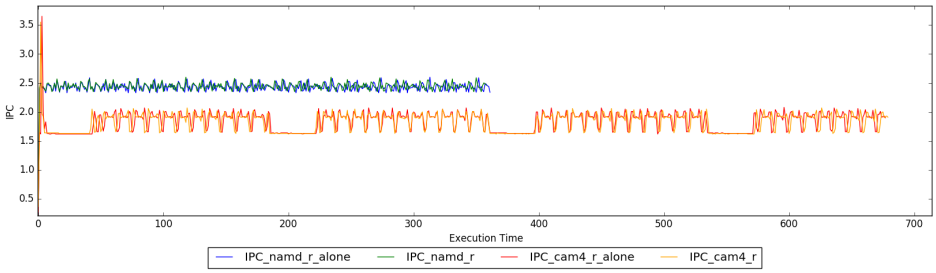
Lastly, in Figure 4.5 we present the case of namd_r. Interestingly enough, namd_r is not only highly resistant to any interference (no changes in its ipc), but also does not create any contention (no significant changes in its co-runners ipcs).



(a) 1 namd_r with 1 lbm_r



(b) 1 namd_r with 1 xalancbmk_r



(c) 1 namd_r with 1 cam4_r

Figure 4.5: IPC of namd_r and its co-runners in various scenarios

The above examples are a strong indicator that whether an application will experience and/or create performance degradation in a scenario might be inherent characteristics of the application itself, and not dependent on the application mix in the scenario. This observation has also been made and verified by Tang et al. in [26] . If these characteristics can be correlated to low-level metrics (Performance Monitoring Units, PMUs), such as the LLC mpki, then one can predict them without placing the application in a possibly detrimental for its performance co-execution scenario, and make useful decisions in regards of its placement in a server.

4.3 Noise and Sensitivity

Noise (often referred to as ”contentiousness”) and *sensitivity* are two terms that have been used in bibliography ([26]) to describe how much an application suffers from or can create performance degradation in multi-application scenarios. In general, an application is considered **noisy** if it results in significant degradation of its co-runner's performance, and **quiet** when it leaves its co-runner completely unaffected. Similarly, we call an application **sensitive** when its performance is constantly affected in the presence of a co-runner, and **insensitive** when it almost never is.

Prior work has been controversial about whether sensitivity and noise are correlated. Jiang et al. ([18]) conclude that there is a correlation, and applications are either sensitive and noisy or insensitive and quiet, whereas other works ([14],[31],[26]) find cases of other combinations, such as noisy and sensitive. In our work, we consider the two characteristics not correlated, meaning that an application's level of noise does not necessarily determine its level of sensitivity and vice versa. Noise and sensitivity reflect two different aspects of an application's behavior: how much it uses a shared resource and how much it benefits from it. Usage does not always mean benefit. Resources that act as performance optimization mechanisms, such as the LLC, fall under that case, as their effectiveness relies upon the application's data patterns. For example, if an application has no reuse pattern in its data, then it constantly brings new entries in the LLC without ever reusing them. In that case, it creates high contention for its co-runners, constantly evicting their entries, but does not suffer itself if its data gets evicted, since it wouldn't reuse them anyway.

4.4 Previous Work on PMU-based Classification

Several attempts have been made to classify applications in regards of their noise and sensitivity based on performance counters. We present below the most representative ones.

Lin et al. in [16] classify applications into four different colors depending on the slowdown they experience when run with 1/4 of the LLC compared to when they run with the whole LLC. Qureshi and Patt in [22] study the performance of applications when run with various portions of the LLC, and divide them into *high-*, *low-* and *saturating-utility*. Both of those works address only the subject of sensitivity and not that of noise, and examine only the LLC as a shared resource. In addition, they require multiple runs of an application, which is unrealistic in production environments.

Xie and Loh introduce an animalistic classification ([28]) . An application can either be a *turtle* (low use of the LLC), a *sheep* (low LLC miss rate, insensitive to the amount of LLC it is allocated), a *rabbit* (low LLC miss rate, sensitive to the amount of LLC it is allocated) or a *devil* (high LLC miss rate). The authors employ a mechanism proposed in [22] to dynamically measure the least amount of LLC an application needs to achieve an acceptably low LLC miss rate (expressed in relation to the LLC miss rate of the solo run). Applications are placed into a category according to their respective values for LLC accesses, LLC miss rate, LLC misses and the aforementioned LLC amount (expressed in ways). This approach however faces some limitations expressed in [31] . For example, Xie

et Loh state that *devils* are applications with high contentiousness due to high LLC miss rates and low sensitivity, as they exhibit low data reuse. But because their classification takes into consideration only LLC-related metrics, they fail to account for contention and sensitivity in other resources, such as the DRAM bandwidth. In [31], the authors show that some applications that show high LLC miss rates and would classify as *devils* are highly sensitive when it comes to other resources, like DRAM bandwidth and prefetchers.

Zhuralev et al. ([31]) propose a *pain* based classification scheme, defining *pain* as the product of sensitivity with noise. To calculate sensitivity they use stack distance profiles, whereas for noise the LLC accesses per million instructions are used. They also implement a scheme that is based only on *LLC miss rate*, supporting the case that the latter is sufficient in making scheduling decisions. Although their *pain* scheme slightly outperformed the *LLC miss rate*, they continue on to use the latter in their contention-aware scheduler, as it is much less complex to implement and performs almost the same.

Lastly, in [26], Tang et al. attempt to predict an application's noise and sensitivity using linear regression. To correctly account for both characteristics, and shared resources other than the LLC, they propose using as performance counters the LLC_lines_in/ms for memory bandwidth usage and the $(L2\_lines\_in - LLC\_lines\_in)/ms$ for LLC usage. Their conclusions are rather interesting: although they were able to create a linear model using regression to predict an application's contentiousness, a similar model could not be created for sensitivity, which proved to be much more complex to identify based on their performance counters.

4.5 Designing a Non-Intrusive, Lightweight Classification Algorithm

As we have already mentioned, when it comes to addressing interference all choices have advantages and disadvantages, and each designer is called to decide which choices are worth the overhead and which not. We wish to create a classification mechanism that bares the following characteristics:

- **Preventive**: Contrary to other approaches, we want to be able to prevent interference from happening, not detect it after it happens. Having an estimation of how applications will interact before they actually do is of high value, especially in cases where one of them has strict performance goals.
- **Lightweight**: The amount of PMUs measured must be such that it will not add overheads during execution. Also, we do not wish to maintain large databases or other models (e.g. neural networks). Finally, our solution must be as computationally inexpensive as possible.
- **Non-Intrusive**: Because we wish to use profiling, a necessary small overhead has already been added to out mechanism. However, since profiling will be done (if necessary) upon application arrival in one of the system's servers, we want the time

it spends there to be ”useful”: its performance must not be affected in any way, so we cannot use any intrusive benchmarks or cache allocation techniques to observe how performance is impacted. This is a constrain we decide to pose since our mechanism is targeted to commercial cloud environments, where profit is proportionate to an application's ”useful” time.

4.5.1 Defining the Classes

Taking into consideration the previous constrains, we decided to adopt a reverse-engineering approach: first categorize our applications according to the behavior we observe in co-execution scenarios, and then see if any combination of PMUs can discern between categories. Determining whether an application is e.g. noisy or quiet is not straightforward, even if one has abundant resources and time to examine applications, as those terms are loosely defined. Sensitivity in regards of the LLC can be observed by executing an application with various amounts of LLC ways, and then analyzing how its performance was affected (similarly to [22] and [8]). To that extent, we used Intel CAT to run each application with 1 up to 19 ways, and created an ipc-cache ways curve for each application. Applications whose performance continued to improve as the cache ways increased were labeled as *sensitive*, and those who after 1-2 ways showed no performance as *insensitive*. All other applications exhibited a performance saturation after a certain amount of ways (different for each application), and were labeled as *potentially sensitive*, meaning the could suffer from contention under certain pressure, but not always. In a machine that also supported Intel MBA we could have performed a similar analysis for memory bandwidth, but since we do not have that ability in our current processor, we had a different approach. Bandwidth is on of the shared resources where usage most of the times means benefit. Even if the data brought from memory are a result of prefetching and end up not being useful to the application's execution, the data are still fetched, so if bandwidth is dominated by someone else the application will suffer long stalls, and thus performance losses, waiting for the requested data. To test if some of our labeled as insensitive and potentially sensitive applications have been misplaced, we examine their co-executions with an in-house microbenchmark that saturates memory bandwidth. Indeed, applications like stream and jacobi-2d, which do not seem to have performance gains from increasing LLC capacity, are affected when contention in the memory bus is extreme, and thus we move them to the potentially sensitive category.

Noise is more difficult to define. Some benchmarks were very easily labeled when we observed the results from their co-executions, as they exhibited the same behavior regardless their co-runner. Lbm_r, stream, hpcg and others consistently caused a Deg of at least 20%, and so we label them as *noisy*. Other applications, like namd_r, leela_r and exchange2_r always left their co-runner unaffected, so we label them as *quiet*. However, most of the applications did not exhibit a striking behavior, so we decided to establish a ”reference benchmark”, one that would be ”in the middle” in both categories. Luckily enough, cactuBSSN_r showcased the necessary characteristics: it had median or very close to median values in all PMUs. We consider it to be a fitting reference since it is nei-

ther insensitive to interference nor extremely sensitive, neither contentious nor completely quiet. Our strategy was to run all benchmarks with cactuBBSN_r and observe the degradation they caused it. If $Deg > 10\%$, we label the benchmark as noisy, whereas if Deg is almost 0 we label them as quiet. Not to our surprise, all insensitive applications where also quiet. Since applications in the insensitive category need minimum cache capacity and memory bandwidth, we expect them to not access shared resources to a large extent, and thus not create problems for their co-runners. Every other benchmark was labeled as *potentially noisy*.

Although a Deg of 10% might seem rather small, we would like to underline the fact that we are discussing 2-application workloads. The caches and bandwidth in our processor are significantly large, so they most likely can handle the majority of 2-application scenarios. In real production environments, all cores are used, so an interference that might seem negligible in a 2-application scenario is highly exacerbated in a 10-application scenario, as we will see in later chapters.

The thresholds for Deg mentioned below are empirical and based on averaged values from our experiments, and we do not claim that they are universal or inerrant. However, we observed that they are capable of correctly capturing the vast majority of cases, so we consider them adequate for the purpose of this thesis. In the cases where Deg was marginal, we examined more co-runs to select a category. Our general strategy was to be conservative, and if in great doubt place an application in the potentially noisy/sensitive class.

4.5.2 PMU Patterns

After positioning our benchmarks in categories, our first step was to examine whether different PMUs could indicate sensitivity and/or noise. We run each application in isolation collecting all the metrics listed in Table 4.2 using our modified version of the PQoS API. We also collected information about the LLC occupancy and DRAM bandwidth using the built-in functions of PQoS. To select the metrics we would analyze, we took into consideration the observations made in [25] and [19], along with our observations regarding the memory subsystem.

Apart from the raw PMUs data, we combined PMUs to create new metrics, such as miss rates.

As we mentioned before, applications that maintain their data in private to each core parts and do not greatly use shared resources, are quiet and insensitive. This can be translated into having low values for LLC accesses per kilo instructions (LLC acpki) and memory bandwidth. In Figure 4.6 and Figure 4.7 we present the LLC acpki for a set of representative benchmarks. Insensitive and quiet benchmar5ks have a LLC acpki around 1, significantly lower than all others.

A rather strong indicator can be memory bandwidth. Applications that highly use the memory bus during solo execution, will continue to do so in co-execution scenarios, and are most probably noisy. As we see in Figure 4.8, noisy applications do indeed have much higher memory bandwidth values.

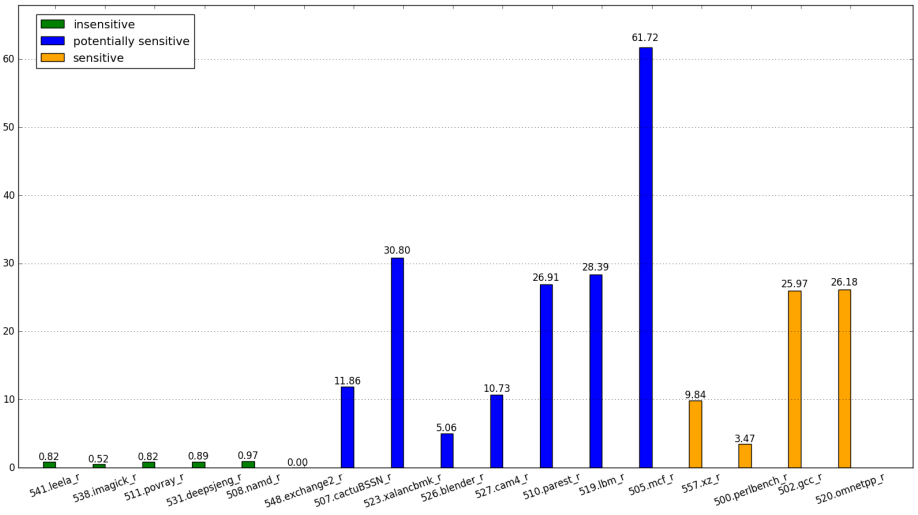


Figure 4.6: LLC acpki
Benchmarks labeled according to sensitivity

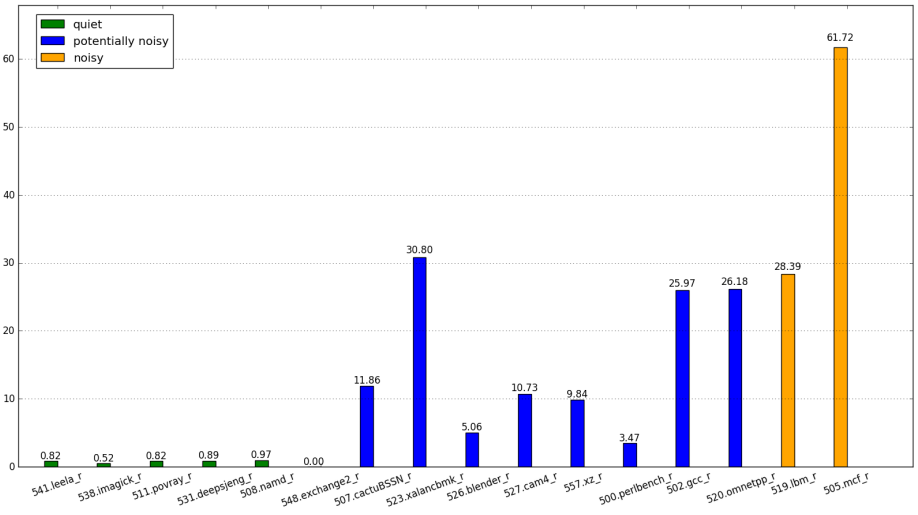


Figure 4.7: LLC acpki
Benchmarks labeled according to noise

We also considered the methodology described in [19] to detect memory boundness. Molka et al. suggest that stall-related counters could reveal if an application's performance heavily depends on the LLC and memory. If a large fraction of the cycles an application spends halted (*CYCLE_ACTIVITY.STALLS_L2_PENDING*) is due to pending requests to lower levels of memory (LLC + DRAM), it means it cannot overlap waiting for those requests with useful computation. As a result, extra misses in the LLC or congestion in the bandwidth will result in more stalls that cannot be "disguised" by computation, making

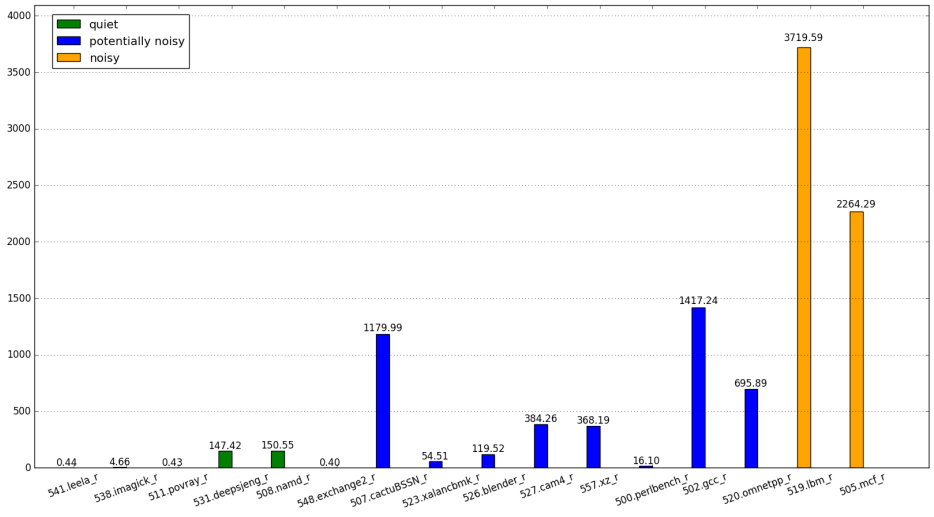


Figure 4.8: DRAM Bandwidth
Benchmarks labeled according to noise

stalls a promising indicator of sensitivity. Again, there didn't seem to be any clear pattern that could be outlined in Figure 4.9. It is worth noting that mem stalls (stalls because of a pending request to the main memory) might be higher than L2 miss pending stalls (stalls because of a pending request from L2) because of L1 prefetchers.

Two PMUs that are widely considered strongly related to noise and sensitivity are LLC mpki and miss rate (misses per access). Many prior works ([14],[31],[26]) have reasoned both in favor and against LLC mpki accurately describing application behavior. A brief look to Figure 4.10 shows that there are cases LLC mpki can be misleading. For example, many potentially sensitive and sensitive applications have LLC mpkis similar to those of insensitive applications. That is because an application might have a data set almost the size of the LLC; in that case, the LLC mpki in the solo run would not be high, but any co-runner that stresses the LLC would result in a performance drop. We do see that noisy applications show higher LLC mpkis, but the difference is not that large to safely draw any conclusions. Miss rate is not that useful either, as we see quiet applications having large miss rates. In those cases, the applications rarely access the LLC, so they are not affected by co-runners, but those rare accesses happen to be mainly misses, resulting in high miss rates.

Lastly, we examined if individual application performance can be correlated with PMUs. A strong correlation between the ipcs of one category's applications and a PMU, e.g. LLC mpki, could suggest that for this category, performance can be "translated" into low lever metrics, that could later be used to characterize the category. We used Pearson's Correlation Coefficient ([2]), but found no specific pattern in the way ipc correlated with various PMUs.

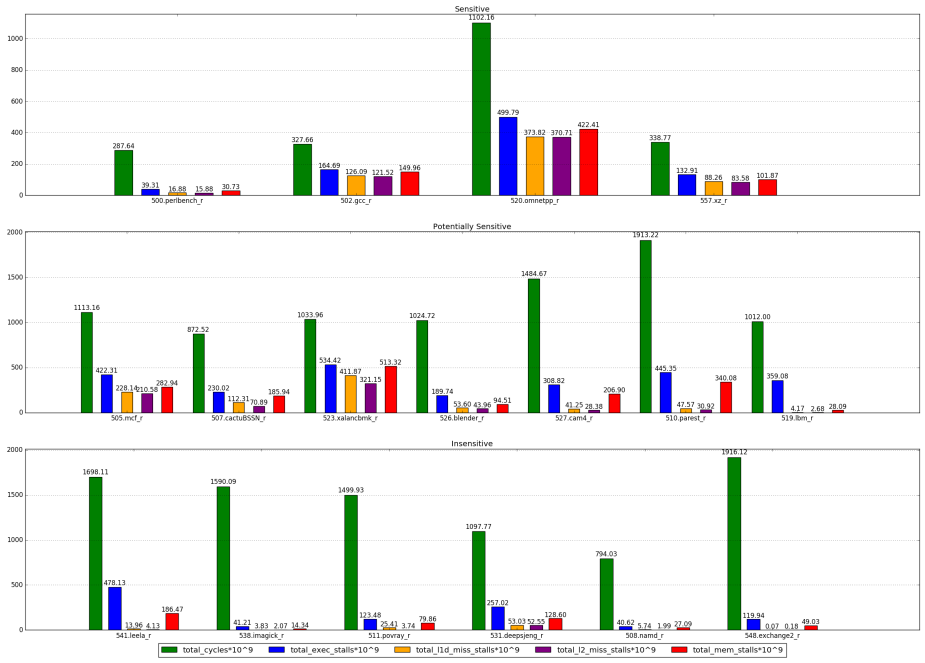


Figure 4.9: Stalls due to data requests
Benchmarks labeled according to sensitivity

After the aforementioned analysis, we can sum up our conclusions in the following points:

- There seems to be no easily detectable pattern in PMUs that can be used differentiate categories. If such a pattern exists, it is too complex for the human eye to detect.
- In the cases where some categories are discernible, (e.g. high memory bandwidth in noisy applications), it is unclear where the thresholds between classes should be placed, and we think that empirically setting them would be arbitrary.
- Some metrics yield interesting insights into application behavior and can perhaps be used as part of a classification algorithm.

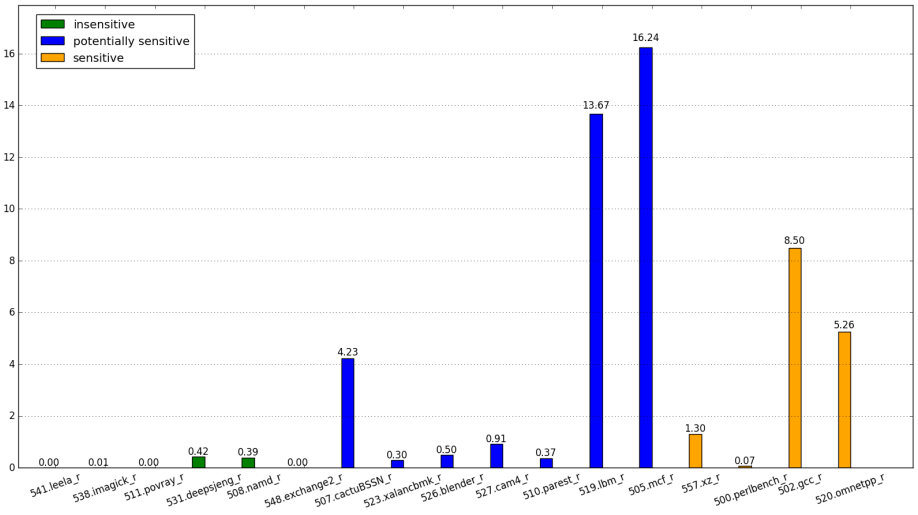


Figure 4.10: LLC mpki
Benchmarks labeled according to sensitivity

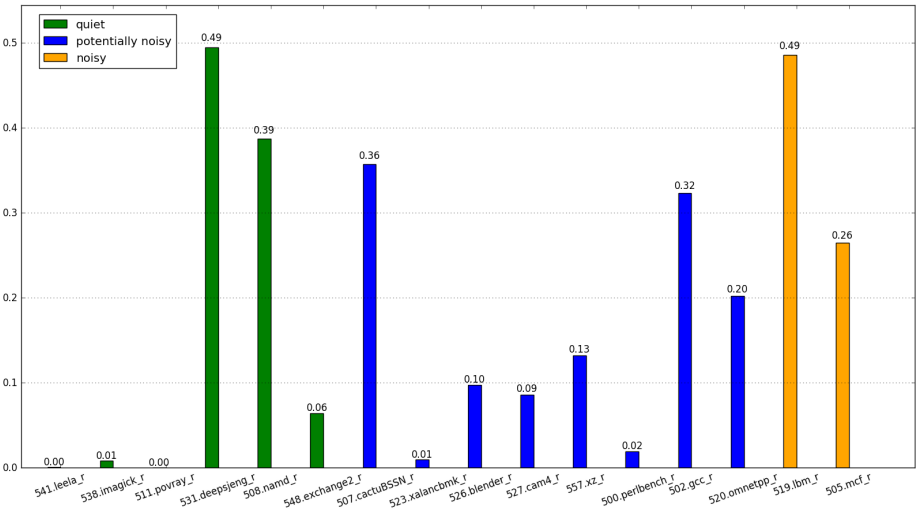


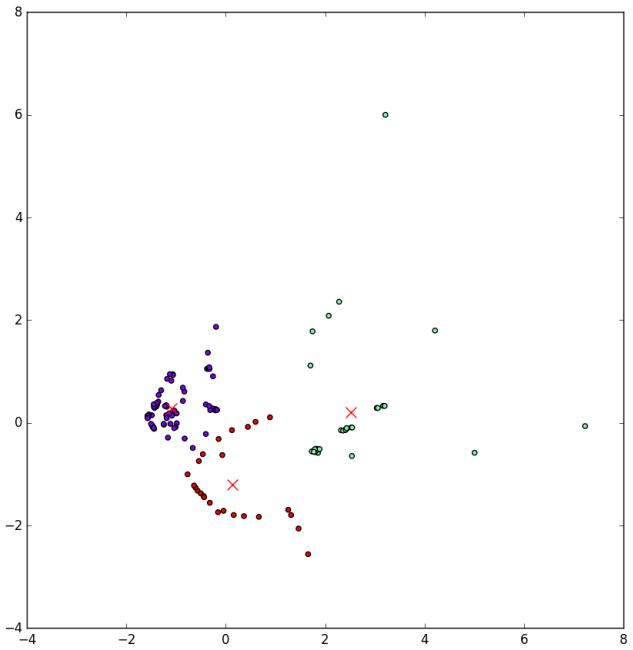
Figure 4.11: LLC miss rate (misses/accesses)
Benchmarks labeled according to noise

4.5.3 K-Means Clustering

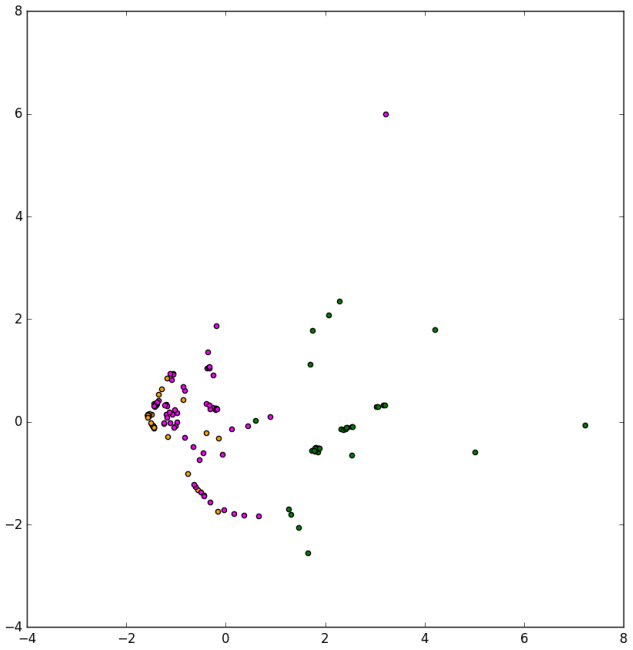
Although an empirically created classification algorithm could not be created, the fact that the few PMU patterns we could detect agreed with our assumptions about the corresponding parts of memory and application behavior led us to consider pairing PMUs and our insights with already established algorithms. K-means ([12]) is a well-known clustering algorithm used to split N observations into k clusters. Given an initial set of k means (centroids), the algorithm alternates between two steps. It assigns each observation to the cluster whose mean has the least squared Euclidean distance (the ”closer” mean), and then re-calculates each centroid as the mean of the observations in the respective cluster. K-means can be applied to observations that have more than 3 dimensions, also called features. Each observation has a specific value for each feature in the feature set. Our goal was to see if k-means could find the borders between classes, e.g. the threshold of memory bandwidth above which an application can be labeled noisy.

We begun our experiments with noise, as our observations suggest that borders between noisy, potentially noisy and quiet applications might actually exist. In our experiments we used the implementation included in the scikit-learn python framework. Standardization was applied to all data using the StandardScaler method of the python sklearn.preprocessing package. Standardization (or Z-score normalization) is a feature scaling method that rescales the values of a vector to have zero-mean and unit-variance. Its use was necessary to avoid biases because of differences in the PMUs units. For example, memory bandwidth can have values of up to 12.000 MB, whereas LLC mpki is typically around 5-10. The algorithm was tested with feature sets containing all possible combinations of the PMUs most strongly related to noise: LLC acpki, LLC mpki, LLC miss rate and memory bandwidth. Figure 4.12 and Figure 4.13 shows two representative examples of how the algorithm creates clusters, projected into 2-d space using Principal Component Analysis (PCA), a dimensionality reduction algorithm. We immediately see that noisy applications, such as stream and hpcg, can quite effectively be separated from the others. However, potentially noisy and quiet classes are more difficult to tell apart, as their respective applications seem to overlap. The results where similar when we examined sensitivity. This time, the features used by the algorithm were LLC acpki, LLC mpki, LLC miss rate, memory bandwidth, total L2 stalls and total L2 stalls/total cycles. Here, borders are even more blurry (Figure 4.14) confirming previous work that sensitivity is more difficult to detect than noise ([26]).

Our conclusion is that k-means succeeds only partially because it assumes separable, ”spherical” clusters, which clearly is not the case for all classes. The clusters are also expected to be of similar size, which also does not apply in out dataset.

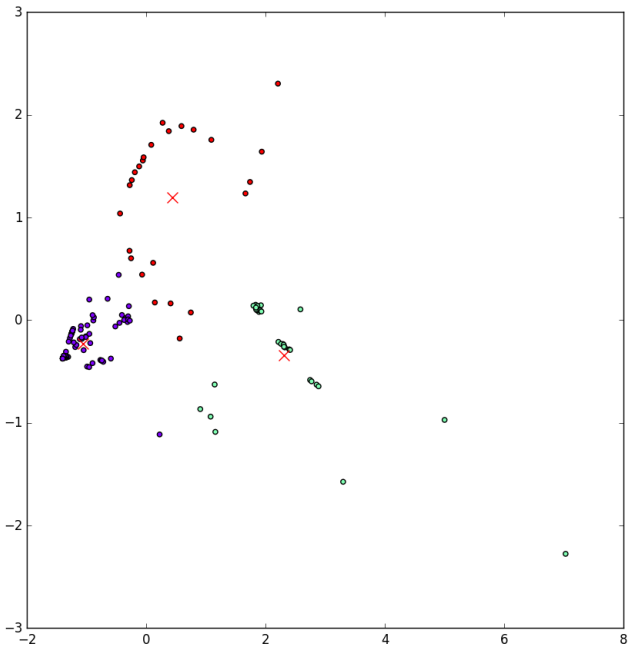


(a) k-means clusters

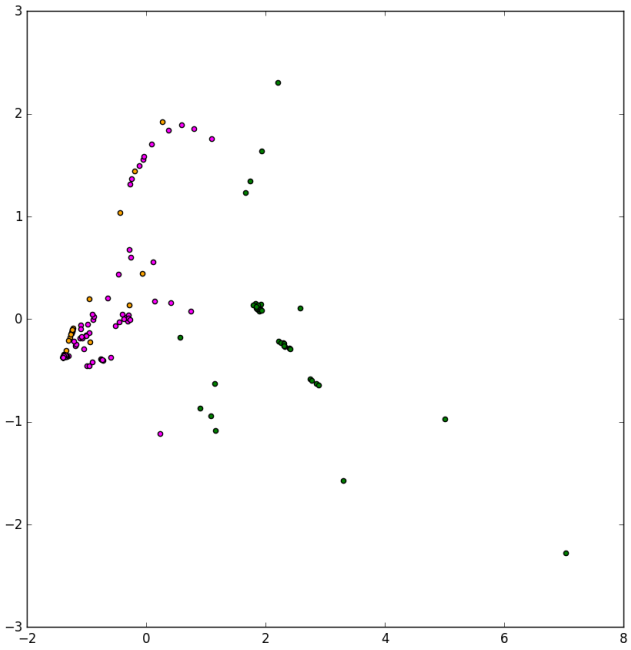


(b) actual clusters

Figure 4.12: Noise: k-means clusters VS actual clusters
Features: LLC acpki, LLC mpki, DRAM bandwidth, LLC miss rate

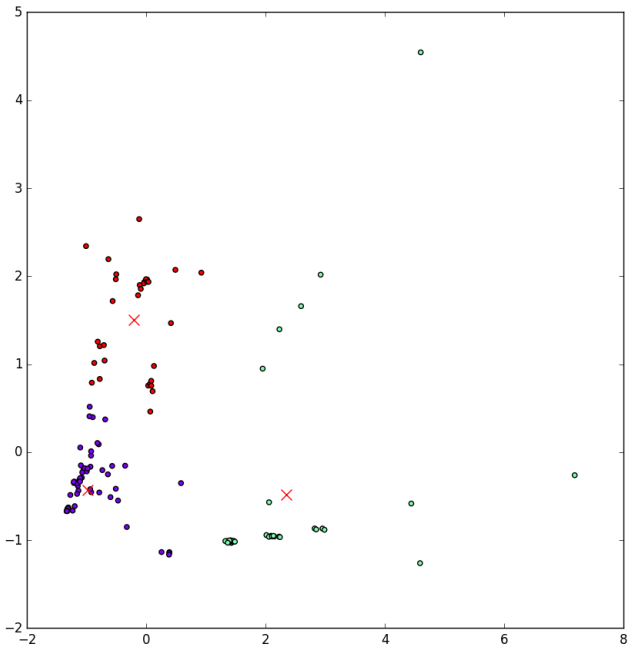


(a) k-means clusters

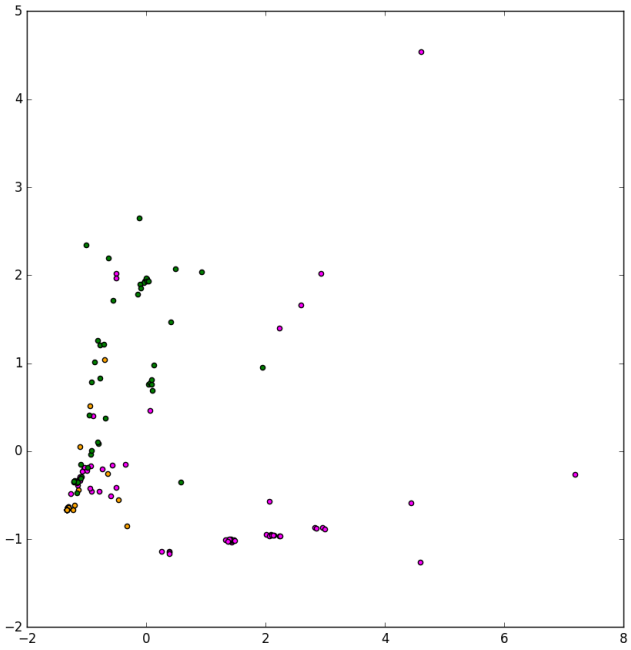


(b) actual clusters

Figure 4.13: Noise: k-means clusters VS actual clusters
Features: LLC mpki, DRAM bandwidth, LLC miss rate



(a) k-means clusters



(b) actual clusters

Figure 4.14: Sensitivity: k-means clusters VS actual clusters
Features: LLC acpki, LLC mpki, DRAM bandwidth, L2_pending_stalls/tot_cycles

Event Name	Description
LONGEST_LAT_CACHE.MISS	Core-originated cacheable demand requests the missed in LLC
LONGEST_LAT_CACHE.REFERENCE	Core-originated cacheable demand requests that refer to the LLC
L2_RQSTS.MISS	All requests that miss in the L2 cache
L2_RQSTS.REFERENCES	All L2 requests
MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops
MEM_UOPS_RETIRED.ALL_STORES	Retired store uops that split across a cacheline boundary
BR_INST_RETIRED.ALL_BRANCHES	All (macro) branch instructions retired
CYCLE_ACTIVITY.STALLS_L2_PENDING	Execution stalls while L2 cache miss demand load is outstanding
CYCLE_ACTIVITY.STALLS_MEM_ANY	Execution stalls while memory subsystem has an outstanding load
CYCLE_ACTIVITY.STALLS_L1D_PENDING	Execution stalls while L1 cache miss demand load is outstanding
CYCLE_ACTIVITY.CYCLES_L2_PENDING	Cycles while L2 cache miss demand load is outstanding
CYCLE_ACTIVITY.CYCLES_MEM_ANY	Cycles while memory subsystem has an outstanding load
CYCLE_ACTIVITY.STALLS_TOTAL	Total execution stalls
CYCLE_ACTIVITY.CYCLES_L1D_PENDING	Cycles while L1 cache miss demand load is outstanding
RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining from sync)
CPU_CLK_UNHALTED.THREAD_P	Thread cycles when thread is not in halt state
DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Load misses in all DTLB levels that cause page walks
DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Store misses in all DTLB levels that cause page walks
L2_TRANS.DEMAND_DATA_RD	Demand Data Read requests that access L2 cache
L2_TRANS.RFO	RFO requests that access L2 cache
L2_TRANS.L1D_WB	L1D writebacks that access L2 cache
L2_TRANS.L2_WB	L2 writebacks that access L2 cache

Table 4.2: Performance Monitoring Events - Broadwell Architecture

Chapter 5

An Application Classifier using Machine Learning

In this chapter we describe our final approach to creating an application classifier to prevent interference consequences. Since our previous efforts, which centered around more simplistic designs and algorithms, proved to be incapable of detecting the underlying correlations between PMUs, we decided to adopt a more sophisticated approach and employ machine learning methods. As our problem is a classification one, our work was to create a representative training set, select and train a suitable for the problem classifier and manipulate its parameters to reach an optimal result. We continue on to present a short summary of necessary machine learning background, outline the challenges we faced when designing our classifier and evaluate our final implementation.

5.1 Machine Learning Background

Machine learning comprises of a large set of algorithms and tools employed to essentially program computers to learn from data. Though more challenging to understand those algorithms, they are particularly useful in problems too complex for a statistical or empirical analysis to yield a solution, for cases with many involved parameters and a large design space that requires a lot of hand-tuning and extensive lists of rules, or for problems that include large amounts of data or need to dynamically adapt in the presence of new datapoints.

In general, machine learning methods can be categorized according to a variety of criteria, such as:

- Whether they are trained with human supervision or not (supervised, unsupervised, semisupervised or reinforcement learning).
- Their ability to learn and adapt while operating (online or batch learning).

- Whether they try to make predictions by uncovering patterns in the training data or simply compare new datapoints with known ones (model-based or instance-based learning).

The pipeline of the machine learning process proceeds as follows (figure 5.1). The first and perhaps most important step is to acquire or create a representative dataset. It is usually said that ”models are only as good as their data”, since a machine learning algorithm creates a model based on the datapoints upon it is trained; the more characteristic of the the general population the training data, the more the possibilities that for the model to make accurate predictions. We will present several data-preprocessing steps in later sections. The dataset is then split into a training set and a test set. The training set will be the datapoints upon which the algorithm will be trained, whereas the test set is used to evaluate the model produced and tune its parameters. This process is usually iterative: the model can be re-trained and re-tested until it yields satisfactory results.

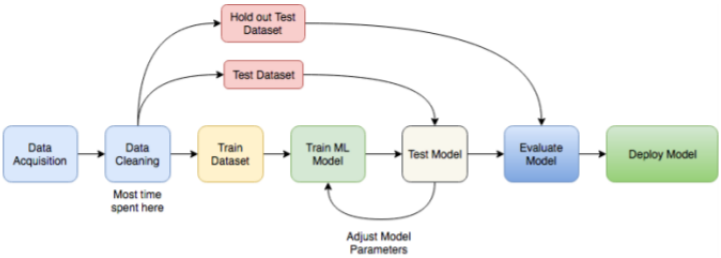


Figure 5.1: Machine learning process overview

The problems tackled by machine learning techniques can be split into to distinctive categories: classification (predicting classes) and regression (predicting values). Let x be the input to a model, and y the corresponding (predicted) output. If y is a discrete -categorical value (a discrete class label), we have a classification problem, whereas if it is a real number (e.g. an integer or a floating point value) we have a regression problem. Some machine learning algorithms, such as decision trees, can tackle both categories. Other ones can either be applied to only one category of problems, or need extensive and complex modification to be fitting for applying to both categories. Some popular classification algorithms include decision trees, support vector machines, logistic regression, naive Bayes, k-nearest neighbors etc.

5.1.1 Data Preparation

As we already mentioned, data is one of the most crucial parts in a machine learning model, and must fulfill the following criteria.

1. **Sufficient Quantity**. For machine learning algorithms to actually be able to learn, a large amount of datapoints are required. This is especially evident in [11], where

even fairly simple algorithms were able to perform almost the same as very complex ones when given enough data.

2. **Representative Instances**. Even if one has a vast amount of data available, those must be representative of the general population and the new cases the model might come to face. For example, it would be useless to train a regression model to predict the average temperature for a month using training data mainly from summers months. Note that it is not necessary for all classes or ranges of values to be equally represented: the dataset must contain datapoints in the same analogy as they are present in the general population.

3. **Good Quality**. A dataset ideally should not contain errors, outliers or irrelevant noise.

4. **Relevant Features**. The features by which datapoints are characterized also play a key role. More features might uncover more intricate patterns, but too many or irrelevant ones introduce a significant amount of noise to the system.

5.1.1.1 Feature Selection

Feature selection is the (most of the times) automatic selection of data attributes that are most relevant to a specific problem. It usually acts as a filter, removing features that might be redundant or introduce false patterns. Reducing the number of features a model needs to perform adequately is desirable, as it results in a faster model, whose underlying processes are less complex and more easily understood. Nevertheless, if not done carefully feature selection can introduce bias and lead to overfitting. Feature selection methods can be split into three categories:

1. **Filter Methods**: A statistical measure is used to assign a score to each feature. All features are then ranked by score and either selected or removed.

2. **Embedded Methods**: Those methods learn which features mostly contribute to the accuracy of a model while the model is being created.

3. **Wrapper Methods**: Different combinations of features are created, evaluated and compared to each other. The evaluation is done by a predictive model, which assigns each combination with a score based on model accuracy. A well known wrapper method is the Recursive Feature Elimination (RFE) algorithm. RFE iteratively fits a model using data with a specific set of features and removes the weakest one based on the model's ranking, until it reaches a specific number of features.

5.1.1.2 Feature Scaling

In the K-Means subsection we mentioned using *standardization*, a step that is necessary when different features have different units. Each feature vector is standardized to

have a mean of zero and a standard deviation of one, replacing each element x use a new x' such that

$$x' = \frac{x - \overline{x}}{\sigma}$$

where $\overline{x}$ and σ are the original vector's mean and standard deviation respectively. Other methods of scaling include *min-max normalization* (data is scaled to the range of $[0, 1]$ or $[-1, 1]$) and *scaling to unit length* (vector is scaled so that its complete length is 1).

Note that in machine learning models, the scaling transformation must be created upon only the training set and not the whole dataset.

5.1.2 Training and Test Set

Creating a training and a test set from your datapoints is a key step, as one must make sure that both are characteristic of the whole dataset so that the algorithm is accurately trained and evaluated. A testing set can be created by sampling the dataset in either a *random* or *stratified* way. If the dataset is large enough, then random sampling should be sufficient to adequately capture all trends. However, in very small or incomplete datasets, one may prefer opting for stratified sampling: the dataset is divided into homogeneous subgroups called strata, and the right number of instances is sampled from each stratum to guarantee that the test set is representative of the overall population. Selecting the criterion according which the strata are created is pivotal, as it must be the feature or combination of features that best represents the dataset's characteristics.

5.1.3 Support Vector Machines

A Support Vector Machine (SVM) is a quite versatile and powerful supervised machine learning model, that can perform both regression and classification (linear and non-linear), though it usually used for the latter. It is also particularly suitable for small and medium sized datasets.

When used for classification, the SVM algorithm plots data instances as points in a n-dimensional space (where n is the number of features), with the value of each feature being the value of a particular coordinate. Then, classification is performed by finding the maximum margin hyperplane that differentiates best the two classes. We call support vectors the co-ordinates of individual instances, and support vector machine the frontier (hyperplane) which best divides the two classes.

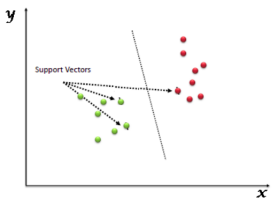


Figure 5.2: SVM classification example

5.1.3.1 Linear and Non-Linear SVM

When the two classes can be separated by a straight line we call them linearly separable. If we demand that all instances of a class are on the one side of the hyperplane, we are conducting *hard margin classification*. However, this only performs well if there are no outliers and the data is indeed linearly separable, as an outlier can either prevent a hard margin classifier from finding a hyperplane, or find a hyperplane that is not satisfactory.

To avoid such issues it is preferable to use a more flexible model, one that balances keeping the margin between the classes as large as possible and limiting margin violations. This is called *soft margin classification*, and can be controlled with the regularization parameter of the classifier, discussed later on.

Many times though a dataset cannot be separated by a straight line. In this case, a method called *kernel trick* is employed to try and create a hyperplane. A kernel is a mathematical function which transforms a low dimensional input space to a higher dimensional one, which essentially means converting the problem from non-separable to separable.

5.1.3.2 Tuning Parameters

The SVM algorithm has a number of parameters that can be used to customize the model. The first one is the **kernel** used to create the hyperplane, with the most common kernel types being linear, polynomial and gaussian. The kernel type determines the mathematical function used to compute the hyperplane, and is chosen based on the way classes are formed and how they are separated(linearly or not). Two other important parameters are regularization and gamma. **Regularization** (or C) indicates how much we want to avoid misclassifying each training example. For large values of C, a smaller-margin hyperplane will be chosen, if that hyperplane is more effective in classifying all the training points correctly. Conversely, a very small value of C will result to larger-margin separating hyperplane, even if that hyperplane misclassifies more points. **Gamma** defines how far the influence of a single training instance reaches. With low gamma, instances far away from plausible lines are considered in calculation for the hyperplane, whereas high gamma means the points close to plausible lines are considered in calculation.

5.1.4 Classification Problems

Classification problems, as we have already mentioned, are defined as problems where for an input x the output is a discrete value y, representing a class. Before continuing, we would like to present some of the different types of classification, the challenges one might face when trying to train classifier and the main metrics used for evaluating the later's efficiency.

5.1.4.1 Binary and Multiclass Classification

Binary classifiers can distinguish between only two classes, whereas *multiclass* (or multinomial) classifiers can distinguish between more than two classes. Some algorithms,

such as naive Bayes, can handle multiple classes directly, while others, like SVM, are strictly binary. However, binary classifiers can be manipulated into performing multiclass classification by either using a one-versus-all or a one-versus-one strategy. In *one-versus-all* (OVA), N binary classifiers are trained for N classes, with classifier N_i distinguishing between class i and all other classes merged together. For a datapoint to be classified, each classifier produces a decision score. The one with the higher score is selected, and the datapoint is place in the corresponding class. In *one-versus-one* (OVO) strategy, one classifier is trained for each pair of classes, resulting in $\frac{N*(N-1)}{2}$ classifiers. A new datapoint is placed in the class that is chosen by the majority of the classifiers.

5.1.4.2 Overfitting and Underfitting

The performance of a machine learning model is judged by its predictions' accuracy when faced with an unknown dataset, and we have already discussed the role of a high-quality training set in achieving good accuracy. However, a model with a great training set might underperform either because the model is too simple to accurately describe the population, or too complex to generalize well.

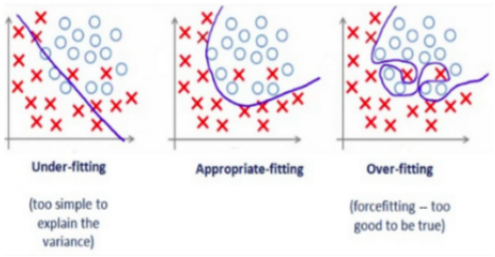


Figure 5.3: Three classifiers for the same data, showcasing under- and overfitting.

Overfitting refers to a model that has learned the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data are picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and prevent the model from generalizing effectively. Overfitting is more likely with nonparametric and nonlinear models that have more flexibility when learning a target function. Therefore, many of these models also include parameters or techniques to control how much detail the model learns.

Techniques such as cross validation, dataset enrichment, features removal, ensembling, etc. can be used to prevent a model from overfitting. In the case of SVM, the aforementioned parameters of C and gamma can be used to mitigate overfitting phenomena. In general, if an SVM classifier is overfitting, C and gamma should be decreased. In figure 5.4, we show SVM models trained with a gaussian kernel and different C and gamma values, and how this affects the resulting hyperplane.

Underfitting refers to a model that can neither fit the training data nor generalize to new data. It is often not discussed, as it is easily detected during training given a good

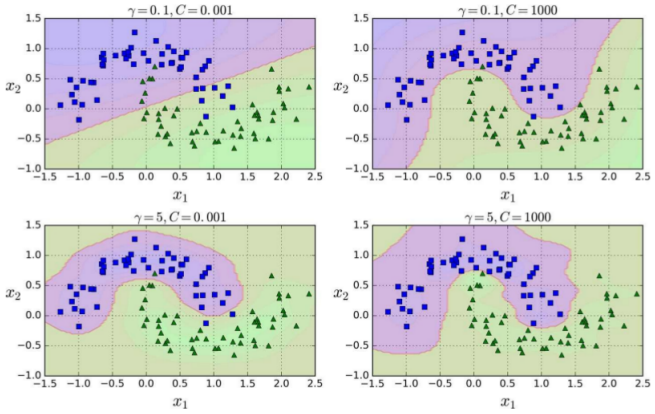


Figure 5.4: SVM classifiers trained with different C and gamma values

performance metric. When it is present, a more complex model is required.

5.1.4.3 Performance Metrics

To quantify how a classifier is performing, and detect phenomena of over- or underfitting, data scientists rely on evaluation metrics. The most important of them are:

- **Confusion matrix**: Although itself not a performance measure, the confusion matrix can provide very useful insights into the distribution of the test set into different classes, and is the foundation of almost all other metrics.

		Predicted class: *P*	Predicted class: *N*
Actual Class	*P*	True Positives (TP)	False Negatives (FN)
	N	False Positives (FP)	True Negatives (TN)

Figure 5.5: Confusion matrix of a binary classification problem

Suppose we have a binary classification problem with two classes, P and N, and a classifier being evaluated on the instances of a test set. According to each instance's actual and predicted class, it will be characterized as TP (true positive), FP (false positive), FN (false negative) and TN (true negative). Ideally, the number of false positives and false negatives would be zero. In a classification problem with N classes, the confusion matrix is a N*N array defined similarly to binary problems. Each row of the matrix represents the results of prediction for the corresponding class at that row, while each column represents the actual class. The diagonal cells show the number of correct classifications, while the off diagonal cells represent the misclassified predictions.

- **Accuracy**: Accuracy in classification problems is the number of correct predictions made by the model over all predictions made, also defined as:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

with respect to the confusion matrix. In general, accuracy is not the preferred performance metric when it comes to classifiers, as it is affected by skewed datasets (datasets where some classes are much more populated others), introducing bias towards the most popular class.

- **Precision**: Using the confusion matrix, precision is defined ass:

$$Precision = \frac{TP}{TP + FP}$$

Precision is a metric of the portion of positive predictions that were actually positive.

- **Recall** or **Sensitivity**: Typically used alongside precision, recall shows the proportion of actual positives that were identified correctly:

$$Recall = \frac{TP}{TP + FN}$$

In a classification task, a precision score of 1.0 for a class A informs us that every item labeled as belonging to A does indeed belong to it, but says nothing about the number of items from A that were misplaced into other classes. On the other hand, a recall of 1.0 means that every item that should have been labeled as A was indeed labeled as such, but says nothing about how many other items were incorrectly also labeled as A. Ideally, a classifier would have both high recall and precision. In reality, those two metrics have an inverse relationship, and increasing one comes at the cost of decreasing the other. Depending on the classification problem, one should choose which of the two better applies and should receive more attention.

- **F1 score**: A metric that combines precision and recall and is better suited when balance between the two is needed (and classes are unevenly populated), defined as:

$$F1\_score = 2 * \frac{Precision * Recall}{Precision + Recall} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

- **ROC curve and ROC-AUC score**: The ROC curve is a performance measurement used for a classification problem at various threshold settings. ROC-AUC score is a measure of separability, i.e. it displays the model's capability to distinguish between classes, and is equal to the area underneath the ROC curve. Given a binary

classification problem, the True Positive Rate, or the Recall that we have defined above is given by the type:

$$TPR = \frac{TP}{TP + FN}$$

while the false positive rate is defined as:

$$FPR = 1 - TPR = \frac{FN}{TP + FN}$$

The ROC curve is created by plotting TPR against FPR:

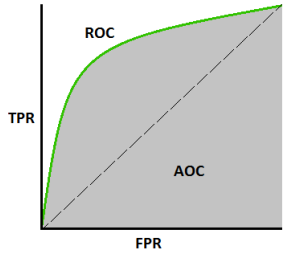


Figure 5.6: ROC curve

An ideal model has a ROC-AUC score near 1, meaning it can correctly discern classes.

For further information about machine learning models, the reader can refer to [9].

5.2 The Noise and Sensitivity Classifiers

We have already discussed the nature of our supervised classification problem, but we will mention it again for completeness. Our goal was to build two classifiers, one for noise and one for sensitivity, that receive as input a vector of application features in the form of PMU values and produce an output in the form of a discrete number, either ”0”, ”1” or ”2”, which represents the class the input has been placed to. Table 5.1 illustrates the correspondence between numbers and classes for each classifier.

Our dataset consisted of the 140 benchmarks described in the previous chapter, with the same noise and sensitivity labels that were then determined. Because our dataset was very small for machine learning algorithms, our design had to be extremely careful for our model to yield meaningful results. All the processes described below were performed twice, once for the noise classifier and once for the sensitivity classifier.

Our first step was to divide the dataset into train and test sets. Here, we experimented with both random and stratified sampling. The strata in the stratified sampling were created using the classes’ labels, so that our final sets would have the three classes in the

Output	Noise Class
0	Quiet
1	Potentially Noisy
2	Noisy

Output	Sensitivity Class
0	Insensitive
1	Potentially Sensitive
2	Sensitive

Table 5.1: Classes of the Noisy and Sensitivity Classifiers

same ratios as the original dataset. Since we had such limited data, random sampling was expected to create train and tests sets not representative of the population. Indeed, stratified sampling created train and test sets which contained datapoints of discrete classes in ratios that differed from those of the original dataset by 1.4% on average and 3% maximum. On the other hand, random sampling's sets differed by 8.3% on average and 14.5% maximum. Thus, we chose the stratified sampling method, and split our original dataset into 70% train set - 30% test set. Note that the train and test sets for the two classifiers were different.

Next, we scaled our train set using the sklearn.preprocessing.StandardScaler() method, which performs standardization. We then executed a feature selection algorithm to determine which features were the most useful. The features that were examined are outlined in Table 5.2. Our algorithm of choice was RFE with a linear regression model, and the top 6 features for each classifier are presented sorted in Table 5.3. At this point, we cross-examined RFEs results with our own past observations and insights. The features selected by the algorithm for the noise classifier agreed with our expectations, although memory bandwidth was only the fourth most important feature. This might be because although noisy applications exhibit very high memory bandwidths and are easily discernible, the bandwidth values of quiet and potentially noisy applications overlap, and thus bandwidth alone cannot tell them apart. When it came to sensitivity, the results were not as expected. The ratio of stalls due to shared resources to total cycles ranked first, as we anticipated, but LLC acpki ranked last, probably for the same reasons that memory bandwidth ranked only fourth in the noise classifier. However, because RFE takes into consideration the accuracy of classifiers, which as we have mentioned is not the best metric for multilabel classification, we decided to train our classifiers with more than one feature sets, and decide on the results. The final feature sets tested are presented in Table 5.2.

The algorithm we chose as the foundation of our model was SVM. SVM classifiers perform very well on small datasets, as they create the hyperplane taking into consideration only the points close to it (support vectors) and not all the datapoints. In addition, they are rather resistant to outliers. We employed the implementation denoted as SVC in python's scikit-learn framework, and used the one-versus-one multiclass classification strategy. To improve our models and mitigate overfitting, we experimented with different values for the C and gamma parameters using 10-fold cross validation with grid search, as well as with both linear and gaussian kernels. Also, when running the cross validation method we

total features examined	LLC acpki, LLC mpki, LLC miss rate, DRAM Bandwidth, total L2 pending miss stalls, total L2 pending miss stalls/total cycles, store buffer stalls
noise feature set	LLC acpki, LLC mpki, LLC miss rate
sensitivity feature set	LLC mpki, LLC acpki, DRAM bandwidth, total L2 pending miss stalls/total cycles

Table 5.2: Features used in final classifiers

noise	LLC mpki, LLC miss rate, LLC acpki, DRAM Bandwidth, total L2 pending miss stalls
sensitivity	total L2 pending miss stalls/total cycles, LLC mpki, DRAM Bandwidth, total L2 pending miss stalls, LLC miss rate

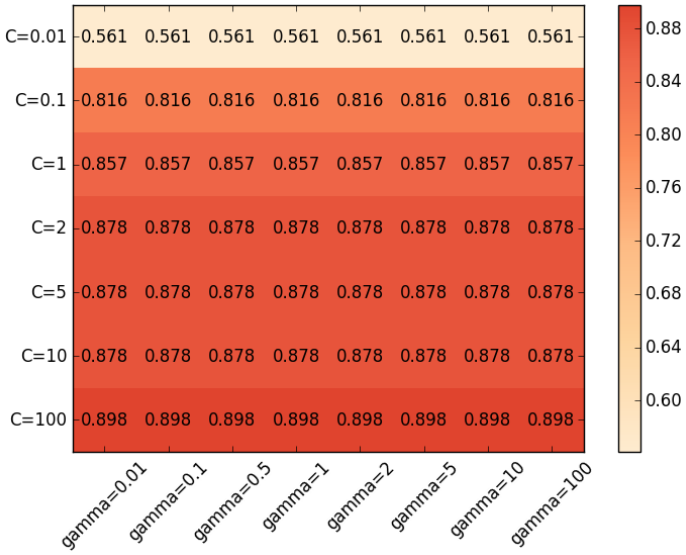
Table 5.3: Top 5 features ranked from most to least significant using RFE

evaluated each model with a number of different scoring functions: accuracy, precision, recall and f1 score (both micro and macro averages).

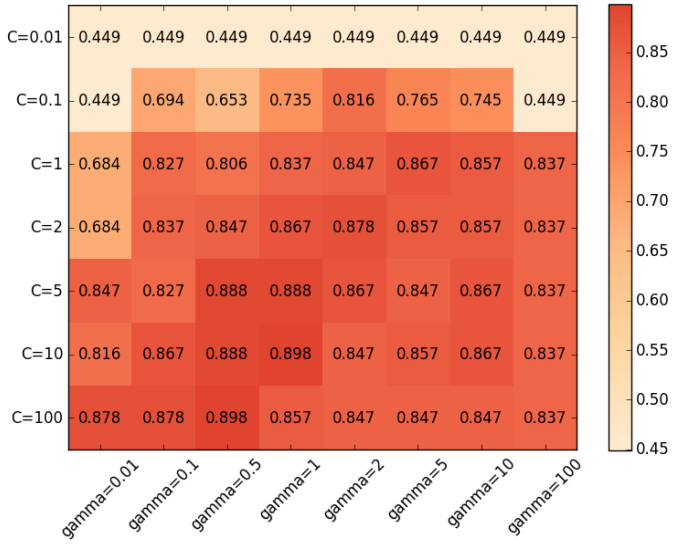
Because our train and test sets are small, the results of this analysis were treated with caution. There were models that performed very well, but high values for C and gamma, which suggests overfitting. In Figure 5.7 we see the macro averaged recall scores of different combinations between C and gamma, for two different noise classifiers, one with linear kernel and one with gaussian (using the same feature and train set). The gaussian kernel clearly outperforms the linear one, as different classes overlap in the feature space and a straight line cannot effectively separate them.

In many cases, higher values of C and gamma got better scores during the grid search, as they led to overfitting. Similarly, very low values of C and gamma did not create a sophisticated enough hyperplane to capture the classes (Figure 5.8). It is worth noting though that there were specific combinations of C-gamma values that outperformed the respective maximum values pair. Nevertheless, the scores were not drastically different, which is expected when taking into consideration the small size of our dataset.

To confirm our intuition that LLC acpki is a valuable feature in sensitivity classification, although RFE pointed otherwise, we compared the scores of a sensitivity classifier with feature set [LLC mpki, LLC acpki, memory bandwidth, total L2 pending miss stalls/total cycles] with one with a feature set containing the top 4 features from RFE, namely [LLC mpki, memory bandwidth, total L2 pending miss stalls, total L2 pending miss stalls/total cycles]. It is evident in Figure 5.9 that our intuition is correct, with the first classifier performing notably better.



(a) linear kernel



(b) gaussian kernel

Figure 5.7: Parameter search for two classifiers, one with linear kernel and one with gaussian Scores in cross validation

We chose the 5 best noise and 5 best sensitivity classifiers, which were then evaluated upon their respective test set (noise or sensitivity). In order to further validate the models that appeared to be optimal though cross validation, we hand-tuned models with the respective parameters and examined their scores under several scoring functions. The two final classifiers are outlined in Table 5.4, and their scores are presented in Table 5.5.

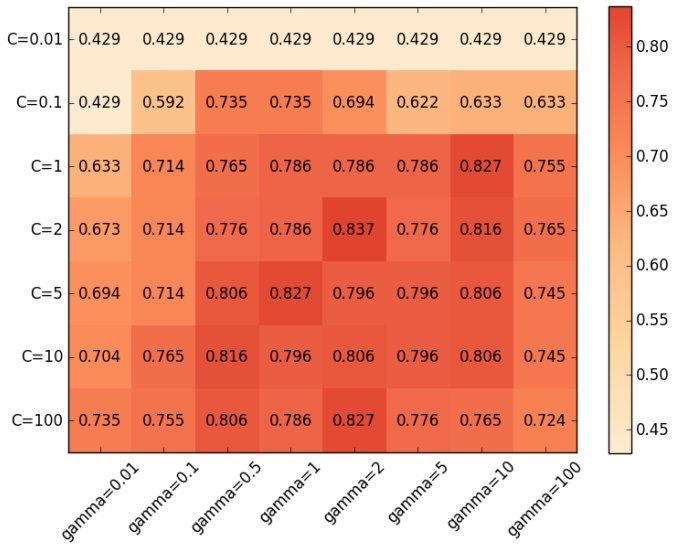


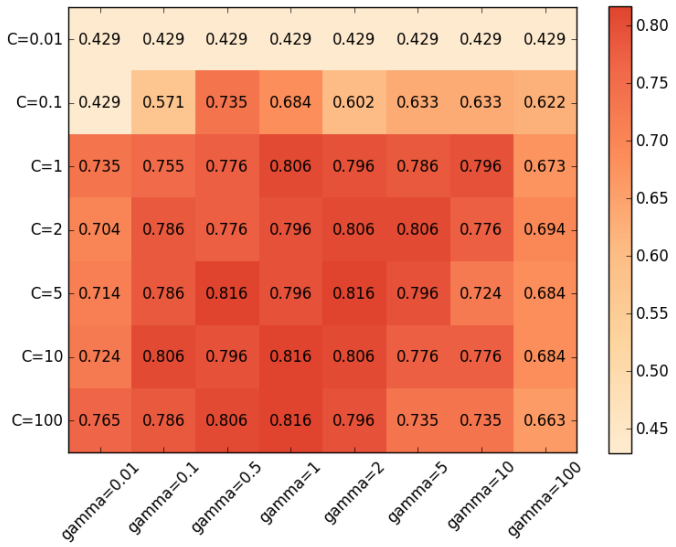
Figure 5.8: Parameter search for a sensitivity classifier. Very low C values lead to underfitting.

Classifier Type	Feature Set	Kernel	C	Gamma
noise	LLC mpki, LLC acpki, LLC miss rate	gaussian	10	1
sensitivity	LLC mpki, LLC acpki, DRAM bandwidth, total L2 pending miss stalls/total cycles	gaussian	2	1

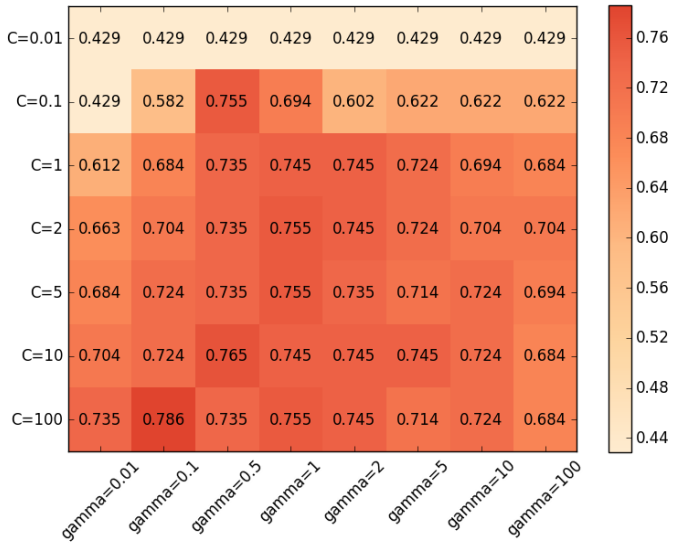
Table 5.4: Final noise and sensitivity classifiers' specifications

Classifier Type	noise	sensitivity
Accuracy	0.8333	0.8095
Recall (macro)	0.8333	0.8095
Recall (micro)	0.8005	0.7787
F1 score (macro)	0.8322	0.8095
F1 score (micro)	0.8271	0.7902

Table 5.5: Final noise and sensitivity classifiers' scores on test set



(a) feature set = [LLC mpki, LLC acpki, memory bandwidth, total L2 pending miss stalls/total cycles]



(b) feature set = [LLC mpki, memory bandwidth, total L2 pending miss stalls, total L2 pending miss stalls/total cycles]

Figure 5.9: Parameter search for two sensitivity classifiers with different feature sets

At this point, it is interesting to present the two classifiers' confusion matrices (noise and sensitivity respectively):

$$\begin{bmatrix} 7 & 5 & 0 \\ 0 & 19 & 0 \\ 0 & 2 & 9 \end{bmatrix} \quad \begin{bmatrix} 6 & 3 & 2 \\ 0 & 17 & 1 \\ 0 & 2 & 11 \end{bmatrix}$$

As we can see, in both cases the majority of mispredictions involves the potentially noisy or potentially sensitive class. This is due to the nature of our dataset: both those classes have around 60-70 datapoints, with each other class having around 40 datapoints. In such a small dataset, this difference in sizes was enough to create a small bias in favor of the potentially-x classes, as the classifier "saw" more of their instances during training. We tried to eliminate this phenomenon using a data preparation technique called *over-sampling*. With over-sampling, artificial datapoints are created and used to pad classes that have small populations. We used the Synthetic Minority Over-sampling TEchnique (SMOTE), which places new instances on the lines connecting already existing (in the original dataset) instances of each minority class. We then trained two new classifiers, using train sets originating from the over-sampled dataset. Their confusion matrices are presented below (noise and sensitivity respectively):

$$\begin{bmatrix} 14 & 1 & 2 \\ 3 & 17 & 1 \\ 1 & 0 & 14 \end{bmatrix} \quad \begin{bmatrix} 16 & 0 & 1 \\ 4 & 13 & 0 \\ 6 & 2 & 10 \end{bmatrix}$$

The number of mispredictions did not change drastically, but their nature did; we now see for example sensitive applications being mistaken for insensitive. Although these classifiers might be considered more fair, we chose to not use them for two main reasons. First, the fact that the potentially-x classes have more instances is a characteristic of the general application population. In reality, there aren't many applications that are always noise or always insensitive, and most applications fall in the space in between. Second, we prefer our classification to be conservative when labeling an application as insensitive or quiet. Take the case of an application A that the classifier cannot decide whether to label it as quiet or potentially noisy. If A is quiet, but gets labeled as potentially noisy, no future co-runner will suffer. We might not get the maximum performance gain compared to a random scheduling as we will choose to schedule A with caution, thinking it might affect its co-runner, but we will certainly not lose. If on the other hand A is potentially noisy, but gets labeled as quiet, we might schedule it with a sensitive co-runner, thinking A will not affect its performance, when in reality it will. In the case we decided to keep the classifiers which were trained with the oversampled dataset, we could introduce our "preference" of not easily classifying an application as quiet or sensitive by changing the relative weights between the classes. SVC's implementation gives the designer the ability to control this level of "preference", and train classifiers that act accordingly.

5.3 Interference Aware Scheduling using Noise and Sensitivity Classifiers

Finally, we would like to illustrate how our classifiers can be utilized in a cloud environment to facilitate interference aware scheduling. We examined the following scenario: we had a server containing 10-core multiprocessors, and two application pools, one with high-priority (HP) applications and one with low-priority (LP) ones. Each multiprocessor could host one copy of a HP application and nine copies of a LP application. Our goal was to create HP-LP pairs such so that average performance degradation of HP applications is minimal. This resembles a commercial cloud where applications with strict QoS goals would be of higher priority, while best effort, batch applications would be used to fill any remaining cores.

Our scheduling algorithm was based on the following set of rules:

1. Select all sensitive HP applications and all quiet LP applications, and pair as many as possible.
2. Select all insensitive HP applications and all noisy LP applications, and pair as many as possible.
3. If there are remaining quiet LP applications, pair them with potentially sensitive HP applications.
4. If there are remaining insensitive HP applications, pair them with potentially noisy LP applications.
5. When faced with a mix of sensitive and potentially sensitive HP applications, and another one of noisy and potentially noisy BP applications, pair the potentially sensitive HPs with the noisy BPs and the sensitive HPs with the potentially noisy HPs.

We proceed to present some representative mixes of applications, and how our scheduler faced them.

5.3.1 Scenario 1: 1 noisy LP + 1 potentially noisy LP + 1 sensitive HP + 1 insensitive HP

The first example showcases exactly the benefits of a-priori knowledge of an application's behavior. The noisy LP (lbm_r) is co-scheduled with the insensitive HP (exchange2_)r, and the potentially noisy LP (omnetpp_r_star) with the sensitive HP (omnetpp_r_rand27). The Deg for the two HPs is 6% and 33% (19.5% on average), and their ipcs are plotted in Figure 5.10 and Figure 5.11

If the applications were paired vice-versa, the Deg for the HPs would have been 2% and 64%, significantly worse on average (33%).

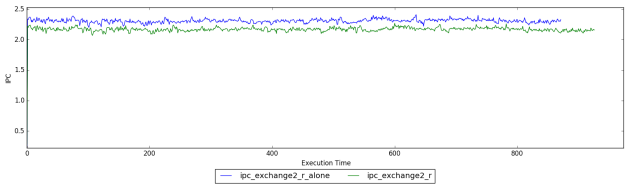


Figure 5.10: IPC, selected co-location: 1 exhange2_r with 9 lbm_r

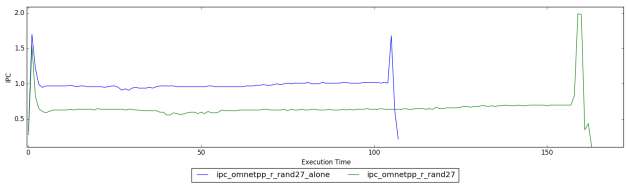


Figure 5.11: IPC, selected co-location: 1 omnetpp_r_rand27 with 9 omnetpp_r_star

5.3.2 Scenario 2: 1 noisy LP + 1 potentially noisy LP + 1 sensitive HP + 1 potentially sensitive HP

In this case, the scheduler chooses to place 1 noisy BP (lbm_) with 1 potentially sensitive HP (blender_r), and 1 potentially noisy BP (omnetpp_r_star) with 1 sensitive HP (omnetpp_r_rand27). Since there is no quiet LP, it tries to place the sensitive HP with the next "less noisy" LP. The HPs show a Deg of 29% and 30% (29.5%), and their IPCs are shown in Figure 5.12 and Figure 5.13. In the alternative scenario, the HPs' Deg would be 11% and 64% (37.5%). Here, one of the previous scenario's "harmful" collocation, that of (omnetpp_r_star) with (omnetpp_r_rand27), is in this scenario a necessary choice, since the alternative creates much performance degradation in total for the HPs.

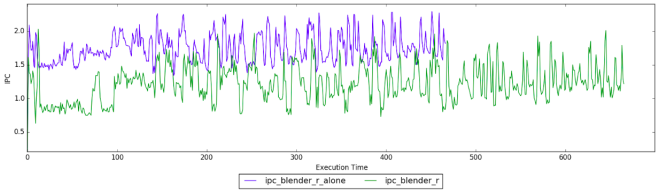


Figure 5.12: IPC, selected co-location: 1 blender_r with 9 lbm_r

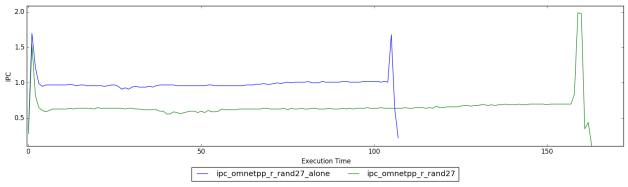


Figure 5.13: IPC, selected co-location: 1 omnetpp_r_rand27 with 9 omnetpp_r_star

5.3.3 Scenario 3:1 noisy LP + 1 quiet LP + 2 potentially sensitive HPs

Unfortunately, in this application mix our scheduler cannot do much, since both HPs are labeled as potentially sensitive, so it makes the pairs at random. A potential extension to tackle such cases would be for the scheduler to decide based on the values of individual PMUs, e.g. try to co-locate with the noisy LP the potentially sensitive BP with th lower LLC mpki, but this is beyond the scopes of this thesis. We present the two possible co-locations (Figure 5.14, Figure 5.15, Figure 5.16, Figure 5.17), and the Deg of each HP.

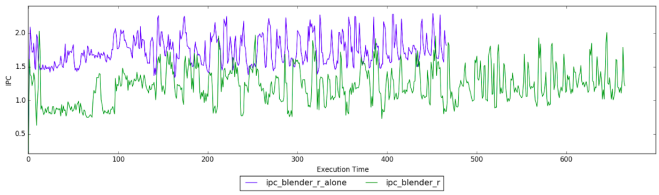


Figure 5.14: IPC, selected co-location: 1 blender_r with 9 lbm_r
Deg of HP = 29%

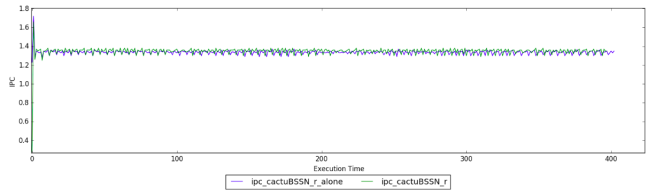


Figure 5.15: IPC, selected co-location: 1 cactuBSSN_r with 9 exchange2_r
Deg of HP = 0%

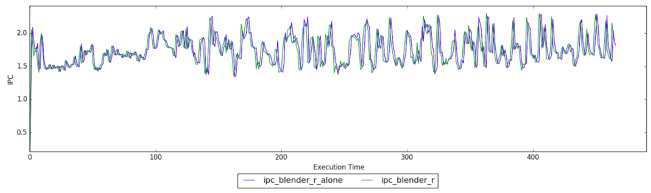


Figure 5.16: IPC, alternative co-location: 1 blender_r with 9 exchange2_r
Deg of HP = 0%

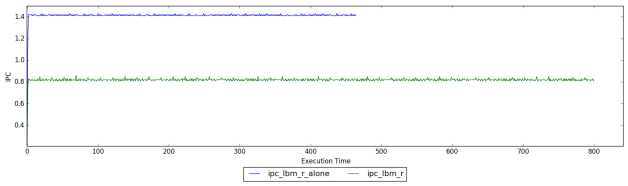


Figure 5.17: IPC, alternative co-location: 1 cactuBSSN_r with 9 lbm_r
Deg of HP = 22%

An important observation we can make by looking at co-executions like the one above is the effect of multiple threads. In our initial analysis, we discussed scenarios that consisted of only one thread per application. Here, we see how certain characteristics are augmented when multiple copies of an application are present. The fact that potentially noisy applications, when in many copies, create significant contention is not unexpected; this is why we labeled them as *potentially* noisy in the first place. Noisy applications in many copies cause so much contention that even insensitive applications might exhibit a small performance degradation, as shown in Figure 5.10.

Chapter 6

Conclusion and Future Work

Interference due to application behavior in multicore systems has proven to be the main bottleneck for resource utilization and efficient execution. Multiprocessors contain a number of cores, each one having a private Level 1 and Level 2 cache, and all of them sharing the rest of the chip's resources. In environments with a very large number of applications, such as datacenters or HPC clusters, the amount of stress placed on the Last Level Cache and DRAM Bandwidth leads to significant performance degradation. Especially in the case of commercial clouds, where some applications require strict performance guarantees, interference prevention and mitigation is of utter importance. In the beginning of these thesis, we examined several prior approaches that aimed to tackle different aspects of the problem: detect application interference and differentiate it from workload fluctuations and normal application phase changes, or specify the resource that is suffering from contention and the application causing it, predict performance degradation of different co-execution scenarios. After carefully analyzing those proposals, we identified their individual trade-offs and evaluated which mechanisms provide enough benefits to cancel their overheads. We concluded that there is no mechanism proposed that can accurately predict the behavior of an application in a co-execution scenario in regards of interference that is based only on metrics gathered while an application is running in an isolated environment.

We then executed a representative set of co-execution scenarios to validate that interference is indeed detrimental to performance. Our results indicated that contention in some cases can be so high that the ability to predict it before it happens, rather than trying to detect it while it is happening, can be of great significance. In addition, several applications showcased constant behavior in all co-execution scenarios, leading us to believe that the contention an application creates or the impact contention has on its performance are characteristics inherent to the application, and can perhaps be derived by examining other aspects of its behavior. Our goal was to design a mechanism that could deduce how noisy (capable of creating contention) or sensitive (prone to suffering from contention) an application is based on a set of low-level performance counters (PMUs) gathered during isolated execution. This mechanism had to abide by the following constrains: bare minimum profiling overheads, rely solely on isolated performance and not on experimental

co-execution scenarios, and be able to predict cases of interference prior to scheduling and not detect them after scheduling the application in a production environment.

The first step was defining the two characteristics we wanted to predict: noise and sensitivity. We determined the level of sensitivity one application has by observing its performance when it was allocated different amounts of LLC capacity, and categorized applications into *insensitive* when LLC capacity didn't affect performance, *sensitive* when performance constantly improved as LLC capacity increased and *potentially sensitive* when performance improved until a certain capacity threshold, after which remained the same. To determine the level of contention caused by an application we utilized a *reference benchmark*. According to the level of performance degradation an application caused to our reference benchmark we labeled it as *noisy* (high degradation, *quiet* (no degradation) or *potentially noisy* (moderate degradation). The labels each application received where additionally confirmed by examining its behavior in our executed scenarios.

We then composed extensive profiles for each application by collecting a large amount of low-level performance counters during execution. The counters we focused on where those related with the LLC, the memory bandwidth, other memory-related components (such as TLBs) and their interactions with private caches and DRAM. From those profiles, we tried to detect trends in the values of PMUs that could be correlated with the labels of the applications. Although some metrics like LLC acpki and memory bandwidth did exhibit general patterns for some of our categories (e.g. noisy applications having high memory bandwidth values) we couldn't outline a specific set of rules to fully characterize the application distribution into our categories. However, the exposure of even some general trends motivated us to examine more complex mechanisms, such as clustering algorithms. We experimented with k-means clustering, a well-established algorithm that divides instances into classes according to a set of features. Although we evaluated a variety of feature sets, the PMU patterns describing behavior and the boarders between classes proved to be too complex for k-means to uncover them.

Our final approach was to utilize machine learning techniques, which have proven to be particularly effective in recognizing intricate relationships between data features. More specifically, we aimed to tackle our supervised learning problem with two multi-class classifiers, one for noise and one for sensitivity. The main challenge in our case was the significantly small size of our dataset, which required scrupulous manipulations to avoid misleading results. To overcome that hurdle, after carefully selecting representative train and test sets as well as a list of data features for each classifier, we trained a large collection of SVM classifiers with various feature sets and kernels. We also tried to mitigate overfitting phenomena by experimenting with the parameters (C, gamma) of the classifier. We evaluated all models using a list of scoring functions (accuracy, recall, precision, f1 score). Our two final classifiers had (recall) scores of 0.833 and 0.8095 for noise and sensitivity classification respectively. We then showcased how our classifiers can be utilized in a cloud environment with high- and low-priority applications to make optimal scheduling decisions and avoid or minimize interference effects on performance.

Our work can be extended towards various directions. One intriguing approach would be to further break down each category, especially the potentially-x ones, to make even

more accurate predictions about their behaviors. Another idea is to explore if it possible to assign each application with a specific score that can be directly utilized to predict the amount of contention it might create/suffer. Our categories were intentionally generic due to the lack of a large dataset, but obtaining profiles from more applications might reveal new pattern that can be exploited.

Interesting work can also be done on how behaviors scale on scenarios with more than two threads. We briefly discussed in our last chapter how certain characteristics are augmented when there are more than one co-executors present, and further investigation is needed to determine how classification can be utilized in more complex co-execution scenarios. Machine learning techniques are a promising approach to the matter, as they have only recently started being employed by system engineers and computer architects, and there is plenty of room to experiment with their abilities.

Bibliography

[1] José Nelson Amaral, Edson Borin, Dylan R Ashley, Caian Benedicto, Elliot Colp, Joao Henrique Stange Hoffmam, Marcus Karpoff, Erick Ochoa, Morgan Redshaw, and Raphael Ernani Rodrigues. The alberta workloads for the spec cpu 2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 159–168. IEEE, 2018.

[2] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[3] Arnamoy Bhattacharyya, Stelios Sotiriadis, and Cristiana Amza. Online phase detection and characterization of cloud applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 98–105. IEEE, 2017.

[4] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, pages 77–88. ACM, 2013.

[5] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 127–144. ACM, 2014.

[6] Ashutosh S Dhodapkar and James E Smith. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 217. IEEE Computer Society, 2003.

[7] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 83. IEEE Computer Society Press, 2012.

[8] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117. IEEE, 2018.

[9] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2017.

[10] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 22. ACM, 2011.

[11] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. 2009.

[12] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[13] Ram Srivatsa Kannan, Animesh Jain, Michael A Laurenzano, Lingjia Tang, and Jason Mars. Proctor: Detecting and investigating interference in shared datacenters. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–86. IEEE, 2018.

[14] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using os observations to improve performance in multicore systems. *IEEE micro*, 28(3):54–66, 2008.

[15] Gabriella Laatikainen, Arto Ojala, and Oleksiy Mazhelis. Cloud services pricing models. In *International Conference of Software Business*, pages 117–129. Springer, 2013.

[16] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378. IEEE, 2008.

[17] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[18] Jason Mars and Mary Lou Soffa. Synthesizing contention. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 17–25. ACM, 2009.

[19] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang E Nagel. Detecting memory-boundedness with hardware performance counters. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 27–38. ACM, 2017.

[20] Priya Nagpurkar, P Hind, Chandra Krintz, Peter F Sweeney, and VT Rajan. Online phase detection algorithms. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 13–pp. IEEE, 2006.

[21] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 219–230, 2013.

[22] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE, 2006.

[23] A Rajaraman and J Ullman. Textbook on mining of massive datasets. 2011.

[24] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 57–68. ACM, 2011.

[25] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 62–75. ACM, 2015.

[26] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 12–21. ACM, 2011.

[27] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *ACM SIGARCH computer architecture news*, volume 40, pages 423–436. ACM, 2012.

[28] Yuejian Xie and Gabriel Loh. Dynamic classification of program memory behaviors in cmps. In *the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[29] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi 2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391. ACM, 2013.

[30] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *ACM SIGARCH Computer Architecture News*, 44(2):33–47, 2016.

[31] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM Sigplan Notices*, volume 45, pages 129–142. ACM, 2010.