Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Χαρακτηρισμός των μετροπρογραμμάτων Koala

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΚΑΟΥΚΗΣ

Επιβλέπων : Γεώργιος Γκούμας
Καθηγητής, Ε.Μ.Π.

Συν-Επιβλέπων : Νίκος Βασιλάκης
Επικ. Καθηγητής, Πανεπιστήμιο Μπράουν

Αθήνα, Φεβρουάριος 2026

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Χαρακτηρισμός των μετροπρογραμμάτων Koala

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΚΑΟΥΚΗΣ

Επιβλέπων : Γεώργιος Γκούμας
Καθηγητής, Ε.Μ.Π.

Συν-Επιβλέπων : Νίκος Βασιλάκης
Επικ. Καθηγητής, Πανεπιστήμιο Μπράουν

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Φεβρουαρίου 2026.

.....	.....	.....
Γεώργιος Γκούμας Καθηγητής, Ε.Μ.Π.	Νεκτάριος Κοζύρης Καθηγητής, Ε.Μ.Π.	Νίκος Βασιλάκης Επικ. Καθηγητής, Πανεπιστήμιο Μπράουν

Αθήνα, Φεβρουάριος 2026

Γεώργιος Καούκης

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

Περίληψη

Το Koala αποτελεί μια σουίτα μετροπρογραμμάτων επικεντρωμένη στην έρευνα αξιολόγησης επιδόσεων για το κέλυφος Unix και Linux. Συνδυάζει μια συστηματική συλλογή από ποικιλόμορφα προγράμματα κελύφους προερχόμενα από πραγματικά σενάρια χρήσης, ρεαλιστικά σύνολα εισόδων διαφόρων ειδών που υποστηρίζουν μικρής και μεγάλης κλίμακας εκτελέσεις για τα προγράμματα αυτά, εκτενή ανάλυση και χαρακτηρισμό που βοηθούν στην κατανόησή τους, καθώς και επιπλέον υποδομή και εργαλεία που στοχεύουν στη χρηστικότητα και την αναπαραγωγιμότητα στην έρευνα συστημάτων. Τα μετροπρογράμματα Koala εκτελούν ένα ευρύ φάσμα κοινών εργασιών κελύφους· αξιοποιούν όλα τα κύρια γλωσσικά χαρακτηριστικά του κελύφους POSIX· χρησιμοποιούν μια ποικιλία από εργαλεία POSIX, GNU Coreutils και λογισμικό τρίτων· λειτουργούν σε εισόδους μεταβαλλόμενου μεγέθους και σύνθεσης· και παρουσιάζουν ένα ευρύ φάσμα προφίλ εκτέλεσης, καθιστώντας τα ιδιαίτερα ετερογενή και κατάλληλα για την αξιολόγηση ποικίλων στρατηγικών βελτιστοποίησης. Η εφαρμογή του Koala σε πέντε συστήματα που επιδρούν στην εκτέλεση προγραμμάτων κελύφους παρέχει μια πιο ολοκληρωμένη εικόνα των μεταξύ τους συμβιβασμών, γενικεύει τα κύρια ευρήματά τους και συμβάλλει σε μια βαθύτερη κατανόηση των συστημάτων αυτών.

Λέξεις κλειδιά

Κέλυφος, Unix, Linux, Μετροπρογράμματα, POSIX, Αξιολόγηση Επιδόσεων

Ευχαριστίες

Θα ήθελα να εκφράσω την ειλικρινή μου ευγνωμοσύνη σε όλους όσοι με στήριξαν κατά την ολοκλήρωση αυτής της διπλωματικής εργασίας και συνέβαλαν στο να γίνουν τα χρόνια μου ως φοιτητής τόσο ουσιαστικά και δημιουργικά.

Πρωτίστως, θέλω να ευχαριστήσω τον επιβλέποντά μου, Καθ. Γεώργιο Γκούμα, για την εμπιστοσύνη του, την καθοδήγησή του και για την ευκαιρία που μου έδωσε να πραγματοποιήσω τη διπλωματική μου εργασία στο Εργαστήριο Υπολογιστικών Συστημάτων. Η στήριξή του έχει επηρεάσει σημαντικά την ακαδημαϊκή μου εξέλιξη, καθώς τα μαθήματά του ήταν καθοριστικά για την αναζωπύρωση του ενθουσιασμού μου για τον τομέα κατά το τρίτο έτος των σπουδών μου.

Είμαι επίσης βαθύτατα ευγνώμων στον Καθ. Νίκο Βασιλάκη για την ευκαιρία που μου έδωσε να πραγματοποιήσω αυτή τη διπλωματική εργασία σε συνεργασία με την ομάδα Atlas του Πανεπιστημίου Brown. Η συνεχής ανατροφοδότηση και η εύστοχη καθοδήγησή του υπήρξαν καθοριστικές για την ολοκλήρωση αυτής της εργασίας και διαμόρφωσαν τα πρώτα στάδια της ερευνητικής μου πορείας. Ιδιαίτερες ευχαριστίες οφείλω στον Ευάγγελο Λάμπρου, για τη διαρκή υποστήριξή του, τις γόνιμες συζητήσεις και την πρακτική του βοήθεια, καθώς και στον Δρ. Lukas Lazarek, του οποίου οι συνεισφορές και τα εύστοχα σχόλια διαμόρφωσαν σημαντικά την κατεύθυνση και τη σαφήνεια αυτού του έργου.

Καθώς η παρούσα διπλωματική εργασία εκπονήθηκε στο πλαίσιο μιας ευρύτερης ερευνητικής προσπάθειας, θα ήθελα επίσης να ευχαριστήσω τον Ethan Williams για τη σημαντική και εκτενή συνεισφορά του, καθώς και τους υπόλοιπους συν-συγγραφείς του άρθρου Koala—Zhuoxuan Zhang, Καθ. Michael Greenberg και Καθ. Κωνσταντίνο Καλλά—για τη συνεργασία τους καθ' όλη τη διάρκεια του ευρύτερου έργου.

Επιπλέον, ευχαριστώ θερμά τους συναδέλφους μου από την ερευνητική μονάδα «Αρχιμήδης», την Ελένη, τον Γιάννη και τη Δανάη, για την υπομονή και τη στήριξή τους κατά τη διάρκεια της απαιτητικής διαδικασίας συγγραφής και ολοκλήρωσης της παρούσας διπλωματικής.

Θα ήθελα επίσης να εκφράσω τις ευχαριστίες μου στον Παναγιώτη, τον «ξάδελφό» μου και συνάδελφο απόφοιτο της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, ο οποίος με βοήθησε σταθερά να διατηρώ την προσήλωσή μου και με στήριξε καθ' όλη τη διάρκεια των σπουδών μας στο ΕΜΠ, στον Φίλιππο, που πέρα από πολλές εξόδους και διακοπές μοιραστήκαμε και τις τελευταίες εξεταστικές, καθώς και στους συναδέλφους «compañeros», Αναστασία, Γιώργο, Νικόλα, Αθηνά και Ναταλία που ήταν δίπλα μου σε διαλέξεις, εργασίες και εξετάσεις. Ευχαριστώ ακόμα τον Γιάννη, τον Νικόλα, τον Θέμη, τη Ναταλία, τον Ανδρόνικο, τη Λυδία, καθώς και όλους τους υπόλοιπους φίλους, συγγενείς και συνεργάτες, που—ο καθένας με τον δικό του τρόπο—φρόντισαν ώστε τα φοιτητικά μου χρόνια να είναι γεμάτα με τόσες χαρούμενες αναμνήσεις.

Πάνω απ' όλα, θα ήθελα να ευχαριστήσω τους γονείς μου, Ζαφείρη και Μαρία, των οποίων η ατελείωτη υπομονή, η αδιάκοπη στήριξη και η ακλόνητη πίστη σε εμένα αποτέλεσαν το θεμέλιο όλων των προσπαθειών μου.

Γεώργιος Καούκης,

Αθήνα, 19η Φεβρουαρίου 2026

Περιεχόμενα

Κατάλογος πινάκων

Κατάλογος σχημάτων

Κατάλογος κωδίκων

Κεφάλαιο 1

Εισαγωγή

Το κέλυφος Unix είναι δημοφιλές, ευέλικτο και ισχυρό, με ποικιλία εφαρμογών που εκτείνονται από την επεξεργασία δεδομένων και τη διαχείριση συστημάτων έως τη βιοπληροφορική και τη συνεχή ολοκλήρωση και ανάπτυξη. Παράλληλα, το κέλυφος έχει συγκεντρώσει σημαντικό ερευνητικό ενδιαφέρον, με πρόσφατη ακαδημαϊκή δραστηριότητα [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] να έχει οδηγήσει στην ανάπτυξη αρκετών βελτιστοποιητών που στοχεύουν στην επιτάχυνση προγραμμάτων κελύφους μέσω παράλληλων συστημάτων [3, 5, 7], κατανεμημένων συστημάτων [1, 11, 12], και άλλων μορφών κλιμάκωσης [4, 9, 13, 14].

Παρ' όλα αυτά, υπάρχει ένα σημαντικό πρόσκομμα για την αποτελεσματική διεξαγωγή έρευνας γύρω από το κέλυφος: δεν υπάρχει κάποια καθιερωμένη σουίτα μετροπρογραμμάτων για την αξιολόγηση των συστημάτων αυτών, με αποτέλεσμα να είναι δύσκολο να αξιολογηθούν νέα συστήματα με ρεαλιστικό και συγκρίσιμο τρόπο [15]. Έτσι, οι ερευνητές συχνά αναγκάζονται να δημιουργούν αυτοσχέδιες μικροδοκιμές αξιολόγησης (microbenchmarks), να συλλέγουν αποσπασματικά παραδείγματα από αποθετήρια ή να βασίζονται σε τυποποιημένες δοκιμές που εστιάζουν κυρίως στη συμπεριφορική ισοδυναμία και όχι στην απόδοση. Κατά συνέπεια, η έρευνα στο πεδίο στερείται βασικών πλεονεκτημάτων της συστηματικής συγκριτικής αξιολόγησης, όπως η επαναληψιμότητα, η αναπαραγωγιμότητα και η δίκαιη σύγκριση μεταξύ διαφορετικών συστημάτων.

Το Koala είναι μια σουίτα μετροπρογραμμάτων προσανατολισμένη στην έρευνα για την αξιολόγηση επιδόσεων [16], η οποία στοχεύει στο κέλυφος Unix και στο λειτουργικό σύστημα Linux, συνδυάζοντας συστηματική συλλογή προγραμμάτων, εκτενή χαρακτηρισμό και επαναχρησιμοποιήσιμη υποδομή. Για την επίτευξη αυτού του στόχου, το Koala παρέχει μια ποικίλη συλλογή από 126 πραγματικά προγράμματα κελύφους που εκτελούν εργασίες οι οποίες απαντώνται συχνά στην πράξη. Προσφέρει ρεαλιστικά δεδομένα εισόδου σε τρεις κλίμακες (ελάχιστη, μικρή και πλήρη), καθώς και αυτοματοποιημένα εργαλεία για την εγκατάσταση εξαρτήσεων, την εκτέλεση των μετροπρογραμμάτων, την επικύρωση των αποτελεσμάτων και την αναφορά της απόδοσης.

Η παρούσα διπλωματική εργασία αποτελεί υποσύνολο της ευρύτερης ερευνητικής προσπάθειας που παρουσιάστηκε στη δημοσίευση [16] στο USENIX ATC '25. Ο κύριος άξονας και η βασική συνεισφορά της παρούσας εργασίας επικεντρώνονται στην πρακτική εφαρμογή, τον μετασχηματισμό των σεναρίων και την εκτενή συγκριτική αξιολόγηση διαφορετικών συστημάτων. Μέσα από αυτή τη διαδικασία, αποδεικνύεται έμπρακτα η χρησιμότητα και η αναγκαιότητα της σουίτας Koala για την αξιολόγηση τόσο υφιστάμενων όσο και μελλοντικών συστημάτων βελτιστοποίησης. Συγκεκριμένα, η εργασία διερευνά τις πρακτικές προεκτάσεις της χρήσης του Koala αξιολογώντας πέντε διαφορετικές προσεγγίσεις και συστήματα επιτάχυνσης: τον εναλλακτικό διερμηνέα `zsh` [17], το Shark [15], το GNU `parallel` [18], το PaSh [5, 7] και το *hS* [14]. Για κάθε σύστημα αναλύονται εκτενώς ο μηχανισμός λειτουργίας του, η προσπάθεια υιοθέτησης και οι απαιτούμενοι μετασχηματισμοί κώδικα σε πραγματικά προγράμματα, ενώ παρουσιάζονται οι επιταχύνσεις που επιτυγχάνονται, καθώς και τα πλεονεκτήματα ή μειονεκτήματα κάθε προσέγγισης.

Η δομή της παρούσας διπλωματικής εργασίας οργανώνεται ως εξής: Αρχικά, παρουσιάζεται το απαραίτητο θεωρητικό υπόβαθρο, εστιάζοντας στις βασικές αφαιρέσεις, τη ροή δεδομένων και τη λειτουργία του κελύφους Unix (Κεφάλαιο 2). Στη συνέχεια, παρατίθεται η σχετική ερευνητική βιβλιογραφία, εξετάζοντας υφιστάμενα συστήματα βελτιστοποίησης, συλλογές προγραμμάτων κελύφους, τοπία αξιολόγησης και προηγούμενες προσπάθειες συγκριτικής μέτρησης (Κεφάλαιο 3). Ακολουθεί

η ανάλυση ενός αντιπροσωπευτικού κινητήριου παραδείγματος, το οποίο αναδεικνύει εμπράκτως τις προκλήσεις της επιτάχυνσης κελύφους και προετοιμάζει το έδαφος για την ανάγκη μιας ολοκληρωμένης υποδομής (Κεφάλαιο 4). Κατόπιν, περιγράφεται ο σχεδιασμός και η υλοποίηση της σουίτας Koala, αναλύοντας τις σχεδιαστικές αρχές, τα υπολογιστικά πεδία και τα σύνολα των μετροπρογραμμάτων (Κεφάλαιο 5).

Έπειτα, αναπτύσσεται ο κύριος κορμός της εργασίας, ο οποίος κατανέμεται σε πέντε διακριτά κεφάλαια αξιολόγησης συστημάτων. Ειδικότερα, εξετάζεται η εκτέλεση των σεναρίων μέσω του διερμηνέα `zsh` (Κεφάλαιο 6), αναλύονται οι στατικοί μετασχηματισμοί βάσει του Shark (Κεφάλαιο 7), μελετάται ο χειροκίνητος παραλληλισμός με χρήση του GNU `parallel` (Κεφάλαιο 8), αξιολογείται η δυναμική παραλληλοποίηση ροών δεδομένων του PaSh (Κεφάλαιο 9), και διερευνάται η υποθετική εκτέλεση εκτός σειράς (speculative out-of-order execution) του *hS* (Κεφάλαιο 10).

Τέλος, η εργασία ολοκληρώνεται με τη σύνθεση των ευρημάτων και τα τελικά συμπεράσματα που προκύπτουν από την αξιολόγηση (Κεφάλαιο 11), καθώς και με την παρουσίαση του προτεινόμενου μελλοντικού έργου για την επέκταση της έρευνας (Κεφάλαιο 12). Η υποδομή και τα μετροπρογράμματα της σουίτας Koala διατίθενται ελεύθερα με άδεια MIT στο `https://github.com/kbensh/koala`.

Κεφάλαιο 2

Υπόβαθρο

Σε αυτό το κεφάλαιο παρουσιάζεται το απαραίτητο θεωρητικό υπόβαθρο για την εις βάθος κατανόηση της συμπεριφοράς, της δομής και της πολυπλοκότητας των προγραμμάτων κελύφους Unix. Αρχικά, αναλύονται το περιβάλλον του κελύφους και οι βασικές αφαιρέσεις που διέπουν την εκτέλεση και τη σύνθεση εντολών. Στη συνέχεια, εξετάζονται οι εγγενείς ιδιαιτερότητες της γλώσσας, όπως η στενή εξάρτηση της συντακτικής ανάλυσης από τον χρόνο εκτέλεσης και οι αυστηροί κανόνες ανάπτυξης λέξεων (word expansions) του προτύπου POSIX. Μέσα από αυτή την ανάλυση αναδεικνύονται οι θεμελιώδεις περιορισμοί και οι τεχνικές προκλήσεις που καθιστούν την αυτόματη βελτιστοποίηση και τον παραλληλισμό των προγραμμάτων κελύφους ένα ιδιαίτερα απαιτητικό ερευνητικό πρόβλημα, προετοιμάζοντας το έδαφος για τη συστηματική μελέτη που ακολουθεί.

2.1 Το Κέλυφος Unix

Το κέλυφος Unix αποτελεί τον ακρογωνιαίο λίθο της αλληλεπίδρασης χρήστη-συστήματος εδώ και δεκαετίες, λειτουργώντας ταυτόχρονα ως ένας ισχυρός διαδραστικός διερμηνέας εντολών και ως μια εκφραστική γλώσσα προγραμματισμού υψηλού επιπέδου. Σύμφωνα με τη φιλοσοφία του Unix [19], παρέχει απλούς και γενικούς μηχανισμούς σύνθεσης μικρών, ανεξάρτητων εργαλείων σε μεγαλύτερα προγράμματα. Μέσω αυτού του περιβάλλοντος, οι χρήστες έχουν τη δυνατότητα να συνθέτουν και να εκτελούν πολύπλοκα προγράμματα, συνδυάζοντας μικρότερες, εξειδικευμένες και ανεξάρτητες εντολές σε σύνθετες ροές επεξεργασίας.

Η λειτουργία και η συμπεριφορά του κελύφους καθορίζονται από το πρότυπο POSIX, το οποίο θεσπίζει τους κανόνες σύνταξης και τις σημασιολογικές ιδιότητες, εξασφαλίζοντας τη διαλειτουργικότητα και τη φορητότητα προγραμμάτων μεταξύ διαφορετικών λειτουργικών συστημάτων και αρχιτεκτονικών [20]. Σε αντίθεση με τις παραδοσιακές γλώσσες προγραμματισμού γενικού σκοπού, όπως η C ή η Java, το κέλυφος δεν σχεδιάστηκε με πρωταρχικό στόχο την υλοποίηση πολύπλοκων αλγορίθμων ή δομών δεδομένων, αλλά για τον αποδοτικό συντονισμό και τη διασύνδεση υπαρχόντων προγραμμάτων, εργαλείων και υπηρεσιών συστήματος. Με αυτόν τον τρόπο, λειτουργεί ως ένα *συγκολλητικό* επίπεδο λογισμικού, επιτρέποντας στους προγραμματιστές να κατασκευάζουν γρήγορα λειτουργικούς αγωγούς επεξεργασίας δεδομένων, αξιοποιώντας έτοιμα και δοκιμασμένα δομικά στοιχεία. Η εκφραστικότητά του έχει αναδειχθεί στη βιβλιογραφία, καθώς εργασίες που απαιτούν εκτενή υλοποίηση σε γλώσσες όπως η Java μπορούν συχνά να εκφραστούν με μία μόνο εντολή κελύφους [21].

Σύγχρονη Χρήση και Σημασία Παρά την εμφάνιση νεότερων γλωσσών και εργαλείων, το κέλυφος διατηρεί αμείωτη τη σπουδαιότητά του και παραμένει αναπόσπαστο τμήμα της σύγχρονης πληροφορικής υποδομής. Κατατάσσεται συστηματικά ανάμεσα στις πιο δημοφιλείς γλώσσες προγραμματισμού [22], ενώ πρόσφατες μελέτες καταδεικνύουν σημαντική αύξηση της χρήσης του [14]. Το κέλυφος είναι πανταχού παρόν σε κρίσιμους τομείς, όπως η διαχείριση συστημάτων, η αυτοματοποίηση επαναλαμβανόμενων εργασιών, η μαζική επεξεργασία δεδομένων, η βιοπληροφορική, καθώς και η ενορχήστρωση υπολογιστικών ροών σε κατανεμημένα περιβάλλοντα και υποδομές νέφους. Ιδιαίτερα στο σύγχρονο οικοσύστημα ανάπτυξης λογισμικού, που κυριαρχείται από τεχνολογίες όπως το Docker και πλατφόρμες όπως το Kubernetes, το κέλυφος αποτελεί το καθιερωμένο πρότυπο για τον ορισμό σημείων

```
#!/bin/bash
mkfifo pipe1 pipe2
grep ”Failed” log1 > pipe1 & grep ”Failed” log2 > pipe2 &
cat pipe1 pipe2 | sort | uniq -c
```

Κώδικας 2.1: Οκνηρία σε ρεαλιστικό σενάριο επεξεργασίας καταλόγων. Η εντολή `cat` διαβάζει πρώτα από το `t1` και μόνο αφού ολοκληρώσει την ανάγνωση αυτού του FIFO προχωρά στο `t2`. Επομένως, η δεύτερη διεργασία `grep` μπλοκάρει, επειδή δεν υπάρχει ακόμη διεργασία που να διαβάζει από το `t2`, και συνεχίζει μόνο όταν τελειώσει η πρώτη `grep`.

εισόδου, την προετοιμασία περιβαλλόντων και την εκτέλεση σεναρίων συνεχούς ολοκλήρωσης [23]. Ως εκ τούτου, τα προγράμματα κελύφους δεν αποτελούν απλώς βοηθητικά εργαλεία, αλλά εκτελούνται συχνά σε υποδομές μεγάλης κλίμακας, επηρεάζοντας άμεσα την αξιοπιστία, την ταχύτητα και τη συνολική απόδοση κρίσιμων συστημάτων παραγωγής. Η εκτεταμένη αυτή χρήση καθιστά την κατανόηση των μηχανισμών του σε βάθος, καθώς και τη βελτιστοποίηση της εκτέλεσής του, ιδιαίτερα σημαντική για τη σύγχρονη έρευνα συστημάτων.

Βασικές Αφαιρέσεις και Ροές Δεδομένων Η εκφραστική ισχύς και η ευελιξία του κελύφους απορρέουν από ένα σύνολο απλών, αλλά ιδιαίτερα ισχυρών αφαιρέσεων, οι οποίες ενσαρκώνουν τη φιλοσοφία του Unix για τη σύνθεση μικρών και εξειδικευμένων εργαλείων. Κεντρικό ρόλο διαδραματίζουν οι ροές δεδομένων, οι οποίες υλοποιούνται ως σειριακές ακολουθίες bytes χωρίς εγγενή δομή και μεταφέρονται μεταξύ διεργασιών μέσω μηχανισμών όπως οι αγωγοί ή τα αρχεία. Οι ανώνυμοι αγωγοί επιτρέπουν την άμεση σύνδεση της τυπικής εξόδου (`stdout`) μιας εντολής με την τυπική είσοδο (`stdin`) μιας άλλης, δημιουργώντας σωληνώσεις (pipelines), ενώ οι επώνυμοι αγωγοί (Unix FIFOs) προσφέρουν μόνιμα κανάλια επικοινωνίας στο σύστημα αρχείων. Σε αυτό το πλαίσιο, ο πυρήνας του λειτουργικού συστήματος αναλαμβάνει τον ρόλο της διαχείρισης, της δρομολόγησης και του συγχρονισμού των δεδομένων, επιτρέποντας την ταυτόχρονη εκτέλεση των συνδεδεμένων εντολών χωρίς ο χρήστης να χρειάζεται να διαχειριστεί ρητά νήματα ή μηχανισμούς συγχρονισμού.

Επιπρόσθετα, κάθε εντολή στο περιβάλλον του κελύφους αντιμετωπίζεται ως μια αυτόνομη και ανεξάρτητη μονάδα υπολογισμού, η οποία διαβάζει δεδομένα, τα επεξεργάζεται και παράγει αποτελέσματα. Το σύνολο των διαθέσιμων εντολών είναι πρακτικά απεριόριστο, καθώς οποιοδήποτε εκτελέσιμο αρχείο στο σύστημα μπορεί να κληθεί ως εντολή. Οι εντολές μπορεί να είναι υλοποιημένες σε οποιαδήποτε γλώσσα προγραμματισμού (C, Python, Rust, κ. λπ.) ή να διατίθενται μόνο σε δυαδική μορφή, με άγνωστη εσωτερική δομή και συμπεριφορά. Το χαρακτηριστικό αυτό, αν και προσφέρει μεγάλη ευελιξία, καθιστά ιδιαίτερα δύσκολη την πρόβλεψη της επίδοσης και των απαιτήσεων σε πόρους χωρίς τη διενέργεια εμπειρικών μετρήσεων.

Για τον συντονισμό αυτών των μονάδων, το κέλυφος παρέχει μια σειρά από τελεστές σύνθεσης, όπως ο σειριακός τελεστής (`;`), ο τελεστής εκτέλεσης στο παρασκήνιο (`&`), ο τελεστής αγωγού (`|`) και οι λογικοί τελεστές (`&&`, `||`). Μέσω αυτών, ο προγραμματιστής μπορεί να ορίσει με ακρίβεια τις χρονικές και λογικές εξαρτήσεις μεταξύ των διεργασιών.

Ωστόσο, η διαχείριση της ροής δεδομένων παρουσιάζει και σημαντικές προκλήσεις. Πολλές εντολές σχεδιάζονται να είναι «οκνηρές» ή ακολουθούν αυστηρά σειριακή λογική κατανάλωσης, διαβάζοντας δεδομένα μόνο όταν είναι απολύτως έτοιμες να τα επεξεργαστούν. Αν και αυτή η συμπεριφορά είναι συχνά επιθυμητή για την ελαχιστοποίηση της χρήσης μνήμης, σε σύνθετους αγωγούς μπορεί να οδηγήσει σε φαινόμενα μπλοκαρίσματος και υποεκμετάλλευσης των διαθέσιμων υπολογιστικών πόρων, όπως φαίνεται στο παραπάνω παράδειγμα (Κώδικας 2.1).

Διάκριση Κελύφους και Εντολών Για την ορθή ανάλυση της απόδοσης, είναι θεμελιώδους σημασίας ο σαφής διαχωρισμός των αρμοδιοτήτων μεταξύ του ίδιου του διερμηνέα κελύφους και των επιμέρους

εντολών που αυτός καλεί. Ο διερμηνέας (π.χ. `bash`, `zsh`) είναι υπεύθυνος για τη συντακτική ανάλυση του κώδικα, την ανάπτυξη μεταβλητών, τη διαχείριση των περιγραφέων αρχείων και τη δημιουργία διεργασιών μέσω των κλήσεων συστήματος `fork` και `exec`. Αντιθέτως, ο κύριος υπολογιστικός φόρτος και η επεξεργασία των δεδομένων πραγματοποιούνται από τις εξωτερικές εντολές.

Ένα κρίσιμο στοιχείο που συχνά παραβλέπεται είναι η επίδραση των παραμέτρων εκτέλεσης στη συμπεριφορά των εντολών. Οι παράμετροι αυτές δεν μεταβάλλουν μόνο τη λειτουργικότητα, αλλά μπορούν να επηρεάσουν σημαντικά το προφίλ απόδοσης και τη δυνατότητα παραλληλισμού. Για παράδειγμα, μια εντολή που κανονικά επεξεργάζεται δεδομένα ως ροή μπορεί, με την προσθήκη κατάλληλων επιλογών, να απαιτήσει την πλήρη φόρτωση του αρχείου στη μνήμη ή να πραγματοποιήσει επιτόπιες τροποποιήσεις[1], μεταβάλλοντας ριζικά τον τρόπο αλληλεπίδρασης με το σύστημα αρχείων και τις υπόλοιπες διεργασίες.

Ιδιότυπη Συντακτική Ανάλυση και Σημασιολογία Σε αντίθεση με τις παραδοσιακές γλώσσες προγραμματισμού, η συντακτική ανάλυση (parsing) του κελύφους δεν μπορεί να διαχωριστεί αυστηρά από την εκτέλεσή του. Όπως έχει αναδειχθεί από έρευνες γύρω από τον σχεδιασμό του κελύφους ως γλώσσας ειδικού σκοπού (Domain-Specific Language) [25], η ερμηνεία του κώδικα εξαρτάται άμεσα από τη δυναμική κατάσταση του περιβάλλοντος κατά τον χρόνο εκτέλεσης. Για παράδειγμα, η ύπαρξη ενός ψευδωνύμου (`alias`) ή η δυναμική ανάθεση μιας μεταβλητής μπορεί να αλλάξει ριζικά τον τρόπο με τον οποίο ο διερμηνέας μεταφράζει την αμέσως επόμενη γραμμή κώδικα. Η αρχιτεκτονική αυτή ιδιαιτερότητα έχει καταστήσει ιστορικά εξαιρετικά δύσκολη τη δημιουργία εργαλείων στατικής ανάλυσης, καθώς η κατασκευή ενός ακριβούς αφηρημένου συντακτικού δέντρου (abstract syntax tree - AST) προϋποθέτει συχνά την εκτέλεση τμημάτων του ίδιου του προγράμματος.

Η Διαδικασία Ανάπτυξης Λέξεων Ένας από τους πιο κρίσιμους μηχανισμούς του κελύφους POSIX, ο οποίος ευθύνεται σε μεγάλο βαθμό για την πολυπλοκότητα και τη δυναμική του φύση, είναι η αυστηρή διαδικασία ανάπτυξης λέξεων που λαμβάνει χώρα *πριν* από την εκτέλεση οποιασδήποτε εντολής [25, 26]. Το πρότυπο ορίζει μια ακολουθία επτά διακριτών σταδίων μετασχηματισμού, τα οποία εφαρμόζονται ιεραρχικά:

1. Ανάπτυξη χαρακτήρα tilde (~).
2. Ανάπτυξη παραμέτρων και μεταβλητών (parameter expansion).
3. Υποκατάσταση εντολών (command substitution).
4. Αριθμητική ανάπτυξη (arithmetic expansion).
5. Διαχωρισμός πεδίων (field splitting).
6. Ανάπτυξη διαδρομών (pathname expansion/globbing).
7. Αφαίρεση εισαγωγικών (quote removal).

Η εν λόγω διαδικασία καθιστά τον κώδικα του κελύφους εξαιρετικά μεταβλητό. Μια απλή εντολή που συντακτικά φαντάζει ως μια ακίνδυνη μεταφορά δεδομένων, μπορεί—μετά το στάδιο της υποκατάστασης εντολών ή της ανάπτυξης διαδρομών—να μετατραπεί σε μια βαριά υπολογιστική εργασία που εμπλέκει χιλιάδες αρχεία στο σύστημα. Συνεπώς, κάθε προσπάθεια βελτιστοποίησης ή παραλληλισμού οφείλει να διασφαλίζει ότι αυτοί οι μετασχηματισμοί έχουν αξιολογηθεί ορθά, γεγονός που αποδεικνύει την ανεπάρκεια των αμιγώς στατικών προσεγγίσεων στην πλειονότητα των πραγματικών σεναρίων.

[1] Χαρακτηριστικό παράδειγμα αποτελεί η εντολή `sed`: στην τυπική της χρήση λειτουργεί ως φίλτρο, επιτρέποντας τον παραλληλισμό σε έναν αγωγό. Ωστόσο, η χρήση της σημαίας `-i` επιβάλλει την εγγραφή στο αρχικό αρχείο, απαιτώντας τη δημιουργία προσωρινών αρχείων και περιορίζοντας τις δυνατότητες παράλληλης επεξεργασίας [24].

Περιορισμοί και Προκλήσεις Παρά τη χρησιμότητα και την ευρεία διάδοσή του, το μοντέλο προγραμματισμού του κελύφους χαρακτηρίζεται από σημαντικούς περιορισμούς που δυσχεραίνουν την ανάπτυξη αποδοτικών και αξιόπιστων εφαρμογών:

- **Περιορισμένη Κλιμάκωση και Παραλληλισμός:** Αν και το κέλυφος υποστηρίζει βασικές μορφές παραλληλισμού μέσω αγωγών και παρασκηνιακής εκτέλεσης, η αποτελεσματική αξιοποίηση των σύγχρονων πολυπύρηνων αρχιτεκτονικών παραμένει δύσκολη. Ο χρήστης συχνά καταφεύγει σε πολύπλοκους χειροκίνητους χειρισμούς, χρησιμοποιώντας τελεστές και εντολές όπως `&` και `wait` [25] ή σε εξωτερικά εργαλεία, όπως το GNU `parallel` [18], διαδικασία που είναι χρονοβόρα και επιρρεπής σε σφάλματα.

- **Δυναμική και Αδιαφανής Συμπεριφορά:** Όπως προαναφέρθηκε, λόγω της ιδιότυπης σημασιολογίας και των σταδίων επέκτασης, η εκτέλεση ενός προγράμματος εξαρτάται σε μεγάλο βαθμό από την κατάσταση του περιβάλλοντος, την ύπαρξη και τα δικαιώματα αρχείων, καθώς και από ρυθμίσεις του συστήματος που μπορεί να μεταβληθούν δυναμικά. Αυτή η ιδιότητα, σε συνδυασμό με την αδιαφάνεια των εξωτερικών εκτελέσιμων, καθιστά την πρόβλεψη της συμπεριφοράς ιδιαίτερα δυσχερή.

- **Επιρρέπεια σε Σφάλματα και Περιορισμένη Ασφάλεια:** Το κέλυφος στερείται πολλών μηχανισμών ασφαλείας που συναντώνται σε σύγχρονες γλώσσες προγραμματισμού, όπως ισχυρό σύστημα τύπων ή δομημένη διαχείριση εξαιρέσεων. Ως αποτέλεσμα, απλά προγραμματιστικά λάθη, όπως η χρήση μη ορισμένων μεταβλητών σε κρίσιμες εντολές, μπορούν να επιφέρουν μη αναστρέψιμες συνέπειες. Χαρακτηριστικό παράδειγμα αποτελεί το κρίσιμο σφάλμα που εντοπίστηκε στο σενάριο εγκατάστασης της πλατφόρμας Steam για Linux [27], όπου η έλλειψη ελέγχου της μεταβλητής `STEAMROOT` οδήγησε στην ακούσια εκτέλεση της εντολής:

  ```
  rm -rf "$STEAMROOT/"*
  ```

  Σε περιπτώσεις όπου η μεταβλητή ήταν κενή, η εντολή μετατρεπόταν δυναμικά σε `rm -rf "/"*`, προκαλώντας τη διαγραφή ολόκληρου του συστήματος αρχείων του χρήστη.

Οι παραπάνω ιδιότητες συνθέτουν ένα περιβάλλον στο οποίο η συστηματική ανάλυση επιδόσεων και η αυτόματη βελτιστοποίηση αποτελούν ανοικτά και απαιτητικά ερευνητικά προβλήματα.

Κεφάλαιο 3

Σχετικό Έργο

Στο παρόν κεφάλαιο παρουσιάζεται η υφιστάμενη βιβλιογραφία και το ευρύτερο ερευνητικό τοπίο γύρω από το κέλυφος Unix. Το κεφάλαιο διαρθρώνεται σε δύο βασικούς άξονες. Αρχικά, εξετάζονται οι προσπάθειες βελτιστοποίησης και επιτάχυνσης των προγραμμάτων κελύφους, οι οποίες εκτείνονται από εναλλακτικούς διερμηνείς και παραδοσιακούς μεταγλωττιστές, έως σύγχρονα δυναμικά και κατανεμημένα συστήματα. Στη συνέχεια, αναλύεται το τοπίο της αξιολόγησης στον χώρο των συστημάτων και του κελύφους, αναδεικνύοντας τις ελλείψεις των υφιστάμενων συλλογών προγραμμάτων και τεκμηριώνοντας την ανάγκη για μια συστηματική, προσανατολισμένη στην απόδοση σουίτα, όπως το Koala.

3.1 Προσπάθειες Επιτάχυνσης και Βελτιστοποίησης

Η ανάγκη για ριζική βελτίωση της απόδοσης των προγραμμάτων κελύφους έχει πυροδοτήσει την ανάπτυξη πληθώρας συστημάτων, τα οποία επιχειρούν να υπερβούν τους περιορισμούς της αυστηρά σειριακής εκτέλεσης. Η αυξημένη ερευνητική δραστηριότητα γύρω από το κέλυφος Unix έχει οδηγήσει σε καινοτόμες λύσεις που καλύπτουν ένα ευρύτατο φάσμα τεχνικών. Ανάμεσα σε αυτές ξεχωρίζουν μέθοδοι για τη βελτιστοποίηση λειτουργιών εισόδου/εξόδου (I/O) [15], την αυτόματη σύνθεση (synthesis) νέων αγωγών [3], την αυτόματη παραλληλοποίηση σωληνώσεων [5, 7], καθώς και τη σύντηξη εντολών (fusion) με στόχο τη μείωση του κόστους επικοινωνίας μεταξύ των διεργασιών [28]. Παράλληλα, ιδιαίτερο ενδιαφέρον παρουσιάζουν οι μελέτες που αποκλίνουν από τα παραδοσιακά συστήματα, εστιάζοντας στην αποδοτική εκτέλεση σεναρίων σε κινητές συσκευές (mobile environments) [29] ή στον δυναμικό επαναπροσδιορισμό (refinement) των κλήσεων συστήματος (system calls) [30].

Ειδικότερα, στον τομέα της κλιμάκωσης (scaling) και της κατανεμημένης εκτέλεσης, η σύγχρονη έρευνα παρουσιάζει ρηξικέλευθες αρχιτεκτονικές λύσεις. Για παράδειγμα, το POSH [1] υιοθετεί τη στρατηγική της μεταφόρτωσης με επίγνωση δεδομένων (data-aware offloading), δρομολογώντας τον υπολογισμό απευθείας στην πηγή των δεδομένων προκειμένου να ελαχιστοποιηθεί η δικτυακή συμφόρηση. Στοχεύοντας σε υποδομές συστοιχιών (clusters), συστήματα όπως το DiSh [11] καθιστούν εφικτή την απρόσκοπτη εκτέλεση σε πολλαπλούς κόμβους, ενώ το Fractal [12] ενσωματώνει επιπλέον κρίσιμους μηχανισμούς ανοχής σφαλμάτων (fault tolerance). Διευρύνοντας περαιτέρω αυτό το πεδίο, πλαίσια όπως το SPLaSh [24] εισάγουν το κέλυφος στο οικοσύστημα του υπολογιστικού νέφους (cloud computing), ερευνώντας τη δυναμική και ελαστική του κλιμάκωση μέσω μοντέλων Function-as-a-Service (FaaS).

Η παρούσα εργασία δεν αποσκοπεί σε μια εξαντλητική καταγραφή όλων των παραπάνω κατηγοριών. Αντιθέτως, επικεντρώνεται σε πέντε αντιπροσωπευτικά περιβάλλοντα που εστιάζουν στην επιτάχυνση σε επίπεδο μεμονωμένου συστήματος (single-node) και ενσαρκώνουν θεμελιωδώς διαφορετικές σχεδιαστικές φιλοσοφίες. Αυτά τα συστήματα, τα οποία αποτελούν και το αντικείμενο της εκτενούς αξιολόγησης στα επόμενα κεφάλαια, είναι τα εξής:

- **zsh** [17]: Ένας εναλλακτικός, εξαιρετικά παραμετροποιήσιμος διερμηνέας. Παρότι αναπτύχθηκε με γνώμονα τη βελτίωση της διαδραστικής εμπειρίας του χρήστη συγκριτικά με το bash, υλοποιεί μια διαφορετική εσωτερική μηχανή διαχείρισης μνήμης και συντακτικής ανάλυσης, και

μπορεί να επηρεάσει την ταχύτητα των σεναρίων με ενσωματωμένες εντολές κελύφους (shell built-ins).

- **Shark [15]:** Αποτελεί ένα από τα πρώτα ερευνητικά συστήματα που αντιμετώπισαν τα προγράμματα κελύφους με εργαλεία δανεισμένα από τη θεωρία των παραδοσιακών μεταγλωττιστών. Η βασική του καινοτομία έγκειται στην αντιμετώπιση των προσπελάσεων στο σύστημα αρχείων ως αναφορές σε μεταβλητές. Μέσω της κατασκευής κατευθυνόμενων ακυκλικών γραφημάτων (DAGs), το Shark εφαρμόζει στατικές βελτιστοποιήσεις, όπως η εξάλειψη περιττών λειτουργιών (π.χ. η άσκοπη χρήση της εντολής `cat`), η αντικατάσταση προσωρινών αρχείων με αγωγούς για τη μείωση του κόστους I/O, και ο στατικός παραλληλισμός ανεξάρτητων εντολών.

- **GNU `parallel` [18]:** Αποτελεί το ευρύτερα αποδεκτό εργαλείο για τη ρητή, χειροκίνητη παραλληλοποίηση στο οικοσύστημα του Unix. Παρέχει στους προγραμματιστές προηγμένους μηχανισμούς για τη διάσπαση δεδομένων και την ταυτόχρονη εκτέλεση πολλαπλών διεργασιών. Ωστόσο, η εντυπωσιακή του απόδοση συνοδεύεται από μια κρίσιμη απαίτηση: η ευθύνη για τη διατήρηση των εξαρτήσεων δεδομένων και τη διασφάλιση της λειτουργικής ορθότητας μετατίθεται αποκλειστικά στον χρήστη.

- **PaSh [7]:** Αντιπροσωπεύει την αιχμή της σύγχρονης έρευνας στην αυτόματη παραλληλοποίηση μέσω μεταγλώττισης πάνω-στην-ώρα (Just-in-Time). Το σύστημα εναλλάσσει δυναμικά φάσεις ανάλυσης και εκτέλεσης, μετασχηματίζοντας τις σωληνώσεις σε γράφους ροής δεδομένων (Data-Flow Graphs). Βασιζόμενο σε μια βιβλιοθήκη επισημειώσεων (annotations) που κωδικοποιεί τη συμπεριφορά των εντολών POSIX, το PaSh εισάγει παράλληλα σχήματα εκτέλεσης, διασφαλίζοντας τη σημασιολογική ισοδυναμία (semantic equivalence) με την αρχική σειριακή μορφή του σεναρίου.

- ***hS* [14]:** Προτείνει μια ριζοσπαστική αρχιτεκτονική που αντλεί έμπνευση από τη μικροαρχιτεκτονική των σύγχρονων επεξεργαστών, εισάγοντας την υποθετική εκτέλεση εκτός σειράς (speculative out-of-order execution) στο επίπεδο του λειτουργικού συστήματος. Το *hS* εκτελεί προκαταβολικά μελλοντικές εντολές σε απομονωμένα περιβάλλοντα (sandboxes). Μέσω συστηματικής ιχνηλάτησης (tracing) του I/O και των μεταβλητών περιβάλλοντος, κατοχυρώνει (commits) τα αποτελέσματα εφόσον δεν ανιχνευθούν συγκρούσεις, ενώ σε αντίθετη περίπτωση αναιρεί την εκτέλεση (rollback) και επιστρέφει στην ασφαλή σειριακή ροή.

Οι προσεγγίσεις αυτές αποδεικνύουν ότι ο χώρος βελτιστοποίησης του κελύφους είναι πολυδιάστατος. Εντούτοις, η τελική αποτελεσματικότητα κάθε συστήματος εξαρτάται άρρηκτα από τη συντακτική δομή, τον όγκο δεδομένων και τη δυναμική συμπεριφορά του εκάστοτε προγράμματος. Το γεγονός αυτό καθιστά αναγκαία τη χρήση ενός ρεαλιστικού και εκτενούς πλαισίου αξιολόγησης, προκειμένου να αποτυπωθούν συστηματικά αυτά τα όρια.

3.2 Το Τοπίο της Συγκριτικής Αξιολόγησης

Η πρόοδος στον σχεδιασμό υπολογιστικών συστημάτων είναι άμεσα συνυφασμένη με τη δυνατότητα αξιόπιστης, επαναλήψιμης και δίκαιης συγκριτικής αξιολόγησης (benchmarking). Η ύπαρξη κοινά αποδεκτών σουιτών (benchmark suites) επιτρέπει την αντιπαραβολή διαφορετικών αρχιτεκτονικών υπό κοινές συνθήκες, τη διασταύρωση πειραματικών ευρημάτων και τη συστηματική μελέτη των συμβιβασμών (trade-offs) μεταξύ απόδοσης και χρήσης πόρων [16]. Ιστορικά, η καθιέρωση τέτοιων υποδομών αποτέλεσε τον καταλύτη για την ωρίμανση πολλών ερευνητικών πεδίων, αντικαθιστώντας τα αποσπασματικά πειράματα με αυστηρές, τυποποιημένες μεθοδολογίες.

Καθιερωμένες Σουίτες στην Έρευνα Συστημάτων

Η πρόοδος στην επιστήμη των υπολογιστών και στο ευρύτερο πεδίο της έρευνας συστημάτων εξαρτάται σε μεγάλο βαθμό από τη δυνατότητα δίκαιων και *αντικειμενικών συγκρίσεων*. Σε αυτό το

πλαίσιο, η χρήση ανοιχτών και επαναχρησιμοποιήσιμων τυποποιημένων μετροπρογραμμάτων αποτελεί το θεμέλιο της επιστημονικής αξιολόγησης. Σε διάφορους τομείς, η καθιέρωση τέτοιων εργαλείων υπήρξε καταλυτική: πρότυπα αξιολόγησης υλικού όπως τα SPEC [31], βάσεων δεδομένων όπως το TPC [32], πολυπύρηνων αρχιτεκτονικών όπως το PARSEC [33] και εφαρμογών υπολογιστικού νέφους όπως το CloudSuite [34], έχουν θεσπίσει αυστηρές διαδικασίες επικύρωσης και κοινά σημεία αναφοράς.

Αντίστοιχα, στον χώρο του λογισμικού και των γλωσσών προγραμματισμού, η ανάγκη για ρεαλιστικά φορτία εργασίας οδήγησε στη δημιουργία εξειδικευμένων εργαλείων. Σουίτες όπως η DaCapo [35, 36] για τη Java, καθώς και τα μετροπρογράμματα Gabriel [37] για τη LISP, επέτρεψαν τη σε βάθος μελέτη διαχειριζόμενων περιβαλλόντων εκτέλεσης σε συνθήκες παραγωγής (όπως η συλλογή απορριμμάτων και η συμπεριφορά αντικειμενοστρεφών ιεραρχιών [38]). Παράλληλα, η αναπαραγωγιμότητα σύγχρονων ερευνών υποστηρίζεται ενεργά από σύνολα δεδομένων αναφοράς σε διάφορα πεδία, όπως το EMNIST [39] στη μηχανική μάθηση, το Defects4J [40] στη δοκιμή λογισμικού και το Magma [41] στην αξιολόγηση εργαλείων ασφάλειας. Κοινό χαρακτηριστικό όλων αυτών των επιτυχημένων προσπαθειών είναι η έμφαση σε ολοκληρωμένες εφαρμογές και σαφώς ορισμένες διαδικασίες εκτέλεσης. Σε αντίθεση με τις μικροδοκιμές επίδοσης που απομονώνουν μεμονωμένες λειτουργίες, οι ολοκληρωμένες σουίτες αποτυπώνουν τις περίπλοκες αλληλεπιδράσεις ενός συστήματος, αναδεικνύοντας φαινόμενα που δεν εμφανίζονται σε συνθετικά σενάρια. Η ύπαρξη μιας καθιερωμένης σουίτας επιταχύνει ουσιαστικά την εξέλιξη του εκάστοτε ερευνητικού χώρου.

Σε προγενέστερη έρευνα έχει μελετηθεί η συμπεριφορά προγραμμάτων είτε σε περιβάλλοντα προσομοίωσης είτε πάνω σε συγκεκριμένες υλοποιήσεις υλικού [42, 43, 44], ωστόσο μια τέτοια προσέγγιση συνδέει τα συμπεράσματα με τη συγκεκριμένη πλατφόρμα εκτέλεσης και το αντίστοιχο λειτουργικό περιβάλλον. Στον χώρο του κελύφους απαιτείται η συστηματική ανάδειξη ιδιοτήτων που είναι ανεξάρτητες από το εκάστοτε υλικό (hardware) και το λειτουργικό σύστημα, ώστε τα προγράμματα να μπορούν να χρησιμοποιηθούν ως γενικά και φορητά φορτία εργασίας για την αξιολόγηση διαφορετικών υπολογιστικών υποστρωμάτων. Κατά συνέπεια, καθίσταται αναγκαία η ύπαρξη μιας ανοιχτής και προσεκτικά σχεδιασμένης σουίτας αξιολόγησης ειδικά προσανατολισμένης σε προγράμματα κελύφους.

Όπως ακριβώς περιπτώσεις σαν τα DaCapo και Gabriel κάλυψαν ιστορικά κενά στην αξιολόγηση των προγραμματιστικών περιβαλλόντων της εποχής τους, το σύστημα Koala έρχεται να καλύψει μια αντίστοιχη σύγχρονη ανάγκη. Αποτελεί την πρώτη συστηματική προσπάθεια δημιουργίας μιας ανοιχτής σουίτας αξιολόγησης επιδόσεων για το κέλυφος, εισάγοντας τη συστηματοποίηση, τη συγκρισιμότητα και την αναπαραγωγιμότητα σε έναν τομέα που μέχρι σήμερα στερούνταν αντίστοιχων προτύπων.

Υφιστάμενες Προσεγγίσεις και Ελλείψεις Αξιολόγησης στο Κέλυφος

Η πληθώρα των συστημάτων βελτιστοποίησης που παρουσιάστηκαν προηγουμένως καταδεικνύει την επιτακτική ανάγκη για ένα τυποποιημένο, εύχρηστο και αναπαραγώγιμο πλαίσιο αξιολόγησης. Εντούτοις, ελλείψει μιας καθιερωμένης μεθοδολογίας, οι δημιουργοί αυτών των συστημάτων είχαν στη διάθεσή τους εξαιρετικά περιορισμένες επιλογές προκειμένου να τεκμηριώσουν τα οφέλη των προσεγγίσεών τους.

Οι διαθέσιμες πρακτικές που χρησιμοποιούνται για το κέλυφος ταξινομούνται σε τέσσερις βασικές κατηγορίες, καθεμία από τις οποίες παρουσιάζει θεμελιώδεις ελλείψεις όσον αφορά την αποτίμηση επιδόσεων σε ρεαλιστικά συστήματα:

1. **Μικροδοκιμές (Shell Microbenchmarks):** Σουίτες όπως το ShellBench [45], το zsh-bench [46], τα μετροπρογράμματα του Oils [47] και το UnixBench [48] παρέχουν μικρά, απομονωμένα τμήματα κώδικα (συνήθως 8 έως 95 γραμμών). Αν και είναι αναντικατάστατα εργαλεία για την άσκηση πίεσης σε μεμονωμένα χαρακτηριστικά του διερμηνέα (π.χ. ταχύτητα υποκελύφων, αντικατάσταση μεταβλητών ή μαθηματικές πράξεις), υστερούν σημαντικά σε ρεαλισμό. Αποκλίνουν ριζικά από τα πραγματικά φορτία εργασίας, καθώς δεν εκτελούν ολοκληρωμένους υπολο-

γισμούς (end-to-end), δεν διαχειρίζονται μεγάλα σύνολα δεδομένων και δεν ενορχηστρώνουν ετερογενείς εξωτερικές εντολές, καθιστώντας αδύνατη την ολιστική αξιολόγηση των συστημάτων επιτάχυνσης.

2. **Δοκιμασίες Προτύπων και Ορθότητας (Correctness & Standards Tests):** Εργαλεία όπως η σουίτα δοκιμών POSIX [49], η σουίτα Smoosh [8], οι διαγνωστικές ρουτίνες της βιβλιοθήκης Modernish [50], καθώς και διάφορα πλαίσια αυτοματοποιημένου ελέγχου (όπως τα shellspec [51], shunit [52], bats-core [53] και tetworks [54]), διασφαλίζουν την αυστηρή συμμόρφωση της υλοποίησης ενός κελύφους με τα πρότυπα. Επικεντρώνονται, ωστόσο, αποκλειστικά στον έλεγχο της λειτουργικότητας, παρά στον χαρακτηρισμό συστημάτων σε περιβάλλοντα που περιλαμβάνουν αδιαφανή (opaque) στοιχεία. Αδυνατώντας να προσομοιώσουν το μέγεθος, την πολυπλοκότητα και τα δεδομένα εισόδου των πραγματικών προγραμμάτων, δεν προσφέρουν απολύτως καμία πληροφορία για το πώς ένα σύστημα βελτιστοποίησης επηρεάζει την πραγματική απόδοση στον χρόνο εκτέλεσης.

3. **Αναλύσεις Ανοιχτού Κώδικα και Εμπειρικές Μελέτες (Open-source / Empirical Studies):** Πρόσφατες μελέτες συγκεντρώνουν και αναλύουν εκατομμύρια προγράμματα (όπως από το GitHub ή προγράμματα δόμησης Linux) για να κατανοήσουν πραγματικά σενάρια χρήσης του κελύφους. Μελετούν τη χρήση ψευδωνύμων (aliases), τις πρακτικές ασφαλείας και την αλληλεπίδραση των χρηστών με τον κώδικα [55, 56]. Παρά την ανεκτίμητη στατική τους αξία, αυτά τα αποθετήρια δεν παρέχουν την απαραίτητη υποδομή για την εκτέλεσή τους. Τυπικά δεν διαθέτουν δεδομένα εισόδου, σενάρια αρχικοποίησης (setup scripts) ή ρητές δηλώσεις εξαρτήσεων, αποτελώντας συχνά θορυβώδη ή ημιτελή τμήματα κώδικα που δεν μπορούν να χρησιμοποιηθούν για δυναμική συγκριτική αξιολόγηση.

4. **Αυτοσχέδιες Συλλογές (Ad-hoc Collections & Evaluations):** Απουσία κοινής υποδομής αξιολόγησης για τα συστήματα επιτάχυνσης που παρουσιάστηκαν στην Ενότητα 3.1, η ιστορικά επικρατέστερη πρακτική υπήρξε η δημιουργία αυτοσχέδιων συνόλων, συχνά επιλεγμένων με το χέρι [1, 7, 11, 15, 29]. Εκτός από τον προφανή κίνδυνο μεροληψίας (bias) υπέρ του εκάστοτε συστήματος, αυτές οι μικρές και κατακερματισμένες συλλογές παρουσιάζουν έλλειψη μηχανισμών αυτοματοποιημένης εγκατάστασης και επαλήθευσης. Αυτό οδηγεί σε άδικες και ασύμβατες συγκρίσεις, αναδεικνύοντας περαιτέρω την επιτακτική ανάγκη για μια τυποποιημένη σουίτα.

Ως αποτέλεσμα, η έλλειψη μιας τυποποιημένης, ρεαλιστικής και επαναχρησιμοποιήσιμης σουίτας αξιολόγησης έχει οδηγήσει σε ένα κατακερματισμένο τοπίο, όπου οι ισχυρισμοί για βελτιώσεις στην απόδοση συχνά βασίζονται σε αποσπασματικές μετρήσεις που δεν μπορούν να αναπαραχθούν ή να συγκριθούν με άλλες προσεγγίσεις. Αυτή η κατάσταση υπονομεύει την επιστημονική πρόοδο, καθώς καθιστά δύσκολη την αντικειμενική αξιολόγηση των νέων ιδεών και την κατανόηση των πραγματικών ορίων των συστημάτων επιτάχυνσης.

Ο Ρόλος και η Συνεισφορά του Koala Η σουίτα Koala σχεδιάστηκε με στόχο να καλύψει αυτό το κενό, υιοθετώντας αρχές που έχουν αποδειχθεί επιτυχείς σε άλλα πεδία της έρευνας συστημάτων. Συγκεκριμένα, περιλαμβάνει ρεαλιστικά προγράμματα κελύφους προερχόμενα από πραγματικές εργασίες, καλύπτει πολλαπλά υπολογιστικά πεδία και παρέχει διαφορετικές κλίμακες εισόδου, επιτρέποντας τόσο ταχεία διερεύνηση και επιβεβαίωση ορθότητας, όσο και εκτενή αξιολόγηση. Παράλληλα, συνοδεύεται από αυτοματοποιημένη υποδομή εγκατάστασης, εκτέλεσης και επικύρωσης αποτελεσμάτων, διασφαλίζοντας την αναπαραγωγιμότητα. Η ανάλυση της σουίτας δείχνει ότι καλύπτεται το σύνολο των βασικών χαρακτηριστικών του κελύφους POSIX, ενώ τα προγράμματα παρουσιάζουν σημαντική ποικιλία τόσο σε στατικά όσο και σε δυναμικά χαρακτηριστικά, όπως τον χρόνο επεξεργαστή (CPU), την κατανάλωση μνήμης και τη δραστηριότητα εισόδου/εξόδου (I/O). Τα φορτία εργασίας κλιμακώνονται από σύντομα σενάρια έως εκτελέσεις μεγάλης διάρκειας σε εκτεταμένα σύνολα δεδομένων. Με τον τρόπο αυτό, το Koala επιτρέπει τη μετάβαση από αποσπασματικές μετρήσεις σε μια συνεκτική, αναπαραγώγιμη και επαναχρησιμοποιήσιμη μεθοδολογία αξιολόγησης. Το πλαίσιο αυτό καθι-

στά δυνατή τη συστηματική σύγκριση διαφορετικών προσεγγίσεων επιτάχυνσης στο ίδιο, ελεγχόμενο περιβάλλον.

Κεφάλαιο 4

Κινητήριο Παράδειγμα

Για να γίνει εμφανής ο τρόπος με τον οποίο η σουίτα Koala υποστηρίζει τη συγκριτική αξιολόγηση διαφορετικών συστημάτων επιτάχυνσης κελύφους, είναι ωφέλιμο να εξεταστεί ένα αντιπροσωπευτικό παράδειγμα. Η ανάλυση ενός ρεαλιστικού σεναρίου αναδεικνύει τη φιλοσοφία σχεδιασμού, τις προκλήσεις εκτέλεσης και τεκμηριώνει την ανάγκη για μια ολοκληρωμένη υποδομή αξιολόγησης.

4.1 Αναλυτική Λειτουργία του Σεναρίου Μετεωρολογικών Δεδομένων

Το κινητήριο παράδειγμα—το σενάριο `weather`—βασίζεται στο σενάριο `temp-analytics.sh`, το οποίο ανήκει στο σύνολο μετροπρογραμμάτων `weather` της σουίτας. Ο Κώδικας 4.1 παρουσιάζει την απλοποιημένη μορφή αυτού του σεναρίου.

```
#!/bin/bash

d="./data/temperatures";
for y in $(seq $start $end); do
  cat $d/$y | cut -c 89-92 | grep -v 999 | sort -rn | head -n1 > max.$y
  cat $d/$y | cut -c 89-92 | grep -v 999 | sort -n | head -n1 > min.$y
  cat $d/$y | cut -c 89-92 | grep -v 999 | awk "{t += \$1; i++} END {print t/i}" >
  ↪ avg.$y
done
```

Κώδικας 4.1: Παράδειγμα ανάλυσης μετεωρολογικών δεδομένων στο Koala (`weather`). Υλοποίηση υπολογισμού μέγιστης, ελάχιστης και μέσης θερμοκρασίας μέσω επαναληπτικής δομής και ανεξάρτητων σωληνώσεων.

Η αρχιτεκτονική και η εκτέλεση του συγκεκριμένου προγράμματος βασίζονται σε έναν συνδυασμό μιας επαναληπτικής δομής ελέγχου και ροών επεξεργασίας δεδομένων μέσω σωληνώσεων. Πιο συγκεκριμένα, ο κεντρικός πυρήνας του σεναρίου είναι ένας βρόχος `for`, ο οποίος, μέσω της εντολής `seq`, διατρέχει διαδοχικά μια καθορισμένη ακολουθία ετών (από το `$start` έως το `$end`). Κατά τη διάρκεια κάθε επανάληψης, το πρόγραμμα αναλύει το αντίστοιχο αρχείο ιστορικών μετεωρολογικών δεδομένων (`$d/$y`) για να εξάγει τρία στατιστικά στοιχεία: τη μέγιστη, την ελάχιστη και τη μέση θερμοκρασία του έτους. Για την επίτευξη αυτού του υπολογισμού, το σώμα του βρόχου απαρτίζεται από τρεις διακριτές, ανεξάρτητες σωληνώσεις. Κάθε μία από αυτές τις τρεις σωληνώσεις ξεκινά με ακριβώς τα ίδια στάδια προεπεξεργασίας:

1. Η εντολή `cat` διαβάζει τα δεδομένα του αρχείου.

2. Η εντολή `cut -c 89-92` εξάγει τους συγκεκριμένους χαρακτήρες κάθε γραμμής όπου καταγράφεται η θερμοκρασία.

3. Η εντολή `grep -v 999` φιλτράρει και απορρίπτει τις εγγραφές που περιέχουν την τιμή `999`, η οποία λειτουργεί ως δείκτης για ελλιπή ή μη έγκυρα δεδομένα.

Από αυτό το σημείο, τα «καθαρισμένα» δεδομένα ακολουθούν διαφορετική πορεία ανάλογα με τη ζητούμενη μετρική:

- Για τη **μέγιστη θερμοκρασία**, τα δεδομένα ταξινομούνται κατά φθίνουσα αριθμητική σειρά (`sort -rn`), και η `head -n1` απομονώνει την πρώτη γραμμή, αποθηκεύοντάς τη στο αρχείο `max.$y`.
- Για την **ελάχιστη θερμοκρασία**, εφαρμόζεται αύξουσα αριθμητική ταξινόμηση (`sort -n`) και η `head -n1` εξάγει το αποτέλεσμα στο `min.$y`.
- Για τη **μέση θερμοκρασία**, τα δεδομένα τροφοδοτούνται στην `awk`, η οποία αθροίζει τις τιμές (`t += \$1`), μετράει το πλήθος τους (`i++`) και εκτυπώνει τον μέσο όρο (`t/i`) στο τέλος της ανάγνωσης (στο μπλοκ `END`).

Το γεγονός ότι οι συγκεκριμένες σωληνώσεις είναι δομικά ανεξάρτητες μεταξύ τους εντός του βρόχου, αναδεικνύει το σενάριο σε ιδανικό πεδίο δοκιμών για τη μελέτη διαφορετικών στρατηγικών επιτάχυνσης.

4.2 Πρακτικά Χαρακτηριστικά του Σεναρίου

Το συγκεκριμένο πρόγραμμα επιλέχθηκε επειδή συγκεντρώνει με απτό τρόπο τα πρακτικά χαρακτηριστικά ενός ρεαλιστικού φόρτου εργασίας, ενσαρκώνοντας τις σχεδιαστικές αρχές του Koala:

- **Αντιπροσωπευτικότητα και Ρεαλισμός:** Το σενάριο δεν αποτελεί μια τεχνητή μικροδοκιμή. Αντίθετα, εκτελεί πραγματικούς στατιστικούς υπολογισμούς πάνω σε ένα ευρέως γνωστό σύνολο ιστορικών δεδομένων καιρού—το NOAA [57]. Επίσης, χρησιμοποιείται ως παράδειγμα αναφοράς σε ένα κλασικό βιβλίο για το οικοσύστημα του Hadoop [58].
- **Κλιμάκωση (Scalability):** Η εγγενής δυνατότητά του να αναλύει αρχεία επαναληπτικά για πολλαπλά έτη σημαίνει ότι ο φόρτος εργασίας του μπορεί να αυξομειωθεί εύκολα (π.χ. αναλύοντας μια δεκαετία έναντι ενός αιώνα δεδομένων). Η υποδομή του Koala αξιοποιεί αυτό το χαρακτηριστικό για να παρέχει δεδομένα σε μικρή ή μεγάλη κλίμακα.
- **Ποικιλομορφία Δομών (Diversity):** Η αρμονική συνύπαρξη μιας παραδοσιακής δομής ελέγχου (ο βρόχος `for`) με ροές δεδομένων (οι πολλαπλές σωληνώσεις) και ποικίλα εργαλεία POSIX (`cut`, `grep`, `sort`, `awk`), συνθέτει ένα απαιτητικό περιβάλλον που δοκιμάζει εξαντλητικά την ικανότητα οποιουδήποτε εργαλείου.

4.3 Πεδίο Δοκιμών για Προσεγγίσεις Βελτιστοποίησης

Αυτή ακριβώς η ρεαλιστική σύνθεση του κώδικα προσφέρει το ιδανικό πεδίο για να αναδειχθούν οι δυνατότητες, αλλά και οι διαφορετικές απαιτήσεις υιοθέτησης, των σύγχρονων εργαλείων βελτιστοποίησης. Κάθε σύστημα προσεγγίζει και επιταχύνει το ίδιο ακριβώς σενάριο με ριζικά διαφορετικό τρόπο, βελτιστοποιώντας διαφορετικές δομές:

- Ο διερμηνέας **zsh** εκτελεί το σενάριο ως έχει, επηρεάζοντας την απόδοση των ενσωματωμένων λειτουργιών του κελύφους (shell built-ins), όπως η εκχώρηση μεταβλητών και ο βρόχος `for`.
- Το σύστημα **Shark** απαιτεί χειροκίνητη παρέμβαση στον κώδικα από τον συγγραφέα. Αφαιρεί τις περιττές κλήσεις της `cat` (μεταφέροντας την είσοδο απευθείας στην `cut`) και εκτελεί τις τρεις σωληνώσεις παράλληλα στο παρασκήνιο.

- Το εργαλείο **GNU** `parallel` απαιτεί επίσης χειροκίνητη αναδιαμόρφωση του κώδικα. Ο χρήστης καλείται να επιλέξει ποιο τμήμα θα παραλληλίσει, εφαρμόζοντάς το είτε στον εξωτερικό βρόχο είτε στο εσωτερικό των τριών σωληνώσεων.
- Το σύστημα **PaSh** παραλληλοποιεί αυτόματα τα επιμέρους υπολογιστικά στάδια *μέσα* σε κάθε μεμονωμένη σωλήνωση χρησιμοποιώντας έναν μεταγλωττιστή πάνω-στην-ώρα (Just-in-Time).
- Το σύστημα ***hS*** αξιοποιεί την υποθετική εκτέλεση (speculative execution) για να «ξετυλίξει» δυναμικά τις επαναλήψεις του βρόχου (loop unrolling) και να εκτελέσει τις εντολές ταυτόχρονα και εκτός σειράς (out-of-order).

Καθίσταται σαφές ότι η πολυπλοκότητα αυτού του σεναρίου καταδεικνύει εμπράκτως τόσο τα αναμενόμενα οφέλη (speedups) όσο και τους δυνητικούς περιορισμούς της κάθε προσέγγισης σε ρεαλιστικές συνθήκες παραγωγής, αναδεικνύοντας γιατί δεν αρκούν οι απλές μικροδοκιμές.

4.4 Η Ανάγκη για μια Ολοκληρωμένη Σουίτα

Όπως καταδεικνύει η ανάλυση του παραπάνω παραδείγματος, η εκτέλεση σε ρεαλιστικά σενάρια είναι ο μοναδικός αξιόπιστος δρόμος για να αποκαλυφθεί η πραγματική συμπεριφορά κάθε εργαλείου βελτιστοποίησης. Ωστόσο, προκειμένου η αξιολόγηση αυτή να είναι επιστημονικά ορθή και αναπαραγώγιμη, οι ερευνητές χρειάζονται τη δυνατότητα να εκτελούν τα ίδια ακριβώς σενάρια, με κοινά δεδομένα και κάτω από ένα απόλυτα ελεγχόμενο περιβάλλον.

Αυτή η διαπίστωση καθιστά επιτακτική την ανάγκη μετάβασης από τα μεμονωμένα, σποραδικά παραδείγματα σε μια οργανωμένη και πλήρως αυτοματοποιημένη υποδομή. Αντί να αναλώνεται χρόνος στην κατασκευή ιδιόκτητων και μη συγκρίσιμων φορτίων εργασίας για κάθε νέο σύστημα, απαιτείται ένα ενιαίο πλαίσιο που θα παρέχει άμεσα, δίκαια και συγκρίσιμα αποτελέσματα. Με γνώμονα αυτή την ανάγκη, στο επόμενο κεφάλαιο παρουσιάζεται αναλυτικά η αρχιτεκτονική και η υλοποίηση της σουίτας Koala, ορίζοντας αυστηρά τις σχεδιαστικές αρχές της και επεκτείνοντας τη φιλοσοφία αυτού του κινητήριου παραδείγματος σε ένα ευρύ φάσμα 126 πραγματικών προγραμμάτων.

Κεφάλαιο 5

Σχεδιασμός και Υλοποίηση της Σουίτας Koala

Έχοντας παρουσιάσει ένα αντιπροσωπευτικό παράδειγμα και ένα δείγμα της φιλοσοφίας της σουίτας στο προηγούμενο κεφάλαιο, το παρόν κεφάλαιο εστιάζει στην αναλυτική παρουσίαση του Koala. Αρχικά, ορίζονται οι βασικές σχεδιαστικές του αρχές και η μεθοδολογία επιλογής των μετροπρογραμμάτων. Στη συνέχεια, περιγράφεται η αρχιτεκτονική της σουίτας, η οποία περιλαμβάνει συνολικά 126 προγράμματα πραγματικής χρήσης από ποικίλες πηγές και υπολογιστικά πεδία, η υποδομή που εξασφαλίζει την αυτοματοποιημένη και αναπαραγώγιμη εκτέλεσή τους, και τέλος ένα τμήμα της στατικής και δυναμικής ανάλυσης που αναδεικνύει την ποικιλομορφία της.

5.1 Σχεδιαστικές Αρχές

Ο σχεδιασμός του Koala καθοδηγήθηκε από ένα σύνολο θεμελιωδών αρχών, οι οποίες αποτυπώνουν τις βασικές απαιτήσεις για μια σύγχρονη σουίτα αξιολόγησης συστημάτων κελύφους.

- **Αντιπροσωπευτικότητα:** Τα σενάρια της σουίτας αποτελούν πραγματικά προγράμματα που εκτελούν ουσιαστικούς υπολογισμούς πάνω σε πραγματικά δεδομένα. Η επιλογή αυτή επιτρέπει τη μελέτη της συμπεριφοράς των συστημάτων σε συνθήκες κοντά στη χρήση παραγωγής, σε αντίθεση με απομονωμένες μικροδοκιμές επίδοσης.
- **Ποικιλομορφία:** Η σουίτα καλύπτει ένα ευρύ φάσμα υπολογιστικών πεδίων, όπως ανάλυση δεδομένων, βιοπληροφορική, διαχείριση συστημάτων και μηχανική μάθηση, με διαφορετικά προφίλ εκτέλεσης, ενώ αξιοποιεί μεγάλο εύρος εντολών και χαρακτηριστικών του κελύφους. Με τον τρόπο αυτό αποφεύγεται η υπερπροσαρμογή της αξιολόγησης σε συγκεκριμένα μοτίβα προγραμμάτων.
- **Χαρακτηρισμός Φορτίων:** Εκτός από την παροχή μετροπρογραμμάτων, η σουίτα συνοδεύεται από συστηματικό χαρακτηρισμό τόσο των στατικών όσο και των δυναμικών ιδιοτήτων τους. Η προσέγγιση αυτή επιτρέπει την κατανόηση του τρόπου με τον οποίο διαφορετικά συστήματα επηρεάζουν την εκτέλεση πραγματικών προγραμμάτων.
- **Κλιμάκωση:** Η σουίτα παρέχει δεδομένα εισόδου σε πολλαπλές κλίμακες (ελάχιστη, μικρή, μεγάλη), επιτρέποντας τόσο γρήγορη διερεύνηση όσο και εκτεταμένες αξιολογήσεις μεγάλης διάρκειας. Τα πλήρη σύνολα δεδομένων φτάνουν σε μέγεθος εκατοντάδων gigabytes, επιτρέποντας την ανάλυση φορτίων εργασίας με έντονη δραστηριότητα εισόδου/εξόδου.
- **Αυτοματοποίηση και Αναπαραγωγιμότητα:** Η σουίτα Koala παρέχει υποδομή που αυτοματοποιεί τη ρύθμιση του περιβάλλοντος, την εγκατάσταση εξαρτήσεων, την εκτέλεση και την επικύρωση αποτελεσμάτων, διευκολύνοντας την αναπαραγωγή πειραμάτων από διαφορετικούς ερευνητές.

Μεθοδολογία Επιλογής Μετροπρογραμμάτων

Η συγκρότηση της σουίτας βασίστηκε σε συστηματική διαδικασία επιλογής, με στόχο τη δημιουργία ενός αντιπροσωπευτικού και πρακτικά χρήσιμου συνόλου προγραμμάτων.

Πηγές Προέλευσης Τα σενάρια προέρχονται από ποικίλες πηγές πραγματικής χρήσης, καλύπτοντας διαφορετικές χρονικές περιόδους και προγραμματιστικά στυλ:

- **Αποθετήρια ανοικτού κώδικα:** Έργα από πλατφόρμες όπως GitHub και GitLab, τα οποία χρησιμοποιούνται σε πραγματικά περιβάλλοντα [59, 60].
- **Ακαδημαϊκή βιβλιογραφία:** Σενάρια που έχουν χρησιμοποιηθεί για την αξιολόγηση συστημάτων επιτάχυνσης κελύφους [1, 4, 10].
- **Εκπαιδευτικά παραδείγματα και κλασική βιβλιογραφία:** Κλασικά παραδείγματα σύνθεσης εντολών από έργα όπως το Unix for Poets και το Unix50, τα οποία αναδεικνύουν τη φιλοσοφία του Unix [61, 62]. Συμπεριλαμβάνονται παραδείγματα από ιστορικά άρθρα, βιβλία και εγχειρίδια [19, 58, 63].
- **Σύγχρονες ροές εργασίας και πραγματικά δεδομένα:** Για παράδειγμα, σενάρια που αλληλεπιδρούν με σύγχρονα μοντέλα θεμελίωσης ή επεξεργάζονται εκτενή πραγματικά σύνολα δεδομένων [64, 65, 66].

Κριτήρια Συμπερίληψης και Αποκλεισμού Για την επιλογή των σεναρίων εφαρμόστηκαν συγκεκριμένα κριτήρια, με πρωταρχικό στόχο τη συγκέντρωση ρεαλιστικών παραδειγμάτων εργασιών. Επιδιώχθηκε η συμπερίληψη προγραμμάτων, είτε απλών είτε πολύπλοκων, τα οποία επεξεργάζονται πραγματικά δεδομένα και χαρακτηρίζονται από ετερογένεια και ποικιλομορφία, τόσο ως προς τα συντακτικά χαρακτηριστικά POSIX όσο και ως προς τα χαρακτηριστικά εκτέλεσής τους. Παράλληλα, αποφεύχθηκε σκόπιμα η ενσωμάτωση σεναρίων χωρίς διαθέσιμα δεδομένα ή λειτουργικές εξαρτήσεις, καθώς και η χρήση υπερβολικά απλοϊκών, τετριμμένων ή μη ρεαλιστικών προγραμμάτων, τα οποία δεν αντιπροσωπεύουν πραγματικές συνθήκες χρήσης.

5.2 Επισκόπηση της Σουίτας

Με βάση τις παραπάνω σχεδιαστικές αρχές, συγκροτήθηκε το πλήρες σύνολο της σουίτας Koala. Σε αντίθεση με μικροδοκιμές επίδοσης, τα προγράμματα Koala εκτελούν ολοκληρωμένες εργασίες, επιτρέποντας τη μελέτη σύνθετων αλληλεπιδράσεων μεταξύ εντολών, ροών δεδομένων και συστήματος αρχείων.

Ο Πίνακας 5.1 παρουσιάζει συνοπτικά το πλήρες σύνολο μετροπρογραμμάτων Koala και τα βασικά χαρακτηριστικά τους. Τα προγράμματα οργανώνονται σε επιμέρους σύνολα, τα οποία μοιράζονται το είδος της εργασίας, την προέλευση, τα δεδομένα εισόδου ή τα υπολογιστικά χαρακτηριστικά. Στις επόμενες ενότητες αναλύονται τα βασικά περιγραφικά στοιχεία της σουίτας.

Υπολογιστικά Πεδία

Η σουίτα Koala καλύπτει ένα ευρύ φάσμα υπολογιστικών πεδίων, αντανακλώντας τόσο παραδοσιακές όσο και σύγχρονες χρήσεις του κελύφους. Τα πεδία αυτά συμβολίζονται με αγγλικά αρχικά στον Πίνακα 5.1, ενώ παρακάτω παρουσιάζονται συνοπτικά με πλήρη ελληνική περιγραφή.

Τα προγράμματα ανάλυσης δεδομένων (**DA – Data Analytics**) περιλαμβάνουν σενάρια που εξάγουν, μετασχηματίζουν και συνοψίζουν μεγάλα σύνολα δεδομένων, όπως μετεωρολογικά αρχεία, δεδομένα δημόσιων μεταφορών και γονιδιωματικά σύνολα. Συνολικά περιλαμβάνονται 11 προγράμματα κατανεμημένα σε τρία σύνολα μετροπρογραμμάτων.

Τα προγράμματα διαχείρισης συστήματος (**SA – System Administration**) αντιστοιχούν σε τυπικές εργασίες ρύθμισης, συντήρησης και ανάλυσης καταγραφών συστήματος. Αποτελούν χαρακτηριστικό παράδειγμα της καθημερινής χρήσης του κελύφους σε πραγματικά περιβάλλοντα παραγωγής και περιλαμβάνουν 7 σενάρια χωρισμένα σε δύο σύνολα.

Οι ροές συνεχούς ολοκλήρωσης και ανάπτυξης (**CI – Continuous Integration**) περιλαμβάνουν σενάρια που αυτοματοποιούν διαδικασίες δόμησης λογισμικού, ανάλυσης και δοκιμών. Στη σουίτα Koala αντιπροσωπεύονται από 23 προγράμματα κατανεμημένα σε δύο σύνολα.

Τα προγράμματα μηχανικής μάθησης (**ML – Machine Learning**) περιλαμβάνουν είτε σενάρια που υλοποιούν στάδια μάθησης είτε σενάρια που λειτουργούν ως «γλώσσα συγκόλλησης», συνδέοντας εξωτερικά εργαλεία και βιβλιοθήκες σε ενιαίες ροές εργασίας. Περιλαμβάνουν 27 προγράμματα κατανεμημένα σε τρία σύνολα.

Τα σενάρια αυτοματοποίησης (**AN – Automation**) περιλαμβάνουν 18 σενάρια, κατανεμημένα σε δύο σύνολα, που αυτοματοποιούν ετερογενείς εργασίες, όπως κρυπτογράφηση αρχείων, μετατροπή πολυμέσων ή ειδικές εργασίες επεξεργασίας περιεχομένου.

Τέλος, η κατηγορία των λοιπών προγραμμάτων (**MI – Miscellaneous**) περιλαμβάνει 40 σενάρια που δεν εντάσσονται σε συγκεκριμένο πεδίο, αλλά αναδεικνύουν την εκφραστικότητα του κελύφους μέσα από ετερογενείς εργασίες, όπως μετασχηματισμούς κειμένου ή υλοποίηση μηχανών αναζήτησης.

Πέρα από την ταξινόμηση κατά πεδίο εφαρμογής, η σουίτα συνδυάζει και διαφορετικά στυλ υπολογισμού, τα οποία εμφανίζονται μετά την κάθετο στη στήλη $\mathscr{D}$ του Πίνακα 5.1. Παραδείγματα αποτελούν τα σύνολα εξαγωγής δεδομένων (DE), τα σενάρια επεξεργασίας κειμένου πολλαπλών σταδίων (TP) και τα σύνολα αυτοματοποίησης εργασιών (AN), τα οποία αντιπροσωπεύουν διαφορετικά πρότυπα χρήσης του κελύφους.

Πίνακας 5.1: Σύνοψη των μετροπρογραμμάτων της σουίτας Koala. Ο πίνακας συνοψίζει όλα τα υποσύνολα μετροπρογραμμάτων της σουίτας Koala. Στήλη 1: χαρακτηριστικά ταυτοποίησης, όπου παρουσιάζεται το όνομα κάθε συνόλου μετροπρογραμμάτων και (ενδεικτικά) προγράμματα που περιλαμβάνει. Στήλες 2–4: περιγραφικά χαρακτηριστικά, που συνοψίζουν το υπολογιστικό πεδίο εφαρμογής ($\mathscr{D}$), τον αριθμό προγραμμάτων του συνόλου (#.sh) και τις συνολικές γραμμές κώδικα (LoC). Στήλη 5: σύνοψη του μεγέθους των δεδομένων εισόδου κάθε προγράμματος. Στήλες 6–7: στατικά χαρακτηριστικά, δηλαδή ο αριθμός συντακτικών δομών του κελύφους (#Cons) και ο αριθμός διακριτών εντολών (#Cmd). Στήλες 8–11: δυναμικά χαρακτηριστικά εκτέλεσης, όπως ο χρόνος εκτέλεσης του κελύφους (t_S), ο χρόνος εκτέλεσης εντολών (t_C), η κατανάλωση μνήμης (Mem) και ο συνολικός όγκος εισόδου/εξόδου (I/O). Στήλες 12–13: χαρακτηριστικά συστήματος, συμπεριλαμβανομένου του αριθμού κλήσεων συστήματος (#SC) και ανοικτών περιγραφέων αρχείων (#FD). Στήλη 14: πηγή προέλευσης προγραμμάτων.

(Ο πίνακας παρουσιάζεται στην επόμενη σελίδα)

Πίνακας 5.1: Μέρος Α'

Benchmark/Script	Surface			Inputs		Syntax		Dynamic				System		Source
	$\mathscr{D}$	#.sh	LoC	Small	Full	#Cons	#Cmd	t_S	t_C	Mem	I/O	#SC	#FD	
analytics	SA/DA	4	53	1.94GB	78.9GB	10	21	10ms	84s	334MB	23.0GB	117.3m	84	[1, 4, 67]
`nginx.sh`			19			10	13	~0	1s	10.3MB	1.79GB			
…														
bio	DA/BI	4	823	24.3GB	114GB	17	66	51s	6720s	25.1GB	352GB	35.3m	79	[68, 69, 70]
`data.sh`			226			13	44	46s	3521s	25.1GB	287GB			
…														
ci-cd	CI/BS	21	2592	N/A		20	134	30ms	128s	461MB	2.35GB	3.5m	885	[10, 59]
…														
covid	DA/DE	5	53	3.34GB	5.08GB	5	6	~0	67s	1011MB	80.6GB	14.3m	150	[66]
`1.sh`			9			5	6	~0	12s	13.2MB	17.5GB			
…														
file-mod	AN/MI	5	41	4.35GB	39.2GB	11	10	99ms	235s	164MB	13.9GB	1.5m	61	[1, 4, 55]
`encrypt.sh`			11			10	6	~0	2s	2.82MB	5.10GB			
`img-conv.sh`			11			11	6	99ms	170s	145MB	3.83GB			
…														
inference	ML/DA	3	61	3.85GB	11.7GB	15	27	40ms	1586s	7.16GB	83.7GB	37.9m	81	[64, 71]
…														
ml	ML/DA	1	47	4.71GB	15.0GB	7	1	~0	1156s	7.87GB	41.6GB	14.9m	10	[72]
nlp	TP/ML	23	303	3k bks	115.9k bks	10	19	15s	851s	9.62MB	272GB	178.6m	526	[61]
`bigrams.sh`			19			10	16	710ms	50s	7.93MB	22.5GB			
…														

Πίνακας 5.1: Μέρος Β'

Benchmark/Script	Surface			Inputs		Syntax		Dynamic				System		Source
	$\mathscr{D}$	#.sh	LoC	Small	Full	#Cons	#Cmd	t_S	t_C	Mem	I/O	#SC	#FD	
oneliners	AN/TP	13	119	202MB	13.5GB	10	24	60ms	14s	199MB	3.77GB	4.5m	288	
`spell.sh`			11			6	7	~0	1s	8.38MB	523MB			[73]
`top-n.sh`			2			5	6	~0	960ms	8.25MB	408MB			[74]
`uniq-ips.sh`			2			4	3	~0	60ms	8.15MB	54.2MB			[75]
…														[19, 63]
pkg	CI/AN	2	43	110 pkgs	2.0k pkgs	16	22	5s	201s	572MB	15.3GB	35.2m	132	[60, 76]
`pacaur.sh`			29			14	19	5s	81s	490MB	14.6GB			
`proginf.sh`			14			11	6	10ms	120s	572MB	709MB			
repl	SA/MI	3	586	N/A		20	55	~0	24s	197MB	1.21GB	5.6m	79	[1, 77]
…														
unixfun	MI/TP	36	70	599MB	59.1GB	4	12	~0	5s	9.80MB	5.71GB	944.0k	733	[62]
`1.sh`			2			3	2	~0	50ms	680KB	108MB			
`2.sh`			2			3	3	~0	260ms	7.51MB	209MB			
…														
weather	DA/DE	2	74	893MB	146GB	11	20	260ms	958s	94.2MB	39.1GB	56.7m	50	[58]
…														
web-search	MI/TP	4	34	833MB	8.61GB	14	30	11s	1343s	1.32GB	17.1GB	112.6m	174	[78]
…														
Min		1	34	1.05MB	44.9MB	4	1	0	5.4	9.62MB	1.21GB	944.0k	10	
Mean		9	349.9	3.66GB	38.0GB	12.1	31.9	6.1	955.7	3.17GB	67.9GB	44.2m	238	
Median		4	65.5	1.54GB	13.5GB	11	21.5	0.1	218.2	398MB	20.1GB	25.0m	108	
Max		36	2592	24.3GB	146GB	20	134	52	6720.9	25.1GB	352GB	178.6m	885	

Σύνολα Μετροπρογραμμάτων

Η σουίτα Koala αποτελείται από δεκατέσσερα σύνολα προγραμμάτων, τα οποία παρουσιάζονται με αλφαβητική σειρά και καλύπτουν διαφορετικά υπολογιστικά πρότυπα και στυλ χρήσης του κελύφους [16]. Κάθε σύνολο περιλαμβάνει προγράμματα με κοινή προέλευση, παρόμοια δεδομένα εισόδου ή αντίστοιχα υπολογιστικά χαρακτηριστικά.

Το `analytics` περιλαμβάνει τέσσερα προγράμματα ανάλυσης αρχείων καταγραφής, τα οποία εκτελούν λειτουργίες φιλτραρίσματος και σύνοψης γεγονότων [1, 4]. Τα προγράμματα επεξεργάζονται συνολικά 78.9GB γραμμικών δεδομένων, όπως ίχνη TCP, αρχεία πρόσβασης Nginx και δεδομένα σάρωσης ZMap, προερχόμενα από πραγματικές δικτυακές καταγραφές [79, 80, 81, 82]. Η μικρή έκδοση των δεδομένων αποτελεί περικομμένο υποσύνολο των ίδιων αρχείων.

Το `bio` αποτελείται από τέσσερα προγράμματα επεξεργασίας γονιδιωματικών και μεταγραφιωματικών δεδομένων. Ένα πρόγραμμα εκτελεί ανάλυση πληθυσμιακής γονιδιωματικής [68, 69], ενώ τα υπόλοιπα τρία υλοποιούν στάδια της πλατφόρμας TERA-Seq για επεξεργασία αλληλουχιών RNA [70]. Τα σενάρια εμφανίζουν διακλάδωση και σύγκλιση ροών επεξεργασίας (με δομές *fan-out/fan-in*), παραλληλισμό τύπου ουράς εργασιών (work-queue) και έντονη πρόσβαση σε μεγάλα αρχεία, με συνολικά δεδομένα εισόδου 114GB [83].

Το `ci-cd` περιλαμβάνει 21 προγράμματα δόμησης λογισμικού, όπως σενάρια δόμησης για τα λογισμικά Lua, Memcached, Redis και SQLite [10], καθώς και τη σουίτα δοκιμών του εργαλείου `makeself` [59]. Τα προγράμματα αυτά χρησιμοποιούνται σε ροές συνεχούς ολοκλήρωσης και χαρακτηρίζονται από έντονες εξαρτήσεις και πολύπλοκη ροή ελέγχου, αλλά περιορισμένο όγκο δεδομένων εισόδου.

Το `covid` περιέχει πέντε προγράμματα που υπολογίζουν στατιστικά στοιχεία δημόσιων συγκοινωνιών κατά την περίοδο της πανδημίας [66]. Τα σενάρια διατίθενται σε δύο εκδοχές: μία που βασίζεται σε σύνθεση κλασικών εργαλείων Unix όπως `cut`, `sort` και `uniq`, και μία υλοποιημένη ως μονολιθικό πρόγραμμα `awk`. Τα δεδομένα εισόδου αποτελούνται από 5.08GB αρχείων CSV.

Το `file-mod` περιλαμβάνει πέντε προγράμματα μετασχηματισμού αρχείων, όπως συμπίεση, κρυπτογράφηση και μετατροπή μορφότυπων. Δύο από αυτά επεξεργάζονται αρχεία καταγραφής δικτύου χρησιμοποιώντας `openssl` [55, 84], ενώ τα υπόλοιπα εκτελούν μετατροπές πολυμέσων σε εκατοντάδες αρχεία εικόνας και ήχου [1, 4], με συνολικό μέγεθος δεδομένων 39.2GB.

Το `inference` αποτελείται από τρία προγράμματα που εκτελούν εργασίες συμπερασμού χρησιμοποιώντας μεγάλα μοντέλα θεμελίωσης [65, 85, 86]. Παραδείγματα περιλαμβάνουν περιγραφή εικόνων [71], παραγωγή playlists [64] και ταξινόμηση ιερογλυφικών εικόνων [65]. Τα προγράμματα λειτουργούν ως περιβλήματα (wrappers) για εξωτερικές εργασίες μηχανικής μάθησης [87, 88, 89], επεξεργαζόμενα συνολικά 11.7GB δεδομένων.

Το `ml` αντιστοιχεί σε τυπικό παράδειγμα εργασιών μηχανικής μάθησης, όπου ένα μονολιθικό πρόγραμμα Python έχει αποσυντεθεί σε πολλαπλά στάδια κελύφους (εισαγωγή δεδομένων, εκπαίδευση, ταξινόμηση και αξιολόγηση), επιτρέποντας τη μελέτη αρθρωτών ροών εργασίας [72], με τα δεδομένα εισόδου να ανέρχονται σε 15.0GB.

Το `nlp` περιλαμβάνει 23 σενάρια επεξεργασίας φυσικής γλώσσας από το tutorial Unix for Poets [61]. Τα περισσότερα από αυτά αποτελούνται από μία ή δύο γραμμές και μπορούν να συνδυαστούν σε μεγαλύτερους αγωγούς, λειτουργώντας πάνω σε συλλογή άνω των 115.000 βιβλίων [90].

Το `oneliners` περιλαμβάνει 11 αλυσίδες αγωγών που προέρχονται τόσο από την κλασική βιβλιογραφία του Unix όσο και από σύγχρονες πηγές [19, 61, 63, 67, 73, 74, 75]. Τα σενάρια χρησιμοποιούν σύνθετους τελεστές ροής όπως απεικόνιση (map), φιλτράρισμα (filter) και αναγωγή (reduction) και αναδεικνύουν τη φιλοσοφία σύνθεσης του κελύφους, εφαρμοσμένα σε δεδομένα συνολικού μεγέθους 13.5GB.

Το `pkg` αποτελείται από δύο προγράμματα που αυτοματοποιούν την εγκατάσταση πακέτων και την ανάλυση δικαιωμάτων λογισμικού [76, 91, 92]. Οι εκτελέσεις περιλαμβάνουν λήψη, δόμηση και ανάλυση εκατοντάδων πακέτων για το πρώτο πρόγραμμα και χιλιάδων για το δεύτερο.

Το `repl` περιλαμβάνει δύο ανεξάρτητα προγράμματα, ένα για έλεγχο ασφαλείας και ένα για αναπαραγωγή ανάπτυξης λογισμικού σε αποθετήριο Git. Και τα δύο δίνουν έμφαση σε σύνθετη αλληλεπίδραση με το σύστημα αρχείων.

```
sample-benchmark
├── scripts       # Benchmark scripts
│   ├── a.sh
│   ├── b.sh
│   └── ...sh
├── install.sh    # Dependency installation
├── fetch.sh      # Input data download/preparation
├── execute.sh    # Script execution
├── validate.sh   # Hash generation & verification
└── clean.sh      # Input and output file cleanup
```

Σχήμα 5.1: Αρχιτεκτονική Διεπαφής Μετροπρογραμμάτων. Η ροή εργασιών αποτελείται από πέντε σπονδυλωτά σενάρια ελέγχου. Ο κεντρικός οδηγός `main.sh` ενορχηστρώνει την εκτέλεση, διασφαλίζοντας την αποδοτική διαχείριση των πόρων και την ακεραιότητα των δεδομένων.

Το `unixfun` περιέχει 36 προγράμματα από το Unix50 challenge [62], τα οποία μιμούνται κλασικούς υπολογισμούς επεξεργασίας κειμένου σε συνολικό όγκο δεδομένων 59.1GB.

Το `weather` αποτελείται από δύο προγράμματα που υπολογίζουν στατιστικά μεγέθη πάνω σε ιστορικά δεδομένα θερμοκρασίας, βασισμένα σε ένα παράδειγμα από το [58]. Το πρώτο πρόγραμμα είναι παρόμοιο με το παράδειγμα του προηγούμενου κεφαλαίου (Κώδικας 4.1) ενώ το δεύτερο εκτελεί επιπλέον ανάλυση και αναπαράγει το γνωστό διάγραμμα καιρού του Edward Tufte [93]. Ορισμένες από τις φάσεις επεξεργασίας τους αντιστοιχούν εννοιολογικά σε υπολογισμούς τύπου MapReduce και Spark, αναδεικνύοντας σενάρια επεξεργασίας δεδομένων μεγάλης κλίμακας. Τα δεδομένα προέρχονται από το NOAA [57] και καλύπτουν πολλαπλά έτη ιστορικών μετρήσεων, συνολικού μεγέθους 146GB.

Τέλος, το `web-search` υλοποιεί μια πλήρη ροή αναζήτησης ιστού (`crawl`, `index`, `query`) χρησιμοποιώντας αποκλειστικά εργαλεία κελύφους και προηγμένους τελεστές ροής δεδομένων πάνω σε αρχεία μεγέθους 8.61GB.

5.3 Υποδομή και Διαμόρφωση Εκτέλεσης

Η αρχιτεκτονική του Koala βασίζεται σε μια τυποποιημένη διεπαφή (interface) που διέπει τον κύκλο ζωής κάθε προγράμματος. Η προσέγγιση αυτή διασφαλίζει τον ντετερμινισμό των μετρήσεων και την ομοιομορφία μεταξύ ετερογενών φόρτων εργασίας, ορίζοντας ρητά τις φάσεις προετοιμασίας, εκτέλεσης και επαλήθευσης.

Η προδιαγραφή αυτή υλοποιείται μέσω πέντε σεναρίων υποδομής (Σχήμα 5.1). Συγκεκριμένα:

1. Το `install.sh` εγκαθιστά τις απαραίτητες εξαρτήσεις λογισμικού.
2. Το `fetch.sh` αναλαμβάνει τη λήψη ή παραγωγή των δεδομένων εισόδου και δέχεται όρισμα που καθορίζει το μέγεθός τους (`--min`, `--small`, `--full`).
3. Το `execute.sh` εκτελεί τα προγράμματα του συνόλου και συλλέγει βασικές μετρικές, όπως χρόνο εκτέλεσης και κατανάλωση πόρων.
4. Το `validate.sh` επαληθεύει την ορθότητα της εξόδου συγκρίνοντας τα παραγόμενα αποτελέσματα με προκαθορισμένες τιμές αναφοράς μέσω υπολογισμού κατακερματισμών.
5. Το `clean.sh` απομακρύνει προσωρινά αρχεία και παραγόμενα αποτελέσματα.

Ενδεικτικά, στο παράρτημα παρατίθενται τα σενάρια υποδομής για το `weather` (Κώδικες A.1 έως A.5).

Πίνακας 5.2: Εκτιμώμενος χρόνος ενσωμάτωσης ανά σύνολο μετροπρογραμμάτων. Ο πίνακας παρουσιάζει τον εκτιμώμενο χρόνο ενσωμάτωσης (σε ώρες) για κάθε υποσύνολο μετροπρογραμμάτων της σουίτας Koala. Οι εκτιμήσεις βασίζονται στην εμπειρία των ερευνητών κατά τη διαδικασία ενσωμάτωσης και περιλαμβάνουν την προσαρμογή των μετροπρογραμμάτων και την προετοιμασία των σεναρίων υποδομής.

Υποσύνολο Μετροπρογραμμάτων	$\sim T$ (ώρες)
`analytics`	40
`bio`	60
`ci-cd`	75
`covid`	65
`file-mod`	60
`ml`	30
`inference`	35
`nlp`	10
`oneliners`	10
`pkg`	30
`repl`	25
`unixfun`	10
`weather`	30
`web-search`	40
Σύνολο	520

Το Koala περιλαμβάνει επίσης έναν κεντρικό οδηγό εκτέλεσης, το `main.sh`, ο οποίος καλεί διαδοχικά τα παραπάνω σενάρια για ένα ή περισσότερα μετροπρογράμματα. Ο οδηγός δέχεται παραμέτρους είτε μέσω σημαιών είτε μέσω μεταβλητών περιβάλλοντος και τις προωθεί στα επιμέρους σενάρια. Ενδεικτικά, η μεταβλητή `KOALA_SHELL` καθορίζει τον διερμηνέα κελύφους που θα χρησιμοποιηθεί (π. χ. `bash`, `zsh` ή συγκεκριμένη διαδρομή εκτελέσιμου), με προεπιλογή το `sh`.

Επιπλέον, παρέχονται σημαίες για εξειδικευμένη συλλογή δεδομένων και έλεγχο της εκτέλεσης. Η σημαία `--resources` ενεργοποιεί την καταγραφή μετρικών δυναμικής ανάλυσης—όπως ο χρόνος εκτέλεσης, η κατανάλωση μνήμης και η ένταση δραστηριότητας εισόδου/εξόδου (I/O)—οι οποίες παρουσιάζονται αναλυτικά σε επόμενο κεφάλαιο. Για απλή χρονομέτρηση διατίθεται η σημαία `--time` (ή `-t`). Τέλος, μέσω της επιλογής `--scripts` (ή `-s`), η οποία ακολουθείται από τα ονόματα των επιθυμητών σεναρίων, ο χρήστης δύναται να εκτελέσει επιλεκτικά συγκεκριμένα σενάρια από ένα σύνολο μετροπρογραμμάτων. Ένας χρήστης μπορεί να εκτελέσει το πρόγραμμα `temp-analytics.sh` (που μοιάζει με τον Κώδικα 4.1) από το `weather`, με το `main.sh` και τη μικρή κλίμακα δεδομένων ως εξής:

```
./main.sh --small weather --scripts temp-analytics
```

Η σχεδίαση της υποδομής στοχεύει στη γρήγορη συλλογή προκαταρκτικών αποτελεσμάτων και στη διευκόλυνση συγκρίσεων μεταξύ συστημάτων. Παραμένει εσκεμμένα ελαφριά και επεκτάσιμη, επιτρέποντας στους χρήστες να την προσαρμόσουν στις ιδιαίτερες ανάγκες τους.

Εκτέλεση σε Απομονωμένο Περιβάλλον

Η σουίτα Koala παρέχει προαιρετική υποστήριξη εκτέλεσης σε απομονωμένο περιβάλλον μέσω Docker. Διατίθεται αρχείο `Dockerfile` το οποίο δημιουργεί περιβάλλον βασισμένο σε Debian και εγκαθιστά τις βασικές εξαρτήσεις της σουίτας. Υποστηρίζεται επίσης η δυναμική δημιουργία εικόνων ειδικά προσαρμοσμένων σε κάθε σύνολο μετροπρογραμμάτων, με ενσωματωμένη κλήση του `main.sh`. Η χρήση container επιτρέπει αναπαραγώγιμα πειράματα χωρίς τροποποίηση του συστήματος του χρήστη. Η σουίτα αποφεύγει τη χρήση προνομιακών εντολών, περιορίζοντας τις απαιτήσεις σε δικαιώματα διαχειριστή μόνο όπου είναι απολύτως αναγκαίο.

Ενσωμάτωση Νέου Μετροπρογράμματος

Η προσθήκη νέου μετροπρογράμματος στο Koala απαιτεί την προσαρμογή των πέντε βασικών σεναρίων υποδομής. Συγκεκριμένα, περιλαμβάνει: (1) τον ορισμό εξαρτήσεων στο `install.sh`, (2) την προετοιμασία δεδομένων πολλαπλών μεγεθών στο `fetch.sh`, (3) την αυτοματοποιημένη εκτέλεση στο `execute.sh`, (4) τον ορισμό λογικής επικύρωσης στο `validate.sh`, και (5) τον καθαρισμό προσωρινών αρχείων στο `clean.sh`.

Ο απαιτούμενος χρόνος ενσωμάτωσης κυμάνθηκε από 10 έως 80 εργατοώρες ανά υποσύνολο. Μικρά και αυτοτελή σύνολα όπως τα `oneliners`, `unixfun` και `nlp` ενσωματώθηκαν σε περίπου 10–12 ώρες. Αντίθετα, σύνθετα ή έντονα εξαρτώμενα από δεδομένα σύνολα όπως τα `file-mod`, `bio` και `ci-cd` απαίτησαν σημαντικά περισσότερη προσπάθεια λόγω μεγέθους, εξαρτήσεων και αυστηρών απαιτήσεων ορθότητας. Σημαντικό μέρος του χρόνου αφιερώθηκε στην προσαρμογή δεδομένων και στη διασφάλιση της ορθότητας υπό διαφορετικά περιβάλλοντα εκτέλεσης, γεγονός που αναδεικνύει τη σημασία μιας επαναχρησιμοποιήσιμης και τυποποιημένης υποδομής.

5.4 Συνοπτικός Χαρακτηρισμός της Σουίτας

Για να τεκμηριωθεί η καταλληλότητα και η αντιπροσωπευτικότητα του Koala ως εργαλείο αξιολόγησης επιδόσεων, πραγματοποιήθηκε εκτενής χαρακτηρισμός των μετροπρογραμμάτων του. Ο χαρακτηρισμός αυτός αναδεικνύει την εγγενή ποικιλομορφία της σουίτας, προσεγγίζοντας τα σενάρια τόσο από στατική (συντακτική) όσο και από δυναμική (συμπεριφορά κατά την εκτέλεση) σκοπιά. Στοιχεία της ανάλυσης παρουσιάζονται στον Πίνακα 5.1, ενώ λεπτομερή διαγράμματα και η πλήρης στατιστική ανάλυση των παρακάτω μετρικών παραλείπονται χάριν συντομίας και είναι διαθέσιμα στην κεντρική δημοσίευση της σουίτας [16].

Στατικός και Συντακτικός Χαρακτηρισμός Η εξαγωγή των συντακτικών χαρακτηριστικών πραγματοποιήθηκε μέσω της βιβλιοθήκης `libdash` [94], η οποία αξιοποιεί τον ορισμό αφηρημένης σύνταξης του Smoosh [8]. Η ανάλυση καταδεικνύει την έντονη συντακτική ποικιλομορφία μεταξύ των υπολογιστικών πεδίων: ενώ εργασίες όπως η επεξεργασία φυσικής γλώσσας (`nlp`) και η ανάλυση δεδομένων (`covid`) βασίζονται κυρίως σε εκτενείς, γραμμικές σωληνώσεις, πεδία όπως η διαχείριση υποδομών (`ci-cd`, `pkg`) κάνουν βαριά χρήση σύνθετων δομών ελέγχου (`if`, `while`, `case`), υποκελυφών και συναρτήσεων. Επιπλέον, επιμέρους σενάρια (π.χ. `bio`, `web-search`) ενσωματώνουν ρητές κατασκευές χειροκίνητου παραλληλισμού (`&`, `wait`). Όσον αφορά την κατανομή των εντολών, μολονότι ως προς τη συχνότητα εμφάνισης δεσπόζουν τα βασικά εργαλεία του Unix (π.χ. `echo`, `cat`, `grep`), το 54,4% του συνόλου των μοναδικών εντολών της σουίτας αντιστοιχεί σε προσαρμοσμένα εκτελέσιμα (custom binaries) ή τοπικές συναρτήσεις. Το χαρακτηριστικό αυτό επιβεβαιώνει τον ρόλο του κελύφους ως «συγκολλητικού ιστού» και αναδεικνύει το βασικό εμπόδιο για την ανάπτυξη συστημάτων στατικής επιτάχυνσης, καθώς αυτά καλούνται να διαχειριστούν τον κύριο όγκο του υπολογισμού ως αδιαφανή «μαύρα κουτιά».

Δυναμικό Προφίλ Εκτέλεσης Επειδή η στατική ανάλυση δεν αποτυπώνει τις πραγματικές απαιτήσεις σε πόρους, πραγματοποιήθηκε εκτενής δυναμικός χαρακτηρισμός μέσω παρακολούθησης σε επίπεδο πυρήνα (αξιοποιώντας το εικονικό σύστημα αρχείων `/proc`, το εργαλείο `strace` και συνεχή δειγματοληψία μέσω `psutil`). Η ανάλυση ανέδειξε τεράστια ποικιλομορφία, επιβεβαιώνοντας ότι η σουίτα δοκιμάζει τα όρια όλων των υπολογιστικών υποσυστημάτων. Ειδικότερα, ο πραγματικός χρόνος εκτέλεσης κυμαίνεται από ελάχιστα δευτερόλεπτα (π.χ. `unixfun`) έως αρκετές ώρες στα πλήρη σύνολα δεδομένων (π.χ. `bio`, `web-search`). Ένα κρίσιμο εύρημα αφορά την κατανομή του χρόνου εκτέλεσης: ενώ σε ορισμένα σενάρια το κέλυφος καταναλώνει σημαντικό υπολογιστικό χρόνο, στα βαρύτερα φορτία εργασίας η εκτέλεση κυριαρχείται σχεδόν πλήρως από τα εξωτερικά εκτελέσιμα, καθιστώντας τον παραλληλισμό τη μόνη βιώσιμη στρατηγική επιτάχυνσης.

Η ετερογένεια της σουίτας αντικατοπτρίζεται εξίσου στη χρήση μνήμης και τη δραστηριότητα εισόδου/εξόδου (I/O). Το αποτύπωμα μνήμης εκτείνεται σε πέντε τάξεις μεγέθους: από εξαιρετικά ελαφριές διαδικασίες ροής (streaming) που απαιτούν λιγότερο από 1 MB, έως εργασίες φόρτωσης μοντέλων και διατήρησης γονιδιωματικών δομών που δεσμεύουν δεκάδες Gigabytes. Αντίστοιχα, το προφίλ I/O ποικίλλει ριζικά, περιλαμβάνοντας τόσο αμιγώς υπολογιστικά (CPU-bound) προγράμματα με ελάχιστη χρήση δίσκου, όσο και εργασίες μαζικής σάρωσης αρχείων (I/O-bound) με ρυθμούς διαμεταγωγής που προσεγγίζουν το 1 GB/s. Τέλος, ο βαθμός αλληλεπίδρασης με τον πυρήνα του λειτουργικού συστήματος κυμαίνεται από 10 έως σχεδόν 900 ταυτόχρονα ανοικτούς περιγραφείς αρχείων (file descriptors) και φτάνει έως και εκατοντάδες εκατομμύρια κλήσεις συστήματος (system calls), αναδεικνύοντας το υψηλό κόστος της διαδιεργασιακής επικοινωνίας (IPC) μέσω ανώνυμων σωληνώσεων.

Ανάλυση Κύριων Συνιστωσών (PCA) Για να αποτυπωθεί συνολικά η διαφορετικότητα και η ποικιλομορφία της σουίτας, εφαρμόστηκε ανάλυση κύριων συνιστωσών (PCA) [95] σε δύο ανεξάρτητες αναπαραστάσεις των μετροπρογραμμάτων. Η μέθοδος αυτή μειώνει τη διάσταση των δεδομένων, διατηρώντας και αναδεικνύοντας τις βασικές δομικές διαφορές μεταξύ των προγραμμάτων. Στην πρώτη προσέγγιση, η ανάλυση βασίστηκε σε έναν συνδυασμό στατικών και δυναμικών χαρακτηριστικών εκτέλεσης, αποδεικνύοντας ότι τα σενάρια διασκορπίζονται σε έναν ιδιαίτερα ευρύ υπολογιστικό χώρο με ποικίλα συντακτικά και συμπεριφορικά προφίλ. Στη δεύτερη προσέγγιση, αξιοποιήθηκαν διανυσματικές αναπαραστάσεις (embeddings) του πηγαίου κώδικα, παραγόμενες από το προεκπαιδευμένο μοντέλο `text-embedding-3-large` της OpenAI [96], με στόχο την αποτύπωση υψηλού επιπέδου πληροφορίας σχετικά με τη δομή και τη σημασιολογία των προγραμμάτων [97, 98]. Τα αποτελέσματα και των δύο πολυδιάστατων αναλύσεων συγκλίνουν στο ίδιο ισχυρό συμπέρασμα: η σουίτα Koala καλύπτει ένα εξαιρετικά ευρύ φάσμα συντακτικών, σημασιολογικών και δυναμικών μοτίβων, παρέχοντας ένα πλήρως αντιπροσωπευτικό σύνολο ρεαλιστικών φορτίων εργασίας.

5.5 Σύνοψη

Με βάση τον σχεδιασμό, την υλοποίηση και τον εκτενή χαρακτηρισμό που προηγήθηκε, η σουίτα Koala συνιστά ένα ρεαλιστικό, ποικιλόμορφο και πλήρως αυτοματοποιημένο εργαλείο για την αξιολόγηση προγραμμάτων κελύφους. Η ικανότητά της να καλύπτει ευρύ φάσμα συντακτικών μοτίβων και δυναμικών συμπεριφορών την καθιστά ένα ελεγχόμενο και αξιόπιστο πεδίο δοκιμών για τη μελέτη υφιστάμενων αλλά και μελλοντικών συστημάτων επιτάχυνσης.

Έχοντας θέσει αυτό το θεμέλιο, το υπόλοιπο της εργασίας επικεντρώνεται στην πρακτική αξιοποίηση της σουίτας. Στα επόμενα πέντε κεφάλαια, το Koala χρησιμοποιείται για τη συστηματική αξιολόγηση πέντε διαφορετικών προσεγγίσεων βελτιστοποίησης, οι οποίες βασίζονται σε διαφορετικές στρατηγικές:

1. Τον εναλλακτικό διερμηνέα `zsh` (Κεφάλαιο 6).
2. Τον μεταγλωττιστή στατικής βελτιστοποίησης I/O Shark (Κεφάλαιο 7).
3. Το εργαλείο χειροκίνητου παραλληλισμού GNU `parallel` (Κεφάλαιο 8).
4. Το σύστημα δυναμικής παραλληλοποίησης σωληνώσεων PaSh (Κεφάλαιο 9).
5. Την αρχιτεκτονική καιροσκοπικής εκτέλεσης εκτός σειράς *hS* (Κεφάλαιο 10).

Η αξιολόγηση κάθε συστήματος δεν περιορίζεται στη μέτρηση της τελικής απόδοσης, αλλά δίνεται αντίστοιχη έμφαση στην κατανόηση του τρόπου λειτουργίας του, καθώς και στην *προσπάθεια υιοθέτησης (adoption effort)*—δηλαδή στο επίπεδο χειροκίνητης παρέμβασης, συντακτικών τροποποιήσεων ή προσθήκης επισημειώσεων (annotations) που απαιτείται ώστε να εκτελεστούν ρεαλιστικά σενάρια χρήσης.

Η ανάλυση ξεκινά στο επόμενο κεφάλαιο με την εκτέλεση της σουίτας μέσω του `zsh`, με στόχο να διερευνηθεί εμπειρικά κατά πόσο η απλή αντικατάσταση του διερμηνέα—χωρίς καμία τροποποίηση στον κώδικα—μπορεί να επηρεάσει την απόδοση αντιπροσωπευτικών φορτίων εργασίας.

Κεφάλαιο 6

Αξιολόγηση Εναλλακτικού Διερμηνέα `zsh`

Το παρόν κεφάλαιο ανοίγει τον κύκλο των πειραματικών αξιολογήσεων της σουίτας Koala, εξετάζοντας τη συμπεριφορά των ρεαλιστικών φορτίων εργασίας όταν εκτελούνται από έναν εναλλακτικό διερμηνέα κελύφους. Συγκεκριμένα, αξιολογείται το `zsh` [17] (Z Shell), ένας εξαιρετικά δημοφιλής και παραμετροποιήσιμος διερμηνέας, ο οποίος αποτελεί πλέον την προεπιλεγμένη επιλογή διαδραστικού κελύφους σε περιβάλλοντα όπως το macOS.

Βασικός στόχος αυτής της αξιολόγησης είναι να διαπιστωθεί εμπειρικά εάν η απλή αντικατάσταση της υποκείμενης μηχανής εκτέλεσης (engine)—χωρίς την εφαρμογή πολύπλοκων στατικών μετασχηματισμών ή τεχνικών παραλληλοποίησης—επηρεάζει την ορθότητα και την ταχύτητα εκτέλεσης σεναρίων γραμμένων σε πρότυπο POSIX, συγκριτικά με το καθιερωμένο `bash`.

6.1 Μηχανισμός Λειτουργίας και Χαρακτηριστικά

Σε αντίθεση με συστήματα που αναδιαρθρώνουν ενεργά τον κώδικα (όπως το Shark ή το GNU `parallel`, τα οποία θα αναλυθούν στα επόμενα κεφάλαια), το `zsh` εκτελεί το πρόγραμμα *ως έχει*. Ωστόσο, ο τρόπος με τον οποίο το zsh αναλύει και διαχειρίζεται τη μνήμη, καθώς και η υλοποίηση των εσωτερικών του δομών, διαφέρουν από την αντίστοιχη αρχιτεκτονική του `bash`. Αυτές οι διαφορές σε επίπεδο διερμηνέα μπορούν να επηρεάσουν την ταχύτητα εκτέλεσης των ενσωματωμένων εντολών (shell built-ins), την αποδοτικότητα της ανάθεσης μεταβλητών, καθώς και την ταχύτητα επεξεργασίας δομών ελέγχου (όπως οι βρόχοι `for` και `while`). Προκειμένου να διασφαλιστεί η συμβατότητα και η ορθή εκτέλεση προγραμμάτων που ακολουθούν την αυστηρή φιλοσοφία του Unix, το `zsh` διαθέτει μια ενσωματωμένη λειτουργία εξομοίωσης (`emulate sh`), η οποία προσαρμόζει τη σημασιολογία του στις προδιαγραφές του προτύπου POSIX.

6.2 Το Κινητήριο Παράδειγμα: Το Σενάριο `weather`

Η συμπεριφορά του σεναρίου `weather` (Κώδικας 4.1) αποτελεί μια χρήσιμη περίπτωση μελέτης. Το μετροπρόγραμμα βασίζεται σε έναν κεντρικό επαναληπτικό βρόχο (`for`) και στην επέκταση μεταβλητών, προτού καλέσει στο εσωτερικό του τρεις σωληνώσεις εντολών (π.χ. `awk`, `sort`).

Δεδομένου ότι το `zsh` εκτελεί αυτόν τον κώδικα χωρίς μετασχηματισμούς, η ταχύτητα ολοκλήρωσης του σεναρίου εξαρτάται άμεσα από τη βελτιστοποίηση των εσωτερικών λειτουργιών (built-ins) του διερμηνέα. Η πλήρης ταύτιση της απόδοσης στο `weather` επιβεβαιώνει εμπειρικά ότι η υλοποίηση των εσωτερικών δομών ελέγχου και της διαχείρισης μεταβλητών του `zsh` είναι συγκρίσιμη με εκείνη του `bash`, και δεν εισάγει συμφόρηση (bottleneck) στην επεξεργασία του βρόχου.

6.3 Προσπάθεια Υιοθέτησης

Όπως αναφέρθηκε και παραπάνω, σε αντίθεση με τις μεθόδους χειροκίνητου παραλληλισμού, η προσπάθεια υιοθέτησης του `zsh` μέσω της σουίτας Koala δεν απαιτεί *καμία* τροποποίηση κώδικα. Η υποβολή των 126 σεναρίων της σουίτας στον νέο διερμηνέα επιτεύχθηκε με την απλή αλλαγή της

μεταβλητής περιβάλλοντος `KOALA_SHELL`, η οποία δρομολογεί την εκτέλεση σε ένα προσαρμοσμένο, αλλά εξαιρετικά απλό, σενάριο εκκίνησης (Κώδικας 6.1).

```
#!/bin/zsh

emulate sh
source "$@"
```

Κώδικας 6.1: Σενάριο εκκίνησης του zsh στο Koala. Το σενάριο ενεργοποιεί τη λειτουργία εξομοίωσης POSIX (`emulate sh`) πριν φορτώσει και εκτελέσει τον κώδικα του μετροπρογράμματος, διασφαλίζοντας συμβατότητα.

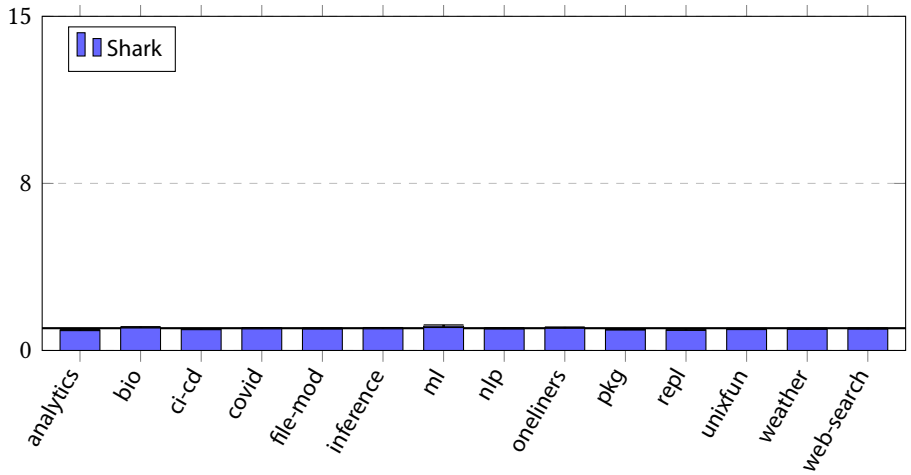
Η αξιοποίηση της εντολής `source` είναι κρίσιμη, καθώς επιτρέπει στο `zsh` να εκτελέσει τον κώδικα του μετροπρογράμματος απευθείας εντός του τρέχοντος περιβάλλοντος, διατηρώντας έτσι ανέπαφη τη ροή επικύρωσης και συλλογής μετρικών του Koala.

6.4 Ανάλυση Απόδοσης

Μεθοδολογία Για την εξαγωγή των πειραματικών αποτελεσμάτων, τα μετροπρογράμματα εκτελέστηκαν χρησιμοποιώντας την έκδοση `zsh` v5.9 υπό τη λειτουργία εξομοίωσης `sh`. Ως αυστηρό σημείο αναφοράς (baseline) χρησιμοποιήθηκε το `bash` v5.2.21, επίσης ρυθμισμένο σε λειτουργία συμβατότητας (`--posix`). Για όλα τα πειράματα χρησιμοποιήθηκε ελεγχόμενο περιβάλλον εκτέλεσης (στιγμιότυπο AWS `c6i.4xlarge`), εξοπλισμένο με 32GB μνήμης και επεξεργαστή 16 πυρήνων στα 3.5 GHz, υπό το λειτουργικό σύστημα Ubuntu 24.04.1. Οι μετρήσεις πραγματοποιήθηκαν στα μικρά σύνολα δεδομένων (`--small`).

Ορθότητα και Συνολική Επιτάχυνση Κεντρικό εύρημα της πειραματικής διαδικασίας είναι ότι, υπό την εκτέλεση του `zsh`, και τα 126 μετροπρογράμματα της σουίτας ολοκληρώθηκαν με επιτυχία *χωρίς ιδιαίτερη επιβράδυνση*.

Στο Σχήμα 6.1 απεικονίζονται οι σχετικές επιταχύνσεις (speedups) του `zsh` συγκριτικά με το `bash`.



Σχήμα 6.1: Σχετικές επιταχύνσεις του zsh στα μετροπρογράμματα της σουίτας Koala. Τα αποτελέσματα επιβεβαιώνουν ότι η απόδοση του `zsh` ταυτίζεται σχεδόν απόλυτα με την απόδοση του `bash` για τη συντριπτική πλειονότητα των σεναρίων.

Όπως αποτυπώνεται στο διάγραμμα, η απόδοση παραμένει πρακτικά αμετάβλητη για όλα τα μετροπρογράμματα. Συνολικά, ο διερμηνέας zsh κατέγραψε μια οριακή και στατιστικά ασήμαντη μέση επιβράδυνση της τάξης του **0.16%** σε σχέση με το bash. Το αποτέλεσμα αυτό καταδεικνύει ότι, παρά τις εσωτερικές τους αρχιτεκτονικές διαφορές, αμφότεροι οι διερμηνείς διαχειρίζονται τις κλήσεις συστήματος, τις ροές δεδομένων και τη δημιουργία θυγατρικών διεργασιών με πανομοιότυπη αποδοτικότητα.

6.5 Σύνοψη

Η αξιολόγηση του zsh μέσω του Koala εξάγει ένα σαφές συμπέρασμα: η μετάβαση από το bash στο zsh (υπό καθεστώς εξομοίωσης sh) επιτυγχάνεται με απολύτως μηδενική προσπάθεια στον πηγαίο κώδικα και δεν οδηγεί σε μετρήσιμη επιβράδυνση κατά την εκτέλεση ρεαλιστικών προγραμμάτων κελύφους. Κατά συνέπεια, οι χρήστες που προτιμούν το zsh για τα προηγμένα διαδραστικά του χαρακτηριστικά μπορούν να εκτελούν μεγάλης κλίμακας υπολογιστικά σενάρια διατηρώντας ακέραιη την απόδοση του συστήματος.

Παρ' όλα αυτά, η απλή αντικατάσταση της μηχανής εκτέλεσης δεν μεταβάλλει τα βασικά σημεία συμφόρησης (bottlenecks) των προγραμμάτων κελύφους, όπως τη σειριακή εκτέλεση και την εκτεταμένη, συχνά περιττή, χρήση του συστήματος αρχείων. Για την επίτευξη ουσιαστικών επιταχύνσεων, απαιτείται συνήθως αναδιάρθρωση του ίδιου του κώδικα. Με αφετηρία αυτή τη διαπίστωση, το επόμενο κεφάλαιο περνά από την παθητική εκτέλεση στην ενεργή βελτιστοποίηση, εξετάζοντας το σύστημα Shark. Εκεί αναλύεται πώς η στατική ανάλυση και ο συντακτικός μετασχηματισμός *πριν* από την εκτέλεση μπορούν να εξαλείψουν το περιττό I/O και να αποκαλύψουν κρυμμένο παραλληλισμό, αυξάνοντας παράλληλα την προσπάθεια υιοθέτησης.

Κεφάλαιο 7

Αξιολόγηση Shark: Στατική Βελτιστοποίηση Εισόδου/Εξόδου

Σε αυτό το κεφάλαιο αναλύεται και αξιολογείται η στρατηγική της στατικής βελτιστοποίησης, χρησιμοποιώντας ως σημείο αναφοράς τη φιλοσοφία του συστήματος Shark [15]. Σε αντίθεση με τον εναλλακτικό διερμηνέα `zsh` που εξετάστηκε στο προηγούμενο κεφάλαιο, η προσέγγιση του Shark απαιτεί τον εκτεταμένο συντακτικό μετασχηματισμό του πηγαίου κώδικα *πριν* από την εκτέλεσή του. Εστιάζει κυρίως στη μείωση του κόστους επικοινωνίας μεταξύ των διεργασιών και στην εξάλειψη περιττών λειτουργιών στο σύστημα αρχείων (I/O).

7.1 Μηχανισμός Λειτουργίας και Χαρακτηριστικά

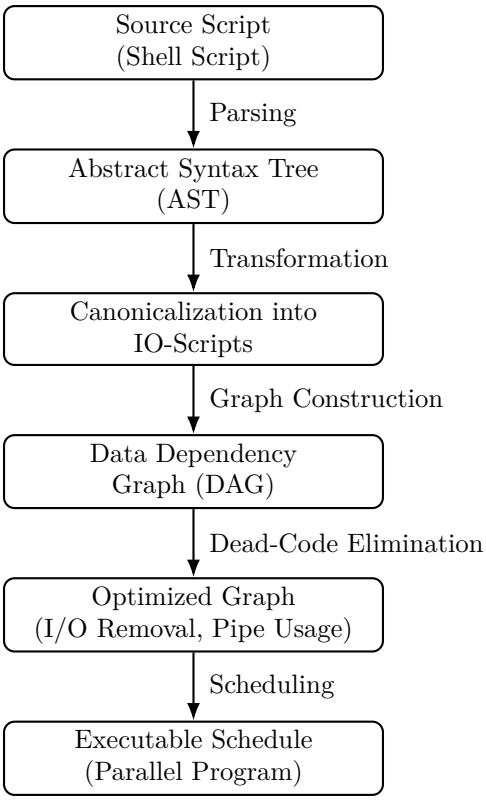
Το Shark αποτελεί ένα από τα πρώτα συστήματα που αντιμετωπίζουν τα προγράμματα κελύφους ως κανονικά αντικείμενα μεταγλώττισης, εφαρμόζοντας τεχνικές ανάλυσης και βελτιστοποίησης αντίστοιχες με εκείνες των παραδοσιακών γλωσσών προγραμματισμού. Η βασική του παραδοχή είναι ότι οι προσπελάσεις στο σύστημα αρχείων (εγγραφές και αναγνώσεις) μπορούν να θεωρηθούν ανάλογες με τις αναφορές σε μεταβλητές σε άλλες γλώσσες. Αυτή η παραδοχή επιτρέπει την εφαρμογή ανάλυσης εξαρτήσεων και τεχνικών βελτιστοποίησης ροής δεδομένων σε επίπεδο ολόκληρου του προγράμματος.

Με βάση αυτή την προσέγγιση, το Shark υλοποιεί ένα σύνολο βελτιστοποιήσεων με στόχο τη μείωση της χρήσης του συστήματος αρχείων (το οποίο αποτελεί συχνά σημείο συμφόρησης) και τη βελτίωση της τοπικότητας των δεδομένων. Ο Πίνακας 7.1 συνοψίζει τις τέσσερις κύριες κατηγορίες βελτιστοποιήσεων που εφαρμόζει το σύστημα.

Βελτιστοποίηση	**Ορισμός**	**Στόχος**
Σύστημα Αρχείων (Filesystem)	Εξάλειψη δημιουργίας περιττών αρχείων	Μείωση I/O και περιττών υπολογισμών
Διοχέτευση (Pipelining)	Μετατροπή εγγραφής/ανάγνωσης αρχείων σε αγωγούς (pipes)	Μείωση I/O, βελτίωση τοπικότητας & ταυτοχρονισμού
Παραλληλισμός (Parallelization)	Ταυτόχρονος προγραμματισμός ανεξάρτητων εντολών	Αύξηση ταυτοχρονισμού
Κλήση Εντολών (Command Invocation)	Μετασχηματισμός προγράμματος με χρήση γνώσης πεδίου (π.χ. `cat`-elimination)	Βελτιστοποίηση χρήσης εντολών

Πίνακας 7.1: Οι βελτιστοποιήσεις του Shark και οι στόχοι τους. Απόδοση του ομώνυμου πίνακα από το [15].

Για την επίτευξη αυτών των στατικών μετασχηματισμών, το Shark λειτουργεί θεωρητικά ως ένας βελτιστοποιητικός μεταγλωττιστής μέσω μίας αυστηρά καθορισμένης διαδικασίας (Σχήμα 7.1). Αρχικά, αναλύει συντακτικά τον κώδικα και κατασκευάζει ένα αφηρημένο συντακτικό δέντρο (AST). Στη συνέχεια, κανονικοποιεί τις λειτουργίες εισόδου/εξόδου μετατρέποντάς τες σε μια ενδιάμεση αναπαράσταση (*IO-Scripts*), η οποία αποτυπώνει ρητά τις αλληλεπιδράσεις με το σύστημα αρχείων. Αξιο-



Σχήμα 7.1: Επισκόπηση του Shark. Οπτικοποίηση του αγωγού μεταγλώττισης και βελτιστοποίησης. Η διαδικασία ξεκινά με την κατασκευή του συντακτικού δέντρου και καταλήγει στην παραγωγή ενός βελτιστοποιημένου σεναρίου μέσω της ανάλυσης γράφων εξαρτήσεων.

ποιώντας αυτή τη δομή, κατασκευάζει ένα κατευθυνόμενο ακυκλικό γράφημα (DAG) εξαρτήσεων δεδομένων. Πάνω σε αυτό το γράφημα εκτελείται αφαίρεση περιττών λειτουργιών I/O, εξαλείφοντας αρχεία που δημιουργούνται αλλά δεν διαβάζονται ποτέ. Τα εναπομείναντα ενδιάμεσα αρχεία μετατρέπονται σε ανώνυμους αγωγούς. Τέλος, το βελτιστοποιημένο γράφημα διασχίζεται κατά πλάτος (breadth-first) προκειμένου να εντοπιστούν ανεξάρτητοι κόμβοι, οι οποίοι μετατρέπονται πίσω σε AST ως παράλληλες εντολές προς ταυτόχρονη εκτέλεση.

Πέρα από τον συντακτικό μετασχηματισμό, στο αρχικό σύστημα δινόταν ιδιαίτερη έμφαση στη μείωση του κόστους δημιουργίας διεργασιών, μετατρέποντας συχνά χρησιμοποιούμενες εξωτερικές εντολές (όπως η `grep`) σε δυναμικές βιβλιοθήκες (shared objects / DLLs). Αν και η παρούσα αξιολόγηση εστιάζει αποκλειστικά στους *στατικούς* μετασχηματισμούς, αυτή η τεχνική δυναμικής φόρτωσης αποτελούσε βασικό πυλώνα για την ολιστική επιτάχυνση της εκτέλεσης στο πρωτότυπο σύστημα.

Είναι σημαντικό να διευκρινιστεί ότι το πρωτότυπο σύστημα Shark αποτελεί ένα ερευνητικό έργο το οποίο δεν είναι διαθέσιμο ως λογισμικό ανοιχτού κώδικα ή λειτουργικό εργαλείο. Κατά συνέπεια, στην παρούσα εργασία εφαρμόζονται *χειροκίνητα* οι θεωρητικοί συντακτικοί μετασχηματισμοί που περιγράφει η φιλοσοφία του Shark στο σύνολο των μετροπρογραμμάτων της σουίτας Koala. Στη συνέχεια, εκτελούνται τόσο οι αρχικές όσο και οι βελτιστοποιημένες εκδόσεις προκειμένου να χαρακτηριστεί η απόδοση και να αναδειχθούν τα όρια των βελτιστοποιήσεων.

7.2 Στατικός Μετασχηματισμός: Το Σενάριο `weather`

Για την κατανόηση της πρακτικής εφαρμογής αυτών των βελτιστοποιήσεων, αξιοποιείται το χαρακτηριστικό μετροπρόγραμμα `weather` (Κώδικας 4.1), το οποίο αποτελείται από έναν βρόχο `for` που εκτελεί τρεις ανεξάρτητες σωληνώσεις. Εφαρμόζοντας χειροκίνητα τις αρχές του Shark (Κώδικας 7.1), υλοποιούνται δύο κύριες παρεμβάσεις:

1. **Αφαίρεση περιττών διεργασιών (Useless Use of Cat):** Απαλείφεται η περιττή κλήση της εντολής `cat` (που απλώς διοχετεύει τα περιεχόμενα ενός αρχείου σε αγωγό), αντικαθιστώντας την με άμεση ανακατεύθυνση εισόδου (<). Αυτό εξοικονομεί τον χρόνο δημιουργίας μιας επιπλέον διεργασίας.

2. **Στατικός Παραλληλισμός:** Εκμεταλλευόμενοι τη στατική γνώση ότι οι τρεις σωληνώσεις γράφουν σε διακριτά αρχεία εξόδου και δεν επηρεάζουν η μία την άλλη (απουσία εξαρτήσεων εγγραφής/ανάγνωσης), τις τοποθετούμε στο παρασκήνιο (μέσω του τελεστή &) και εισάγουμε την εντολή συγχρονισμού `wait` στο τέλος κάθε επανάληψης του βρόχου.

```
#!/bin/bash

d="./data/temperatures"
for y in $(seq $start $end); do
    cut -c 89-92 < "$d/$y" | grep -v 999 | sort -rn | head -n 1 > "max.$y" &
    cut -c 89-92 < "$d/$y" | grep -v 999 | sort -n  | head -n 1 > "min.$y" &
    cut -c 89-92 < "$d/$y" | grep -v 999 | awk '{t+=$1; i++} END {print t/i}' >
    ↪  "avg.$y" &
    wait
done
```

Κώδικας 7.1: Το σενάριο `weather` μετασχηματισμένο με χρήση του Shark. Η διαδικασία απαίτησε χειροκίνητη παρέμβαση για την αφαίρεση της `cat` και την προσθήκη τελεστών παραλληλισμού στο παρασκήνιο.

Αυτή η προσέγγιση επιτυγχάνει ταυτόχρονη εκτέλεση σε επίπεδο εντολών, αξιοποιώντας τους διαθέσιμους πυρήνες και μειώνοντας δραστικά τον συνολικό χρόνο ολοκλήρωσης του βρόχου. [1]

7.3 Προσπάθεια Υιοθέτησης

Ενώ το παράδειγμα του `weather` απαίτησε ήπιες και κατανοητές παρεμβάσεις, η χειροκίνητη εφαρμογή της φιλοσοφίας του Shark σε ευρύτερα και πιο πολύπλοκα σενάρια ανάλυσης δεδομένων αποδείχθηκε μια χρονοβόρα και περίπλοκη διαδικασία. Χαρακτηριστικό παράδειγμα αποτελεί το μετροπρόγραμμα εξαγωγής τριγραμμάτων (`count-trigrams`) από το `nlp`. Σημειώνεται ότι ολόκληρος ο πηγαίος κώδικας των αρχικών και μετασχηματισμένων σεναρίων παρατίθεται αυτούσιος στο παράρτημα (Κώδικες C.1 και C.2).

Στην αρχική του μορφή (Κώδικας 7.2), ο αλγόριθμος ακολουθεί μια κλασική και ασφαλή προσέγγιση «δημιουργίας προσωρινών αρχείων» (tempfiles) για την αποθήκευση ενδιάμεσων καταστάσεων της ροής δεδομένων.

Για να επιτευχθούν οι στόχοι του Shark—δηλαδή η εξάλειψη της εγγραφής στον δίσκο μέσω αντικατάστασης των ενδιάμεσων αρχείων με επώνυμους αγωγούς (FIFOs) και η παραλληλοποίηση ανεξάρτητων εντολών—απαιτήθηκε εκτενής τροποποίηση της δομής του κώδικα, όπως φαίνεται στο παρακάτω απόσπασμα (Κώδικας 7.3).

Η μετατροπή αυτή αναδεικνύει τρεις βασικές προκλήσεις που καθιστούν την προσπάθεια υιοθέτησης ιδιαίτερα απαιτητική για τον μέσο χρήστη, ή ακόμα και για συστήματα αυτόματης μεταγλώττισης χωρίς ισχυρές εγγυήσεις ασφαλείας:

[1] Σημειώνεται ότι, εφόσον και οι επαναλήψεις του βρόχου είναι ανεξάρτητες, θα μπορούσε να εφαρμοστεί και παραλληλισμός σε επίπεδο επανάληψης, τοποθετώντας ολόκληρο τον βρόχο στο παρασκήνιο, όπως θα γίνει μετέπειτα με τη χρήση GNU `parallel` στο Κεφάλαιο 8.

```
9     TEMPDIR=$(mktemp -d)
10    cat > ${TEMPDIR}/${input}.words
11    tail +2 ${TEMPDIR}/${input}.words > ${TEMPDIR}/${input}.nextwords
12    tail +3 ${TEMPDIR}/${input}.words > ${TEMPDIR}/${input}.nextwords2
13    paste ${TEMPDIR}/${input}.words \${TEMPDIR}/${input}.nextwords
      ↪  ${TEMPDIR}/${input}.nextwords2 |
14    sort | uniq -c
15    rm -rf ${TEMPDIR}
```

Κώδικας 7.2: Απόσπασμα από το σενάριο `count-trigrams` του συνόλου `nlp`. Χρήση προσωρινών αρχείων για την αποθήκευση και επεξεργασία ενδιάμεσων ροών δεδομένων πριν από τη συγχώνευσή τους.

```
14    mkfifo "$f_raw1" "$f_raw2" "$f_raw3" "$f_tail2" "$f_tail3"
15    tail -n +2 < "$f_raw2" > "$f_tail2" &
16    tail -n +3 < "$f_raw3" > "$f_tail3" &
17    paste "$f_raw1" "$f_tail2" "$f_tail3" | sort | uniq -c > "$output_file" &
18    agg_pid=$!
19    tr -c 'A-Za-z' '\n' < "$input_file" | \
20    grep -v '^[[:space:]]*$' | \
21    tee "$f_raw2" "$f_raw3" > "$f_raw1"
22    wait $agg_pid
```

Κώδικας 7.3: Απόσπασμα από το σενάριο `count-trigrams` μετασχηματισμένο με χρήση του Shark. Αντικατάσταση προσωρινών αρχείων με FIFOs και προσεκτική εκκίνηση των καταναλωτών στο παρασκήνιο για αποφυγή αδιεξόδων (deadlocks).

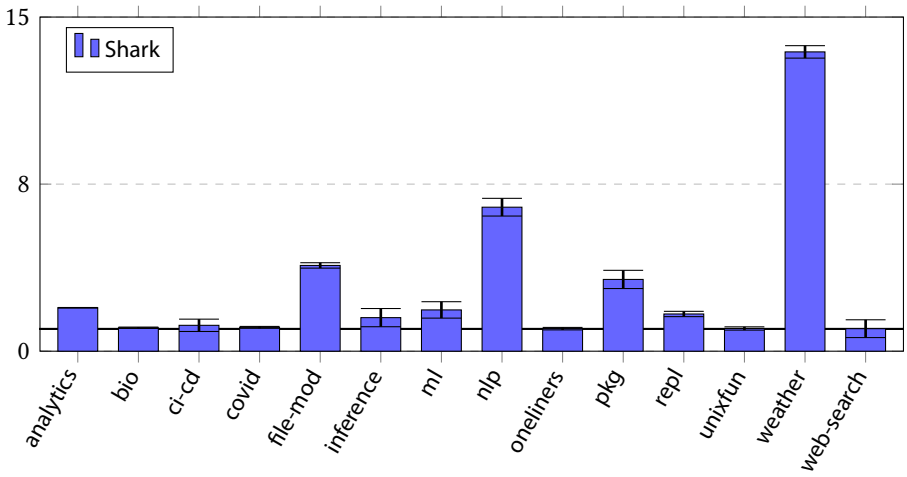
1. **Διαχείριση Πολυπλοκότητας Ροών και Αδιέξοδα (Deadlocks):** Η χρήση FIFOs επιβάλλει τη χρήση της εντολής `tee` (γραμμή 21) για τον διαχωρισμό της εισόδου σε πολλαπλά κανάλια. Αυτό εισάγει άμεσα το κλασικό πρόβλημα συγχρονισμού Παραγωγού-Καταναλωτή. Οι εντολές κατανάλωσης (π.χ. `tail` και `paste` στις γραμμές 14-16) πρέπει υποχρεωτικά να εκκινούν στο παρασκήνιο (`&`) *πριν* ο παραγωγός (`tee`) αρχίσει να γράφει στα FIFOs. Εάν η σειρά αντιστραφεί, το κέλυφος μπλοκάρει περιμένοντας αναγνώστη, οδηγώντας σε μόνιμο αδιέξοδο.

2. **Χειροκίνητος Έλεγχος Πόρων:** Καθώς το Shark βασίζεται σε στατικές παρεμβάσεις, δεν προσφέρει αυτόματο μηχανισμό διαμοιρασμού φόρτου (load balancing). Για την αποφυγή εξάντλησης των διαθέσιμων διεργασιών του συστήματος (fork bomb) κατά την εκτέλεση χιλιάδων αρχείων, απαιτήθηκε η χειροκίνητη υλοποίηση ενός μηχανισμού διαχείρισης εργασιών, με `wait` και σημαφόρους, ο οποίος παρουσιάζεται αναλυτικά στο παράρτημα (Κώδικας C.2).

3. **Πρόκληση Ορθότητας και Διόγκωση Κώδικα:** Η αναγνωσιμότητα του κώδικα μειώνεται αισθητά. Η πραγματική επιχειρησιακή λογική επικαλύπτεται από πολύπλοκες λεπτομέρειες διαχείρισης διεργασιών, ενώ ο κίνδυνος εισαγωγής συνθηκών ανταγωνισμού (race conditions) είναι υψηλός, αυξάνοντας σημαντικά τον χρόνο ανάπτυξης και αποσφαλμάτωσης (debugging).

7.4 Ανάλυση Απόδοσης

Όπως αποτυπώνεται στο Σχήμα 7.2, η στρατηγική του Shark πέτυχε σημαντικά κέρδη απόδοσης σε ολόκληρη τη σουίτα Koala, με τις επιταχύνσεις να κυμαίνονται μεταξύ **1.01× και 13.43×**, ανά-



Σχήμα 7.2: Σχετικές επιταχύνσεις του Shark στα μετροπρογράμματα της σουίτας Koala. Το σύστημα επιτυγχάνει σημαντικές επιταχύνσεις σε σενάρια με παραλληλισμό βρόχων, αλλά υστερεί σε αυστηρά σειριακούς φόρτους ή διασυνδεδεμένους αγωγούς.

λογα με τα χαρακτηριστικά του εκάστοτε σεναρίου. Οι πιο θεαματικές βελτιώσεις παρατηρήθηκαν σε μετροπρογράμματα που περιλαμβάνουν τετριμμένα παραλληλοποιήσιμες (trivially parallelizable) επαναλήψεις βρόχων. Ενδεικτικά, το Shark βελτίωσε τα σύνολα `weather` και `nlp` κατά **13.43×** και **6.46×** αντίστοιχα, καθώς ανεξάρτητες εργασίες προγραμματίστηκαν ταυτόχρονα αξιοποιώντας αποτελεσματικά τους διαθέσιμους πόρους. Αντιθέτως, σενάρια με ισχυρές αλληλεξαρτήσεις που χρησιμοποιούν ήδη εκτενώς σωληνώσεις, όπως τα `covid`, `oneliners`, `bio` και `unixfun`, προσφέρουν λιγότερες ευκαιρίες για τις παρεμβάσεις του Shark, οδηγώντας σε οριακές επιταχύνσεις που κυμάνθηκαν μεταξύ **1.01× και 1.06×**. Ομοίως, σενάρια με αυστηρές σειριακές λειτουργίες, όπως εκείνα του συνόλου `ci-cd`, εμφάνισαν περιορισμένη βελτίωση (**1.16×**). Το μετροπρόγραμμα `web-search` παρουσίασε τη μικρότερη βελτίωση (**1.01×**), καθώς η αρχική του υλοποίηση εκτελεί ήδη τρεις υπολογισμούς n-gram παράλληλα, μην επιτρέποντας περαιτέρω παραλληλισμό. Στο ίδιο πλαίσιο, οι βελτιστοποιήσεις που επικεντρώθηκαν αποκλειστικά στην κλήση εντολών (όπως η αφαίρεση της `cat` ή η προσθήκη της σημαίας `--posix`) δεν επέφεραν σημαντικά οφέλη: σε σενάρια όπου υπήρχαν *μόνο* τέτοιες ευκαιρίες (όπως στο `web-search`), η μέση επιτάχυνση ήταν περιορισμένη. Αντίθετα, οι βελτιστοποιήσεις που εξαλείφουν προσωρινά αρχεία για ενδιάμεση αποθήκευση μπορούν να προσφέρουν πιο ουσιαστικές επιταχύνσεις, όμως, όπως αναλύθηκε στην Ενότητα 7.3, εισάγουν σημαντική πολυπλοκότητα λόγω της ανάγκης για αυστηρό συντονισμό μεταξύ παραγωγών και καταναλωτών.

Αξίζει να υπογραμμιστεί ότι η στατική φύση του μετασχηματισμού διασφαλίζει πως το Shark προσφέρει αποκλειστικά επιταχύνσεις (ακόμη και οριακές της τάξης του 1.01×), χωρίς να εισάγει καμία επιβάρυνση χρόνου εκτέλεσης (runtime overhead) σε αντίθεση με τα εργαλεία δυναμικής ανάλυσης. Αυτό συμβαίνει διότι οι βελτιστοποιήσεις του συστήματος αποτελούν, ουσιαστικά, την αυτοματοποιημένη εφαρμογή καθιερωμένων βέλτιστων πρακτικών προγραμματισμού κελύφους (shell scripting best practices). Σήμερα, αρκετές από αυτές τις τακτικές—όπως η εξάλειψη της περιττής χρήσης της `cat`—έχουν ενσωματωθεί ευρέως ως κανόνες στατικής ανάλυσης σε σύγχρονα εργαλεία ελέγχου κώδικα (linters), όπως το ShellCheck [99], επιβεβαιώνοντας τη διαχρονική αξία της φιλοσοφίας του Shark.

7.5 Σύνοψη

Τα αποτελέσματα στο Koala επιβεβαιώνουν ότι οι βελτιστοποιήσεις του Shark είναι ιδιαίτερα αποδοτικές σε σενάρια που περιλαμβάνουν λειτουργίες σε πολλαπλές εισόδους και ανεξάρτητες εντολές, προσφέροντας επιταχύνσεις έως και **13.43×**. Ωστόσο, αποδεικνύονται λιγότερο αποτελεσματικές για scripts που περιορίζονται από την είσοδο/έξοδο (I/O-bound) ή για σενάρια που δεν είναι εύκολα

παραλληλοποιήσιμα λόγω ισχυρών εξαρτήσεων. Παράλληλα, η αξιολόγηση δείχνει ότι η σημαντική προσπάθεια υιοθέτησης καθιστά την αυστηρά στατική προσέγγιση λιγότερο ελκυστική για τον μέσο χρήστη που αναζητά εύκολη και ασφαλή επιτάχυνση.

Στο επόμενο κεφάλαιο εξετάζουμε το εργαλείο GNU `parallel`, το οποίο απαιτεί επίσης χειροκίνητη παρέμβαση στον κώδικα, αλλά αυξάνει τις επιδόσεις που μπορούν να επιτευχθούν μέσω ρητής παραλληλοποίησης δεδομένων.

Κεφάλαιο 8

Αξιολόγηση GNU `parallel`: Χειροκίνητος Παραλληλισμός

Στο παρόν κεφάλαιο εξετάζεται η προσέγγιση του ρητού παραλληλισμού, με κύριο όχημα αξιολόγησης το GNU `parallel` [18]. Σε αντίθεση με τη στατική ανάλυση του Shark που επιχειρεί να βελτιστοποιήσει τον κώδικα με βάση αναγνωρισμένα μοτίβα εισόδου/εξόδου, το GNU `parallel` αποτελεί ένα εργαλείο που παρέχει πλήρη έλεγχο στον προγραμματιστή, απαιτώντας ωστόσο την πλήρη παρέμβασή του για την αναδιάρθρωση της λογικής του προγράμματος.

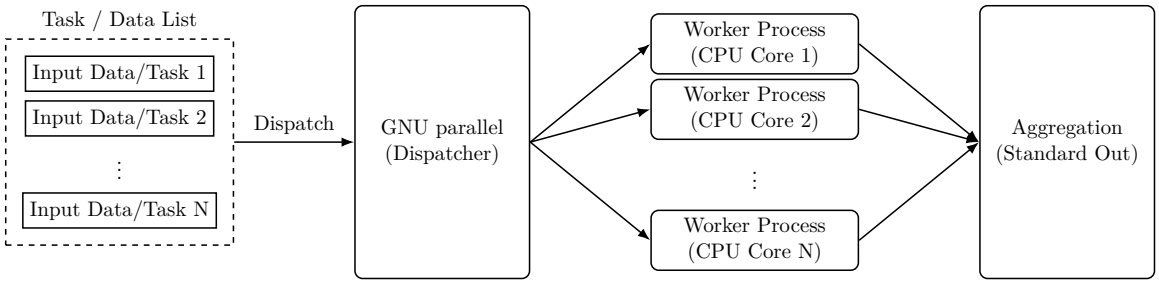
8.1 Μηχανισμός Λειτουργίας και Χαρακτηριστικά

Το GNU `parallel` αποτελεί ένα από τα πλέον διαδεδομένα εργαλεία χειροκίνητης παραλληλοποίησης στο οικοσύστημα του Unix. Η λειτουργικότητά του παρέχεται μέσω της εντολής `parallel` η οποία προσφέρει τη δυνατότητα διάσπασης συνόλων δεδομένων και ταυτόχρονης εκτέλεσης πολλαπλών ανεξάρτητων διεργασιών. Αρχικός σκοπός της δημιουργίας του ήταν να λειτουργήσει όπως η εντολή `xargs`, αλλά με ικανότητα να διαχειρίζεται ορθά τα κενά διαστήματα (spaces) και τα εισαγωγικά (quotes) στα ορίσματα. Το εργαλείο υποστηρίζει τόσο παραλληλία δεδομένων (data parallelism) όσο και παραλληλία εργασιών (task parallelism), ενώ παρέχει τη δυνατότητα εκτέλεσης διεργασιών σε απομακρυσμένους κόμβους μέσω ασφαλών συνδέσεων. Από προεπιλογή, το GNU `parallel` εκτελεί μία εργασία ανά διαθέσιμο πυρήνα. Ένα από τα ισχυρότερα χαρακτηριστικά του είναι η ικανότητα να διαβάζει δεδομένα από μια σωλήνωση (pipe), να τα διασπά σε μπλοκ (blocks) και να διοχετεύει κάθε μπλοκ σε ξεχωριστές, ταυτόχρονες εντολές. Σε αντίθεση με τα αυτόματα συστήματα παραλληλοποίησης, το GNU `parallel` δεν αξιοποιεί καμία πληροφορία σχετικά με τη σημασιολογία, το αφηρημένο συντακτικό δέντρο ή τις εξαρτήσεις του προγράμματος. Η ευθύνη για τον ορθό διαχωρισμό των δεδομένων και τη διατήρηση της λειτουργικής ορθότητας μεταφέρεται εξ ολοκλήρου στον χρήστη. Λανθασμένες παραμετροποιήσεις μπορούν να οδηγήσουν τόσο σε σημαντική επιβράδυνση όσο και στην παραγωγή εσφαλμένων αποτελεσμάτων. Το GNU `parallel` χρησιμοποιείται στην παρούσα εργασία ως βασικό σημείο αναφοράς, αντιπροσωπεύοντας την προσέγγιση της χειροκίνητης παραλληλοποίησης σε επίπεδο χρήστη.

8.2 Χειροκίνητη Παραλληλοποίηση: Το Σενάριο `weather`

Για να γίνει κατανοητός ο τρόπος λειτουργίας του εργαλείου, αξιοποιείται εκ νέου το κινητήριο παράδειγμα του μετροπρογράμματος `weather` (Κώδικας 4.1). Για την παραλληλοποίηση μέσω του GNU `parallel`, δεν αρκεί η απλή προσθήκη μιας σημαίας στο κέλυφος. Αντιθέτως, το πρόγραμμα πρέπει να μετασχηματιστεί (Κώδικας 8.1). Συγκεκριμένα, η λογική του βρόχου πρέπει να ενθυλακωθεί σε μια διακριτή συνάρτηση (στο παράδειγμα ονομάζεται `process_year`). Κατόπιν, η τροφοδοσία των ετών υλοποιείται συνήθως μέσω ενός γεννήτορα (όπως η εντολή `seq`), ο οποίος διοχετεύει τα δεδομένα απευθείας στο `parallel`.

Αυτή η προσέγγιση, αν και αποδοτική, μετατρέπει ένα ενιαίο δομημένο σενάριο σε ένα σύνολο από διακριτές συναρτήσεις.



Σχήμα 8.1: Επισκόπηση του GNU `parallel`. Ο διανομέας (Dispatcher) λαμβάνει μια λίστα εργασιών ή δεδομένων και τις αναθέτει δυναμικά σε πολλαπλούς εργάτες (Worker Processes) που εκτελούνται παράλληλα σε διαφορετικούς πυρήνες του επεξεργαστή. Τα επιμέρους αποτελέσματα συλλέγονται και συγκεντρώνονται (Aggregation) στην τυπική έξοδο.

```
#!/bin/bash

d="./data/temperatures"
process_year() {
    y=$1
    cut -c 89-92 "$d/$y" | grep -v 999 | sort -rn | head -n 1 > "max.$y"
    cut -c 89-92 "$d/$y" | grep -v 999 | sort -n  | head -n 1 > "min.$y"
   cut -c 89-92 "$d/$y" | grep -v 999 | awk '{t+=$1; i++} END{print t/i}' > "avg.$y"
}
export -f process_year
export d
seq "$start" "$end" | parallel --jobs "$(nproc)" process_year
```

Κώδικας 8.1: Το σενάριο `weather` μετασχηματισμένο με χρήση του GNU `parallel`. Η διαδικασία απαιτεί ριζική τροποποίηση της αρχιτεκτονικής του κώδικα, εξαγωγή συναρτήσεων και ρητή κλήση του παραλληλοποιητή.

8.3 Προσπάθεια Υιοθέτησης

Η εφαρμογή του εργαλείου στα μετροπρογράμματα του Koala ήταν εστιασμένη σε σενάρια τα οποία αποτελούν φυσικές επιλογές για παράλληλη εκτέλεση λόγω της δομής τους. Η έμφαση δόθηκε σε ροές που επεξεργάζονται ανεξάρτητα αρχεία εισόδου, και στην αποδοτική επεξεργασία βασισμένη σε τμήματα (segment-based processing) που απαιτεί ελάχιστο ή καθόλου συγχρονισμό, αξιοποιώντας συχνά την παράμετρο `--pipe` για τον παραλληλισμό ροών δεδομένων. Παρά την επιλεκτική αυτή χρήση, η ορθή εφαρμογή του GNU `parallel` αποδεικνύεται στην πράξη μια εξαιρετικά απαιτητική, μη τετριμμένη διαδικασία [6] και η μετατροπή σειριακών σωληνώσεων σε παράλληλες απαιτεί συχνά πολλαπλές επαναλήψεις για την επίτευξη ορθότητας και βέλτιστης επίδοσης.

Εξαγωγή Περιβάλλοντος και Συντακτική Πολυπλοκότητα Το GNU `parallel` εκκινεί νέες διεργασίες κελύφους (`bash -c`) στο παρασκήνιο. Για να είναι διαθέσιμες οι τοπικές συναρτήσεις σε αυτά τα υποκελύφη, ο χρήστης πρέπει υποχρεωτικά να τις εξάγει στο περιβάλλον μέσω της εντολής `export -f`. Επιπλέον, η διαχείριση μεταβλητών και συμβολοσειρών εντός της παραμέτρου εκτέλεσης απαιτεί πολύπλοκες δομές χαρακτήρων διαφυγής (escaping), καθιστώντας την αποσφαλμάτωση (debugging) ιδιαίτερα δυσχερή.

Διαχείριση Ενδιάμεσων Δεδομένων και Σωληνώσεων Σε αντίθεση με την απλή σύνταξη των αγωγών (`|`) του κελύφους, ο χρήστης του GNU `parallel` πρέπει συχνά να διαχειριστεί ρητά τη ροή των δεδομένων. Στο παράδειγμα του σεναρίου `covid-1` από το σύνολο `covid`, παρατηρείται η ανάγκη χρήσης προσωρινών καταλόγων (`mktemp`), ορισμού μεγέθους τεμαχίων δεδομένων μέσω της παραμέτρου `--block` (`chunk_size`), και χρήσης μηχανισμών καθαρισμού (`trap`) για τα προσωρινά αρχεία. Η απλή σωλήνωση του αρχικού σεναρίου (Κώδικας 8.2) μετατρέπεται σε ένα σύνθετο σενάριο (Κώδικας 8.3) που απαιτεί εξαγωγή συναρτήσεων και ρητό συντονισμό μέσω της παραμέτρου `--pipe`.

```
cat "$1" |
  sed 's/T..:..:..//' |
  cut -d ',' -f 1,3 |
```

Κώδικας 8.2: Απόσπασμα από το σενάριο `covid-1` του συνόλου `covid`. Το σενάριο περιέχει μία σωλήνωση με εύκολα παραλληλοποιήσιμα αρχικά στάδια. Ο πλήρης κώδικας παρατίθεται στο παράρτημα (Κώδικας C.3).

```
chunk_size=${chunk_size:-100M}
process_chunk() {
  sed 's/T..:..:..//'| cut -d ',' -f 1,3
}
export -f process_chunk
tmp_dir=$(mktemp -d)
trap "rm -rf $tmp_dir" EXIT
cat "$INPUT" | parallel --pipe --block "$chunk_size" -j "$MAX_PROCS" process_chunk
 ↪  > "$tmp_dir/combined.tmp"
```

Κώδικας 8.3: Απόσπασμα από το σενάριο `covid-1` μετασχηματισμένο με χρήση του GNU `parallel`. Η παραλληλοποίηση απαιτεί χειροκίνητο τεμαχισμό δεδομένων. Ο πλήρης κώδικας παρατίθεται στο παράρτημα (Κώδικας C.4).

Πολυπλοκότητα Σύνταξης και Ορθότητα Για να προχωρήσουμε σε πιο έντονη παραλληλοποίηση, όταν η παραλληλία δεν είναι εμφανής, ανακύπτει συχνά μία νέα πρόκληση, ειδικά σε σενάρια που απαιτούν συγχρονισμό ή διατήρηση της σειράς των δεδομένων. Για παράδειγμα, όταν παραλληλοποιούμε τα τελευταία στάδια σωληνώσεων του σεναρίου `spell` (Κώδικας 8.4), δημιουργείται η ανάγκη για διατήρηση της σειράς (ώστε να λειτουργήσουν σωστά εντολές όπως η `uniq`). Έτσι, επιβάλλεται η χρήση της σημαίας `-k` (*keep order*) [6], με την παράλειψή της να οδηγεί σε λανθασμένα αποτελέσματα λόγω του μη-ντετερμινιστικού χρονοπρογραμματισμού των εργασιών.

Επιπλέον, η διαχείριση πολλαπλών ροών εισόδου/εξόδου απαιτεί συχνά τη χειροκίνητη δημιουργία επώνυμων σωληνώσεων (`mkfifo`) και πολύπλοκες δομές εντολών που περιλαμβάνουν χαρακτήρες διαφυγής. Όπως φαίνεται στον Κώδικα 8.5, η εντολή γίνεται δυσανάγνωστη, ενώ η απόδοση εξαρτάται δραματικά από «μαγικές» σταθερές όπως το `--block`, όπου λανθασμένη ρύθμιση μπορεί να οδηγήσει σε δραματικές καθυστερήσεις [6].

8.4 Ανάλυση Απόδοσης

Συνολικά, το GNU `parallel` πέτυχε μια **μέση επιτάχυνση της τάξης του 2.6×** σε όλη τη σουίτα, με τη διακύμανση μεταξύ των σεναρίων να είναι μεγάλη, αλλά τα αποτελέσματα να παραμένουν θετικά

```
#!/bin/bash
# Calculate mispelled words in an input

dict=$SUITE_DIR/inputs/dict.txt

cat $1 |
    sed 's/[^[:print:]]//g' |      # remove non-printing characters
    col -bx |                      # remove backspaces / linefeeds
    tr -cs A-Za-z '\n' |           # split on non-alphabetic characters
    tr A-Z a-z |                   # map upper to lower case
    tr -d '[:punct:]' |            # remove punctuation
    sort |                         # put words in alphabetical order
    uniq |                         # remove duplicate words
    comm -23 - $dict               # report words not in dictionary
```

Κώδικας 8.4: Το σενάριο `spell` του συνόλου `oneliners`. Τα πρώτα στάδια της σωλήνωσης είναι εύκολα παραλληλοποιήσιμα, ενώ τα τελευταία τρία χρειάζονται προσεκτική διαχείριση.

```
mkfifo $TEMP1
parallel "cat {} | col -bx | tr -cs A-Za-z '\n' | tr A-Z a-z | \
tr -d '[:punct:]' | sort > $TEMP_C1" ::: $IN &
sort -m $TEMP1 | parallel -k --jobs ${JOBS} --pipe --block "$BLOCK_SIZE" "uniq" |
uniq | parallel -k --jobs ${JOBS} --pipe --block "$BLOCK_SIZE" "grep -vx -f $dict -"
```

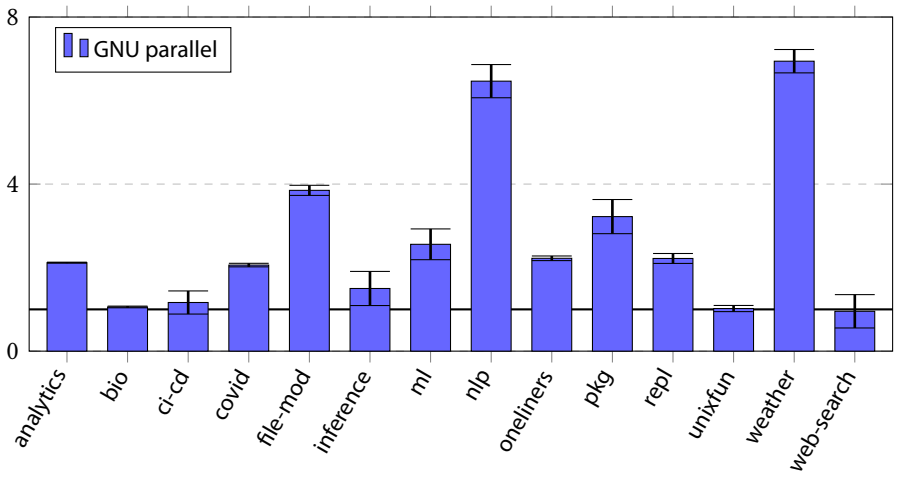
Κώδικας 8.5: Απόσπασμα από το σενάριο `spell` μετασχηματισμένο με χρήση του GNU `parallel`. Απαιτούνται FIFOs και χρήση της σημαίας `-k` που διατηρεί τη σειρά εξόδου, εξασφαλίζοντας ορθότητα. Ο πλήρης κώδικας παρατίθεται στο παράρτημα (Κώδικας C.5).

για σχεδόν όλα τα μετροπρογράμματα.

Όπως απεικονίζεται στο Σχήμα 8.2, η ανταμοιβή για την υψηλή προσπάθεια υιοθέτησης είναι εντυπωσιακή σε σενάρια που περιορίζονται από την είσοδο/έξοδο (I/O-bound) ή είναι εύκολα παραλληλοποιήσιμα. Για παράδειγμα, το **`file-mod`**, το οποίο μετατρέπει ταυτόχρονα πολλαπλά αρχεία πολυμέσων με υψηλή αξιοποίηση των διαθέσιμων πυρήνων, επιταχύνθηκε κατά **3.85×**. Παρομοίως, το **`nlp`**, το οποίο επεξεργάζεται πολλαπλά αρχεία δεδομένων ανεξάρτητα, σημείωσε επιτάχυνση **6.46×**.

Αντιθέτως, οι επιταχύνσεις του GNU `parallel` είναι περιορισμένες σε σενάρια που στερούνται αυτών των χαρακτηριστικών. Στο **`ci-cd`** **(1.16×)**, η βελτίωση ήταν ελάχιστη, καθώς τα σενάρια αυτά είτε αξιοποιούν ήδη εσωτερικό παραλληλισμό κατά την κλήση των εντολών τους (π.χ. μεταγλώττιση πολλαπλών αρχείων), είτε περιλαμβάνουν εντολές που δεν επωφελούνται από το μοντέλο παραλληλοποίησης του `parallel` (όπως η `git` ή η `find`).

Τέλος, υπήρξαν περιπτώσεις όπου το εργαλείο δεν απέφερε ουσιαστική βελτίωση ή και προκάλεσε ελαφριά επιβράδυνση λόγω της φύσης του σεναρίου. Στο μετροπρόγραμμα **`unixfun`** **(1.02×)**, τα στάδια της σωλήνωσης εξαρτώνται άμεσα από την έξοδο των προηγούμενων σταδίων. Αυτή η αλληλεξάρτηση απέτρεψε το GNU `parallel` από το να εκμεταλλευτεί πλήρως τον παραλληλισμό. Ομοίως, σενάρια που απαιτούν πολύπλοκο διαχωρισμό και συγχώνευση, όπως το **`web-search`** **(0.95×)**, περιόρισαν δραστικά τις δυνατότητες του εργαλείου, καταγράφοντας επιβράδυνση.



Σχήμα 8.2: Σχετικές επιταχύνσεις του GNU `parallel` στα μετροπρογράμματα της σουίτας Koala. Το εργαλείο προσφέρει κορυφαίες επιταχύνσεις σε απολύτως παραλληλοποιήσιμα σενάρια, αλλά επιβαρύνει μικρούς φόρτους εργασίας.

8.5 Σύνοψη

Το GNU `parallel` αντιπροσωπεύει μια προσέγγιση άμεσης, ρητής, χειροκίνητης παραλληλοποίησης στο οικοσύστημα του Unix. Η αξιολόγηση στο Koala επιβεβαιώνει ότι μπορεί να επιταχύνει σημαντικά I/O-bound εργασίες και βρόχους χωρίς αλληλεξαρτήσεις (όπως το `file-mod`), προσφέροντας κατά μέσο όρο επιτάχυνση **2.6×**. Ωστόσο, η απόδοσή του περιορίζεται σημαντικά σε σενάρια με σειριακές λειτουργίες ή πολύπλοκες απαιτήσεις διαχωρισμού και συγχώνευσης (π.χ. `unixfun`, `web-search`).

Επιπλέον, το τίμημα αυτής της απόδοσης είναι το σημαντικό κόστος υιοθέτησης. Η ανάγκη για αναδιάρθρωση του κώδικα, εξαγωγή συναρτήσεων, χειροκίνητη διαχείριση προσωρινών αρχείων και προσεκτική τοποθέτηση σημαιών (όπως η `-k`), μεταθέτει το βάρος της ορθότητας εξ ολοκλήρου στον προγραμματιστή, καθιστώντας το λιγότερο κατάλληλο για την αυτοματοποιημένη βελτιστοποίηση λογισμικού μεγάλης κλίμακας.

Τα ευρήματα αυτά καταδεικνύουν ένα σαφές κενό: η κοινότητα χρειάζεται συστήματα που να προσφέρουν τις εντυπωσιακές επιταχύνσεις του GNU `parallel`, αλλά με την ασφάλεια και την ευκολία χρήσης ενός απλού διερμηνέα (όπως είδαμε στο `zsh`). Το επόμενο κεφάλαιο εισάγει την έννοια της αυτόματης παραλληλοποίησης μέσω του συστήματος PaSh, το οποίο επιχειρεί να γεφυρώσει αυτό το χάσμα, προσφέροντας ασφαλή, αυτόματο παραλληλισμό δεδομένων με ελάχιστη παρέμβαση χρήστη.

Κεφάλαιο 9

Αξιολόγηση PaSh: Αυτόματη Δυναμική Παραλληλοποίηση

Τα προηγούμενα κεφάλαια ανέδειξαν ότι, ενώ η χειροκίνητη παραλληλοποίηση (GNU `parallel`) και η στατική βελτιστοποίηση (Shark) μπορούν να αποφέρουν σημαντικά κέρδη απόδοσης, απαιτούν ριζικές παρεμβάσεις στον πηγαίο κώδικα και μεταθέτουν το βάρος της ορθότητας στον προγραμματιστή. Στο παρόν κεφάλαιο εξετάζεται μια σύγχρονη, εναλλακτική προσέγγιση μέσω του PaSh, το οποίο επιχειρεί να προσφέρει αυτοματοποιημένη επιτάχυνση διατηρώντας την αρχική σημασιολογία του προγράμματος.

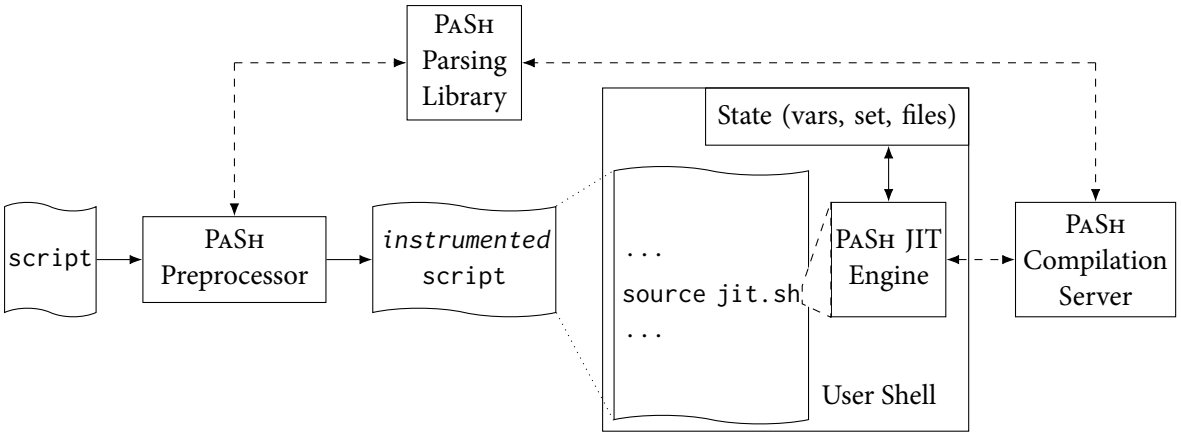
9.1 Μηχανισμός Λειτουργίας και Χαρακτηριστικά

Το PaSh [7] αποτελεί ένα σύστημα αυτόματης παραλληλοποίησης προγραμμάτων κελύφους POSIX, το οποίο εκτελεί προγράμματα Unix με παράλληλο τρόπο μέσω μηχανισμού δυναμικής μεταγλώττισης (Just-in-Time - JIT). Για την αντιμετώπιση της δυναμικής φύσης του κελύφους, το PaSh εναλλάσσει φάσεις μεταγλώττισης και εκτέλεσης, συλλέγοντας σε πραγματικό χρόνο πληροφορίες για την κατάσταση του κελύφους, του συστήματος αρχείων και του περιβάλλοντος.

Το σύστημα (Σχήμα 9.1) αναγνωρίζει δυνητικά παραλληλοποιήσιμες περιοχές του προγράμματος και τις μετατρέπει σε γράφους ροής δεδομένων, στους οποίους κάθε εντολή αντιστοιχεί σε κόμβο και κάθε ροή δεδομένων σε ακμή. Η πληροφορία παραλληλοποιησιμότητας κωδικοποιείται μέσω ενός συστήματος *επισημειώσεων (annotations)*, το οποίο περιγράφει τις κατηγορίες παραλληλισμού των εντολών και λαμβάνει υπόψη τα ορίσματα και τις επιλογές τους. Στη συνέχεια, εφαρμόζονται μετασχηματισμοί στους γράφους αυτούς με στόχο τον εντοπισμό παραλληλίας δεδομένων (data parallelism) και εργασιών, όπως η διάσπαση των ροών σε ανεξάρτητα τμήματα και η αναδιοργάνωση των εντολών. Μετά την ολοκλήρωση των μετασχηματισμών, οι γράφοι μετατρέπονται εκ νέου σε κώδικα κελύφους, στον οποίο εισάγονται κατάλληλες εντολές παραλληλισμού (π.χ. `&`, `wait`) και επώνυμοι αγωγοί, ώστε να υλοποιείται ρητά η παράλληλη εκτέλεση. Ο βαθμός παραλληλίας είναι παραμετροποιήσιμος και ρυθμίζεται δυναμικά με χρήση της σημαίας `--width`. Για τη διασφάλιση της λειτουργικής ισοδυναμίας με τη σειριακή εκτέλεση, το PaSh βασίζεται σε ένα μοντέλο ροής δεδομένων με επίγνωση σειράς (order-aware dataflow) [6], το οποίο διατηρεί τις απαραίτητες εξαρτήσεις μεταξύ εισόδων και εξόδων. Επιπλέον, πριν από κάθε μετασχηματισμό, αποθηκεύεται η κατάσταση του κελύφους για την αποφυγή παρενεργειών. Σε περιπτώσεις όπου η δυναμική μεταγλώττιση αποτυγχάνει ή η παραλληλοποίηση κρίνεται μη ασφαλής (π.χ. άγνωστες εντολές), το PaSh επιστρέφει αυτόματα (fallback) στη σειριακή εκτέλεση.

9.2 Μεταγλώττιση Πάνω-στην-Ώρα: Το Σενάριο `weather`

Επιστρέφοντας στο παράδειγμα, παρότι ο χρήστης εκτελεί το αρχικό σενάριο `weather` (Κώδικας 4.1) αμετάβλητο, το PaSh παρεμβάλλεται δυναμικά κατά την εκτέλεσή του. Διαβάζοντας τις διαθέσιμες επισημειώσεις, το σύστημα αντιλαμβάνεται ότι η εντολή `cut` στην πρώτη σωλήνωση είναι πλήρως παραλληλοποιήσιμη (χωρίς εσωτερική κατάσταση), ενώ η εντολή `sort` απαιτεί ειδική λειτουργία συγχώνευσης (`merge`). Βάσει αυτής της πληροφορίας, το PaSh κατασκευάζει και εκτελεί εσωτερικά έναν γράφο ροής δεδομένων: διασπά αυτόματα την είσοδο (`split`), τη διοχετεύει σε πολλαπλούς



Σχήμα 9.1: Επισκόπηση του PaSh. Το PaSh ενορχηστρώνει τα σενάρια με κλήσεις στη μηχανή πάνω-στην-ώρα, η οποία κατά την εκτέλεση προωθεί τμήματα του προγράμματος στον διακομιστή μεταγλώττισης (PaSh compilation server) για δυναμική επεξεργασία κατά την εκτέλεση.

επώνυμους αγωγούς (FIFOs) που αντιστοιχούν στο πλάτος παραλληλισμού (width), επεξεργάζεται τα δεδομένα παράλληλα, και τελικά συγχωνεύει τα επιμέρους αποτελέσματα (αξιοποιώντας την `sort -m`).

Παρακάτω φαίνεται μία απλοποιημένη αναπαράσταση του κώδικα κελύφους που παράγεται αυτόματα από το PaSh για τον συντονισμό αυτών των λειτουργιών (Κώδικας 9.1), ενώ το εκτενές πρόγραμμα παρατίθεται, για λόγους πληρότητας, στο παράρτημα (Κώδικας B.1).

9.3 Προσπάθεια Υιοθέτησης

Σε αντίθεση με το Shark και το GNU `parallel`, η αρχιτεκτονική του PaSh προσφέρει μια εμπειρία *μηδενικής προσπάθειας* (zero-effort) υιοθέτησης από την πλευρά του χρήστη. Ο προγραμματιστής απαλλάσσεται από την ανάγκη τροποποίησης του αρχικού κώδικα.

Η αξιολόγηση μέσω της σουίτας Koala πραγματοποιήθηκε απαιτώντας αποκλειστικά τον ορισμό της αντίστοιχης μεταβλητής περιβάλλοντος στον οδηγό της σουίτας Koala. Ο βαθμός παραλληλίας για τα πειράματα ήταν 4, και η αντίστοιχη μεταβλητή περιβάλλοντος ορίστηκε ως εξής:

```
KOALA_SHELL="./pa.sh --width 4"
```

Σημειώνεται ότι το PaSh διαθέτει τη δυνατότητα υποκατάστασης άγνωστων τμημάτων κώδικα μέσω `alias` ή συναρτήσεων που μπορούν να επισημειωθούν χειροκίνητα με πληροφορίες παραλληλισμού. Αν και αυτή η πρακτική θα μπορούσε να εξαγάγει περαιτέρω επιταχύνσεις, αποφασίσαμε να *μην* την εφαρμόσουμε, καθώς θα απαιτούσε σημαντικό επιπλέον φόρτο εργασίας, αναιρώντας το βασικό πλεονέκτημα του συστήματος.

9.4 Ανάλυση Απόδοσης

Τα αποτελέσματα απόδοσης απεικονίζονται στο Σχήμα 9.2. Το PaSh επέτυχε σημαντικές επιταχύνσεις σε σενάρια με σωληνώσεις πολλαπλών σταδίων (multi-stage pipelines) ή βρόχους `for` χωρίς εξαρτήσεις δεδομένων μεταξύ των επαναλήψεων. Ενδεικτικά, στα σύνολα **`oneliners`** και **`covid`** πέτυχε επιτάχυνση **2.15×** και **1.83×** αντίστοιχα, καθώς βασίζονται σε κλασικά εργαλεία POSIX που περιλαμβάνονται στη βιβλιοθήκη επισημειώσεων του συστήματος. Ωστόσο, η αξιολόγηση μέσω του Koala ανέδειξε κρίσιμους περιορισμούς που επηρεάζουν δραστικά τη γενίκευση της απόδοσής του.

Ο σημαντικότερος περιορισμός έγκειται στο πρόβλημα των επισημειώσεων (annotation burden). Σε πεδία όπου ο υπολογιστικός φόρτος διεξάγεται από εξειδικευμένα εκτελέσιμα προγράμματα για τα

```
#!/bin/bash

mkfifo f{0..10}
cat "./data/temperatures/2000" >f0 &
split f0 f1 f2 &
cut -c 89-92 <f1 >f3 &
cut -c 89-92 <f2 >f4 &
grep -v 999 <f3 >f5 &
grep -v 999 <f4 >f6 &
sort -rn <f5 >f7 &
sort -rn <f6 >f8 &
sort -rn -m f8 f9 >f10 &
head -n 1 <f10 >"max.2000" &
# ...
# --- Alternatively, for the AVG calculation (Merge -> Sequential Awk) ---
merge f9 10 | awk '{t+=$1; i++} END {if (i>0) print t/i}' > "avg.2000"
wait
rm f{0..10}
```

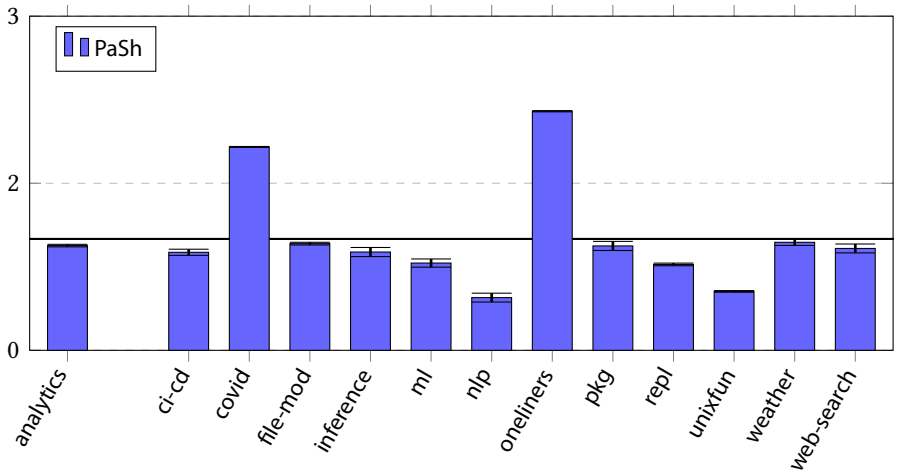
Κώδικας 9.1: Απλοποιημένη αναπαράσταση για το σενάριο `weather` μετασχηματισμένο με χρήση του PaSh. Παρουσιάζεται ο γράφος ροής δεδομένων (DFG) που δημιουργεί και εκτελεί αυτόματα το PaSh στο παρασκήνιο για μία εκ των σωληνώσεων του σεναρίου.

οποία το PaSh στερείται επισημειώσεων, το σύστημα δεν εφαρμόζει παραλληλισμό. Ως αποτέλεσμα, σε σύνολα όπως τα **pkg** (**0.94**×) και **file-mod** (**0.96**×), δεν παρατηρείται επιτάχυνση, αλλά αντίθετα καταγράφεται μία μικρή επιβράδυνση λόγω της επιβάρυνσης (overhead) της δυναμικής ανάλυσης.

Επιπλέον, η απουσία κατάλληλων συντακτικών δομών και το αυξημένο κόστος μεταγλώττισης αποτελούν πρόσθετους ανασταλτικούς παράγοντες. Το PaSh προϋποθέτει την ύπαρξη συγκεκριμένων συντακτικών μοτίβων (π.χ. μεγάλες σωληνώσεις ή data-parallel βρόχους) για να επέμβει. Σενάρια όπως τα **repl** (**0.77**×) και **ci-cd** (**0.88**×), τα οποία αποτελούνται κυρίως από σειριακές εντολές συστήματος και δεν περιέχουν δομές που να επιδέχονται παραλληλισμό, επιβραδύνονται περαιτέρω, καθώς εισάγεται επιπλέον επιβάρυνση (overhead) λόγω της JIT μηχανής. Τέλος, αξίζει να σημειωθεί ότι στην περίπτωση του συνόλου **bio**, η εκτέλεση απέτυχε πλήρως, καταδεικνύοντας τις δυσκολίες της δυναμικής μεταγλώττισης στη διαχείριση εξαιρετικά πολύπλοκων εξαρτήσεων.

9.5 Σύνοψη

Η αξιολόγηση στο Koala καταδεικνύει ότι το PaSh μπορεί να προσφέρει ουσιαστικές επιταχύνσεις σε σενάρια που εμπίπτουν στο πεδίο παραλληλισμού του, με έναν αυτοματοποιημένο και ασφαλή τρόπο. Ωστόσο, η αποτελεσματικότητά του εξαρτάται άμεσα από δύο βασικούς παράγοντες: τη διαθεσιμότητα επισημειώσεων για τις εκάστοτε εντολές και το κατά πόσο η δομή των προγραμμάτων επιδέχεται τους μετασχηματισμούς στους οποίους το σύστημα μπορεί να επέμβει. Όταν το σύστημα συναντά αδιαφανή εργαλεία ή αυστηρά σειριακές δομές, η προσέγγιση της ασφαλούς υποχώρησης (conservative fallback) αναιρεί τα πιθανά κέρδη απόδοσης. Αυτός ο περιορισμός αποτελεί σημαντικό ανοικτό ζήτημα στην τρέχουσα έρευνα γύρω από το κέλυφος. Για να παρακαμφθεί ο σκόπελος των χειροκίνητων επισημειώσεων και των αδιαφανών εξωτερικών εντολών, απαιτείται μια διαφορετική προσέγγιση. Το επόμενο, και τελευταίο, κεφάλαιο της αξιολόγησης διερευνά αυτή την κατεύθυνση μέσω του συστήματος *hS*, το οποίο επιχειρεί να προσφέρει αυτόματη επιτάχυνση γενικού σκοπού αξιοποιώντας την υποθετική εκτέλεση εκτός σειράς και τη δυναμική ιχνηλάτηση.



Σχήμα 9.2: Σχετικές επιταχύνσεις του PaSh στα μετροπρογράμματα της σουίτας Koala. Το PaSh προσφέρει επιταχύνσεις σε γνωστά εργαλεία, αλλά επιστρέφει σε σειριακή εκτέλεση όταν απουσιάζουν οι επισημειώσεις ή κατάλληλες συντακτικές δομές.

Κεφάλαιο 10

Αξιολόγηση *hS*: Υποθετική Εκτέλεση Εκτός Σειράς

Στο προηγούμενο κεφάλαιο διαπιστώθηκε ότι η αυτόματη παραλληλοποίηση (μέσω του PaSh) πλεονεκτεί χάρη στη μηδενική προσπάθεια υιοθέτησης, αλλά περιορίζεται δραστικά από το «πρόβλημα των επισημειώσεων», αποτυγχάνοντας να επιταχύνει άγνωστα, αδιαφανή εργαλεία. Στο παρόν κεφάλαιο εξετάζεται μια καινούρια, εναλλακτική προσέγγιση, το *hS*, το οποίο επιχειρεί να προσφέρει αυτόματη επιτάχυνση γενικού σκοπού, παρακάμπτοντας την ανάγκη για πρόσθετη πληροφορία σχετικά με τις εντολές που εκτελούνται.

10.1 Μηχανισμός Λειτουργίας και Χαρακτηριστικά

Το *hS* εισάγει μια διαφορετική προσέγγιση αυτόματης επιτάχυνσης βασισμένη στην εκτέλεση εκτός σειράς (out-of-order execution) [14]. Αντί να ακολουθεί αυστηρά τη συντακτική σειρά του προγράμματος, επιχειρεί να εκτελεί εντολές «καιροσκοπικά», δηλαδή προβαίνοντας σε υποθετική εκτέλεση (speculative execution), προβλέποντας ότι δεν θα προκύψουν παραβιάσεις εξαρτήσεων δεδομένων. Η προσέγγιση αυτή εδράζεται στην παρατήρηση ότι πολλά προγράμματα κελύφους είναι υπερβολικά σειριακά, παρά το γεγονός ότι περιέχουν ανεξάρτητες ή ασθενώς εξαρτώμενες εντολές. Το σύστημα προεπεξεργάζεται τον κώδικα και κατασκευάζει έναν γράφο μερικής διάταξης που αποτυπώνει τη συντακτική σειρά. Στη συνέχεια, κατά την εκτέλεση, συλλέγει δυναμικά πληροφορίες για τις εξαρτήσεις μέσω ιχνηλάτησης (tracing) των προσπελάσεων στο σύστημα αρχείων και στο περιβάλλον.

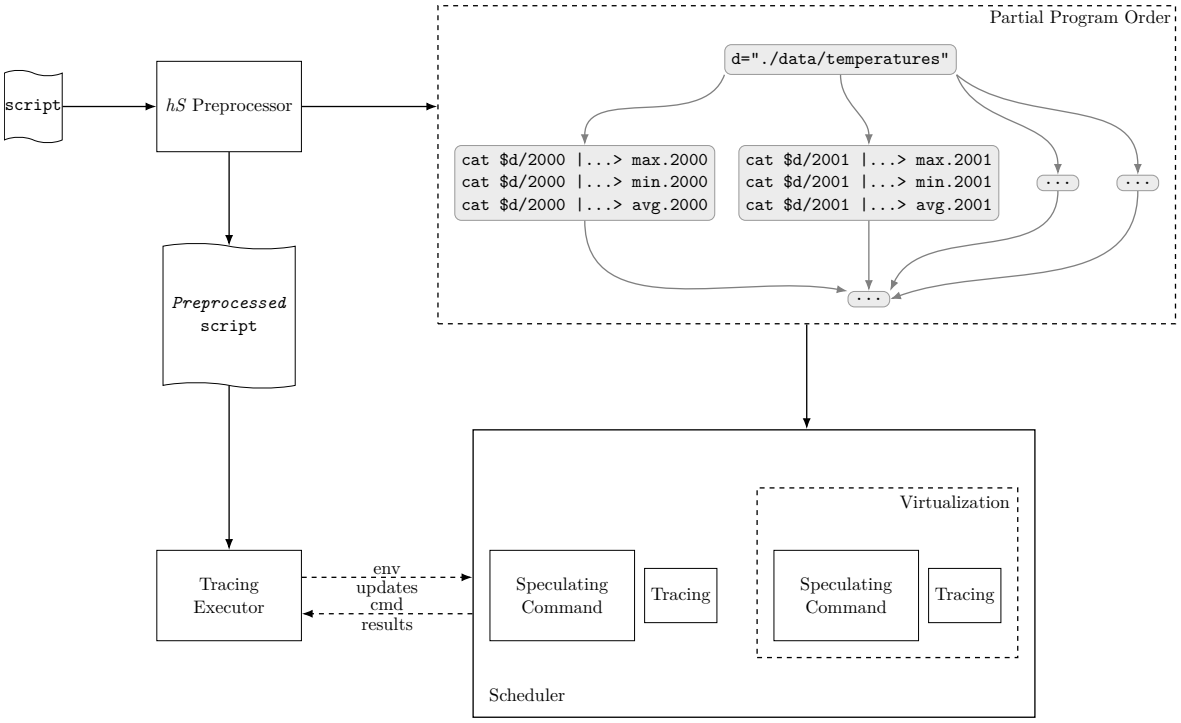
Όπως φαίνεται στο Σχήμα 10.1, το *hS* εκτελεί μελλοντικές εντολές σε απομονωμένα περιβάλλοντα (sandboxes), χρησιμοποιώντας μηχανισμούς εικονικοποίησης και επικαλυπτόμενα συστήματα αρχείων (overlay filesystems). Οι παρενέργειες των εντολών καταγράφονται προσωρινά. Όταν μια εντολή ολοκληρωθεί, τα αποτελέσματά της ενσωματώνονται στο κύριο σύστημα (commit) μόνο εφόσον επιβεβαιωθεί ότι δεν παραβιάστηκαν εξαρτήσεις από προηγούμενες εντολές. Σε περίπτωση ανίχνευσης σύγκρουσης (conflict), οι υποθετικές εκτελέσεις ακυρώνονται (rollback) και η εκτέλεση συνεχίζεται με ασφάλεια σύμφωνα με τη σωστή σημασιολογική σειρά.

Ο ενσωματωμένος χρονοπρογραμματιστής εναλλάσσει φάσεις εκτέλεσης και επικύρωσης. Η τεχνική αυτή επιτρέπει την αξιοποίηση παραλληλίας εργασιών σε αδιαφανή εκτελέσιμα χωρίς εκ των προτέρων προδιαγραφές, καθιστώντας το *hS* ένα σύστημα γενικής χρήσης.

10.2 Υποθετική Εκτέλεση: Το Σενάριο `weather`

Για την κατανόηση της πρακτικής εφαρμογής της υποθετικής εκτέλεσης, εξετάζεται η λειτουργία του χρονοδρομολογητή (scheduler) του *hS* στο σενάριο `weather` (Κώδικας 4.1). Σε αντίθεση με το PaSh, το οποίο κατασκευάζει στατικά FIFOs, το *hS* εκκινεί τις διαδοχικές σωληνώσεις (pipelines H1, H2, H3 του βρόχου) εντός sandboxes *πριν* ολοκληρωθούν οι προηγούμενες εξαρτήσεις τους.

Ο Πίνακας 10.1 αποτυπώνει αναλυτικά την εσωτερική αυτή συμπεριφορά, θεωρώντας ως έτος εκκίνησης το 2000 (`start` = 2000).



Σχήμα 10.1: Επισκόπηση του *hS*. Το σύστημα εκτελεί μελλοντικές εντολές προκαταβολικά (speculation) σε απομονωμένα περιβάλλοντα (sandboxes) και εντοπίζει τις προσπελάσεις I/O (με χρήση `ptrace`) και την κατάσταση του περιβάλλοντος. Εάν δεν υπάρξουν συγκρούσεις εξαρτήσεων (conflicts), τα αποτελέσματα ενσωματώνονται στο σύστημα αρχείων (commit), διαφορετικά η εκτέλεση αναιρείται (rollback) και επαναλαμβάνεται [14].

10.3 Προσπάθεια Υιοθέτησης

Όπως ακριβώς και το PaSh, το *hS δεν απαιτεί τροποποίηση κώδικα*. Ωστόσο, το *hS* επεκτείνει αυτή την προσέγγιση: η απουσία επισημειώσεων σημαίνει ότι ο χρήστης δεν απαιτεί γνώση της εσωτερικής λειτουργίας των εντολών που καλεί. Αυτό είναι ιδιαίτερα κρίσιμο στην περίπτωση των προσαρμοσμένων εκτελέσιμων, τα οποία συχνά αντιμετωπίζονται ως «μαύρα κουτιά». Όπως προκύπτει από την ανάλυση των προγραμμάτων, τα εκτελέσιμα αυτά συνιστούν την πλειονότητα των εντολών που εμφανίζονται στα υπό εξέταση σενάρια. Επιπλέον, σε αρκετές περιπτώσεις εντάσσονται σε εξειδικευμένα υπολογιστικά πεδία τα οποία δεν είναι απαραίτητα οικεία σε γενικούς προγραμματιστές, όπως για παράδειγμα τα εργαλεία που περιλαμβάνονται στο `bio` που ανήκουν στον τομέα της βιοπληροφορικής. Ενώ σε άλλα συστήματα ο χρήστης θα έπρεπε να αναλύσει χειροκίνητα τη συμπεριφορά αυτών των προγραμμάτων (π.χ. ποια αρχεία διαβάζουν ή τροποποιούν), το *hS* επιτρέπει την αυτόματη παράλληλη εκτέλεσή τους μέσω της υποθετικής εκτέλεσης εκτός σειράς. Το σύστημα αναλαμβάνει τη διασφάλιση της ορθότητας, προβαίνοντας αυτόματα σε αναίρεση (rollback) της εκτέλεσης μόνο εάν ανιχνευθεί δυναμικά μια πραγματική σύγκρουση εξαρτήσεων, χωρίς επιπλέον παρέμβαση από τον χρήστη.

Καθώς το *hS* βρίσκεται αυτή τη στιγμή υπό ενεργή ανάπτυξη, για την παρούσα αξιολόγηση χρησιμοποιήθηκε ένα πρώιμο πρωτότυπο (early-stage prototype) που παραχωρήθηκε από τους δημιουργούς του. Για την εκτέλεση των σεναρίων του Koala, τα μετροπρογράμματα παρέμειναν αμετάβλητα, με μοναδική τροποποίηση τον ορισμό της μεταβλητής περιβάλλοντος:

```
KOALA_SHELL="hs"
```

Ωστόσο, λόγω της πειραματικής φύσης του εργαλείου, δεν ήταν εφικτή η επιτυχής εκτέλεση ολόκληρης της σουίτας Koala. Για σύνολα όπως τα `analytics`, `bio`, `ml`, `nlp`, `unixfun`, `weather` και `web-`

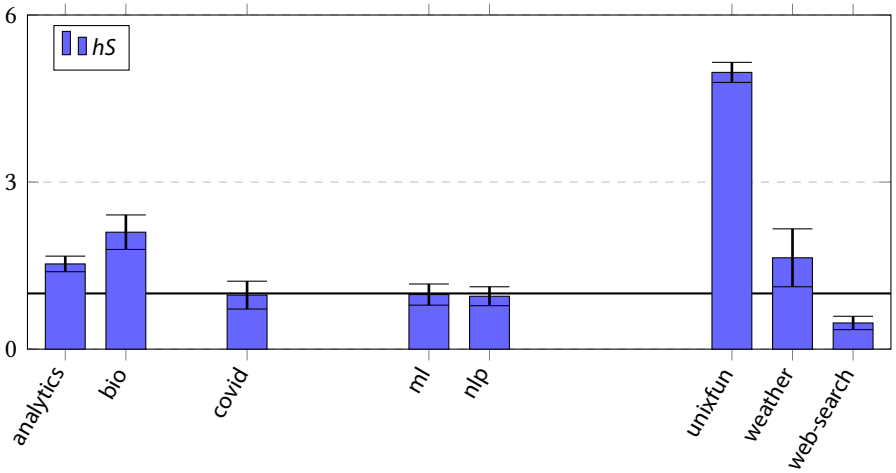
Χρονικό Βήμα	Ενέργεια Χρονοδρομολογητή *hS* στο Παρασκήνιο
t_0	Εκκίνηση `H1(2000)`: Εκτελείται κανονικά, καθώς είναι η πρώτη εντολή της σειράς.
t_1	Υποθετική εκτέλεση: Έναρξη των `H2(2000)` και `H3(2000)` σε απομονωμένα περιβάλλοντα. Οι αναγνώσεις/εγγραφές τους ιχνηλατούνται.
t_2	Το κέλυφος προχωρά στο επόμενο έτος (`y=2001`). Ξεκινά η υποθετική `H1(2001)` ενώ οι `H2`/`H3` του 2000 εκτελούνται ακόμη.
t_3	Η `H1(2000)` ολοκληρώνεται με επιτυχία.
t_4	Η `H2(2000)` ολοκληρώνεται. Το σύστημα ελέγχει τα ίχνη I/O και διαπιστώνει ότι δεν υπάρχουν συγκρούσεις με την H1 → Οριστικοποίηση (Commit) των αλλαγών της στο πραγματικό σύστημα αρχείων.
t_5	Η `H3(2000)` ολοκληρώνεται. Ομοίως δεν ανιχνεύονται συγκρούσεις → Οριστικοποίηση (Commit).
t_6	Ο βρόχος συνεχίζεται, έχοντας πρακτικά επικαλύψει (overlap) τον χρόνο εκτέλεσης των επιμέρους εντολών.

Πίνακας 10.1: Λειτουργία του χρονοδρομολογητή του *hS* στο σενάριο `weather`. Οι `H1`, `H2` και `H3` αντιστοιχούν στις τρεις ανεξάρτητες σωληνώσεις εντός του βρόχου `for`.

`search`, το *hS* εκτέλεσε επιτυχώς τουλάχιστον ένα υποσύνολο των σεναρίων. Αντιθέτως, τα υποσύνολα `ci-cd`, `file-mod`, `inference`, `oneliners`, `pkg` και `repl` αποκλείστηκαν πλήρως από την αξιολόγηση του *hS*, καθώς η εκτέλεσή τους είτε δεν ολοκληρώθηκε επιτυχώς είτε παρήγαγε λανθασμένα αποτελέσματα εξαιτίας της αδυναμίας του πρωτοτύπου να διαχειριστεί τις συγκεκριμένες δομές τους.

10.4 Ανάλυση Απόδοσης

Για την αποτίμηση του συστήματος, εστιάσαμε στα μετροπρογράμματα που εκτελέστηκαν επιτυχώς. Τα αποτελέσματα αποτυπώνονται στο Σχήμα 10.2.



Σχήμα 10.2: Σχετικές επιταχύνσεις του *hS* στα μετροπρογράμματα της σουίτας Koala. Το σύστημα προσφέρει αξιόλογες επιταχύνσεις όταν οι προβλέψεις επιβεβαιώνονται, αλλά τιμωρείται αυστηρά από το κόστος ιχνηλάτησης σε μικρές διεργασίες.

Η ανάλυση ανέδειξε ότι το *hS* προσφέρει σημαντικά οφέλη σε σενάρια που περιλαμβάνουν συν-

τακτικές περιοχές χωρίς αλληλεξαρτήσεις. Οι επιταχύνσεις κυμαίνονται από **1.53× έως 4.97×**, με τα σύνολα **`analytics`** και **`unixfun`** να βρίσκονται στα δύο άκρα αυτού του εύρους αντίστοιχα. Άλλα μετροπρογράμματα που παρουσίασαν αξιοσημείωτες επιταχύνσεις περιλαμβάνουν τα **`weather`** (**1.64×**) και **`bio`** (**2.10×**). Στα σενάρια αυτά, οι υποθετικές εντολές λειτουργούν σε ανεξάρτητα αρχεία, με αποτέλεσμα οι προβλέψεις του χρονοδρομολογητή να επιβεβαιώνονται συστηματικά (υψηλό speculation hit rate), επιτρέποντας στο σύστημα να αποκρύψει πλήρως τις καθυστερήσεις του συστήματος αρχείων.

Στον αντίποδα, σενάρια που εμπλέκουν άμεσες εξαρτήσεις μεταξύ των σταδίων δεν μπόρεσαν να επωφεληθούν από την υποθετική προσέγγιση. Σε αυτά τα σενάρια σημειώθηκαν σοβαρές επιβραδύνσεις, οι οποίες κυμάνθηκαν από **0.97×** (**στο `covid`**) έως και **0.47×** (**στο `web-search`**), διαμορφώνοντας έναν μέσο όρο επιβράδυνσης **0.72×** για αυτές τις κατηγορίες.

Αυτές οι επιβραδύνσεις μπορούν να αποδοθούν κυρίως στους εξής παράγοντες:

1. **Συχνές Ακυρώσεις (Rollbacks):** Όταν τα σενάρια χαρακτηρίζονται από ισχυρές, άμεσες εξαρτήσεις δεδομένων, η υποθετική εκτέλεση δεν αποδεικνύεται αποτελεσματική, οδηγώντας σε κοστοβόρες διαδικασίες ακύρωσης και επανεκτέλεσης.

2. **Κόστος Απομόνωσης και Ιχνηλάτησης (Isolation & Tracing Overhead):** Το *hS* βασίζεται στο `ptrace` για την παρακολούθηση των κλήσεων συστήματος και σε επικαλυπτόμενα συστήματα αρχείων για την απομόνωση. Σε φόρτους εργασίας με πολλές γρήγορες, βραχύβιες διεργασίες, το κόστος εναλλαγής περιβάλλοντος για την ιχνηλάτηση κάθε I/O κλήσης επικρατεί, υποβαθμίζοντας την απόδοση.

Τέλος, η προσέγγιση περιορίζεται από εντολές με μη εικονικοποιήσιμες παρενέργειες, όπως η δικτυακή επικοινωνία (`curl`, `wget`), οι οποίες δεν μπορούν να εκτελεστούν προκαταβολικά και να αναιρεθούν χωρίς συνέπειες.

10.5 Σύνοψη

Το *hS* υποδεικνύει ότι οι αρχές της εκτέλεσης εκτός σειράς μπορούν να εφαρμοστούν με επιτυχία στο επίπεδο του λειτουργικού συστήματος και του κελύφους. Η ικανότητά του να επιταχύνει αδιαφανή προγράμματα χωρίς καμία παρέμβαση από τον χρήστη αποτελεί σημαντική εξέλιξη προς την πλήρη αυτοματοποίηση. Ωστόσο, τα ευρήματα της σουίτας Koala υπογραμμίζουν ότι, στην τρέχουσα πειραματική του μορφή, η υψηλή υπολογιστική επιβάρυνση των μηχανισμών παρακολούθησης και οι κυρώσεις από τις λανθασμένες υποθετικές εκτελέσεις αποτελούν σημαντικά εμπόδια που χρήζουν περαιτέρω βελτιστοποίησης για γενικευμένη χρήση.

Κεφάλαιο 11

Συμπεράσματα

Η συστηματική ανάλυση μέσω της σουίτας Koala αναδεικνύει ότι η επιτάχυνση προγραμμάτων κελύφους δεν μπορεί να αντιμετωπιστεί ως ένα μονοδιάστατο πρόβλημα παραλληλισμού, αλλά ως ένας σύνθετος χώρος σχεδιαστικών επιλογών που εξαρτάται από τη φύση των φορτίων εργασίας, τη σημασιολογία των εντολών και τη δομή των ίδιων των σεναρίων. Ο συνδυασμός στατικού και δυναμικού χαρακτηρισμού των μετροπρογραμμάτων, καθώς και η εφαρμογή πολλαπλών συστημάτων επιτάχυνσης προσφέρουν μια ολοκληρωμένη εικόνα του χώρου βελτιστοποίησης του κελύφους.

11.1 Διαφορετικές Στρατηγικές Εκτέλεσης και τα Όριά τους

Η εφαρμογή των συστημάτων `zsh`, Shark, GNU `parallel`, PaSh και *hS* στη σουίτα Koala δείχνει ότι κάθε προσέγγιση είναι αποτελεσματική σε διαφορετικές κατηγορίες προγραμμάτων και αξιοποιεί διαφορετικές ευκαιρίες βελτιστοποίησης.

- **Εναλλακτικός διερμηνέας (`zsh`):** Η απλή αντικατάσταση του διερμηνέα εκτέλεσης δεν οδηγεί σε ουσιαστική μεταβολή της απόδοσης των σεναρίων, γεγονός που υποδηλώνει ότι η επιλογή διαφορετικού διερμηνέα δεν εισάγει πρόσθετη επιβάρυνση κατά την εκτέλεση προγραμμάτων κελύφους. Η αυξημένη παραμετροποιησιμότητα και τα προηγμένα διαδραστικά χαρακτηριστικά του `zsh` μπορούν να υιοθετηθούν χωρίς να επηρεάζεται αρνητικά η απόδοση των υπολογιστικών σεναρίων.

- **Συντακτικοί μετασχηματισμοί (Shark):** Το Shark επιτυγχάνει σημαντικές επιταχύνσεις σε σενάρια με ανεξάρτητες επαναλήψεις ή αγωγούς εντολών, φθάνοντας έως και 13.43× σε ορισμένες περιπτώσεις. Τα οφέλη μειώνονται όταν οι αγωγοί δεδομένων είναι ήδη βελτιστοποιημένοι ή όταν οι εξαρτήσεις μεταξύ σταδίων περιορίζουν τους διαθέσιμους μετασχηματισμούς.

- **Χειροκίνητος παραλληλισμός (GNU `parallel`):** Η χρήση του GNU `parallel` αποδίδει ιδιαίτερα καλά σε εργασίες έντονης εισόδου/εξόδου ή σε βρόχους χωρίς αλληλεξαρτήσεις, με μέσες επιταχύνσεις περίπου 2.6×. Ωστόσο, απαιτεί χειροκίνητη τροποποίηση των προγραμμάτων, γεγονός που περιορίζει τη γενικότητα της προσέγγισης.

- **Παραλληλοποίηση με γνώση εντολών (PaSh):** Το PaSh επιταχύνει αυτόματα σενάρια που περιέχουν αγωγούς εντολών, όμως η αποτελεσματικότητά του εξαρτάται από τη διαθεσιμότητα επισημειώσεων και από το κατά πόσο το πρόγραμμα εμπίπτει στο μοντέλο παραλληλοποίησης που υποστηρίζει.

- **Εκτέλεση εκτός σειράς (*hS*):** Το *hS* προσφέρει οφέλη σε περιπτώσεις με ανεξάρτητα στάδια υπολογισμού, αλλά η απόδοσή του επηρεάζεται από το κόστος απομόνωσης και ιχνηλάτησης και την ύπαρξη εξαρτήσεων, οδηγώντας σε αστάθεια ή ακόμη και επιβραδύνσεις σε ορισμένα προγράμματα.

Συνολικά, φαίνεται ότι δεν υπάρχει μία καθολικά βέλτιστη τεχνική επιτάχυνσης· κάθε σύστημα αξιοποιεί διαφορετικά χαρακτηριστικά του κελύφους και εμφανίζει διαφορετικούς περιορισμούς.

11.2 Πολυμορφία Συμπεριφοράς

Η συντακτική ανάλυση δείχνει ότι τα προγράμματα κελύφους λειτουργούν κυρίως ως μηχανισμός ενορχήστρωσης εξωτερικών εργαλείων. Παρότι οι πιο συχνές εντολές προέρχονται από POSIX εργαλεία και GNU Coreutils, το 54% των μοναδικών εντολών στα μετροπρογράμματα αντιστοιχεί σε προσαρμοσμένα εκτελέσιμα ή συναρτήσεις χρηστών. Το γεγονός αυτό επιβεβαιώνει ότι τα πραγματικά προγράμματα κελύφους λειτουργούν κυρίως ως συνδετικός μηχανισμός που ενοποιεί ετερογενή προγράμματα. Η ιδιότητα αυτή δυσκολεύει τις στατικές τεχνικές βελτιστοποίησης, καθώς η συμπεριφορά μεγάλου μέρους των εντολών παραμένει αδιαφανής για το σύστημα επιτάχυνσης [16].

Ο δυναμικός χαρακτηρισμός αποκαλύπτει ότι τα φορτία εργασίας του Koala καλύπτουν ευρύ φάσμα συμπεριφορών ως προς τον χρόνο εκτέλεσης και τη χρήση πόρων.

- Ο χρόνος χρήσης επεξεργαστή κυμαίνεται από δευτερόλεπτα έως ώρες.
- Η χρήση μνήμης και το συνολικό μέγεθος εισόδου/εξόδου διαφέρουν κατά αρκετές τάξεις μεγέθους.
- Πολλά προγράμματα κυριαρχούνται από εξωτερικές εντολές, ενώ άλλα δαπανούν σημαντικό χρόνο στον ίδιο τον διερμηνέα του κελύφους.

Η ποικιλία αυτή εξηγεί γιατί τεχνικές που εστιάζουν αποκλειστικά στον παραλληλισμό δεδομένων δεν αποδίδουν ομοιόμορφα σε όλα τα μετροπρογράμματα. Το κέλυφος χρησιμοποιείται όχι μόνο για επεξεργασία δεδομένων αλλά και για ενορχήστρωση σύνθετων εργασιών συστήματος.

11.3 Η Αξία της Συστηματικής Αξιολόγησης

Η απουσία καθιερωμένων μετροπρογραμμάτων αποτελούσε σημαντικό εμπόδιο στην έρευνα επιτάχυνσης του κελύφους. Η σουίτα Koala παρέχει:

- ρεαλιστικά φορτία εργασίας με πραγματικά δεδομένα,
- αυτοματοποιημένη εγκατάσταση και επικύρωση αποτελεσμάτων,
- αναπαραγώγιμη υποδομή αξιολόγησης,
- ευρεία κάλυψη συντακτικών και δυναμικών χαρακτηριστικών.

Η εφαρμογή της σουίτας σε διαφορετικές προσεγγίσεις επιτάχυνσης ανέδειξε τη σημασία της αξιολόγησης υπό κοινές και αναπαραγώγιμες συνθήκες, επιτρέποντας τη σαφή αποτύπωση των συμβιβασμών μεταξύ απόδοσης, πολυπλοκότητας υλοποίησης και προσπάθειας υιοθέτησης. Επομένως, η ύπαρξη μιας συστηματικής και κοινά αποδεκτής σουίτας είναι απαραίτητη για δίκαιες συγκρίσεις και αξιόπιστη πρόοδο στο πεδίο.

11.4 Συνολική Αποτίμηση

Η ανάλυση μέσω της σουίτας καταδεικνύει ότι το κέλυφος Unix αποτελεί ένα ιδιαίτερα ετερογενές περιβάλλον εκτέλεσης. Τα διαφορετικά φορτία εργασίας, οι εξωτερικές εντολές και η ποικιλία δυναμικών συμπεριφορών καθιστούν δύσκολη την ύπαρξη μιας ενιαίας στρατηγικής επιτάχυνσης. Η συμβολή του Koala έγκειται στο ότι επιτρέπει, για πρώτη φορά, συγκρίσιμη και συστηματική αξιολόγηση συστημάτων επιτάχυνσης σε ρεαλιστικά σενάρια, αναδεικνύοντας τόσο τις δυνατότητες όσο και τα όριά τους.

Κεφάλαιο 12

Μελλοντικό Έργο και Συμπεράσματα

Η παρούσα διπλωματική ολοκληρώνεται έχοντας παρουσιάσει, αναλύσει και αξιοποιήσει τη σουίτα Koala. Μέσα από τη συστηματική μελέτη των δομικών χαρακτηριστικών του κελύφους Unix και τον εντοπισμό των πρακτικών περιορισμών των υφιστάμενων επιταχυντών μπορεί κανείς να κατανοήσει τι καθιστά τη βελτιστοποίηση του κελύφους ένα τόσο σύνθετο πρόβλημα με σημαντικές τεχνικές προκλήσεις. Σε αυτό το τελικό κεφάλαιο, σκιαγραφούνται οι ερευνητικές κατευθύνσεις που ανοίγονται για το μέλλον και συνοψίζονται τα τελικά συμπεράσματα της εργασίας.

12.1 Μελλοντικό Έργο

Η υποδομή και τα ευρήματα της παρούσας εργασίας ανοίγουν πολλαπλές κατευθύνσεις για τη μελλοντική έρευνα.

Επέκταση της Σουίτας με Νέα Μοτίβα Εργασιών

Παρότι η σουίτα Koala καλύπτει ήδη ένα ευρύ φάσμα εφαρμογών, νέες κατηγορίες σεναρίων μπορούν να ενισχύσουν περαιτέρω την αντιπροσωπευτικότητά της:

- **Σενάρια έντονης δικτυακής αλληλεπίδρασης:** Σενάρια που βασίζονται σε συνεχή επικοινωνία με απομακρυσμένες υπηρεσίες ή σε συγχρονισμό δεδομένων θα επέτρεπαν τη μελέτη συστημάτων όπου η καθυστέρηση δικτύου (network latency) κυριαρχεί έναντι του τοπικού I/O.
- **Μη ντετερμινιστικές εκτελέσεις:** Σενάρια που εξαρτώνται από τυχαιότητα ή από τη δυναμική κατάσταση του συστήματος μπορούν να αναδείξουν τους περιορισμούς τεχνικών παράλληλης εκτέλεσης και εκτέλεσης εκτός σειράς, οι οποίες βασίζονται σε ακριβείς προβλέψεις.
- **Διαδραστικές συνεδρίες:** Η μοντελοποίηση της πραγματικής, διαδραστικής χρήσης του κελύφους θα επιτρέψει τη μελέτη του χρόνου απόκρισης (latency) και της συνολικής εμπειρίας χρήστη (QoE).

Αξιολόγηση Νέων Αρχιτεκτονικών Συστημάτων

Η ποικιλία των μετροπρογραμμάτων καθιστά το Koala ιδανικό για την αξιολόγηση νέων κατηγοριών συστημάτων, πέρα από την τοπική επιτάχυνση σε έναν κόμβο:

- **Κατανεμημένα shells:** Συστήματα όπως το DiSh [11] και το POSH [1] επιτρέπουν την εκτέλεση προγραμμάτων κελύφους σε πολλαπλούς κόμβους (clusters). Μετροπρογράμματα με μεγάλο όγκο δεδομένων προσφέρουν το φυσικό πεδίο αξιολόγησης του κόστους έναντι των οφελών της δικτυακής κατανομής.
- **Ανοχή σφαλμάτων και συνέχιση εκτέλεσης:** Συστήματα όπως το Fractal [12] εισάγουν μηχανισμούς επαναφοράς (checkpointing) και συνέχειας. Μακρόχρονα μετροπρογράμματα της σουίτας μπορούν να χρησιμοποιηθούν για τη μέτρηση της πραγματικής επιβάρυνσης αυτών των τεχνικών ανοχής σφαλμάτων.

Σχεδιασμός Επόμενης Γενιάς Βελτιστοποιητών

Τα ευρήματα από την εφαρμογή των συστημάτων υποδεικνύουν βασικές κατευθύνσεις για τους βελτιστοποιητές της επόμενης γενιάς:

- **Προηγμένη στατική ανάλυση:** Προσεγγίσεις όπως αυτή που παρουσιάζεται στο [100] στοχεύουν στην καλύτερη κατανόηση της σημασιολογίας των σεναρίων κελύφους, γεγονός που μπορεί να μειώσει την ανάγκη για επισημειώσεις σε δυναμικά συστήματα όπως το PaSh.
- **Μείωση κόστους παρακολούθησης:** Η εμπειρία από το *hS* υποδεικνύει την ανάγκη για μετακίνηση της ιχνηλάτησης (tracing) από το επίπεδο χρήστη (π.χ. FUSE / ptrace) απευθείας στο επίπεδο του πυρήνα (π.χ. μέσω eBPF), μειώνοντας την υπολογιστική επιβάρυνση (overhead) κατά την καιροσκοπική εκτέλεση.
- **Βελτιστοποίηση εισόδου/εξόδου και χώρου:** Η αποδοτική διαχείριση της ροής δεδομένων, η ελαχιστοποίηση των προσωρινών αρχείων και η βελτιστοποίηση χρήσης μνήμης είναι εξίσου σημαντικές με τον παραλληλισμό του υπολογιστικού φόρτου.

12.2 Τελικά Συμπεράσματα

Η παρούσα διπλωματική εργασία κατέδειξε ότι το κέλυφος αποτελεί ένα πλούσιο, ετερογενές και πολύπλοκο υπολογιστικό περιβάλλον. Η συστηματική ανάλυση και αξιολόγηση μέσω του Koala επέτρεψε τη χαρτογράφηση των πραγματικών χαρακτηριστικών των σεναρίων κελύφους, τη δίκαιη σύγκριση διαφορετικών συστημάτων επιτάχυνσης και την ανάδειξη των σχεδιαστικών συμβιβασμών που απαιτούνται.

Συνοψίζοντας, τα τρία κομβικά συμπεράσματα αυτής της διπλωματικής εργασίας είναι τα εξής:

1. **Δεν υπάρχει «χρυσή τομή» (silver bullet):** Η ποικιλία στη δυναμική συμπεριφορά, από σενάρια δευτερολέπτων έως φορτία ωρών και από ελάχιστη έως τεράστια χρήση μνήμης και λειτουργιών εισόδου/εξόδου, αποδεικνύει ότι καμία μεμονωμένη στρατηγική επιτάχυνσης δεν μπορεί να καλύψει όλες τις ανάγκες.
2. **Το πρόβλημα των «μαύρων κουτιών»:** Το κέλυφος λειτουργεί κυρίως ως γλώσσα ενορχήστρωσης. Η εκτεταμένη εξάρτηση από προσαρμοσμένα, εξωτερικά εκτελέσιμα καθιστά τις τεχνικές που βασίζονται αυστηρά σε προϋπάρχουσες επισημειώσεις μη πρακτικές για γενικευμένη χρήση, υπογραμμίζοντας την ανάγκη για δυναμική ανάλυση.
3. **Ο συμβιβασμός (trade-off) μεταξύ προσπάθειας υιοθέτησης και τελικής απόδοσης:** Ενώ η χειροκίνητη παραλληλοποίηση μπορεί θεωρητικά να προσεγγίσει τα υψηλά επίπεδα απόδοσης, επιβάλλει ένα σημαντικό προγραμματιστικό κόστος. Στον αντίποδα, η αυτοματοποιημένη βελτιστοποίηση (όπως η μεταγλώττιση JIT του PaSh και η εκτέλεση εκτός σειράς του *hS*) παρέχει μια πιο πρακτική εναλλακτική, προσφέροντας σημαντικές επιταχύνσεις με ελάχιστη έως μηδενική παρέμβαση στον πηγαίο κώδικα. Ωστόσο, η προσέγγιση αυτή συνοδεύεται από νέες προκλήσεις, καθώς τα συστήματα καλούνται να αντισταθμίσουν το εγγενές κόστος της δυναμικής ενορχήστρωσης απέναντι στα οφέλη του παραλληλισμού. Το γεγονός αυτό αναδεικνύει τη διαφανή, αυτόματη επιτάχυνση αδιαφανών εντολών ως ένα σημαντικό ερευνητικό ζήτημα για το κέλυφος.

Τελικά, η πρόοδος στην έρευνα για το κέλυφος εξαρτάται άμεσα από την ύπαρξη κοινών, ρεαλιστικών και αναπαραγώγιμων πλαισίων. Το Koala διατίθεται ως εργαλείο ανοικτού κώδικα, ενώ η μόνιμη αρχειοθέτηση των δεδομένων εισόδου και των εξαρτήσεων διασφαλίζει τη μακροπρόθεσμη αναπαραγωγιμότητα των αποτελεσμάτων. Μέσα από αυτά τα χαρακτηριστικά, η σουίτα Koala φιλοδοξεί να αποτελέσει το απόλυτο σημείο αναφοράς, συμβάλλοντας στη διαμόρφωση ενός σταθερού πλαισίου για πιο συστηματική, δίκαιη και συγκρίσιμη μελλοντική έρευνα στον τομέα.

Βιβλιογραφία

[1] Deepti Raghavan κ.ά. "POSH: a data-aware shell". Στο: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4. DOI: 10.5555/3489146.3489188. URL: https://dl.acm.org/doi/10.5555/3489146.3489188.

[2] Aurèle Mahéo, Pierre Sutra και Tristan Tarrant. "The serverless shell". Στο: *Proceedings of the 22nd International Middleware Conference: Industrial Track*. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, σσ. 9–15. ISBN: 9781450391528. DOI: 10.1145/3491084.3491426. URL: https://doi.org/10.1145/3491084.3491426.

[3] Jiasi Shen, Martin Rinard και Nikos Vasilakis. "Automatic synthesis of parallel unix commands and pipelines with KumQuat". Στο: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, σσ. 431–432. ISBN: 9781450392044. DOI: 10.1145/3503221.3508400. URL: https://doi.org/10.1145/3503221.3508400.

[4] Diomidis Spinellis και Marios Fragkoulis. "Extending Unix Pipelines to DAGs". Στο: *IEEE Transactions on Computers* 66.9 (2017), σσ. 1547–1561. DOI: 10.1109/TC.2017.2695447. URL: https://ieeexplore.ieee.org/document/7903579.

[5] Nikos Vasilakis κ.ά. "PaSh: Light-Touch Data-Parallel Shell Processing". Στο: *16th European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, σσ. 49–66. ISBN: 9781450383349. DOI: 10.1145/3447786.3456228. URL: https://doi.org/10.1145/3447786.3456228.

[6] Shivam Handa κ.ά. "An order-aware dataflow model for parallel Unix pipelines". Στο: *International Conference on Functional Programming*. Τόμ. 5. New York, NY, USA: Association for Computing Machinery, Αύγ. 2021. DOI: 10.1145/3473570. URL: https://doi.org/10.1145/3473570.

[7] Konstantinos Kallas κ.ά. "Practically Correct, Just-in-Time Shell Script Parallelization". Στο: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Ιούλ. 2022, σσ. 769–785. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/kallas.

[8] Michael Greenberg και Austin J. Blatt. "Executable formal semantics for the POSIX shell". Στο: *Proceedings of the ACM on Programming Languages* 4.POPL (Δεκ. 2019). DOI: 10.1145/3371111. URL: https://doi.org/10.1145/3371111.

[9] Michael Greenberg, Konstantinos Kallas και Nikos Vasilakis. "The Future of the Shell: Unix and Beyond". Στο: *Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, σσ. 240–241. ISBN: 9781450384384. DOI: 10.1145/3458336.3465296. URL: https://doi.org/10.1145/3458336.3465296.

[10] Charlie Curtsinger και Daniel W. Barowy. "Riker: Always-Correct and Fast Incremental Builds from Simple Specifications". Στο: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Ιούλ. 2022, σσ. 885–898. ISBN: 978-1-939133-29-70. URL: https://www.usenix.org/conference/atc22/presentation/curtsinger.

[11] Tammam Mustafa κ.ά. "DiSh: Dynamic Shell-Script Distribution". Στο: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Απρ. 2023, σσ. 341–356. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/mustafa.

[12] Zhicheng Huang κ.ά. "Fractal: Fault-Tolerant Shell-Script Distribution". Στο: *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. Renton, WA: USENIX Association, Μάι. 2026.

[13] Aurèle Mahéo, Pierre Sutra και Tristan Tarrant. "The serverless shell". Στο: *Proceedings of the 22nd International Middleware Conference: Industrial Track*. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, σσ. 9–15. ISBN: 9781450391528. DOI: 10.1145/3491084.3491426. URL: https://doi.org/10.1145/3491084.3491426.

[14] Georgios Liargkovas κ.ά. "Executing Shell Scripts in the Wrong Order, Correctly". Στο: *19th Workshop on Hot Topics in Operating Systems*. HotOS '23. Providence, RI, USA: Association for Computing Machinery, 2023, σσ. 103–109. ISBN: 9798400701955. DOI: 10.1145/3593856.3595891. URL: https://doi.org/10.1145/3593856.3595891.

[15] Emery D. Berger. *Optimizing Shell Scripting Languages*. Αδημοσίευτη ερευνητική εργασία UMCS TR-2003-009. University of Massachusetts Amherst, 2003.

[16] Evangelos Lamprou κ.ά. "The Koala Benchmarks for the Shell: Characterization and Implications". Στο: *2025 USENIX Annual Technical Conference (USENIX ATC '25)*. Boston, MA: USENIX Association, Ιούλ. 2025, σσ. 449–64. ISBN: 978-1-939133-48-9. URL: https://www.usenix.org/conference/atc25/presentation/lamprou.

[17] Paul Falstad. *Z shell (zsh)*. 2022. URL: https://zsh.sourceforge.io/.

[18] Ole Tange. "GNU Parallel - The Command-Line Power Tool". Στο: *;login: The USENIX Magazine* 36.1 (Φεβ. 2011), σσ. 42–47. URL: https://doi.org/10.5281/zenodo.16303.

[19] Dennis M. Ritchie και Ken Thompson. "The UNIX time-sharing system". Στο: *CACM* 17.7 (Ιούλ. 1974), σσ. 365–375. ISSN: 0001-0782. DOI: 10.1145/361011.361061. URL: https://doi.org/10.1145/361011.361061.

[20] S. Greenberg και I.H. Witten. "Directing the User Interface: How People Use Command-Based Computer Systems". Στο: *IFAC Proceedings Volumes* 21.5 (1988). 3rd IFAC Conference on Analysis, Design and Evaluation of Man-Machine Systems 1988, Oulu, Finland, 14-16 June 1988, σσ. 349–355. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)53932-4. URL: https://www.sciencedirect.com/science/article/pii/S1474667017539324.

[21] Michael Greenberg, Konstantinos Kallas και Nikos Vasilakis. "Unix Shell Programming: The Next 50 Years". Στο: *Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, σσ. 104–111. ISBN: 9781450384384. DOI: 10.1145/3458336.3465294. URL: https://doi.org/10.1145/3458336.3465294.

[22] Github Inc. *The top programming languages*. 2022.

[23] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". Στο: *Linux journal* 2014.239 (2014), σ. 2.

[24] Νικόλαος Παγώνας. "SPLaSh: Κλιμάκωση Προγραμμάτων Κελύφους σε Πλατφόρμες Χωρίς Διακομιστή". Diploma Thesis. Athens, Greece: National Technical University of Athens, School of Electrical και Computer Engineering, Ιούν. 2024. URL: http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/19144.

[25] Michael Greenberg. *The POSIX Shell Is an Interactive DSL for Concurrency*. URL: https://cs.pomona.edu/~michael/papers/dsldi2018.pdf.

[26] Michael Greenberg. "Word expansion supports POSIX shell interactivity". Στο: *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*. Programming '18. Nice, France: Association for Computing Machinery, 2018, σσ. 153–160. ISBN: 9781450355131. DOI: 10.1145/3191697.3214336. URL: https://doi.org/10.1145/3191697.3214336.

[27] Valve Corporation. *Steam for Linux: Critical file deletion bug due to unset environment variables*. GitHub Issue Tracker. GitHub Issue #3671. 2015.

[28] Anna Herlihy, Periklis Chrysogelos και Anastasia Ailamaki. "Boosting Efficiency of External Pipelines by Blurring Application Boundaries". Στο: *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL: https://www.cidrdb.org/cidr2022/papers/p81-herlihy.pdf.

[29] Keith Winstein και Hari Balakrishnan. "Mosh: an interactive remote shell for mobile clients". Στο: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, σ. 15. DOI: 10.5555/2342821.2342836. URL: https://dl.acm.org/doi/10.5555/2342821.2342836.

[30] Alexander J. Gaidis, Vaggelis Atlidakis και Vasileios P. Kemerlis. "SysXCHG: Refining Privilege with Adaptive System Call Filters". Στο: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, σσ. 1964–1978. ISBN: 9798400700507. DOI: 10.1145/3576915.3623137. URL: https://doi.org/10.1145/3576915.3623137.

[31] Cloyce D Spradling. "SPEC CPU2006 benchmark tools". Στο: *ACM SIGARCH Computer Architecture News* 35.1 (2007), σσ. 130–134.

[32] Meikel Poess και Chris Floyd. "New TPC benchmarks for decision support and web commerce". Στο: *SIGMOD Rec.* 29.4 (Δεκ. 2000), σσ. 64–71. ISSN: 0163-5808. DOI: 10.1145/369275.369291. URL: https://doi.org/10.1145/369275.369291.

[33] Christian Bienia κ.ά. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". Στο: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. New York, NY, USA: Association for Computing Machinery, 2008, σσ. 72–81. DOI: 10.1145/1454115.1454128. URL: https://dl.acm.org/doi/10.1145/1454115.1454128.

[34] Michael Ferdman κ.ά. "Clearing the clouds: a study of emerging scale-out workloads on modern hardware". Στο: *SIGARCH Comput. Archit. News* 40.1 (Μαρ. 2012), σσ. 37–48. ISSN: 0163-5964. DOI: 10.1145/2189750.2150982. URL: https://doi.org/10.1145/2189750.2150982.

[35] Stephen M. Blackburn κ.ά. "The DaCapo benchmarks: java benchmarking development and analysis". Στο: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, σσ. 169–190. ISBN: 1595933484. DOI: 10.1145/1167473.1167488. URL: https://doi.org/10.1145/1167473.1167488.

[36] Philipp Lengauer κ.ά. "A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008". Στο: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. L'Aquila, Italy: Association for Computing Machinery, 2017, σσ. 3–14. ISBN: 9781450344043. DOI: 10.1145/3030207.3030211. URL: https://doi.org/10.1145/3030207.3030211.

[37] Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. The MIT Press, Αύγ. 1985. ISBN: 9780262256193. DOI: 10.7551/mitpress/5298.001.0001. URL: https://doi.org/10.7551/mitpress/5298.001.0001.

[38] Urs Hölzle και David Ungar. "Do Object-Oriented Languages Need Special Hardware Support?" Στο: *Proceedings of the 9th European Conference on Object-Oriented Programming*. ECOOP '95. Berlin, Heidelberg: Springer-Verlag, 1995, σσ. 283–302. ISBN: 3540601600. DOI: 10.5555/646153.679532. URL: https://dl.acm.org/doi/10.5555/646153.679532.

[39] Gregory Cohen κ.ά. "EMNIST: Extending MNIST to handwritten letters". Στο: *2017 International Joint Conference on Neural Networks (IJCNN)*. 2017, σσ. 2921–2926. DOI: 10.1109/IJCNN.2017.7966217. URL: https://ieeexplore.ieee.org/document/7966217.

[40] René Just, Darioush Jalali και Michael D. Ernst. "Defects4J: a database of existing faults to enable controlled testing studies for Java programs". Στο: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, σσ. 437–440. ISBN: 9781450326452. DOI: 10.1145/2610384.2628055. URL: https://doi.org/10.1145/2610384.2628055.

[41] Ahmad Hazimeh, Adrian Herrera και Mathias Payer. "Magma: A Ground-Truth Fuzzing Benchmark". Στο: *Proc. ACM Meas. Anal. Comput. Syst.* 4.3 (Νοέ. 2020). DOI: 10.1145/3428334. URL: https://doi.org/10.1145/3428334.

[42] Yu Gan κ.ά. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". Στο: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, σσ. 3–18. ISBN: 9781450362405. DOI: 10.1145/3297858.3304013. URL: https://doi.org/10.1145/3297858.3304013.

[43] Lieven Eeckhout, Andy Georges και Koen De Bosschere. "How java programs interact with virtual machines at the microarchitectural level". Στο: *SIGPLAN Not.* 38.11 (Οκτ. 2003), σσ. 169–186. ISSN: 0362-1340. DOI: 10.1145/949343.949321. URL: https://doi.org/10.1145/949343.949321.

[44] Matthias Hauswirth κ.ά. "Automating vertical profiling". Στο: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: Association for Computing Machinery, 2005, σσ. 281–296. ISBN: 1595930310. DOI: 10.1145/1094811.1094834. URL: https://doi.org/10.1145/1094811.1094834.

[45] Koichi Nakashima. *ShellBench: A POSIX Shell Benchmark Suite*. Accessed: 2025-04-28. 2021.

[46] Roman Perepelitsa και Contributors. *zsh-bench: Benchmark for Interactive Zsh*. Accessed: 2025-04-28. 2021.

[47] Andy Chu και Contributors. *Oils Benchmarks*. Accessed: 2025-04-28. 2021.

[48] Kirk D. Lucas και Contributors. *UnixBench: The BYTE UNIX Benchmark Suite*. Accessed: 2025-04-28. 2012.

[49] The Open Group. *VSCPCTS 2016 Test Suite*. Accessed: 2025-01-01.

[50] Martijn Dekker. *Modernish*. Accessed: 2025-06-02. 2016.

[51] Koichi Nakashima και contributors. *ShellSpec - A Full-featured BDD Framework for Shell Scripts*. Accessed: 2025-01-01.

[52] Kate Ward. *shUnit2 - xUnit unit testing framework for Bourne based shell scripts*. Accessed: 2025-01-01.

[53] Bats-Core Project. *Bats-Core - Bash Automated Testing System*. Accessed: 2025-01-01.

[54] The Open Group. *The Test Environment Toolkit*. Accessed: 2025-01-01.

[55] Michael Schröder και Jürgen Cito. "An empirical investigation of command-line customization". Στο: *Empirical Software Engineering* 27.2 (14 Δεκ. 2021), σ. 30. DOI: 10.1007/s10664-021-10036-y. URL: https://doi.org/10.1007/s10664-021-10036-y.

[56] Yiwen Dong κ.ά. “Bash in the Wild: Language Usage, Code Smells, and Bugs ”. Στο: *ACM Transactions on Software Engineering and Methodology* 32.1 (31 Ιαν. 2023), σσ. 1–22. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3517193. URL: https://dl.acm.org/doi/10.1145/3517193 (επίσκεψη 19/06/2024).

[57] *National Oceanic and Atmospheric Administration (NOAA)*. Accessed: 2025-01-13.

[58] Tom White. *Hadoop: The Definitive Guide*. O’Reilly Media, Inc, 2009. ISBN: 9780596521974.

[59] Stéphane Peter. *makeself - Make self-extractable archives on Unix*. Accessed: 2025-01-13.

[60] Armando Cerna. *Pacaur building script*. Accessed: 2025-01-13.

[61] Kenneth Ward Church. *Unix for Poets*. 1994. URL: https://web.stanford.edu/class/cs124/kwc-unix-for-poets.pdf.

[62] Pawan Bhandari. *Solutions to unixgame.io*. Accessed: 2020-04-14. 2020.

[63] Dave Taylor και Brandon Perry. *Wicked Cool Shell Scripts: 101 Scripts for Linux, OS X, and UNIX Systems*. No Starch Press, 2016. ISBN: 978-1-59327-602-7.

[64] Evangelos Lamprou. “Foundation Models and Unix”. Στο: *Paged Out!* 6 (Μαρ. 2025), σ. 9.

[65] Alexander Kirillov κ.ά. “Segment Anything”. Στο: *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2023, σσ. 3992–4003. DOI: 10.1109/ICCV51070.2023.00371.

[66] Eleftheria Tsaliki και Diomedes Spinellis. *The Real Numbers for Athens Buses*. 2020. URL: https://insidestory.gr/article/noymera-leoforeia-athinas.

[67] Adam Drake. *Command-line Tools Can Be 235x Faster Than Your Hadoop Cluster*. Accessed: 2025-06-01. 2014.

[68] Enrico Cappellini, Frido Welker και *et al.* Pandolfi. “Early Pleistocene enamel proteome from Dmanisi resolves Stephanorhinus phylogeny”. Στο: *Nature* 574.7776 (Οκτ. 2019), σσ. 103–107. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1555-y. URL: https://www.nature.com/articles/s41586-019-1555-y.

[69] Jon Puritz. *Bio594: Using genomic techniques to examine the evolution of populations*. 2019.

[70] Fadhl Ibrahim κ.ά. “TERA-Seq: true end-to-end sequencing of native RNA molecules for transcriptome characterization”. Στο: *Nucleic Acids Research* 49.20 (2021), e115. DOI: 10.1093/nar/gkab713.

[71] Justine Tunney. *Bash One-Liners for LLMs*. Accessed: 2025-06-01. 2023.

[72] Fabian Pedregosa κ.ά. “Scikit-learn: Machine Learning in Python”. Στο: *J. Mach. Learn. Res.* 12.null (Νοέ. 2011), σσ. 2825–2830. ISSN: 1532-4435. DOI: 10.5555/1953048.2078195. URL: https://dl.acm.org/doi/10.5555/1953048.2078195.

[73] Jon Bentley. “Programming pearls: a spelling checker”. Στο: *CACM* 28.5 (Μάι. 1985), σσ. 456–462. ISSN: 0001-0782. DOI: 10.1145/3532.315102. URL: https://doi.org/10.1145/3532.315102.

[74] Jon Bentley, Don Knuth και Doug McIlroy. “Programming pearls: a literate program”. Στο: *CACM* 29.6 (Ιούν. 1986), σσ. 471–483. ISSN: 0001-0782. DOI: 10.1145/5948.315654. URL: https://doi.org/10.1145/5948.315654.

[75] Marek Majkowski. *When Bloom filters don’t bloom*. Accessed: 2025-01-13. Μαρ. 2020.

[76] Nikos Vasilakis κ.ά. “Preventing Dynamic Library Compromise on node via RWX-Based Privilege Reduction”. Στο: *ACM Conference on Computer and Communications Security (CCS)*. 2021, σσ. 1821–1838.

[77] Abebe Israel. *VPS Audit*. Accessed: 2025-01-13.

[78] Brown University Department of Computer Science. *CSCI 1380: Distributed Computer Systems*. Accessed: 2025-06-04. 2025.

[79] Zakir Durumeric, Eric Wustrow και J. Alex Halderman. "ZMap: fast internet-wide scanning and its security applications". Στο: *Proceedings of the 22nd USENIX Conference on Security*. SEC'13. Washington, D.C.: USENIX Association, 2013, σσ. 605–620. ISBN: 9781931971034. DOI: 10.5555/2534766.2534818.

[80] U.S. Securities και Exchange Commission. *EDGAR Log File Data Sets*. Accessed June 3, 2025. 2024.

[81] Sadjad Fouladi κ.ά. "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers". Στο: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Ιούλ. 2019, σσ. 475–488. ISBN: 978-1-939133-03-8. URL: http://www.usenix.org/conference/atc19/presentation/fouladi.

[82] Elias Dabbas. *Web Server Access Logs*. Accessed June 3, 2025. 2020.

[83] The SAM/BAM Format Specification Working Group. *Sequence Alignment/Map Format Specification*. Version 1.6, last modified on 6 Nov 2024. Accessed: 2025-01-13.

[84] Netresec. *Publicly available PCAP files*. Accessed: 2025-06-02. 2025.

[85] Gemma Team. "Gemma 3 Technical Report". Στο: (2025). arXiv: 2503.19786 [cs.CL]. URL: https://arxiv.org/abs/2503.19786.

[86] Tianqi Zhao κ.ά. "CLAP: Contrastive Language-Audio Pre-training Model for Multi-modal Sentiment Analysis". Στο: *Proceedings of the 2023 ACM International Conference on Multimedia Retrieval*. ICMR '23. Thessaloniki, Greece: Association for Computing Machinery, 2023, σσ. 622–626. ISBN: 9798400701788. DOI: 10.1145/3591106.3592296. URL: https://doi.org/10.1145/3591106.3592296.

[87] *Ollama: Run large language models locally*. Accessed: 2025-06-02. 2023.

[88] Adam Paszke κ.ά. "PyTorch: an imperative style, high-performance deep learning library". Στο: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[89] Simon Willison. *LLM: A CLI tool and Python library for interacting with Large Language Models*. Accessed: 2025-06-02.

[90] Michael S. Hart και Project Gutenberg. *Project Gutenberg*. 1971.

[91] *Arch User Repository (AUR)*. Accessed: 2025-01-13.

[92] *npm.js*. Accessed: 2025-01-13.

[93] Edward Tufte. *New York City Weather Chart*. Accessed: 2025-06-02. 2004.

[94] libdash developers. *libdash*.

[95] Hervé Abdi και Lynne J Williams. "Principal component analysis". Στο: *Wiley interdisciplinary reviews: computational statistics* 2.4 (2010), σσ. 433–459. DOI: 10.1002/wics.101. URL: https://wires.onlinelibrary.wiley.com/doi/10.1002/wics.101.

[96] OpenAI. *OpenAI API: Embeddings Guide*. Accessed: 2025-06-02. 2024. URL: https://platform.openai.com/docs/guides/embeddings.

[97] Alina Petukhova, João P. Matos-Carvalho και Nuno Fachada. "Text clustering with large language model embeddings". Στο: *International Journal of Cognitive Computing in Engineering* 6 (2025), σσ. 100–108. ISSN: 2666-3074. DOI: https://doi.org/10.1016/j.ijcce.2024.11.004. URL: https://www.sciencedirect.com/science/article/pii/S2666307424000482.

[98] Saiteja Utpala, Alex Gu και Pin-Yu Chen. "Language Agnostic Code Embeddings". Στο: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. Επιμέλεια υπό Kevin Duh, Helena Gomez και Steven Bethard. Mexico City, Mexico: Association for Computational Linguistics, Ιούν. 2024, σσ. 678–691. DOI: 10.18653/v1/2024.naacl-long.38. URL: https://aclanthology.org/2024.naacl-long.38.

[99] Vidar Holen κ.ά. *ShellCheck - A shell script static analysis tool*. Accessed: 2025-10-14. 2012. URL: https://www.shellcheck.net/.

[100] Lukas Lazarek κ.ά. "From Ahead-of- to Just-in-Time and Back Again: Static Analysis for Unix Shell Programs". Στο: *2025 Workshop on Hot Topics in Operating Systems*. HotOS '25. Banff, AB, Canada: Association for Computing Machinery, 2025, σσ. 88–95. ISBN: 9798400714757. DOI: 10.1145/3713082.3730395. URL: https://doi.org/10.1145/3713082.3730395.

Παράρτημα Α

Παραδείγματα Σεναρίων Υποδομής για το σύνολο `weather`

```
#!/bin/bash

sudo apt-get update
sudo apt-get install -y --no-install-recommends curl \
    wget \
    unzip \
    coreutils \
    gzip \
    gawk \
    sed \
    findutils \
    git \
    python3 \
    python3-pip \
    python3-venv

pip install --break-system-packages --upgrade pip
pip install --break-system-packages \
    numpy \
    matplotlib
```

Κώδικας Α.1: Παράδειγμα σεναρίου `install.sh` για το σύνολο `weather`.

```
#!/bin/bash

TOP=$(git rev-parse --show-toplevel)

eval_dir="${TOP}/weather"
input_dir="${eval_dir}/inputs"

URL='https://atlas.cs.brown.edu/data'
URL=$URL/max-temp
FROM=2000
TO=2015

n_samples=99999
suffix="full"
```

```
15
16  mkdir -p "${input_dir}"
17  size=full
18  for arg in "$@"; do
19      case "$arg" in
20      --small) size=small ;;
21      --min) size=min ;;
22      esac
23  done
24  if [[ "$size" == "min" ]]; then
25      if [[ -f "$input_dir/temperatures.min.txt" && -f
        ↪  "$input_dir/tuft_weather.$size.txt" ]]; then
26          echo "Data already downloaded and extracted."
27          exit 0
28      fi
29      min_inputs="$eval_dir/min_inputs/"
30      mkdir -p "$input_dir"
31      cp -r "$min_inputs"/* "$input_dir/"
32      python3 $eval_dir/scripts/generate_input.py $input_dir/tuft_weather.$size.txt
        ↪  --size $size
33      exit 0
34  fi
35  if [[ "$size" == "small" ]]; then
36      if [[ -f "$input_dir/temperatures.small.txt" && -f
        ↪  "$input_dir/tuft_weather.$size.txt" ]]; then
37          echo "Data already downloaded and extracted."
38          exit 0
39      fi
40      data_url="${URL}/temperatures.small.tar.gz"
41      wget --no-check-certificate "$data_url" -O
        ↪  "$input_dir/temperatures.small.tar.gz" || {
42          echo "Failed to download $data_url"
43          exit 1
44      }
45      tar -xzf "$input_dir/temperatures.small.tar.gz" -C "$input_dir" --no-same-owner
        ↪  || {
46          echo "Failed to extract $input_dir/temperatures.small.tar.gz"
47          exit 1
48      }
49      rm "$input_dir/temperatures.small.tar.gz"
50      mv "$input_dir/inputs/temperatures.small.txt"
        ↪  "$input_dir/temperatures.small.txt"
51      rm -rf "$input_dir/inputs"
52      python3 $eval_dir/scripts/generate_input.py $input_dir/tuft_weather.$size.txt
        ↪  --size $size
53      exit 0
54  fi
55  if [[ -f "$input_dir/temperatures.full.txt" && -f
    ↪  "$input_dir/tuft_weather.$size.txt" ]]; then
56      echo "Data already downloaded and extracted."
57      exit 0
```

```
fi
data_url="${URL}/temperatures.full.tar.gz"
wget --no-check-certificate "$data_url" -O "$input_dir/temperatures.full.tar.gz" ||
 ↪  {
    echo "Failed to download $data_url"
    exit 1
}
tar -xzf "$input_dir/temperatures.full.tar.gz" -C "$input_dir" --no-same-owner || {
    echo "Failed to extract $input_dir/temperatures.full.tar.gz"
    exit 1
}
rm "$input_dir/temperatures.full.tar.gz"
mv "$input_dir/inputs/temperatures.full.txt" "$input_dir/temperatures.full.txt"
rm -rf "$input_dir/inputs"
python3 $eval_dir/scripts/generate_input.py $input_dir/tuft_weather.$size.txt
 ↪  --size $size
```

Κώδικας Α.2: Παράδειγμα σεναρίου `fetch.sh` για το σύνολο `weather`.

```
#!/bin/bash

TOP=$(git rev-parse --show-toplevel)
eval_dir="${TOP}/weather"
outputs_dir="${eval_dir}/outputs"
scripts_dir="${eval_dir}/scripts"
input_dir="${eval_dir}/inputs"
export LC_ALL=C
size=full
selected_scripts=""
while (($#)); do
    case $1 in
        --small)    size=small ;;
        --min)      size=min ;;
        -s|--scripts)
            shift
            while (($#)) && [[ $1 != -* ]]; do
                selected_scripts+=" $1"
                shift
            done
            continue
            ;;
    esac
    shift
done

KOALA_SHELL=${KOALA_SHELL:-bash}
export BENCHMARK_CATEGORY="weather"

should_run() {
```

```
    script_name=$1
    if [ -z "$selected_scripts" ]; then
        return 0
    fi
    for selected in $selected_scripts; do
        if [ "$selected" = "$script_name" ]; then
            return 0
        fi
    done
    return 1
}
if should_run "max-temp"; then
    echo "max-temp"
    export input_file="${input_dir}/temperatures.$size.txt"
    export statistics_dir="$outputs_dir/statistics.$size"
    mkdir -p "$statistics_dir"
    BENCHMARK_INPUT_FILE="$(realpath "$input_file")"
    export BENCHMARK_INPUT_FILE
    BENCHMARK_SCRIPT="$(realpath "${scripts_dir}/temp-analytics.sh")"
    export BENCHMARK_SCRIPT
    $KOALA_SHELL "$scripts_dir/temp-analytics.sh"
    echo "$?"
fi
if should_run "tuft-weather"; then
    echo "tuft-weather"
    export BENCHMARK_SCRIPT="$scripts_dir/tuft-weather.sh"
    export BENCHMARK_INPUT_FILE="$input_dir/tuft_weather.${size}.txt"
    mkdir -p "$outputs_dir/$size"
    $KOALA_SHELL "$BENCHMARK_SCRIPT" "$BENCHMARK_INPUT_FILE" "$size" >
     ↪  "$outputs_dir/$size/turf_weather.log"
    echo "$?"
    rm -rf "$outputs_dir/$size/plots" || true
    mkdir -p "$outputs_dir/$size/plots"
    if [ -d "$eval_dir/plots" ]; then
        mv "$eval_dir/plots"/* "$outputs_dir/$size/plots/"
    fi
fi
```

Κώδικας Α.3: Παράδειγμα σεναρίου `execute.sh` για το σύνολο `weather`.

```
#!/bin/bash

TOP=$(git rev-parse --show-toplevel)

eval_dir="${TOP}/weather"

size="full"
generate=false
selected_scripts=""
```

```
10
11  while (($#)); do
12      case $1 in
13          --generate) generate=true ;;
14          --small)    size=small ;;
15          --min)      size=min ;;
16          -s|--scripts)
17              shift
18              while (($#)) && [[ $1 != -* ]]; do
19                  selected_scripts+=" $1"
20                  shift
21              done
22              continue
23              ;;
24      esac
25      shift
26  done
27
28  statistics_dir="${eval_dir}/outputs/statistics.$size"
29  correct_dir="${eval_dir}/correct-results/statistics.$size"
30
31  should_run() {
32      script_name=$1
33      if [ -z "$selected_scripts" ]; then
34          return 0
35      fi
36      for selected in $selected_scripts; do
37          if [ "$selected" = "$script_name" ]; then
38              return 0
39          fi
40      done
41      return 1
42  }
43  if $generate; then
44      if should_run "max-temp"; then
45          mkdir -p "$correct_dir"
46          cp -r "$statistics_dir"/* "$correct_dir"
47      fi
48      if should_run "tuft-weather"; then
49          hash_dir="$eval_dir/hashes/$size"
50          hash_file="$hash_dir/tuft-weather.hash"
51          plot_root="$eval_dir/outputs/$size/plots"
52          mkdir -p "$hash_dir"
53          find "$plot_root" -type f -name '*.png' ! -path '*/tmp/*' -print0 | sort -z
              ↪  | tr '\0' '\n' > "$hash_file"
54      fi
55      exit 0
56  fi
57
58  if should_run "max-temp"; then
59      diff -q "$statistics_dir/average.txt" "$correct_dir/average.txt"
```

```
    echo average.$size $?
    diff -q "$statistics_dir/min.txt" "$correct_dir/min.txt"
    echo min.$size $?
    diff -q "$statistics_dir/max.txt" "$correct_dir/max.txt"
    echo max.$size $?
fi

if should_run "tuft-weather"; then
    hash_dir="$eval_dir/hashes/$size"
    hash_file="$hash_dir/tuft-weather.hash"
    plot_root="$eval_dir/outputs/$size/plots"
    all_exist=true
    while IFS= read -r filepath; do
        if [ ! -f "$filepath" ]; then
            echo "Missing: $filepath"
            all_exist=false
        fi
    done < "$hash_file"
    if [ "$all_exist" = true ]; then
        echo "tuft-weather 0"
    else
        echo "tuft-weather 1"
    fi
fi
```

Κώδικας Α.4: Παράδειγμα σεναρίου `validate.sh` για το σύνολο `weather`.

```
#!/bin/bash

for arg in "$@"; do
    case "$arg" in
        "-f") force=true ;;
    esac
done
TOP=$(git rev-parse --show-toplevel)
input_dir="${TOP}/weather/inputs"
outputs_dir="${TOP}/weather/outputs"
rm -rf "${outputs_dir}"
if [ "$force" = true ]; then
    rm -rf "${input_dir}"
fi
```

Κώδικας Α.5: Παράδειγμα σεναρίου `clean.sh` για το σύνολο `weather`.

Παράρτημα Β

Πλήρες Πρόγραμμα Κελύφους που Παράγεται από το PaSh για το παράδειγμα `weather`

```
#!/bin/bash

cd "$(dirname "${0}")"
[ -z "${PASH_TOP}" ]
rm_pash_fifos() {
    rm -f /tmp/xUz5/fifo9
    rm -f /tmp/xUz5/fifo10
     # ...similarly for the rest of the fifos...
}
mkfifo_pash_/tmp/xUz5/fifos() {
    mk/tmp/xUz5/fifo /tmp/xUz5/fifo9
    mk/tmp/xUz5/fifo /tmp/xUz5/fifo10
    # ...similarly for the rest of the fifos...
}
rm_pash_/tmp/xUz5/fifos; mk/tmp/xUz5/fifo_pash_/tmp/xUz5/fifos; pids_to_kill=""
d="./data/temperatures"

{ r_split "./data/temperatures/2000" 1000000 \ /tmp/xUz5/fifo9 /tmp/xUz5/fifo10 & }
pids_to_kill="${!} ${pids_to_kill}"
{ r_wrap bash -c ' cut -c 89-92 ' 0< /tmp/xUz5/fifo9 > /tmp/xUz5/fifo11 & }
pids_to_kill="${!} ${pids_to_kill}"
{ r_wrap bash -c ' cut -c 89-92 ' 0< /tmp/xUz5/fifo10 > /tmp/xUz5/fifo12 & }
pids_to_kill="${!} ${pids_to_kill}"
{ r_wrap bash -c ' grep -v 999 ' 0< /tmp/xUz5/fifo11 > /tmp/xUz5/fifo13 & }
pids_to_kill="${!} ${pids_to_kill}"
{ r_wrap bash -c ' grep -v 999 ' 0< /tmp/xUz5/fifo12 > /tmp/xUz5/fifo14 & }
pids_to_kill="${!} ${pids_to_kill}"
{ r_unwrap 0< /tmp/xUz5/fifo13 > /tmp/xUz5/fifo15 & }
pids_to_kill="${!} ${pids_to_kill}"
{ r_unwrap 0< /tmp/xUz5/fifo14 > /tmp/xUz5/fifo16 & }
pids_to_kill="${!} ${pids_to_kill}"
{ dgsh-tee -i /tmp/xUz5/fifo15 -o /tmp/xUz5/fifo20 -I -f -b 5M & }
pids_to_kill="${!} ${pids_to_kill}"
{ dgsh-tee -i /tmp/xUz5/fifo16 -o /tmp/xUz5/fifo21 -I -f -b 5M & }
pids_to_kill="${!} ${pids_to_kill}"
{ sort -r -n 0< /tmp/xUz5/fifo20 > /tmp/xUz5/fifo17 & }
pids_to_kill="${!} ${pids_to_kill}"
{ sort -r -n 0< /tmp/xUz5/fifo21 > /tmp/xUz5/fifo18 & }
```

```
pids_to_kill="${!} ${pids_to_kill}"
{ dgsh-tee -i /tmp/xUz5/fifo17 -o /tmp/xUz5/fifo22 -I -f -b 5M & }
pids_to_kill="${!} ${pids_to_kill}"
{ dgsh-tee -i /tmp/xUz5/fifo18 -o /tmp/xUz5/fifo23 -I -f -b 5M & }
pids_to_kill="${!} ${pids_to_kill}"
{ sort -r -n -m /tmp/xUz5/fifo22 /tmp/xUz5/fifo23 > /tmp/xUz5/fifo6 & }
pids_to_kill="${!} ${pids_to_kill}"
{ head -n 1 0< /tmp/xUz5/fifo6 > "max.2000" & }
pids_to_kill="${!} ${pids_to_kill}"
#...
# For average calculation, replace the last two steps with:
r_merge intermediate_stream1 intermediate_stream2 > merged_stream
# Process stateful commands sequentially (Reduce)
awk '{t+=$1; i++} END {if (i>0) print t/i}' < merged_stream > "avg.2000"
# Wait for completion, cleanup FIFOs, and exit
source wait_for_output_and_sigpipe_rest.sh ${!}; rm_pash_fifos;
(exit "${internal_exec_status}")
```

Κώδικας Β.1: Πλήρες πρόγραμμα κελύφους που παράγεται από το PaSh για το παράδειγμα `weather` (`--width=2`). Το πρόγραμμα αυτό συντονίζει την εκτέλεση των εντολών του σεναρίου, διαχειρίζεται τα ενδιάμεσα αρχεία και τις ροές δεδομένων, και εκμεταλλεύεται τον παραλληλισμό όπου είναι δυνατόν.

Παράρτημα C

Κώδικας Συγκριτικής Αξιολόγησης

C.1 Μετροπρόγραμμα `count-trigrams`

```
#!/bin/bash

IN=${IN:-$SUITE_DIR/inputs/pg}
OUT=${1:-$SUITE_DIR/outputs/4_3b/}
ENTRIES=${ENTRIES:-1000}
mkdir -p "$OUT"
pure_func() {
    input=$1
    TEMPDIR=$(mktemp -d)
    cat > ${TEMPDIR}/${input}.words
    tail +2 ${TEMPDIR}/${input}.words > ${TEMPDIR}/${input}.nextwords
    tail +3 ${TEMPDIR}/${input}.words > ${TEMPDIR}/${input}.nextwords2
    paste ${TEMPDIR}/${input}.words \${TEMPDIR}/${input}.nextwords
     ↪  ${TEMPDIR}/${input}.nextwords2 |
    sort | uniq -c
    rm -rf ${TEMPDIR}
}
export -f pure_func
for input in $(ls ${IN} | head -n ${ENTRIES} | xargs -I arg1 basename arg1)
do
    cat $IN/$input | tr -c 'A-Za-z' '\n' | grep -v "^\s*$" | pure_func $input >
     ↪  ${OUT}/${input}.trigrams
done
```

Κώδικας C.1: Το σενάριο `count-trigrams`. Χρησιμοποιεί προσωρινά αρχεία και σειριακή εκτέλεση.

```
#!/bin/bash

IN="${IN:-$SUITE_DIR/inputs/pg}"
OUT="${1:-$SUITE_DIR/outputs/4_3b/}"
ENTRIES="${ENTRIES:-1000}"
mkdir -p "$OUT"
pure_func() {
    input_file="$1"
    output_file="$2"
```

```
    job_id=$(basename "$input_file")
    tmp_dir="$OUT/.tmp_$$_${job_id}"; mkdir -p "$tmp_dir"
    f_raw1="$tmp_dir/raw1"; f_raw2="$tmp_dir/raw2"; f_raw3="$tmp_dir/raw3"
    f_tail2="$tmp_dir/tail2"; f_tail3="$tmp_dir/tail3"
    mkfifo "$f_raw1" "$f_raw2" "$f_raw3" "$f_tail2" "$f_tail3"
    tail -n +2 < "$f_raw2" > "$f_tail2" &
    tail -n +3 < "$f_raw3" > "$f_tail3" &
    paste "$f_raw1" "$f_tail2" "$f_tail3" | sort | uniq -c > "$output_file" &
    agg_pid=$!
    tr -c 'A-Za-z' '\n' < "$input_file" | \
    grep -v '^[[:space:]]*$' | \
    tee "$f_raw2" "$f_raw3" > "$f_raw1"
    wait $agg_pid
    rm -rf "$tmp_dir"
}
MAX_PROCS=$(nproc 2>/dev/null || echo 4)
job_count=0; processed_count=0
for file in "$IN"/*; do
    [ -f "$file" ] || continue
    if [ "$processed_count" -ge "$ENTRIES" ]; then
        break
    fi
    input_name=${file##*/}
    pure_func "$file" "$OUT/${input_name}.trigrams" &
    job_count=$((job_count + 1))
    if [ "$job_count" -ge "$MAX_PROCS" ]; then
        wait; job_count=0
    fi
    processed_count=$((processed_count + 1))
done
wait
```

Κώδικας C.2: Το σενάριο `count-trigrams` μετασχηματισμένο με χρήση του Shark. Χρήση FIFOs και παραλληλισμού παρασκηνίου.

C.2 Μετροπρόγραμμα `covid-1`

```
#!/bin/bash

cat "$1" |
  sed 's/T..:..:..//' |
  cut -d ',' -f 1,3 |
  sort -u |
  cut -d ',' -f 1 |
  sort |
  uniq -c |
  awk "{print \$2,\$1}"
```

Κώδικας C.3: Το σενάριο `covid-1`. Τυπική σωλήνωση επεξεργασίας.

```
#!/bin/bash

INPUT="$1"
MAX_PROCS=${MAX_PROCS:-$(nproc)}
chunk_size=${chunk_size:-100M}
process_chunk() {
  sed 's/T..:..:..//'| cut -d ',' -f 1,3
}
export -f process_chunk
tmp_dir=$(mktemp -d)
trap "rm -rf $tmp_dir" EXIT
cat "$INPUT" | parallel --pipe --block "$chunk_size" -j "$MAX_PROCS" process_chunk
 ↪  > "$tmp_dir/combined.tmp"
sort -u "$tmp_dir/combined.tmp" |
  cut -d ',' -f 1 |
  sort |
  uniq -c |
  awk '{print $2,$1}'
```

Κώδικας C.4: Το σενάριο `covid-1` μετασχηματισμένο με χρήση του GNU `parallel`. Χειροκίνητη τμηματοποίηση και ορισμός συναρτήσεων.

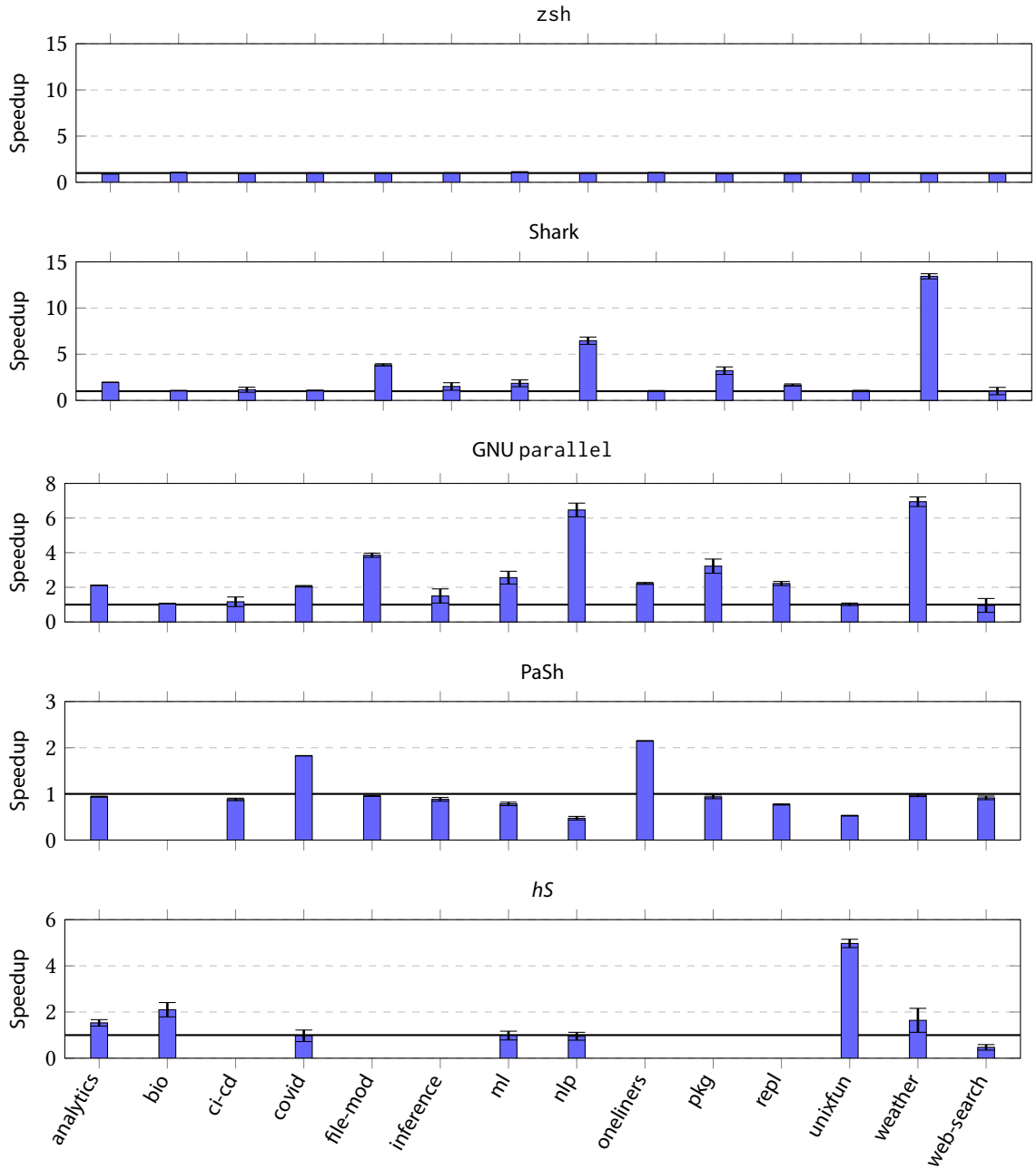
C.3 Μετροπρόγραμμα `spell`

```
#!/bin/bash
# Calculate mispelled words in an input

dict=$SUITE_DIR/inputs/dict.txt

TEMP_C1="/tmp/{/}.out1"
TEMP1=$(seq -w 0 $(($JOBS - 1)) | sed 's+^+/tmp/in+' | sed 's/$/.out1/' | tr '\n' '
 ↪  ')
TEMP1=$(echo $TEMP1)
mkfifo $TEMP1
parallel "cat {} | col -bx | tr -cs A-Za-z '\n' | tr A-Z a-z | \
tr -d '[:punct:]' | sort > $TEMP_C1" ::: $IN &
sort -m $TEMP1 | parallel -k --jobs ${JOBS} --pipe --block "$BLOCK_SIZE" "uniq" |
uniq | parallel -k --jobs ${JOBS} --pipe --block "$BLOCK_SIZE" "grep -vx -f $dict -"
rm $TEMP1
```

Κώδικας C.5: Το σενάριο `spell` μετασχηματισμένο με χρήση του GNU `parallel`. Πολύπλοκη διαχείριση ροών.

Παράρτημα D

Συγκριτικά Διαγράμματα Επιτάχυνσης



Σχήμα D.1: Συγκριτική επισκόπηση επιταχύνσεων. Τα διαγράμματα παρουσιάζουν τη σχετική επιτάχυνση (speedup) για τα 14 κύρια σύνολα μετροπρογραμμάτων. Η απουσία ράβδων υποδηλώνει ότι η εκτέλεση απέτυχε ή παρήγαγε εσφαλμένο αποτέλεσμα.

National Technical University of Athens
School of Electrical and Computer Engineering
Department of Computer Science

Characterization of the Koala Benchmarks

DIPLOMA THESIS

GEORGIOS KAOUKIS

Supervisor : Georgios Goumas
Professor, NTUA

Co-Supervisor : Nikos Vasilakis
Assistant Professor, Brown University

Athens, February 2026

National Technical University of Athens
School of Electrical and Computer Engineering
Department of Computer Science

Characterization of the Koala Benchmarks

DIPLOMA THESIS

GEORGIOS KAOUKIS

Supervisor : Georgios Goumas
Professor, NTUA

Co-Supervisor : Nikos Vasilakis
Assistant Professor, Brown University

Approved by the examining committee on February 19, 2026.

Georgios Goumas Professor, NTUA	Nectarios Koziris Professor, NTUA	Nikos Vasilakis Assistant Professor, Brown University

Athens, February 2026

Georgios Kaoukis

Graduate of the School of Electrical and Computer Engineering NTUA

Abstract

Koala is a benchmark suite aimed at performance-oriented research targeting the Unix and Linux shell. It combines a systematic collection of diverse shell programs derived from tasks found in the wild, various real inputs to these programs facilitating small and large deployments, extensive analysis and characterization aiding their understanding, as well as additional infrastructure and tooling aimed at usability and reproducibility in systems research. The Koala benchmarks execute a broad spectrum of common shell tasks; they leverage all main language features of the POSIX shell; they use a variety of POSIX tools, GNU Coreutils, and third-party software; they operate on inputs of varying size and composition; and they exhibit a wide range of performance characteristics, making them heterogeneous and suitable for evaluating a variety of optimization strategies. Applying Koala to five systems that affect shell program execution offers a broader perspective on their trade-offs, generalizes their key results, and contributes to a better understanding of these systems.

Keywords

Shell, Unix, Linux, Benchmarks, POSIX, Performance Evaluation

Acknowledgements

I would like to express my sincere gratitude to everyone who supported me during the completion of this thesis and contributed to making my years as a student so meaningful and creative.

First and foremost, I want to thank my supervisor, Prof. Georgios Goumas, for his trust, his guidance, and for giving me the opportunity to conduct my thesis at the Computing Systems Laboratory. His support has significantly influenced my academic development, as his courses were instrumental in rekindling my enthusiasm for the field during the third year of my studies.

I am also deeply grateful to Prof. Nikos Vasilakis for the opportunity to carry out this thesis in collaboration with the Atlas group at Brown University. His continuous feedback and insightful guidance were pivotal to the completion of this work and shaped the early stages of my research journey. Special thanks are due to Evangelos Lamprou, for his constant support, fruitful discussions, and practical assistance, as well as to Dr. Lukas Lazarek, whose contributions and insightful comments significantly shaped the direction and clarity of this project.

As this thesis was developed within the framework of a broader research effort, I would also like to thank Ethan Williams for his significant and extensive contribution, as well as the other co-authors of the Koala paper—Zhuoxuan Zhang, Prof. Michael Greenberg, and Prof. Konstantinos Kallas—for their collaboration throughout the wider project.

Furthermore, I extend my warm thanks to my colleagues from the "Archimedes" research unit, Eleni, Giannis, and Danai, for their patience and support during the demanding process of writing and finalizing this thesis.

I would also like to express my thanks to Panagiotis, my "cousin" and fellow graduate of the School of Electrical and Computer Engineering, who steadily helped me maintain my focus and supported me throughout our studies at the NTUA; to Filippos, with whom, beyond many outings and vacations, I shared our final exam periods; and to my fellow "compañeros", Anastasia, Giorgos, Nikolas, Athina, and Natalia, who were by my side during lectures, assignments, and exams. I also thank Giannis, Nikolas, Themis, Natalia, Andronikos, Lydia, as well as all my other friends, relatives, and associates, who—each in their own way—ensured that my student years were filled with so many happy memories.

Above all, I would like to thank my parents, Zafiris and Maria, whose endless patience, constant support, and unwavering belief in me formed the foundation of all my efforts.

Georgios Kaoukis,

Athens, February 19, 2026

Contents

List of Tables

List of Figures

List of Source Code Listings

Chapter 1

Introduction

The Unix shell is popular, versatile, and powerful, with a variety of applications ranging from data processing and system administration to bioinformatics and continuous integration and deployment. At the same time, the shell has garnered significant research interest, with recent academic activity [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] having led to the development of several optimizers aimed at accelerating shell programs via parallel systems [3, 5, 7], distributed systems [1, 11, 12], and other forms of scaling [4, 9, 13, 14].

Despite this, a major obstacle hinders the effective conduct of shell-related research: there is no established benchmark suite for evaluating these systems, making it difficult to evaluate new systems in a realistic and comparable manner [15]. Thus, researchers are often forced to create ad-hoc microbenchmarks, collect fragmented examples from repositories, or rely on standardized tests that focus primarily on behavioral equivalence rather than performance. Consequently, research in the field lacks fundamental benefits of systematic benchmarking, such as repeatability, reproducibility, and fair comparison among different systems.

Koala is a research-oriented benchmark suite for performance evaluation [16], which targets the Unix shell and the Linux operating system, combining a systematic collection of programs, extensive characterization, and reusable infrastructure. To achieve this goal, Koala provides a diverse collection of 126 real shell programs that perform tasks frequently encountered in practice. It offers realistic input data at three scales (minimal, small, and full), as well as automated tools for dependency installation, benchmark execution, result validation, and performance reporting.

This thesis is a subset of the broader research effort presented in the publication [16] at USENIX ATC '25. The main axis and core contribution of this work focus on the practical application, script transformation, and extensive comparative evaluation of different systems. Through this process, the utility and necessity of the Koala suite for evaluating both existing and future optimization systems are practically demonstrated. Specifically, the thesis explores the practical implications of using Koala by evaluating five different acceleration approaches and systems: the alternative interpreter `zsh` [17], Shark [15], GNU `parallel` [18], PaSh [5, 7], and *hS* [14]. For each system, its operating mechanism, adoption effort, and required code transformations on real programs are extensively analyzed, while the achieved speedups, as well as the advantages or disadvantages of each approach, are presented.

The structure of this thesis is organized as follows: Initially, the necessary theoretical background is presented, focusing on the core abstractions, data flow, and operation of the Unix shell (Chapter 2). Next, the relevant research literature is outlined, examining existing optimization systems, collections of shell programs, evaluation landscapes, and previous benchmarking efforts (Chapter 3). This is followed by the analysis of a representative motivating example, which practically highlights the challenges of shell acceleration and sets the stage for the necessity of a comprehensive infrastructure (Chapter 4). Subsequently, the design and implementation of the Koala suite are described, detailing the design principles, computational domains, and benchmark sets (Chapter 5).

Then, the main body of the work is developed, which is distributed across five distinct system evaluation chapters. Specifically, it examines the execution of scripts via the `zsh` interpreter (Chapter 6), analyzes the static transformations based on Shark (Chapter 7), studies manual parallelism using GNU `parallel` (Chapter 8), evaluates the dynamic data flow parallelization of PaSh (Chapter 9), and explores the speculative out-of-order execution of *hS* (Chapter 10).

Finally, the thesis concludes with the synthesis of the findings and the final conclusions drawn from the evaluation (Chapter 11), as well as the presentation of the proposed future work for extending the research (Chapter 12). The infrastructure and benchmarks of the Koala suite are freely available under the MIT license at https://github.com/kbensh/koala.

Chapter 2

Background

This chapter presents the necessary theoretical background for an in-depth understanding of the behavior, structure, and complexity of Unix shell programs. First, the shell environment and the core abstractions that govern execution and command composition are analyzed. Then, the inherent peculiarities of the language are examined, such as the tight coupling of parsing to execution time and the strict word expansion rules of the POSIX standard. Through this analysis, the fundamental limitations and technical challenges that make the automatic optimization and parallelization of shell programs a particularly demanding research problem are highlighted, paving the way for the systematic study that follows.

2.1 The Unix Shell

The Unix shell has been the cornerstone of user-system interaction for decades, acting simultaneously as a powerful interactive command interpreter and an expressive high-level programming language. According to the Unix philosophy [19], it provides simple and general mechanisms for composing small, independent tools into larger programs. Through this environment, users have the ability to compose and execute complex programs by combining smaller, specialized, and independent commands into complex processing flows.

The operation and behavior of the shell are defined by the POSIX standard, which establishes the syntax rules and semantic properties, ensuring interoperability and portability of programs across different operating systems and architectures [20]. Unlike traditional general-purpose programming languages like C or Java, the shell was not designed primarily for implementing complex algorithms or data structures, but for the efficient coordination and interconnection of existing programs, tools, and system services. In this way, it acts as a *glue* software layer, allowing developers to quickly build functional data processing pipelines using ready-made and tested building blocks. Its expressiveness has been highlighted in the literature, as tasks that require extensive implementation in languages like Java can often be expressed with a single shell command [21].

Modern Use and Significance Despite the emergence of newer languages and tools, the shell maintains its importance undiminished and remains an integral part of modern computing infrastructure. It is consistently ranked among the most popular programming languages [22], while recent studies demonstrate a significant increase in its usage [14]. The shell is ubiquitous in critical areas such as system administration, the automation of repetitive tasks, bulk data processing, bioinformatics, as well as the orchestration of computational workflows in distributed environments and cloud infrastructures. Particularly in the modern software development ecosystem, dominated by technologies like Docker and platforms like Kubernetes, the shell is the established standard for defining entry points, preparing environments, and executing continuous integration scripts [23]. Therefore, shell programs are not merely auxiliary tools, but are frequently executed in large-scale infrastructures, directly affecting the reliability, speed, and overall performance of critical production systems. This extensive use makes a deep understanding of its mechanisms, as well as the optimization of its execution, particularly important for modern systems research.

```
#!/bin/bash
mkfifo pipe1 pipe2
grep "Failed" log1 > pipe1 & grep "Failed" log2 > pipe2 &
cat pipe1 pipe2 | sort | uniq -c
```

Listing 2.1: Laziness in a realistic catalog processing scenario. The `cat` command first reads from `t1` and only after finishing reading this FIFO does it proceed to `t2`. Therefore, the second `grep` process blocks, because there is not yet a process reading from `t2`, and continues only when the first `grep` finishes.

Core Abstractions and Data Flows The expressive power and flexibility of the shell stem from a set of simple, yet highly powerful abstractions, which embody the Unix philosophy of composing small and specialized tools. A central role is played by data flows, which are implemented as serial sequences of bytes with no inherent structure and are transferred between processes via mechanisms like pipes or files. Anonymous pipes allow the direct connection of the standard output (`stdout`) of one command to the standard input (`stdin`) of another, creating pipelines, while named pipes (Unix FIFOs) offer persistent communication channels in the file system. In this context, the operating system kernel assumes the role of managing, routing, and synchronizing the data, allowing the concurrent execution of connected commands without the user needing to explicitly manage threads or synchronization mechanisms.

Additionally, each command in the shell environment is treated as an autonomous and independent computational unit that reads data, processes it, and produces results. The set of available commands is practically limitless, as any executable file on the system can be invoked as a command. Commands can be implemented in any programming language (C, Python, Rust, etc.) or may only be available only in binary format, with an unknown internal structure and behavior. This feature, while offering great flexibility, makes predicting performance and resource requirements particularly difficult without conducting empirical measurements.

To coordinate these units, the shell provides a series of composition operators, such as the sequential operator (`;`), the background execution operator (`&`), the pipe operator (`|`), and the logical operators (`&&`, `||`). Through these, the programmer can precisely define the temporal and logical dependencies between processes.

However, data flow management also presents significant challenges. Many commands are designed to be "lazy" or strictly follow serial consumption logic, reading data only when they are absolutely ready to process it. Although this behavior is often desirable for minimizing memory usage, in complex pipelines it can lead to blocking phenomena and underutilization of available computational resources, as shown in the above example (Listing 2.1).

Distinction Between Shell and Commands For proper performance analysis, a clear separation of responsibilities between the shell interpreter itself and the individual commands it invokes is of fundamental importance. The interpreter (e.g., `bash`, `zsh`) is responsible for parsing the code, expanding variables, managing file descriptors, and creating processes via the `fork` and `exec` system calls. In contrast, the main computational workload and data processing are performed by the external commands.

A critical element that is often overlooked is the impact of execution parameters on command behavior. These parameters not only alter functionality but can significantly affect the performance profile and parallelization potential. For example, a command that normally processes data as a stream might, with the addition of appropriate flags, require fully loading the file into memory or perform in-place modifications[1], radically changing how it interacts with the file system and other processes.

[1] A characteristic example is the `sed` command: in its standard use, it acts as a filter, allowing parallelization in a pipeline. However, using the `-i` flag forces writing to the original file, requiring the creation of temporary files and limiting the possibilities for parallel processing [24].

Peculiar Parsing and Semantics Unlike traditional programming languages, the parsing of the shell cannot be strictly separated from its execution. As highlighted by research surrounding the design of the shell as a Domain-Specific Language [25], code interpretation depends directly on the dynamic state of the environment at runtime. For example, the existence of an `alias` or the dynamic assignment of a variable can radically change how the interpreter translates the immediately following line of code. This architectural peculiarity has historically made the creation of static analysis tools extremely difficult, as constructing an accurate abstract syntax tree (AST) often requires executing parts of the program itself.

The Word Expansion Process One of the most critical mechanisms of the POSIX shell, which is largely responsible for its complexity and dynamic nature, is the strict word expansion process that takes place *before* the execution of any command [25, 26]. The standard defines a sequence of seven distinct transformation stages, which are applied hierarchically:

1. Tilde expansion (~).
2. Parameter and variable expansion.
3. Command substitution.
4. Arithmetic expansion.
5. Field splitting.
6. Pathname expansion/globbing.
7. Quote removal.

This process makes shell code highly variable. A simple command that syntactically appears to be a harmless data transfer can—after the command substitution or pathname expansion stage—turn into a heavy computational task involving thousands of files in the system. Consequently, any optimization or parallelization effort must ensure that these transformations have been correctly evaluated, demonstrating the inadequacy of purely static approaches in the majority of real-world scenarios.

Limitations and Challenges Despite its utility and widespread adoption, the shell programming model is characterized by significant limitations that hinder the development of efficient and reliable applications:

- **Limited Scaling and Parallelization:** Although the shell supports basic forms of parallelism through pipelines and background execution, effectively utilizing modern multi-core architectures remains difficult. The user often resorts to complex manual manipulations, using operators and commands like `&` and `wait` [25] or external tools, such as GNU `parallel` [18], a process that is time-consuming and error-prone.
- **Dynamic and Opaque Behavior:** As mentioned earlier, due to its peculiar semantics and expansion stages, the execution of a program heavily depends on the environment's state, file existence and permissions, and system settings that may change dynamically. This property, combined with the opacity of external binaries, makes predicting behavior particularly difficult.
- **Error-Proneness and Limited Security:** The shell lacks many security mechanisms found in modern programming languages, such as a strong type system or structured exception handling. As a result, simple programming errors, like using undefined variables in critical commands, can have irreversible consequences. A characteristic example is the critical bug identified in the Steam installation script for Linux [27], where the lack of validation for the `STEAMROOT` variable led to the unintended execution of the command:

```
rm -rf "$STEAMROOT/"*
```

In cases where the variable was empty, the command dynamically transformed into `rm -rf "/"*`, causing the deletion of the user's entire file system.

The above properties compose an environment in which systematic performance analysis and automatic optimization remain open and demanding research problems.

Chapter 3

Related Work

This chapter presents the existing literature and the broader research landscape surrounding the Unix shell. The chapter is structured along two main axes. First, it examines efforts to optimize and accelerate shell programs, ranging from alternative interpreters and traditional compilers to modern dynamic and distributed systems. Next, it analyzes the evaluation landscape in the systems and shell domain, highlighting the shortcomings of existing program collections and substantiating the need for a systematic, performance-oriented suite, such as Koala.

3.1 Acceleration and Optimization Efforts

The need for a radical improvement in the performance of shell programs has sparked the development of numerous systems attempting to overcome the limitations of strictly serial execution. Increased research activity around the Unix shell has led to innovative solutions covering a vast array of techniques. Among these stand out methods for optimizing input/output (I/O) operations [15], the automatic synthesis of new pipelines [3], the automatic parallelization of pipelines [5, 7], as well as command fusion aimed at reducing inter-process communication overhead [28]. Concurrently, studies diverging from traditional systems are of particular interest, focusing on efficient script execution in mobile environments [29] or the dynamic refinement of system calls [30].

Specifically, in the domain of scaling and distributed execution, modern research presents groundbreaking architectural solutions. For instance, POSH [1] adopts a data-aware offloading strategy, routing computation directly to the data source to minimize network congestion. Targeting cluster infrastructures, systems like DiSh [11] enable seamless execution across multiple nodes, while Fractal [12] additionally integrates critical fault tolerance mechanisms. Broadening this field further, frameworks like SPLaSh [24] introduce the shell into the cloud computing ecosystem, investigating its dynamic and elastic scaling via Function-as-a-Service (FaaS) models.

This work does not aim to provide an exhaustive catalog of all the aforementioned categories. Instead, it focuses on five representative environments that target single-node acceleration and embody fundamentally different design philosophies. These systems, which are the subject of extensive evaluation in the following chapters, are:

- **`zsh` [17]:** An alternative, highly customizable interpreter. Although developed with the goal of improving the interactive user experience compared to `bash`, it implements a different internal memory management and parsing engine, and can impact the execution speed of scripts with shell built-ins.
- **Shark [15]:** Represents one of the first research systems to treat shell programs with tools borrowed from traditional compiler theory. Its core innovation lies in treating file system accesses as variable references. By constructing directed acyclic graphs (DAGs), Shark applies static optimizations, such as eliminating redundant operations (e.g., useless use of `cat`), replacing temporary files with pipes to reduce I/O overhead, and statically parallelizing independent commands.
- **GNU `parallel` [18]:** The most widely accepted tool for explicit, manual parallelization in the Unix ecosystem. It provides developers with advanced mechanisms for splitting data and execut-

ing multiple processes concurrently. However, its impressive performance comes with a critical requirement: the responsibility for maintaining data dependencies and ensuring functional correctness is shifted entirely to the user.

- **PaSh [7]:** Represents the cutting edge of modern research in automatic parallelization via Just-in-Time compilation. The system dynamically alternates between analysis and execution phases, transforming pipelines into Data-Flow Graphs. Relying on an annotations library that encodes the behavior of POSIX commands, PaSh introduces parallel execution schemes, ensuring semantic equivalence with the script's original serial form.

- ***hS* [14]:** Proposes a radical architecture drawing inspiration from the microarchitecture of modern processors, introducing speculative out-of-order execution at the operating system level. *hS* prospectively executes future commands in isolated environments (sandboxes). Through systematic tracing of I/O and environment variables, it commits results provided no conflicts are detected, otherwise it rolls back execution and returns to the safe serial flow.

These approaches demonstrate that the shell optimization space is multidimensional. However, the ultimate effectiveness of each system is inextricably linked to the syntactic structure, data volume, and dynamic behavior of the specific program. This fact necessitates the use of a realistic and extensive evaluation framework to systematically capture these limits.

3.2 The Benchmarking Landscape

Progress in computing systems design is directly intertwined with the ability to perform reliable, repeatable, and fair benchmarking. The existence of commonly accepted benchmark suites allows the comparison of different architectures under common conditions, the cross-validation of experimental findings, and the systematic study of trade-offs between performance and resource utilization [16]. Historically, the establishment of such infrastructures has been the catalyst for the maturation of many research fields, replacing ad-hoc experiments with rigorous, standardized methodologies.

Established Suites in Systems Research

Progress in computer science and the broader field of systems research depends heavily on the capability for fair and *objective comparisons*. In this context, the use of open and reusable standardized benchmarks serves as the foundation of scientific evaluation. In various domains, the establishment of such tools has been catalytic: hardware evaluation standards like SPEC [31], databases like TPC [32], multicore architectures like PARSEC [33], and cloud computing applications like CloudSuite [34], have established rigorous validation procedures and common reference points.

Similarly, in the software and programming language space, the need for realistic workloads led to the creation of specialized tools. Suites such as DaCapo [35, 36] for Java, as well as the Gabriel benchmarks [37] for LISP, enabled the in-depth study of managed execution environments under production conditions (such as garbage collection and object-oriented hierarchy behavior [38]). Concurrently, the reproducibility of modern research is actively supported by reference datasets in various fields, like EMNIST [39] in machine learning, Defects4J [40] in software testing, and Magma [41] in security tool evaluation. A common characteristic of all these successful efforts is the emphasis on end-to-end applications and clearly defined execution procedures. Unlike performance microbenchmarks that isolate individual operations, comprehensive suites capture the intricate interactions of a system, highlighting phenomena that do not appear in synthetic scenarios. The existence of an established suite substantially accelerates the evolution of a given research area.

Prior research has studied program behavior either in simulation environments or on specific hardware implementations [42, 43, 44]; however, such an approach binds the conclusions to the specific execution platform and the corresponding operating environment. In the shell domain, there is a need

to systematically highlight properties that are independent of the underlying hardware and operating system, so that programs can be used as general and portable workloads for evaluating different computational substrates. Consequently, an open and carefully designed evaluation suite specifically tailored for shell programs becomes necessary.

Just as suites like DaCapo and Gabriel historically filled gaps in evaluating the programming environments of their time, the Koala system emerges to address a corresponding modern need. It represents the first systematic effort to create an open performance evaluation suite for the shell, introducing systematization, comparability, and reproducibility into a field that until now lacked such standards.

Existing Approaches and Evaluation Shortcomings in the Shell

The multitude of optimization systems presented earlier demonstrates the pressing need for a standardized, user-friendly, and reproducible evaluation framework. However, in the absence of an established methodology, the creators of these systems had extremely limited options at their disposal to substantiate the benefits of their approaches.

The available practices used for the shell are classified into four main categories, each of which presents fundamental shortcomings regarding performance evaluation on realistic systems:

1. **Shell Microbenchmarks:** Suites like ShellBench [45], zsh-bench [46], the Oils benchmarks [47], and UnixBench [48] provide small, isolated code snippets (typically 8 to 95 lines). While they are irreplaceable tools for stress-testing individual interpreter features (e.g., subshell speed, variable substitution, or mathematical operations), they fall significantly short in realism. They deviate radically from real-world workloads, as they do not perform end-to-end computations, do not manage large datasets, and do not orchestrate heterogeneous external commands, rendering a holistic evaluation of acceleration systems impossible.

2. **Correctness & Standards Tests:** Tools such as the POSIX test suite [49], the Smoosh suite [8], the diagnostic routines of the Modernish library [50], as well as various automated testing frameworks (like shellspec [51], shunit [52], bats-core [53], and TETworks [54]), ensure strict compliance of a shell implementation with the standards. However, they focus exclusively on functional verification, rather than characterizing systems in environments involving opaque components. Failing to simulate the size, complexity, and input data of real programs, they offer absolutely no insight into how an optimization system affects actual runtime performance.

3. **Open-source and Empirical Studies:** Recent studies collect and analyze millions of programs (such as from GitHub or Linux build scripts) to understand real-world shell usage scenarios. They study the use of aliases, security practices, and user interaction with the code [55, 56]. Despite their invaluable static value, these repositories do not provide the necessary infrastructure for their execution. They typically lack input data, setup scripts, or explicit dependency declarations, often comprising noisy or incomplete code segments that cannot be used for dynamic benchmarking.

4. **Ad-hoc Collections & Evaluations:** In the absence of a common evaluation infrastructure for the acceleration systems presented in Section 3.1, the historically predominant practice has been the creation of ad-hoc, often hand-picked collections [1, 7, 11, 15, 29]. Beyond the obvious risk of bias in favor of the respective system, these small and fragmented collections lack automated installation and verification mechanisms. This leads to unfair and incompatible comparisons, further highlighting the urgent need for a standardized suite.

As a result, the lack of a standardized, realistic, and reusable evaluation suite has led to a fragmented landscape, where claims of performance improvements are often based on piecemeal measurements that cannot be reproduced or compared with other approaches. This situation undermines scientific progress, as it makes it difficult to objectively evaluate new ideas and understand the true limits of acceleration systems.

The Role and Contribution of Koala The Koala suite was designed to fill this gap, adopting principles that have proven successful in other fields of systems research. Specifically, it includes realistic shell programs derived from real-world tasks, covers multiple computational domains, and provides different input scales, enabling both rapid exploration for correctness verification and extensive evaluation. Simultaneously, it is accompanied by an automated infrastructure for installation, execution, and result validation, ensuring reproducibility. Analysis of the suite shows that it covers all the core features of the POSIX shell, while the programs exhibit significant variety in both static and dynamic characteristics, such as CPU time, memory consumption, and I/O activity. Workloads scale from brief scripts to long-running executions on extensive datasets. In this way, Koala enables the transition from fragmented measurements to a cohesive, reproducible, and reusable evaluation methodology. This framework makes it possible to systematically compare different acceleration approaches within the same controlled environment.

Chapter 4

Motivating Example

To illustrate how the KOALA suite supports the comparative evaluation of different shell acceleration systems, it is beneficial to examine a representative example. The analysis of a realistic script highlights the design philosophy, the execution challenges, and substantiates the need for a comprehensive evaluation infrastructure.

4.1 Detailed Operation of the Weather Data Script

The motivating example—the `weather` script—is based on the `temp-analytics.sh` script, which belongs to the `weather` benchmark set of the suite. Listing 4.1 presents the simplified form of this script.

```
#!/bin/bash

d="./data/temperatures";
for y in $(seq $start $end); do
  cat $d/$y | cut -c 89-92 | grep -v 999 | sort -rn | head -n1 > max.$y
  cat $d/$y | cut -c 89-92 | grep -v 999 | sort -n | head -n1 > min.$y
  cat $d/$y | cut -c 89-92 | grep -v 999 | awk "{t += \$1; i++} END {print t/i}" >
  ↪ avg.$y
done
```

Listing 4.1: Example of weather data analysis in KOALA (`weather`). Implementation of maximum, minimum, and average temperature calculation via an iterative structure and independent pipelines.

The architecture and execution of this specific program are based on a combination of an iterative control structure and data processing flows via pipelines. More specifically, the core of the script is a `for` loop, which, through the `seq` command, sequentially iterates over a defined sequence of years (from `$start` to `$end`). During each iteration, the program analyzes the corresponding historical weather data file (`$d/$y`) to extract three statistical metrics: the maximum, minimum, and average temperature of the year. To perform this calculation, the body of the loop consists of three distinct, independent pipelines. Each of these three pipelines begins with exactly the same preprocessing stages:

1. The `cat` command reads the data from the file.
2. The `cut -c 89-92` command extracts the specific characters of each line where the temperature is recorded.
3. The `grep -v 999` command filters and discards records containing the value `999`, which serves as an indicator for missing or invalid data.

From this point, the "cleaned" data follows a different path depending on the requested metric:

- For the **maximum temperature**, the data is sorted in descending numerical order (`sort -rn`), and `head -n1` isolates the first line, saving it to the `max.$y` file.
- For the **minimum temperature**, ascending numerical sorting is applied (`sort -n`) and `head -n1` outputs the result to `min.$y`.
- For the **average temperature**, the data is fed into `awk`, which sums the values (`t += \$1`), counts them (`i++`), and prints the average (`t/i`) at the end of the reading process (in the `END` block).

The fact that these specific pipelines are structurally independent of each other within the loop makes the script an ideal testbed for studying different acceleration strategies.

4.2 Practical Characteristics of the Script

This specific program was chosen because it tangibly brings together the practical characteristics of a realistic workload, embodying the design principles of Koala:

- **Representativeness and Realism:** The script is not an artificial microbenchmark. Instead, it performs actual statistical calculations on a widely known set of historical weather data—NOAA [57]. It is also used as a reference example in a classic book on the Hadoop ecosystem [58].
- **Scalability:** Its inherent ability to iteratively analyze files for multiple years means its workload can be easily scaled up or down (e.g., analyzing a decade versus a century of data). The Koala infrastructure leverages this characteristic to provide data at a small or large scale.
- **Diversity:** The harmonious coexistence of a traditional control structure (the `for` loop) with data flows (the multiple pipelines) and various POSIX tools (`cut`, `grep`, `sort`, `awk`), composes a demanding environment that exhaustively tests the capability of any tool.

4.3 Testbed for Optimization Approaches

This exact realistic composition of the code provides the ideal testing ground to highlight both the capabilities and the different adoption requirements of modern optimization tools. Each system approaches and accelerates the same script in a radically different way, optimizing different structures:

- The **`zsh`** interpreter executes the script as is, affecting the performance of shell built-ins, such as variable assignment and the `for` loop.
- The **Shark** system requires manual code intervention from the author. It removes redundant `cat` calls (passing the input directly to `cut`) and executes the three pipelines in parallel in the background.
- The **GNU `parallel`** tool also requires manual code refactoring. The user must choose which part to parallelize, applying it either to the outer loop or inside the three pipelines.
- The **PaSh** system automatically parallelizes the individual computational stages *inside* each individual pipeline using a Just-in-Time compiler.
- The ***hS*** system utilizes speculative execution to dynamically unroll the loop iterations and execute the commands concurrently and out-of-order.

It becomes clear that the complexity of this script practically demonstrates both the expected speedups and the potential limitations of each approach under realistic production conditions, highlighting why simple microbenchmarks are insufficient.

4.4 The Need for a Comprehensive Suite

As the analysis of the above example demonstrates, execution on realistic scripts is the only reliable way to reveal the true behavior of each optimization tool. However, for this evaluation to be scientifically sound and reproducible, researchers need the ability to execute the exact same scripts, with common data, and under a completely controlled environment.

This realization makes the transition from isolated, sporadic examples to an organized and fully automated infrastructure imperative. Instead of wasting time building proprietary and non-comparable workloads for each new system, a unified framework is required that will provide immediate, fair, and comparable results. Guided by this need, the next chapter presents in detail the architecture and implementation of the Koala suite, strictly defining its design principles and expanding the philosophy of this motivating example to a broad spectrum of 126 real programs.

Chapter 5

Design and Implementation of the Koala Suite

Having presented a representative example and a glimpse of the suite's philosophy in the previous chapter, this chapter focuses on the detailed presentation of Koala. First, its core design principles and the methodology for selecting the benchmarks are defined. Subsequently, the architecture of the suite is described, which includes a total of 126 real-world programs from various sources and computational domains, the infrastructure that ensures their automated and reproducible execution, and finally a section on the static and dynamic analysis that highlights its diversity.

5.1 Design Principles

The design of Koala was guided by a set of fundamental principles, which capture the core requirements for a modern shell system evaluation suite.

- **Representativeness:** The scripts in the suite constitute real programs that perform substantial computations on real data. This choice allows the study of system behavior under conditions close to production use, as opposed to isolated performance microbenchmarks.
- **Diversity:** The suite covers a broad spectrum of computational domains, such as data analytics, bioinformatics, system administration, and machine learning, with different execution profiles, while utilizing a wide range of shell commands and features. In this way, overfitting the evaluation to specific program patterns is avoided.
- **Workload Characterization:** In addition to providing benchmarks, the suite is accompanied by a systematic characterization of both their static and dynamic properties. This approach enables an understanding of how different systems affect the execution of real programs.
- **Scaling:** The suite provides input data at multiple scales (minimal, small, large), allowing for both rapid exploration and extensive, long-running evaluations. The full datasets reach hundreds of gigabytes in size, enabling the analysis of workloads with intense input/output activity.
- **Automation and Reproducibility:** The Koala suite provides infrastructure that automates environment setup, dependency installation, execution, and result validation, facilitating the reproduction of experiments by different researchers.

Benchmark Selection Methodology

The composition of the suite was based on a systematic selection process, aiming to create a representative and practically useful set of programs.

Sources of Origin The scripts originate from various sources real-world usage, covering different time periods and programming styles:

- **Open-source repositories:** Projects from platforms like GitHub and GitLab, which are used in real environments [59, 60].

- **Academic literature:** Scripts that have been used to evaluate shell acceleration systems [1, 4, 10].
- **Educational examples and classic literature:** Classic examples of command composition from projects like Unix for Poets and Unix50, which highlight the Unix philosophy [61, 62]. Examples from historical articles, books, and manuals are included [19, 58, 63].
- **Modern workflows and real data:** For example, scripts that interact with modern foundation models or process extensive real-world datasets [64, 65, 66].

Inclusion and Exclusion Criteria Specific criteria were applied for selecting the scripts, with the primary goal of gathering realistic examples of tasks. The goal was the inclusion of programs, whether simple or complex, that process real data and are characterized by heterogeneity and diversity, both in terms of POSIX syntactic features and their execution characteristics. At the same time, the incorporation of scripts without available data or functional dependencies was deliberately avoided, as well as the use of overly simplistic, trivial, or unrealistic programs that do not represent actual usage conditions.

5.2 Overview of the Suite

Based on the aforementioned design principles, the complete set of the Koala suite was assembled. Unlike performance microbenchmarks, Koala programs perform end-to-end tasks, allowing the study of complex interactions among commands, data flows, and the file system.

Table 5.1 briefly presents the complete set of Koala benchmarks and their key characteristics. The programs are organized into subsets, which share the type of task, origin, input data, or computational characteristics. The following sections analyze the main descriptive elements of the suite.

Computational Domains

The Koala suite covers a broad spectrum of computational domains, reflecting both traditional and modern uses of the shell. These domains are denoted by their initials in Table 5.1, while below they are briefly presented in full.

Data Analytics (**DA**) programs include scripts that extract, transform, and summarize large datasets, such as meteorological records, public transit data, and genomic sets. In total, 11 programs distributed across three benchmark sets are included.

System Administration (**SA**) programs correspond to typical tasks of configuration, maintenance, and system log analysis. They represent a characteristic example of daily shell usage in real production environments and include 7 scripts divided into two sets.

Continuous Integration (**CI**) flows include scripts that automate software build, analysis, and testing processes. In the Koala suite, they are represented by 23 programs distributed across two sets.

Machine Learning (**ML**) programs include either scripts implementing learning tasks or scripts acting as glue code, connecting external tools and libraries into unified workflows. They comprise 27 programs distributed across three sets.

Automation (**AN**) scripts include 18 scripts, distributed across two sets, that automate heterogeneous tasks, such as file encryption, media conversion, or specific content processing tasks.

Finally, the Miscellaneous (**MI**) category includes 40 scripts that do not fall into a specific domain but highlight the expressiveness of the shell through heterogeneous tasks, such as text transformations or the implementation of search engines.

Beyond the classification by application domain, the suite also combines different computational styles, which appear after the slash in the $\mathscr{D}$ column of Table 5.1. Examples include data extraction (DE) sets, multi-stage text processing (TP) scripts, and task automation (AN) sets, which represent different patterns of shell usage.

Table 5.1: Summary of the Koala suite benchmarks. The table summarizes all benchmark subsets of the Koala suite. Column 1: identification characteristics, presenting the name of each benchmark set and (indicatively) the programs it includes. Columns 2–4: descriptive characteristics, summarizing the computational application domain ($\mathscr{D}$), the number of programs in the set (#.sh), and the total lines of code (LoC). Column 5: summary of the input data size for each program. Columns 6–7: static characteristics, namely the number of shell syntactic constructs (#Cons) and the number of distinct commands (#Cmd). Columns 8–11: dynamic execution characteristics, such as shell execution time (t_S), command execution time (t_C), memory consumption (Mem), and total I/O volume. Columns 12–13: system characteristics, including the number of system calls (#SC) and open file descriptors (#FD). Column 14: source of origin of the programs.

(The table is presented on the next page)

Table 5.1: Part A

Benchmark/Script	Surface			Inputs		Syntax		Dynamic				System		Source
	$\mathscr{D}$	#.sh	LoC	Small	Full	#Cons	#Cmd	t_S	t_C	Mem	I/O	#SC	#FD	
analytics	SA/DA	4	53	1.94GB	78.9GB	10	21	10ms	84s	334MB	23.0GB	117.3m	84	[1, 4, 67]
`nginx.sh`			19			10	13	~0	1s	10.3MB	1.79GB			
…														
bio	DA/BI	4	823	24.3GB	114GB	17	66	51s	6720s	25.1GB	352GB	35.3m	79	[68, 69, 70]
`data.sh`			226			13	44	46s	3521s	25.1GB	287GB			
…														
ci-cd	CI/BS	21	2592	N/A		20	134	30ms	128s	461MB	2.35GB	3.5m	885	[10, 59]
…														
covid	DA/DE	5	53	3.34GB	5.08GB	5	6	~0	67s	1011MB	80.6GB	14.3m	150	[66]
`1.sh`			9			5	6	~0	12s	13.2MB	17.5GB			
…														
file-mod	AN/MI	5	41	4.35GB	39.2GB	11	10	99ms	235s	164MB	13.9GB	1.5m	61	[1, 4, 55]
`encrypt.sh`			11			10	6	~0	2s	2.82MB	5.10GB			
`img-conv.sh`			11			11	6	99ms	170s	145MB	3.83GB			
…														
inference	ML/DA	3	61	3.85GB	11.7GB	15	27	40ms	1586s	7.16GB	83.7GB	37.9m	81	[64, 71]
…														
ml	ML/DA	1	47	4.71GB	15.0GB	7	1	~0	1156s	7.87GB	41.6GB	14.9m	10	[72]
nlp	TP/ML	23	303	3k bks	115.9k bks	10	19	15s	851s	9.62MB	272GB	178.6m	526	[61]
`bigrams.sh`			19			10	16	710ms	50s	7.93MB	22.5GB			
…														

Table 5.1: Part B

Benchmark/Script	Surface			Inputs		Syntax		Dynamic				System		Source
	$\mathscr{D}$	#.sh	LoC	Small	Full	#Cons	#Cmd	t_S	t_C	Mem	I/O	#SC	#FD	
oneliners	AN/TP	13	119	202MB	13.5GB	10	24	60ms	14s	199MB	3.77GB	4.5m	288	
`spell.sh`			11			6	7	~0	1s	8.38MB	523MB			[73]
`top-n.sh`			2			5	6	~0	960ms	8.25MB	408MB			[74]
`uniq-ips.sh`			2			4	3	~0	60ms	8.15MB	54.2MB			[75]
…														[19, 63]
pkg	CI/AN	2	43	110 pkgs	2.0k pkgs	16	22	5s	201s	572MB	15.3GB	35.2m	132	[60, 76]
`pacaur.sh`			29			14	19	5s	81s	490MB	14.6GB			
`proginf.sh`			14			11	6	10ms	120s	572MB	709MB			
repl	SA/MI	3	586	N/A		20	55	~0	24s	197MB	1.21GB	5.6m	79	[1, 77]
…														
unixfun	MI/TP	36	70	599MB	59.1GB	4	12	~0	5s	9.80MB	5.71GB	944.0k	733	[62]
`1.sh`			2			3	2	~0	50ms	680KB	108MB			
`2.sh`			2			3	3	~0	260ms	7.51MB	209MB			
…														
weather	DA/DE	2	74	893MB	146GB	11	20	260ms	958s	94.2MB	39.1GB	56.7m	50	[58]
…														
web-search	MI/TP	4	34	833MB	8.61GB	14	30	11s	1343s	1.32GB	17.1GB	112.6m	174	[78]
…														
Min		1	34	1.05MB	44.9MB	4	1	0	5.4	9.62MB	1.21GB	944.0k	10	
Mean		9	349.9	3.66GB	38.0GB	12.1	31.9	6.1	955.7	3.17GB	67.9GB	44.2m	238	
Median		4	65.5	1.54GB	13.5GB	11	21.5	0.1	218.2	398MB	20.1GB	25.0m	108	
Max		36	2592	24.3GB	146GB	20	134	52	6720.9	25.1GB	352GB	178.6m	885	

Benchmark Sets

The Koala suite consists of fourteen sets of programs, which are presented in alphabetical order and cover different computational paradigms and shell usage styles [16]. Each set includes programs with a common origin, similar input data, or corresponding computational characteristics.

`analytics` includes four log file analysis programs, which perform filtering and event summarization [1, 4]. The programs process a total of 78.9GB of line-based data, such as TCP traces, Nginx access logs, and ZMap scan data, originating from real network captures [79, 80, 81, 82]. The small version of the data consists of a truncated subset of the same files.

`bio` consists of four genomic and transcriptomic data processing programs. One program performs population genomics analysis [68, 69], while the other three implement stages of the TERA-Seq platform for RNA sequence processing [70]. The scripts exhibit branching and merging of processing flows (with *fan-out/fan-in* structures), work-queue parallelism, and intensive access to large files, with total input data size of 114GB [83].

`ci-cd` includes 21 software build programs, such as build scripts for software such as Lua, Memcached, Redis, and SQLite [10], as well as the test suite of the `makeself` tool [59]. These programs are used in continuous integration workflows and are characterized by heavy dependencies and complex control flow, but a limited volume of input data.

`covid` contains five programs that calculate public transit statistics during the pandemic period [66]. The scripts are available in two versions: one based on the composition of classic Unix tools like `cut`, `sort`, and `uniq`, and one implemented as a monolithic `awk` program. The input data consists of 5.08GB of CSV files.

`file-mod` includes five file transformation programs, such as compression, encryption, and format conversion. Two of them process network log files using `openssl` [55, 84], while the rest perform media conversions on hundreds of image and audio files [1, 4], with a total data size of 39.2GB.

`inference` consists of three programs that perform inference tasks using large foundation models [65, 85, 86]. Examples include image captioning [71], playlist generation [64], and hieroglyphic image classification [65]. The programs function as wrappers for external machine learning tasks [87, 88, 89], processing a total of 11.7GB of data.

`ml` corresponds to a typical example of machine learning workflows, where a monolithic Python program has been decomposed into multiple shell stages (data ingestion, training, classification, and evaluation), allowing the study of modular workflows [72], with input data amounting to 15.0GB.

`nlp` includes 23 natural language processing scripts from the Unix for Poets tutorial [61]. Most of these consist of one or two lines and can be combined into larger pipelines, operating on a collection of over 115,000 books [90].

`oneliners` includes 11 pipeline chains originating from both the classic Unix literature and modern sources [19, 61, 63, 67, 73, 74, 75]. The scripts use complex flow operators such as map, filter, and reduction, and highlight the shell's composition philosophy, applied to data with a total size of 13.5GB.

`pkg` consists of two programs that automate package installation and software permissions analysis [76, 91, 92]. Executions include downloading, building, and analyzing hundreds of packages for the first program and thousands for the second.

`repl` includes two independent programs, one for security auditing and one for reproducing software development in a Git repository. Both emphasize complex interaction with the file system.

`unixfun` contains 36 programs from the Unix50 challenge [62], which mimic classic text processing computations on a total data volume of 59.1GB.

`weather` consists of two programs that compute statistical metrics on historical temperature data, based on an example from [58]. The first program is similar to the example from the previous chapter (Listing 4.1) while the second performs additional analysis and reproduces the well-known weather chart by Edward Tufte [93]. Some of their processing phases conceptually correspond to MapReduce and Spark computations, highlighting large-scale data processing scenarios. The data originates from NOAA [57] and covers multiple years of historical measurements, with a total size of 146GB.

```
sample-benchmark
├── scripts      # Benchmark scripts
│   ├── a.sh
│   ├── b.sh
│   └── ...sh
├── install.sh   # Dependency installation
├── fetch.sh     # Input data download/preparation
├── execute.sh   # Script execution
├── validate.sh  # Hash generation & verification
└── clean.sh     # Input and output file cleanup
```

Figure 5.1: Benchmark Interface Architecture. The workflow consists of five modular control scripts. The main driver `main.sh` orchestrates the execution, ensuring efficient resource management and data integrity.

Finally, `web-search` implements a complete web search pipeline (`crawl`, `index`, `query`) using exclusively shell tools and advanced data flow operators on files totaling 8.61GB.

5.3 Infrastructure and Execution Configuration

The architecture of KOALA is based on a standardized interface that governs the lifecycle of each program. This approach ensures deterministic measurements and uniformity across heterogeneous workloads, explicitly defining the preparation, execution, and verification phases.

This specification is implemented via five infrastructure scripts (Fig. 5.1). Specifically:

1. `install.sh` installs the necessary software dependencies.
2. `fetch.sh` handles downloading or generating the input data and accepts an argument specifying their size (`--min`, `--small`, `--full`).
3. `execute.sh` runs the programs of the set and collects basic metrics, such as execution time and resource consumption.
4. `validate.sh` verifies the correctness of the output by comparing the generated results with predefined reference values via hash computation.
5. `clean.sh` removes temporary files and generated results.

Indicatively, the appendix lists the infrastructure scripts for `weather` (Listings A.1 to A.5).

KOALA also includes a central execution driver, `main.sh`, which sequentially calls the aforementioned scripts for one or more benchmarks. The driver accepts parameters either via flags or environment variables and forwards them to the individual scripts. For instance, the `KOALA_SHELL` variable defines the shell interpreter to be used (e.g., `bash`, `zsh`, or a specific executable path), defaulting to `sh`.

Furthermore, flags are provided for specialized data collection and execution control. The `--resources` flag activates the recording of dynamic analysis metrics—such as execution time, memory consumption, and input/output (I/O) activity intensity—which are detailed in a subsequent chapter. For simple timing, the `--time` (or `-t`) flag is available. Finally, via the `--scripts` (or `-s`) option, followed by the names of the desired scripts, the user can selectively execute specific scripts from a benchmark set. A user can run the `temp-analytics.sh` program (which resembles Listing 4.1) from `weather`, using `main.sh` and the small data scale as follows:

```
./main.sh --small weather --scripts temp-analytics
```

Table 5.2: Estimated integration time per benchmark set. The table presents the estimated integration time (in hours) for each benchmark subset of the KOALA suite. Estimates are based on the researchers' experience during the integration process and include adapting the benchmarks and preparing the infrastructure scripts.

Υποσύνολο Μετροπρογραμμάτων	$\sim T$ (ώρες)
`analytics`	40
`bio`	60
`ci-cd`	75
`covid`	65
`file-mod`	60
`ml`	30
`inference`	35
`nlp`	10
`oneliners`	10
`pkg`	30
`repl`	25
`unixfun`	10
`weather`	30
`web-search`	40
Σύνολο	520

The design of the infrastructure aims at the rapid collection of preliminary results and facilitating comparisons between systems. It deliberately remains lightweight and extensible, allowing users to adapt it to their specific needs.

Execution in an Isolated Environment

The KOALA suite provides optional support for execution in an isolated environment via Docker. A `Dockerfile` is provided which creates a Debian-based environment and installs the suite's base dependencies. Dynamic generation of images specifically tailored to each benchmark set is also supported, with a built-in call to `main.sh`. The use of containers allows reproducible experiments without modifying the user's system. The suite avoids using privileged commands, restricting the requirement for administrator rights only to where it is absolutely necessary.

Integrating a New Benchmark

Adding a new benchmark to KOALA requires adapting the five base infrastructure scripts. Specifically, it includes: (1) defining dependencies in `install.sh`, (2) preparing multi-sized data in `fetch.sh`, (3) automated execution in `execute.sh`, (4) defining validation logic in `validate.sh`, and (5) cleaning up temporary files in `clean.sh`.

The required integration time ranged from 10 to 80 person-hours per subset. Small and self-contained sets like `oneliners`, `unixfun`, and `nlp` were integrated in about 10–12 hours. In contrast, complex or heavily data-dependent sets like `file-mod`, `bio`, and `ci-cd` demanded significantly more effort due to size, dependencies, and strict correctness requirements. A significant portion of the time was devoted to data adaptation and ensuring correctness under different execution environments, a fact that highlights the importance of a reusable and standardized infrastructure.

5.4 Summary Characterization of the Suite

To substantiate the suitability and representativeness of KOALA as a performance evaluation tool, an extensive characterization of its benchmarks was performed. This characterization highlights the inherent diversity of the suite, approaching the scripts from both a static (syntactic) and dynamic (runtime behavior) perspective. Elements of the analysis are presented in Table 5.1, while detailed charts and the full statistical analysis of the following metrics are omitted for brevity and are available in the suite's main publication [16].

Static and Syntactic Characterization The extraction of syntactic features was performed via the `libdash` library [94], which utilizes Smoosh's abstract syntax definition [8]. The analysis demonstrates intense syntactic diversity among computational domains: while tasks like natural language processing (`nlp`) and data analysis (`covid`) rely primarily on extensive, linear pipelines, domains like infrastructure management (`ci-cd`, `pkg`) make heavy use of complex control structures (`if`, `while`, `case`), subshells, and functions. Moreover, individual scripts (e.g., `bio`, `web-search`) incorporate explicit manual parallelization constructs (`&`, `wait`). Regarding command distribution, although basic UNIX tools dominate in frequency (e.g., `echo`, `cat`, `grep`), 54.4% of the total unique commands in the suite correspond to custom binaries or local functions. This feature confirms the shell's role as "glue code" and highlights the primary obstacle for developing static acceleration systems, as they are called upon to manage the bulk of the computation as opaque "black boxes".

Dynamic Execution Profile Because static analysis does not capture true resource requirements, extensive dynamic characterization was performed via kernel-level monitoring (utilizing the `/proc` virtual file system, the `strace` tool, and continuous sampling via `psutil`). The analysis revealed immense diversity, confirming that the suite tests the limits of all computational subsystems. Specifically, actual execution time ranges from a few seconds (e.g., `unixfun`) to several hours on full datasets (e.g., `bio`, `web-search`). A crucial finding concerns the distribution of execution time: while in some scripts the shell consumes significant computational time, in heavier workloads execution is almost entirely dominated by external binaries, making parallelization the only viable acceleration strategy.

The heterogeneity of the suite is equally reflected in memory usage and input/output (I/O) activity. The memory footprint spans five orders of magnitude: from extremely lightweight streaming processes requiring less than 1 MB, to model loading and genomic structure maintenance tasks that allocate tens of Gigabytes. Correspondingly, the I/O profile varies radically, including both purely computational (CPU-bound) programs with minimal disk usage, and bulk file scanning tasks (I/O-bound) with throughput rates approaching 1 GB/s. Finally, the degree of interaction with the operating system kernel ranges from 10 to nearly 900 simultaneously open file descriptors and reaches up to hundreds of millions of system calls, highlighting the high cost of inter-process communication (IPC) via anonymous pipes.

Principal Component Analysis (PCA) To holistically capture the distinctiveness and diversity of the suite, principal component analysis (PCA) [95] was applied to two independent representations of the benchmarks. This method reduces data dimensionality while preserving and highlighting fundamental structural differences among the programs. In the first approach, the analysis was based on a combination of static and dynamic execution characteristics, proving that the scripts scatter across a particularly broad computational space with varied syntactic and behavioral profiles. In the second approach, source code embeddings generated by OpenAI's pre-trained `text-embedding-3-large` model [96] were utilized, aiming to capture high-level information regarding the structure and semantics of the programs [97, 98]. The results of both multidimensional analyses converge on the same strong conclusion: the KOALA suite covers an extremely broad spectrum of syntactic, semantic, and dynamic patterns, providing a fully representative set of realistic workloads.

5.5 Summary

Based on the design, implementation, and extensive characterization that preceded, the KOALA suite constitutes a realistic, diverse, and fully automated tool for evaluating shell programs. Its ability to cover a wide range of syntactic patterns and dynamic behaviors makes it a controlled and reliable testbed for studying both existing and future acceleration systems.

Having established this foundation, the remainder of the work focuses on the practical utilization of the suite. In the next five chapters, KOALA is used to systematically evaluate five different optimization approaches, which are based on different strategies:

1. The alternative interpreter `zsh` (Chapter 6).
2. The static I/O optimization compiler Shark (Chapter 7).
3. The manual parallelization tool GNU `parallel` (Chapter 8).
4. The dynamic pipeline parallelization system PASH (Chapter 9).
5. The speculative out-of-order execution architecture *hS* (Chapter 10).

The evaluation of each system is not limited to measuring final performance, but equal emphasis is placed on understanding its operating mode, as well as the adoption effort—namely, the level of manual intervention, syntactic modifications, or addition of annotations required to execute realistic usage scenarios.

The analysis begins in the next chapter by executing the suite through `zsh`, aiming to empirically investigate whether simply replacing the interpreter—without any code modification—can affect the performance of representative workloads.

Chapter 6

Evaluation of the Alternative Interpreter `zsh`

This chapter begins the cycle of experimental evaluations of the KOALA suite, examining the behavior of realistic workloads when executed by an alternative shell interpreter. Specifically, it evaluates `zsh` [17] (Z Shell), an extremely popular and customizable interpreter that is now the default interactive shell in environments such as macOS.

The primary goal of this evaluation is to empirically determine whether simply replacing the underlying execution engine—without applying complex static transformations or parallelization techniques—affects the correctness and execution speed of scripts written in the POSIX standard, compared to the established `bash`.

6.1 Operating Mechanism and Characteristics

Unlike systems that actively restructure code (such as Shark or GNU `parallel`, which will be analyzed in subsequent chapters), `zsh` executes the program *as is*. However, the way `zsh` parses and manages memory, as well as the implementation of its internal structures, differ from the corresponding architecture of `bash`. These interpreter-level differences can affect the execution speed of shell built-ins, the efficiency of variable assignment, and the processing speed of control structures (such as `for` and `while` loops). To ensure compatibility and correct execution of programs that follow the strict UNIX philosophy, `zsh` features a built-in emulation mode (`emulate sh`), which adjusts its semantics to the specifications of the POSIX standard.

6.2 The Motivating Example: The `weather` Script

The behavior of the `weather` script (Listing 4.1) serves as a useful case study. The benchmark is based on a central iterative loop (`for`) and variable expansion, before invoking three command pipelines (e.g., `awk`, `sort`) inside it.

Given that `zsh` executes this code without transformations, the completion speed of the script depends directly on the optimization of the interpreter's built-ins. The near-identical performance observed in `weather` empirically confirms that the implementation of `zsh`'s internal control structures and variable management is comparable to that of `bash`, and does not introduce a bottleneck in processing the loop.

6.3 Adoption Effort

As mentioned above, unlike manual parallelization methods, the adoption effort of `zsh` via the KOALA suite requires *no* code modification. Executing the suite's 126 scripts with the new interpreter was achieved by simply changing the `KOALA_SHELL` environment variable, which routes execution to a custom yet extremely simple startup script (Listing 6.1).

The use of the `source` command is critical, as it allows `zsh` to execute the benchmark code directly within the current environment, thus keeping KOALA's validation and metric collection flow intact.

```
#!/bin/zsh

emulate sh
source "$@"
```

Listing 6.1: zsh startup script in Koala. The script enables POSIX emulation mode (`emulate sh`) before loading and executing the benchmark code, ensuring compatibility.

6.4 Performance Analysis

Methodology To extract the experimental results, the benchmarks were executed using `zsh` version 5.9 under `sh` emulation mode. `bash` v5.2.21, also set to compatibility mode (`--posix`), was used as the strict baseline. Experiments were conducted in a controlled execution environment (AWS `c6i.4xlarge` instance), equipped with 32GB of memory and a 16-core processor at 3.5 GHz, running the Ubuntu 24.04.1 operating system. Measurements were taken on the small datasets (`--small`).

Correctness and Overall Speedup A key finding of the experimental process is that, under `zsh` execution, all 126 benchmarks of the suite completed successfully *without significant slowdown*.

Fig. 6.1 depicts the relative speedups of `zsh` compared to `bash`.

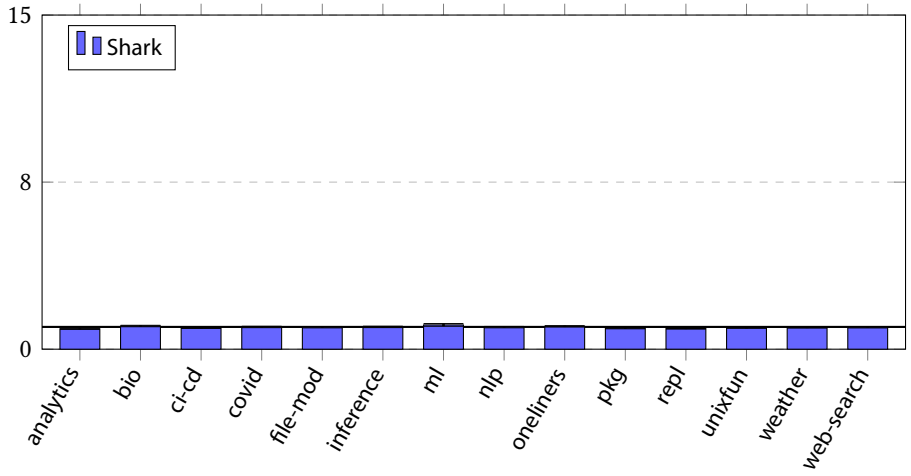


Figure 6.1: Relative speedups of `zsh` on the Koala suite benchmarks. The results confirm that the performance of `zsh` almost perfectly matches the performance of `bash` for the vast majority of the scripts.

As shown in the chart, performance remains practically unchanged for all benchmarks. Overall, the `zsh` interpreter recorded a marginal and statistically insignificant average slowdown of **0.16%** relative to `bash`. This result demonstrates that, despite their internal architectural differences, both interpreters manage system calls, data flows, and child process creation with identical efficiency.

6.5 Summary

The evaluation of `zsh` via Koala draws a clear conclusion: the transition from `bash` to `zsh` (under `sh` emulation) is achieved with absolutely no modifications to the source code and does not lead to measurable slowdown during the execution of realistic shell programs. Consequently, users who prefer `zsh` for its advanced interactive features can execute large-scale computational scripts while maintaining full system performance.

Nevertheless, simply replacing the execution engine does not alter the fundamental bottlenecks of shell programs, such as serial execution and the extensive, often unnecessary, use of the file system. To achieve substantial speedups, restructuring the code itself is usually required. Starting from this realization, the next chapter moves from passive execution to active optimization, examining the Shark system. It analyzes how static analysis and syntactic transformation *prior* to execution can eliminate unnecessary I/O and uncover hidden parallelism, while simultaneously increasing the adoption effort.

Chapter 7

Evaluation of Shark: Static Input/Output Optimization

This chapter analyzes and evaluates the strategy of static optimization, using the philosophy of the Shark system [15] as a reference point. Unlike the alternative interpreter `zsh` examined in the previous chapter, the Shark approach requires extensive syntactic transformation of the source code *prior* to its execution. It focuses primarily on reducing inter-process communication costs and eliminating redundant operations in the file system (I/O).

7.1 Operating Mechanism and Characteristics

Shark represents one of the first systems to treat shell programs as standard compilation objects, applying analysis and optimization techniques analogous to those of traditional programming languages. Its fundamental assumption is that file system accesses (writes and reads) can be considered analogous to variable references in other languages. This assumption allows the application of dependency analysis and data flow optimization techniques at the whole-program level.

Based on this approach, Shark implements a set of optimizations aimed at reducing file system usage (which is often a bottleneck) and improving data locality. Table 7.1 summarizes the four main categories of optimizations implemented by the system.

Optimization	Definition	Goal
Filesystem	Elimination of redundant file creation	Reduction of I/O and redundant computations
Pipelining	Conversion of file writes/reads into pipes	Reduction of I/O, improvement of locality & concurrency
Parallelization	Concurrent scheduling of independent commands	Increase in concurrency
Command Invocation	Program transformation using domain knowledge (e.g., cat-elimination)	Optimization of command usage

Table 7.1: The optimizations of Shark and their goals. Translation of the namesake table from [15].

To achieve these static transformations, Shark theoretically operates as an optimizing compiler through a strictly defined process (Fig. 7.1). First, it parses the code and constructs an abstract syntax tree (AST). Next, it normalizes the input/output operations by converting them into an intermediate representation (IO-Scripts), which explicitly captures the interactions with the file system. Leveraging this structure, it constructs a directed acyclic graph (DAG) of data dependencies. On this graph, it performs the removal of redundant I/O operations, eliminating files that are created but never read. The remaining intermediate files are converted into anonymous pipes. Finally, the optimized graph is traversed breadth-first in order to identify independent nodes, which are converted back into an AST so that the corresponding commands are scheduled for concurrent execution.

Beyond syntactic transformation, the original system placed particular emphasis on reducing process creation costs by converting frequently used external commands (like `grep`) into dynamic libraries

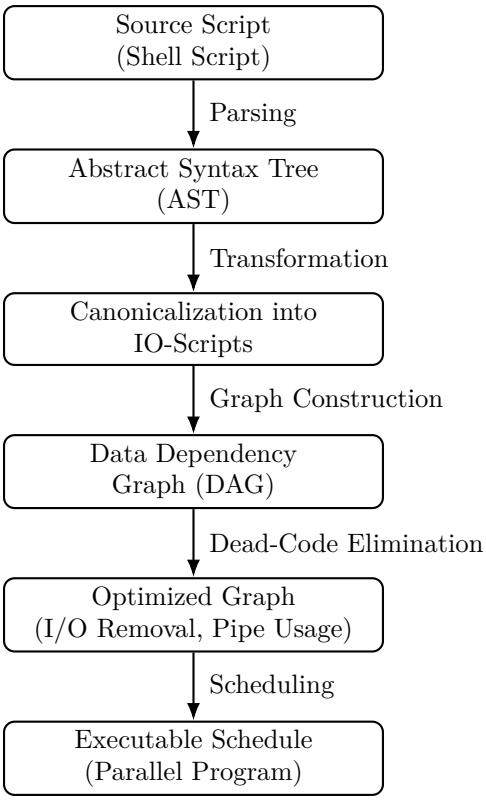


Figure 7.1: Overview of Shark. Visualization of the compilation and optimization pipeline. The process begins with the construction of the syntax tree and concludes with the generation of an optimized script through the analysis of dependency graphs.

(shared objects / DLLs). Although the current evaluation focuses exclusively on *static* transformations, this dynamic loading technique was a core pillar for the holistic acceleration of execution in the prototype system.

It is important to clarify that the prototype Shark system is a research project which is not available as open-source software or a functional tool. Consequently, in this work, the theoretical syntactic transformations described by the Shark philosophy are applied *manually* to all benchmarks in the Koala suite. Subsequently, both the original and optimized versions are executed in order to characterize performance and highlight the limits of the optimizations.

7.2 Static Transformation: The `weather` Script

To understand the practical application of these optimizations, the characteristic `weather` benchmark (Listing 4.1) is utilized, which consists of a `for` loop executing three independent pipelines. By manually applying the principles of Shark (Listing 7.1), two main interventions are implemented:

1. **Removal of redundant processes (Useless Use of Cat):** The redundant invocation of the `cat` command (which merely pipes the contents of a file) is eliminated, replacing it with direct input redirection (<). This saves the time of creating an additional process.

2. **Static Parallelization:** Exploiting the static knowledge that the three pipelines write to distinct output files and do not affect each other (i.e. there are no write/read dependencies), we place them in the background (via the `&` operator) and introduce the synchronization command `wait` at the end of each loop iteration.

This approach achieves concurrent execution at the command level, utilizing available cores and

```
#!/bin/bash

d="./data/temperatures"
for y in $(seq $start $end); do
    cut -c 89-92 < "$d/$y" | grep -v 999 | sort -rn | head -n 1 > "max.$y" &
    cut -c 89-92 < "$d/$y" | grep -v 999 | sort -n  | head -n 1 > "min.$y" &
    cut -c 89-92 < "$d/$y" | grep -v 999 | awk '{t+=$1; i++} END {print t/i}' > "avg.$y" &
    wait
done
```

Listing 7.1: The `weather` script transformed using Shark. The process required manual intervention to remove `cat` and add background parallelization operators.

```
    TEMPDIR=$(mktemp -d)
    cat > ${TEMPDIR}/${input}.words
    tail +2 ${TEMPDIR}/${input}.words > ${TEMPDIR}/${input}.nextwords
    tail +3 ${TEMPDIR}/${input}.words > ${TEMPDIR}/${input}.nextwords2
    paste ${TEMPDIR}/${input}.words \${TEMPDIR}/${input}.nextwords ${TEMPDIR}/${input}.nextwords2 |
    sort | uniq -c
    rm -rf ${TEMPDIR}
```

Listing 7.2: Excerpt from the `count-trigrams` script of the `nlp` set. Use of temporary files to store and process intermediate data flows before merging them.

drastically reducing the overall completion time of the loop. [1]

7.3 Adoption Effort

While the `weather` example required mild and comprehensible interventions, the manual application of the Shark philosophy to broader and more complex data analysis scripts proved to be a time-consuming and complicated process. A characteristic example is the trigram extraction benchmark (`count-trigrams`) from `nlp`. It is noted that the entire source code of the original and transformed scripts is provided verbatim in the appendix (Listings C.1 and C.2).

In its original form (Listing 7.2), the algorithm follows a classic and safe "temporary file creation" (tempfiles) approach to store intermediate states of the data flow.

To achieve Shark's goals—namely, eliminating disk writes by replacing intermediate files with named pipes (FIFOs) and parallelizing independent commands—extensive modification of the code structure was required, as shown in the following excerpt (Listing 7.3).

This conversion highlights three main challenges that make the adoption effort particularly demanding for the average user, or even for automatic compilation systems without strong safety guarantees:

1. **Flow Complexity Management and Deadlocks:** The use of FIFOs mandates the use of the `tee` command (line 21) to split the input into multiple channels. This immediately introduces the classic Producer-Consumer synchronization problem. Consumer commands (e.g., `tail` and `paste`

[1] Note that, since the loop iterations are also independent, iteration-level parallelization could also be applied by placing the entire loop in the background, as will be done later using GNU `parallel` in Chapter 8.

```
14      mkfifo "$f_raw1" "$f_raw2" "$f_raw3" "$f_tail2" "$f_tail3"
15      tail -n +2 < "$f_raw2" > "$f_tail2" &
16      tail -n +3 < "$f_raw3" > "$f_tail3" &
17      paste "$f_raw1" "$f_tail2" "$f_tail3" | sort | uniq -c > "$output_file" &
18      agg_pid=$!
19      tr -c 'A-Za-z' '\n' < "$input_file" | \
20      grep -v '^[[:space:]]*$' | \
21      tee "$f_raw2" "$f_raw3" > "$f_raw1"
22      wait $agg_pid
```

Listing 7.3: Excerpt from the `count-trigrams` script transformed using Shark. Replacement of temporary files with FIFOs and careful background initiation of consumers to avoid deadlocks.

on lines 14-16) must start in the background (`&`) *before* the producer (`tee`) begins writing to the FIFOs. If the order is reversed, the shell blocks waiting for a reader, leading to a permanent deadlock.

2. **Manual Resource Control:** As Shark relies on static interventions, it does not offer an automatic load balancing mechanism. To avoid exhausting the system's available processes (fork bomb) when processing thousands of files, the manual implementation of a job management mechanism, with `wait` and semaphores, was required, which is presented in detail in the appendix (Listing C.2).

3. **Correctness Challenge and Code Bloat:** Code readability is noticeably reduced. The actual business logic is overshadowed by complex process management details, while the risk of introducing race conditions is high, significantly increasing development and debugging time.

7.4 Performance Analysis

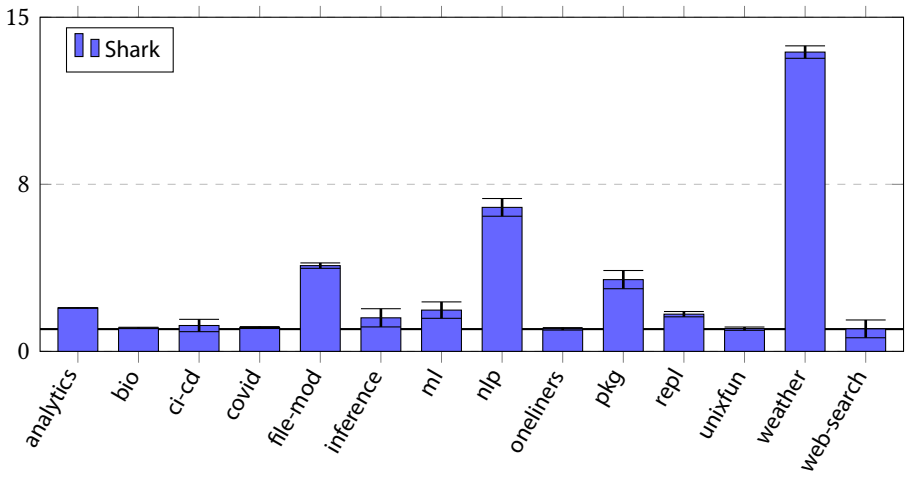


Figure 7.2: Relative speedups of Shark on the KOALA suite benchmarks. The system achieves significant speedups in scenarios with loop parallelization, but lags in strictly serial workloads or interconnected pipelines.

As depicted in Fig. 7.2, the Shark strategy achieved significant performance gains across the entire KOALA suite, with speedups ranging from **1.01× to 13.43×**, depending on the characteristics of each script. The most spectacular improvements were observed in benchmarks containing trivially paralleliz-

able loop iterations. Indicatively, Shark improved the `weather` and `nlp` sets by **13.43×** and **6.46×** respectively, as independent tasks were scheduled concurrently, effectively utilizing available resources. Conversely, scenarios with strong interdependencies that already use pipelines extensively, such as `covid`, `oneliners`, `bio`, and `unixfun`, offer fewer opportunities for Shark's interventions, leading to marginal speedups that ranged between **1.01× and 1.06×**. Similarly, scenarios with strictly serial operations, like those in the `ci-cd` set, showed limited improvement (**1.16×**). The `web-search` benchmark presented the smallest improvement (**1.01×**), as its original implementation already executes three n-gram calculations in parallel, not allowing further parallelization. In the same context, optimizations focusing exclusively on command invocation (such as removing `cat` or adding the `--posix` flag) did not yield significant benefits: in scenarios where *only* such opportunities existed (like in `web-search`), the average speedup was limited. In contrast, optimizations that eliminate temporary files for intermediate storage can offer more substantial speedups, but, as analyzed in Section 7.3, they introduce significant complexity due to the need for strict coordination between producers and consumers.

It is worth underlining that the static nature of the transformation ensures that Shark offers exclusively speedups (even marginal ones of the order of 1.01×), without introducing any runtime overhead, unlike dynamic analysis tools. This occurs because the system's optimizations correspond to the application of established shell scripting best practices. Today, several of these tactics—such as eliminating the useless use of `cat`—have been widely incorporated as static analysis rules in modern code-checking tools (linters), like ShellCheck [99], confirming the timeless value of Shark's philosophy.

7.5 Summary

The results in Koala confirm that Shark's optimizations are particularly efficient in scenarios involving operations on multiple inputs and independent commands, offering speedups up to **13.43×**. However, they prove less effective for scripts that are I/O-bound or for scenarios that are not easily parallelizable due to strong dependencies. Concurrently, the evaluation shows that the significant adoption effort makes the strictly static approach less attractive for the average user seeking easy and safe acceleration.

In the next chapter, we examine the GNU `parallel` tool, which also requires manual intervention in the code, but increases the achievable performance through explicit data parallelization.

Chapter 8

Evaluation of GNU `parallel`: Manual Parallelism

This chapter examines the approach of explicit parallelism, using GNU `parallel` [18] as the primary evaluation vehicle. In contrast to the static analysis of Shark, which attempts to optimize code based on recognized input/output patterns, GNU `parallel` is a tool that provides full control to the programmer, while requiring their full intervention for restructuring the program logic.

8.1 Operating Mechanism and Characteristics

GNU `parallel` is one of the most widespread manual parallelization tools in the Unix ecosystem. Its functionality is provided through the `parallel` command, which offers the ability to split datasets and simultaneously execute multiple independent processes. Its original purpose was to function similarly to `xargs`, but with the ability to correctly handle spaces and quotes in arguments. The tool supports both data parallelism and task parallelism, while providing the capability to execute processes on remote nodes via secure connections. By default, GNU `parallel` executes one job per available core. One of its strongest features is the ability to read data from a pipe, split it into blocks, and direct each block to separate, concurrent commands. Unlike automatic parallelization systems, GNU `parallel` does not utilize any information regarding the semantics, the abstract syntax tree, or the dependencies of the program. The responsibility for correct data partitioning and maintaining functional correctness is transferred entirely to the user. Incorrect configurations can lead to both significant slowdowns and the production of erroneous results. GNU `parallel` is used in this work as a basic reference point, representing the user-level manual parallelization approach.

8.2 Manual Parallelization: The `weather` Scenario

To understand how the tool operates, the motivating example of the `weather` benchmark is utilized once again (Listing 4.1). For parallelization via GNU `parallel`, simply adding a flag to the shell is not sufficient. Instead, the program must be transformed (Listing 8.1). Specifically, the loop logic must be encapsulated in a discrete function (named `process_year` in the example). Subsequently, the year feed is usually implemented via a generator (such as the `seq` command), which pipes the data directly into `parallel`.

This approach, while efficient, converts a single structured script into a set of discrete functions.

8.3 Adoption Effort

The application of the tool to the Koala benchmarks was focused on scenarios that are natural choices for parallel execution due to their structure. Emphasis was placed on flows that process independent input files, and on efficient segment-based processing that requires little or no synchronization, often utilizing the `-pipe` parameter for data flow parallelization. Despite this selective use, the correct application of GNU `parallel` proves in practice to be an extremely demanding non-trivial process [6], and the conversion of serial pipelines to parallel ones often requires multiple iterations to achieve correctness and optimal performance.

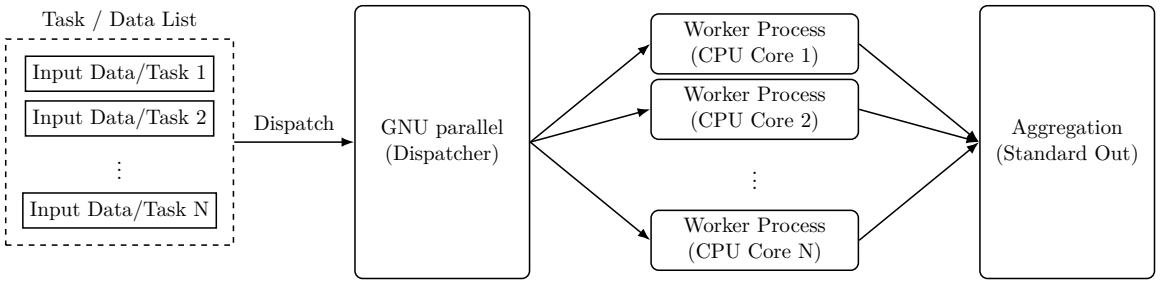


Figure 8.1: Overview of GNU `parallel`. The dispatcher receives a list of tasks or data and assigns them dynamically to multiple worker processes running in parallel on different processor cores. Individual results are collected and aggregated to the standard output.

```
#!/bin/bash

d="./data/temperatures"
process_year() {
    y=$1
    cut -c 89-92 "$d/$y" | grep -v 999 | sort -rn | head -n 1 > "max.$y"
    cut -c 89-92 "$d/$y" | grep -v 999 | sort -n  | head -n 1 > "min.$y"
    cut -c 89-92 "$d/$y" | grep -v 999 | awk '{t+=$1; i++} END{print t/i}' >
     ↪  "avg.$y"
}
export -f process_year
export d
seq "$start" "$end" | parallel --jobs "$(nproc)" process_year
```

Listing 8.1: The `weather` scenario transformed using GNU `parallel`. The process requires a radical modification of the code architecture, function extraction, and explicit invocation of the parallelizer.

Environment Export and Syntactic Complexity GNU `parallel` starts new shell processes (`bash -c`) in the background. For local functions to be available in these subshells, the user must explicitly export them to the environment using the `export -f` command. Furthermore, managing variables and strings within the execution parameter requires complex escaping patterns, making debugging particularly difficult.

Intermediate Data and Pipeline Management In contrast to the simple syntax of shell pipes (`|`), the user of GNU `parallel` must often explicitly manage the data flow. In the example of the `covid-1` script from the `covid` set, there is a need to use temporary directories (`mktemp`), define data chunk sizes via the `-block` (`chunk_size`) parameter, and use cleanup mechanisms (`trap`) for temporary files. The simple pipeline of the original script (Listing 8.2) is transformed into a complex script (Listing 8.3) that requires function extraction and explicit coordination via the `-pipe` parameter.

Syntactic Complexity and Correctness To move towards more intensive parallelization, when parallelism is not obvious, a new challenge often arises, especially in scenarios requiring synchronization or data order preservation. For example, when parallelizing the last pipeline stages of the `spell` script (Listing 8.4), the need for order preservation arises (so that commands like `uniq` function correctly). Thus, the use of the `-k` (*keep order*) flag is mandated [6], as its omission leads to incorrect results due to the non-deterministic scheduling of tasks.

```
cat "$1" |
  sed 's/T..:..:..//' |
  cut -d ',' -f 1,3 |
```

Listing 8.2: Excerpt from the `covid-1` script of the `covid` set. The script contains a pipeline with easily parallelizable initial stages. The full code is provided in the appendix (Listing C.3).

```
chunk_size=${chunk_size:-100M}
process_chunk() {
  sed 's/T..:..:..//'| cut -d ',' -f 1,3
}
export -f process_chunk
tmp_dir=$(mktemp -d)
trap "rm -rf $tmp_dir" EXIT
cat "$INPUT" | parallel --pipe --block "$chunk_size" -j "$MAX_PROCS" process_chunk
 ↪  > "$tmp_dir/combined.tmp"
```

Listing 8.3: Excerpt from the `covid-1` script transformed using GNU `parallel`. Parallelization requires manual data chunking. The full code is provided in the appendix (Listing C.4).

Furthermore, managing multiple input/output flows often requires the manual creation of named pipes (`mkfifo`) and complex command structures involving escape characters. As shown in Listing 8.5, the command becomes difficult to read, while performance depends dramatically on "magic" constants such as `--block`, where incorrect tuning can lead to dramatic delays [6].

8.4 Performance Analysis

Overall, GNU `parallel` achieved an **average speedup of approximately 2.6×** across the suite, with the variance between scripts being high, but with results remaining positive for almost all benchmarks.

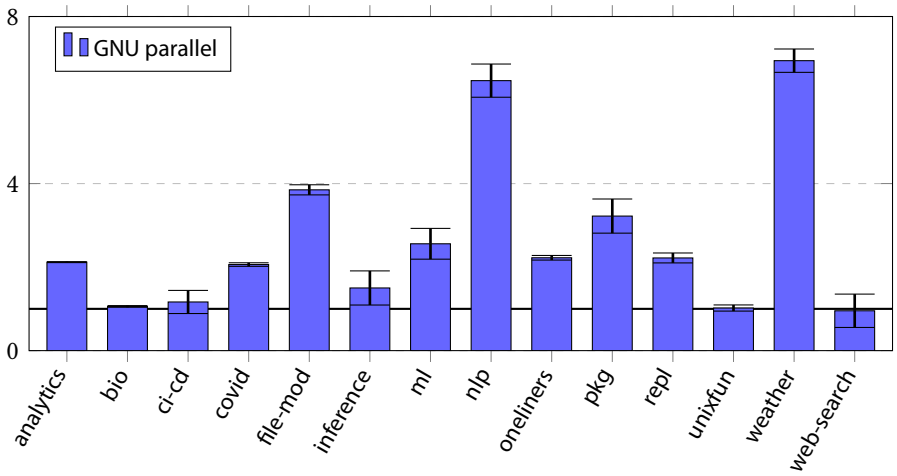


Figure 8.2: Relative speedups of GNU `parallel` on the KOALA suite benchmarks. The tool offers top-tier speedups in perfectly parallelizable scenarios but imposes overhead on small workloads.

As shown in Fig. 8.2, the reward for the high adoption effort is impressive in scenarios that are I/O-

```
#!/bin/bash
# Calculate mispelled words in an input

dict=$SUITE_DIR/inputs/dict.txt

cat $1 |
    sed 's/[^[:print:]]//g' |      # remove non-printing characters
    col -bx |                      # remove backspaces / linefeeds
    tr -cs A-Za-z '\n' |           # split on non-alphabetic characters
    tr A-Z a-z |                   # map upper to lower case
    tr -d '[:punct:]' |            # remove punctuation
    sort |                         # put words in alphabetical order
    uniq |                         # remove duplicate words
    comm -23 - $dict               # report words not in dictionary
```

Listing 8.4: The `spell` script of the `oneliners` set. The initial stages of the pipeline are easily parallelizable, while the last three require careful management.

```
mkfifo $TEMP1
parallel "cat {} | col -bx | tr -cs A-Za-z '\n' | tr A-Z a-z | \
tr -d '[:punct:]' | sort > $TEMP_C1" ::: $IN &
sort -m $TEMP1 | parallel -k --jobs ${JOBS} --pipe --block "$BLOCK_SIZE" "uniq" |
uniq | parallel -k --jobs ${JOBS} --pipe --block "$BLOCK_SIZE" "grep -vx -f $dict
 ↪ -"
```

Listing 8.5: Excerpt from the `spell` script transformed using GNU `parallel`. FIFOs and the use of the `-k` flag to preserve output order are required, ensuring correctness. The full code is provided in the appendix (Listing C.5).

bound or easily parallelizable. For example, **`file-mod`**, which simultaneously converts multiple media files with high utilization of available cores, was accelerated by **3.85×**. Similarly, **`nlp`**, which processes multiple data files independently, recorded a speedup of **6.46×**.

Conversely, the speedups of GNU `parallel` are limited in scenarios lacking these characteristics. In **`ci-cd`** (**1.16×**), the improvement was minimal, as these scripts either already exploit internal parallelism during command invocation (e.g., multi-file compilation) or include commands that do not benefit from the `parallel` model (such as `git` or `find`).

Finally, there were cases where the tool did not yield substantial improvement or even caused a slight slowdown due to the nature of the script. In the **`unixfun`** (**1.02×**) benchmark, the pipeline stages depend directly on the output of the preceding stages. This interdependence prevented GNU `parallel` from fully exploiting parallelism. Similarly, scripts requiring complex splitting and merging, such as **`web-search`** (**0.95×**), drastically limited the tool's capabilities to exploit parallelism, recording a slowdown.

8.5 Summary

GNU `parallel` represents an approach based on direct, explicit, manual parallelization in the Unix ecosystem. Evaluation in Koala confirms that it can significantly accelerate I/O-bound tasks and loops without interdependencies (such as `file-mod`), offering an average speedup of **2.6×**. However, its per-

formance is significantly limited in scenarios with serial operations or complex splitting and merging requirements (e.g., `unixfun`, `web-search`).

Furthermore, the price of this performance is the significant adoption cost. The need for code restructuring, function extraction, manual temporary file management, and careful flag placement (such as `-k`), shifts the burden of correctness entirely to the programmer, making it less suitable for the automated optimization of large-scale software.

These findings demonstrate a clear gap: the community needs systems that offer the impressive speedups of GNU `parallel` but with the safety and ease of use of a simple interpreter (as seen with `zsh`). The next chapter introduces the concept of automatic parallelization via the PaSh system, which attempts to bridge this gap by offering safe, automatic data parallelism with minimal user intervention.

Chapter 9

Evaluation of PaSh: Automatic Dynamic Parallelization

The previous chapters highlighted that while manual parallelization (GNU `parallel`) and static optimization (Shark) can yield significant performance gains, they require radical interventions in the source code and shift the burden of correctness to the programmer. This chapter examines a modern, alternative approach through PaSh, which attempts to provide automated acceleration while preserving the program's original semantics.

9.1 Operating Mechanism and Characteristics

PaSh [7] is a system for the automatic parallelization of POSIX shell programs, which executes Unix programs in a parallel fashion via a Just-in-Time (JIT) compilation mechanism. To address the dynamic nature of the shell, PaSh interleaves compilation and execution phases, collecting real-time information about the state of the shell, the file system, and the environment.

The system (Fig. 9.1) identifies potentially parallelizable regions of the program and converts them into data-flow graphs, where each command corresponds to a node and each data flow to an edge. Parallelizability information is encoded through a system of *annotations*, which describe the parallelization categories of commands and takes into account their arguments and options. Subsequently, transformations are applied to these graphs to identify data and task parallelism, such as splitting flows into independent segments and reorganizing commands. After completing the transformations, the graphs are converted back into shell code, into which appropriate parallelization commands (e.g., `&`, `wait`) and named pipes are inserted to explicitly implement parallel execution. The degree of parallelism is configurable and adjusted dynamically using the `-width` flag. To ensure functional equivalence with serial execution, PaSh relies on an order-aware dataflow model [6], which maintains the necessary dependencies between inputs and outputs. Furthermore, before each transformation, the shell state is saved to avoid side effects. In cases where dynamic compilation fails or parallelization is deemed unsafe (e.g., unknown commands), PaSh automatically falls back to serial execution.

9.2 Just-in-Time Compilation: The `weather` Scenario

Returning to the example, although the user executes the original `weather` script (Listing 4.1) unchanged, PaSh intervenes dynamically during its execution. By reading the available annotations, the system recognizes that the `cut` command in the first pipeline is fully parallelizable (stateless), while the `sort` command requires a specialized merge operation (`merge`). Based on this information, PaSh constructs and executes internally a data-flow graph: it automatically partitions the input (`split`), directs it into multiple named pipes (FIFOs) corresponding to the parallelization width, processes the data in parallel, and finally merges the individual results (utilizing `sort -m`).

Below is a simplified representation of the shell code generated automatically by PaSh to coordinate these functions (Listing 9.1), while the full program is provided, for completeness, in the appendix (Listing B.1).

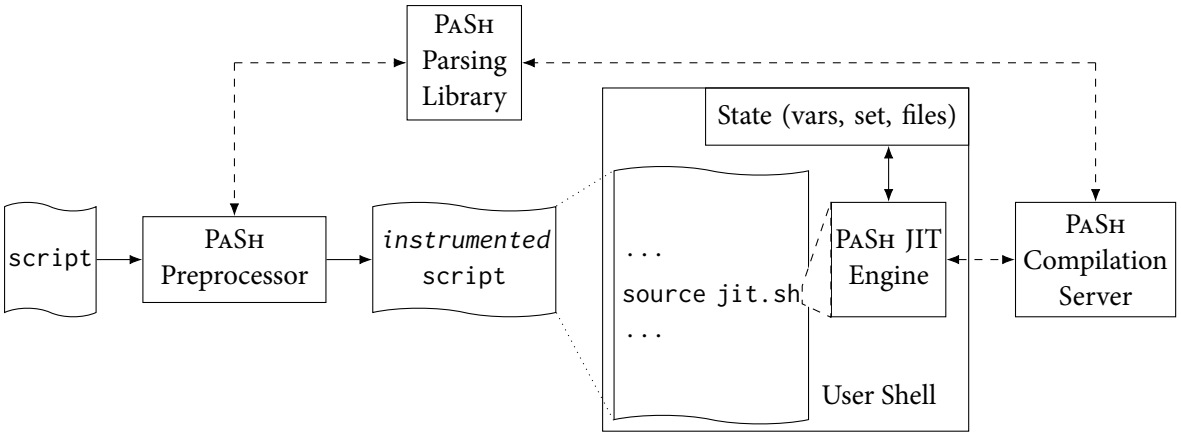


Figure 9.1: Overview of PaSh. The PaSh system orchestrates scripts with calls to the JIT engine, which during execution forwards parts of the program to the PaSh compilation server for dynamic processing at runtime.

```
#!/bin/bash

mkfifo f{0..10}
cat "./data/temperatures/2000" >f0 &
split f0 f1 f2 &
cut -c 89-92 <f1 >f3 &
cut -c 89-92 <f2 >f4 &
grep -v 999 <f3 >f5 &
grep -v 999 <f4 >f6 &
sort -rn <f5 >f7 &
sort -rn <f6 >f8 &
sort -rn -m f8 f9 >f10 &
head -n 1 <f10 >"max.2000" &
# ...
# --- Alternatively, for the AVG calculation (Merge -> Sequential Awk) ---
merge f9 10 | awk '{t+=$1; i++} END {if (i>0) print t/i}' > "avg.2000"
wait
rm f{0..10}
```

Listing 9.1: Simplified representation for the `weather` script transformed using PaSh. Shows the data-flow graph (DFG) that PaSh automatically creates and executes in the background for one of the script's pipelines.

9.3 Adoption Effort

In contrast to Shark and GNU `parallel`, the architecture of PaSh offers a *zero-effort* adoption experience from the user's perspective. The programmer is relieved of the need to modify the original code.

Evaluation via the Koala suite was performed requiring only the definition of the corresponding environment variable in the Koala driver. The parallelization degree for the experiments was 4, and the corresponding environment variable was set as follows:

```
KOALA_SHELL="./pa.sh --width 4"
```

It is noted that PaSh has the capability to substitute unknown code segments via `alias` or functions that can be manually annotated with parallelization information. While this practice could extract fur-

ther speedups, we decided *not* to apply it, as it would require significant additional workload, negating the system's primary advantage.

9.4 Performance Analysis

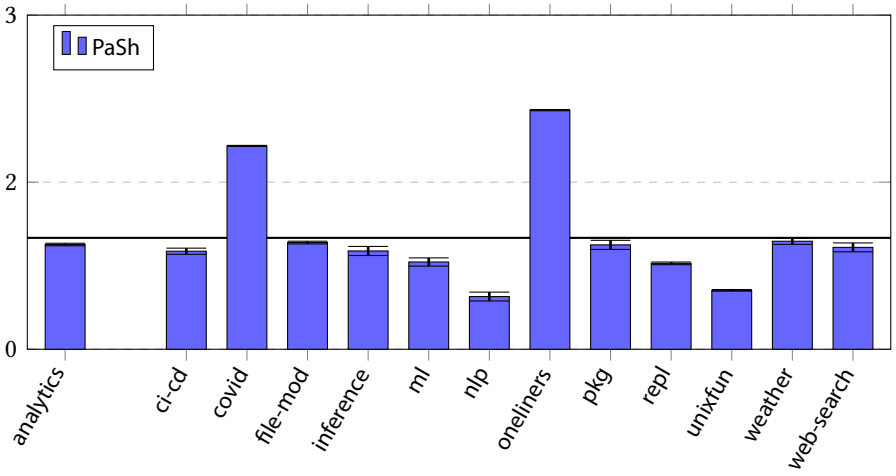


Figure 9.2: Relative speedups of PaSh on the Koala suite benchmarks. PaSh offers speedups on known tools but returns to serial execution when annotations or suitable syntactic structures are absent.

The performance results are illustrated in Fig. 9.2. PaSh achieved significant speedups in scenarios with multi-stage pipelines or `for` loops without data dependencies between iterations. Indicatively, in the **`oneliners`** and **`covid`** sets, it achieved speedups of **2.15×** and **1.83×** respectively, as they are based on classic POSIX tools included in the system's annotation library. However, evaluation via Koala highlighted critical limitations that drastically affect the generalization of its performance.

The most significant limitation lies in the annotation burden. In domains where the computational load is handled by specialized executables for which PaSh lacks annotations, the system does not apply parallelization. As a result, for sets such as **`pkg`** (**0.94×**) and **`file-mod`** (**0.96×**), no speedup is observed; instead, a slight slowdown is recorded due to the overhead of dynamic analysis.

Furthermore, the absence of suitable syntactic structures and the increased cost of compilation constitute additional limiting factors. PaSh presupposes the existence of specific syntactic patterns (e.g., large pipelines or data-parallel loops) to intervene. Scenarios such as **`repl`** (**0.77×**) and **`ci-cd`** (**0.88×**), which consist mainly of serial system commands and do not contain structures amenable to parallelization, are slowed down as additional overhead is introduced due to the JIT engine. Finally, it is worth noting that in the case of the **`bio`** benchmark set, execution failed completely, demonstrating the difficulties of dynamic compilation in managing extremely complex dependencies.

9.5 Summary

Evaluation in Koala demonstrates that PaSh can provide substantial speedups in scenarios that fall within its parallelization scope, in an automated and safe manner. However, its effectiveness depends directly on two main factors: the availability of annotations for the respective commands and the extent to which the program structure allows for the transformations the system can implement. When the system encounters opaque tools or strictly serial structures, the conservative fallback approach negates potential performance gains. This limitation remains a significant open issue in current shell research. To bypass the hurdle of manual annotations and opaque external commands, a different approach is required. The next and final chapter of the evaluation explores this direction through the *hS* system, which

attempts to offer general-purpose automatic acceleration by utilizing speculative out-of-order execution and dynamic tracing.

Chapter 10

Evaluation of *hS*: Speculative Out-of-Order Execution

The previous chapter demonstrated that automatic parallelization (via PASH) benefits from zero adoption effort but is drastically limited by the "annotation burden," failing to accelerate unknown, opaque tools. This chapter examines a new, alternative approach, *hS*, which attempts to offer general-purpose automatic acceleration by bypassing the need for additional information regarding the commands being executed.

10.1 Operating Mechanism and Characteristics

hS introduces a different approach to automatic acceleration based on out-of-order execution [14]. Instead of strictly following the program's syntactic order, it attempts to execute commands "opportunistically," i.e., by performing speculative execution, predicting that no dependency violations will occur. This approach is grounded in the observation that many shell programs are excessively serial, despite containing independent or weakly dependent commands. The system pre-processes the code and constructs a partial order graph that captures the syntactic sequence. Subsequently, during execution, it dynamically collects dependency information by tracing access to the file system and the environment.

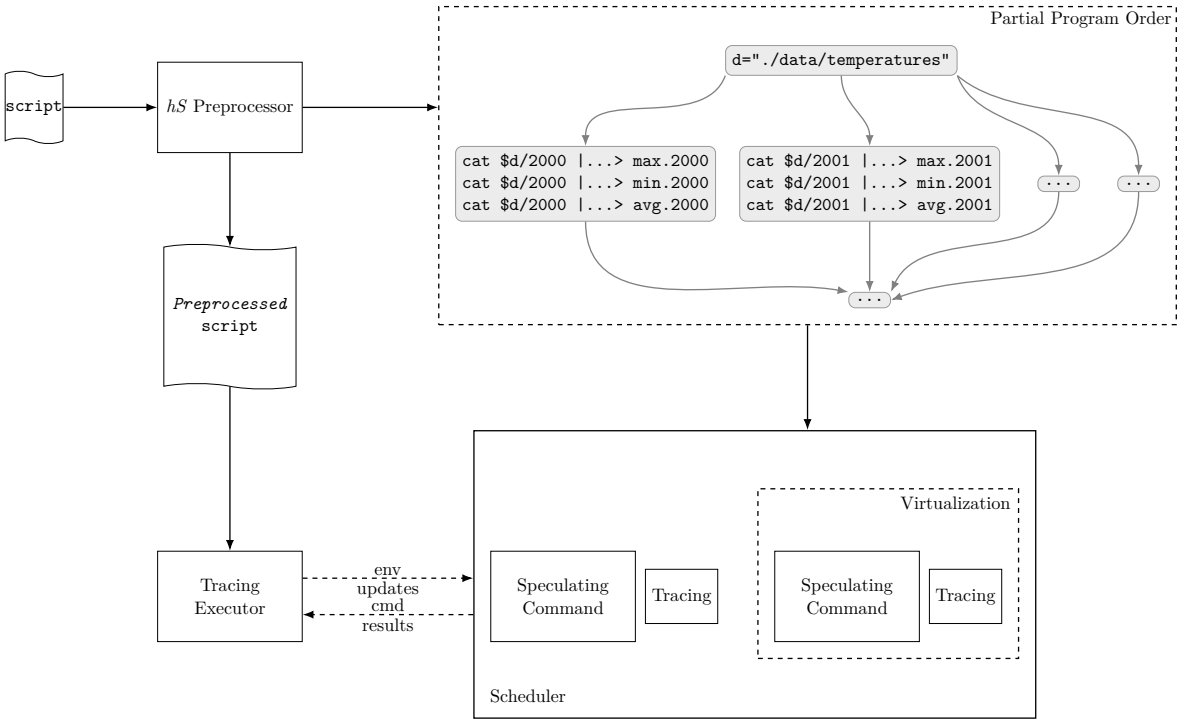


Figure 10.1: Overview of *hS*. The system executes future commands ahead of time (speculation) in isolated environments (sandboxes) and detects I/O accesses (using `ptrace`) and environment state. If no dependency conflicts occur, the results are incorporated into the file system (commit); otherwise, the execution is undone (rollback) and repeated [14].

As shown in Fig. 10.1, *hS* executes future commands in isolated environments (sandboxes), using virtualization mechanisms and overlay filesystems. The side effects of the commands are recorded temporarily. When a command completes, its results are merged into the main system (commit) only after confirming that no dependencies from previous commands were violated. In the event a conflict is detected, speculative executions are cancelled (rollback) and execution continues safely according to the correct semantic order.

The integrated scheduler alternates between execution and validation phases. This technique allows for the exploitation of task parallelism in opaque executables without prior specifications, making *hS* a general-purpose system.

10.2 Speculative Execution: The `weather` Scenario

To understand the practical application of speculative execution, the operation of the *hS* scheduler is examined in the `weather` scenario (Listing 4.1). In contrast to PaSh, which constructs static pipelines based on FIFOs, *hS* launches successive pipelines (pipelines H1, H2, H3 of the loop) within sandboxes *before* their previous dependencies have completed.

Table 10.1 details this internal behavior, assuming a starting year of 2000 (`start` = 2000).

Time Step	*hS* Scheduler Action in the Background
t_0	Start `H1(2000)`: Executes normally, as it is the first command in the sequence.
t_1	Speculative execution: Start of `H2(2000)` and `H3(2000)` in isolated environments. Their reads/writes are traced.
t_2	The shell proceeds to the next year (`y=2001`). Speculative `H1(2001)` begins while `H2/H3` of 2000 are still running.
t_3	`H1(2000)` completes successfully.
t_4	`H2(2000)` completes. The system checks I/O traces and finds no conflicts with H1 → Commit of its changes to the real file system.
t_5	`H3(2000)` completes. Similarly, no conflicts are detected → Commit.
t_6	The loop continues, having effectively overlapped the execution time of individual commands.

Table 10.1: Operation of the *hS* scheduler in the `weather` scenario. `H1`, `H2`, and `H3` correspond to the three independent pipelines within the `for` loop.

10.3 Adoption Effort

Just like PaSh, *hS does not require code modification*. However, *hS* extends this approach: the absence of annotations means that the user does not require knowledge of the internal workings of the commands they call. This is particularly critical in the case of custom executables, which are often treated as ”black boxes.” As revealed by the program analysis, these executables constitute the majority of commands appearing in the examined scenarios. Furthermore, in several cases, they fall into specialized computational fields that are not necessarily familiar to general programmers, such as the tools included in `bio` which belong to the field of bioinformatics. While in other systems the user would have to manually analyze the behavior of these programs (e.g., which files they read or modify), *hS* allows for their automatic parallel execution through speculative out-of-order execution. The system handles correctness by automatically performing a rollback of the execution only if a real dependency conflict is dynamically detected, without additional user intervention.

As *hS* is currently under active development, an early-stage prototype provided by its creators was used for this evaluation. For the execution of the Koala scripts, the benchmarks remained unchanged, with the only modification being the definition of the environment variable:

```
KOALA_SHELL="hs"
```

However, due to the experimental nature of the tool, it was not possible to successfully execute the entire Koala suite. For sets such as `analytics`, `bio`, `ml`, `nlp`, `unixfun`, `weather`, and `web-search`, *hS* successfully executed at least a subset of the scripts. Conversely, the subsets `ci-cd`, `file-mod`, `inference`, `oneliners`, `pkg`, and `repl` were completely excluded from the *hS* evaluation, as their execution either did not complete successfully or produced incorrect results due to the prototype's inability to handle their specific structures.

10.4 Performance Analysis

For the evaluation of the system, we focused on the benchmarks that were executed successfully. The results are depicted in Fig. 10.2.

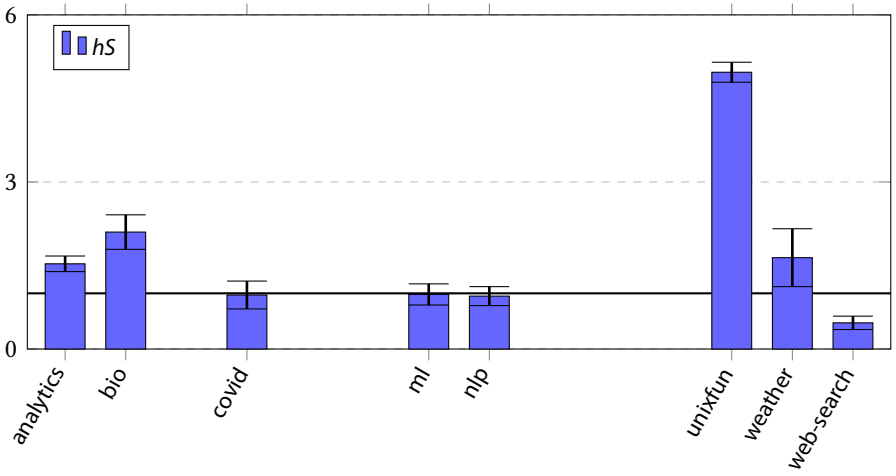


Figure 10.2: Relative speedups of *hS* on the Koala suite benchmarks. The system offers significant speedups when predictions are confirmed but is severely penalized by tracing costs in small processes.

The analysis highlighted that *hS* offers significant benefits in scenarios involving syntactic regions without interdependencies. Speedups range from **1.53× to 4.97×**, with the **`analytics`** and **`unixfun`** sets at the two ends of this range, respectively. Other benchmarks showing notable speedups include **`weather`** **(1.64×)** and **`bio`** **(2.10×)**. In these scripts, speculative commands operate on independent files, resulting in systematically confirmed scheduler predictions (high speculation hit rate), allowing the system to fully hide file system latencies.

On the other hand, scenarios involving direct dependencies between stages were unable to benefit from the speculative approach. These scenarios saw serious slowdowns, ranging from **0.97× (in `covid`)** to **0.47× (in `web-search`)**, resulting in an average slowdown of **0.72×** for these categories.

These slowdowns can be primarily attributed to the following factors:

1. **Frequent Rollbacks:** When scripts are characterized by strong, direct data dependencies, speculative execution proves ineffective, leading to costly cancellation and re-execution processes.

2. **Isolation & Tracing Overhead:** *hS* relies on `ptrace` for monitoring system calls and on overlay filesystems for isolation. In workloads with many fast, short-lived processes, the context-switching cost for tracing every I/O call dominates, degrading performance.

Finally, the approach is limited by commands with non-virtualizable side effects, such as network communication (`curl`, `wget`), which cannot be executed ahead of time and undone without consequences.

10.5 Summary

hS indicates that the principles of out-of-order execution can be successfully applied at the operating system and shell levels. Its ability to accelerate opaque programs without any user intervention is a significant development toward full automation. However, the findings of the Koala suite underscore that, in its current experimental form, the high computational overhead of monitoring mechanisms and the penalties from incorrect speculation are significant obstacles requiring further optimization for generalized use.

Chapter 11

Conclusions

The systematic analysis through the Koala suite highlights that shell program acceleration cannot be treated as a one-dimensional parallelism problem, but rather as a complex space of design choices that depends on the nature of the workloads, the semantics of the commands, and the structure of the scripts themselves. The combination of static and dynamic characterization of the benchmarks, as well as the application of multiple acceleration systems, provides a comprehensive picture of the shell optimization space.

11.1 Different Execution Strategies and Their Limits

The application of the `zsh`, Shark, GNU `parallel`, PaSh, and *hS* systems to the Koala suite shows that each approach is effective for different categories of programs and exploits different optimization opportunities.

- **Alternative interpreter (`zsh`):** Simply replacing the execution interpreter does not lead to a substantial change in script performance, which suggests that choosing a different interpreter does not introduce measurable overhead during the execution of shell programs. The increased customizability and advanced interactive features of `zsh` can be adopted without negatively affecting the performance of computational scripts.

- **Syntactic transformations (Shark):** Shark achieves significant speedups in scripts with independent iterations or command pipelines, reaching up to 13.43× in some cases. The benefits decrease when data pipelines are already optimized or when dependencies between stages limit the available transformations.

- **Manual parallelism (GNU `parallel`):** The use of GNU `parallel` performs particularly well in I/O-intensive tasks or in loops without interdependencies, with average speedups of approximately 2.6×. However, it requires manual modification of the programs, which limits the practical applicability of the approach.

- **Parallelization with command awareness (PaSh):** PaSh automatically accelerates scripts containing command pipelines; however, its effectiveness depends on the availability of annotations and on whether the program falls within the parallelization model it supports.

- **Out-of-order execution (*hS*):** *hS* offers benefits in cases with independent computation stages, but its performance is affected by isolation and tracing costs and the presence of dependencies, leading to instability or even slowdowns in some programs.

Overall, it appears that there is no universally optimal acceleration technique; each system exploits different features of the shell and exhibits different limitations.

11.2 Diversity of Behavior

Syntactic analysis shows that shell programs function primarily as an orchestration mechanism for external tools. Although the most frequent commands originate from POSIX tools and GNU Coreutils, 54% of the unique commands in the benchmarks correspond to custom binaries or user functions. This fact confirms that real-world shell programs function mainly as a connective mechanism that connects heterogeneous programs. This property complicates static optimization techniques, as the behavior of a large portion of the commands remains opaque to the acceleration system [16].

Dynamic characterization reveals that Koala workloads cover a wide range of behaviors in terms of execution time and resource usage.

- CPU usage time ranges from seconds to hours.
- Memory usage and total input/output volume differ by several orders of magnitude.
- Many programs are dominated by external commands, while others spend significant time within the shell interpreter itself.

This variety explains why techniques focusing exclusively on data parallelism do not perform uniformly across all benchmarks. The shell is used not only for data processing but also for the orchestration of complex system tasks.

11.3 The Value of Systematic Evaluation

The absence of established benchmarks has constituted a significant obstacle in shell acceleration research. The Koala suite provides:

- realistic workloads with real data,
- automated installation and result validation,
- reproducible evaluation infrastructure,
- broad coverage of syntactic and dynamic characteristics.

The application of the suite to different acceleration approaches highlighted the importance of evaluation under common and reproducible conditions, allowing a clear depiction of trade-offs between performance, implementation complexity, and adoption effort. Therefore, the existence of a systematic and commonly accepted suite is essential for fair comparisons and reliable progress in the field.

11.4 Overall Assessment

The analysis through the suite demonstrates that the Unix shell constitutes a particularly heterogeneous execution environment. The different workloads, the external commands, and the variety of dynamic behaviors make it difficult for a single acceleration strategy to dominate. The contribution of Koala lies in the fact that it enables, for the first time, a comparable and systematic evaluation of acceleration systems in realistic scenarios, highlighting both their capabilities and their limits.

Chapter 12

Future Work and Conclusions

This thesis concludes after presenting, analyzing, and utilizing the KOALA suite. Through the systematic study of the structural characteristics of the UNIX shell and the identification of the practical limitations of existing accelerators, it becomes clear what makes shell optimization such a complex problem with significant technical challenges. In this final chapter, the research directions for future work are outlined, and the final conclusions of the work are summarized.

12.1 Future Work

The infrastructure and findings of the present work open multiple directions for future research.

Expanding the Suite with New Workload Patterns

Although the KOALA suite already covers a wide range of applications, new categories of scripts could further enhance its representativeness:

- **Network-intensive interaction scenarios:** Scripts based on continuous communication with remote services or data synchronization would allow the study of systems where network latency dominates over local I/O.
- **Non-deterministic executions:** Scenarios that depend on randomness or the dynamic state of the system can highlight the limitations of parallel execution and out-of-order execution techniques, which rely on accurate predictions.
- **Interactive sessions:** Modeling actual interactive shell usage will allow for the study of response time (latency) and the overall quality of experience (QoE).

Evaluating New System Architectures

The variety of benchmarks makes KOALA ideal for evaluating new categories of systems, beyond local single-node acceleration:

- **Distributed shells:** Systems such as DISH [11] and POSH [1] allow the execution of shell programs across multiple nodes (clusters). Benchmarks with large data volumes offer a natural setting for evaluating the cost-benefit ratio of network distribution.
- **Fault tolerance and execution continuity:** Systems such as FRACTAL [12] introduce checkpointing and continuity mechanisms. Long-running benchmarks from the suite can be used to measure the actual overhead of these fault tolerance techniques.

Designing Next-Generation Optimizers

Findings from the application of the systems indicate key directions for next-generation optimizers:

- **Advanced static analysis:** Approaches such as the one presented in [100] aim at a better understanding of the semantics of shell scripts, which can reduce the need for annotations in dynamic systems such as PaSh.
- **Reducing monitoring costs:** Experience from *hS* suggests the need to move tracing from the user level (e.g., FUSE / ptrace) directly to the kernel level (e.g., via eBPF), reducing the computational overhead during speculative execution.
- **Input/Output and space optimization:** Efficient data flow management, minimization of temporary files, and optimization of memory usage are just as important as the parallelization of the computational workload.

12.2 Final Conclusions

This thesis has demonstrated that the shell constitutes a rich, heterogeneous, and complex computational environment. Systematic analysis and evaluation through Koala allowed the mapping of the actual characteristics of shell scripts, the fair comparison of different acceleration systems, and the highlighting of the required design trade-offs.

In summary, the three key conclusions of this thesis are as follows:

1. **There is no "silver bullet":** The variety in dynamic behavior, from scripts lasting seconds to workloads lasting hours, and from minimal to massive memory and I/O usage, proves that no single acceleration strategy can cover all needs.
2. **The problem of "black boxes":** The shell functions primarily as an orchestration language. The extensive reliance on custom, external binaries makes techniques based strictly on pre-existing annotations impractical for generalized use, underscoring the need for dynamic analysis.
3. **The trade-off between adoption effort and final performance:** While manual parallelization can theoretically approach high levels of performance, it imposes a significant programming cost. On the other hand, automated optimization (such as the JIT compilation of PaSh and the out-of-order execution of *hS*) provides a more practical alternative, offering significant speedups with minimal to zero intervention in the source code. However, this approach is accompanied by new challenges, as systems must balance the inherent cost of dynamic orchestration against the benefits of parallelism. This fact highlights the transparent, automatic acceleration of opaque commands as a major research challenge for the shell.

Ultimately, progress in shell research depends directly on the existence of shared, realistic, and reproducible frameworks. Koala is available as an open-source tool, while the permanent archiving of input data and dependencies ensures the long-term reproducibility of the results. Through these features, the Koala suite aspires to become a point of reference, contributing to the formation of a stable framework for more systematic, fair, and comparable future research in the field.

Bibliography

[1] Deepti Raghavan et al. "POSH: a data-aware shell." In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4. DOI: 10.5555/3489146.3489188. URL: https://dl.acm.org/doi/10.5555/3489146.3489188.

[2] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. "The serverless shell." In: *Proceedings of the 22nd International Middleware Conference: Industrial Track*. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, pp. 9–15. ISBN: 9781450391528. DOI: 10.1145/3491084.3491426. URL: https://doi.org/10.1145/3491084.3491426.

[3] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. "Automatic synthesis of parallel unix commands and pipelines with KumQuat." In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 431–432. ISBN: 9781450392044. DOI: 10.1145/3503221.3508400. URL: https://doi.org/10.1145/3503221.3508400.

[4] Diomidis Spinellis and Marios Fragkoulis. "Extending Unix Pipelines to DAGs." In: *IEEE Transactions on Computers* 66.9 (2017), pp. 1547–1561. DOI: 10.1109/TC.2017.2695447. URL: https://ieeexplore.ieee.org/document/7903579.

[5] Nikos Vasilakis et al. "PaSh: Light-Touch Data-Parallel Shell Processing." In: *16th European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 49–66. ISBN: 9781450383349. DOI: 10.1145/3447786.3456228. URL: https://doi.org/10.1145/3447786.3456228.

[6] Shivam Handa et al. "An order-aware dataflow model for parallel Unix pipelines." In: *International Conference on Functional Programming*. Vol. 5. New York, NY, USA: Association for Computing Machinery, Aug. 2021. DOI: 10.1145/3473570. URL: https://doi.org/10.1145/3473570.

[7] Konstantinos Kallas et al. "Practically Correct, Just-in-Time Shell Script Parallelization." In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 769–785. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/kallas.

[8] Michael Greenberg and Austin J. Blatt. "Executable formal semantics for the POSIX shell." In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019). DOI: 10.1145/3371111. URL: https://doi.org/10.1145/3371111.

[9] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. "The Future of the Shell: Unix and Beyond." In: *Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 240–241. ISBN: 9781450384384. DOI: 10.1145/3458336.3465296. URL: https://doi.org/10.1145/3458336.3465296.

[10] Charlie Curtsinger and Daniel W. Barowy. "Riker: Always-Correct and Fast Incremental Builds from Simple Specifications." In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 885–898. ISBN: 978-1-939133-29-70. URL: https://www.usenix.org/conference/atc22/presentation/curtsinger.

[11] Tammam Mustafa et al. “DiSh: Dynamic Shell-Script Distribution.” In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 341–356. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/mustafa.

[12] Zhicheng Huang et al. “Fractal: Fault-Tolerant Shell-Script Distribution.” In: *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. Renton, WA: USENIX Association, May 2026.

[13] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. “The serverless shell.” In: *Proceedings of the 22nd International Middleware Conference: Industrial Track*. Middleware ’21. Québec city, Canada: Association for Computing Machinery, 2021, pp. 9–15. ISBN: 9781450391528. DOI: 10.1145/3491084.3491426. URL: https://doi.org/10.1145/3491084.3491426.

[14] Georgios Liargkovas et al. “Executing Shell Scripts in the Wrong Order, Correctly.” In: *19th Workshop on Hot Topics in Operating Systems*. HotOS ’23. Providence, RI, USA: Association for Computing Machinery, 2023, pp. 103–109. ISBN: 9798400701955. DOI: 10.1145/3593856.3595891. URL: https://doi.org/10.1145/3593856.3595891.

[15] Emery D. Berger. *Optimizing Shell Scripting Languages*. Tech. rep. UMCS TR-2003-009. University of Massachusetts Amherst, 2003.

[16] Evangelos Lamprou et al. “The Koala Benchmarks for the Shell: Characterization and Implications.” In: *2025 USENIX Annual Technical Conference (USENIX ATC ’25)*. Boston, MA: USENIX Association, July 2025, pp. 449–64. ISBN: 978-1-939133-48-9. URL: https://www.usenix.org/conference/atc25/presentation/lamprou.

[17] Paul Falstad. *Z shell (zsh)*. 2022. URL: https://zsh.sourceforge.io/.

[18] Ole Tange. “GNU Parallel - The Command-Line Power Tool.” In: *;login: The USENIX Magazine* 36.1 (Feb. 2011), pp. 42–47. URL: https://doi.org/10.5281/zenodo.16303.

[19] Dennis M. Ritchie and Ken Thompson. “The UNIX time-sharing system.” In: *CACM* 17.7 (July 1974), pp. 365–375. ISSN: 0001-0782. DOI: 10.1145/361011.361061. URL: https://doi.org/10.1145/361011.361061.

[20] S. Greenberg and I.H. Witten. “Directing the User Interface: How People Use Command-Based Computer Systems.” In: *IFAC Proceedings Volumes* 21.5 (1988). 3rd IFAC Conference on Analysis, Design and Evaluation of Man-Machine Systems 1988, Oulu, Finland, 14-16 June 1988, pp. 349–355. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)53932-4. URL: https://www.sciencedirect.com/science/article/pii/S1474667017539324.

[21] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. “Unix Shell Programming: The Next 50 Years.” In: *Workshop on Hot Topics in Operating Systems*. HotOS ’21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 104–111. ISBN: 9781450384384. DOI: 10.1145/3458336.3465294. URL: https://doi.org/10.1145/3458336.3465294.

[22] Github Inc. *The top programming languages*. 2022.

[23] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment.” In: *Linux journal* 2014.239 (2014), p. 2.

[24] Νικόλαος Παγώνας. “SPLaSh: Κλιμάκωση Προγραμμάτων Κελύφους σε Πλατφόρμες Χωρίς Διακομιστή.” Diploma Thesis. Athens, Greece: National Technical University of Athens, School of Electrical and Computer Engineering, June 2024. URL: http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/19144.

[25] Michael Greenberg. *The POSIX Shell Is an Interactive DSL for Concurrency*. URL: https://cs.pomona.edu/~michael/papers/dsldi2018.pdf.

[26] Michael Greenberg. "Word expansion supports POSIX shell interactivity." In: *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*. Programming '18. Nice, France: Association for Computing Machinery, 2018, pp. 153–160. ISBN: 9781450355131. DOI: 10.1145/3191697.3214336. URL: https://doi.org/10.1145/3191697.3214336.

[27] Valve Corporation. *Steam for Linux: Critical file deletion bug due to unset environment variables*. GitHub Issue Tracker. GitHub Issue #3671. 2015.

[28] Anna Herlihy, Periklis Chrysogelos, and Anastasia Ailamaki. "Boosting Efficiency of External Pipelines by Blurring Application Boundaries." In: *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL: https://www.cidrdb.org/cidr2022/papers/p81-herlihy.pdf.

[29] Keith Winstein and Hari Balakrishnan. "Mosh: an interactive remote shell for mobile clients." In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 15. DOI: 10.5555/2342821.2342836. URL: https://dl.acm.org/doi/10.5555/2342821.2342836.

[30] Alexander J. Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. "SysXCHG: Refining Privilege with Adaptive System Call Filters." In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 1964–1978. ISBN: 9798400700507. DOI: 10.1145/3576915.3623137. URL: https://doi.org/10.1145/3576915.3623137.

[31] Cloyce D Spradling. "SPEC CPU2006 benchmark tools." In: *ACM SIGARCH Computer Architecture News* 35.1 (2007), pp. 130–134.

[32] Meikel Poess and Chris Floyd. "New TPC benchmarks for decision support and web commerce." In: *SIGMOD Rec.* 29.4 (Dec. 2000), pp. 64–71. ISSN: 0163-5808. DOI: 10.1145/369275.369291. URL: https://doi.org/10.1145/369275.369291.

[33] Christian Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications." In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. New York, NY, USA: Association for Computing Machinery, 2008, pp. 72–81. DOI: 10.1145/1454115.1454128. URL: https://dl.acm.org/doi/10.1145/1454115.1454128.

[34] Michael Ferdman et al. "Clearing the clouds: a study of emerging scale-out workloads on modern hardware." In: *SIGARCH Comput. Archit. News* 40.1 (Mar. 2012), pp. 37–48. ISSN: 0163-5964. DOI: 10.1145/2189750.2150982. URL: https://doi.org/10.1145/2189750.2150982.

[35] Stephen M. Blackburn et al. "The DaCapo benchmarks: java benchmarking development and analysis." In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 169–190. ISBN: 1595933484. DOI: 10.1145/1167473.1167488. URL: https://doi.org/10.1145/1167473.1167488.

[36] Philipp Lengauer et al. "A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008." In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. L'Aquila, Italy: Association for Computing Machinery, 2017, pp. 3–14. ISBN: 9781450344043. DOI: 10.1145/3030207.3030211. URL: https://doi.org/10.1145/3030207.3030211.

[37] Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. The MIT Press, Aug. 1985. ISBN: 9780262256193. DOI: 10.7551/mitpress/5298.001.0001. URL: https://doi.org/10.7551/mitpress/5298.001.0001.

[38] Urs Hölzle and David Ungar. "Do Object-Oriented Languages Need Special Hardware Support?" In: *Proceedings of the 9th European Conference on Object-Oriented Programming*. ECOOP '95. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 283–302. ISBN: 3540601600. DOI: 10.5555/646153.679532. URL: https://dl.acm.org/doi/10.5555/646153.679532.

[39] Gregory Cohen et al. "EMNIST: Extending MNIST to handwritten letters." In: *2017 International Joint Conference on Neural Networks (IJCNN)*. 2017, pp. 2921–2926. DOI: 10.1109/IJCNN.2017.7966217. URL: https://ieeexplore.ieee.org/document/7966217.

[40] René Just, Darioush Jalali, and Michael D. Ernst. "Defects4J: a database of existing faults to enable controlled testing studies for Java programs." In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 437–440. ISBN: 9781450326452. DOI: 10.1145/2610384.2628055. URL: https://doi.org/10.1145/2610384.2628055.

[41] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma: A Ground-Truth Fuzzing Benchmark." In: *Proc. ACM Meas. Anal. Comput. Syst.* 4.3 (Nov. 2020). DOI: 10.1145/3428334. URL: https://doi.org/10.1145/3428334.

[42] Yu Gan et al. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18. ISBN: 9781450362405. DOI: 10.1145/3297858.3304013. URL: https://doi.org/10.1145/3297858.3304013.

[43] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. "How java programs interact with virtual machines at the microarchitectural level." In: *SIGPLAN Not.* 38.11 (Oct. 2003), pp. 169–186. ISSN: 0362-1340. DOI: 10.1145/949343.949321. URL: https://doi.org/10.1145/949343.949321.

[44] Matthias Hauswirth et al. "Automating vertical profiling." In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 281–296. ISBN: 1595930310. DOI: 10.1145/1094811.1094834. URL: https://doi.org/10.1145/1094811.1094834.

[45] Koichi Nakashima. *ShellBench: A POSIX Shell Benchmark Suite*. Accessed: 2025-04-28. 2021.

[46] Roman Perepelitsa and Contributors. *zsh-bench: Benchmark for Interactive Zsh*. Accessed: 2025-04-28. 2021.

[47] Andy Chu and Contributors. *Oils Benchmarks*. Accessed: 2025-04-28. 2021.

[48] Kirk D. Lucas and Contributors. *UnixBench: The BYTE UNIX Benchmark Suite*. Accessed: 2025-04-28. 2012.

[49] The Open Group. *VSCPCTS 2016 Test Suite*. Accessed: 2025-01-01.

[50] Martijn Dekker. *Modernish*. Accessed: 2025-06-02. 2016.

[51] Koichi Nakashima and contributors. *ShellSpec - A Full-featured BDD Framework for Shell Scripts*. Accessed: 2025-01-01.

[52] Kate Ward. *shUnit2 - xUnit unit testing framework for Bourne based shell scripts*. Accessed: 2025-01-01.

[53] Bats-Core Project. *Bats-Core - Bash Automated Testing System*. Accessed: 2025-01-01.

[54] The Open Group. *The Test Environment Toolkit*. Accessed: 2025-01-01.

[55] Michael Schröder and Jürgen Cito. "An empirical investigation of command-line customization." In: *Empirical Software Engineering* 27.2 (Dec. 14, 2021), p. 30. DOI: 10.1007/s10664-021-10036-y. URL: https://doi.org/10.1007/s10664-021-10036-y.

[56] Yiwen Dong et al. “Bash in the Wild: Language Usage, Code Smells, and Bugs.” In: *ACM Transactions on Software Engineering and Methodology* 32.1 (Jan. 31, 2023), pp. 1–22. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3517193. URL: https://dl.acm.org/doi/10.1145/3517193 (visited on 06/19/2024).

[57] *National Oceanic and Atmospheric Administration (NOAA)*. Accessed: 2025-01-13.

[58] Tom White. *Hadoop: The Definitive Guide*. O’Reilly Media, Inc, 2009. ISBN: 9780596521974.

[59] Stéphane Peter. *makeself - Make self-extractable archives on Unix*. Accessed: 2025-01-13.

[60] Armando Cerna. *Pacaur building script*. Accessed: 2025-01-13.

[61] Kenneth Ward Church. *Unix for Poets*. 1994. URL: https://web.stanford.edu/class/cs124/kwc-unix-for-poets.pdf.

[62] Pawan Bhandari. *Solutions to unixgame.io*. Accessed: 2020-04-14. 2020.

[63] Dave Taylor and Brandon Perry. *Wicked Cool Shell Scripts: 101 Scripts for Linux, OS X, and UNIX Systems*. No Starch Press, 2016. ISBN: 978-1-59327-602-7.

[64] Evangelos Lamprou. “Foundation Models and Unix.” In: *Paged Out!* 6 (Mar. 2025), p. 9.

[65] Alexander Kirillov et al. “Segment Anything.” In: *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2023, pp. 3992–4003. DOI: 10.1109/ICCV51070.2023.00371.

[66] Eleftheria Tsaliki and Diomedes Spinellis. *The Real Numbers for Athens Buses*. 2020. URL: https://insidestory.gr/article/noymera-leoforeia-athinas.

[67] Adam Drake. *Command-line Tools Can Be 235x Faster Than Your Hadoop Cluster*. Accessed: 2025-06-01. 2014.

[68] Enrico Cappellini, Frido Welker, and *et al.* Pandolfi. “Early Pleistocene enamel proteome from Dmanisi resolves Stephanorhinus phylogeny.” In: *Nature* 574.7776 (Oct. 2019), pp. 103–107. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1555-y. URL: https://www.nature.com/articles/s41586-019-1555-y.

[69] Jon Puritz. *Bio594: Using genomic techniques to examine the evolution of populations*. 2019.

[70] Fadhl Ibrahim et al. “TERA-Seq: true end-to-end sequencing of native RNA molecules for transcriptome characterization.” In: *Nucleic Acids Research* 49.20 (2021), e115. DOI: 10.1093/nar/gkab713.

[71] Justine Tunney. *Bash One-Liners for LLMs*. Accessed: 2025-06-01. 2023.

[72] Fabian Pedregosa et al. “Scikit-learn: Machine Learning in Python.” In: *J. Mach. Learn. Res.* 12.null (Nov. 2011), pp. 2825–2830. ISSN: 1532-4435. DOI: 10.5555/1953048.2078195. URL: https://dl.acm.org/doi/10.5555/1953048.2078195.

[73] Jon Bentley. “Programming pearls: a spelling checker.” In: *CACM* 28.5 (May 1985), pp. 456–462. ISSN: 0001-0782. DOI: 10.1145/3532.315102. URL: https://doi.org/10.1145/3532.315102.

[74] Jon Bentley, Don Knuth, and Doug McIlroy. “Programming pearls: a literate program.” In: *CACM* 29.6 (June 1986), pp. 471–483. ISSN: 0001-0782. DOI: 10.1145/5948.315654. URL: https://doi.org/10.1145/5948.315654.

[75] Marek Majkowski. *When Bloom filters don’t bloom*. Accessed: 2025-01-13. Mar. 2020.

[76] Nikos Vasilakis et al. “Preventing Dynamic Library Compromise on node via RWX-Based Privilege Reduction.” In: *ACM Conference on Computer and Communications Security (CCS)*. 2021, pp. 1821–1838.

[77] Abebe Israel. *VPS Audit*. Accessed: 2025-01-13.

[78] Brown University Department of Computer Science. *CSCI 1380: Distributed Computer Systems*. Accessed: 2025-06-04. 2025.

[79] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. "ZMap: fast internet-wide scanning and its security applications." In: *Proceedings of the 22nd USENIX Conference on Security*. SEC'13. Washington, D.C.: USENIX Association, 2013, pp. 605–620. ISBN: 9781931971034. DOI: 10.5555/2534766.2534818.

[80] U.S. Securities and Exchange Commission. *EDGAR Log File Data Sets*. Accessed June 3, 2025. 2024.

[81] Sadjad Fouladi et al. "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 475–488. ISBN: 978-1-939133-03-8. URL: http://www.usenix.org/conference/atc19/presentation/fouladi.

[82] Elias Dabbas. *Web Server Access Logs*. Accessed June 3, 2025. 2020.

[83] The SAM/BAM Format Specification Working Group. *Sequence Alignment/Map Format Specification*. Version 1.6, last modified on 6 Nov 2024. Accessed: 2025-01-13.

[84] Netresec. *Publicly available PCAP files*. Accessed: 2025-06-02. 2025.

[85] Gemma Team. "Gemma 3 Technical Report." In: (2025). arXiv: 2503.19786 [cs.CL]. URL: https://arxiv.org/abs/2503.19786.

[86] Tianqi Zhao et al. "CLAP: Contrastive Language-Audio Pre-training Model for Multi-modal Sentiment Analysis." In: *Proceedings of the 2023 ACM International Conference on Multimedia Retrieval*. ICMR '23. Thessaloniki, Greece: Association for Computing Machinery, 2023, pp. 622–626. ISBN: 9798400701788. DOI: 10.1145/3591106.3592296. URL: https://doi.org/10.1145/3591106.3592296.

[87] *Ollama: Run large language models locally*. Accessed: 2025-06-02. 2023.

[88] Adam Paszke et al. "PyTorch: an imperative style, high-performance deep learning library." In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[89] Simon Willison. *LLM: A CLI tool and Python library for interacting with Large Language Models*. Accessed: 2025-06-02.

[90] Michael S. Hart and Project Gutenberg. *Project Gutenberg*. 1971.

[91] *Arch User Repository (AUR)*. Accessed: 2025-01-13.

[92] *npm.js*. Accessed: 2025-01-13.

[93] Edward Tufte. *New York City Weather Chart*. Accessed: 2025-06-02. 2004.

[94] libdash developers. *libdash*.

[95] Hervé Abdi and Lynne J Williams. "Principal component analysis." In: *Wiley interdisciplinary reviews: computational statistics* 2.4 (2010), pp. 433–459. DOI: 10.1002/wics.101. URL: https://wires.onlinelibrary.wiley.com/doi/10.1002/wics.101.

[96] OpenAI. *OpenAI API: Embeddings Guide*. Accessed: 2025-06-02. 2024. URL: https://platform.openai.com/docs/guides/embeddings.

[97] Alina Petukhova, João P. Matos-Carvalho, and Nuno Fachada. "Text clustering with large language model embeddings." In: *International Journal of Cognitive Computing in Engineering* 6 (2025), pp. 100–108. ISSN: 2666-3074. DOI: https://doi.org/10.1016/j.ijcce.2024.11.004. URL: https://www.sciencedirect.com/science/article/pii/S2666307424000482.

[98] Saiteja Utpala, Alex Gu, and Pin-Yu Chen. "Language Agnostic Code Embeddings." In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. Ed. by Kevin Duh, Helena Gomez, and Steven Bethard. Mexico City, Mexico: Association for Computational Linguistics, June 2024, pp. 678–691. DOI: 10.18653/v1/2024.naacl-long.38. URL: https://aclanthology.org/2024.naacl-long.38.

[99] Vidar Holen et al. *ShellCheck - A shell script static analysis tool*. Accessed: 2025-10-14. 2012. URL: https://www.shellcheck.net/.

[100] Lukas Lazarek et al. "From Ahead-of- to Just-in-Time and Back Again: Static Analysis for Unix Shell Programs." In: *2025 Workshop on Hot Topics in Operating Systems*. HotOS '25. Banff, AB, Canada: Association for Computing Machinery, 2025, pp. 88–95. ISBN: 9798400714757. DOI: 10.1145/3713082.3730395. URL: https://doi.org/10.1145/3713082.3730395.

Appendix A

Infrastructure Script Examples for the `weather` Benchmark Set

```
#!/bin/bash

sudo apt-get update
sudo apt-get install -y --no-install-recommends curl \
    wget \
    unzip \
    coreutils \
    gzip \
    gawk \
    sed \
    findutils \
    git \
    python3 \
    python3-pip \
    python3-venv

pip install --break-system-packages --upgrade pip
pip install --break-system-packages \
    numpy \
    matplotlib
```

Listing A.1: Example of `install.sh` script for the `weather` benchmark set.

```
#!/bin/bash

TOP=$(git rev-parse --show-toplevel)

eval_dir="${TOP}/weather"
input_dir="${eval_dir}/inputs"

URL='https://atlas.cs.brown.edu/data'
URL=$URL/max-temp
FROM=2000
TO=2015

n_samples=99999
suffix="full"
```

```
15
16  mkdir -p "${input_dir}"
17  size=full
18  for arg in "$@"; do
19      case "$arg" in
20      --small) size=small ;;
21      --min) size=min ;;
22      esac
23  done
24  if [[ "$size" == "min" ]]; then
25      if [[ -f "$input_dir/temperatures.min.txt" && -f
        ↪  "$input_dir/tuft_weather.$size.txt" ]]; then
26          echo "Data already downloaded and extracted."
27          exit 0
28      fi
29      min_inputs="$eval_dir/min_inputs/"
30      mkdir -p "$input_dir"
31      cp -r "$min_inputs"/* "$input_dir/"
32      python3 $eval_dir/scripts/generate_input.py $input_dir/tuft_weather.$size.txt
        ↪  --size $size
33      exit 0
34  fi
35  if [[ "$size" == "small" ]]; then
36      if [[ -f "$input_dir/temperatures.small.txt" && -f
        ↪  "$input_dir/tuft_weather.$size.txt" ]]; then
37          echo "Data already downloaded and extracted."
38          exit 0
39      fi
40      data_url="${URL}/temperatures.small.tar.gz"
41      wget --no-check-certificate "$data_url" -O
        ↪  "$input_dir/temperatures.small.tar.gz" || {
42          echo "Failed to download $data_url"
43          exit 1
44      }
45      tar -xzf "$input_dir/temperatures.small.tar.gz" -C "$input_dir"
        ↪  --no-same-owner || {
46          echo "Failed to extract $input_dir/temperatures.small.tar.gz"
47          exit 1
48      }
49      rm "$input_dir/temperatures.small.tar.gz"
50      mv "$input_dir/inputs/temperatures.small.txt"
        ↪  "$input_dir/temperatures.small.txt"
51      rm -rf "$input_dir/inputs"
52      python3 $eval_dir/scripts/generate_input.py $input_dir/tuft_weather.$size.txt
        ↪  --size $size
53      exit 0
54  fi
55  if [[ -f "$input_dir/temperatures.full.txt" && -f
    ↪  "$input_dir/tuft_weather.$size.txt" ]]; then
56      echo "Data already downloaded and extracted."
57      exit 0
```

```
fi
data_url="${URL}/temperatures.full.tar.gz"
wget --no-check-certificate "$data_url" -O "$input_dir/temperatures.full.tar.gz"
 ↪  || {
    echo "Failed to download $data_url"
    exit 1
}
tar -xzf "$input_dir/temperatures.full.tar.gz" -C "$input_dir" --no-same-owner ||
 ↪  {
    echo "Failed to extract $input_dir/temperatures.full.tar.gz"
    exit 1
}
rm "$input_dir/temperatures.full.tar.gz"
mv "$input_dir/inputs/temperatures.full.txt" "$input_dir/temperatures.full.txt"
rm -rf "$input_dir/inputs"
python3 $eval_dir/scripts/generate_input.py $input_dir/tuft_weather.$size.txt
 ↪  --size $size
```

Listing A.2: Example of `fetch.sh` script for the `weather` benchmark set.

```
#!/bin/bash

TOP=$(git rev-parse --show-toplevel)
eval_dir="${TOP}/weather"
outputs_dir="${eval_dir}/outputs"
scripts_dir="${eval_dir}/scripts"
input_dir="${eval_dir}/inputs"
export LC_ALL=C
size=full
selected_scripts=""
while (($#)); do
    case $1 in
        --small)    size=small ;;
        --min)      size=min ;;
        -s|--scripts)
            shift
            while (($#)) && [[ $1 != -* ]]; do
                selected_scripts+=" $1"
                shift
            done
            continue
            ;;
    esac
    shift
done

KOALA_SHELL=${KOALA_SHELL:-bash}
export BENCHMARK_CATEGORY="weather"

```

```
should_run() {
    script_name=$1
    if [ -z "$selected_scripts" ]; then
        return 0
    fi
    for selected in $selected_scripts; do
        if [ "$selected" = "$script_name" ]; then
            return 0
        fi
    done
    return 1
}
if should_run "max-temp"; then
    echo "max-temp"
    export input_file="${input_dir}/temperatures.$size.txt"
    export statistics_dir="$outputs_dir/statistics.$size"
    mkdir -p "$statistics_dir"
    BENCHMARK_INPUT_FILE="$(realpath "$input_file")"
    export BENCHMARK_INPUT_FILE
    BENCHMARK_SCRIPT="$(realpath "${scripts_dir}/temp-analytics.sh")"
    export BENCHMARK_SCRIPT
    $KOALA_SHELL "$scripts_dir/temp-analytics.sh"
    echo "$?"
fi
if should_run "tuft-weather"; then
    echo "tuft-weather"
    export BENCHMARK_SCRIPT="$scripts_dir/tuft-weather.sh"
    export BENCHMARK_INPUT_FILE="$input_dir/tuft_weather.${size}.txt"
    mkdir -p "$outputs_dir/$size"
    $KOALA_SHELL "$BENCHMARK_SCRIPT" "$BENCHMARK_INPUT_FILE" "$size" >
     ↪  "$outputs_dir/$size/turf_weather.log"
    echo "$?"
    rm -rf "$outputs_dir/$size/plots" || true
    mkdir -p "$outputs_dir/$size/plots"
    if [ -d "$eval_dir/plots" ]; then
        mv "$eval_dir/plots"/* "$outputs_dir/$size/plots/"
    fi
fi
```

Listing A.3: Example of `execute.sh` script for the `weather` benchmark set.

```
#!/bin/bash

TOP=$(git rev-parse --show-toplevel)

eval_dir="${TOP}/weather"

size="full"
generate=false
```

```
selected_scripts=""

while (($#)); do
    case $1 in
        --generate) generate=true ;;
        --small)    size=small ;;
        --min)      size=min ;;
        -s|--scripts)
            shift
            while (($#)) && [[ $1 != -* ]]; do
                selected_scripts+=" $1"
                shift
            done
            continue
            ;;
    esac
    shift
done

statistics_dir="${eval_dir}/outputs/statistics.$size"
correct_dir="${eval_dir}/correct-results/statistics.$size"

should_run() {
    script_name=$1
    if [ -z "$selected_scripts" ]; then
        return 0
    fi
    for selected in $selected_scripts; do
        if [ "$selected" = "$script_name" ]; then
            return 0
        fi
    done
    return 1
}
if $generate; then
    if should_run "max-temp"; then
        mkdir -p "$correct_dir"
        cp -r "$statistics_dir"/* "$correct_dir"
    fi
    if should_run "tuft-weather"; then
        hash_dir="$eval_dir/hashes/$size"
        hash_file="$hash_dir/tuft-weather.hash"
        plot_root="$eval_dir/outputs/$size/plots"
        mkdir -p "$hash_dir"
        find "$plot_root" -type f -name '*.png' ! -path '*/tmp/*' -print0 | sort
        ↪  -z | tr '\0' '\n' > "$hash_file"
    fi
    exit 0
fi

if should_run "max-temp"; then
```

```
    diff -q "$statistics_dir/average.txt" "$correct_dir/average.txt"
    echo average.$size $?
    diff -q "$statistics_dir/min.txt" "$correct_dir/min.txt"
    echo min.$size $?
    diff -q "$statistics_dir/max.txt" "$correct_dir/max.txt"
    echo max.$size $?
fi

if should_run "tuft-weather"; then
    hash_dir="$eval_dir/hashes/$size"
    hash_file="$hash_dir/tuft-weather.hash"
    plot_root="$eval_dir/outputs/$size/plots"
    all_exist=true
    while IFS= read -r filepath; do
        if [ ! -f "$filepath" ]; then
            echo "Missing: $filepath"
            all_exist=false
        fi
    done < "$hash_file"
    if [ "$all_exist" = true ]; then
        echo "tuft-weather 0"
    else
        echo "tuft-weather 1"
    fi
fi
```

Listing A.4: Example of `validate.sh` script for the `weather` benchmark set.

```
#!/bin/bash

for arg in "$@"; do
    case "$arg" in
        "-f") force=true ;;
    esac
done
TOP=$(git rev-parse --show-toplevel)
input_dir="${TOP}/weather/inputs"
outputs_dir="${TOP}/weather/outputs"
rm -rf "${outputs_dir}"
if [ "$force" = true ]; then
    rm -rf "${input_dir}"
fi
```

Listing A.5: Example of `clean.sh` script for the `weather` benchmark set.

Appendix B

Full Shell Program Generated by PaSh for the `weather` Example

```
#!/bin/bash

cd "$(dirname "${0}")"
[ -z "${PASH_TOP}" ]
rm_pash_fifos() {
    rm -f /tmp/xUz5/fifo9
    rm -f /tmp/xUz5/fifo10
     # ...similarly for the rest of the fifos...
}
mkfifo_pash_/tmp/xUz5/fifos() {
    mk/tmp/xUz5/fifo /tmp/xUz5/fifo9
    mk/tmp/xUz5/fifo /tmp/xUz5/fifo10
    # ...similarly for the rest of the fifos...
}
rm_pash_/tmp/xUz5/fifos; mk/tmp/xUz5/fifo_pash_/tmp/xUz5/fifos; pids_to_kill=""
d="./data/temperatures"

{ r_split "./data/temperatures/2000" 1000000 \ /tmp/xUz5/fifo9 /tmp/xUz5/fifo10 &
 ↪  }
pids_to_kill="${!} ${pids_to_kill}"
{ r_wrap bash -c ' cut -c 89-92 ' 0< /tmp/xUz5/fifo9 > /tmp/xUz5/fifo11 & }
pids_to_kill="${!} ${pids_to_kill}"
{ r_wrap bash -c ' cut -c 89-92 ' 0< /tmp/xUz5/fifo10 > /tmp/xUz5/fifo12 & }
pids_to_kill="${!} ${pids_to_kill}"
{ r_wrap bash -c ' grep -v 999 ' 0< /tmp/xUz5/fifo11 > /tmp/xUz5/fifo13 & }
pids_to_kill="${!} ${pids_to_kill}"
{ r_wrap bash -c ' grep -v 999 ' 0< /tmp/xUz5/fifo12 > /tmp/xUz5/fifo14 & }
pids_to_kill="${!} ${pids_to_kill}"
{ r_unwrap 0< /tmp/xUz5/fifo13 > /tmp/xUz5/fifo15 & }
pids_to_kill="${!} ${pids_to_kill}"
{ r_unwrap 0< /tmp/xUz5/fifo14 > /tmp/xUz5/fifo16 & }
pids_to_kill="${!} ${pids_to_kill}"
{ dgsh-tee -i /tmp/xUz5/fifo15 -o /tmp/xUz5/fifo20 -I -f -b 5M & }
pids_to_kill="${!} ${pids_to_kill}"
{ dgsh-tee -i /tmp/xUz5/fifo16 -o /tmp/xUz5/fifo21 -I -f -b 5M & }
pids_to_kill="${!} ${pids_to_kill}"
{ sort -r -n 0< /tmp/xUz5/fifo20 > /tmp/xUz5/fifo17 & }
pids_to_kill="${!} ${pids_to_kill}"
```

```
{ sort -r -n 0< /tmp/xUz5/fifo21 > /tmp/xUz5/fifo18 & }
pids_to_kill="${!} ${pids_to_kill}"
{ dgsh-tee -i /tmp/xUz5/fifo17 -o /tmp/xUz5/fifo22 -I -f -b 5M & }
pids_to_kill="${!} ${pids_to_kill}"
{ dgsh-tee -i /tmp/xUz5/fifo18 -o /tmp/xUz5/fifo23 -I -f -b 5M & }
pids_to_kill="${!} ${pids_to_kill}"
{ sort -r -n -m /tmp/xUz5/fifo22 /tmp/xUz5/fifo23 > /tmp/xUz5/fifo6 & }
pids_to_kill="${!} ${pids_to_kill}"
{ head -n 1 0< /tmp/xUz5/fifo6 > "max.2000" & }
pids_to_kill="${!} ${pids_to_kill}"
#...
# For average calculation, replace the last two steps with:
r_merge intermediate_stream1 intermediate_stream2 > merged_stream
# Process stateful commands sequentially (Reduce)
awk '{t+=$1; i++} END {if (i>0) print t/i}' < merged_stream > "avg.2000"
# Wait for completion, cleanup FIFOs, and exit
source wait_for_output_and_sigpipe_rest.sh ${!}; rm_pash_fifos;
(exit "${internal_exec_status}")
```

Listing B.1: Full shell program generated by PaSh for the `weather` example (`--width=2`). This program coordinates the execution of the script's commands, manages intermediate files and data flows, and exploits parallelism wherever possible.

Appendix C

Benchmark Code

C.1 The `count-trigrams` Benchmark

```
#!/bin/bash

IN=${IN:-$SUITE_DIR/inputs/pg}
OUT=${1:-$SUITE_DIR/outputs/4_3b/}
ENTRIES=${ENTRIES:-1000}
mkdir -p "$OUT"
pure_func() {
    input=$1
    TEMPDIR=$(mktemp -d)
    cat > ${TEMPDIR}/${input}.words
    tail +2 ${TEMPDIR}/${input}.words > ${TEMPDIR}/${input}.nextwords
    tail +3 ${TEMPDIR}/${input}.words > ${TEMPDIR}/${input}.nextwords2
    paste ${TEMPDIR}/${input}.words \${TEMPDIR}/${input}.nextwords
     ↪  ${TEMPDIR}/${input}.nextwords2 |
    sort | uniq -c
    rm -rf ${TEMPDIR}
}
export -f pure_func
for input in $(ls ${IN} | head -n ${ENTRIES} | xargs -I arg1 basename arg1)
do
    cat $IN/$input | tr -c 'A-Za-z' '\n' | grep -v "^\s*$" | pure_func $input >
     ↪  ${OUT}/${input}.trigrams
done
```

Listing C.1: The `count-trigrams` script. Uses temporary files and sequential execution.

```
#!/bin/bash

IN="${IN:-$SUITE_DIR/inputs/pg}"
OUT="${1:-$SUITE_DIR/outputs/4_3b/}"
ENTRIES="${ENTRIES:-1000}"
mkdir -p "$OUT"
pure_func() {
    input_file="$1"
    output_file="$2"
```

```
    job_id=$(basename "$input_file")
    tmp_dir="$OUT/.tmp_$$_${job_id}"; mkdir -p "$tmp_dir"
    f_raw1="$tmp_dir/raw1"; f_raw2="$tmp_dir/raw2"; f_raw3="$tmp_dir/raw3"
    f_tail2="$tmp_dir/tail2"; f_tail3="$tmp_dir/tail3"
    mkfifo "$f_raw1" "$f_raw2" "$f_raw3" "$f_tail2" "$f_tail3"
    tail -n +2 < "$f_raw2" > "$f_tail2" &
    tail -n +3 < "$f_raw3" > "$f_tail3" &
    paste "$f_raw1" "$f_tail2" "$f_tail3" | sort | uniq -c > "$output_file" &
    agg_pid=$!
    tr -c 'A-Za-z' '\n' < "$input_file" | \
    grep -v '^[[:space:]]*$' | \
    tee "$f_raw2" "$f_raw3" > "$f_raw1"
    wait $agg_pid
    rm -rf "$tmp_dir"
}
MAX_PROCS=$(nproc 2>/dev/null || echo 4)
job_count=0; processed_count=0
for file in "$IN"/*; do
    [ -f "$file" ] || continue
    if [ "$processed_count" -ge "$ENTRIES" ]; then
        break
    fi
    input_name=${file##*/}
    pure_func "$file" "$OUT/${input_name}.trigrams" &
    job_count=$((job_count + 1))
    if [ "$job_count" -ge "$MAX_PROCS" ]; then
        wait; job_count=0
    fi
    processed_count=$((processed_count + 1))
done
wait
```

Listing C.2: The `count-trigrams` script transformed using Shark. Usage of FIFOs and background parallelization.

C.2 The `covid-1` Benchmark

```
#!/bin/bash

cat "$1" |
  sed 's/T..:..:..//' |
  cut -d ',' -f 1,3 |
  sort -u |
  cut -d ',' -f 1 |
  sort |
  uniq -c |
  awk "{print \$2,\$1}"
```

Listing C.3: The `covid-1` script. Typical processing pipeline.

```
#!/bin/bash

INPUT="$1"
MAX_PROCS=${MAX_PROCS:-$(nproc)}
chunk_size=${chunk_size:-100M}
process_chunk() {
  sed 's/T..:..:..//'| cut -d ',' -f 1,3
}
export -f process_chunk
tmp_dir=$(mktemp -d)
trap "rm -rf $tmp_dir" EXIT
cat "$INPUT" | parallel --pipe --block "$chunk_size" -j "$MAX_PROCS" process_chunk
 ↪  > "$tmp_dir/combined.tmp"
sort -u "$tmp_dir/combined.tmp" |
  cut -d ',' -f 1 |
  sort |
  uniq -c |
  awk '{print $2,$1}'
```

Listing C.4: The `covid-1` script transformed using GNU `parallel`. Manual chunking and function definition.

C.3 The `spell` Benchmark

```
#!/bin/bash
# Calculate mispelled words in an input

dict=$SUITE_DIR/inputs/dict.txt

TEMP_C1="/tmp/{/}.out1"
TEMP1=$(seq -w 0 $(($JOBS - 1)) | sed 's+^+/tmp/in+' | sed 's/$/.out1/' | tr '\n' '
 ↪  ')
TEMP1=$(echo $TEMP1)
mkfifo $TEMP1
parallel "cat {} | col -bx | tr -cs A-Za-z '\n' | tr A-Z a-z | \
tr -d '[:punct:]' | sort > $TEMP_C1" ::: $IN &
sort -m $TEMP1 | parallel -k --jobs ${JOBS} --pipe --block "$BLOCK_SIZE" "uniq" |
uniq | parallel -k --jobs ${JOBS} --pipe --block "$BLOCK_SIZE" "grep -vx -f $dict
 ↪  -"
rm $TEMP1
```

Listing C.5: The `spell` script transformed using GNU `parallel`. Complex flow management.

Appendix D

Comparative Speedup Charts

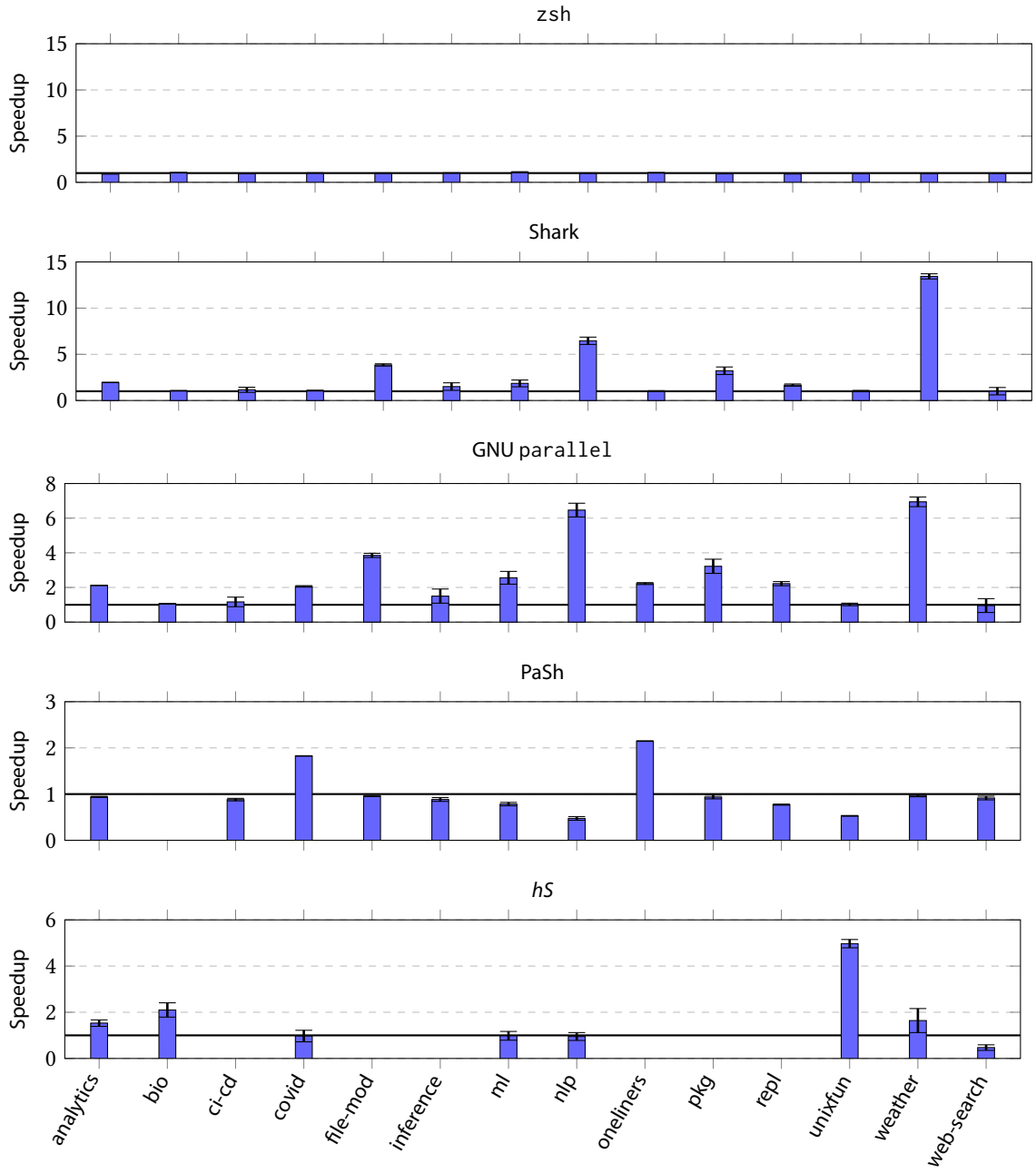


Figure D.1: Comparative overview of speedups. The charts present the relative speedup for the 14 main benchmark suites. The absence of bars indicates that the execution failed or produced incorrect results.