ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Παράλληλη Εκτέλεση Έξυπνων Συμβολαίων με Επίγνωση του Επιπέδου Συγκρούσεων στο Φόρτο Εργασίας

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Δ. Αλεξόπουλος

Αθήνα, Νοέμβριος 2023

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

Παράλληλη Εκτέλεση Έξυπνων Συμβολαίων με Επίγνωση του Επιπέδου Συγκρούσεων στο Φόρτο Εργασίας

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Δ. Αλεξόπουλος

Επιβλέπων: Αριστείδης Παγουρτζής
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14η Νοεμβρίου 2023.

. . . . .	. . . . .	. . . . .
Ελευθέριος Κόκορης-Κόγιας	Γεώργιος Γκούμας	Αριστείδης Παγουρτζής
Επίκ. Καθηγητής ISTA	Αν. Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2023

NATIONAL TECHINCAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Parallel Smart Contract Execution with Contention Awareness

DIPLOMA THESIS

Ioannis D. Alexopoulos

Athens, November 2023

NATIONAL TECHINCAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Parallel Smart Contract Execution with Contention Awareness

DIPLOMA THESIS

Ioannis D. Alexopoulos

Supervisor: Aris Pagourtzis
Professor, NTUA

Approved by the three-member examination committee on November 14th 2023.

. . . . .	. . . . .	. . . . .
Lefteris Kokoris-Kogias	Georgios Goumas	Aris Pagourtzis
Assistant Professor, ISTA	Associate Professor, NTUA	Professor, NTUA

Athens, November 2023

. . . . . .

Ιωάννης Δ. Αλεξόπουλος
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

στους γονείς μου

Περίληψη

Η εισαγωγή του modular design σε συστήματα blockchain έχει αποφέρει σημαντικές βελτιώσεις στην απόδοση τους. Αυτή η σπονδυλωτή προσέγγιση επικεντρώνεται στο διαχωρισμό των επιπέδων consensus και εκτέλεσης, επιτρέποντας έτσι την ταυτόχρονη διανομή των block στο δίκτυο. Οι παραδοσιακές μέθοδοι για το χειρισμό της παράλληλης εκτέλεσης transactions περιλαμβάνουν είτε την ταξινόμηση των συναλλαγών σε μη συγκρουόμενες ομάδες χρησιμοποιώντας μια απαισιόδοξη προσέγγιση είτε την αισιόδοξη εκτέλεση όλων των συναλλαγών, με διακοπή και εκ νέου εκτέλεση κατά την ανίχνευση σύγκρουσης. Ωστόσο, αυτές οι μέθοδοι δεν αξιολογούνται ούτε είναι κατάλληλες υπό υψηλά ανταγωνιστικά (contended) φορτία εργασίας, κάτι που όπως δείχνουμε είναι ένα σύνηθες φαινόμενο στα υπάρχοντα συστήματα blockchain.

Στην παρούσα εργασία, παρουσιάζουμε ένα μηχανισμό παράλληλης εκτέλεσης για έξυπνα συμβόλαια (smart contracts) που συνδυάζει ένα νέο αρχιτεκτονικό μοντέλο, που ονομάζεται *Loose Coupling* το οποίο αντιμετωπίζει τους περισσότερους υπάρχοντες φορείς επιθέσεων σε συνδυασμό με μια μηχανή εκτέλεσης που λειτουργεί αποτελεσματικά υπό contended workloads. Στο πλαίσιο του *Loose Coupling* οι κόμβοι δημιουργούν ασύγχρονα μεταδεδομένα (pre-execution) από συναλλαγές εκτός του κρίσιμου μονοπατιού του consensus, γεγονός που επιτρέπει την απόσβεση του κόστους μεταξύ των συμμετεχόντων. Στη συνέχεια, οι κόμβοι χρησιμοποιούν αυτά τα μεταδεδομένα για να εκτελούν αποτελεσματικά τις συναλλαγές παράλληλα. Επιπλέον, μια βελτιστοποίηση που εισαγάγαμε είναι η ελαχιστοποίηση του cache coherence traffic με την δρομολόγηση των συναλλαγών μιας αλυσίδας εξάρτησης στον ίδιο πυρήνα. Τέλος, τα αποτελέσματα αξιολόγησης της μηχανής εκτέλεσής μας δείχνουν ότι υπερτερεί έναντι μιας σύγχρονης προσέγγισης (Block-STM) έως και 1,4x σε contended workloads.

0.0.1 Λέξεις-Κλειδιά

Blockchain, Έξυπνα συμβόλαια, Παράλληλη εκτέλεση

Abstract

The introduction of modularity to blockchain system architectures has yielded substantial performance improvements. This modular approach focuses on separating the consensus and execution layers, thereby allowing concurrent block distribution.

Traditional methods for handling concurrent transaction processing involve either sorting transactions into non-conflicting batches using a pessimistic approach or optimistically executing all transactions, resorting to aborting and re-executing upon conflict detection. However, these methods are are neither evaluated nor well-suited under highly contended workloads, which as we show is a common occurrence in existing blockchain systems.

In this work, we present a parallel-execution engine for smart contracts that combines a novel architectural model called *Loose Coupling* which mitigates most existing attack vectors and a context-aware execution engine that operates efficiently under highly contended workloads. Under Loose Coupling nodes asynchronously generate metadata (pre-execution) from transactions outside the critical path of consensus which allows the cost to be amortized among participants. Consequently, nodes use this metadata to execute transactions in parallel efficiently. Additionally, one optimization we introduced is to minimize cache coherence traffic by scheduling transactions of a dependency chain on the same core. Benchmark results of our execution engine show that it outperforms a leading approach (Block-STM) by up to 1.4x in contended workloads.

Keywords

Blockchain, Smart Contracts, Parallel Execution

Contents

1 Εισαγωγή

1.1 Κίνητρο

Η τεχνολογία blockchain έχει αναδειχθεί ως μία από τις πιο επαναστατικές καινοτομίες της ψηφιακής εποχής, αναδιαμορφώνοντας τις βιομηχανίες και επαναπροσδιορίζοντας τον τρόπο με τον οποίο διεξάγονται, επαληθεύονται και καταγράφονται οι συναλλαγές. Τα συστήματα blockchain υποστηρίζουν σήμερα μια οικονομία τρισεκατομμυρίων δολαρίων και με την προοπτική του "Web 3.0" προκύπτουν νέες περιπτώσεις χρήσης, καθώς ο αριθμός των χρηστών αυξάνεται ραγδαία ([49]). Το "Web 3.0" ή ο "αποκεντρωμένος ιστός" είναι η επόμενη εξελικτική φάση του διαδικτύου που οραματίζεται ένα πιο ανοικτό, αποκεντρωμένο ψηφιακό οικοσύστημα με την αξιοποίηση της τεχνολογίας blockchain για την προώθηση μεγαλύτερης ιδιωτικότητας, ασφάλειας και ιδιοκτησίας των δεδομένων. Η ανάπτυξη του "Web 3.0" περιλαμβάνει ένα ευρύ φάσμα εφαρμογών και εννοιών, όπως πλατφόρμες αποκεντρωμένης χρηματοδότησης (DeFi), Non-Fungible Tokens (NFTs), αποκεντρωμένες λύσεις αποθήκευσης, συστήματα διαχείρισης ταυτότητας και άλλα.

Όπως διερευνούμε περαιτέρω στο κεφάλαιο 2, στην καρδιά των περισσότερων σύγχρονων συστημάτων Blockchain βρίσκεται ένα ντετερμινιστικό πρωτόκολλο state machine replication (2.1.4), το οποίο επιτρέπει στους χρήστες να συμφωνούν και να εφαρμόζουν στην τοπική τους κατάσταση, μια ακολουθία από blocks συναλλαγών. Κάθε συναλλαγή περιέχει κώδικα έξυπνων συμβολαίων (smart contracts) (2.1.6) γραμμένο σε γλώσσες προγραμματισμού όπως η Solidity (Ethereum) και η Move (Aptos), και κάθε οντότητα που εκτελεί το block των συναλλαγών μέσω μιας μηχανής εκτέλεσης, όπως

η εικονική μηχανή (EVM) του Ethereum, πρέπει να συμφωνήσει στην ίδια τελική κατάσταση.

Με την αύξηση της δημοτικότητας των εφαρμογών "Web 3.0" τα τελευταία χρόνια, μια κεντρική πρόκληση υπήρξε η βελτίωση της απόδοσης των υποκείμενων συστημάτων Blockchain. Τα τελευταία χρόνια έχει προταθεί πληθώρα λύσεων κλιμάκωσης, όπως π.χ:

- **Sharding:** η κατάτμηση του δικτύου blockchain σε μικρότερα ανεξάρτητα υποσύνολα που ονομάζονται shards ([48].
- **Sidechains:** ξεχωριστά δίκτυα blockchain που συνδέονται με ένα "main" blockchain (mainnet) που μπορούν να επεξεργάζονται συναλλαγές παράλληλα και στη συνέχεια να επιστρέφουν στο mainnet ([30].
- **Layer 2 Solutions:** πρωτόκολλα ή πλαίσια που χτίζονται πάνω σε υπάρχοντα blockchains για την αποφόρτιση μέρους της επεξεργασίας των συναλλαγών από την κύρια αλυσίδα. Αξιοσημείωτα παραδείγματα περιλαμβάνουν τα κανάλια πληρωμών ([10]), τα Optimistic και ZK rollups ([51], [47]).

Μια πολλά υποσχόμενη ανεξάρτητη προσέγγιση είναι η εφαρμογή μιας αρθρωτής αρχιτεκτονικής (2.2.2), η οποία διαχωρίζει τις διαδικασίες ενός blockchain μεταξύ εξειδικευμένων επιπέδων, επιτρέποντας τον σχεδιασμό ενός πιο βέλτιστου, κλιμακούμενου και ασφαλούς συστήματος. Σε αντίθεση με μια μονολιθική σχεδίαση όπως το Bitcoin [36] όπου μια ενιαία διεργασία είναι υπεύθυνη για όλες τις λειτουργίες, δηλαδή την εκτέλεση, το consensus και τη διαθεσιμότητα δεδομένων, τα modular blockchains αποσυνδέουν το consensus από την εκτέλεση, επιτρέποντας την ταυτόχρονη διάδοση και εκτέλεση των block.

1.2 Διατύπωση Προβλήματος

Τα modular συστήματα περιορίζονται παραδοσιακά από τον περιορισμό της απόδοσης των υποκείμενων αλγορίθμων consensus. Ωστόσο, πρόσφατες έρευνες έχουν δείξει ότι το σημείο συμφόρησης αυτών των συστημάτων μετατοπίζεται πλέον από το consensus στο επίπεδο εκτέλεσης. Αυτό συμβαίνει επειδή, καθώς αυξάνεται ο όγκος των συναλλαγών σε ένα δίκτυο blockchain, η διαδικασία του consensus μπορεί να ομαδοποιηθεί

και να αποσβεστεί [6], με αποτέλεσμα να μειώνονται οι χρονικές απαιτήσεις σε σύγκριση με τη διαδικασία εκτέλεσης. Κατά συνέπεια, το γεγονός αυτό προκάλεσε μια νέα εστίαση στη διερεύνηση των περιορισμών του στρώματος εκτέλεσης των συστημάτων blockchain.

1.3 Υπάρχουσες Λύσεις

1.3.1 Συνοπτική παρουσίαση των υφιστάμενων προσεγγίσεων

Η πλειονότητα των υφιστάμενων μηχανών εκτέλεσης έξυπνων συμβολαίων, μία από τις πιο ευρέως χρησιμοποιούμενες εκ των οποίων είναι η Ethereum Virtual Machine (EVM) [50], εκτελούν τις συναλλαγές σειριακά και δεν αξιοποιούν τις δυνατότητες των σύγχρονων αρχιτεκτονικών CPU. Για το λόγο αυτό, έχει υπάρξει ένα κύμα έρευνας σε μηχανές παράλληλης εκτέλεσης έξυπνων συμβολαίων [44], [25]. Ο στόχος αυτών των μηχανών εκτέλεσης είναι η εκτέλεση ενός block από n διατεταγμένες συναλλαγές $tx_1 < tx_2 < ... < tx_n$ παράγοντας μια τελική κατάσταση ισοδύναμη με την κατάσταση που παράγεται από την εκτέλεση των συναλλαγών σειριακά.

Συγκρούσεις στην παράλληλη εκτέλεση block

Μια βασική έννοια στην παράλληλη εκτέλεση την οποία πρέπει να προσδιορίσουμε στο πλαίσιο ενός blockchain είναι πότε εμφανίζονται συγκρούσεις μεταξύ συναλλαγών. Οι συγκρούσεις καθορίζουν πότε οι συναλλαγές δεν μπορούν να εκτελεστούν παράλληλα χωρίς να θυσιαστεί η ντετερμινιστική σειροποιησιμότητα (serializability).

1. **Account Conflict:** Όταν δύο νήματα επεξεργάζονται ταυτόχρονα το υπόλοιπο ή άλλα χαρακτηριστικά μιας διεύθυνσης ενός λογαριασμού (address account), δεν είμαστε σίγουροι αν το αποτέλεσμα είναι συνεπές με το αποτέλεσμα της σειριακής επεξεργασίας.

2. **Σύγκρουση αποθήκευσης της ίδιας διεύθυνσης:** Όταν και τα δύο συμβόλαια τροποποιούν την αποθήκευση μιας παγκόσμιας μεταβλητής (global variable).

3. **Cross-Contract Call Conflict:** Εάν το συμβόλαιο A διατεθεί πρώτο, το συμβόλαιο B πρέπει να περιμένει μέχρι να ολοκληρωθεί η διανομή του A για να καλέσει

το συμβόλαιο Α. Ωστόσο, όταν οι συναλλαγές είναι παράλληλες, δεν υπάρχει τέτοια αλληλουχία, γεγονός που οδηγεί σε σύγκρουση.

Οι προσεγγίσεις της παράλληλης εκτέλεσης έξυπνων συμβολαίων μπορούν να κατηγοριοποιηθούν κατά προσέγγιση σε δύο ομάδες: την αισιόδοξη και την απαισιόδοξη, ενώ υπάρχουν και λύσεις που προκύπτουν και από τις δύο ομάδες.

- **Αισιόδοξες προσεγγίσεις:**

  - Οι αισιόδοξες προσεγγίσεις, όπως το Block-STM [25], έχουν σχεδιαστεί γύρω από τις αρχές της Software Transactional Memory (STM) [16]. Οι βιβλιοθήκες STM με αισιόδοξο έλεγχο ταυτόχρονης εκτέλεσης καταγράφουν τις προσπελάσεις μνήμης, επαληθεύουν κάθε συναλλαγή μετά την εκτέλεσή της και διακόπτουν για να εκτελέσουν εκ νέου τις συναλλαγές όταν ο έλεγχος επικύρωσης υποδεικνύει σύγκρουση. Το Block-STM χρησιμοποιείται σε παραγωγή στο Aptos [21].

- **Απαισιόδοξες προσεγγίσεις:**

  - Σε αντίθεση με τις αισιόδοξες προσεγγίσεις, οι απαισιόδοξες προσεγγίσεις περιορίζουν αυστηρά την πρόσβαση στους πόρους απαιτώντας από τις συναλλαγές να δηλώνουν τους πόρους στους οποίους σκοπεύουν να έχουν πρόσβαση πριν από την εκτέλεση. Το γεγονός ότι οι συναλλαγές που προσπελαύνουν διαφορετικές θέσεις μνήμης μπορούν πάντα να εκτελούνται παράλληλα επιτρέπει στη μηχανή εκτέλεσης να δημιουργεί ένα ντετερμινιστικό χρονοδιάγραμμα και να εκτελεί ταυτόχρονα μη συγκρουόμενες συναλλαγές. Ορισμένα παραδείγματα αυτής της προσέγγισης περιλαμβάνουν τα FuelVM, Solana και Sui ([22], [44], [32]).

- **Σύνθετες προσεγγίσεις:**

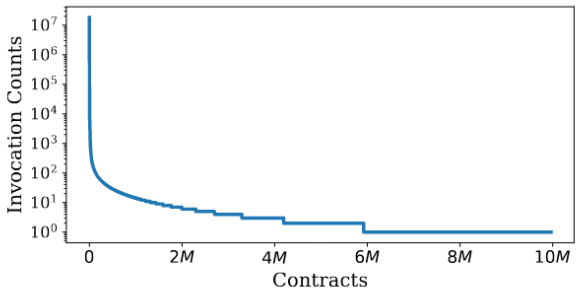
  - Προτείνοντας ένα συνδυασμό τόσο αισιόδοξων όσο και απαισιόδοξων προσεγγίσεων, το Polygon [39] παρουσίασε πρόσφατα μια σύνθετη λύση η οποία αποκαλείται ”προσέγγιση ελάχιστων μεταδεδομένων”. Η υλοποίηση του Block-STM στην αλυσίδα Polygon Proof of Stake (PoS) περιλαμβάνει μια διαδικασία που ονομάζεται *pre-execution* όπου οι πόροι που διαβάζει και γράφει κάθε συναλλαγή εξάγονται από τον block builder, και στη συνέχεια τεκμηριώνονται σε μια δομή δεδομένων κατευθυνόμενου ακυκλικού γράφου (DAG) (βλ. ενότητα 2.4.4 για λεπτομέρειες), η οποία προσαρτάται ως μεταδεδομένα στο block. Αυτό επιτρέπει στους validators να αποφεύγουν τις περιττές επανεκτελέσεις και τα απαισιόδοξα κλειδώματα.

1.3.2 Αδυναμίες των υφιστάμενων προσεγγίσεων

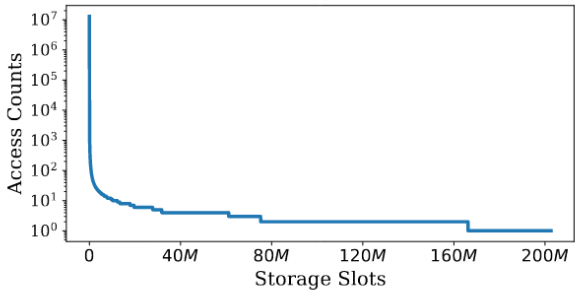
Αισιόδοξες προσεγγίσεις: Παρόλο που οι αισιόδοξες μηχανές παράλληλης εκτέλεσης δηλώνουν εντυπωσιακά νούμερα απόδοσης, στην πράξη η ανάλυση του φόρτου εργασίας ([40]) έχει δείξει ότι όχι μόνο ο φόρτος εργασίας είναι έντονα ανταγωνιστικός με αποτέλεσμα να απαιτείται επανεκτέλεση μεγάλου αριθμού συναλλαγών, αλλά και μεγάλες αλυσίδες εξαρτήσεων μεταξύ συναλλαγών καταλαμβάνουν συχνά μεγάλο ποσοστό του χρόνου εκτέλεσης. Αυτό μπορεί να οφείλεται στην πρόσβαση σε δημοφιλή έξυπνα συμβόλαια για εφαρμογές όπως τα αποκεντρωμένα χρηματιστήρια (DEX) ή NFT minting. Επιπλέον, η ομάδα με επικεφαλής τον Ray Neiheiser κατά τη διάρκεια της πρακτικής μου άσκησης στο ISTA διεξήγαγε μια ανάλυση του ιστορικού συναλλαγών δημοφιλών εφαρμογών blockchain, η οποία υποστηρίζει πλήρως τους ισχυρισμούς σχετικά με τα υφιστάμενα επίπεδα contention σε blockchain workloads. Ο φόρτος εργασίας είναι τέτοιος, ώστε σε ορισμένες περιπτώσεις, οι αισιόδοξες προσεγγίσεις μπορεί να αποδίδουν χειρότερα ακόμη και από μια σειριακή εκτέλεση ενός block. Το σχήμα 1.1 [33] απεικονίζει το φαινόμενο *hot spot* των ανισομερώς κατανεμημένων προσπελάσεων δεδομένων (skewed data accesses).

Πεσιμιστικές προσεγγίσεις: Ενώ οι απαισιόδοξες προσεγγίσεις αποτρέπουν τις περιττές επανεκτελέσεις υπό υψηλά ανταγωνιστικούς φόρτους εργασίας, εισάγεται πρόσθετη πολυπλοκότητα για τους προγραμματιστές έξυπνων συμβολαίων, καθώς πρέπει να δηλώνουν ρητά ποιες διευθύνσεις πόρων επιτρέπεται να προσπελαστούν από το πρόγραμμα. Επιπλέον, σε ορισμένες περιπτώσεις, η ακριβής πρόβλεψη των πόρων στους οποίους θα γίνει πρόσβαση μπορεί να μην είναι δυνατή κατά τη διάρκεια του χρόνου δημιουργίας της συναλλαγής. Κατά συνέπεια, διαμορφώνεται ένα υπερβολικά απαισιόδοξο χρονοδιάγραμμα, με αποτέλεσμα τη διαδοχική εκτέλεση συναλλαγών που διαφορετικά θα μπορούσαν να έχουν εκτελεστεί παράλληλα

Σύνθετες προσεγγίσεις: Τέλος, στην περίπτωση της προσέγγισης "ελάχιστων μεταδεδομένων" του Polygon, καθώς οι συναλλαγές εκτελούνται στο κρίσιμο μονοπάτι του consensus κατά τη διάρκεια της δημιουργίας block, η συνολική απόδοση περιορίζεται σε μεγάλο βαθμό και οι διαθέσιμοι πόροι του συστήματος δεν αξιοποιούνται πλήρως.



(a) Hot contracts



(b) Hot storage slots

Σχήμα 1.1: *[33] Τα φαινόμενα hot spot για α) τις συμβάσεις και (β) θέσεις αποθήκευσης. Οι δείκτες συμβολαίων και θέσεων (άξονες Χ) είναι ταξινομημένοι κατά φθίνουσα σειρά ως προς τις προσπελάσεις τους. Οι αριθμοί των κλήσεων και των προσπελάσεων (άξονες Υ) απεικονίζονται σε λογαριθμική κλίμακα.*

1.4 Προτεινόμενη Λύση

Όλες οι τρέχουσες προσεγγίσεις στη βιβλιογραφία που προσφέρουν παράλληλη εκτέλεση έξυπνων συμβολαίων φαίνεται ότι δεν χειρίζονται το contention με αποτελεσματικό τρόπο και έτσι χάνουν ένα μεγάλο ποσοστό της απόδοσης που υποστηρίζουν.

Στο πλαίσιο της πρακτικής μου άσκησης στην ομάδα SPiDerS (Secure, Private, and Decentralized Systems) του ISTA με επικεφαλής τον επίκουρο καθηγητή Λευτέρη Κόκορη-Κόγια, συνέβαλα σε ένα έργο υπό την επίβλεψη του Ray Neilheiser με στόχο την αντιμετώπιση των περιορισμών που παρουσιάζουν οι τρέχουσες λύσεις.

Η πρότασή μας είναι μια υβριδική λύση μεταξύ πλήρως αποσυνδεδεμένων και μονολιθικών consensus και execution, η οποία ονομάζεται *loose coupling*. Η βασική ιδέα του loose coupling είναι ότι οι κόμβοι υποθετικά προ-εκτελούν συναλλαγές από τη δεξαμενή συναλλαγών τους και υπολογίζουν το *σύνολο ανάγνωσης-εγγραφής* (read-write set) κάθε συναλλαγής πριν από τη σειρά τους να προτείνουν ένα block. Ακριβώς εξαιτίας του τελευταίου, η προ-εκτέλεση είναι υποθετική (speculative), αφού λαμβάνει χώρα έναντι μιας δυνητικά παρωχημένης έκδοσης της κατάστασης του συστήματος. Όταν ένας κόμβος εκλέγεται ως proposer για έναν γύρο, αντλεί μια ομάδα προ-εκτελεσμένων συναλλαγών, υπολογίζει ένα γράφημα (DAG) από τις εξαρτήσεις ανάγνωσης-εγγραφής μεταξύ τους και τοποθετεί αυτές τις πληροφορίες ως μεταδεδομένα μαζί με το προτεινόμενο block. Κατά την ανάλυση πραγματικών φόρτων εργασίας, καταλήξαμε στο συμπέρασμα ότι η παλαιότητα της κατάστασης των παραγωγών block πολύ σπάνια επηρεάζει την ορθότητα του παραγόμενου γράφου εξαρτήσεων. Επιπλέον, το loose coupling αξιοποιεί το πλεονέκτημα ενός αποσυνδεδεμένου συστήματος, το οποίο σε συνδυασμό με μια *Order-Execute architecture* επιτρέπει στο consensus και execution να λειτουργούν ταυτόχρονα σε διαφορετικούς γύρους με pipelined τρόπο. Η διαδικασία προ-εκτέλεσης γίνεται επίσης ταυτόχρονα με το consensus και execution αξιοποιώντας τις δυνατότητες πολυπύρηνης επεξεργασίας κάθε κόμβου. Για το λόγο αυτό, η καθυστέρηση που επιβάλλεται από την προ-εκτέλεση μοιράζεται μεταξύ των συμμετεχόντων παραγωγών block, όπως εξηγείται στο 3.5.

Προκειμένου να μεγιστοποιήσουμε το κέρδος απόδοσης της προσέγγισής μας, χρησιμοποιήσαμε διάφορες σχεδιαστικές βελτιστοποιήσεις, συγκεκριμένα:

- τα μεταδεδομένα που λαμβάνονται μέσω της προ-εκτέλεσης συσκευάζονται μαζί

με το προτεινόμενο block και χρησιμοποιούνται από άλλους validator κόμβους για να σχηματίσουν έναν ντετερμινιστικό χρονοπρογραμματισμό των συναλλαγών του block στους υπολογιστικούς πόρους τους.

- αξιοποιώντας χρόνο που διαφορετικά θα σπαταλιόταν σε επανεκτελέσεις ακυρωμένων συναλλαγών σε χρήσιμες ασύγχρονες λειτουργίες (π.χ. επαλήθευση υπογραφής).

- pinning των νημάτων εκτέλεσης (thread pinning) σε πυρήνες κατά τη διάρκεια της εκτέλεσης όλων των συναλλαγών σε ένα block, αναθέτοντας επίσης διαδοχικές συναλλαγές σε μια αλυσίδα εξάρτησης στον ίδιο πυρήνα, ελαχιστοποιώντας τα invalidation της κρυφής μνήμης cache.

- αξιοποίηση των μεταδεδομένων πριν από την εκτέλεση για το κατάλληλο προφιλτράρισμα των συναλλαγών και τη μείωση του αριθμού των μεγάλων αλυσίδων που συμφορούν τη μηχανή εκτέλεσης, προστατεύοντας παράλληλα το σύστημα από επιθέσεις άρνησης παροχής υπηρεσιών (DoS).

1.4.1 Επίκεντρο της διπλωματικής εργασίας

Το επίκεντρο αυτής της διπλωματικής εργασίας είναι ο σχεδιασμός μιας ντετερμινιστικής μηχανής παράλληλης εκτέλεσης ενός block συναλλαγών η οποία χρησιμοποιεί το γράφημα εξαρτήσεων που υπολογίζεται από την προ-εκτέλεση όπως περιγράφεται στην 1.4. Στη μηχανή εκτέλεσης, λαμβάνουμε επίσης υπόψη το affinity των νημάτων (βλ. 2.5.1) με την αντιστοίχιση των νημάτων σε πυρήνες και τον χρονοπρογραμματισμό των αλυσίδων transactions στον ίδιο πυρήνα (βλ. 3.6). Αξιολογήσαμε τη μηχανή εκτέλεσης κάτω από υποθέσεις τέλειας πρόβλεψης και δείξαμε ότι υπερτερεί έναντι του Block-STM [25] έως και 1,4x. Επιπλέον, παρόλο που το έργο είναι ακόμα σε εξέλιξη και η αξιολόγησή του αποτελεί μελλοντική εργασία, έχω διαμορφώσει τα σενάρια ενορχήστρωσης για την εκτέλεση ενός benchmark πλήρους συστήματος στο AWS και παρουσιάζω την πλήρη ροή εργασίας στο 5.4. Τέλος, παρουσιάζω ολόκληρο το σύστημα, το οποίο ενσωματώνει τη μηχανή εκτέλεσης έξυπνων συμβολαίων, προκειμένου να παρέχω μια πιο καθολική και ολοκληρωμένη κατανόηση του θέματος.

1.5 Δομή Διπλωματικής Εργασίας

- Στο κεφάλαιο 2 παρέχουμε το απαραίτητο θεωρητικό υπόβαθρο: Ξεκινάμε από τις βασικές αρχές της αρχιτεκτονικής blockchain, φτάνοντας σταδιακά στην κατανόηση του τρόπου λειτουργίας των σύγχρονων συστημάτων blockchain.
- Στο Κεφάλαιο 3 περιγράφουμε το μοντέλο του συστήματος και αναλύουμε τις σχεδιαστικές μας επιλογές.
- Στο κεφάλαιο 4 παρουσιάζουμε διεξοδικά τις λεπτομέρειες υλοποίησης του συστήματός μας.
- Στο Κεφάλαιο 5 αξιολογούμε την εργασία μας, την οποία συγκρίνουμε με την προηγούμενη κατάσταση, ενώ εξηγούμε εν συντομία το δικό μας προσαρμοσμένο workload.
- Στο Κεφάλαιο 6 παρέχουμε μια σύνοψη της συνεισφοράς μας καθώς και πιθανές μελλοντικές κατευθύνσεις εργασίας.

2 Υπόβαθρο

2.1 Βασικά στοιχεία Blockchain

2.1.1 Ορισμός blockchain

Ένα blockchain μπορεί να περιγραφεί ως ένα αμετάβλητο καθολικό (ledger) σχεδιασμένο για την καταγραφή συναλλαγών. Λειτουργεί στο πλαίσιο ενός αποκεντρωμένου δικτύου ομοτίμων που δεν εμπιστεύονται πλήρως ο ένας τον άλλον. Κάθε ομότιμος κατέχει ένα αντίγραφο του ledger. Μέσω ενός πρωτοκόλλου consensus (2.1.3), οι ομότιμοι επικυρώνουν τις συναλλαγές, τις ομαδοποιούν σε blocks και δημιουργούν μια αλυσίδα hashes που συνδέει αυτά τα blocks. Αυτή η διαδικασία οργανώνει τις συναλλαγές με έναν ακολουθιακό τρόπο, εξασφαλίζοντας την απαραίτητη συνέπεια του ledger.

2.1.2 Consensus

Ένα πρωτόκολλο consensus είναι ένας μηχανισμός μέσω του οποίου ένα κατανεμημένο δίκτυο κόμβων ή συμμετεχόντων σε ένα σύστημα συμφωνεί σε μια ενιαία, συνεπή κατάσταση του συστήματος, ακόμη και όταν οι μεμονωμένοι κόμβοι μπορεί να έχουν διαφορετικές πληροφορίες. Στο πλαίσιο των συστημάτων blockchain, τα πρωτόκολλα consensus διασφαλίζουν ότι όλοι οι ειλικρινείς κόμβοι του δικτύου, δηλαδή οι κόμβοι που ακολουθούν πιστά το πρωτόκολλο, καταλήγουν σε συμφωνία σχετικά με την εγκυρότητα και τη σειρά των συναλλαγών. Η επιλογή του πρωτοκόλλου consensus εξαρτάται από παράγοντες όπως η φύση του δικτύου (δημόσιο ή ιδιωτικό), το επιθυ-

μητό επίπεδο αποκέντρωσης, τις απαιτήσεις ασφάλειας, την ενεργειακή απόδοση και τους στόχους επεκτασιμότητας. Κάθε πρωτόκολλο συνοδεύεται από τα δικά του αντισταθμιστικά οφέλη.

2.1.3 Εξυπνά συμβόλαια

Τα έξυπνα συμβόλαια αντιπροσωπεύουν εκτελέσιμα προγράμματα υπολογιστών που αποθηκεύονται και εκτελούνται σε ένα δίκτυο blockchain. Τα συμβόλαια αυτά περιλαμβάνουν ένα σύνολο οδηγιών κώδικα που καθορίζουν συγκεκριμένες συνθήκες. Όταν ικανοποιούνται αυτές οι προκαθορισμένες συνθήκες, το έξυπνο συμβόλαιο δρομολογεί συγκεκριμένες ενέργειες ή αποτελέσματα. Εξελίσσοντας από τις απλές peer to peer μεταφορές του πρωτοκόλλου του Bitcoin, τα περισσότερα σύγχρονα οικοσυστήματα λειτουργούν μέσω συναλλαγών που περιέχουν κώδικα έξυπνων συμβολαίων, ο οποίος εκτελείται σε κατάλληλες μηχανές εκτέλεσης, όπως η Ethereum Virtual Machine (EVM) ([12]).

2.2 Μοντέλο Συστήματος

2.2.1 Order-Execute Αρχιτεκτονική

Τα περισσότερα σύγχρονα permissioned συστήματα blockchain λειτουργούν με την αρχιτεκτονική order-execute, η οποία μπορεί να περιγραφεί στις ακόλουθες τρεις βασικές λειτουργίες:

1. Ένας κόμβος του δικτύου που ενεργεί ως παραγωγός block διατάσσει τις συναλλαγές και προτείνει ένα block στους υπόλοιπους σε μια προσπάθεια επίτευξης consensus
2. Αφού επιτευχθεί το consensus, κάθε κόμβος εκτελεί όλες τις συναλλαγές του block χρησιμοποιώντας μια ντετερμινιστική μηχανή εκτέλεσης που εξασφαλίζει συνεπείς καταστάσεις συστήματος.
3. Κάθε κόμβος ενημερώνει την τοπική του κατάσταση, σύμφωνα με τα αποτελέσματα του προηγούμενου βήματος εκτέλεσης.

2.2.2 Modular Design

Αντλώντας έμπνευση από τις τρέχουσες εξελίξεις ([15], [21], [45], [14]) η σύγχρονη προσέγγιση περιλαμβάνει το διαχωρισμό της εκτέλεσης και του consensus σε διακριτά αρθρωτά επίπεδα. Αν και οι κόμβοι μπορούν να εκπληρώνουν και τους δύο ρόλους, μπορούν να επιλέξουν να επιλέξουν οποιαδήποτε μεμονωμένη επιλογή, με αποτέλεσμα να μεγιστοποιείται η ευελιξία του συστήματος.

Στο επίπεδο consensus, το consensus αντιμετωπίζεται ως "μαύρο κουτί" (black-box). Κάθε κόμβος συναίνεσης παράγει μια πανομοιότυπη ακολουθία μπλοκ $B_1, B_2, ..., B_i$ τα οποία στη συνέχεια προωθεί στο στρώμα εκτέλεσης για να εκτελεστούν ντετερμινιστικά.

Αυτός ο διαχωρισμός της εκτέλεσης και του consensus επιτρέπει την εφαρμογή πρωτοκόλλων υψηλής απόδοσης όπως είναι η δομή του Narwahl ([15], όπου το στρώμα consensus αποτελείται από ένα πρωτόκολλο διάδοσης δεδομένων και τον αλγόριθμο consensus. Στο Narwahl ([15]), οι κόμβοι consensus λειτουργούν αποκλειστικά με μεταδεδομένα, όπως τα block hashes, και ως εκ τούτου δεν είναι σε θέση να επικυρώσουν σωστά τις συναλλαγές κατά τη διάρκεια του consensus. Κατά συνέπεια, το consensus παράγει ένα "dirty-ledger" ([45]), δηλαδή ένα blockchain που περιέχει δυνητικά άκυρες συναλλαγές. Παρ' όλα αυτά, εάν πανομοιότυπα block λαμβάνονται με την ίδια σειρά από τους κόμβους εκτέλεσης (κάτι που διασφαλίζεται από το consensus), ένας ντετερμινιστικός αλγόριθμος εκτέλεσης εγγυάται ότι θα ακυρωθούν οι ίδιες ακριβώς συναλλαγές.

2.3 Παράλληλη εκτέλεση έξυπνων συμβολαίων

Όπως αναφέραμε στο 1.2 η σειριακή εκτέλεση των συναλλαγών έξυπνων συμβολαίων έχει γίνει ένα σημαντικό bottleneck και εμποδίζει την ευρεία υιοθέτηση των συστημάτων blockchain. Η άμεση εφαρμογή τεχνικών από τη βιβλιογραφία των βάσεων δεδομένων δεν είναι εφικτή λόγω των θεμελιωδών διαφορών μεταξύ των βάσεων δεδομένων και των έξυπνων συμβολαίων. Σε ένα περιβάλλον βάσεων δεδομένων, τα write-sets των συναλλαγών είναι γενικά προκαθορισμένα και γνωστά εκ των προτέρων. Ωστόσο, αυτή η υπόθεση δεν ισχύει όσον αφορά τα έξυπνα συμβόλαια, καθώς

τα εν λόγω συμβόλαια μπορούν να ενσωματώνουν σύνθετη και ποικίλη λογική. Στη σημερινή εποχή, ένας αυξανόμενος αριθμός εργαλείων παρέχει τη δυνατότητα παράλληλης εκτέλεσης συναλλαγών έξυπνων συμβολαίων, ωστόσο, όπως φαίνεται στο 1.3 οι τεχνικές τους για την παράλληλη εκτέλεση ποικίλλουν.

```
function transfer(address _receiver, uint256 _amount) /*...*/
    {
        balances[msg.sender] -= _amount; //
        balances[_receiver] += _amount,
   // ...
}
```

Listing 2.1: *Μετρητές Solidity, μια κοινή πηγή contention*

2.3.1 Απαισιόδοξες προσεγγίσεις

Οι απαισιόδοξες προσεγγίσεις αποφεύγουν εντελώς τις συγκρούσεις απαιτώντας από τις συναλλαγές να προσδιορίζουν όλους τους πόρους που μπορούν να διαβάσουν ή να γράψουν κατά τη διάρκεια της ζωής τους, επιτρέποντας στη μηχανή εκτέλεσης να δρομολογεί συναλλαγές ταυτόχρονα χωρίς να απαιτεί μηχανισμούς επικύρωσης ή επανεκτέλεσης.

Αυτή η προσέγγιση πρωτοεμφανίστηκε σε μια πρόταση του Ethereum [3] και έκτοτε αναπτύχθηκε σε μια δημοφιλή λύση στρώματος-2 που ονομάζεται FuelVM [22]. Εκτός από το FuelVM, το Solana [44] είναι ένα blockchain που επίσης αξιοποιεί αυτή την προσέγγιση. Ωστόσο, αυτό απαιτεί από τους προγραμματιστές έξυπνων συμβολαίων να προσθέτουν υποδείξεις, οι οποίες μπορεί να επιβαρύνουν τον προγραμματιστή και να οδηγήσουν σε υπερβολικά απαισιόδοξα κλειδώματα, καθώς σε ορισμένες περιπτώσεις μπορεί να μην είναι δυνατόν να προσδιοριστεί εκ των προτέρων ποιοι πόροι θα προσπελαστούν κατά τη διάρκεια της εκτέλεσης της συναλλαγής.

2.3.2 Αισιόδοξος έλεγχος συγχρονισμού στην εκτέλεση έξυπνων συμβολαίων

Ομοίως με την εφαρμογή στις βάσεις δεδομένων, ο αισιόδοξος έλεγχος ταυτόχρονης εκτέλεσης έχει αξιοποιηθεί σε συστήματα blockchain με στόχο την αύξηση της απόδοσης. Οι βιβλιοθήκες Software Transactional Memory (STM) με OCC ([43], [27]) καταγράφουν τις προσπελάσεις στη μνήμη, επικυρώνουν τις συναλλαγές μετά την εκτέλεση για τον εντοπισμό συγκρούσεων και πραγματοποιούν abort και re-execution για να εξασφαλίσουν τη συνοχή των δεδομένων.

2.3.3 Block-STM

Από τη βάση τους, οι βιβλιοθήκες STM δεν εγγυώνται τα ίδια αποτελέσματα όταν οι συναλλαγές εκτελούνται πολλές φορές, κάτι που θα τις καθιστούσε ακατάλληλες για την περίπτωση χρήσης των συστημάτων blockchain, καθώς οι συμμετέχοντες validators πρέπει να καταλήγουν στην ίδια τελική κατάσταση. Ωστόσο, έχει γίνει πρόοδος σε βιβλιοθήκες Deterministic STM, οι οποίες εγγυώνται ότι η εκτέλεση έχει μια προκαθορισμένη σειρά σειριοποίησης, που ισοδυναμεί με τη σειριακή εκτέλεση κάθε συναλλαγής στο μπλοκ. Η Block-STM [25] (Block-level Software Transactional Memory) είναι μια τέτοια μηχανή παράλληλης εκτέλεσης που εκτελεί αποτελεσματικά ένα block από συναλλαγές, ενώ διαχειρίζεται τις εξαρτήσεις και τον συντονισμό μεταξύ των νημάτων.

Η είσοδος στην Block-STM είναι ένα block συναλλαγών, αποτελούμενο από n συναλλαγές. Αυτό το block ορίζει μια προκαθορισμένη σειρά σειριοποίησης $tx_1 < tx_2 < ... < tx_n$. Ο στόχος είναι να εκτελεστούν αυτές οι συναλλαγές μέσα στο block, με αποτέλεσμα μια τελική κατάσταση ισοδύναμη με αυτή που επιτυγχάνεται με τη διαδοχική εκτέλεση των συναλλαγών με τη σειρά $tx_1, tx_2, ..., tx_n$. Κάθε συναλλαγή στο Block-STM μπορεί να υποστεί πολλαπλές επαναλήψεις εκτέλεσης, οι οποίες ονομάζονται ενσαρκώσεις (incarnations). Ένα incarnation θεωρείται ότι έχει διακοπεί όταν είναι απαραίτητη μια επόμενη εκτέλεση με αυξημένο αριθμό incarnation. Μια έκδοση αποτελείται από έναν δείκτη συναλλαγής και έναν αριθμό incarnation.

Το Block-STM χρησιμοποιεί μια δομή δεδομένων πολλαπλών εκδόσεων (multi-version) στη μνήμη για την υποστήριξη ταυτόχρονων αναγνώσεων και εγγραφών από συναλλαγές. Αυτή η δομή δεδομένων αποθηκεύει την τελευταία τιμή που γράφτηκε για κάθε

θέση μνήμης μαζί με τη σχετική έκδοση συναλλαγής. Όταν μια συναλλαγή tx_i διαβάζει από μια θέση μνήμης, ανακτά την τιμή που γράφτηκε από την υψηλότερη συναλλαγή που προηγείται της tx_i στην προκαθορισμένη σειρά σειριοποίησης, μαζί με την αντίστοιχη έκδοση. Οι συναλλαγές μπορούν να διαβάσουν τιμές ακόμη και αν οι επόμενες συναλλαγές έχουν κάνει updates, αρκεί αυτά τα updates να προέρχονται από συναλλαγές με υψηλότερο δείκτη. Εάν καμία προηγούμενη συναλλαγή δεν έχει γράψει σε μια θέση, τότε μια ανάγνωση επιλύεται με βάση την κατάσταση πριν από την εκτέλεση του τρέχοντος block.

2.3.4 Προεκτέλεση και ανίχνευση εξαρτήσεων

Προηγούμενες ερευνητικές εργασίες [7], [9] και [18] έχουν εκμεταλλευτεί τα μοναδικά χαρακτηριστικά της περίπτωσης χρήσης blockchain για να βελτιώσουν την απόδοση του STM. Η στρατηγική τους περιλαμβάνει τον προ-υπολογισμό των εξαρτήσεων, δημιουργώντας έναν κατευθυνόμενο ακυκλικό γράφο που αναπαριστά τις συναλλαγές. Αυτές οι συναλλαγές μπορούν στη συνέχεια να εκτελεστούν χρησιμοποιώντας ένα fork-join schedule. Το αποτέλεσμα είναι ένα χρονοδιάγραμμα που έχει επίγνωση των εξαρτήσεων και, ως εκ τούτου, αποφεύγει τις διακοπές/επανεκτελέσεις λόγω συγκρούσεων.

Σε περιπτώσεις όπου οι συμμετέχοντες έχουν κίνητρο να καταγράφουν και να μοιράζονται αυτό το γράφημα εξαρτήσεων, υπάρχει η δυνατότητα να μειωθεί η επιβάρυνση του προ-υπολογισμού για κάποιους από αυτούς. Με άλλα λόγια, έχοντας ορισμένες οντότητες υπεύθυνες για τον υπολογισμό και τη διανομή του γραφήματος εξαρτήσεων, άλλες οντότητες μπορεί να είναι σε θέση να παραλείψουν αυτό το βήμα γεγονός που οδηγεί σε βελτιωμένη επίδοση.

3 Σχεδίαση

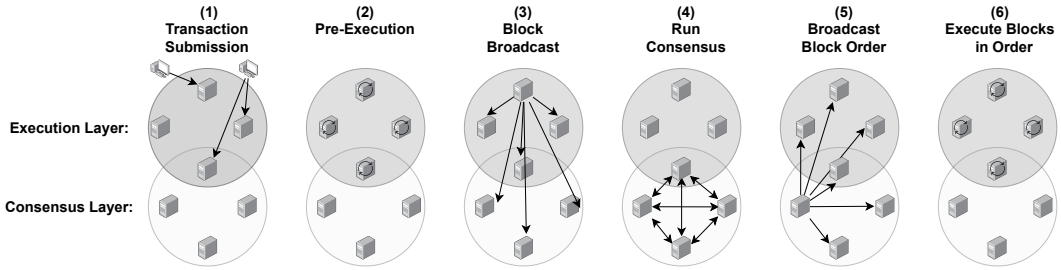
3.1 Επισκόπηση

Με βάση την ενδελεχή ανάλυση πραγματικών workloads, με επικεφαλής τον Ray Neiheiser, και την κατανόηση του τρόπου με τον οποίο η σύνθεση των block επηρεάζει την απόδοση, διατυπώσαμε σαφείς σχεδιαστικούς στόχους για το σύστημά μας. Πρώτον, επιδιώξαμε να αποφύγουμε την παράλληλη εκτέλεση OCC (2.4.2), καθώς δεν αποδίδει καλά σε contented workloads. Επίσης, αποφύγαμε μια απαισιόδοξη προσέγγιση (βλ. **??**) που επιβαρύνει τους προγραμματιστές και τους χρήστες έξυπνων συμβολαίων. Επιπλέον, ο στόχος μας ήταν να δημιουργήσουμε έναν αλγόριθμο για τους proposers που γεμίζει τα blocks με συναλλαγές με τρόπο που να αξιοποιούνται βέλτιστα οι πόροι CPU στους κόμβους εκτέλεσης.

Η πρωταρχική στρατηγική που χρησιμοποιείται στο σύστημά μας για την επίτευξη των δηλωθέντων στόχων περιλαμβάνει μια "χαλαρά συνδεδεμένη" ("loosely coupled") φάση προ-εκτέλεσης, η οποία χρησιμεύει για τη δημιουργία των απαραίτητων μεταδεδομένων για την επακόλουθη εκτέλεση με "επίγνωση του concurrency".

3.2 Loose Coupling

Στο πλήρως αποσυνδεδεμένο σύστημα (decoupled system) που περιγράφεται στο μοντέλο του συστήματος, όπου τα block παράγονται από τους κόμβους consensus, οι παραγωγοί block δεν διαθέτουν το state που είναι απαραίτητο για την προ-εκτέλεση συναλλαγών, καθώς η εκτέλεση των blocks και η ενημέρωση του state είναι πλήρως



Σχήμα 3.1: *Κύκλος ζωής συναλλαγών στη χαλαρή σύζευξη*

διαχωρισμένη από το consensus. Ως εκ τούτου, η κύρια πρόκληση στη σχεδίαση του συστήματός μας είναι να καταστήσουμε δυνατή την προ-εκτέλεση, διατηρώντας παράλληλα τα οφέλη του decoupling.

Ως πρώτο βήμα, αντί οι κόμβοι consensus που δεν έχουν γνώση της κατάστασης να συγκεντρώνουν, να ομαδοποιούν και να διαδίδουν τις συναλλαγές των χρηστών, η ευθύνη αυτή μεταφέρεται στο επίπεδο εκτέλεσης, όπου οι κόμβοι εκτέλεσης παράγουν "Καλά Blocks" συναλλαγών. Ως αποτέλεσμα, το στρώμα consensus θα λαμβάνει την είσοδό του από το στρώμα εκτέλεσης αντί να τη λαμβάνει από τον χρήστη. Ονομάζουμε αυτή την προσέγγιση στην αποσύνδεση "Loose Coupling".

Ο πλήρης κύκλος ζωής μιας συναλλαγής παρουσιάζεται στο Σχήμα 3.1. Η διαδικασία ξεκινά με τους clients να αλληλεπιδρούν με κόμβους από το επίπεδο εκτέλεσης. Στη συνέχεια, αυτοί οι κόμβοι προ-εκτελούν και προ-εγκρίνουν τις συναλλαγές (π.χ. ελέγχουν για τέλη συναλλαγών, επαληθεύουν τις υπογραφές των πελατών κ.λπ.), τις ομαδοποιούν σε "Καλά Blocks" και προσθέτουν μεταδεδομένα όπως οι εξαρτήσεις της συναλλαγής και οι χρόνοι εκτέλεσης στο block (βήμα 1).

Στη συνέχεια, στο τρίτο βήμα, οι κόμβοι εκτέλεσης περνούν στη φάση διάδοσης και μεταδίδουν τα επιμέρους blocks τους σε όλους τους κόμβους ταυτόχρονα.

Στη συνέχεια, το consensus λειτουργεί πάνω στα hashes των blocks χωρίς να απαιτείται γνώση του state παρόμοια με το Narwhal [15] (τέταρτο βήμα). Σε αυτό το πλαίσιο, αντιμετωπίζουμε το consensus ως ένα μαύρο κουτί όπου μπορεί να χρησιμοποιηθεί οποιοσδήποτε αλγόριθμος ανάλογα με τις συγκεκριμένες απαιτήσεις του συστήματος. Μετά τον τερματισμό του consensus, οι κόμβοι consensus στη συνέχεια μεταδίδουν την προκύπτουσα σειρά block στους κόμβους εκτέλεσης (πέμπτο μέρος), οι οποίοι στη συνέχεια εκτελούν τα blocks ντετερμινιστικά σύμφωνα με τη προκαθορισμένη σειρά

(τελευταίο μέρος).

Με την παρεμβολή του στρώματος εκτέλεσης μεταξύ του πελάτη και του στρώματος consensus μπορούμε να αξιοποιήσουμε πλήρως τις εγγυήσεις liveness και safety που καθιερώθηκαν στο Narwhal [15] καθώς το στρώμα consensus παραμένει αμετάβλητο. Μεταφέρουμε απλώς τη μετάδοση block στο στρώμα εκτέλεσης έτσι ώστε ο κόμβος που παράγει το block να μπορεί να προ-εκτελεί συναλλαγές. Ως εκ τούτου, εφόσον υπάρχει τουλάχιστον ένας ειλικρινής κόμβος εκτέλεσης που μεταδίδει σωστά το block του στην πλειοψηφία των consensus nodes, το consensus θα παράγει με συνέπεια μια διατεταγμένη αλυσίδα blocks. Αντίστροφα, οποιαδήποτε block το οποίο δεν μεταδίδεται στην πλειονότητα των consensus nodes δεν μπορεί να προκύψει ως αποτέλεσμα από τον μηχανισμό consensus.

4 Υλοποίηση

Ενσωματώσαμε το σύστημά μας με το Aptos [21] σε συνδυασμό με το Block-STM [25]. Πρώτον, εισάγαμε την προ-εκτέλεση στη φάση δημιουργίας των block, που περιλαμβάνει τη δημιουργία "Καλών Blocks" και την προσθήκη μετα-δεδομένων. Δεύτερον, αλλάξαμε τον αλγόριθμο εκτέλεσης ώστε να λαμβάνουμε υπόψη τις εξαρτήσεις και τον χρόνο εκτέλεσης των συναλλαγών και να επαληθεύουμε τις υπογραφές των συναλλαγών κατά τη διάρκεια του χρόνου αδράνειας των πυρήνων και όχι στο κρίσιμο μονοπάτι εκτέλεσης. Επιλέξαμε το Aptos καθώς είναι ένα έργο υψηλής ποιότητας που ήδη ακολουθεί ένα αποσυνδεδεμένο μοντέλο συναίνεσης και εκτέλεσης.

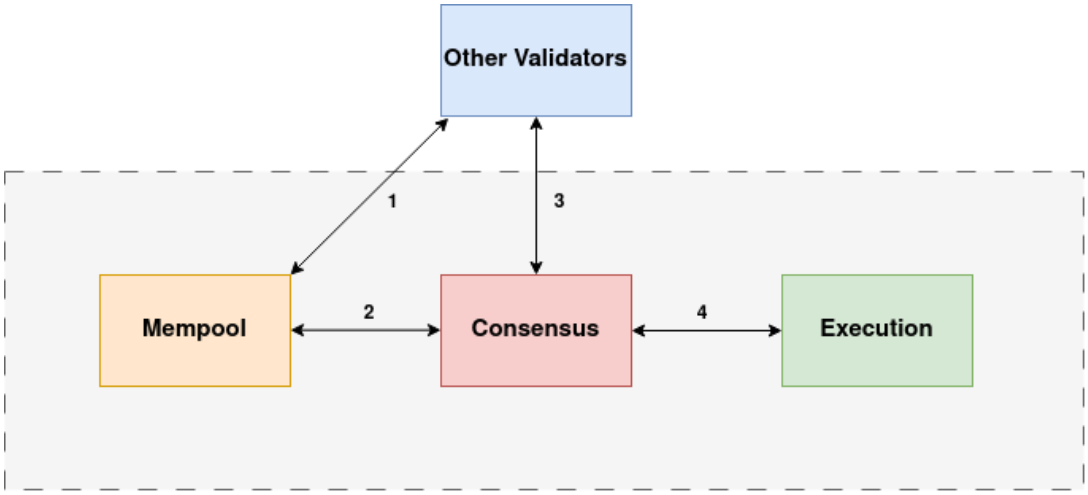
4.1 Επισκόπηση

Παρουσιάζουμε τώρα τα βασικά στοιχεία που υπάρχουν σε έναν κόμβο που εκτελεί το Aptos:

Consensus: Το blockchain Aptos χρησιμοποιεί ένα πρωτόκολλο consensus που ονομάζεται AptosBFT το οποίο βασίζεται στο Jolteon [24] και στο HotStuff [6].

Execution: Η μηχανή εκτέλεσης που χρησιμοποιείται για την παράλληλη εκτέλεση συναλλαγών είναι μια άμεση υλοποίηση του Block-STM [25], όπως περιγράφεται στο 2.4.3.

Mempool: Το Mempool είναι ένας κοινόχρηστος buffer που περιέχει τις συναλλαγές που έχουν υποβληθεί στο σύστημα αλλά δεν έχουν ακόμη συμφωνηθεί μέσω του μηχανισμού consensus και δεν έχουν εκτελεστεί.



Σχήμα 4.1: *Αλληλεπίδραση επιμέρους συνιστωσών*

Αυτά τα στοιχεία αλληλεπιδρούν μεταξύ τους με τους ακόλουθους τρόπους:

(1) Mempool ↔ Other Validators: Όταν μια νέα συναλλαγή προστίθεται στο mempool από ένα αίτημα χρήστη, το mempool μοιράζεται αυτή τη συναλλαγή με άλλους κόμβους validator στο σύστημα. Όταν ένας validator λαμβάνει μια συναλλαγή από το mempool ενός άλλου validator, την προσθέτει στον τοπικό αποθηκευτικό χώρο του.

(2) Consensus ↔ Mempool: Όταν ένας validator είναι proposer, η μονάδα consensus του αντλεί ένα block συναλλαγών από το mempool του και σχηματίζει ένα προτεινόμενο block.

(3) Consensus ↔ Other Validators: Εάν ένας validator είναι proposer, η μονάδα consensus του διαδίδει το προτεινόμενο block συναλλαγών σε άλλους validators.

(4) Συναίνεση ↔ Εκτέλεση: Αφού ένα block προστεθεί στο "dirty ledger" από το πρωτόκολλο consensus, μεταβιβάζεται στο component εκτέλεσης για να εκτελεστεί ντετερμινιστικά και να γίνει commit.

Κάνοντας χρήση της βάσης κώδικα Aptos ως υπόβαθρο για το σύστημά μας, η κύρια γλώσσα προγραμματισμού που χρησιμοποιήσαμε ήταν η Rust [5]. Η Rust είναι μια σύγχρονη γλώσσα προγραμματισμού η οποία παρέχει ασφάλεια μνήμης χωρίς να θυσιάζει την απόδοση. Οι σημαντικότερες αλλαγές έγιναν γύρω από τη μηχανή εκτέλεσης, προσαρμόζοντάς την ώστε να χειρίζεται προ-υπολογισμένες υποδείξεις εξαρτήσεων (dependency hints).

5 Αξιολόγηση

5.1 Επισκόπηση

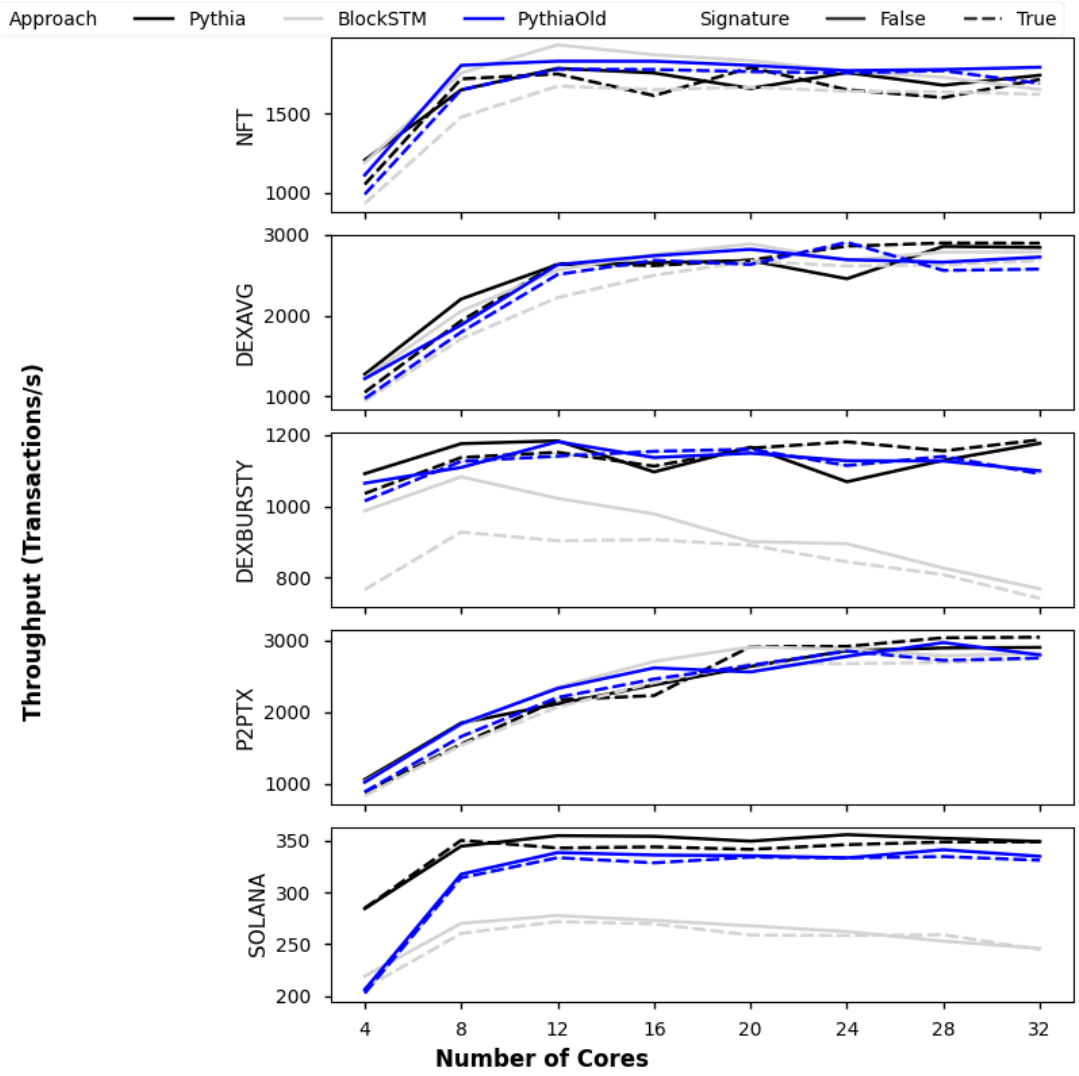
Αξιολογήσαμε το σύστημά σε δύο βήματα: **(1)** Αρχικά αξιολογήσαμε το execution engine ξεχωριστά, χωρίς να λάβουμε υπόψη τη δημιουργία block, ούτε τον χρόνο που δαπανήθηκε κατά την προ-εκτέλεση και στη συνέχεια **(2)** αναπτύξαμε το δικό μας testnet στο AWS [2], προσομοιώνοντας την πλήρη λειτουργία του συστήματος. Ωστόσο, η πλήρης ανάλυση του συστήματος ξεφεύγει από το πεδίο αυτής της εργασίας και αποτελεί μέρος μελλοντικού έργου με επικεφαλής τον Ray Neiheiser.

Πολλές μελέτες αξιολογούν συστήματα blockchain χρησιμοποιώντας απλά workloads συναλλαγών peer-to-peer [25] ή workloads με χαμηλά επίπεδα contention και πολυπλοκότητας. Ενώ έχουν προταθεί καλύτερα πλαίσια αξιολόγησης, όπως το Diablo [26], εξακολουθούν να μην αντιπροσωπεύουν ρεαλιστικά workloads σε πραγματικά blockchain. Για να το αντιμετωπίσουμε αυτό, πραγματοποιήσαμε μια διεξοδική ανάλυση της δραστηριότητας των χρηστών στο Ethereum και το Solana και εντοπίσαμε τέσσερα ρεαλιστικά σενάρια εκτέλεσης: NFT Minting, DEX Trading, Peer-to-Peer (P2P) Συναλλαγές και Mixed Contracts. Αυτά τα σενάρια καλύπτουν ένα ευρύ φάσμα χαρακτηριστικών εκτέλεσης, από υψηλά επίπεδα contention και σύνθετες αλληλεπιδράσεις συμβολαίων έως απλές συναλλαγές P2P. Αυτό επιτρέπει μια πιο ολοκληρωμένη αξιολόγηση των συστημάτων blockchain και της ικανότητάς τους να λειτουργούν σε ρεαλιστικά workloads.

5.2 Εκτέλεση με επίγνωση εξάρτησης

Η αξιολόγηση της μηχανής εκτέλεσης πραγματοποιήθηκε σε Mac Pro (2023) με CPU 24 πυρήνων (16 πυρήνες απόδοσης και 8 πυρήνες απόδοσης) και 64 GB μνήμης RAM.

Το σχήμα 5.1 δείχνει το throughput ανά δευτερόλεπτο για όλους τους φόρτους εργασίας για το BlockSTM και το δικό μας μοντέλο εκτέλεσης χωρίς signature validation (πλήρης γραμμή) και με signature validation (διακεκομμένη γραμμή). Παρατηρούμε ότι η διαφορά απόδοσης μεταξύ των δύο προσεγγίσεων αυξάνεται σημαντικά όταν συμπεριλαμβάνεται η λειτουργία του signature validation ταυτόχρονα με την εκτέλεση ενός block. Αυτό συμβαίνει επειδή το BlockSTM χρησιμοποιεί όλους τους πυρήνες καθ'όλη την διάρκεια της εκτέλεσης με ελάχιστο αδρανή χρόνο για ασύγχρονες λειτουργίες. Αντιθέτως, το σύστημά μας δρομολογεί τις συναλλαγές λαμβάνοντας υπόψη τις εξαρτήσεις και μπορεί να χρησιμοποιήσει τον χρόνο αδράνειας της CPU που κερδίζει από αυτό για να επαληθεύσει τις υπογραφές.



Σχήμα 5.1: *Throughput per Second - Execution Engine*

6 Επίλογος

Σε αυτή την εργασία, προτείναμε έναν σχεδιασμό που επιτρέπει την ντετερμινιστική παράλληλη εκτέλεση έξυπνων συμβολαίων, αξιοποιώντας τις πολυπύρηνες δυνατότητες των σύγχρονων υπολογιστικών μονάδων. Επιπλέον, παρουσιάσαμε έναν δρομολογητή (scheduler) για συναλλαγές smart contracts, ο οποίος χρησιμοποιεί προυπολογισμένες πληροφορίες για να επιταχύνει την εκτέλεση έως και 1.4x σε σύγκριση με το BlockSTM [25] σε πραγματικά workloads.

6.1 Συμπερασματικά Σχόλια

Σύνοψη

Work	Context Driven Execution	Decoupled Consensus & Execution	Intelligent Block Assembly	Susceptibility Performance Attacks
Block-STM [25]	No	Yes	No	High
Thomas Dickerson, et al. [17]	Yes	No	No	High
Polygon [39]	Yes	No	No	Low
FuelVM [22]	Yes	No	No	Low
FSC [34]	Yes	No	No	Low
Solana [44]	Yes	No	No	High
Eve [28]	No	No	Limited	High
Aria [35]	No	No	No	High
This work	Yes	Yes	Yes	Low

Πίνακας 6.1: ***Overview of the different approaches***

Ο Πίνακας 6.1 συγκρίνει διαφορετικές προσεγγίσεις με βάση τέσσερα κύρια κριτήρια:

(1) χρήση μετα-δεδομένων για την αποφυγή συγκρούσεων κατά την εκτέλεση, (2) αποσύνδεση του consensus από το execution, (3) χρήση δεδομένων για την κατασκευή εύκολα parallelizable block, και (4) ευαισθησία σε επιθέσεις.

Από όσο γνωρίζουμε, όλες οι τρέχουσες προσεγγίσεις στη βιβλιογραφία που προσφέρουν παράλληλη εκτέλεση έξυπνων συμβολαίων επιβραδύνονται σημαντικά από contended workloads και διαδοχικές αλυσίδες συναλλαγών και επίσης υπόκεινται σε επιθέσεις απόδοσης. Προ-εκτελούν επίσης συναλλαγές στην κρίσιμη διαδρομή του consensus, μειώνοντας σημαντικά την δυνατή επιτάχυνση.

Στην εργασία μας, αντιμετωπίζουμε αυτές τις ελλείψεις με τη βοήθεια της ασύγχρονης προ-εκτέλεσης, η οποία μας επιτρέπει να σχηματίσουμε meta-data χωρίς να βασιζόμαστε σε επιπλέον inputs από τον χρήστη. Στη συνέχεια χρησιμοποιούμε αυτά τα δεδομένα για να επιταχύνουμε την εκτέλεση των blocks.

6.2 Μελλοντικό Έργο

Οι περισσότερες συνεισφορές που παρουσιάζονται σε αυτή τη διπλωματική εργασία διεξήχθησαν σε συνεργασία με μια ομάδα στο ISTA και το έργο βρίσκεται ακόμα σε εξέλιξη. Απομένουν ακόμα βασικά ζητήματα όπως οι τελικές λεπτομέρειες εφαρμογής και αξιολόγησης του συστήματος. Ένα θεμελίωδες ακόμη πρόβλημα που πρέπει να αντιμετωπιστεί είναι ότι οι συναλλαγές που προ-εκτελεί κάθε κόμβος μπορούν να επικαλύπτονται, με αποτέλεσμα να χάνεται χρόνος εκτέλεσης. Ένας τρόπος για να λυθεί αυτό μπορεί να είναι το sharding των συναλλαγών κάθε κόμβου, έτσι ώστε κάθε κόμβος να προεκτελεί μόνο τις συναλλαγές που θα προτείνει στο μέλλον. Επιπλέον, καθώς ο scheduler της μηχανής εκτέλεσης είναι βελτιστοποιημένος υπό την υπόθεση τέλειας πρόβλεψης, η απόδοσή του πρέπει επίσης να ελεγχθεί κάτω από διαβαθμισμένα ποσοστά λανθασμένων υποδείξεων. Τέλος, τα κίνητρα ενός validator να λειτουργεί σύμφωνα με το πρωτόκολλο και να προτείνει ”Καλά Block” πρέπει να οριστούν με σαφήνεια.

1 Introduction

1.1 Motivation

Blockchain technology has emerged as a one of the most transformative innovations of the digital age, reshaping industries and redefining the way transactions are conducted, verified, and recorded. Blockchain systems are currently supporting a trillion-dollar economy and with the prospect of "Web 3.0" new use cases are arising as the number of users grows rapidly ([49]). "Web 3.0" or the "decentralized web" is the next evolutionary phase of the internet that envisions a more open, decentralized, and user-centric digital ecosystem by leveraging blockchain technology to empower individuals and promote greater privacy, security and ownership of data. The development of "Web 3.0" involves a wide range of applications and concepts, including decentralized finance (DeFi) platforms, non-fungible tokens (NFTs), decentralized storage solutions, identity management systems, and more.

As we further explore in Chapter 2, at the heart of most modern Blockchain systems is a deterministic state machine replication protocol (2.1.4), which allows users to agree on and apply on their local state, a sequence of blocks of transactions. Each transaction contains smart contract code (2.1.6) written in programming languages such as Solidity (Ethereum) and Move (Aptos), and every entity that executes the block of transactions through an execution engine such as the Ethereum Virtual Machine (EVM) must agree on the same final state.

With the rise in popularity of "Web 3.0" applications in recent years, a central challenge has been improving the throughput of the underlying Blockchain systems. There has

been a plethora of scalability solutions proposed in the last years such as:

- **Sharding:** partioning the blockchain network into smaller independent subsets called shards ([48].

- **Sidechains:** separate blockchain networks connected to a "main" blockchain (mainnet) that can process transactions in parallel and then settle back into the mainnet ([30].

- **Layer 2 Solutions:** protocols or frameworks built on top of existing blockchains to offload some of the transaction processing from the main chain. Notable examples include Payment Channels ([10]), Optimistic and ZK rollups ([51], [47]).

One very promising orthogonal approach is applying a modular architecture (2.2.2), which splits up the processes of a blockchain among specialized layers, allowing the design of a more optimal, scalable and secure system. In contrast to a monolithic design such as Bitcoin [36] where a single process is responsible for all functions, namely Execution, Consensus and Data availability, modular blockchains decouple consensus from execution, enabling concurrent block dissemination and execution.

1.2 Problem Statement

These modular systems have traditionally been limited by the throughput constraint of their underlying consensus algorithms. However, recent research has shown [15] that the bottleneck of these systems is now shifting from consensus to the execution layer. This occurs because, as the volume of transactions within a blockchain network rises, the consensus procedure can be batched and amortized [6], resulting in reduced time requirements in comparison to the execution process. Consequently, this has prompted a renewed focus on exploring the limitations of the execution layer of blockchain systems.

1.3 Existing Solutions

1.3.1 Summary of existing approaches

The majority of existing smart contract execution engines, one of the most widely used being the Ethereum Virtual Machine (EVM) [50], execute transactions sequentially and do not utilize the capabilities of modern CPU architectures. For this reason, there has been a surge of research into parallel execution engines for smart contracts [44], [25]. The aim of these execution engines is to execute a block of n ordered transactions $tx_1 < tx_2 < ... <$ txn producing a final state equivalent to the state produced by executing the transactions in sequence.

Conflicts in Parallel Blockchain Execution

A key concept in parallel execution which we need to specify in the context of a blockchain is when conflicts between transactions appear. Conflicts determine when transactions cannot run in parallel without sacrificing deterministic serializability.

1. **Account Conflict:** When two threads process the balance or other attributes of an address account at the same time, we are not sure if the outcome is consistent with the result of sequential processing.
2. **Storage Conflict of the Same Address:** Where both contracts have modified the storage of the same global variable.
3. **Cross-Contract Call Conflict:** If contract A is deployed first, contract B needs to wait until the deployment of contract A is completed to call contract A. However, when the transactions are parallel, there is no such sequence, which leads to conflict.

The approaches of smart contract parallel execution can be roughly categorized into two groups: optimistic and pessimistic, while solutions stemming from both groups also exist.

- **Optimistic Approaches:**
  - Optimistic approaches, such as Block-STM [25], are designed around the principles of optimistically controlled Software Transactional Memory (STM) [16]. STM libraries with optimistic concurrency control record memory accesses, validate every transaction after execution, and abort to re-execute transactions when validation indicates a conflict. Block-STM currently runs in production on Aptos [21].
- **Pessimistic Approaches:**
  - Contrary to optimistic approaches, pessimistic approaches strictly limit resource access by requiring transactions to declare the resources they will access prior to execution. The fact that transactions that access different memory locations can always be executed in parallel allows the execution engine to create a deterministic schedule and execute non-conflicting transactions concurrently. Some examples of this approach include FuelVM, Solana and Sui ([22], [44], [32]).

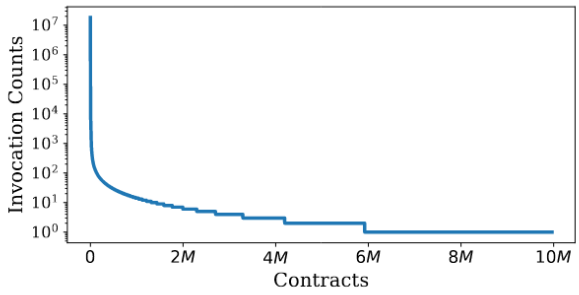
- **Composite Approaches:**

  – Proposing a mix of both optimistic and pessimistic approaches, Polygon [39] recently introduced a composite solution which is termed "the minimum metadata approach". The implementation of Block-STM on the Polygon Proof of Stake (PoS) chain involves a process termed *pre-execution* where the resources that each transaction reads and writes are extracted by the block builder, then documented in a directed acyclic graph (DAG) data structure (see Section 2.4.4 for details), which is appended as metadata to the block. This allows downstream nodes and validators to avoid unnecessary re-executions and pessimistic locking.

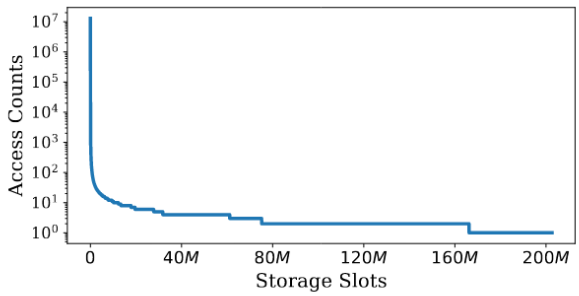
1.3.2 Shortcomings of existing approaches

Optimistic approaches: Although optimistic parallel execution engines claim impressive throughput numbers, in practice workload analysis ([40]) has shown that not only is the workload highly contended resulting in a large number of transactions needing re-execution, but long chains of dependencies between transactions often take up a large percentage of the execution time. This may happen due to accessing popular smart contracts for applications such as decentralized exchanges (DEX) or NFT minting. Furthermore, an analysis of the transaction history of popular blockchain applications was also conducted by the group led by Ray Neiheiser during my internship in ISTA which fully supports the claims regarding the existing levels of contention in blockchain workloads. The workload is such, that in some cases, optimistic approaches may even perform worse than a single thread sequential execution. Figure 1.1 [33] visualizes the *hot spot* phenomenon of skewed data accesses.

Pessimistic approaches: While pessimistic approaches prevent redundant re-executions under highly contended workloads, additional complexity is introduced for smart contract developers as they need to explicitly state which resource addresses are allowed to be accessed by the program. Additionally, in some cases accurately predicting the resources that will be accessed during transaction creation time may not be possible. Consequently, an overly pessimistic schedule is formed, resulting in the sequential execution of transactions that could have otherwise been executed in parallel.



(a) Hot contracts



(b) Hot storage slots

Figure 1.1: *[33] The hot spot phenomenons for (a) contracts and (b) storage slots. The contract and slot indices (X-axises) are sorted in the descending order in terms of their invocation and access counts. The invocation and access counts (Y-axises) are plotted on a logarithmic scale.*

Composite approaches: Finally, in the case of Polygon's "minimum metadata approach", as transactions are executed on the critical path of consensus during block creation, the potential throughput is heavily limited and the available resources of the system are underutilized.

1.4 Proposed Solution

All current approaches in the literature that offer parallel smart-contract execution seem to not handle contention in an efficient manner and thus lose a large percentage of their claimed performance.

As part of my internship in the SPiDerS (Secure, Private, and Decentralized Systems) group at ISTA led by Assistant Professor Lefteris Kokoris-Kogias, I contributed to a project overseen by Ray Neilheiser aiming to address the limitations of current solutions.

Our proposal is a hybrid solution between fully decoupled and monolithic consensus and execution, termed *loose coupling*. The main idea of loose coupling is that nodes speculatively pre-execute transactions from their transaction pool and compute the *read-/write set* of each transaction before their turn to propose a block. Exactly because of the latter, the pre-execution is speculative, since it takes place against a potentially stale version of the system state. When a node is elected as a proposer for a round, it pulls a batch of pre-executed transactions, calculates a graph (DAG) from the read-write dependencies between them and packs this information as metadata with the proposed block. Upon analyzing real-world workload, we concluded that the staleness of the state of block producers very rarely affects the correctness of the produced dependency graph. Furthermore, this loose coupling leverages the advantage of a decoupled system which in combination with an *Order-Execute architecture* allows consensus and execution to function concurrently on different rounds in a pipelined fashion. The pre-execution process is also done concurrently to consensus and execution utilizing the multi-core processing capabilities of each node. For this reason, the latency imposed by pre-execution is shared among participating block producers as is explained in 3.5.

In order to maximize the performance gain of our approach, we employed several design optimizations, namely:

- metadata obtained through pre-execution is packed together with the proposed block and used by other validator nodes to form a deterministic scheduling of the transactions of the block across the resources of the system.

- utilizing time that would otherwise be wasted on re-executions of aborted transactions on useful asynchronous functions (i.e., signature verification).

- pinning spawned threads to cores during the execution of all transactions in a block, while also assigning consecutive tasks in a dependency chain to the same core, minimizing cache invalidation.

- leveraging pre-execution metadata to pre-filter transactions appropriately and reduce the number of long chains congesting the execution engine while the system is also protected from denial of service attacks (DoS).

1.4.1 Focus of the thesis

The center of this thesis is the design of a deterministic parallel execution engine of a block of transactions which utilizes the dependency graph computed from pre-execution as described in 1.4. In the execution engine, we also take thread affinity (see 2.5.1) into consideration by pinning threads to cores and scheduling transaction chains on the same core (see 3.6). We evaluated the execution engine under perfect prediction assumptions and showed that it outperforms Block-STM [25] by up to 1.4x. Furthermore, even though the project is still in progress and its evaluation is future work, I have configured the orchestration scripts for running a full system benchmark on AWS and shown the full workflow in 5.4. Finally, I present the entire system, which encapsulates the smart contract execution engine, in order to provide a more universal and comprehensive understanding.

1.5 Outline

- In Chapter 2 we provide the necessary theoretical background: We start from the fundamentals of blockchain architecture, gradually making our way to understanding the way modern blockchain systems operate.
- In Chapter 3 we describe the system model and analyze our design choices.
- In Chapter 4 we thoroughly show implementation details of our system.
- In Chapter 5 we evaluate our work, comparing it to the previous state while briefly explaining our custom workload.
- In Chapter 6 we provide a summary of our contributions as well as possible future work directions.

2 Background

In this chapter we provide the necessary theoretical background for the rest of the thesis.

2.1 Blockchain Background

2.1.1 Blockchain definition

A blockchain can be described as an unchangeable ledger designed for recording transactions. It operates within a decentralized network of peers who do not fully trust each other. Each peer possesses a copy of the ledger. Through a consensus (2.1.3) protocol, peers validate transactions, group them into blocks, and establish a chain of hashes connecting these blocks. This process organizes transactions in a sequential manner, ensuring the necessary consistency of the ledger.

2.1.2 Cryptography basics

Public Key Cryptography

Public Key Cryptography is a cryptographic method that employs a private key, kept secret, and a public key, shared with the network. This system ensures message authenticity and integrity through advanced cryptographic techniques. For instance, Alice may send a secure message to Bob over the internet using public key cryptography. First, she generates a pair of public and private keys, shares the public key with Bob, and attaches

a digital signature to her message using her private key. This signature verifies her as the message creator, and Bob confirms this using her public key.

In Bitcoin [36], public key cryptography is utilized in wallet creation and transaction signing, crucial elements of the system's operation. Bitcoin employs the Elliptic Curve Digital Signature Algorithm (ECDSA) to generate private and corresponding public keys. Users use the public key to generate public addresses which they use to send and receive funds and the private key to sign transactions, validating their authenticity.

2.1.3 Consensus

A consensus protocol is a mechanism through which a distributed network of nodes or participants in a system agrees on a single, consistent state of the system, even when individual nodes may have differing information. In the context of blockchain systems, consensus protocols ensure that all honest nodes in the network, that is nodes that are following the protocol correctly, come to an agreement about the validity and ordering of transactions. The choice of consensus protocol depends on factors like the nature of the network (public or private), the desired level of decentralization, security requirements, energy efficiency, and scalability goals. Each protocol comes with its own trade-offs and considerations.

2.1.4 State Machine Replication

State Machine Replication (SMR) is a technique to achieve consensus by replicating the execution of a state machine across nodes of a distributed system. A state machine consists of a state of the system and a program that receives a set of inputs and applies them in a sequential order using a transition function to generate an output and update the local state. Therefore, a blockchain can be seen as an example of a state machine, where the machine's state corresponds to the current status of the blockchain and executing a transaction results in a state transition. As a result, the techniques that have been used for solving the SMR problem for over 40 years have direct applications in blockchain technology. The two fundamental desired properties of an SMR protocol are consistency and liveness:

1. **Consistency** guarantees that any two honest nodes store the same prefix of state transitions in their logs, that is if an honest node executes transaction tx as its i_{th} transaction, then it is impossible for a different honest node to to execute tx'_n as its i_{th} transaction.

2. **Liveness** guarantees that every valid transaction submitted to at least one honest node is eventually added to every node's local history.

2.1.5 Permissioned vs Permissionless setting

In a public or permissionless blockchain environment, participation is unrestricted and does not require a specific identity. These types of blockchains typically incorporate a native digital currency and often use consensus based on economic incentives such as Proof of Work (PoW) or Proof of Stake (PoS) ([29]). In contrast, permissioned blockchains function within a confined network of recognized and identifiable participants sharing a common objective but which do not fully trust each other. By capitalizing on the participants' authenticated identities, permissioned blockchains can use traditional Byzantine-fault tolerant (BFT) SMR protocols such as HotStuff [6] or Tendermint [11].

2.1.6 Smart Contracts

Smart contracts represent executable computer programs that are stored and executed on a blockchain network. These contracts comprise a set of code instructions that define specific conditions. When these pre-defined conditions are satisfied, the smart contract initiates specified actions or outcomes. Evolving from the simple peer to peer transfers of the early Bitcoin protocol, most modern ecosystems operate through transactions containing smart contract code, executed in suitable execution engines such as the Ethereum Virtual Machine (EVM) ([12]).

2.2 System Model

2.2.1 Order-Execute Architecture

Most modern permissioned blockchain systems operate in the order-execute architecture, which can be described in the following three basic functions:

1. One node of the network acting as block producer orders the transactions and proposes a block to the rest in an attempt to reach consensus
2. After consensus is reached each node executes all transactions in the block using a deterministic execution engine ensuring consistent server states.
3. Each node updates its local state, according to the results of the previous execution step

2.2.2 Modular Design

Drawing inspiration from contemporary advancements ([15], [21], [45], [14]) the state of the art approach involves separating execution and consensus into distinct modular layers, with server processes divided into execution and consensus nodes. Although nodes can fulfill both roles, they can choose to select any single option, maximizing flexibility in the system.

In the consensus layer, consensus is treated as a black-box. Each consensus node generates an identical sequence of blocks $B_1, B_2, ..., B_i$ which it then passes on to the execution layer to be deterministically executed.

This separation of execution and consensus allows the application of high throughput protocols such as Narwahl structure ([15], where the consensus layer includes a data dissemination protocol and the consensus algorithm. In Narwahl ([15]), consensus nodes operate solely on metadata such as block hashes, and therefore are unable to properly validate transactions during consensus. Consequently, consensus produces a "dirty-ledger" ([45]), that is a blockchain containing potentially invalid transactions. Nonetheless, if identical blocks are received in the same order by executor nodes (guaranteed by consensus), a deterministic execution algorithm guarantees that the same transactions will be invalidated.

2.3 Multi-Version Deterministic Databases

Many ideas that ultimately led to the development of parallel smart contract transaction execution engines stem from database literature.

Deterministic databases: Deterministic databases refer to databases where the operations and results are entirely predicable and consistent, ensuring that given the same initial state and a sequence of operations, the final state of the database will be the same regardless of the execution environment or the order of operations. In other words, the behavior of a deterministic database is entirely determined by the order of its inputs. Deterministic databases hold appeal for various reasons. Their pre-defined transaction order guarantees serializable execution, avoiding deadlocks and aborts related to concurrency control. The absence of aborts ensures strict serializability as transactions consistently read coherent data. Moreover, deterministic execution lessens the reliance on two-phase commit, enhancing the scalability of distributed transaction throughput. Additionally, it removes complexity from replication and recovery processes, as transactions can be replayed in a deterministic manner.

Multi-version databases: The simplest design of a deterministic database executes all transactions serially in the predefined order. Although producing correct results, this obviously does not utilize the parallelism available on modern multi-core systems.
A multi version database keeps track of multiple versions of data items over time. In a multi-version database, whenever a data item is updated, a new version of that data item is created, instead of overwriting the existing data. This is key in increasing the amount of concurrency in transaction processing because reads do not block writes.

Like blockchain systems, databases also commonly face skewed and contended workload in web applications that encounter unpredictable demand spikes (i.e holiday sales, discounts on specific items). One approach used by concurrency control protocols such as BOHM [20] is to partition the resources across the cores of the system as seen in 2.1. However, it should be noted that partitioning suffers heavily under uneven resource access.

To manage load imbalance, databases can employ a shared memory architecture and distribute transactions across the cores of the system.

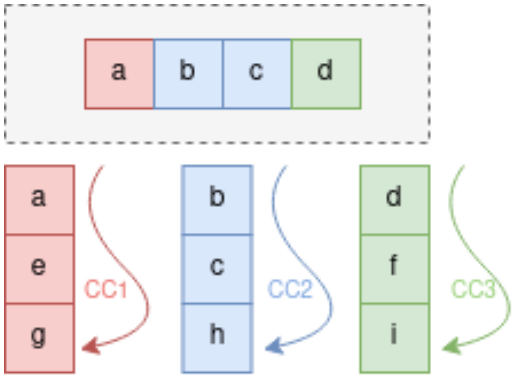


Figure 2.1: *Intra transaction parallelism as used in BOHM. Each concurrency control thread is responsible for a partition of resources.*

2.3.1 Optimistic Concurrency Control

Optimistic Concurrency Control (OCC) is a technique introduced to database systems by H.T Kung and John T. Robinson [31].

The term ”optimistic” in this context refers to the approach's hopeful assumption that conflicts between transactions will be rare. Rather than locking data resources during the entire transaction, as in pessimistic concurrency control, OCC allows transactions to perform operations without such restrictions. The system records a ”backup” of the transaction's changes, often using version numbers, timestamps, or similar mechanisms, in case the transaction needs to be rolled back due to conflicts. Although this approach is predominately followed in non-deterministic databases, enforcing a preset serialization makes the outcome deterministic. The typical phases of OCC are as follows:

1. **Read Phase:**
    - Transactions perform resource reads from the database in parallel, without acquiring locks.
    - The system records a snapshot of the data's state (often using version numbers or timestamps), which serves as a reference for detecting conflicts later.
2. **Modify Phase:**
    - Transactions perform their necessary modifications to the data (writes) without locking accessed resources.

- The system locally records changes as tentative modifications that are not immediately applied to the database.

3. **Validation Phase:**

   - Before committing, the system checks for conflicts by comparing the recorded snapshot with the current state of the data.
   - If the data has not been modified by other transactions, the transaction is conflict-free and can commit.
   - If conflicts are found, the transaction might need to be aborted or restarted.

The ”optimistic” nature of OCC leads to potentially higher performance, better resource utilization, and reduced contention for locks. However, that is only under the assumption that conflicts are indeed rare, as multiple transaction roll-backs sufficiently limit performance.

Aria/Eve: An approach from the database literature is Aria [35] which executes transactions optimistically, and aborts transactions on conflict to be re-executed in a later batch. In order to reduce abort rates, it has an additional reordering phase where it attempts to deterministically reorder transactions to avoid read-after-write aborts. However, similar to Eve [28], Aria does not deal well with the highly contended workloads we expect in this setting leading to high abort rates.

Caracal: Caracal [41] adopts a two-phase processing model, where transactions are batched into epochs and processed through initialization and execution phases. In this approach, the initialization phase manages concurrency control for all transactions within an epoch, while the execution phase actually executes these transactions.

By default, Caracal [41] assigns transactions to processor cores in a round-robin manner. Each core takes care of both initializing and executing the transactions assigned to it. Alternatively, different assignment policies can be used for load balancing or improving data locality.

In a scenario with four transactions, Caracal’s initialization phase uses the transactions’ write sets (sets of rows to be inserted, updated, or deleted) to create pending row versions. These versions act as placeholders with no actual data yet. Shared initialization

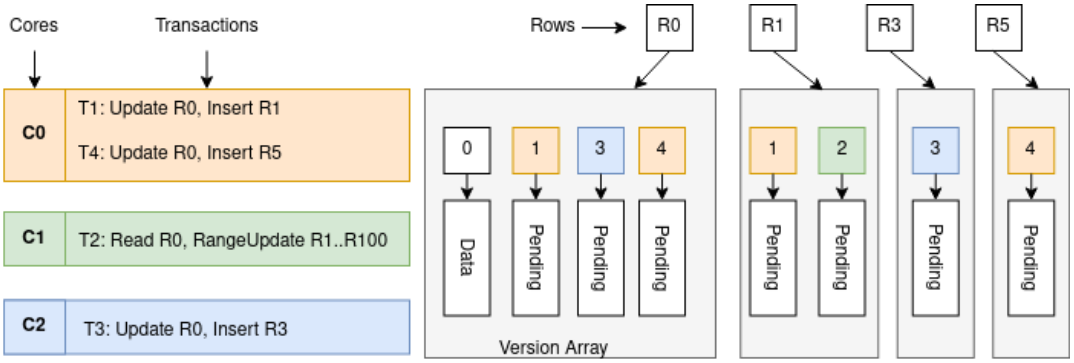


Figure 2.2: *Transaction processing in Caracal.*

is a key feature allowing transactions on different cores to create versions for the same row, thus minimizing the impact of skewed data access patterns.

During the execution phase, writers can update these pending row versions without needing synchronization. This approach is made safe by preventing transactions from aborting after their initial write, thus ensuring that transactions only read committed data. Readers are able to observe all row versions that will be generated in the epoch (due to initialization), enabling them to correctly determine the appropriate previous version to read based on the serial order. This ensures readers consistently access the latest committed version of the data.

For instance, in the depicted example in 2.2, with transactions and row versions, a reader synchronizes with a writer by waiting when a pending version is encountered, allowing the reader to proceed once the version is written. Due to the deterministic nature of execution, there are no deadlocks, and no aborts related to concurrency control, establishing a streamlined and efficient process.

2.4 Parallel Smart Contract Execution

As we mentioned in 1.2 the serial execution of smart contract transactions has become a serious bottleneck and hinders the widespread adoption of blockchains. Applying techniques directly from the database literature is not feasible due to the fundamental differences between databases and smart contracts. In a database context, the write-sets of transactions are generally predetermined and known in advance. However, this

assumption does not hold in the realm of smart contracts, as these contracts can embody complex and diverse logic. At present, an increasing number of projects are providing the ability to execute smart contract transactions in parallel, however as seen in 1.3 their parallel execution techniques vary.

```
function transfer(address _receiver, uint256 _amount) /*...*/
    {
        balances[msg.sender] -= _amount; //
        balances[_receiver] += _amount;
    // ...
}
```

Listing 2.1: *Solidity counters, a common source of contention*

2.4.1 Pessimistic Approaches

Pessimistic Approaches avoid conflicts altogether by requiring transactions to specify all the resources that they may read or write during their lifetime, enabling the blockchain to schedule transactions concurrently without requiring mechanisms for validation or re-execution.

This approach was first proposed in an Ethereum Proposal [3] and was since then deployed in a popular layer-2 solution called FuelVM [22]. Aside from FuelVM, Solana [44] is a blockchain that also leverages this approach. However, this requires smart contract developers to add hints, which can be a burden on the developer and may result in overly pessimistic locking, as in some cases it might not be possible to determine ahead of time which resources will be accessed during the transaction's runtime. Furthermore, similar to optimistic approaches, the performance is still inherently limited by the highly contended workloads we outlined earlier.

FSC [34] attempts to reduce this level of pessimism by combining developer hints to discover hot and cold resources in smart contracts with static analysis to obtain the read and write sets of transactions and schedule them accordingly. This avoids overly pessimistic locking but may result in re-executions when cold resources are accessed concurrently. However, they only use this information for scheduling and are still subject to long sequential paths slowing down the system significantly. Furthermore, they

obtain this information on the critical path of consensus which presents a performance overhead.

2.4.2 Optimistic Concurrency Control in Smart Contract Execution

Similarly to the case of databases, optimistic concurrency control been leveraged in designs aiming to increase performance. Software Transactional Memory (STM) libraries with OCC ([43], [27]) record memory accesses, validate transactions after execution to identify conflicts, and use abort and re-execution to ensure data consistency.

2.4.3 Block-STM

By default, STM libraries do not guarantee the same results when transactions are executed multiple times, something that would make them unsuitable for the use case of Blockchain systems, as validators need to end up on the same final state. However, there has been work on Deterministic STM libraries which guarantee that execution has a predefined serialization order, equivalent to sequentially executing every transaction in the block. Block-STM [25] (Block-level Software Transactional Memory) is one such parallel execution engine that efficiently executes a block of transactions while managing dependencies and coordination among threads.

The input to Block-STM is a block of transactions, consisting of n transactions. This block defines a predetermined serialization order $tx_1 < tx_2 < ... < tx_n$. The objective is to execute these transactions within the block, resulting in a final state equivalent to that achieved by executing the transactions sequentially in the order $tx_1, tx_2, ..., tx_n$. Each transaction in Block-STM might undergo multiple execution iterations, referred to as incarnations. An incarnation is considered aborted when a subsequent execution with an incremented incarnation number is necessary. A version consists of a transaction index and an incarnation number.

Block-STM employs an in-memory multi-version data structure to support concurrent reads and writes by transactions. This data structure stores the latest value written for each memory location along with the associated transaction version. When a transaction tx_i reads from a memory location, it retrieves the value written by the highest

transaction preceding tx_i in the preset serialization order, along with the corresponding version. Transactions can read values even if subsequent transactions have made updates, as long as those updates are from higher-indexed transactions. If no prior transaction has written to a location, then a read is resolved from storage based on the state before the current block execution.

For each incarnation, Block-STM maintains a read-set and a write-set. The read-set contains memory locations read during the incarnation and their corresponding versions. The write-set includes the updates made by the incarnation in the form of (memory location, value) pairs. At the end of execution, the write-set is applied to shared memory (the multi-version data structure). After execution, an incarnation undergoes a process of validation which entails re-reading the read-set and comparing the observed versions. A successful validation indicates that the updates made by the incarnation are still valid, while a failed validation causes the incarnation to abort. Furthermore, as an optimization, the write-set of an aborted incarnation is used as an estimation of the write-set of the next one preventing following transactions from likely aborts.

2.4.4 Pre-execution and Dependency Detection

Previous research [7], [9] and [18] has taken advantage of the unique characteristics of the blockchain use-case to enhance the performance of STM. Their strategy involves pre-computing dependencies, creating a directed acyclic graph that represents the transactions. These transactions can then be executed using a fork-join schedule. The outcome is a schedule that is conscious of dependencies and therefore avoids aborts/re-executions due to conflicts.

In cases where entities are incentivized to record and share this dependency graph, there is a potential to reduce the overhead of pre-computation for some entities. In other words, by having some entities responsible for computing and distributing the dependency graph, other entities may be able to skip this step leading to improved efficiency.

The ”miner-replay paradigm,” explored in [18], involves miners parallelizing block execution using a white-box STM library application. This application produces the serialization order, represented as a ”fork-join” schedule and transmits it alongside the new block proposal from miners to validators through the consensus process. Validators

then utilize the fork-join schedule to deterministically replay the block. ParBlockchain [7] introduced the ”order-execute paradigm” (OXII) for deterministic parallelism. The ordering phase resembles the schedule preparation from [18], while the transaction dependency graph is computed without actual block execution. OXII relies on either the read-write set being known in advance through static analysis or speculative pre-execution to generate the dependency graph among transactions.

OptSmart [9] introduces two improvements. Firstly, the dependency graph is compressed to include only transactions with dependencies, allowing parallel execution of transactions not included in the graph. Secondly, multi-versioned memory is used during execution to avoid write-write conflicts. Hyperledger Fabric [8] and related works [42] follow the ”execute-order-validate paradigm.” This design departs radically from the order-execute paradigm in that Fabric executes transactions before reaching final agreement on their order. This consequently enables the execution phase to abort transactions that would lead to unserializable outcomes before they are ordered. Forerunner [13] follows a Dissemination-Consensus-Execution (DiCE) paradigm where transactions are speculatively pre-executed outside the critical path to provide hints for the final execution process.

2.5 Hardware background

2.5.1 Thread Affinity

Thread affinity refers to the concept of associating a particular thread in a multi-threaded program with a specific central processing unit (CPU) or processor core. This affinity ensures that the thread primarily runs on the designated CPU core.

Thread affinity can be advantageous in several scenarios:

1. **Performance Optimization**: By assigning threads to specific cores, the system can optimize the use of CPU caches and reduce cache coherence traffic, resulting in better performance. When a thread consistently runs on the same core, it can take advantage of the cache locality.

2. **Predictable Performance**: Thread affinity can help achieve more predictable and

deterministic performance. In some real-time systems or applications with strict timing requirements, controlling which threads run on which cores can be critical.

3. **Resource Isolation**: In a multi-threaded or multi-process environment, the isolation of certain threads or tasks from others to prevent interference may be required. Assigning thread affinity can help ensure that critical tasks are not interrupted by less critical ones from the operating system scheduler.

4. **Avoiding Context Switching Overheads**: When threads switch between different cores, it incurs some overhead due to context switching. Thread affinity can reduce these context switching costs by keeping threads on the same core.

It's important to note that while thread affinity can provide performance benefits in some cases, it can also have negative effects. Overusing thread affinity or binding threads too tightly to specific cores can lead to under-utilization of available CPU resources, especially in systems with dynamic workloads. The specific mechanisms for setting thread affinity can vary depending on the operating system and programming language being used. Many modern operating systems provide APIs or tools for managing thread affinity, allowing developers to control which cores threads are assigned to.

3 Design

3.1 System Model

Before diving into our design, we describe the system model on top of which we build.

Processes: We assume the existence of a set of n server processes $p_1, p_2, ..., p_n$ and a set of l client processes $c_1, c_2, ..., c_l$ communicating over a peer-to-peer network under the Public Key Infrastructure Assumption (PKI) where clients and servers are identified through their public keys (see 2.1.2) and entities prove their identity by signing their respective transactions and messages with their private keys. Furthermore, clients and servers communicate over perfect point-to-point channels. As a result, if a process p_i sends a message m_{ij} to process p_j, process p_j will eventually receive m_{ij}.

Network and failures: To deal with network failures, we assume that the network follows a partial synchrony model based on [19]. During periods of asynchrony, messages may be delayed for an arbitrary amount of time. However, after some unknown GST (Global Stabilization Time), message transmission is bound by a known Δ and progress can be made in a deterministic manner. Furthermore, we assume a Byzantine fault model with both Byzantine servers and clients. Byzantine servers correspond to the Byzantine fault model inherited from the chosen consensus framework, with the number of faulty nodes limited to $f \leq n/3$ for most permissioned consensus algorithms for non-faulty replicas to agree on the same transactions in the same order. In this context, a process is considered correct when it adheres to the predefined protocol, while any deviation from this protocol categorizes it as faulty. Meanwhile, Byzantine clients are assumed to possess limited financial and computational resources and are required to

pay transaction fees for their transactions to be included in the blockchain.

Function decoupling: Following the structure of modern approaches [15], [21] the system's architecture separates execution and consensus into distinct layers. Server nodes are divided into two types: *execution* nodes and *consensus* nodes. Although any node can serve both roles, this setup provides flexibility. This decoupling permits the application of the structure introduced in Narwahl [15], where the consensus layer is divided into (a) a data dissemination protocol, and (b) the actual consensus algorithm. This structure considerably accelerates processing compared to traditional methods, ensuring robust consensus across the distributed network.

Consensus: Regarding the consensus layer, we treat consensus as a black box, where each consensus node produces an identical chain of blocks (e.g., $B_1, B_2, ..., B_i$), executed sequentially by all executor nodes. This approach ensures that all nodes reach the same state as long as execution remains strictly deterministic. In Narwahl [15], consensus nodes lack awareness of the system state and operate solely on metadata (e.g., block hashes). Consequently, client transactions cannot be fully validated during consensus. Thus, consensus generates a *dirty ledger* 2.2.1 containing potentially invalid transactions. However, since executors receive the same blocks in the same order (via consensus) and execute them deterministically (invalidating identical transactions), the final state remains consistent across all executors.

Execution: To ensure deterministic outcomes in parallel execution and maintain safety, we adopt a structure similar to Block-STM[25], where transactions are processed in two distinct phases. In the execution phase, transactions are executed, and in the validation phase, the execution results are checked for conflicts due to data dependencies, and transactions are rolled back if necessary and rescheduled for execution.

3.2 Overview

Based on our thorough workload analysis, led by Ray Neiheiser, and our understanding of how block composition influences performance, we formulated clear design goals for our system. First, we aimed to avoid OCC (2.4.2) parallel execution as it does not perform well in contented workloads. Also, we refrained from a pessimistic approach (see 2.4.1) that burdens smart contract developers or users. In addition, our objective

was to create an algorithm for proposers that fills blocks with transactions in a manner that optimally utilizes CPU resources on executor nodes.

The primary strategy employed in our system to achieve the stated goals involves a "loosely coupled" pre-execution phase, which serves to create essential metadata for subsequent "concurrency-aware" execution.

In this section, we outline our approach in three parts. Initially, we detail how our presented system constructs what we term "Good Blocks", which are designed to maximize concurrent execution efficiency. Subsequently, we explain how our system executes pre-execution tasks outside the critical consensus path, thereby facilitating the separation strategy outlined in the system model. Finally, we analyze our design choices for scheduling transactions across cores, attempting to reduce intra-CPU traffic due to cache coherence protocols.

3.3 Good Blocks

In the context of concurrent transaction execution, the composition of a block can significantly impact the system's throughput. This is exemplified by the fact that in an optimistic parallel execution engine, a workload that is entirely sequential may take even longer to process than if it was executed on a single core [25].

We introduce the term "Good Block" to refer to a block that can be efficiently processed in parallel.

In blockchain systems that support smart contract execution, like Ethereum [12], there is typically a global parameter that sets an upper bound on the computational complexity of a block. This parameter represents the maximum computational capacity of the system, as exceeding it can lead to adverse effects such as Denial of Service (DoS) attacks. Block producers ensure that this limit is not surpassed when adding transactions to a block, maintaining the efficiency and security of the blockchain network. For instance, Ethereum uses the concept of Gas to estimate transaction complexity, and a maximum gas parameter acts as an upper bound on a block's complexity.

However, parallel execution introduces additional complexity as a single complexity parameter is insufficient to estimate the time needed to execute a block. Analyses of

Ethereum's workload reveal that while many transactions can run concurrently without conflicts, the majority of blocks are bottlenecked on a single chain of dependent transactions that need to be executed serially and thus dominate the overall execution time [23]. Consequently, execution time can vary significantly depending on the length of these chains.

To estimate computational complexity accurately, we employ two system parameters to describe the maximum block size. First, a maximum gas limit denoted as gas_{max}, which is the maximum amount of gas that all the transactions in the block are allowed to consume, similar to Ethereum's maximum gas limit. Second, a concurrency parameter c_{max} represents the system's ability to process transactions in parallel, such as the number of available processor cores. In this context, the total system capacity can be expressed as $c_{max} * gas_{max}$, and we aim to construct blocks guaranteed to execute within gas_{max}.

To achieve this, we identify three requirements that a block b_i must meet to ensure its execution gas b_i^{gas} never exceeds gas_{max} and we label such a block a "Good Block".

To construct a "Good Block," we require knowledge of the runtime complexity of all transactions and information regarding their inter-dependencies. We will elaborate on how we acquire and process this information in the following section.

3.3.1 Good Block Construction

The system obtains contextual transaction information by having the block producer execute each transaction before including it in the current block. We call this "Pre-Execution" of transactions and we outline a straightforward pre-execution algorithm that produces "Good Blocks" and obtains the necessary metadata in Algorithm 1.

Algorithm 1 *Block Creation*

```
1: cap_total ← ⊥
2: cap_curr ← 0
3: ch_res ← ∅                                              ▷ Set to track transaction chains
4: block ← ∅                                               ▷ Current block
5: procedure CreateGoodBlock(gas_max, c_max, txs)
6:     gas_max ← gas_max / 2
7:     cap_total ← gas_max * c_max                         ▷ Calculate the maximum capacity
8:     for all tx ∈ txs do                                 ▷ Iterate over transactions
9:         writeset, readset, gas ← run(tx)
10:        txchain ← 0                                     ▷ Longest chain length
11:        for all read ∈ readset do                       ▷ Iterate over readset
12:            if read ∈ ch ∧ ch_read > txchain then       ▷ Find longest chain
13:                txchain ← ch_read
14:            end if
15:        end for
16:        if txchain + gas > gas_max ∨ cap_curr + gas > cap_total then
17:            continue                                    ▷ Skip transaction inclusion
18:        end if
19:        cap_curr ← cap_curr + gas                       ▷ Track current capacity
20:        block ← block ∪ (tx, readset, writeset, gas)    ▷ Add tx to Block
21:        for all write ∈ writeset do                     ▷ Iterate over writeset
22:            if write ∉ ch ∨ ch_write < txchain then
23:                ch_write ← txchain                      ▷ Note new chain length
24:            end if
25:        end for
26:    end for
27: end procedure
```

The above algorithm can be easily broken down into the following steps:

1. Begin by limiting the gas_{max} to half of its original value to shorten the longest acceptable dependency chain.

2. Calculate the total block capacity by multiplying the adjusted maximum gas limit gas_{max} with a concurrency parameter c_{max}.

3. For each transaction within the block, do the following:

   a. Execute the transaction to extract information such as the read-set, write-set, and gas usage.

   b. Analyze the read-set to determine the longest dependency chain, utilizing a data structure ch_{res}.

   c. Perform two checks for each transaction:

      i. Verify whether the longest dependency chain exceeds a predetermined limit for sequential execution.

      ii. Check if adding the computational load of the transaction would surpass the total block capacity.

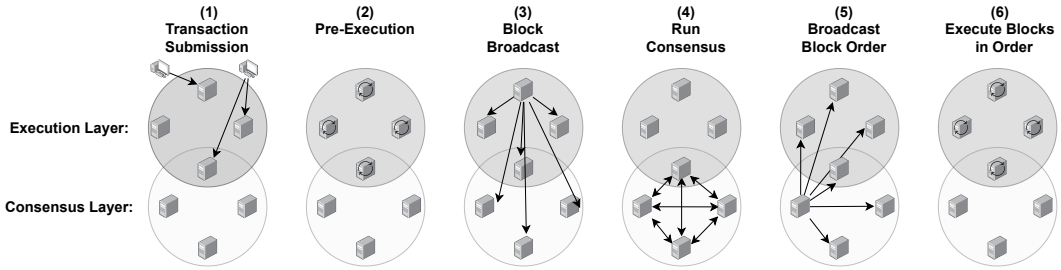


Figure 3.1: *Transaction life cycle in Loose Coupling*

Although speculative pre-execution, if on the critical path of consensus, introduces additional computational overhead, next we demonstrate how we can distribute and amortize this cost leading to a full system speedup.

3.4 Loose Coupling

In the fully decoupled system described in the system model, where blocks are produced by consensus nodes, the block producers lack the state that is necessary to pre-execute transactions as executing blocks and updating the state is completely seperated from consensus. Therefore, the main challenge in designing our system is to enable pre-execution while maintaining the benefits of decoupling.

As a first step, instead of having consensus nodes that lack state knowledge gather, batch, and disseminate client transactions, this responsibility is moved to the execution layer, where execution nodes produce "Good Blocks" of transactions. As a result, the consensus layer will receive their input from the execution layer instead of receiving it from the client. We call this approach to decoupling "Loose Coupling".

The full life cycle of a transaction is shown in Figure 3.1. The process starts with clients interacting with nodes on the execution layer. Subsequently, those nodes pre-execute and pre-validate transactions (i.e., check for transaction fees, verify client signatures, etc.), batch them into "Good Blocks" and include metadata such as the transaction dependencies and runtimes in the block (step 1).

Following that, in the third step, the executor nodes enter the dissemination phase and broadcast their individual blocks to all nodes concurrently.

Consensus then operates on the block hashes without requiring state knowledge sim-

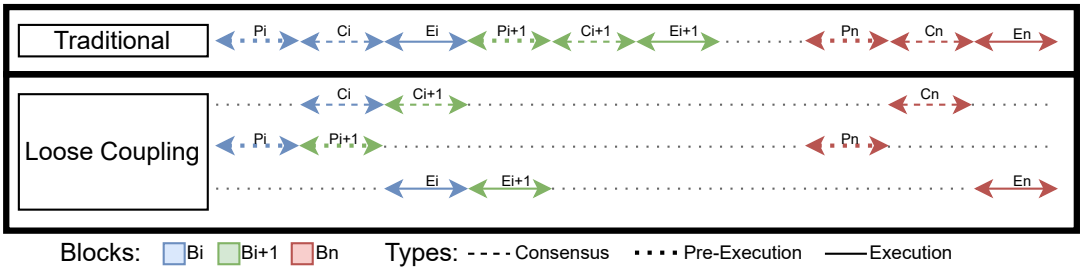


Figure 3.2: *Relationship between Pre-executions P_i, Executions E_i and Consensus instances C_i with and without decoupling.*

ilarly to Narwhal [15] (step four). In this context, we treat consensus as a black box where any consensus algorithm can be used depending on the specific requirements of the system. After consensus terminates, the consensus nodes then broadcast the resulting block order to the executor nodes (fifth part) which then execute the blocks deterministically in order (last part).

By inserting the execution layer between the client and the consensus layer we can fully leverage the liveness and safety guarantees that were established in Narwhal [15] as the consensus layer remains unchanged. We solely move the block broadcast to the execution layer such that the node that produces the block can pre-execute transactions. As such, as long as there is at least one honest executor node that correctly broadcasts its block to a majority of consensus nodes, consensus will consistently produce an ordered chain of blocks. Inversely, any block that is not broadcast to a majority of consensus nodes will not be output by the consensus mechanism.

Even when a block output by the consensus mechanism has been correctly broadcast to a majority of consensus nodes, an execution node might not have correctly received the block yet. However, execution nodes can query missed blocks from any consensus node in a trustless fashion once they received a signed quorum on the block hash.

3.5 Optimistic Block Production

By pre-executing transactions outside of the critical path of consensus we can "hide" the performance impact of pre-execution across the nodes of the network. We show the interplay of consensus C_i, execution E_i, and pre-execution P_i for different blocks

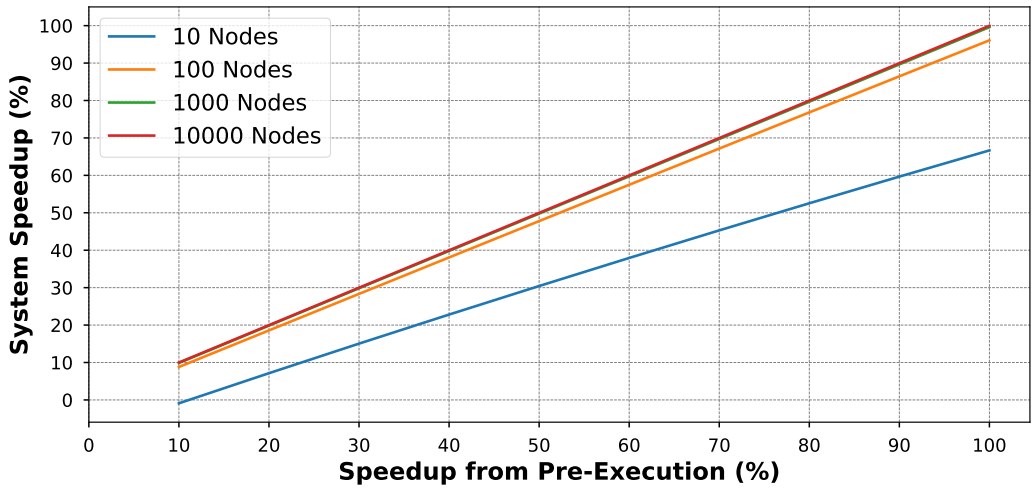


Figure 3.3: *Expected Translation from Pre-Execution Speedup to System Speedup*

$B_i, B_{i+1}, ..., B_n$ in Figure 3.2, comparing the traditional with the loosely coupled deployment. While in a traditional deployment pre-execution has to run on the critical path of consensus which slows down the overall system, in the loosely coupled approach pre-execution runs concurrent to consensus and execution and the cost is distributed over all executors. As such, with an increasing set of executors, even a minimal speedup obtained from the speculative execution quickly compensates the computational overhead of the pre-execution.

For example, consider a system with N executors and the last produced block b_i at round i of consensus. In this context, each executor had to pre-execute $\frac{i}{N}$ blocks and execute all i blocks with the given speedup u by leveraging the metadata we've obtained from the speculative pre-execution. This results in a total cost of $\frac{i}{N} + \frac{i}{u}$ at each executor. Based on this equation, as the number of executors in the system increases, the more the pre-execution cost is diluted among the nodes and the total system speedup converges to the ideal pre-execution speedup.

We simulated this for different values of N and u and the results are shown in Figure 3.3. In a system with a small number of executors (e.g., 4-10) the total speedup of the overall system is only a fraction (around half) of the execution speedup that our mechanism offers. This happens because the potential block proposer pool is limited and the same nodes pre-execute multiple blocks. However, at or above 100 nodes the execution speedup translates almost fully to an overall system speedup as each node

pre-executes a small number of transactions. There can be cases where we obtain no execution speedup by pre-execution, for example when the workload can be run fully in parallel. The overhead is however minimal with a sufficient number of nodes, even in these edge cases.

While *Loose Coupling* allows offsetting the pre-execution cost, depending on the system conditions, the gap between the execution E_i of an instance i and the pre-execution of P_{i+1} can span several blocks. For example, in Figure 3.2 the pre-execution P_i runs roughly two slots ahead of the execution of the previous round E_{i-1}.

Consequently, the pre-execution process is performed with partially stale data, which can result in outdated metadata being passed to the execution phase, leading to a slowdown in the execution protocol or an increase in the number of aborted transactions within a block.

However, based on our workload analysis, we argue that even with partially stale data, we will still, with high confidence, produce an accurate dependency graph and runtime estimates. This is true in most popular applications. because even though data might be partially stale, the extraction of the dependencies is still accurate with a high likelihood: When handling Peer-to-Peer Transactions and DEX Trading, any valid transaction (i.e., when the user has sufficient gas and currency) will touch the same resources regardless of the current state (i.e., touch the user's balance and the currency-pair pool). In this case, the staleness of the metadata from pre-execution does not affect the resulting dependency graph in no way. As for NFT Minting, this is also true until the NFT is exhausted (no more NFTs can be minted after this period). At this point, for a short period, stale state can lead to false positives (e.g. detecting resource accesses that will not happen), similar to invalid transactions, but never to false negatives. The same also applies to on and off-boarding layer-2 solutions which also make up a significant portion of the system load [37]. Therefore, a block producer may overestimate dependencies, and, as a result, overestimate the execution time of a block, but much less likely, underestimate the execution time of a block.

This insight is important as it allows us to execute all transactions fully in parallel during the pre-execution phase, ignoring potentially state-changing interdependencies between transactions. As a result, the cost of pre-execution is minimized as it scales fully with the number of available cores.

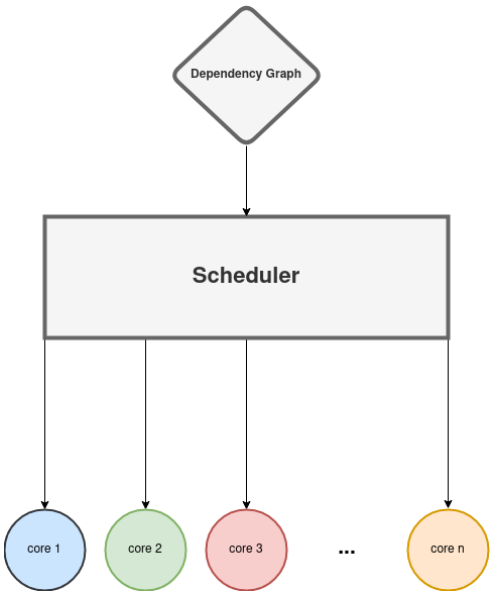


Figure 3.4: *The role of the transaction scheduler*

3.6 Scheduler

Our pre-execution approach under the "loose coupling" model requires changes to the execution engine.

First, we introduce a mechanism that generates an execution schedule using the information obtained from pre-execution. During this process, each node uses the dependency graph and the estimated gas cost of each transaction to deterministically schedule transactions among cores in a manner that avoids conflicts. In our system, a dedicated thread is responsible for scheduling all transactions of a block to all other threads according to a scheduling algorithm presented below. We chose to schedule all transactions of a block at the beginning of processing (static scheduling) due to the general small execution time of a block. A dynamic approach tends to increase the time complexity of the scheduling algorithm and in the case of fast block execution faces the risk of scheduling becoming the bottleneck, as we saw happen in benchmarks.

3.6.1 Model

Let $G = (V, E)$ be a directed acyclic graph (DAG), where the vertices in the set V represent transactions (tasks), and the edges in the set E represent the dependency con-

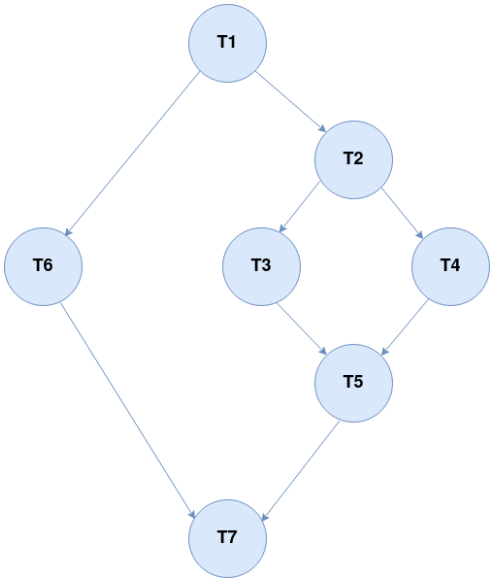


Figure 3.5: *Transaction dependencies depicted in a DAG*

straints (read-write) between those transactions. For example, if transaction t_y reads the results of transaction t_x with $x < y$ the edge $t_x \rightarrow t_y$ would exist in the graph, that is $(v_x, v_y) \in E$. Let $n = |V|$ be the total number of vertices. We use $Pred[v_i] = \{v_j | (v_j, v_i) \in E\}$ to represent the (immediate) predecessors of a vertex $v_i \in V$, and $Succ[v_i] = \{v_j | (v_i, v_j) \in E\}$ to represent the (immediate) successors of v_i in G. Vertices without any predecessors are called source nodes, and the ones without any successors are called target nodes. Every vertex $v_i \in V$ has a weight, denoted by w_i which represents the predicted execution time of transaction t_i.

In the computing model, there are P identical processing units, referred to as cores $(p_1, \ldots, p_P)$. Inter-core communication is achieved through a shared memory mechanism. Each transaction must be scheduled onto a core while adhering to the dependency constraints. The tasks are non-preemptive and atomic, meaning that a processor executes at most one transaction at a given time. Given a specific mapping of tasks onto the computing platform, let $\mu(i)$ denote the index of the processor on which transaction t_i is mapped. In other words, t_i is executed on processor $p_{\mu(i)}$. For every transaction $t_i \in V$, its weight w_i represents the time required for its execution on any core of the system. Moreover, if there exists a dependency constraint between two transactions mapped onto different cores, i.e., $(t_i, t_j) \in E$ and $\mu(i) \neq \mu(j)$, data must be "transmitted" from the cache of $p_{\mu(i)}$ to $p_{\mu(j)}$. The time required for this data transfer is represented by the

edge cost $c_{i,j}$. In practice, this time corresponds to cache coherence traffic among processors. Cache coherence traffic refers to the communication and data transfers that occur between multiple caches in a multiprocessor or multi-core system to maintain cache coherence. By taking cache coherence overhead into account, we attempt to keep dependency chains on the same core and have better data locality.

In multi-core or multiprocessor systems, each core typically has its own cache memory, which stores copies of data from the main memory. Cache coherence ensures that when one core updates a memory location, any other core reading or accessing the same memory location sees the most up-to-date value. Cache coherence traffic is managed by cache coherence protocols like MESI (Modified, Exclusive, Shared, Invalid), MOESI (Modified, Owner, Exclusive, Shared, Invalid), and others. These protocols define how caches communicate and coordinate their actions to maintain data consistency and minimize unnecessary data transfers.

We define a schedule for graph G as follows: each task $v_i \in V$ $1 \leq i \leq n$ is assigned to a processing unit $p_\mu(i), 1 \leq \mu(i) \leq P$.

3.6.2 Algorithm

The algorithm we chose is a modified implementation of the simple heuristic of *BL-EST* [38].

This heuristic operates by maintaining a well-ordered list of tasks that are ready for execution, which consists of all transactions for which all dependencies have already been successfully executed. We denote the set of tasks that have been executed as Ex, and the set of currently ready tasks as $Ready$. Initially, Ex is empty, and $Ready$ consists of tasks $v_i \in V$ that have no predecessors, expressed as $Pred[v_i] = \emptyset$. However, as tasks are executed, new tasks may become ready. At any given moment, the set $Ready$ is defined as follows:

$$Ready = \{v_i \in V \setminus Ex \mid Pred[v_i] = \emptyset \text{ or } \forall (v_j, v_i) \in E, v_j \in Ex\}.$$

During the first phase of our algorithm we assign tasks a heuristic priority, which is determined to be their "bottom level," hence the acronym BL. The bottom level $bl(i)$

assigned to a task $v_i \in V$ is calculated as the maximum weight along the path from v_i to a target node, which is a vertex of the DAG without any successors. Formally:

$$bl(i) = w_i + \begin{cases} 0 & \text{if } Succ[v_i] = \emptyset; \\ \max_{v_j \in Succ[v_i]} bl(j) & \text{otherwise.} \end{cases} \tag{3.1}$$

In the second phase, transactions are assigned to processing cores. At each iteration, our algorithm selects the task of the $Ready$ set with the highest priority and schedules it on the core that would result in the earliest start time of that task. To calculate this time, we iterate over the processors: for each processor, the start time depends on the time when that processor becomes available from other scheduled executions and the finish time of the predecessors of that transaction. If a preceding transaction was scheduled on a different core, an extra cost is added to the calculation to represent cache coherency traffic. We also keep track of the finish time of each processor P_k ($comp_k$) for future rounds of the algorithm.

This heuristic is called BL - EST, for Bottom-Level Earliest-Start-Time, and is described in Algorithm 2. The $Ready$ set is stored in a max-heap structure for efficiently retrieving the tasks with the highest priority, and it is initialized at line 6. The computation of the bottom levels for all tasks (line 5) can easily be performed in a single traversal of the graph in $O(|V| + |E|)$ time. The main loop traverses the DAG and tentatively schedules a task with the largest bottom level on each processor in the loop lines 12-24. The processor with the earliest start time is then saved, and all variables are updated on lines 25-26. Finally, the list of ready tasks is updated in line 27, i.e., $Ex \leftarrow Ex \cup \{v_i\}$, and new ready tasks are accordingly inserted into the max-heap.

The total time complexity of Algorithm 2 is hence $O(|V| \log |V| + p|E|)$:

- $|V| \log |V|$ for the heap operations (we perform $|V|$ times the extraction of the maximum, and the insertion of new ready tasks into the heap).
- $p|E|$ for lines 14-20 as we find the processor with the earliest start time

The space complexity is $O(p + |V| + |E|)$.

Algorithm 2 Modified BL - EST algorithm

procedure BLEST($G = (V, E)$, p)
    // Data: Directed graph $G = (V, E)$, number of processors p
    // Result: For each task $v_i \in V$, allocation $\mu(i)$ and start time $st(i)$
    // For each $(v_i, v_j) \in E$, delay time $com(i, j)$
    $bl \leftarrow$ ComputeBottomLevels(G)
    $Ready \leftarrow$ EmptyHeap
    Insert v_i in $Ready$ with key $bl(i)$ for all v_i without any predecessors
    for $k = 1$ to p **do**
        $comp_k \leftarrow 0$; $send_k \leftarrow 0$; $recv_k \leftarrow 0$
    end for
    while $Ready$ is not empty **do**
        $v_i \leftarrow$ extractMax($Ready$)
        Sort $Pred[v_i]$ in a non-decreasing order of the finish times
        for $k = 1$ to p **do**
            $begin_k \leftarrow comp_k$
            for $v_j \in Pred[v_i]$ **do**
                if $\mu(j) = k$ **then**
                    $begin_k \leftarrow st(j) + w_j + c_{j,i}$
                else
                    $begin_k \leftarrow st(j) + w_j$
                end if
            end for
        end for
        $k^* \leftarrow$ argmin$_k\{begin_k\}$ // Best Processor
        $\mu(i) \leftarrow k^*$
        $st(i) \leftarrow begin_{k^*}$
        Insert new ready tasks into $Ready$
    end while
end procedure

In combination with utilizing thread affinity 2.5.1, that is assigning threads to specific cores, we aim to take advantage of cache locality and reduce cache coherence traffic.

4 Implementation

We integrated our system with Aptos [21] and Block-STM [25]. First, we injected pre-execution in the block-building phase, including the creation of "Good Blocks" and the addition of meta-data to the blocks. Second, we altered the execution algorithm to take dependencies and transaction runtime into account and to verify transaction signatures during idle-time of cores rather than on the critical execution path.We chose Aptos as it is high quality project that already follows a decoupled consensus and execution model.

4.1 Overview

We now present the key components present on a node running Aptos:

Consensus: The Aptos blockchain uses a consensus protocol named AptosBFT which is based on Jolteon [24] and HotStuff [6].

Execution: The execution engine used to execute transactions in parallel is a direct implementation of Block-STM [25] as described in 2.4.3.

Mempool: Mempool is a shared buffer that holds the transactions that have been submitted to the system but are not yet agreed on by consensus or executed.

These components interact with each other in the following ways:

(1) Mempool ↔ Other Validators: When a new transaction is added to the mempool by a client request, the mempool shares this transaction with other validator nodes in the system. When a validator receives a transaction from the mempool of another validator, it adds it to the local storage of the "shared mempool" of the recipient validator.

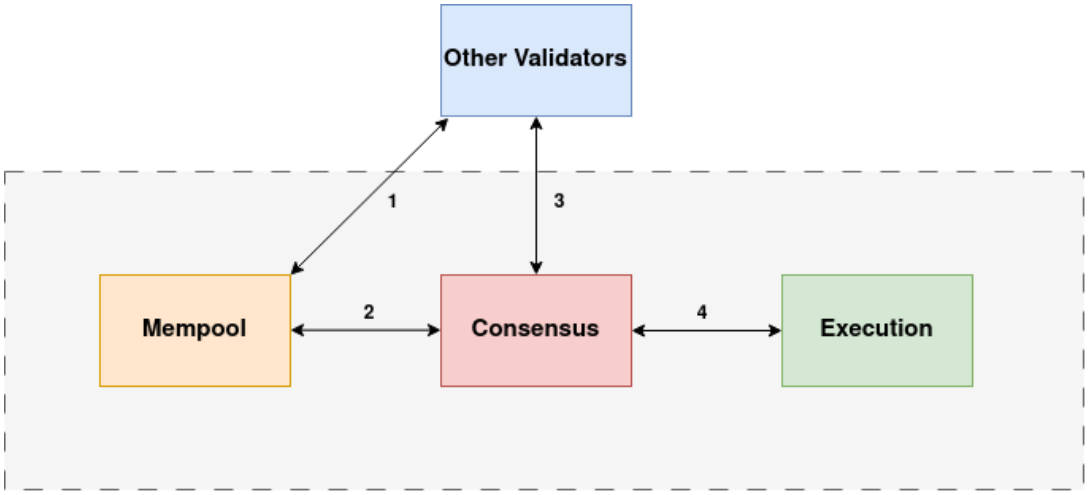


Figure 4.1: *Component Interaction*

(2) Consensus ↔ Mempool: When a validator is a proposer, its consensus component pulls a block of transactions from its mempool and forms a proposed block of transactions.

(3) Consensus ↔ Other Validators: If a validator is a proposer, its consensus component disseminates the proposed block of transactions to other validators.

(4) Consensus ↔ Execution: After a block is added to the "dirty ledger" by the consensus protocol, it is passed to the execution component to be deterministically executed and commited.

Utilizing the Aptos codebase as a foundation for our system, the main programming language we used was Rust [5]. Rust is a modern programming language which provides memory safety without sacrificing performance. The major changes were around the execution engine, adapting it to handle pre-computed dependency hints.

4.2 Execution Engine

Thread flow

The Rust library we employed in the project to enable parallel utilization of available cores was Rayon [4]. Each thread belonging to a Rayon thread pool, executes and commits a block by looping in the function `work_task_with_scope()`, which we present in 4.2. The data structure `scheduler` is shared among all running threads. The basic

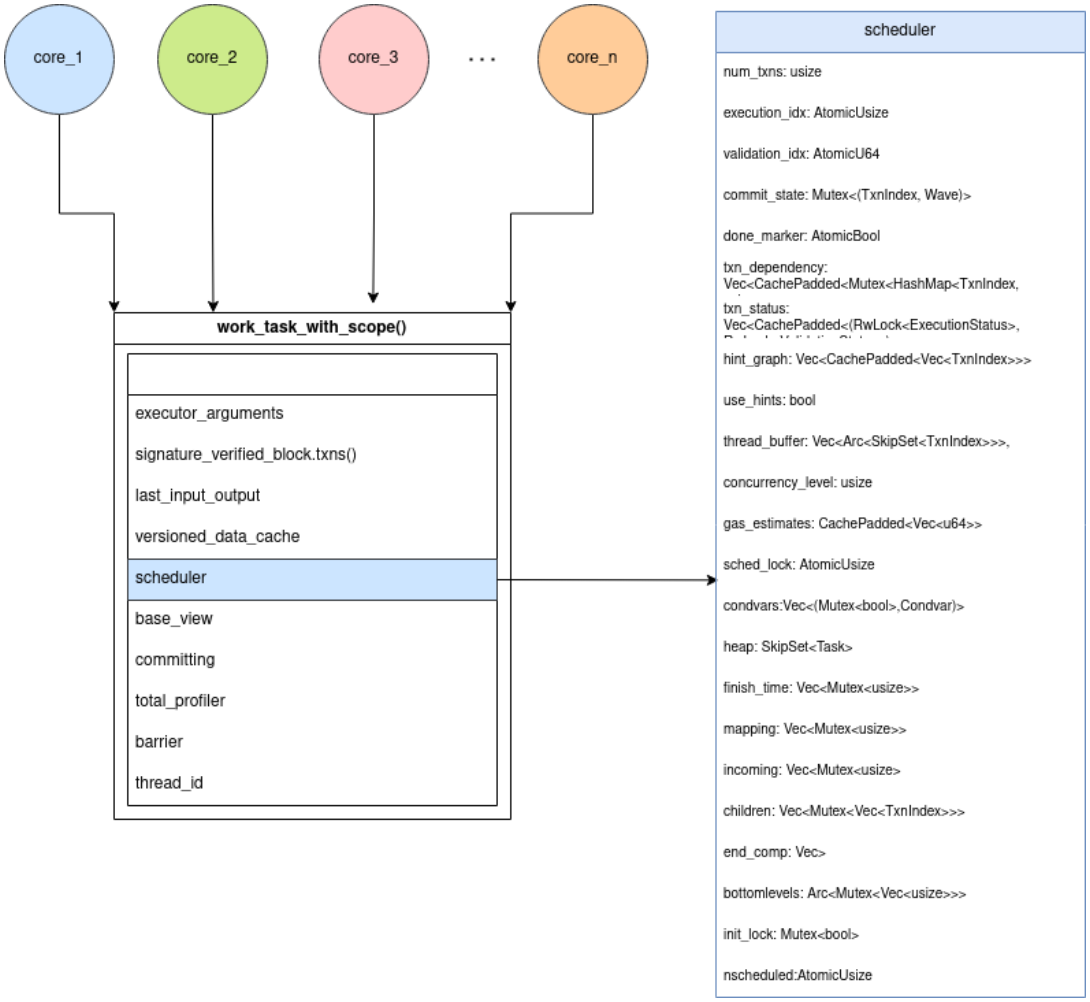


Figure 4.2: *Scheduler data structure*

components of this data structure are shown below:

The core logic of the function `work_task_with_scope()` in Rust :

```
fn work_task_with_scope(...) {
 {
    loop {
        if committing {
            loop {
                if scheduler.try_commit().is_none() {
                    break;
                }
            }
        }
      scheduler_task = match scheduler_task {
        SchedulerTask::ValidationTask(...) => {
```

```
13              let ret = self.validate(...);
14          },
15          SchedulerTask::SigTask(index) => {
16              ...
17          },
18          SchedulerTask::NoTask => {
19              let ret = scheduler.next_task(...);
20          },
21          SchedulerTask::Done => {
22              ...
23          },
24          SchedulerTask::ExecutionTask(...) => {
25              let ret = self.execute(...);
26          },
27          SchedulerTask::PrologueTask => {
28              ...
29          }
30          _ => {break;}
31        }
32      }
33  }
```

Each thread first enters `work_task_with_scope()` with `scheduler_task` initialized as `SchedulerTask::NoTask`, goes into `scheduler.next_task()` and follows the logic of the *scheduler* module. After a first check of whether the `self.done()` flag is set, the first thread that reaches a predefined threshold in its respective `thread_buffer` which holds the transactions already scheduled to it, attempts to schedule another batch of transactions in `sched_next_chunk()`. The first time, sched_setup() is called, so that all necessary information such as the *bottomlevel* of each transaction is available for scheduling.

We outline the calculation of the bottom level of each transaction in 3.1. The tasks without incoming dependency edges are marked as ready to execute in lines 24-28.

```
1  fn sched_setup(&self) {
2      let mut init = self.init_lock.lock();
3      let mut incoming_lock = self.incoming.lock();
4      let mut children_lock = self.children.lock();
5      let mut mapping_lock = self.mapping.lock();
6      if *init == false {
```

```
7           return ();
8       }
9       *init = false;
10      mapping_lock[0] = 0;
11      let mut bottomlevels: Vec<TxnIndex> = vec![0; self.num_txns];
12      for i in (0..self.num_txns).rev() {
13          for node in &*self.hint_graph[i] {
14              incoming_lock[i] += 1;
15              children_lock[*node].push(i);
16              if bottomlevels[*node] < bottomlevels[i] + 1 {
17                  bottomlevels[*node] = bottomlevels[i] + 1;
18              }
19          }
20          if self.hint_graph[i].is_empty() {
21              self.heap.insert(Task {
22                  bottomlevel: bottomlevels[i],
23                  index: i,
24              });
25          }
26      }
27      *self.bottomlevels.lock() = bottomlevels;
28  }
```

If there is no further scheduling to be done, a thread first re-checks for concurrency reasons whether it needs to return `Scheduler::NoTask`. In case there is still work to be done, priority is first given to validation tasks, to limit cascading aborts. If there is no validation task ready, the thread finally attempts to execute a new incarnation of a transaction in line 21.

```
1   pub fn next_task(&self, ...) -> SchedulerTask {
2    loop {
3       if !*local_flag && *finished_val_flag && self.done() {
4           return SchedulerTask::Done;
5       }
6       if *local_flag && self.nscheduled.load() < self.num_txns {
7           *local_flag = false;
8           if let Ok(_) = self.sched.try_lock.()
9               self.sched_setup();
10              let x = self.sched_next_chunk().unwrap();
11              return x;
```

Figure 4.3: *Thread flow of scheduler*

```
        }
        else {
            if !*finished_val_flag {
                if let Some((val,guard)) = self.try_val() {
                    let ret = SchedulerTask::ValidationTask(val, guard);
                    return ret;
                }
            }
            let ex = self.try_exec(thread_id, ...);
            return ex;
        }
    }
 }
...
}
```

Finally, it should be noted than in the rare case that the execution of a transaction incurs a conflict due to a false dependency graph (either stale or byzantine), the validation process will detect, abort and reschedule the transaction accordingly when the dependency is resolved.

Thread Affinity

By (a) scheduling transactions in a way where the majority of inter-dependent transactions are scheduled on the same thread (Algorithm 2) and (b) pinning threads to CPU cores, we aim to reduce the number of cache coherence conflicts. Thread pinning, is a technique in which a thread is bound to a specific CPU core or set of CPU cores in a multi-core processor system. This means that the thread is restricted to running exclusively on the pinned CPU core(s), rather than being scheduled across multiple cores by the operating system's scheduler. We achieve this by using the *affinity* rust crate ([1]) which uses `sched_setaffinity` from the *libc* crate, which in turn calls the `set_thread_affinity` system call.

```
pub static RAYON_EXEC_POOL: Lazy<rayon::ThreadPool> = Lazy::new(|| {
    rayon::ThreadPoolBuilder::new()
        .num_threads(num_cpus::get())
        .thread_name(|index| format!("par_exec_{}", index))
        .spawn_handler(|thread| {
```

```
            std::thread::spawn(|| {
                affinity::set_thread_affinity(&[thread.index()]);
                thread.run();
            });
            Ok(())
        })
        .build()
        .unwrap()
});
```

4.3 Good Block Production

We show he logic of producing ”Good Blocks” as described in 3.3 in the code block below. In lines 13-16, the earliest starting time of each transaction is calculated depending on previous transactions who read common resources. If a transaction exceeds the chosen value of `self.gas_per_core * 2`, it is skipped from the current block and is cached for a block proposal (lines 20-23). The total gas capacity of the block is checked in lines 24-28 and afterwards the dependency chain length is calculated to make sure that it doesn't exceed `self.max_bytes`. Finally, corresponding values are updated to prepare for a new transaction in the next iteration of the loop (lines 37-42) as well as the last touched times for each resource the transaction accessed.

```
    fn add_all (...) -> Vec<SignedTransaction> {
        ...
        while let Some((speculation, status, tx)) = previous.pop_front()
        {
            let txn_len = tx.raw_txn_bytes_len() as u64;
            if self.total_bytes + txn_len > self.max_bytes {
                self.full = true;
                cache.push_back((speculation, status, tx));
                break;
            }
            let mut arrival_time = 0;
            for read in read_set {
                if self.last_touched.contains_key(&read) {
                    arrival_time = max(arrival_time, *self.last_touched.get(&
    read).unwrap())
```

```
                }
            }
            // Check if there is room for the new block.
            let finish_time = arrival_time + gas_used;
            if finish_time > (self.gas_per_core * 2) as u64 {
                cache.push_back((speculation, status, tx));
                continue;
            }
            if self.total_estimated_gas + gas_used > self.gas_per_core * self
.cores {
                self.full = true;
                cache.push_back((speculation, status, tx));
                break;
            }
            let mut dependencies = HashSet::new();
            // Add Read-Write conflicts to dependencies
            if self.total_bytes + txn_len + (dependencies.len() as u64) * (
size_of::<TransactionIdx>() as u64) + (size_of::<u64>() as u64) > self.
max_bytes {
                self.full = true;
                cache.push_back((speculation, status, tx));
                break;
            }

            self.total_bytes += txn_len + dependencies.len() as u64 * size_of
::<TransactionIdx>() as u64 + size_of::<u64>() as u64;
            self.total_estimated_gas += gas_used;

            let current_idx = self.block.len() as TransactionIdx;
            self.estimated_gas.push(gas_used);
            self.dependency_graph.push(dependencies);

            // Update last touched time for used resources.
             ...
            }
        }
    }
```

5 Evaluation

5.1 Overview

We evaluated our system using a two-step process: **(1)** First we evaluated the execution engine separately, without taking account the creation of full blocks, nor the time spent pre-executing, and subsequently **(2)** we deployed our own testnet on AWS [2], simulating a full system operation. However, the full system analysis goes beyond the scope of this thesis and is part of future work under Ray Neiheiser. Therefore we only provide the deployment steps through which the necessary infrastructure to run our testnet is built.

Many studies evaluate blockchain algorithms using simple peer-to-peer transaction workloads [25], or workloads with little contention and complexity. While better evaluation frameworks have been proposed, such as Diablo [26], they still fall short of representing realistic blockchain workloads. To address this, we conducted a thorough analysis of the user activity on Ethereum and Solana and identified four realistic blockchain execution scenarios: NFT Minting, DEX Trading, Peer-to-Peer (P2P) Transactions, and Mixed Contracts. These scenarios cover a wide range of execution characteristics, from heavy contention and complex contract interactions to simple P2P transactions. This allows for a more comprehensive evaluation of blockchain algorithms and their ability to handle the demands of realistic workloads.

5.2 Workloads

In the following, we will describe the four workloads:

P2PTX: First, we created a **Peer-to-Peer Transaction** workload. However, instead of assuming a uniform distribution, we simulated the account distribution of peer-to-peer transactions on the Ethereum Mainnet over 2022.

NFT: Next, the **NFT Minting** workload is derived from Ethereum's minting behavior in 2022, collected and calculated in a similar fashion as the Peer-to-Peer Workload.

DEXAVG/DEXBURSTY: We created two **DEX Trading** Workloads for which we gathered data on the daily distribution of different trading pairs on Uniswap over the course of 2022. From the data, we then obtained a daily average for an *Average DEX Workload* and the thirty most contended days for a **Bursty DEX Workload**.

SOLANA: Finally, for the **Mixed Contracts** workload, we extracted the write sets of Solana transactions and their corresponding gas expenditure. This workload is the most complex among the four, as it involves varying the length of the write-set, the access distribution of resources, and the transaction runtime. We obtained this data by querying a sample of 1000 blocks per day in 2022 and calculated the average similar to before.

5.3 Dependency Aware Execution

The single server evaluation of the execution engine was done on a Mac Pro (2023) with a 24 core CPU (16 performance cores and 8 efficiency cores) and 64 GB of RAM.

Figure 5.1 shows the per-second throughput for all workloads for the baseline (BlockSTM) and our dependency-aware execution model without signature validation (full line) and with signature validation (dotted line). We observe that the performance difference between the two approaches increases significantly when including signature validation concurrently to the block execution. This is the case because BlockSTM uses all the cores throughout the experiment and has little room to verify the block signatures. Meanwhile, our system schedules the transactions taking the dependencies into account and can use the idle CPU time it gains from this to verify the signatures.

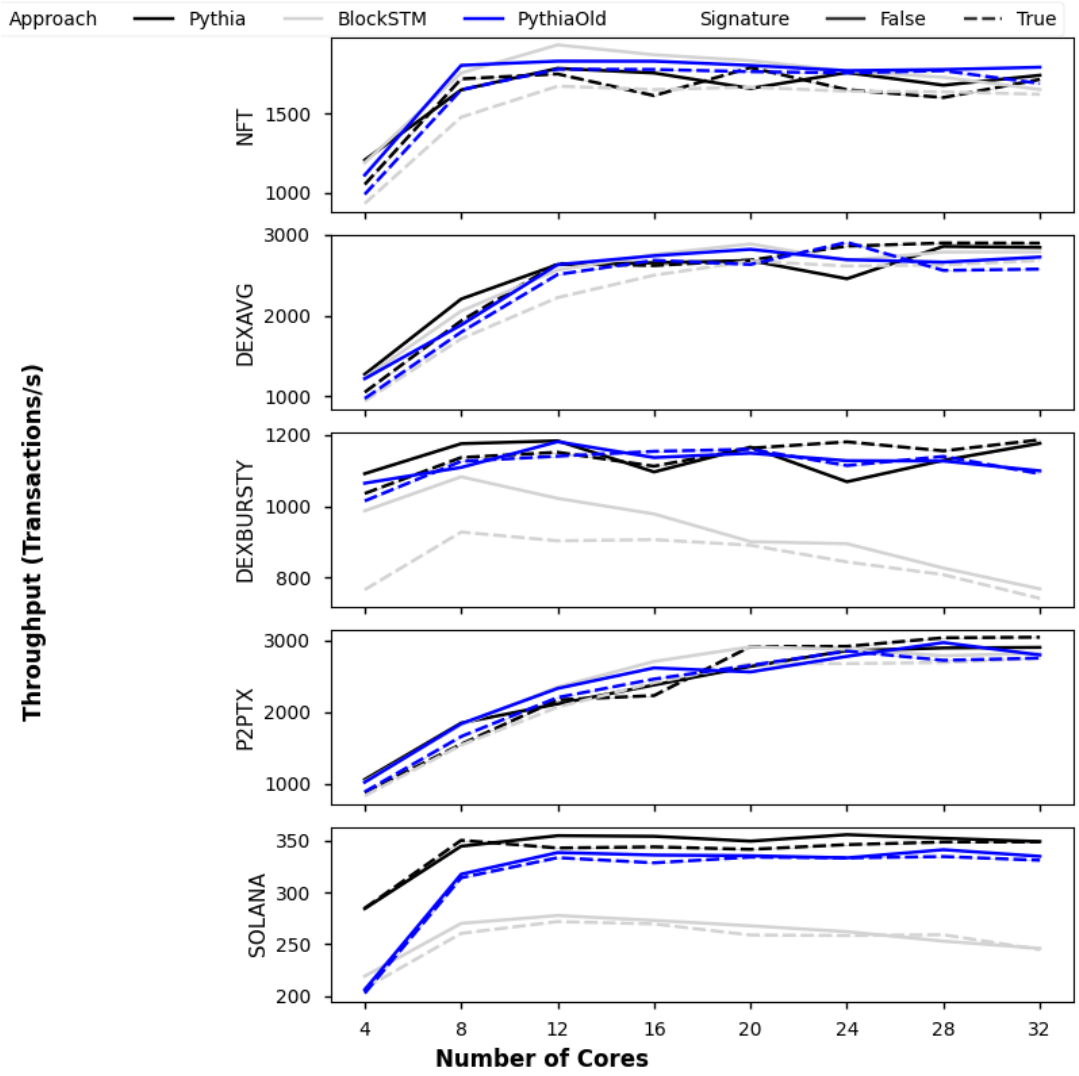


Figure 5.1: *Throughput per Second - Execution Engine*

5.4 Cloud infrastructure deployment

For the full system evaluation setup we modified and debugged existing orchestration scripts provided by Aptos Labs.

Terraform: The deployment was done through Terraform [46], an infrastructure automation tool which operates with declarative configuration files. These configuration files describe the desired state of the infrastructure, and according to them, Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs).

AWS: We deployed a testbed on *Amazon Web Services* after previously attempting to use Google Cloud Platform (GCP), but failing due to quota constraints. Through AWS we provision a cluster of virtual machines, as well as any storage or network requirement.

Kubernetes: To utilize this cluster of virtual machines, we used *Kubernetes* (K8s). K8s provides tools for deploying and scaling containerized applications automatically. It ensures that the desired number of container instances are running, handles failures, and replaces containers that become unhealthy. K8s primarily works with containers, which are lightweight, isolated, and portable environments that package an application and its dependencies. In Kubernetes, the smallest deployable unit that represents a single instance of a running application in a cluster is called a *Pod*.

EKS: To run our containerized application on AWS, we leveraged *Amazon's Elastic Kubernetes Service* (EKS). EKS takes care of the Kubernetes control plane, including master nodes, etcd clusters, and the API server. AWS manages the control plane's availability, scalability, and security.

Docker image: Finally, to create our containerized application, we used the open source platform Docker. The resulting output is a *docker image* which is lightweight, standalone, and executable package that contains all the necessary components to run our system, including the code, runtime, system tools, libraries, and settings.

Starting with the development of our system using Rust and subsequently publishing the Docker image to a repository like Docker Hub, we proceed by using our configuration files for the deployment of an EKS cluster through Terraform. This cluster operates Kubernetes on AWS cloud infrastructure, effectively managing all essential prerequisites, including storage provisioning through Persistent Volume Claims and network

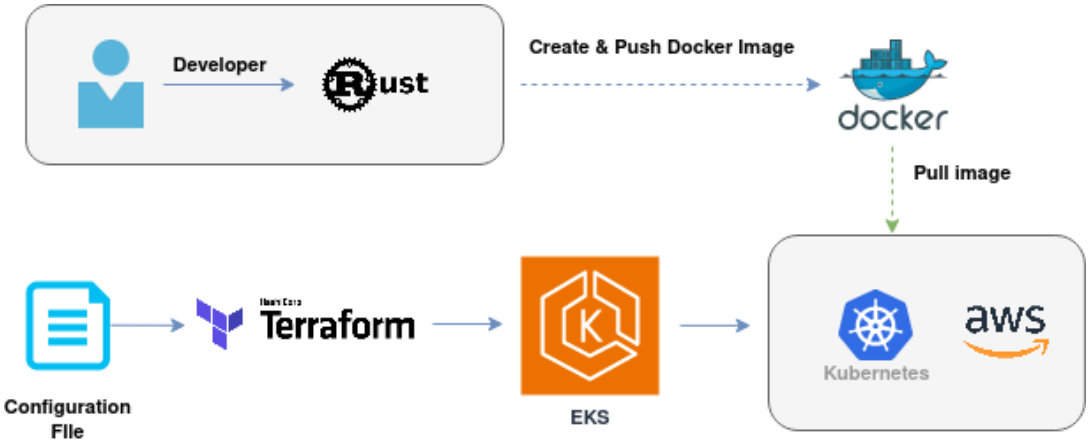


Figure 5.2: *Deployment steps*

administration. Finally, a "Validator" Pod on each virtual machine runs the Docker image containing our built system, and the cluster is fully functional and ready to accept transaction submissions by clients. The full work flow can be seen in 5.2.

6 Conclusion

In this work, we proposed a system design allowing deterministic parallel smart contract execution, leveraging the multi core capabilities of modern computing units. Furthermore, we presented a scheduler for transactions containing smart contract code, which uses pre-computed hints to speed up execution by up to 1.5x compared to BlockSTM [25] in real-world workloads.

6.1 Concluding Remarks

Summary

Work	Context Driven Execution	Decoupled Consensus & Execution	Intelligent Block Assembly	Susceptibility Performance Attacks
Block-STM [25]	No	Yes	No	High
Thomas Dickerson, et al. [17]	Yes	No	No	High
Polygon [39]	Yes	No	No	Low
FuelVM [22]	Yes	No	No	Low
FSC [34]	Yes	No	No	Low
Solana [44]	Yes	No	No	High
Eve [28]	No	No	Limited	High
Aria [35]	No	No	No	High
This work	Yes	Yes	Yes	Low

Table 6.1: ***Overview of the different approaches***

Table 6.1 compares different approaches based on four main criteria: (1) use of contextual data to avoid conflicts during execution, (2) decoupling of consensus from execu-

tion, (3) use of contextual data to construct easily parallelizable blocks, and (4) susceptibility to performance attacks.

To the best of our knowledge, all current approaches in the literature that offer parallel smart-contract execution are either significantly slowed down by highly contended workloads and sequential transaction chains and also subject to performance attacks. They also execute transactions on the critical path of consensus, reducing the potential speedup significantly.

In our work, we tackle these shortcomings with the help of asynchronous pre-execution which allows us to obtain contextual transaction information without relying on developer input. We use this information to build “Good Block”and speed up execution through metadata.

6.2 Future Work

As most work presented in this thesis was conducted in collaboration with a team in ISTA, there is still future work such as the final details of implementation and evaluation of the system. A key problem that needs to be addressed is that the transactions that each node pre-executes can overlap, resulting in wasted execution time. One way to solve this can be sharding the transaction pool of each node so that each each node pre-executes only the transactions it will propose in the future. However this endangers the liveness of the system and potentially the latency of a transaction commit. Additionaly, as the transaction scheduler of the execution engine is optimized under perfect prediction assumption, its performance needs to also be tested under varying percentages of wrong hints. Finally, the incentives of a validator to operate following the protocol and propose ”Good Blocks” need to be formalized.

Bibliography

[1] affinity documentation. https://docs.rs/affinity/latest/affinity/.

[2] Amazon web services. https://aws.amazon.com/.

[3] Easy parallelizability, ethereum eip.

[4] Rayon documentation. https://docs.rs/rayon/latest/rayon/.

[5] Rust programming language. https://www.rust-lang.org/.

[6] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR*, abs/1803.05069, 2018.

[7] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1337–1347, 2019.

[8] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. Optsmart: A space efficient optimistic concurrent execution of smart contracts. *CoRR*, abs/2102.04875, 2021.

[10] Zeta Avarikioti, Krzysztof Pietrzak, Iosif Salem, Stefan Schmid, Samarth Tiwari, and Michelle Yeo. Hide & seek: Privacy-preserving rebalancing on payment channel networks. *IACR Cryptol. ePrint Arch.*, 2021:1401, 2021.

[11] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus, 2019.

[12] Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2013.

[13] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative transaction execution for ethereum. pages 570–587, 10 2021.

[14] Shir Cohen, Guy Goren, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Proof of availability retrieval in a modular blockchain architecture. Cryptology ePrint Archive, Paper 2022/455, 2022. `https://eprint.iacr.org/2022/455`.

[15] George Danezis, Eleftherios Kokoris Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus, 2022.

[16] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In Shlomi Dolev, editor, *Distributed Computing*, pages 194–208, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[17] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts, 2017.

[18] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. *CoRR*, abs/1702.04467, 2017.

[19] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 4 1988.

[20] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, jul 2015.

[21] Aptos Foundation. Aptos whitepaper. `https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf`, 2023. Accessed on 12.04.2023.

[22] Fuel Network. `https://www.fuel.network/`.

[23] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. ACM, may 2022.

[24] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback, 2021.

[25] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing, 2022.

[26] Vincent Gramoli, Rachid Guerraoui, Andrei Lebedev, Chris Natoli, and Gauthier Voron. Diablo: A benchmark suite for blockchains. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 540–556, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, page 289–300, New York, NY, USA, 1993. Association for Computing Machinery.

[28] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-Verify replication for Multi-Core servers. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 237–250, Hollywood, CA, October 2012. USENIX Association.

[29] Aggelos Kiayias, Alexander Russell, Bernardo Machado David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, 2017.

[30] Aggelos Kiayias and Dionysis Zindros. Proof-of-work sidechains. In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 21–34, Cham, 2020. Springer International Publishing.

[31] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, jun 1981.

[32] Mysten Labs. `https://sui.io/`.

[33] Haoran Lin, Yajin Zhou, and Lei Wu. Operation-level concurrent transaction execution for blockchains, 2022.

[34] Ye Lu, Caihua Liu, Meng Zhao, Xiaodong Duo, Pengfei Xu, Zhiyuan Zhou, and Xia Feng. Fsc: A fast smart contract transaction execution approach via read-write static analysis. 2023.

[35] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic oltp database. *Proc. VLDB Endow.*, 13(12):2047–2060, jul 2020.

[36] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[37] Ray Neiheiser, Gustavo Inácio, Luciana Rech, Carlos Montez, Miguel Matos, and Luís Rodrigues. Practical limitations of ethereum's layer-2. *IEEE Access*, 11:8651–8662, 2023.

[38] Yusuf M. Özkaya, Anne Benoit, Bora Uçar, Julien Herrmann, and Ümit V. Çatalyürek. A scalable clustering-based task scheduler for homogeneous processors using DAG partitioning. In *IPDPS 2019 - 33rd IEEE International Parallel & Distributed Processing Symposium*, pages 155–165, Rio de Janeiro, Brazil, May 2019. IEEE.

[39] Polygon. `https://polygon.technology/`.

[40] Suporn Pongnumkul, Chaiyaphum Siripanpornchana, and Suttipong Thajchayapong. Performance analysis of private blockchain platforms in varying workloads. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, 2017.

[41] Dai Qin, Angela Demke Brown, and Ashvin Goel. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 180–194, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 543–557, New York, NY, USA, 2020. Association for Computing Machinery.

[43] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[44] Solana Labs. `https://solana.com/`.

[45] Christos Stefo, Zhuolun Xiang, and Lefteris Kokoris-Kogias. Executing and proving over dirty ledgers. Cryptology ePrint Archive, Paper 2022/1554, 2022. `https://eprint.iacr.org/2022/1554`.

[46] HashiCorp Terraform. `https://www.terraform.io/`.

[47] Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. Blockchain scaling using rollups: A comprehensive survey. *IEEE Access*, 10:93039–93054, 2022.

[48] Gang Wang, Zhijie Shi, Mark Nixon, and Song Han. Sok: Sharding on blockchain. pages 41–61, 10 2019.

[49] Rasanga Weerawarna, Shah Jahan Miah, and Xi Shao. Emerging advances of blockchain technology in finance: a content analysis. *Pers Ubiquit Comput*, 27:1495–1508, 2023.

[50] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[51] Darko Čapko, Srđan Vukmirović, and Nemanja Nedić. State of the art of zero-knowledge proofs in blockchain. In *2022 30th Telecommunications Forum (TELFOR)*, pages 1–4, 2022.