UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Distributed Coordination and Computation Synthesis

Permalink
https://escholarship.org/uc/item/3pf8c8q3

Author
Houshmand, Farzin

Publication Date
2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Distributed Coordination and Computation Synthesis

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Farzin Houshmand

March 2023

Dissertation Committee:

Dr. Mohsen Lesani, Chairperson
Dr. Rajiv Gupta
Dr. Evangelos Christidis
Dr. Vassilis Tsotras

Copyright by
Farzin Houshmand
2023

The Dissertation of Farzin Houshmand is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am indebted to many people who have helped and encouraged me in this undertaking. Thanks to my wife and family for being supportive every step of the way. I would like to thank my family, my parents, Shadi and Sadegh, and my sister, Azin, for always supporting me and giving me the courage whenever I needed it the most. I owe everything to them. Also, I would like to thank my wife and best friend, Aida, who loved, supported, and helped me throughout my studies. Aida, I couldn't have done it without you! Last but not least, thanks to my advisor Mohsen Lesani. I owe a massive debt of gratitude to him for his guidance through the years, and I do not think I could have made it to this point without such an exceptional mentor.

Chapter 2 was originally published as "Hamsaz: Replication Coordination Analysis and Synthesis" in the POPL'19 and the reviewers found it as "well-poised to make a substantial impact in the design and implementation of distributed systems going forward". In regard to this paper, I would like to thank my advisor for his valuable assistance and ideas and the anonymous POPL reviewers for insightful feedback. This work was partially supported by the National Science Foundation grant, CRII: SHF: Certified Byzantine Fault-tolerant Systems (1657204).

Chapter 3 was originally published as "Hamband: RDMA Replicated Data Types" in PLDI'22. This work was partially supported by the National Science Foundation grant 1942711. I would like to thank my co-first author, Javad for his contributions.

Chapter 4 was originally published as "GraFS: Declarative Graph Analytics" in ICFP'21 and is supported by the National Science Foundation grants 1942711, 1718997, and

1910878. I'd like to express my gratitude to my co-authors, Keval Vora, and my advisor. Without your help, I wouldn't be able to do this project.

Finally, chapter 5 was an ongoing collaboration with Google Brain at the time of my defense. The majority of the work was done during my student research internship at Google. I would like to especially thank Ras Bodik for the tremendous amount of help and support. His valuable contributions made the paper stronger. Also, I'd like to thank Charith Mendis, Amit Sabne, Karthik Krishnamurthy, Mangpo Phothilimthana, and Praveen Narayana for their work.

To *shadi* and *sadegh*.

ABSTRACT OF THE DISSERTATION

Distributed Coordination and Computation Synthesis

by

Farzin Houshmand

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2023
Dr. Mohsen Lesani, Chairperson

Ensuring that our increasingly complicated distributed systems are simultaneously reliable and efficient is challenging. Rigorous formal analyses can help make the design and implementation of complex systems both more reliable and efficient and safeguard businesses from unwanted incidents. As a result, program synthesis has always been an area of interest for computer scientists. Traditionally, program synthesis has only been applied to limited domains and has not been studied at the scale of distributed systems. This thesis investigates the application of program synthesis in the domain of distributed systems, a domain that can benefit from program synthesis because of its complexities and nuances. We show that the synthesized systems generally provide better performance and significantly reduce the programmer's efforts to write code. In particular, we apply program synthesis to replication coordination, graph analytics, and tensor computation:

In our first work, Hamsaz, we present a novel sufficient condition for the correctness of the replicated data types called well-coordination, which requires total order between conflicting and causal order between dependent operations. Given the sequential specification

and integrity properties of an object, Hamsaz uses solvers for the theory of sets and relations to automatically find the conflicting and dependent pairs of operations. Furthermore, it minimizes the required coordination between replicas and automatically synthesizes correct-by-construction message-passing protocols that simultaneously guarantee integrity and convergence. Our results show that well-coordinated replicated data types are significantly more responsive than the strongly consistent baseline.

Compared to traditional data centers built with message-passing network adaptors, modern RDMA networks significantly reduce the response time. In our second work, Hamband presents coordination synthesis for RDMA network model. Concretely, it presents RDMA well-coordinated replicated data types, the first hybrid replicated data types for the RDMA network model as well as operational semantics for these data types that capture their required coordination. The semantics divides methods of a given object into three categories, reducible, irreducible conflict-free, and conflicting. Hamband is backed by formal proof of correctness that the replicated objects preserve the convergence and integrity properties. The empirical evaluation shows that Hamband outperforms the throughput of existing message-based and strongly consistent implementations by more than 17x and 2.7x respectively.

Graph analytics elicits insights from large graphs to inform critical decisions for business, safety, and security. However, implementation and especially optimization of graph analytics are error-prone and time-consuming tasks. In our next work, GraFS focuses on the synthesis of graph analytics. GraFS is a high-level declarative specification language for graph analytics and a synthesizer that automatically generates efficient code for various high-performance graph processing frameworks. It features novel semantics-preserving fusion

transformations that optimize the specifications and reduce them to three primitives of graph analytics, namely, reduction over paths, mapping over vertices, and reduction over vertices. Reductions over paths are commonly calculated based on push or pull models that iteratively apply kernel functions at the vertices. This project presents parametric conditions for the correctness and termination of the iterative models and uses these conditions as specifications to synthesize the kernel functions. Experimental results show that the synthesized code matches or outperforms handwritten code, and fusion accelerates execution.

Finally, in collaboration with Google Brain, we took the first step towards synthesizing tensor computation rewrites. Rewriting the computational graph of a tensor program is an important optimization employed by machine learning frameworks and compilers. In this work, we present TensorRight, a verified tensor graph rewrite system consisting of a formal tensor language and its denotational semantics for proving tensor optimization rewrite rules. TensorRight uses symbolic execution based on the semantics of tensor operations and novel dimension definitions to generate verification conditions sufficient to prove the equivalence of rewrites on input tensors with unbounded rank and dimension sizes. We show that TensorRight can represent and prove a comprehensive set of the XLA rewrite rules already existing in its rewrite engine.

Contents

List of Figures xii

List of Tables xiv

1 Introduction 1
- 1.1 Overview of Contributions 2
  - 1.1.1 Replication Coordination Synthesis 2
  - 1.1.2 Graph Analytics Computation Synthesis 4
  - 1.1.3 Tensor Computation Optimization Synthesis 5

2 Hamsaz: Replication Coordination Analysis and Synthesis 8
- 2.1 Introduction 8
- 2.2 Overview 11
- 2.3 Well-Coordination 23
- 2.4 Static Analysis 35
- 2.5 Use-cases 40
- 2.6 Protocols 43
  - 2.6.1 Non-blocking Synchronization Protocol 44
  - 2.6.2 Blocking Synchronization Protocol 52
  - 2.6.3 Dependency-Tacking Protocol 56
- 2.7 Implementation 57
- 2.8 Evaluation 58
- 2.9 Related Works 65

3 RDMA-Enabled Well-Coordination 70
- 3.1 Introduction 70
- 3.2 Overview 74
- 3.3 Replicated Data Types 81
  - 3.3.1 Object Data Types 82
  - 3.3.2 Semantics of Well-Coordinated Replicated Data Types 83
  - 3.3.3 RDMA Replicated Data Types 88
- 3.4 Implementation 96

3.5 Experimental Results 99
3.6 Related Works 105

4 Graph Analytics Fusion and Synthesis 108
4.1 Introduction 108
4.2 Overview 111
4.3 Declarative Graph Analytics 121
4.4 Iterative Models 125
4.5 Specification and Fusion 130
4.5.1 Core Specification Language 132
4.5.2 Fusion 136
4.5.3 Extensions 140
4.6 Mapping Specification to Iteration-Map-Reduce 143
4.6.1 The Correctness of Iterative Path-Based Reduction 144
4.6.2 Synthesis of Iterative Reduction 151
4.7 Experimental Results 153
4.8 Related Work 163

5 Verified Tensor Graph Rewrite 167
5.1 Introduction 167
5.2 Motivation 170
5.2.1 Example Rewrite Rule 171
5.2.2 Representation and Proof 172
5.3 Overview 175
5.3.1 TensorRight Rewrite Rules 176
5.3.2 TensorRight Language Constructs 176
5.3.3 Proving Correctness 177
5.4 TensorRight DSL & Semantics 178
5.4.1 Core Syntax 178
5.4.2 Denotational Semantics 180
5.5 Verification of Rewrite Rules 189
5.5.1 Symbolic Dimension Types 189
5.5.2 The Verification Approach 191
5.5.3 Verifying expressions with reduction operators 193
5.6 Implementation 194
5.6.1 Rosette based Verification 195
5.7 Evaluation 198
5.7.1 Expressiveness 199
5.7.2 TensorRight Deployment 204
5.7.3 Generalizing XLA Rewrite Rules 207
5.8 Related Work 208

6 Conclusion and Future Work 211

Bibliography 214

List of Figures

2.1 Courseware Use-case. 12
2.2 Incorrect Executions and Coordination Avoidance Conditions. Square and circle around method calls in (b) , (d) and (f) are just visual aids to see the movements. 16
2.3 (a) Non-blocking Synchronization Protocol. The symbols $\downarrow$ and $\uparrow$ show requests to and responses from the protocols. Events to the main protocol are shown above and events to the sub-protocols are shown below the horizontal time line. The symbols ① and ② represent events of the first and second TOB sub-protocols respectively. Blocks show the execution of method calls. (b) Blocking Synchronization Protocol. The symbols $\downarrow$ and $\uparrow$ show requests to and responses from the protocols. Diagonal arrows show message transmission. 20
2.4 Correctness of well-coordinated replicated executions 32
2.5 Static analysis to calculate the conflict and dependency relations. The object $\langle \Sigma, \mathcal{I}, \mathcal{M} \rangle$ is given. 36
2.6 Auction and Two Phase Set Use-cases. The conflict graph in (d) is obtained from (b) and (c). 41
2.7 Non-blocking Synchronization Protocol. C and V are call and return value respectively 48
2.8 Multi-Total-Order Broadcast Protocol. M and C are message and class types respectively. 50
2.9 Blocking Synchronization Protocol 53
2.10 Relational Integrity Constrains 59
2.11 Analysis time 61
2.12 (a) Response time for Courseware with the Non-Blocking and SC protocols. (b) Response time for Courseware with the Blocking protocol. (c) Response time for BankAccount. (d) The effect of workload on response time for Courseware. 63

3.1 Bank Account. (a) The user specification, (b) The conflict graph, and (c) The dependency graph 75
3.2 RDMA Replicated Bank Account 77
3.3 Basic Syntax 82
3.4 WRDT State 84

3.5 WRDTs Semantics . . . 86
3.6 WRDT RDMA State . . . 91
3.7 RDMA WRDTs Semantics . . . 93
3.8 More RDMA WRDTs Semantics . . . 94
3.9 The effect of synchronization groups . . . 102
3.10 Project management use-case . . . 102
3.11 Effect of failure on the Counter and ORSet use-cases. . . . 103
3.12 The effect of failure on the courseware use-case. . . . 104

4.1 Workflow of GraFS (Graph Analytics Fusion and Synthesis) . . . 112
4.2 Fusion of the Radius Use-case. . . . 115
4.3 Pull Iterative Reduction. . . . 116
4.4 Unfused and fused implementations of Radius as iteration-map-reduce rounds. 119
4.5 The Correctness of the Pull Model. The path $p \cdot e$ denotes the extension of path p with edge e. . . . 121
4.6 A Subset of Use-cases in GraFS . . . 122
4.7 (a) Grammar for Kernel Functions (b) Example Kernel Functions. (min and + filter none values $\bot$.) . . . 127
4.8 Four Iterative Reduction Methods. $\mathsf{CPreds}^k(v)$: The predecessors of the vertex v that changed in the iteration k . . . 129
4.9 Core Specification Language . . . 131
4.10 Denotational Semantics of the Specification Language . . . 134
4.11 Fusion Rules . . . 137
4.12 More Fusion Rules . . . 138
4.13 Triple-let Form . . . 143
4.14 Correctness and Termination Conditions . . . 148
4.15 (a) Synthesis of the Propagation Function $\mathcal{P}$. (b) Context assertions Example. 151
4.16 Edge-work Ratio: Normalized # of edges processed by the fused over the unfused version. Missing bars correspond to programs not successfully running on input graphs. . . . 155
4.17 Edge-work ratio: # of edges processed by the fused over the unfused version. Missing bars correspond to programs not successfully running on inputs. . . . 156
4.18 (a): Synthesis time and the number of lines of code, (b): Scalability on Ligra. X-axis: # of cores. Y-axis: time is logarithmic scale. H: Handwritten. S: Synthesized. . . . 160

5.1 Motivating examples . . . 173
5.2 TensorRight overview and workflow . . . 175
5.3 Core Specification Language . . . 179
5.4 Denotational semantics of the core language. . . . 182
5.5 Illustration of the (a)expand, (b)transpose, and (c)dot operations. . . . 184
5.6 Reduction . . . 186
5.7 Normalization lemmas for reduction operator. . . . 194
5.8 Reorder transpose and slice. . . . 196
5.9 Fold input pad into convolution. . . . 208

List of Tables

4.1 Execution times (in seconds). H: Handwritten, S: Synthesized, R: the ratio H/S. 155
4.2 Execution Times (in seconds). H: Handwritten, S: Synthesized, R: the ratio H/S, ER: edge-work ratio. Missing cells are due to either missing handwritten use-cases (PR) or not successfully running on an input 161
4.3 Metrics for Comparing Handwritten and Synthesized Code. H: Handwritten, S: Synthesized. (PowerGraph does not require the user to write atomic operations.) 162

5.1 Number of supported rules per disjoint category. The numbers in parentheses indicate rules that have been implemented and verified (and we continue to implement more); some of the not-yet-implemented rules require defining new operations and/or new lemmas. 205
5.2 Number of unsupported rules per each reason. There may be overlaps between reasons. The first reason is not a fundamental limitation because new operations can be added. 205
5.3 Table shows the number of instances a TensorRightverified rule was instantiated for 114 XLA regression benchmarks compiled for multiple generations of TPUs. 206
5.4 Table shows the number of instances a TensorRightgenerated-C++ rule was instantiated for the `MLPerf` suite of benchmarks compiled for multiple generations of TPUs. 206

Chapter 1

Introduction

Program synthesis [334, 201, 441] is a critical research area in computer science that involves generating programs automatically from high-level specifications. It is one of the greatest, longest-standing pursuits of computer science [332]. Program synthesis is a powerful technique that can significantly reduce the time and effort required to develop software applications, making it an essential tool for computer scientists. At its core, program synthesis is the problem of automatically finding the correct computer program given the high-level specification of a problem which often involves searching within a possibly infinite space of candidate programs. The specification allows us to pick out particular programs of interest that we find during the search. This high-level specification can take many forms, such as logical statements[135], input/output examples [196], and partial programs [175]. Different techniques in program synthesis include inductive synthesis [384, 260], deductive synthesis [253, 333], syntax-guided synthesis [429, 21, 228], and counter-example guided synthesis [385, 192, 216]. Moreover, Works such as Sketch [435] and Rosette [449] introduce

solver-aided languages, where one specifies a synthesis problem using a domain-specific language, which is translated into an SMT problem.

There have been many prior works on applications of program synthesis. Traditionally, synthesis has been applied to different domains, such as databases, where it can be used to automatically generate SQL queries [468, 483], compiler and optimization [416, 379], synthesizing structured programs [198, 388, 196], and many more. However, regardless of the recent advancements, there has been little to no work on utilizing program synthesis for distributed systems. Distributed systems are known to be notoriously hard to program. In a distributed system, the execution of a program may consist of interleavings that can affect the correctness of the execution. The situation gets more complicated when network delays, clock drift, and faulty nodes are considered. On the bright side, the complexity of large systems is usually encapsulated in a small piece of code or function. This makes program synthesis for such complex systems effective.

1.1 Overview of Contributions

In this thesis, we outline promising applications of program synthesis in the distributed domain. More concretely, we apply program synthesis to replication coordination, graph analytics, and tensor optimization computations.

1.1.1 Replication Coordination Synthesis

Distribution and replication are an often-used mechanism to achieve fault tolerance and scalability. Embedded control systems replicate controllers to tolerate faults, online

services rely on geo-replicated data stores to manage the ever-growing amount of data and hand-held devices replicate data for offline use. There has been a known dilemma between strong and weak consistency of replicated objects. Strongly consistent replication guarantees the same total order of operations across all replicas and simplifies reasoning about the correctness of such applications. However, synchronization protocols that provide strong consistency need consensus between replicas and, hence, may not be responsive and even available during network failures or offline use. On the other hand, weak consistency notions can be provided with availability and responsiveness but without the same total order of operations across replicas. Unfortunately, the absence of the total order can lead to violation of integrity properties.

In chapter two, our goal is to automatically synthesize a correct-by-construction replicated object that guarantees integrity and convergence and avoids unnecessary coordination. Further, our approach supports notions weaker than causal consistency; it builds upon eventual, causal, and strong notions. We present a static analysis and protocol co-design. The core of our approach is a novel sufficient condition called well-coordination for integrity and convergence of replicated objects. We define notions of conflicting and dependent pairs of methods. Well-coordination requires synchronization between conflicting and causality between dependent operations. We statically analyze the given sequential object and its integrity property, and infer the pairs of conflicting methods (represented as the conflict graph) and dependent methods. We present two novel distributed protocols that provide the well-coordination requirements. The protocols are parametric for the analysis results. We present a non-blocking synchronization protocol based on a novel variant of the total-order-

broadcast protocol. The protocol parameters are decided by a reduction of the minimum synchronization problem to the maximal clique problem on the conflict graph. We also present a synchronization protocol that is blocking but allows some of the conflicting methods to execute without synchronization. The protocol parameters are decided by a reduction of the minimum synchronization problem to the vertex cover problem on the conflict graph.

The distributed system model that was considered for chapter two was the traditional message-passing network model. In chapter three we consider the RDMA network adaptors. RDMA offers two classes of communication primitives: two-sided and one-sided. One-sided communication has similar semantics to the traditional shared memory model. A node directly performs a write or read operation on the memory of another node. The access is performed without involving the CPU of the other node. This class tends to deliver lower response time since it bypasses the network and operating system stack and does not interrupt the CPU of the other node Chapter three presents a novel operational semantics for RDMA replicated data-types and leverages a single one-sided write operation that can be executed in parallel on the replicas to execute a certain category of method calls. This results in even faster coordination in RDMA systems. Further, it defines abstract operational semantics for WRDTs that captures well-coordination conditions. It proves that the concrete semantics of RDMA WRDTs refines the abstract semantics of WRDTs. Therefore, any execution of an RDMA WRDT is well-coordinated.

1.1.2 Graph Analytics Computation Synthesis

Today, graph analytics is used in a wide range of systems and applications, from fraud detection to recommendation engines. It is a critical tool for businesses and organiza-

tions looking to gain insights from complex, interconnected data. High-performance graph processing frameworks are designed to efficiently process and analyze large-scale graphs. These frameworks are becoming increasingly important in today's data-driven world, where large graphs are used to represent complex systems such as social networks, supply chains, and financial transactions. Graph processing frameworks provide a range of features, including distributed computing capabilities that enable parallel processing of large datasets across multiple machines, and optimized graph algorithms for faster processing. Unfortunately, instead of providing high-level abstraction, these frameworks offer low-level APIs for writing custom computation kernels to analyze large-scale graphs. Furthermore, manual optimizations of graph analytics can be time-consuming and error-prone. In particular, showing correctness and termination properties requires reasoning about the flow of values between vertices.

In chapter four, we propose the interface of the graph processing frameworks as the instruction set for graph analytics and introduce Grafs, a graph analytics language, and synthesizer. The Grafs language is a high-level declarative specification language that provides features for common graph processing idioms such as reduction over paths. We show that declarative language can easily and concisely capture common graph analysis problems. Given a specification, the Grafs synthesizer automatically synthesizes code for five graph processing frameworks.

1.1.3 Tensor Computation Optimization Synthesis

The tensor rewrite engine crucial for developing efficient machine-learning applications and is a key component of any machine-learning compiler. It is responsible for

transforming the computation graph of the input program into an optimized form. Developing the tensor rewrite engine by hand is a complex and error-prone process that must be carefully verified to ensure correctness. The large number of rewrite rules and the complexity of the operations they perform make it difficult for developers to manually verify the correctness of the generated code. Even small errors in the implementation of the rewrite rules can lead to incorrect results or even crashes during execution. Furthermore, as hardware and software platforms continue to evolve, the number and complexity of the rewrite rules required to optimize code will only increase. As a result, the development of the tensor rewrite engine must rely on formal methods and formal verification to ensure that the generated code is correct and performs optimally. The use of formal methods and formal verification can help identify errors early in the development process and ensure that the tensor rewrite engine produces efficient and reliable code for a wide range of hardware platforms.

In chapter five, we offer a framework to formally prove the correctness of tensor rewrite rules that are present in a production quality compiler for machine-learning applications such as XLA. We presentTensorRight, a verified tensor rewrite system that is able to both represent and prove the majority of the XLA tensor rewrite rules with unbounded tensor ranks and dimension sizes.TensorRight provides the first formalism of XLA tensor operators in a purely functional manner using denotational semantics. We introduce TensorRightDSL for implementing rewrite rules. The DSL consists of a core language specification for the XLA tensor operators and additional constructs to specify operations on dimensions, which aid in verifying the rewrites. We use the denotational semantics of operators to generate correctness verification conditions for the rewrite rules in a rank and dimension size polymorphic manner.

To achieve this, we embed the semantics of each operator as an executable specification in TensorRightDSL. We generate a verification condition of a rule by symbolically executing the LHS and RHS expressions and asserting their equality.

Chapter 2

Hamsaz: Replication Coordination Analysis and Synthesis

2.1 Introduction

Distributed system replication [65, 273, 375, 57] is an often-used mechanism to achieve fault-tolerance and scalability. Embedded control systems replicate controllers [327] to tolerate faults, online services rely on geo-replicated data stores [121, 123, 134, 301, 313, 314, 440] to manage the ever-growing amount of data and hand-held devices replicate data for off-line use. There has been a known dilemma [164, 182, 183, 9] between strong and weak consistency of replicated objects. Strongly consistent replication (via Viewstamp [362], Paxos [279] and Raft [364] protocols) guarantees the same total order of operations across all replicas. Therefore, if an operation is checked to preserve the integrity properties [36] at a replica, it will certainly preserve them in the other replicas as well. Further, replicas converge as a result

of the same sequence of operations. Therefore, the correctness of replicated execution simply reduces to the correctness of sequential execution. However, synchronisation protocols that provide strong consistency need consensus between replicas and, hence, may not be responsive and even available during network failures or offline use. Although optimized protocols can emerge [123, 237], the strong semantics prevents their availability for offline use. On the other hand, weak consistency notions can be provided with availability and responsiveness but without the same total order of operations across replicas. Many consistency weak notions dubbed eventual consistency [459, 424, 78, 89, 148, 114] simply broadcast the operations that may be arbitrarily reordered. Likewise, causal consistency [277, 18, 65] preserves only the causal order between operations. Unfortunately, the absence of the total order can lead to violation of integrity properties.

However, weak notions can be enough for certain operations to preserve the integrity properties. For example, consider a bank account object with the integrity property that its balance is non-negative. The deposit operation can be executed without any coordination as it cannot make the balance negative. However, a withdraw operation has to synchronize with other withdraw operations to avoid overdrafts. In addition, consistent execution of a withdraw operation may be dependent on the preceding deposit operations in the originating replica. Therefore, the withdraw operation needs both a total order with respect to other withdraw operations and a causal order with respect to preceding deposit operations. We observe that operations have distinct coordination requirements with respect to each other. It is unintuitive for end-users to specify the right consistency requirement for each operation. Requesting too much may degrade performance, and requesting too little may compromise

correctness. Thus, users either resort to the blanket strong consistency for all operations or ignore the problem and use a default notion of weak consistency.

Previous work recognized the problem, proposed hybrid models and took significant steps towards helping the user with consistency choices [446, 299, 300, 45, 431, 189] to avoid coordination [35, 409]. They proposed proof techniques to verify the sufficiency of user-specified consistency choices [189], or require user annotations to identify consistency choices and do not guarantee convergence [45]. Further, many approaches [189, 45, 299] are crucially dependent on causal consistency as the weakest possible notion while others have established the scalability limitations of causal consistency [38]. We will further survey related works in § 2.9. Given a sequential object with its integrity properties, our goal is to automatically synthesize a correct-by-construction replicated object that guarantees integrity and convergence and avoids unnecessary coordination: synchronization and tracking dependency between operations. Further, our approach supports notions weaker than causal consistency; it builds upon eventual, causal and strong notions.

We present a static analysis and protocol co-design. The core of our approach is a novel sufficient condition called well-coordination for integrity and convergence of replicated objects. We define notions of conflicting and dependent pairs of methods. Well-coordination requires synchronization between conflicting and causality between dependent operations. We statically analyze the given sequential object and its integrity property, and infer the pairs of conflicting methods (represented as the conflict graph) and dependent methods. We present two novel distributed protocols that provide the well-coordination requirements. The protocols are parametric for the analysis results. We present a non-blocking synchronization

protocol based on a novel variant of the total-order-broadcast protocol. The protocol parameters are decided by a reduction of the minimum synchronization problem to the maximal clique problem on the conflict graph. We also present a synchronization protocol that is blocking but allows some of the conflicting methods to execute without synchronization. The protocol parameters are decided by a reduction of the minimum synchronization problem to the vertex cover problem on the conflict graph.

We present a tool called Hamsaz that given an object definition, uses off-the-shelf SMT solvers to decide the pairs of conflicting and dependent methods. It then uses the analysis results to avoid coordination and instantiate the protocols to synthesize replicated objects. We successfully synthesized replicated objects for a suite of use-cases that we have adopted from the previous works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments show that compared to the strongly consistent baseline, the synthesized replicated objects are significantly more responsive.

In the rest of the paper, we first present an overview in § 5.3. We define the well-coordination condition and prove its sufficiency for correctness in § 2.3. We present the static analysis and apply it to use-cases in § 2.4 and § 2.5. We then, present the protocols in § 2.6. The implementation and evaluation are presented in § 4.7 and 2.8 before we conclude with related works.

2.2 Overview

In this section, we illustrate the coordination analysis and synthesis with examples.

Class Courseware

let Student := Set ⟨sid: SId⟩ in

let Course := Set ⟨cid: CId⟩ in

let Enrolment :=

  Set ⟨esid: SId, ecid: CId⟩ in

Σ := Student $\times$ Course $\times$ Enrolment

$\mathcal{I} := \lambda\ \langle ss, cs, es\rangle.$

  refIntegrity(es, esid, ss, sid) $\wedge$

  refIntegrity(es, ecid, cs, cid)

register(s) := $\lambda\ \langle ss, cs, es\rangle.$

  $\langle \mathbb{T}, \quad \langle ss\ \cup\ \{s\}, cs, es\rangle, \quad \bot\rangle$

addCourse(c) := $\lambda\ \langle ss, cs, es\rangle.$

  $\langle \mathbb{T}, \quad \langle ss, cs\ \cup\ \{c\}, es\rangle, \quad \bot\rangle$

enroll(s, c) := $\lambda\ \langle ss, cs, es\rangle.$

  $\langle \mathbb{T}, \quad \langle ss, cs, es\ \cup\ \{(s, c)\}\rangle, \quad \bot\rangle$

deleteCourse(c) := $\lambda\ \langle ss, cs, es\rangle.$

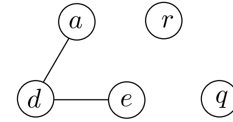
  $\langle \mathbb{T}, \quad \langle ss, cs \setminus \{c\}, es\rangle, \quad \bot\rangle$

query := $\lambda\ \sigma.\ \langle \mathbb{T}, \quad \sigma, \quad \sigma\rangle$

(a) User Specification

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	×	✓
e	✓	✓	✓	✓	✓
d	✓	×	✓	✓	✓
q	✓	✓	✓	✓	✓

(b) $\mathcal{S}$-commute



(e) Conflict Graph $G_{\bowtie}$

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	✓	✓
e	✓	✓	✓	×	✓
d	✓	✓	×	✓	✓
q	✓	✓	✓	✓	✓

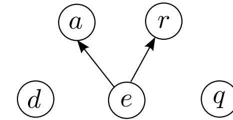
(c) $\mathcal{P}$-concur

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	✓	✓
e	×	×	✓	✓	✓
d	✓	✓	✓	✓	✓
q	✓	✓	✓	✓	✓

(f) Independent

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	×	✓
e	✓	✓	✓	×	✓
d	✓	×	×	✓	✓
q	✓	✓	✓	✓	✓

(d) Concur



(g) Dependency Graph

Figure 2.1: Courseware Use-case.

Object Replication. We define an object as a record $\langle \Sigma, \mathcal{I}, \mathcal{M} \rangle$ that includes the state type Σ, an *invariant* $\mathcal{I}$ that is a predicate on the state, and a set of methods $\mathcal{M}$. Fig. 2.1.(a) represents the courseware object that we have adopted from [189]. The state type Σ is the tuple of three relations for students ss, courses cs and enrolments es of students in courses. A relation is a set of records of fields. The student and course relations ss and cs are simply a set of records of one field, student identifiers sid and course identifiers cid respectively. The enrolment relation es is a set of records of two fields: the student identifier esid and the course identifier ecid, that are foreign keys from the other two relations. The desired invariant $\mathcal{I}$ for the courseware object is the referential integrity of the two foreign keys of the enrolment relation es. Every student identifier esid in the enrolment relation es must refer to an existing student identifier sid in the student relation ss. The condition for course identifiers is similar. We represent referential integrity properties using the $\mathsf{refIntegrity}$ predicate that is $\mathsf{refIntegrity}(R, f, R', f') := \forall r.\ \ r \in R \rightarrow \exists r'.\ r' \in R' \wedge f(r) = f'(r')$. For example, $\mathsf{refIntegrity}(es, \mathsf{esid}, ss, \mathsf{sid})$ states that for every record r in the relation es, there exists a record r' in the relation ss such that esid of r is equal to sid of r' that is $\mathsf{esid}(r) = \mathsf{sid}(r')$ where the field names esid and sid are used as functions on the corresponding records. Methods represent transactions on the object state. A method is a function m from the method parameter(s) and the pre-state σ to a record of $\langle \mathsf{guard}, \mathsf{update}, \mathsf{retv} \rangle$, where guard is the boolean precondition of the method, update is the post-state, and retv is the return value. The courseware object has five methods: $\mathsf{register}$ to register a student, $\mathsf{addCourse}$ to add a course, enroll to enroll a student in a course, $\mathsf{deleteCourse}$ to delete a course and query to obtain the current state of the object. The guard of a method captures the semantic

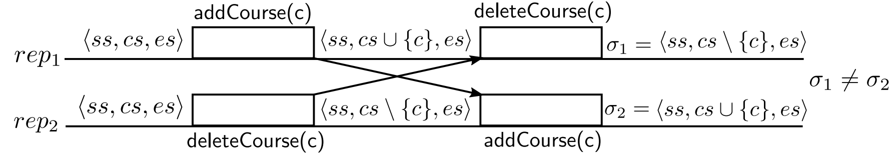
preconditions of the method and *not* the conditions that preserve the invariant. (We present the conditions that preserve the invariant in § 2.3.) For simplicity, the guards in this example are all $\mathbb{T}$. (A guard for the $\mathsf{deleteCourse}$ method could be that the input course should exist in the course relation to be deleted.) All but the query method return no value $\perp$. A method call c is the application of a method to its arguments i.e. a function from the pre-state to a record of $\langle \mathsf{guard}, \mathsf{update}, \mathsf{retv} \rangle$.

Given the definition of a sequential object, the goal is to automatically synthesize a replicated object. The state of the object is replicated across replicas. Clients can call methods at every replica and the calls are communicated between replicas. The replicated object is expected to satisfy both consistency and convergence. *Consistency* is the safety property that every method call is executed only when the guard of the method and the invariant are satisfied. *Convergence* is the safety property that when no call is in transit, all replicas converge to the same state. We want to perform coordination only when necessary to preserve these properties. We say that a method call c is *permissible* in a state σ, written as $\mathcal{P}(\sigma, c)$, if the guard of c is satisfied in σ and c results in a post-state σ' that satisfies the invariant $\mathcal{I}$ that is $\mathcal{I}(\sigma')$. The post-state of a method call is the pre-state of the next in a replica. The initial state is assumed to satisfy the invariant. Therefore, if every call is permissible in its pre-state, then every call is consistent. To execute a method call, we check that it is permissible in its originating replica. Thus, we say that each method call is *locally permissible*. Otherwise, the call is aborted. Still, if the call is simply broadcast, it is not necessarily permissible when it arrives at other replicas. Some calls need coordination. We

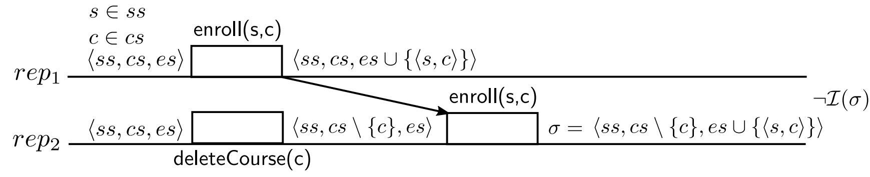
now present representative incorrect executions to showcase the conditions that necessitate coordination.

Well-coordination. Method calls such as adding a course and enrolling a student result in the same state if their order of execution is swapped. However, the resulting state of some pairs of methods calls is dependent on their execution order. Fig. 2.2.(a) shows an execution where a course c is added and deleted concurrently at two replicas. The two method calls are executed without coordination and are broadcast to other replicas and executed on arrival. Thus, the two replicas execute the two method calls in two different orders and their final states diverge. Reordering the execution of adding and removing a value from a set does not result in the same state. (As we will see in § 2.5, particular CRDT sets can converge even when their operations reorder [424].) As Fig. 2.2.(b) shows, we say that two method calls $\mathcal{S}$*-commute* (state-commute) written as $c_1 \leftrightarrows_{\mathcal{S}} c_2$, iff starting from the same pre-state, executing them in either of the two orders results in the same post-state. Otherwise, we say that they $\mathcal{S}$*-conflict* (state-conflict) and need synchronization; they should be executed one at a time so that they have the same order across replicas.

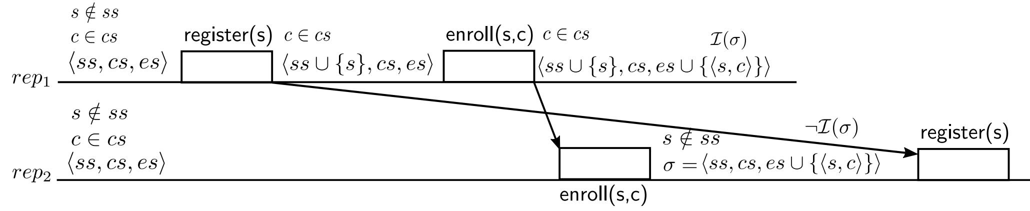
A method call such as registering a student always preserves the invariant. It adds a student and cannot result in a missing student or course in the enrolment relation. Thus, if it is broadcast and executed on a replica whose state satisfies the invariant, it preserves the invariant. We call such method calls *invariant-sufficient*. However, not all method calls are invariant-sufficient. Fig. 2.2.(c) shows an execution where the enrolment of a student s in a course c is executed in the first replica. This method call preserves the invariant as both the student s and the course c belong to the corresponding relations. A method call



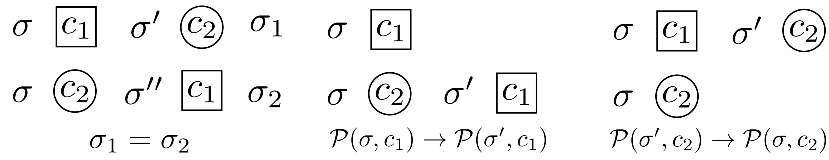
(a) $\mathcal{S}$-conflict



(c) $\mathcal{P}$-conflict



(e) Dependence



(b) $c_1 \leftrightarrows_{\mathcal{S}} c_2$ (d) $c_1 \rightarrow_{\mathcal{P}} c_2$ (f) $c_2 \leftarrow_{\mathcal{P}} c_1$

State-Commutativity $\mathcal{P}$-R-Commutativity $\mathcal{P}$-L-Commutativity

Figure 2.2: Incorrect Executions and Coordination Avoidance Conditions. Square and circle around method calls in (b) , (d) and (f) are just visual aids to see the movements.

that deletes the course c is executed concurrently in the second replica. The enroll call is broadcast and received at the second replica after the delete call. It does not preserve the invariant at the second replica as it is enrolling in a missing course. These two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls such as enrolling in a course and adding the course do not need synchronization. We say that the call c_1 *$\mathcal{P}$-R-commutes* (permissible-right-commutes) with the call c_2 written as $c_1 \rightarrow_{\mathcal{P}} c_2$, iff c_1 stays permissible if it is moved right after c_2. More precisely, as Fig. 2.2.(d) shows, for every state σ, if c_1 is permissible in σ, then it is permissible after applying c_2 to σ as well. We say that a method call c_1 *$\mathcal{P}$-concurs* (permissible-concurs) with another call c_2 iff either c_1 is invariant-sufficient or c_1 $\mathcal{P}$-R-commutes with c_2. Otherwise, we say that c_1 *$\mathcal{P}$-conflicts* (permissible-conflict) with c_2 and they need synchronization. Enrolling in a course $\mathcal{P}$-concurs with adding the course; however, enrolling in a course $\mathcal{P}$-conflicts with deleting the course, therefore; they should synchronize.

We say that two method calls *concur* iff they both $\mathcal{S}$-commute and $\mathcal{P}$-concur with each other. Otherwise, we say they *conflict* and need synchronization. We statically analyze methods of the object and determine whether they satisfy these properties. Fig. 2.1.(b) and (c) show the result of the analysis for $\mathcal{S}$-commute and $\mathcal{P}$-concur on the courseware use-case and based on them, Fig. 2.1.(d) shows the concur relation. The conflict relation is the complement of the concur relation. Fig. 2.1.(e) shows the conflict graph where edges connect pairs of conflicting methods. In our running example, deleting a course conflicts with adding a course and enrolment.

As explained above, invariant-sufficient method calls always preserve the invariant. However, there are calls whose preservation of the invariant is dependent on the calls that have executed before them at that replica. Fig. 2.2.(e) shows an execution where a student is registered and subsequently enrolled in a course. The method calls are broadcast, reordered during transmission and executed in the opposite order in the second replica. The invariant holds after the enrolment in the first replica as it enrolls an existing student in a course. The student has been just registered. However, the enrolment violates the invariant in the second replica. As the student is enrolled before she is registered, a missing student is enrolled which violates the referential integrity of the enrolment relation. Nonetheless, an enrolment is independent of other enrolments. We say that a method call c_2 $\mathcal{P}$*-L-commutes* (permissible-left-commutes) with a method call c_1 written as $c_2 \leftarrow_{\mathcal{P}} c_1$, iff c_2 remains permissible if it is moved left before c_1. More precisely, as Fig. 2.2.(f) shows, for every state σ, if c_2 is permissible in the state resulted from executing c_1 on σ, then c_2 is permissible in σ as well. We say that a method call c_2 is *independent* of c_1 iff c_2 is either invariant-sufficient or $\mathcal{P}$-L-commutes with c_1. The dependencies of a method call is the set of method calls that it is dependent on. If c_2 is dependent on c_1 and c_1 is executed before c_2 in the originating replica of c_2, then c_2 should be applied to other replicas only if c_1 is already applied. In Fig. 2.2.(e), the enrolment is not invariant-sufficient and does not $\mathcal{P}$-L-commute with the registration of the student; thus, the enrolment is dependent on the registration. The enrolment in the second replica should be postponed to after the student is registered. Nonetheless, an enrolment $\mathcal{P}$-L-commutes with other enrolments. Fig. 2.1.(f) shows the result of static analysis for the independence relation on the courseware use-case. The dependence relation is

the complement of the independence relation. The dependence graph is shown in Fig. 2.1.(g). Enrolment is dependent on registration and adding a course.

We say that an execution is *conflict-synchronizing* if the same order is enforced for conflicting method calls across all replicas. We say that an execution is *dependency-preserving* if every method call is executed only after its dependencies from its originating replica are already executed. We define *well-coordinated* executions as locally permissible, conflict-synchronizing and dependency-preserving executions. In § 2.3, we formally define well-coordination and prove that it is sufficient for consistency and convergence of replicas.

Protocols. We now outline our protocols that provide well-coordination and are used to synthesize replicated objects. For the given object, a static analysis finds the conflict and dependency relations that we saw above. The analysis results are used to instantiate the protocols. In this overview, we assume that methods are independent and focus on synchronization of conflicting methods. We outline two protocols. The first protocol is non-blocking and makes progress even if some replicas crash. The second protocol is blocking but can execute calls on one method of a conflicting pair without synchronization.

Non-Blocking Protocol. The high-level idea is to find sets of conflicting calls and synchronize calls in each set. We remember that a clique is a subset of the vertices of a graph such that any of its distinct pair of vertices are adjacent. We find the maximal cliques of the conflict graph and synchronize the methods of each maximal clique with each other. For example, in the conflict graph of the courseware use-case shown in Fig. 2.1.(e), the maximal cliques are $cl_1 = \{d, a\}$ and $cl_2 = \{d, e\}$ where d is deletion, a is addition and e is enrolment. Deletion d and addition a and also deletion d and enrolment e should synchronize with each

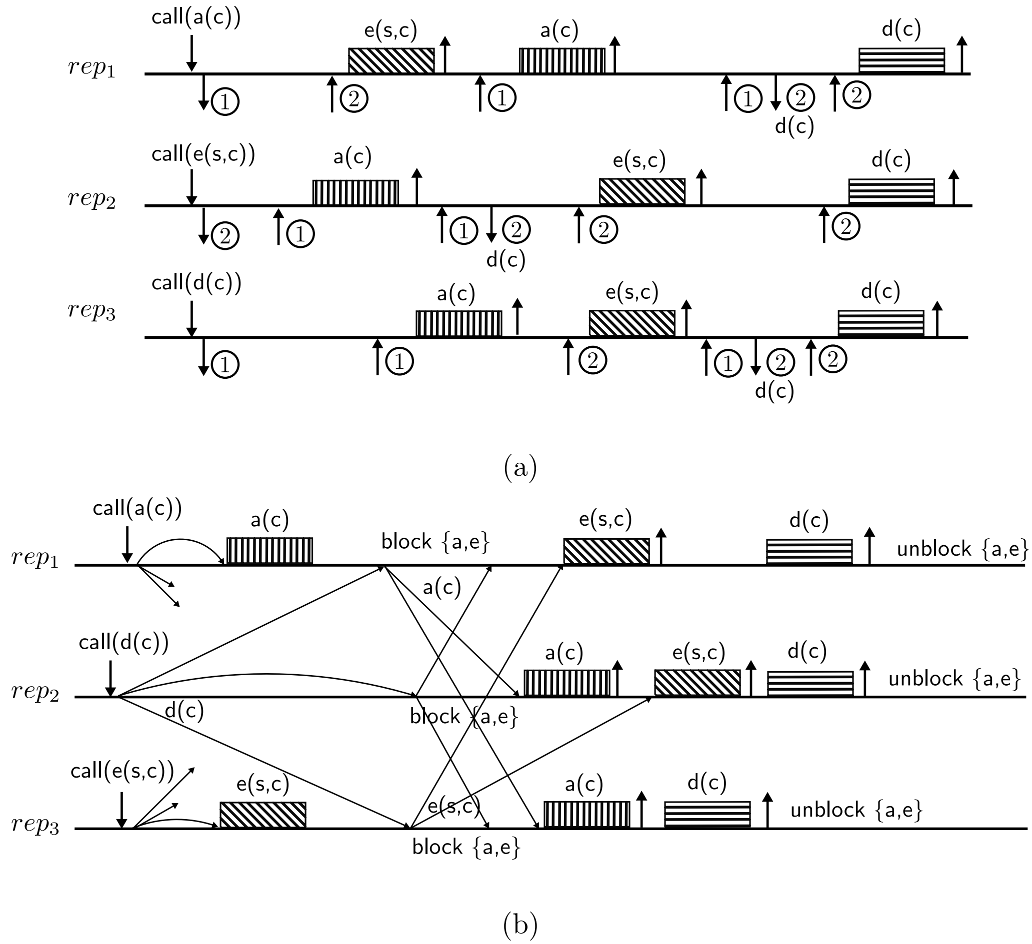


Figure 2.3: (a) Non-blocking Synchronization Protocol. The symbols ↓ and ↑ show requests to and responses from the protocols. Events to the main protocol are shown above and events to the sub-protocols are shown below the horizontal time line. The symbols ① and ② represent events of the first and second TOB sub-protocols respectively. Blocks show the execution of method calls. (b) Blocking Synchronization Protocol. The symbols ↓ and ↑ show requests to and responses from the protocols. Diagonal arrows show message transmission.

other. Deletion d is a member of two cliques and should synchronize in both. We use a variant of the classical total-order broadcast (TOB) protocol to deliver method calls in the same order at all replicas. We use a TOB instance for each maximal clique. In our example, we use the TOB instances tob_1 and tob_2 for the cliques cl_1 and cl_2. A call on a method such as d that is a member of multiple maximal cliques should be totally ordered with respect to methods of each of those cliques. The call is broadcast to each TOB instance and is executed only when it is ordered and delivered by all of them. The non-blocking property of the protocol is derived from the termination property of TOB when a majority of nodes are correct.

As an example, Fig. 2.3.(a) shows an execution of the protocol on the courseware use-case. Three methods are called at three replicas: adding a a course c, enrolling e a student s in the course c and deleting d the course c. The call a is broadcast using tob_1, and the call e is broadcast using tob_2. The call d has to be broadcast to both tob_1 and tob_2. It is first broadcast to tob_1. The sub-protocol tob_1 decides to order and deliver a before d. Thus, a is delivered first and executed at the three replicas. The sub-protocol tob_2 independently delivers e. It is notable that the execution order of e and a that belong to distinct cliques and are broadcast to distinct TOB instances are different in the first and the second replica. Once d is delivered by tob_1, it is broadcast to tob_2. It is finally delivered by tob_2 as well and executed. Thus, the call d is finally executed after both a and e at all replicas.

In the above execution, when the call d is delivered by tob_1, it is implicitly assigned a particular place in the total order of calls in the first clique. However, it cannot execute on delivery from tob_1 and should be broadcast by tob_2. To keep the place of d, other calls

delivered by tob_1 should wait for d to finish its synchronization in the second clique. Therefore, we use a queue per TOB. Method calls that are delivered by a TOB are enqueued to its corresponding queue. A call should wait and can be executed only when it appears at the head of the queues of all TOBs that it is broadcast to. Unfortunately naive implementation of waiting can potentially make mutual waiting and deadlocks. For example, two calls on d can be ordered differently by tob_1 and tob_2 and wait for each other in a deadlock. In § 2.6.1, we revisit this problem and present and use a novel variant of TOB called multi-total-order broadcast (MTOB) that prevents deadlocks.

Blocking Protocol. If two method calls conflict, the previous protocol requires both to go through synchronization. We now present an overview of a protocol that pushes synchronization to only one of the two. Consider that there are two conflicting methods m and m' and we want to let calls on m execute without synchronization. The idea is that calls on m' reach out to other replicas, block the execution of calls on m (so that new calls on m are not accepted) and then replicas exchange updates on preceding calls on m. Once the replicas apply the updates, they have the same set of executed calls on m. Then, the call on m' is executed at all replicas and calls on m are unblocked.

We remember that a minimum vertex cover of a graph is a smallest subset of the vertices such that every edge has at least one endpoint in the cover. To avoid synchronization, we find a minimum vertex cover of the conflict graph and synchronize only when methods in the cover are called. For example, in the conflict graph of the courseware use-case, shown in Fig. 2.1.(e), the minimum vertex cover is the singleton set of the delete method $\{d\}$. Only deletion d performs synchronization and addition a and enrolment e can execute without

synchronization. Further, methods can be assigned weights inversely proportional to their call frequency and weighted minimum vertex cover can optimize the average responsiveness of the replicated object.

As an example, Fig. 2.3.(b) shows an execution of the protocol on the courseware use-case. The first and the third replicas call synchronization-free methods a and e. They are simply broadcast and executed on arrival. In this execution, the delivery of these messages are delayed. The second replica calls method d. The call d is broadcast and on its delivery, all replicas block the conflicting methods a and e. To update other replicas, each replica subsequently broadcasts the set of conflicting method calls that it has executed. The first and third replicas broadcast their calls on a and e respectively. These updates are applied on arrival. After all the updates are applied, every replica has executed the same set of calls that conflict with d although possibly in different orders. Then, the call on d is executed and the conflicting methods are unblocked. This protocol makes replicas wait for each other; thus, crash of a replica can prevent progress of others. Following fundamental impossibility results [164, 182], this protocol has a trade-off between availability and consistency. We will revisit this trade-off in § 2.6.2.

2.3 Well-Coordination

In this section, we define the well-coordination condition and prove that it is sufficient for state integrity and convergence. We first define replicated executions and their correctness. Then, we present the well-coordination conditions and prove that well-coordinated executions are correct.

An object is a record $\langle \Sigma, \mathcal{I}, \mathcal{M} \rangle$ that includes a state type Σ, an invariant $\mathcal{I}$ on the state, and a set of methods $\mathcal{M}$. A method is a function m from the parameters and the pre-state to a record of $\langle \mathsf{guard}, \mathsf{update}, \mathsf{retv} \rangle$, where guard is a boolean expression that defines when the method can be called, and update and retv are expressions for the post-state and the return value. We use guard, update and retv as functions that extract elements of the record. A method call c is a method applied to its argument i.e. a function from the current state to a record of $\langle \mathsf{guard}, \mathsf{update}, \mathsf{retv} \rangle$.

Execution. We first define the context $\mathbf{c}$ for a replicated execution. The state of each replica is initialized to the same state σ_0 that satisfies the invariant $\mathcal{I}$. The user can request a call on a method at every replica that is called the originating replica of the call. The call is then propagated from the originating replica and executed at other replicas. We uniquely identify requests by identifiers.

Definition 1 (Execution Context) *An execution context* $\mathbf{c}$ *is the record* $\langle \sigma_{0\mathbf{c}}, \mathsf{R}_{\mathbf{c}}, \mathsf{call}_{\mathbf{c}}, \mathsf{orig}_{\mathbf{c}} \rangle$ *where* $\sigma_{0\mathbf{c}}$ *is an initial state that satisfies the invariant i.e.* $\mathcal{I}(\sigma_{0\mathbf{c}})$*,* $\mathsf{R}_{\mathbf{c}}$ *is a set of request identifiers,* $\mathsf{call}_{\mathbf{c}}$ *is a function from* $\mathsf{R}_{\mathbf{c}}$ *to method calls, and* $\mathsf{orig}_{\mathbf{c}}$ *is a function from* $\mathsf{R}_{\mathbf{c}}$ *to replicas* $\mathcal{N}$*.*

We model an execution at a replica as a permutation of a set of request identifiers.

Definition 2 (Execution) *In a context* $\mathbf{c}$*, an execution* x *of a set of requests* $R \subseteq \mathsf{R}_{\mathbf{c}}$ *is a bijective from positions* $[0..|R|-1]$ *to* R*.*

We denote the range of x *as* $\mathsf{R}(\mathsf{x})$*. An execution* x *of* R *defines the total order* $\prec_{\mathsf{x}}$ *on* R*. A request* r *precedes another request* r' *in an execution* x *written as* $r \prec_{\mathsf{x}} r'$ *iff* $\mathsf{x}^{-1}(r) < \mathsf{x}^{-1}(r')$*.*

In a replicated execution, calls are propagated and eventually executed at every replica. Convergence is a condition on the state of the replicas after all calls are applied at all replicas. Therefore, a replicated execution is a mapping from replicas to permutations of the same set of calls. For example, Fig. 2.4.(a) shows a replicated execution where nine requests are executed. Propagation of calls from the originating replicas to other replicas creates a visibility relation between calls across replicas. For example, in Fig. 2.4.(a), arrows show the visibility relation. Consequently, the happens-before relation is the transitive closure of the visibility relation and the execution order of each replica. The happens-before relation is acyclic. In Fig. 2.4.(a), as the direction of all arrows is forward, the happens-before relation is acyclic.

Definition 3 (Replicated Execution) *In a context* $\mathbf{c}$*, a replicated execution* xs *is a function from replicas* $\mathcal{N}$ *to executions of* $\mathsf{R}_{\mathbf{c}}$ *such that (1) let the execution order* $\prec_{\mathsf{xs}}$ *on* $\mathcal{N} \times \mathsf{R}_{\mathbf{c}}$ *be defined as: for every replica* n *and pair of requests* r *and* r'*,* $(n, r) \prec_{\mathsf{xs}} (n, r')$ *iff* $r \prec_{\mathsf{xs}(n)} r'$*, (2) let the visibility relation* $\rightsquigarrow_{\mathsf{xs}}$ *on* $\mathcal{N} \times \mathsf{R}_{\mathbf{c}}$ *be defined as: for every request* r*, for every replica* n*,* $(\mathsf{orig}_{\mathbf{c}}(r), r) \rightsquigarrow_{\mathsf{xs}} (n, r)$ *iff* $n \neq \mathsf{orig}_{\mathbf{c}}(r)$*, (3) let the happens-before relation* $\mathsf{hb}_{\mathsf{xs}}$ *be* $(\prec_{\mathsf{xs}} \cup \rightsquigarrow_{\mathsf{xs}})^*$ *then,* $\mathsf{hb}_{\mathsf{xs}}$ *is acyclic.*

The post-state of each call at a replica is the result of applying the call to its pre-state. Thus, a sequence of calls result in a sequence of states.

Definition 4 (State) *In a context* $\mathbf{c}$*, the state function* $\mathbf{s}$ *of an execution* x *is a function from positions* $[0..|\mathsf{R}(x)|]$ *to states* Σ *such that* $\mathbf{s}(0) = \sigma_{0\mathbf{c}}$ *and for every* $0 \leq i < |\mathsf{R}(x)|$*,* $\mathbf{s}(i+1) = \mathsf{update}(\mathsf{call}_{\mathbf{c}}(x(i)))(\mathbf{s}(i))$*. The state function is lifted to replicated executions. The*

state function $\mathbf{ss}$ *of a replicated execution* xs *is a function from replicas* n *in* $\mathcal{N}$ *to the state function of the execution* $\mathsf{xs}(n)$.

Correctness. We now define correctness as convergence and integrity.

A replicated execution is convergent if it leads to the same final state for all replicas.

Definition 5 (Convergent) *A replicated execution* xs *of a context* $\mathbf{c}$ *is convergent iff for every pair of replicas* n *and* n', $\mathbf{ss}(n)(|R_{\mathbf{c}}|) = \mathbf{ss}(n')(|R_{\mathbf{c}}|)$ *where* $\mathbf{ss}$ *is the state function of* xs.

In the definition of methods of an object, the user relies on the invariant in the pre-state. Further, methods have explicit guards that define the subset of states that they are applicable to. We say that a method call is consistent at a state if the invariant and the guard of the method hold in that state. Method calls should be executed only on states that they are consistent in.

Definition 6 (Consistent Call) *A method call* c *is consistent in a state* σ, *written as* $\mathsf{cons}(\sigma, c)$, *iff* $\mathsf{guard}(c)(\sigma)$ *and* $\mathcal{I}(\sigma)$.

The consistency condition is simply lifted to executions and replicated executions.

Definition 7 (Consistent Execution) *In a context* $\mathbf{c}$, *a request* r *is consistent in an execution* x *written as* $\mathsf{cons}(\mathbf{c}, \mathsf{x}, r)$ *iff* $\mathsf{cons}(\mathbf{s}(i), \mathsf{call}_{\mathbf{c}}(r))$ *where* $\mathbf{s}$ *is the state function of* x, *and* i *is* $\mathsf{x}^{-1}(r)$. *In a context* $\mathbf{c}$, *an execution* x *is consistent written as* $\mathsf{cons}(\mathbf{c}, \mathsf{x})$ *iff every request* r *in* $R(\mathsf{x})$ *is consistent in* x. *A replicated execution* xs *of a context* $\mathbf{c}$ *is consistent written as* $\mathsf{cons}(\mathbf{c}, \mathsf{xs})$ *iff for every replica* n, *the execution* $\mathsf{xs}(n)$ *is consistent.*

Consistency of a replicated execution requires invariant preservation (that is state integrity) at all replicas. We define correctness as both consistency and convergence.

Definition 8 (Correct) *A replicated execution is correct iff it is consistent and convergent.*

Well-coordination. Now, we define the well-coordination conditions. We say that a call is permissible in a state iff its guard holds in that state and the invariant holds after the call is applied.

Definition 9 (Permissible Call) *A method call c is permissible in a state σ, written as $\mathcal{P}(\sigma, c)$, iff $\mathsf{guard}(c)(\sigma)$ and $\mathcal{I}(\mathsf{update}(c)(\sigma))$.*

Note that in contrast to the definition of consistency above that requires the invariant to hold in the pre-state, permissibility requires it to hold in the post-state. By induction, permissibility leads to consistency. The initial state satisfies the invariant; thus, for every call, if all the previous calls have maintained the invariant, the call is applied to a state that satisfies the invariant as well. Permissibility implies that the call preserves the invariant. Similar to consistency, permissibility is simply lifted to executions and replicated executions.

Well-coordination requires each call to be permissible in its originating replica. If a call is requested at a replica but is not permissible in its current state, the call should be aborted (and maybe retried later).

Definition 10 (Locally permissible) *A replicated execution xs of a context $\mathbf{c}$ is locally permissible iff every request r is permissible in the execution of its originating replica $\mathsf{orig}_{\mathsf{c}}(r)$.*

Although permissibility is directly checked only locally at the originating replicas, we will show that well-coordination conditions ensure the global permissibility of calls at every replica.

As we saw in Fig. 2.2.(b), we say that two method calls $\mathcal{S}$-commute (state-commute) if starting from every pre-state, the post-state is the same if the calls are reordered.

Definition 11 (State-Commutativity and State-Conflict) *Two method calls c_1 and c_2 S-commute, written as $c_1 \leftrightarrows_{\mathcal{S}} c_2$ iff for every state σ, $\mathsf{update}(c_2)(\mathsf{update}(c_1)(\sigma)) = \mathsf{update}(c_1)(\mathsf{update}(c_2)(\sigma))$. Otherwise, they $\mathcal{S}$-conflict, written as $c_1 \bowtie_{\mathcal{S}} c_2$.*

$\mathcal{S}$-conflicting calls need synchronization since we saw in Fig. 2.2.(a) that they cause state divergence.

We note that $\mathcal{S}$-commutativity and the following properties are defined on (dynamic) method calls; however, they are simply lifted to (static) methods. For instance, we say that two methods $\mathcal{S}$-commute iff all calls on the two $\mathcal{S}$-commute. In § 2.4, we consider these properties on methods.

There are calls such as deposit on a bank account that are always permissible as far as they are applied to a state that satisfies the invariant. We call these calls invariant-sufficient.

Definition 12 (Invariant-Sufficient) *A call c is invariant-sufficient iff for every state σ if $\mathcal{I}(\sigma)$ then $\mathcal{P}(\sigma, c)$.*

Every call is checked to be permissible in its originating replica. However, as we saw in Fig. 2.2.(c), if a call is simply broadcast, when it arrives at other replicas, other calls may have been executed at the destination replicas that were not executed at the originating replica. These extra calls may make the arrived call impermissible. As we saw in Fig. 2.2.(d), we say that a method call $\mathcal{P}$-R-commutes (permissible-right-commutes) another if starting from any state where the former is permissible, moving it right after the latter does not violate permissibility.

Definition 13 (Permissible-Right-Commutativity) *The call c_1 $\mathcal{P}$-R-commutes with the call c_2 written as $c_1 \rightarrow_{\mathcal{P}} c_2$ iff for every state σ, if $\mathcal{P}(\sigma, c_1)$ then $\mathcal{P}(\mathsf{update}(c_2)(\sigma), c_1)$.*

If a call is invariant-sufficient or $\mathcal{P}$-R-commutes another call, we say that the former $\mathcal{P}$-concurs (permissible-concurs) with the latter. Otherwise, we say that the former $\mathcal{P}$-conflicts (permissible-conflicts) with the latter.

Definition 14 (Permissible-Concur and Permissible-Conflict) *A call c_1 $\mathcal{P}$-concurs with a call c_2 iff c_1 is invariant-sufficient or $c_1 \rightarrow_{\mathcal{P}} c_2$. Otherwise, c_1 $\mathcal{P}$-conflicts with c_2.*

A pair of calls can avoid synchronization only if they both $\mathcal{S}$-commute and $\mathcal{P}$-concur with respect to each other.

Definition 15 (Concur and Conflict) *A pair of calls c_1 and c_2 concur iff they state-commute and permissible-concur with each other. Otherwise, they conflict written as $c_1 \bowtie c_2$.*

Concur and conflict relations are symmetric. The conflict relation on methods can be represented as the conflict graph $G_{\bowtie}$: an undirected graph where the vertices are the set of methods and the edges are the pairs of conflicting methods. A replicated execution is conflict-synchronizing if every pair of conflicting calls have the same order across replicas.

Definition 16 (Conflict-synchronizing) *A replicated execution xs of a context c is conflict-synchronizing iff for every pair of requests r and r' in rc_c such that $\mathsf{call}_{\mathsf{c}}(r) \bowtie \mathsf{call}_{\mathsf{c}}(r')$, for every pair of replicas n and n', if $r \prec_{\mathsf{xs}(n)} r'$ then $r \prec_{\mathsf{xs}(n')} r'$.*

Similar to conflict-synchronizing, state-conflict-synchronizing and permissible-conflict-synchronizing are similarly defined with respect to state-conflict and permissible-conflict.

As we saw in Fig. 2.2.(e), when a call arrives at other replicas, other calls that were executed at the originating replica may have not arrived and executed at destination replicas. However, permissibility of the call may be dependent on the missing calls. As we saw in Fig. 2.2.(f), we say that a method call $\mathcal{P}$-L-commutes (permissible-left-commutes) with another if moving the former left before the latter does not render the former impermissible.

Definition 17 (Permissible-Left-Commutative) *A call* c_2 $\mathcal{P}$*-L-commutes a call* c_1*, written as* $c_2 \leftarrow_{\mathcal{P}} c_1$ *iff for every state* σ*, if* $\mathcal{P}(\mathsf{update}(c_1)(\sigma), c_2)$ *then* $\mathcal{P}(\sigma, c_2)$*.*

A call can avoid tracking dependencies to another call if the former is invariant-sufficient or $\mathcal{P}$-L-commutes with the latter.

Definition 18 (Independent and Dependent) *A call* c_2 *is independent of* c_1*, written as* $c_2 \perp\!\!\!\perp c_1$*, iff either* c_2 *is invariant-sufficient or* $c_2 \leftarrow_{\mathcal{P}} c_1$*. Otherwise,* c_2 *is dependent on* c_1*, written as* $c_2 \not\perp\!\!\!\perp c_1$*.*

The dependency relation between methods can be represented as a directed graph that we call the dependency graph. A replicated execution is dependency-preserving if for every call, its preceding dependencies in its originating replica precede it in the other replicas as well.

Definition 19 (Dependency-Preserving) *A replicated execution* xs *of a context* $\mathbf{c}$ *is dependency-preserving iff for every pair of requests* r *and* r' *in* $\mathsf{R}_{\mathbf{c}}$*, such that* $\mathsf{call}_{\mathbf{c}}(r') \not\perp\!\!\!\perp \mathsf{call}_{\mathbf{c}}(r)$*, if* $r \prec_{\mathsf{xs}(\mathsf{orig}_{\mathbf{c}}(r'))} r'$*, then for every replica* n*,* $r \prec_{\mathsf{xs}(n)} r'$*.*

We note that in Theorem 16, call orders in any replica necessitates the same orders in other replicas. In contrast, in Theorem 19, only orders between a call and its preceding calls in its *originating replica* necessitates the same order in other replicas.

A replicated execution is well-coordinated if the permissibility of calls are checked at the originating replicas, conflicting calls are synchronized and the dependencies are preserved. Well-coordination is a sufficient condition for the correctness of replicated executions.

Definition 20 (Well-coordination) *A replicated execution is well-coordinated iff it is locally permissible, conflict-synchronizing, and dependency-preserving.*

Theorem 21 () *Every well-coordinated replicated execution is correct.*

The proof follows from the definition of well-coordination and correct (Theorem 20 and Theorem 8) and the following two lemmas. We present the high-level ideas.

Lemma 22 () *Every $\mathcal{S}$-conflict-synchronizing replicated execution is convergent.*

Consider two executions x and x' from the replicated execution (with the same set of requests possibly in different orders). Assume that x and x' are $\mathcal{S}$-conflict-synchronizing with respect to each other. We prove that these two executions result in the same post-state. By induction, x' can be incrementally converted to x from left to right without changing its final post-state. Assume that the requests until location i are the same in x and x'. Consider the request r at position i in x. If r appears later at position j in x' where $j > i$, then we show that r can be moved left in x' to position i. The requests between i and j in n' precede r in x' but succeed r in x. Therefore, by the $\mathcal{S}$-conflict-synchronization condition, r

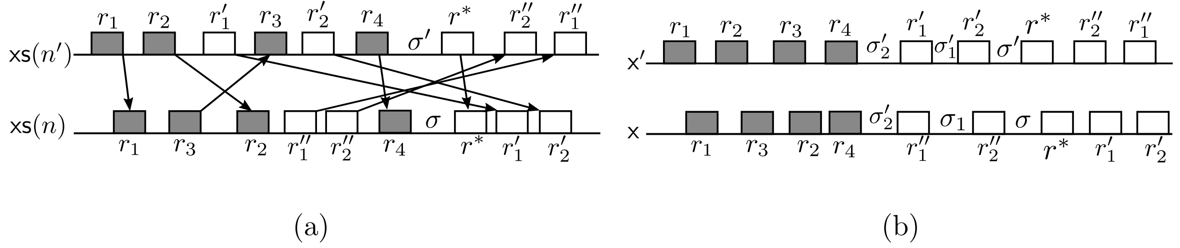


Figure 2.4: Correctness of well-coordinated replicated executions

$\mathcal{S}$-commutes with requests between i and j in x'. Thus, r can be moved left to location i in x' without any change to the post-state.

Lemma 23 () *Every well-coordinated replicated execution is consistent.*

We illustrate the crucial part of the proof by a figure. Let xs be a coordinated replicated execution. To prove consistency of xs, we need to prove consistency of every request at the execution of every replica. We will prove that every request at every replica is permissible. This implies that (1) the guard of every request is satisfied. and (2) the post-state of every request satisfies the invariant. Based on [2] and the fact that the initial state is defined to satisfy the invariant, we have that (3) the pre-state of every request satisfies the invariant. From the facts [1] and [3] above, we have that xs is consistent. We now show the permissibility of every request r^*. The proof is by induction on a linear extension of $\mathsf{hb}_{\mathsf{xs}}$. Let the request r^* at the replica n be the current request. If n is the originating replica of r^*, then r^* is trivially permissible by the locally permissible condition; it states that every replica only originates permissible requests. Otherwise, let n' be the originating replica of r^*. If r^* is invariant-sufficient, we only need to show that the pre-state of r^* in n satisfies the invariant. The pre-state of r^* is either the initial state that by definition satisfies the

invariant or is the post-state of the preceding request in n. By the induction hypothesis, the preceding request is permissible that implies that its post-state satisfies the invariant.

Now we consider that r^* is not invariant-sufficient. We illustrate the proof of permissibility of r^* in Fig. 2.4. Let σ be the pre-state of r^* in $\mathsf{xs}(n)$. We want to show that r^* is permissible in σ. Let σ' be the pre-state of r^* in $\mathsf{xs}(n')$ (the execution of the originating replica). Let R be the requests that precede r^* in both $\mathsf{xs}(n)$ and $\mathsf{xs}(n')$. In Fig. 2.4.(a), R is the set of shaded requests $\{r_1, r_2, r_3, r_4\}$. Let R' be the requests that precede r^* in $xs(n')$ but *do not* precede r^* in $\mathsf{xs}(n)$. In Fig. 2.4.(a), R' is $\{r'_1, r'_2\}$. Consider a request r in R and a request r' in R' such that r' precedes r in $\mathsf{xs}(n')$. In Fig. 2.4.(a), r can be r_4 and r' can be r'_2. The request r' precedes r in $\mathsf{xs}(n')$ but succeeds it in $\mathsf{xs}(n)$. Therefore, by the $\mathcal{S}$-conflict-synchronization condition, r' and r $\mathcal{S}$-commute. In Fig. 2.4.(a), we commute r'_2 with r_4. Then, we commute r'_1 with r_3 and r_4. Thus, by induction, each request in R' from the rightmost to the leftmost in $\mathsf{xs}(n')$ can be moved right to form a block of requests before r^* in $\mathsf{xs}(n')$ without changing the pre-state σ' of r^*. Let x' denote the result of the commute. Fig. 2.4.(b) shows x' where the pre-state of r^* is still σ'. In Fig. 2.4.(a), the requests R' precede r^* in $\mathsf{xs}(n')$ but succeed it in $\mathsf{xs}(n)$. Therefore, by the dependency-preserving condition, r^* is independent of the requests in R'. In Fig. 2.4.(a), r^* is independent of r'_1 and r'_2. By the locally permissible condition and that n' is the originating replica of r^*, the request r^* is permissible at its pre-state σ' in $\mathsf{xs}(n')$. By induction from right to left in x', using the independence condition, r^* is permissible at the pre-state of each request r' in R'. Thus, r^* is permissible at the pre-state of R' that is the post-state of R in x'. In Fig. 2.4.(b), r^* is permissible at the states σ'_1 and σ'_2.

The argument above for moving requests in $\mathsf{xs}(n')$ can be applied to $\mathsf{xs}(n)$ as well. Let R'' be the requests that precede r^* in $xs(n)$ but do not precede it in $xs(n')$. In Fig. 2.4.(a), R'' is $\{r''_1, r''_2\}$. $\mathcal{S}$-commutativity allows moving R'' right in $\mathsf{xs}(n)$. The requests R'' can be moved to form a block immediately before r^* without changing the pre-state of r^*. Let x denote the result of the commute. Fig. 2.4.(b) shows x. The requests $\{r''_1, r''_2\}$ moved right immediately before r^*. The set of requests R appear on the left side of both x and x' although possibly in different orders. By the argument presented above for Theorem 22 using $\mathcal{S}$-commutativity, it is proved that the post-state of the set of requests R in x and x' is the same. We showed above that r^* is permissible in the post-state of R in x'. Thus, r^* is permissible in the post-state of R in x as well. In other words, r^* is permissible at the pre-state of the set of requests R'' in x. In Fig. 2.4.(b), r^* is permissible in σ'_2, the post-state of r_4 in x.

The requests R'' precede r^* in $\mathsf{xs}(n)$ but succeed it in $\mathsf{xs}(n')$. Therefore, by the $\mathcal{P}$-conflict-synchronization condition, each request in R'' $\mathcal{P}$-R-commutes with r^*. In Fig. 2.4.(a), r^* $\mathcal{P}$-R-commute with r''_1 and r''_2. We proved above that the request r^* is permissible at the pre-state of R'' in x. By induction from left to right in x, using the $\mathcal{P}$-R-commutativity, r^* is permissible at the post-state of each request r'' in R''. Therefore, r^* is permissible at its pre-state σ in x. In Fig. 2.4.(b), r^* is permissible at the states σ_1 and σ. Therefore, r^* is permissible at its pre-state in $\mathsf{xs}(n)$.

We note that conflict-synchronization is stronger than dependency-preservation. If a request r both conflicts with and depends on r', it is sufficient to synchronize r with r' and its dependencies to r' do not need to be tracked.

2.4 Static Analysis

In the previous section, we defined conflict and dependency relations between methods. In this section, we recast the definitions as a static analysis that calculates these relations. The user specifies an object $\langle \Sigma, \mathcal{I}, \mathcal{M} \rangle$ where Σ is the state type, $\mathcal{I}$ is the invariant and $\mathcal{M}$ is the set of methods. Given the object, Fig. 2.5 presents two functions $\mathsf{ConflictRel}()$ and $\mathsf{DepRel}()$ that calculate the two relations. We consider each one in turn and apply them to our running example.

The function $\mathsf{ConflictRel}()$ returns the conflict relation as a mapping from pairs of methods $\mathcal{M} \times \mathcal{M}$ to boolean $\mathbb{B}$. It first calculates the $\mathcal{S}$-commutativity relation in the variable SCom (at lines C_6-C_7). Following Theorem 11, for every pair of methods m_1 and m_2, $\mathsf{SCom}(m_1, m_2)$ is true iff the following assertion is valid: for every pre-state σ, argument a_1 of m_1 and argument a_2 for m_2, the post-states of applying the two calls $m_1(a_1)$ and $m_2(a_2)$ on σ in the two different orders are equal. We use the notation $\vdash \mathcal{A}$ to represent whether the assertion $\mathcal{A}$ is valid. To check the validity of an assertion, we use SMT solvers to check the satisfiability of its negation.

For example, Fig. 2.1.(b) shows that the two methods $\mathsf{addCourse}$ and enroll $\mathcal{S}$-commute. Let us see how this is calculated. To calculate the value of $\mathsf{SCom}(\mathsf{addCourse}, \mathsf{enroll})$, the assertion in line C_7 is instantiated to the following assertion. (The pre-state σ is expanded to $\langle ss, cs, es \rangle$, the argument of $\mathsf{addCourse}$ is c and the arguments of enroll are s and c'.)

$$\begin{aligned} \vdash \forall ss, cs, es, c, s, c'.\ &\mathsf{update}(\mathsf{enroll}(s, c'))(\mathsf{update}(\mathsf{addCourse}(c))(\langle ss, cs, es \rangle)) = \\ &\mathsf{update}(\mathsf{addCourse}(c))(\mathsf{update}(\mathsf{enroll}(s, c'))(\langle ss, cs, es \rangle)) \end{aligned} \tag{2.1}$$

```
     fun ConflictRel(): 𝓜 × 𝓜 → 𝔹 {
C1   var SCom: 𝓜 × 𝓜 → 𝔹
C2   var ISuff: 𝓜 → 𝔹
C3   var PRCom, PConcur: 𝓜 × 𝓜 → 𝔹
C4   var Concur, Conflict: 𝓜 × 𝓜 → 𝔹
C5   let 𝓟 := λσ, c. guard(c)(σ) ∧ 𝓘(update(c)(σ))
C6   foreach (m1 ∈ 𝓜, m2 ∈ 𝓜)
C7     SCom(m1, m2) :=
         ⊢ ∀σ, a1, a2
           update(m2(a2))(update(m1(a1))(σ)) =
           update(m1(a1))(update(m2(a2))(σ))
C8   foreach (m ∈ 𝓜)
C9     ISuff(m) :=
         ⊢ ∀σ, a. 𝓘(σ) → 𝓟(σ, m(a))
C10  foreach (m1 ∈ 𝓜, m2 ∈ 𝓜)
C11    PRCom(m1, m2) :=
         ⊢ ∀σ, a1, a2.
           𝓟(σ, m1(a1)) →
           𝓟(update(m2(a2))(σ), m1(a1))
C12    PConcur(m1, m2) := ISuff(m1) or
                          PRCom(m1, m2)
C13  foreach (m1 ∈ 𝓜, m2 ∈ 𝓜)
C14    Concur(m1, m2) := SCom(m1, m2) and
                         PConcur(m1, m2) and
                         PConcur(m2, m1)
C15    Conflict(m1, m2) := not Concur(m1, m2)
C16  return Conflict }
```

```
     fun DepRel(): 𝓜 × 𝓜 → 𝔹 {
D1   var ISuff: 𝓜 → 𝔹
D2   var LRCom: 𝓜 × 𝓜 → 𝔹
D3   var Dep, Indep: 𝓜 × 𝓜 → 𝔹
D4   let 𝓟 := λσ, c. guard(c)(σ) ∧ 𝓘(update(c)(σ))
D5   foreach (m ∈ 𝓜)
D6     ISuff(m) :=
         ⊢ ∀σ, a. 𝓘(σ) → 𝓟(σ, m(a))
D7   foreach (m2 ∈ 𝓜, m1 ∈ 𝓜)
D8     PLCom(m2, m1) :=
         ⊢ ∀σ, a1, a2.
           𝓟(update(m1(a1))(σ), m2(a2)) →
           𝓟(σ, m2(a2))
D9     Indep(m2, m1) := ISuff(m2) or
D10                     PLCom(m2, m1)
D11    Dep(m2, m1) := not Indep(m2, m1)
D12  return Dep }
```

Figure 2.5: Static analysis to calculate the conflict and dependency relations. The object $\langle \Sigma, \mathcal{I}, \mathcal{M} \rangle$ is given.

Based on the object definition in Fig. 2.1.(a), the two expressions can be simplified as follows:

$$
\begin{aligned}
\text{Left exp:} \quad & \mathsf{update}(\mathsf{enroll}(s, c'))(\mathsf{update}(\mathsf{addCourse}(c))(\langle ss, cs, es\rangle)) \quad = \\
& \mathsf{update}(\mathsf{enroll}(s, c'))(\langle ss, cs \ \cup \ \{c\}, es\rangle) \quad = \quad \langle ss, cs \ \cup \ \{c\}, es \ \cup \ \{\langle s, c'\rangle\}\rangle \\
\text{Right exp:} \quad & \mathsf{update}(\mathsf{addCourse}(c))(\mathsf{update}(\mathsf{enroll}(s, c'))(\langle ss, cs, es\rangle)) \quad = \\
& \mathsf{update}(\mathsf{addCourse}(c))(\langle ss, cs, es \ \cup \ \{\langle s, c'\rangle\}\rangle) \quad = \quad \langle ss, cs \ \cup \ \{c\}, es \ \cup \ \{\langle s, c'\rangle\}\rangle
\end{aligned}
\tag{2.2}
$$

The two expressions are equal; thus, the assertion is valid and the two methods $\mathcal{S}$-commute.

Similar to $\mathcal{S}$-commutativity, the other relations are calculated by a validity check for their definitions. In summary, the ConflictRel() function calculates the invariant-sufficiency relation (Theorem 12) in the variable ISuff (at C_8-C_9) and the $\mathcal{P}$-R-commutativity relation (Theorem 13) in the variable PRCom (at C_{10}-C_{11}). They are used to calculate the $\mathcal{P}$-concur relation (Theorem 14) in the variable PConcur (at line C_{12}). Then, the concur relation (Theorem 15) for a pair of methods is calculated in the variable Concur as the conjunct of $\mathcal{S}$-commutativity and $\mathcal{P}$-concur of the method pair with respect to each other (at C_{13}-C_{14}). (We note that $\mathcal{S}$-commutativity is symmetric.) Finally, the conflict relation (Theorem 15) is calculated as the negation of the concur relation in the variable Conflict and returned (at C_{15}-C_{16}). These steps calculate the sub-figures (b) to (e) of Fig. 2.1 in order.

The function DepRel() calculates the dependency relation. It first calculates invariant-sufficiency (Theorem 12) in the variable ISuff (at lines D_5-D_6) and $\mathcal{P}$-L-commutativity (Theorem 17) in the variable PLCom (at D_7-D_8). They are used to calculate the independence relation (Theorem 18) in the variable Indep (at D_9-D_{10}). Finally, the dependence relation (Theorem 18) is calculated as the negation of the independence relation in the variable Dep and returned (at D_{11}-C_{12}).

Fig. 2.1.(f) and (g) show that enroll is dependent on $\mathsf{addCourse}$. Let us see how this is calculated. We show that enroll is not invariant-sufficient and does not $\mathcal{P}$-L-commute with $\mathsf{addCourse}$ either. First, we show that the method enroll is not invariant-sufficient. Intuitively, even if the invariant holds in the pre-state of enroll, it does not trivially hold in its post-state. The invariant-sufficiency assertion that is checked at D_6 is instantiated to the following assertion: (The pre-state σ is expanded to $\langle ss, cs, es\rangle$ and the arguments of enroll are s and c.)

$$\vdash \forall ss, cs, es, s, c.\ \mathcal{I}(\langle ss, cs, es\rangle) \rightarrow \mathcal{P}(\langle ss, cs, es\rangle, \mathsf{enroll}(s, c)) \tag{2.3}$$

After unrolling $\mathcal{P}$, the conclusion of the implication includes the following conjunct

$$\begin{aligned} &\mathcal{I}(\mathsf{update}(\mathsf{enroll}(s,c))(\langle ss, cs, es\rangle)) \;=\; \mathcal{I}(\langle ss, cs, es \,\cup\, \{\langle s, c\rangle\}\rangle) \;= \\ &\mathsf{refIntegrity}(es \,\cup\, \{\langle s, c\rangle\}, \mathsf{esid}, ss, \mathsf{sid}) \;\wedge\; \mathsf{refIntegrity}(es \,\cup\, \{\langle s, c\rangle\}, \mathsf{ecid}, cs, \mathsf{cid}) \end{aligned} \tag{2.4}$$

According to the definition of referential integrity in the caption of Fig. 2.1, the first conjunct is expanded to the following assertion:

$$\forall r.\;\; r \in es \,\cup\, \{\langle s, c\rangle\} \rightarrow \exists r'.\ r' \in ss \wedge \mathsf{esid}(r) = \mathsf{sid}(r') \tag{2.5}$$

We note that s is an unconstrained universally quantified variable in Eq. 2.3, the original invariant-sufficiency assertion. Therefore, to falsify that assertion, the variable s can be instantiated with any student value. Enrolling any student s that is not already in ss violates the above referential integrity property and leads to a counter-example for validity for Eq. 2.3. Intuitively, enrolling a student that is not already in the students relation violates integrity. Hence, the method enroll is not invariant-sufficient.

Next, we show that enroll does not $\mathcal{P}$-L-commute with $\mathsf{addCourse}$. Intuitively, the enroll method does not preserve its permissibility if it is moved left before a preceding

$\mathsf{addCourse}$. The assertion in line D_8 is instantiated to the following assertion. (The pre-state σ is expanded to $\langle ss, cs, es \rangle$, the argument of $\mathsf{addCourse}$ is c and the arguments of enroll are s and c'.)

$$\begin{aligned} &\vdash \forall ss, cs, es, c, s, c'. \\ &\quad \mathcal{P}(\mathsf{update}(\mathsf{addCourse}(c))(\langle ss, cs, es \rangle), \mathsf{enroll}(s, c')) \rightarrow \mathcal{P}(\langle ss, cs, es \rangle, \mathsf{enroll}(s, c')) \end{aligned} \tag{2.6}$$

The counter-example is when $c = c'$, that is the same course is added and enrolled, and $c \notin cs$, that is c is not already an existing course. After expansion and removing the trivially valid guard assertions, we have

$$\mathcal{I}(\langle ss, cs \ \cup \ \{c\}, es \ \cup \ \{s, c\} \rangle) \rightarrow \mathcal{I}(\langle ss, cs, es \ \cup \ \{s, c\} \rangle) \tag{2.7}$$

Expanding the conclusion of the implication results in the following conjunct:

$$\forall r.\ r \in es \ \cup \ \{\langle s, c \rangle\} \rightarrow \exists r'.\ r' \in cs \wedge \mathsf{ecid}(r) = \mathsf{cid}(r') \tag{2.8}$$

This assertion is invalid. For $r = \langle s, c \rangle$, the conclusion never holds as $c \notin cs$. This makes a counter-example for the $\mathcal{P}$-L-commutativity assertion. Thus, enroll does not $\mathcal{P}$-L-commute with $\mathsf{addCourse}$. A call on enroll is dependent on the preceding $\mathsf{addCourse}$ call.

We note that the premise of the implication in Eq. 2.7 does not refute the choice that $c \notin cs$. In the premise of Eq. 2.7, the integrity of the enrolment relation $es \ \cup \ \{\langle s, c \rangle\}$ for the course c may hold only because c was just added and resulted in the course relation $cs \ \cup \ \{c\}$ and not because it already existed in cs.

We note that since local permissibility is a condition of a well-coordinated replicated execution, every call is permissible in its originating node. Therefore, every call in all the conditions above can be additionally assumed to be permissible in a fresh state (unrelated to

the other state variables in the condition). We elided this permissibility condition for brevity. Permissibility even in an unrelated state can provide useful information. In particular, the validity of the guard of the call can provide conditions on the arguments of the call that are independent of the state.

2.5 Use-cases

We now present two use-cases. Fig. 2.6.(a) represents the Auction use-case that we have adopted from CISE [189]. Users can place bids and then the auction can be closed to declare the winner. The state Σ of the object is the record of the set of current bids bs, and the option value w that is either some winning bid or none $\perp$ when the auction is still open. The integrity invariant $\mathcal{I}$ is that if the auction is closed, then the winning bid is the maximum of the non-empty set of bids. Auction offers three methods: place, close and query. While the auction is open, the method place can place a bid b. The method close closes the bid by picking the maximum bid. The method query returns the current state of the auction. It is notable that in the guard of close, we do not need to repeat the condition that the bid set should be non-empty. This condition is declared in the invariant. If a call on close violates the invariant, the call is not permissible and is aborted. In general, the user does not need to restate the invariant as guards. The guard needs to only specify the semantic preconditions of the method. Thus, our specifications are simpler than previous work [189]. As an example of semantic preconditions, the execution of a close call on an auction is meaningful only if the auction is not already closed although it does not violate the invariant. Similarly, placing a bid is meaningful only when the auction is not closed even

Class Auction

$\Sigma := \langle bs : \mathsf{Set\ Int},\ w : \mathsf{Option\ Int}\rangle$

$\mathcal{I} := \lambda\ \langle bs, w\rangle.$

$w \neq \bot \rightarrow (bs \neq \emptyset\ \wedge\ w = \mathsf{some}(\mathsf{max}(bs)))$

$\mathsf{place}(b) := \lambda\ \langle bs, w\rangle.$

$\langle w = \bot,\quad \langle bs\ \cup\ \{b\},\ w\rangle,\quad \bot\rangle$

$\mathsf{close} := \lambda\ \langle bs, w\rangle.$

$\langle w = \bot,\quad \langle bs,\ \mathsf{some}(\mathsf{max}(bs))\rangle,\quad \bot\rangle$

$\mathsf{query} := \lambda\ \sigma.\ \langle \mathbb{T},\ \sigma,\ \sigma\rangle$

(a) User Specification

Class 2PSet

$\Sigma := \langle \mathsf{Set}, \mathsf{Set}\rangle$

$\mathcal{I} := \mathbb{T}$

$\mathsf{add}(e) := \lambda\ \langle A, R\rangle.$

$\langle \mathbb{T},\quad \langle A\ \cup\ \{e\}, R\rangle,\quad \bot\rangle$

$\mathsf{remove}(e) := \lambda\ \langle A, R\rangle.$

$\langle \mathbb{T},\quad \langle A, R\ \cup\ \{e\}\rangle,\quad \bot\rangle$

$\mathsf{contains}(e) := \lambda\ \langle A, R\rangle.$

$\langle \mathbb{T},\quad \langle A, R\rangle,\quad e \in A \setminus R\rangle$

(f) User Specification

	p	c	q
p	✓	×	✓
c	×	✓	✓
q	✓	✓	✓

(b) $\mathcal{S}$-commute

	p	c	q
p	✓	×	✓
c	✓	×	✓
q	✓	✓	✓

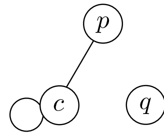
(c) $\mathcal{P}$-concur

	a	r	c
a	✓	✓	✓
r	✓	✓	✓
c	✓	✓	✓

(g) $\mathcal{S}$-commute

	a	r	c
a	✓	✓	✓
r	✓	✓	✓
c	✓	✓	✓

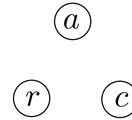
(h) $\mathcal{P}$-concur



(d) Conflict graph

	p	c	q
p	✓	✓	✓
c	×	✓	✓
q	✓	✓	✓

(e) Independent



(i) Conflict graph

	a	r	c
a	✓	✓	✓
r	✓	✓	✓
c	✓	✓	✓

(j) Independent

Figure 2.6: Auction and Two Phase Set Use-cases. The conflict graph in (d) is obtained from (b) and (c).

if the bid is less than the already decided winner which does not violate the integrity of the auction.

Fig. 2.6.(b) shows that the place and close methods $\mathcal{S}$-conflict. a call on place can execute either before or after a call on close. In the former, the close method gets to see the new bid that might be the largest. However, in the latter, the new bid is missed. Therefore, the two executions can diverge. As Fig. 2.6.(c) shows, the place and close methods $\mathcal{P}$-conflict with the close method. The methods place and close are not invariant-sufficient. Their guards require the auction to be open that is not implied by the invariant. If a call on place is pushed after a call on close, the call on place can violate the invariant as it can place a bid larger than the already decided winner. If a call on close is pushed after another call on close, its guard does not hold after the move. As Fig. 2.6.(e) shows, a call on close is dependent on a preceding call on place. The preceding place call can be placing the only bid and if it is removed, the close call gets an empty auction to close that violates the invariant.

Fig. 2.6.(f) shows the 2PSet (two-phase set) use-case that we have adopted from CRDTs [424]. Classical sets $\mathcal{S}$-conflict on adding and removing elements. The two orders do not agree on the final set. However, this is only when the two calls are on the same element. A set with a known finite domain can avoid conflicts and synchronization for unequal elements. In contrast, 2PSet avoids conflicts by changing the set semantics: once an element is removed, it cannot be added again. As Fig. 2.6.(f) shows, it uses two sets to store added and removed elements and the abstract state of the set is the added set minus the removed set. Therefore, the two orders of adding and removing an element result in the same

set: the element is considered to be removed. As Fig. 2.6.(g)-(j) shows, the methods of 2PSet concur and are independent. Thus, 2PSet methods can execute without any coordination.

2.6 Protocols

In the previous sections, we presented how the conflict and dependency relations of a given object are calculated. In this section, we present two concrete protocols that use these relations and implement the well-coordination conditions. The protocols are parametric and instantiated with the object and its conflict and dependency relations. The first protocol is non-blocking. Crash of a replica does not prevent other replicas from making progress. The second protocol is blocking. In return, it can further avoid synchronization. For a pair of conflicting methods, the protocol can push synchronization to only one of them and the other method can execute without synchronization.

In the next two subsections, we focus on synchronization of conflicting methods (and assume that methods are independent). We consider dependencies in the third subsection.

Each protocol declares the request events that it inputs and the response events that it outputs. It also declares the state that it stores at every node. It may include an initialization method that is called once at the beginning of the execution at each node. A protocol may declare and use other protocols. It defines methods for requests from the client and responses from the used protocols. A method may be guarded by a condition. Such a method accepts events only when the condition is satisfied; otherwise, the processing of the event is postponed. In the body the methods, a protocol may issue responses to its client or issue requests to the used protocols.

2.6.1 Non-blocking Synchronization Protocol

In this subsection, we present a non-blocking protocol for synchronization.

Protocol Idea. A subset of the vertices of a graph is a clique iff any of its distinct pair of vertices are adjacent. A clique is maximal if it is not a subset of a larger clique. There are known algorithms [82, 451] that list the set of maximal cliques of a graph.

The methods of a clique of the conflict graph have to all synchronize with each other. The idea is to synchronize only the methods of each maximal clique with each other to minimize synchronization. We use the total-order broadcast (TOB) protocol that employs consensus to deliver messages in the same total order to all nodes [92]. Let Cl denote the set of maximal cliques of the conflict graph. For each maximal clique $cl \in \mathsf{Cl}$, we use a TOB instance $tob(cl)$. Calls on conflicting methods in a maximal clique are broadcast to a TOB instance and delivered with the same total order to all nodes. (A single node (without a loop) is considered a clique but does not need synchronization. Thus, before calculating the maximal cliques, we remove all the single nodes without loops from the conflict graph.) As we saw in Fig. 2.3.(a) for the delete call d, a call c of a method m that is a member of multiple maximal cliques cls should be totally ordered with respect to calls of each of those cliques. We broadcast the call c to every $tob(cl)$ where $cl \in cls$ and execute c only when it is ordered and delivered by all of them. To execute calls in the delivery order from TOBs, we maintain a queue $q(cl)$ for each TOB instance $tob(cl)$. Method calls that are delivered by a TOB instance $tob(cl)$ are enqueued to its corresponding queue $q(cl)$. If cls is the set of maximal cliques containing a method m, a call on m can be executed once it appears at the head of the queues $q(cl)$ for each $cl \in cls$. The call is then dequeued from the queues and

executed. Thus, the execution order of calls at every replica is an extension of the delivery order of each of the TOBs. Therefore, calls to conflicting methods have the same execution order across replicas.

However, deadlocks can happen if the TOB instances are not properly coordinated. Consider two method calls c and c' of a method m that is a member of two cliques cl and cl'. If c and c' are simply broadcast to $tob(cl)$ and $tob(cl')$, c may precede c' in the total order of $tob(cl)$ and succeed it in the total order of $tob(cl')$. Thus, c' cannot appear at the head of the queue $q(cl)$ and waits for c and symmetrically c cannot appear at the head of the queue $q(cl')$ and waits for c'. As a result, c and c' and all later calls in $q(cl)$ and $q(cl')$ will be blocked at all replicas. To prevent deadlocks, firstly, we statically order the maximal cliques Cl and always send a message to TOB instances $tob(cl)$ in the order of their corresponding cliques cl. Secondly, we ensure that if a message is ordered before another by a TOB instance then the next TOB instance respects this order. To this end, we present and use a particular kind of total-order broadcast that respects given total orders on subsets of messages. We call it the multi-total-order broadcast (MTOB) protocol.

In the multi-total-order broadcast (MTOB) protocol, the messages are divided to multiple disjoint subsets called message classes. Each class is associated with a total order. The user broadcasts each message together with its class identifier. She should also broadcast messages of a class in the total order of that class. The protocol delivers messages in a total order that respects (i.e. is an extension of) the order of each message class.

We use MTOB as follows. We define a class as the set of calls of the methods of a clique. As mentioned above, a call is sent to the MTOB instances in a statically-determined

order. For example, in the example above, we assume that $mtob(cl)$ is before $mtob(cl')$ in the static order. Assume that c is delivered before c' by $mtob(cl)$. We broadcast c and c' in order to $mtob(cl')$ with class cl. As the order of messages in class cl is preserved in the delivery order of $mtob(cl')$, c will be delivered before c' by $mtob(cl')$ as well and the deadlock mentioned above cannot happen.

We first present the main protocol and then the multi-total-order-broadcast protocol. They use the classical reliable broadcast and consensus protocols [92].

Main Protocol. The non-blocking protocol is presented in Fig. 2.7. The requests to the protocol are $\mathsf{call}(c)$ to execute a method call c on the replicated object. In response, the protocol issues the response $\mathsf{ret}(c, v)$ to return the value v as the result of the call c or $\mathsf{aborted}(c)$ to indicate that the call c could not be executed without the violation of the invariant and is aborted. The parameter to the protocol is the map $\mathsf{cliques}$. It maps each method in the set of methods M to a list of maximal cliques Cl that the method belongs to. As explained earlier, the set of maximal cliques is calculated and statically sorted to a total order. To prevent deadlocks, every list in the range of $\mathsf{cliques}$ is consistent with this total order. A call on a method m is sent to the TOB instances of the cliques $\mathsf{cliques}(m)$ in order. The protocol uses two protocols: reliable broadcast rb and a multi-total-order broadcast per clique $mtob$. Among other properties, the reliable broadcast guarantees that if a message is delivered by a correct node, then it is eventually delivered by every correct node. In addition to this guarantee, as previously mentioned, MTOB protocol guarantees that messages are delivered in a total order that is an extension of the order of each message class. Each replica

stores the following state: the state σ of the user-defined object, and the queues q, one per maximal clique.

On the invocation of the request $\mathsf{call}(c)$ to execute the call c (at R_0), the protocol finds the method m of c (at R_1) and the set of cliques cls that m belongs to (at R_2). If the set of cliques is empty, no synchronization is needed and the request is sent using the reliable broadcast rb (at R_3-R_4). Otherwise, the request is sent using the MTOB instance $mtob(cl)$ of the first clique cl in cls. As this is the first broadcast, the call can be arbitrarily ordered and no class ($\perp$) is passed as the class (at R_5-R_7). When a call c is delivered by the reliable broadcast rb (at N_0), as no further synchronization is required, it is executed (at N_1). When a call c is delivered by an MTOB instance $mtob(cl)$ (at I_1), we enqueue it to the corresponding queue $q(cl)$ (at I_2), and get the list of cliques cls of the method (at I_3-I_4). If the current clique is the last one in the list (at I_5), we check if the call can be executed (at I_6). Otherwise, we send c to the next MTOB instance $mtob(cl')$. The call is broadcast together with the previous clique cl as the class (at I_7-I_9). A call is ready to be executed if it appears at the head of all the queues of the cliques that the method belongs to (at C_0-C_3). A call c that is ready is dequeued from the queues (at C_4) and executed (at C_5). Then, the queues are checked for next calls that might be ready to execute (at C_6). To check the queues (at Q_1), the call at the head of every queue is checked. Checking is repeated if a call is executed (at Q_2-Q_4 and C_5-C_9). To execute a call (at E_1), it is checked that it is locally permissible i.e. its guard is satisfied and applying it does not violate the invariant (at E_2). If the check is passed, the updated state is stored, the return value v is calculated (at E_3-E_4), and a return response is issued with v (at E_5). Otherwise, an abort response is issued (at

```
     NonBlockingRepObject
       request: call(C)
       response: ret(C, V) | aborted(C)
     Params: cliques: M → List[Cl]
     Using:
       rb: ReliableBroadcast
       mtob: Cl → MultiTotalOrderBroadcast
     State:
       σ: Σ = σ0
       q: Cl → Queue[C] = Cl ↦ ∅
R0   request (call(c))
R1     m ← method(c)
R2     cls ← cliques(m)
R3     if (cls = ∅)
R4       issue request (rb, broadcast(c))
R5     else
R6       cl ← head(cls)
R7       issue request (mtob(cl), broadcast(c, ⊥))
N0   response (rb, deliver(c)) exec(c)
I1   response (mtob(cl), deliver(c))
I2     enq(q(cl), c)
I3     m ← method(c)
I4     cls ← cliques(m)
I5     if (cl = last(cls)) check(c)
I7     else
I8       cl′ ← (cls, cl)
I9       issue request (mtob(cl′), broadcast(c, cl))
C0   fun check(c)
C1     m ← method(c)
C2     cls ← cliques(m)
C3     if (forall cl ∈ cls. head(q(cl)) = c)
C4       foreach(cl ∈ cls) q(cl).deq()
C5       exec(c)
C6       checkQs()
C7       return true
C8     else  return false
Q1   fun checkQs()
Q2     foreach(cl ∈ Cl, q(cl) ≠ ∅)
Q3       c ← head(q(cl))
Q4       if (check(c)) return
E1   fun exec(c)
E2     if (guard(c)(σ) ∧ I(update(c)(σ)))
E3       σ ← update(c)(σ)
E4       v ← retv(c)(σ)
E5       issue response ret(c, v)
E6     else  issue response aborted(c)
```

Figure 2.7: Non-blocking Synchronization Protocol. C and V are call and return value respectively

E_6-E_7). As pairs of conflicting methods are synchronized and methods are independent, a call is permissible in one replica if and only if it is permissible in another.

The protocol is non-blocking: if a quorum (majority) of nodes are correct (not faulty), every request for a call with eventually get a response. The call is first broadcast to the *rb* or an *mtob*. Both will eventually deliver the call. (We will show this property for MTOB with a quorum of correct nodes.) In the former case, the call is executed on arrival. In the latter case, it is put in the corresponding queue and may be broadcast to the next *mtob*. As we explained above, each MTOB preserves the delivery order of the previous MTOBs; thus, two calls can appear in two queues only in the same order and cannot cause a deadlock. Calls eventually arrive at the head of the queues, are dequeued and executed.

Multi-Total-Order Broadcast. The multi-total-order broadcast (MTOB) protocol is presented in Fig. 2.8. The protocol accepts requests to broadcast a message m given its class c. (A message can belong to no class $\perp$. These messages are assumed to be unique.) MTOB delivers messages to every node with the same order and this order respects the order of all message classes. The idea is to have rounds of consensus to agree on the messages to deliver. In each round, nodes propose their current messages for consensus. When the consensus protocol issues the decision response with a set of messages, they are locally sorted using a deterministic sort algorithm and delivered. To respect the order of message classes, a message is proposed only if all the messages before it in the class are already delivered. Starvation of a node and its messages in the case that its proposal is repeatedly not chosen is prevented as follows. MTOB uses a reliable broadcast protocol. Upon a broadcast request

MultiTotalOrderBroadcast

  request: broadcast(M, C)

  response: deliver(M)

Using:

  rb: ReliableBroadcast

  cs: $R \rightarrow$ Consensus

State:

  p: Set[M × C × Int] = $\emptyset$ Pending

  d: Set[M × C × Int] = $\emptyset$ Delivered

  r: Int = 0 Round

  $rank$: $C \rightarrow$ Int = $C \mapsto 0$ Rank

I_0 init()

I_1 issue request $(cs(0), \text{propose}(\emptyset))$

R_0 request (broadcast(m, c))

R_1 if $(c \neq \bot)$

R_2 $rank(c) \leftarrow rank(c) + 1$

R_3 issue request $(rb, \text{broadcast}(m, c, rank(c)))$

R_4 else

R_5 issue request $(rb, \text{broadcast}(m, \bot, 0))$

D_0 response $(rb, \text{deliver}(m, c, i))$

D_1 if $((m, c, i) \notin d)$

D_2 $p \leftarrow p \cup \{(m, c, i)\}$

C_0 response $(cs(r'), \text{decide}(d'))$ if $(r' = r)$

C_1 foreach$((m, c, i) \in \text{sort}(d'))$

C_2 issue response deliver(m)

C_3 $d \leftarrow d \cup d'$

C_4 $p \leftarrow p \setminus d'$

C_5 $r \leftarrow r + 1$

C_6 issue request $(cs(r), \text{propose}(\text{proposal}()))$

P_0 fun proposal()

P_1 $\{(m, c, i) \mid (m, c, i) \in p \wedge$

P_2 $\forall i'.\ 0 < i' < i \rightarrow \exists m'.\ (m', c, i') \in d\}$

Figure 2.8: Multi-Total-Order Broadcast Protocol. M and C are message and class types respectively.

for a message, it is first broadcast with the reliable broadcast protocol to other nodes. Thus, the message will be in the proposal of other nodes and will be eventually chosen.

MTOB uses the reliable broadcast protocol rb and an instance sequence of the consensus protocol cs. MTOB proceeds in rounds R and uses an instance of consensus in each round. It stores the set of pending messages p, the set of delivered messages d, the

number of the current round r, and the rank of the last delivered message for each class $rank$.

The rounds of consensus are kick-started in the initialization function (at I_0-I_1). Upon an MTOB request to broadcast a message, if it belongs to a class (at R_1), the rank for the class is incremented (at R_2) and it is broadcast using the reliable broadcast rb (at R_3). Otherwise, the message is broadcast with no class and zero rank (at R_4). When rb delivers a message (at D_0), if it is not already delivered (at D_1), it is added to the pending set (at D_2). Once the decision of the current round is received (at C_0), its messages are sorted and delivered (at C_1-C_2) and added to the delivered set and removed from the pending set (at C_3-C_4). Then, the node enters the next round and proposes in it (at C_5-C_6). The proposal is the largest subset of the pending messages m such that all the messages before m in its class are already delivered (at P_0-P_2). This condition ensures that the order of each class is preserved. It is notable that as the messages with no class are added to the pending set with rank 0, they always satisfy the proposal condition.

It is assumed that a quorum (majority) of nodes are correct. Let us explain why a message broadcast by a correct node is eventually delivered to every correct node. We consider a message of a class and by induction assume that previous messages of the class are eventually delivered. If the message has been and will be in the decided set of a round, it is or will be eventually delivered by all correct nodes. Otherwise, we assume that it is never in a decided set and thus never in a delivered set. The message is first broadcast using rb. Thus, it is eventually delivered by rb and as it is not in the delivered sets, it will be added to the pending set of all correct nodes. As the previous messages in the class are eventually

delivered, the message will eventually be in the proposed set of all correct nodes. With a quorum of correct nodes, the consensus protocol guarantees eventual decision. Thus, the message will eventually be in the decided set and delivered.

2.6.2 Blocking Synchronization Protocol

The previous protocol requires both calls of a conflicting pair to participate in synchronization. In this section, we introduce a blocking protocol. As we saw in Fig. 2.3.(b) for the add a and enroll e calls, this protocol can make one of the two calls execute without synchronization.

Protocol Idea. Consider two conflicting methods m and m', and two calls c on m and c' on m'. To let the call c execute without synchronization, the other call c' needs to reach out to other nodes, block the execution of calls on m at those nodes and then propagate previous calls on m from every node to other nodes. Then, c' can be executed at all nodes. At the end, the execution of calls on m is unblocked at all nodes. Therefore, the set of calls on m before each call on m' is the same across nodes. This means that the order of every pair of calls on m and m' is the same across nodes.

A vertex cover V' of a graph $\langle V, E \rangle$ is a subset of the vertices V such that every edge in E has at least one endpoint in V'. A minimum vertex cover of a graph is a vertex cover of the smallest size. In a graph with weighted vertices, the weighted minimum vertex cover is a vertex cover of the smallest weight sum. Finding the (weighted) minimum vertex cover is a classical graph problem.

$$
\begin{array}{ll}
 & \mathsf{BlockingRepObject} \\
 & \quad \mathsf{request}: \mathsf{call(C)} \\
 & \quad \mathsf{response}: \mathsf{ret(C,V)} \mid \mathsf{aborted(C)} \\
 & \mathsf{Params}: \\
 & \quad \mathsf{conflict}: \mathsf{M} \to \mathsf{Set[M]} \\
 & \quad \mathsf{cover}: \mathsf{Set[M]} \\
 & \mathsf{Using}: \\
 & \quad rb: \mathsf{ReliableBroadcast} \\
 & \quad tob: \mathsf{M} \to \mathsf{TotalOrderBroadcast} \\
 & \mathsf{State}: \\
 & \quad \sigma: \Sigma = \sigma_0 \\
 & \quad b: \mathsf{M} \to \mathsf{Int} = \mathsf{M} \mapsto 0 \\
 & \quad xed: \mathsf{M} \to \mathsf{Set[C]} = \mathsf{M} \mapsto \emptyset \\
 & \quad act: \mathsf{M} \to \mathbb{B} = \mathsf{M} \mapsto \mathsf{false} \\
 & \quad cnt: \mathsf{C} \to \mathsf{Int} = \mathsf{C} \mapsto 0 \\
R_0 & \mathsf{request}\ (\mathsf{call}(c)) \\
R_1 & \quad m \leftarrow \mathsf{method}(c) \\
R_2 & \quad \mathsf{if}\ (m \notin \mathsf{cover}) \\
R_3 & \quad\quad \mathsf{issue\ request}\ (rb, \mathsf{broadcast}(\mathsf{nsync}(c))) \\
R_4 & \quad \mathsf{else} \\
R_5 & \quad\quad \mathsf{if}\ (m \in \mathsf{conflict}(m)) \\
R_6 & \quad\quad\quad \mathsf{issue\ request}\ (tob(m), \mathsf{broadcast}(\mathsf{sync}(c))) \\
R_7 & \quad\quad \mathsf{else}\ \ \mathsf{issue\ request}\ (rb, \mathsf{broadcast}(\mathsf{sync}(c))) \\
N_0 & \mathsf{response}\ (rb, \mathsf{deliver}(\mathsf{nsync}(c)))\ \mathsf{if}\ b(\mathsf{method}(c)) = 0 \\
N_1 & \quad \mathsf{exec}(c) \\
C_0 & \mathsf{response}\ (tob(m), \mathsf{deliver}(\mathsf{sync}(c)))\ \mathsf{if}\ \neg act(m) \\
C_1 & \quad act(m) \leftarrow \mathsf{true} \\
C_2 & \quad \mathsf{blockAndUpdate}(c) \\
C_3 & \mathsf{response}\ (rb, \mathsf{deliver}(\mathsf{sync}(c))) \\
C_4 & \quad \mathsf{blockAndUpdate}(c) \\
B_1 & \mathsf{fun\ blockAndUpdate}(c) \\
B_2 & \quad \mathsf{foreach}(m' \in \mathsf{conflict}(m)) \\
B_3 & \quad\quad b(m') \leftarrow b(m') + 1 \\
B_4 & \quad cs \leftarrow xed \mid \mathsf{conflict}(m) \\
B_5 & \quad \mathsf{issue\ request}\ (rb, \mathsf{broadcast}(\mathsf{update}(c, cs))) \\
U_0 & \mathsf{response}\ (rb, \mathsf{deliver}(\mathsf{update}(c, cs))) \\
U_1 & \quad \mathsf{foreach}(c' \in cs)\ \ \mathsf{exec}(c') \\
U_2 & \quad cnt(c) \leftarrow cnt(c) + 1 \\
U_3 & \quad \mathsf{if}\ (cnt(c) = \mathcal{N}) \\
U_4 & \quad\quad \mathsf{exec}(c) \\
U_5 & \quad\quad \mathsf{foreach}(m' \in \mathsf{conflict}(m)) \\
U_6 & \quad\quad\quad b(m') \leftarrow b(m') - 1 \\
U_7 & \quad\quad act(m) \leftarrow \mathsf{false} \\
E_0 & \mathsf{fun\ exec}(c) \\
E_1 & \quad \mathsf{if}\ (\mathsf{guard}(c)(\sigma) \wedge \mathcal{I}(\mathsf{update}(c)(\sigma))) \\
E_2 & \quad\quad \sigma \leftarrow \mathsf{update}(c)(\sigma) \\
E_3 & \quad\quad v \leftarrow \mathsf{retv}(c)(\sigma) \\
E_4 & \quad\quad \mathsf{issue\ response\ ret}(c, v) \\
E_5 & \quad\quad \mathsf{add}(xed(m), c) \\
E_6 & \quad \mathsf{else}\ \ \mathsf{issue\ response\ aborted}(c)
\end{array}
$$

Figure 2.9: Blocking Synchronization Protocol

In the interest of avoiding synchronization, we find the minimum vertex cover of the conflict graph. Only the methods in the cover synchronize and the rest can execute without synchronization. To execute a method in the cover, the requesting node has to reach out to all nodes and block and solicit the conflicting methods. If the user calls a method more often than others or favors its responsiveness, she can assign a lower weight to that method and apply the weighted minimum vertex cover. Methods can be assigned weights inversely proportional to their call frequency. To enforce that a method becomes synchronization-free, its weight can be assigned to infinity.

Protocol. The blocking synchronisation protocol is presented in Fig. 2.9. It accepts requests to execute calls and in return issues responses with the return value or that the call is aborted. The parameters to the protocol are the map conflict that maps every method to its set of conflicting methods and a vertex cover of the conflict graph cover. The protocol uses two classical protocols: the reliable broadcast rb and a total-order broadcast per method tob. The protocol stores the following state at each node: the user-defined state of the object σ, a mapping b from each method to the number of times that it is blocked, a mapping xed from each method to the set of executed calls on that method, a mapping act from each method to whether there is an active execution of a call on the method, a mapping cnt from each call to the number of messages received for it.

Upon a request to execute a method call c (at R_0), if its method m is not a member of the cover (at R_1-R_2), it can be executed without synchronization. So, it is broadcast using rb as a non-synchronizing nsync call (at R_3). Otherwise, the call should synchronize with conflicting methods (at R_4) and it is broadcast as a synchronizing sync call. If m has a

self-loop in the conflict graph, then c should synchronize with other calls on m. To order calls on m, they are broadcast using the total-order-broadcast $tob(m)$ (at R_5-R_6). If m does not have a self-loop, c only needs to synchronize with calls on other methods. Thus, c is broadcast using the reliable broadcast rb.

Upon receiving a non-synchronizing call that is not blocked (at N_0), it is executed (at N_1). A call on a blocked method should wait until it is unblocked. When a synchronizing call c on a method m is received from a total-order broadcast $tob(m)$, if the execution of another call on m is not active (at C_0), it is recorded that the execution of a call on m is active (at C_1). On the other hand, when a synchronizing call is received from the reliable broadcast rb (at C_3), it does not need to prevent other calls on m as m does not conflict with itself. In both cases (at C_2 and C_4), each method that conflicts with m is blocked (at B_2), and the calls on the conflicting methods that this node has executed are broadcast as an update to other nodes (at B_3-B_5). When an update arrives (at U_0), its calls are executed (at U_1) and the number of received updates for the call is incremented (at U_2). When an update from all nodes is received (at U_3), the call is executed (at U_4), the previously blocked methods for c are unblocked (at $U_5 and U_6$), and it is recorded that the execution of a call on m is no longer active (at U_7). The execution of a call (at E_0-E_7) is similar to the previous protocol.

As mentioned earlier, this protocol brings more synchronization-freedom. However, either progress or consistency and convergence of nodes may be affected by crashes. Blocked operations are only unblocked when update messages are received from all the other nodes. If the update message from a node is not received, calls on the blocked methods cannot be

executed. Either the network is slow or that node has crashed. If other nodes assume the former, the latter may be the case and they can never execute the blocked methods. On the other hand, if other nodes assume that the node has crashed, the network may be just slow. In particular, if a correct node n is mistakenly suspected while a synchronizing call c is being executed, consider that other nodes refrain from waiting for n, execute c and unblock conflicting methods before n blocks conflicting methods. Then, a node n' can execute a call c' on a conflicting method. The call c' can reach and execute at the suspected node n before it executes c. Thus, c' is after c at n' but before it at n. Therefore, the two conflicting method calls have different orders in different nodes. Further, c can become impermissible after c'. Thus, this can cause divergence and violation of integrity at node n.

2.6.3 Dependency-Tacking Protocol

In the presented synchronization protocols, we assumed that method calls were independent. However, as we saw in Fig. 2.2.(e), permissibility of a call at a node may be dependent on the preceding calls at that node; the call may not be permissible at other nodes.

We saw that method calls may or may not need to synchronize before execution. If a call did not need synchronization, it was simply broadcast and was immediately executed on arrival. For both the non-blocking protocol (Fig. 2.7) and the blocking protocol (Fig. 2.9) this was at N_1. However, if it has dependencies, they should be tracked at the originating node and broadcast together with the call. The receiving nodes should apply the call only after its dependencies are applied. On the other hand, some calls go through synchronization before execution. When synchronization is finished for a call c, it may or may not be

permissible in different nodes. For the non-blocking protocol (Fig. 2.7), this is at C_5 and for the blocking protocol (Fig. 2.9), this is at U_4. If there is a node n that finds c permissible, every node can become permissible for c after n propagates the dependencies. The call c is aborted only if it is impermissible at every node. We use a protocol that is the inverse of the classical atomic commit protocol. The decision is abort if every replica votes for abort and is commit otherwise. Every node that finds c permissible votes for commit together with the dependencies of c and every node that finds it impermissible votes for abort. If a node receives the abort decision, it aborts the execution of c. If a node receives the commit decision, it waits for the dependencies. After the dependencies are applied, the call c is permissible and is executed.

2.7 Implementation

In this section, we describe the implementation of our synthesis tool, Hamsaz. The input to Hamsaz is the definition of an object that includes the state type and invariants on the state along with methods. Hamsaz synthesizes non-blocking and blocking replicated objects. It also outputs the baseline sequentially consistent replicated object. Hamsaz consists of two main parts: (1) determining the conflicts and dependencies and (2) instantiating the protocols.

Conflict and Dependency Analysis. We use the CVC4 [52] SMT solver [53] to decide the validity of concur and independence relations for pairs of methods. In particular, we use the theory of linear arithmetic, inductive datatypes, and more importantly, the theory

of finite sets [48] and the follow-up theory of finite relations [343] that is recently added to CVC4. Decidable fragments of set theory [95] is an active area of research [96, 272, 443].

To decide the validity of a condition, Hamsaz may decompose the invariant to conjuncts. As an example, consider whether $\mathsf{enroll}(s_1, c_1)$ $\mathcal{P}$-concurs with $\mathsf{enroll}(s_2, c_2)$ in the Courseware use-case presented in Fig. 2.1. We focus on the invariant $\mathsf{refIntegrity}(es, \mathsf{esid}, ss, \mathsf{sid})$; the other invariant is similar. The invariant is unrolled to $\forall e.\ e \in es \rightarrow \exists s.\ s \in ss \wedge \mathsf{esid}(e) = \mathsf{sid}(s)$. We decompose it to the following two conjuncts based on whether the referential integrity involves the enrolled student s_1: (1) $\forall e.\ e \in es \wedge \mathsf{esid}(e) = s_1 \rightarrow \exists s.\ s \in ss \wedge \mathsf{esid}(e) = \mathsf{sid}(s)$, (2) $\forall e.\ e \in es \wedge \mathsf{esid}(e) \neq s_1 \rightarrow \exists s.\ s \in ss \wedge \mathsf{esid}(e) = \mathsf{sid}(s)$. For the first one, the call $\mathsf{enroll}(s_1, c_1)$ $\mathcal{P}$-R-commutes with the call $\mathsf{enroll}(s_2, c_2)$. For the second one, the call $\mathsf{enroll}(s_1, c_1)$ is invariant-sufficient.

Protocols. We implemented the parametric protocols presented in § 2.6. Given the analysis results, we apply the graph optimizations and then instantiate the protocols with the optimization results. We implemented our protocols on top of APPIA [97], the accompanying toolkit of [92]. It is a Java library of basic communication abstractions. We implemented our protocols on top of the basic broadcast, total-order broadcast and consensus protocols. We also implemented the sequentially consistent baseline. It uses a total-order broadcast instance to deliver calls to all nodes in the same order.

2.8 Evaluation

We applied Hamsaz to a suite of use-cases to synthesize non-blocking and blocking replicated objects and compared their performance with the sequentially consistent baseline.

Use-cases. The use-cases are the following: $\mathsf{Counter}$: It can increment and decrement an integer value. $\mathsf{NNCounter}$: The non-negative counter has the invariant that the counter value should be non-negative. $\mathsf{Register}$: A register stores a value and provides methods to read and write it. $\mathsf{BankAccount}$: The invariant is a non-negative balance. CSet: The classical set provides add, remove and contains methods. GSet: The grow-only set (adopted from [424]) provides adding (but not removing) an element contains methods. Both methods can execute without coordination. FDSet: A finite-domain set provides the classical set operations on a predefined finite set of elements. Thus, it can avoid coordination between calls on different elements. $\mathsf{2PSet}$ (two-phase set) (adopted from [424]) and $\mathsf{Auction}$ (adopted from [189]) that we saw in Fig. 2.6. The suite includes relational use-cases as well. Relational

$$\mathsf{unique}(R, f) := \forall r, r'.\ \ r \in R \wedge r' \in R \wedge f(r) = f(r') \rightarrow r = r'$$

$$\mathsf{refIntegrity}(R, f, R', f') := \forall r.\ \ r \in R \rightarrow \exists r'.\ r' \in R' \wedge f(r) = f'(r')$$

$$\mathsf{rowIntegrity}(R, p) := \forall r.\ \ r \in R \rightarrow p(r)$$

Figure 2.10: Relational Integrity Constrains

integrity properties are specified using three predicates that we present in Fig. 2.10. The property $\mathsf{unique}(R, f)$ states that the values of the field f in the records of the relation R are unique. The property $\mathsf{refIntegrity}(R, f, R', f')$ states that for every record r in R, there exists a record r' in R' such that the field f of r is equal to the field f' of r'. The property $\mathsf{rowIntegrity}(R, p)$ states that every record of the relation R satisfies the predicate p. The relational uses cases are the following. $\mathsf{Courseware}$: We saw the courseware use-case (adopted from [189]) in Fig. 2.1. It requires referential integrity for the student and course identifiers.

2PCourseware: It uses 2PSet to reduce conflicts in Courseware. Payroll: The payroll use-case (adopted from [35]) stores employee and department relations. It requires uniqueness of employee identifiers, referential integrity for the department identifiers of employees, non-null values for employee names and non-negative salaries. It supports adding and removing employees and departments, and increasing and decreasing employee salaries. Tournament: The tournament use-case (adopted from [45]) stores players, tournaments, and enrolments. It requires uniqueness of player and tournament identifiers, referential integrity of player and tournament identifiers in enrolments, and that each player has a positive budget, each tournament has a size within a cap, and each active tournament has at least one player. It supports adding and removing players and tournaments, adding funds for a player, enrolling and disenrolling a player in a tournament, and beginning and ending a tournament.

Platform. The experiments are done on a cluster with 4 computing nodes. Each node has 2 AMD Opteron 6272 CPUs with a total 8 cores with 64GB ECC protected memory of RAM and a 40Gbps high-bandwidth low-latency InfiniBand network. The OS running on the cluster is CentOS 7.4 Linux x86_64 with the kernel version 3.10.0-862.3.2.el7. JDK is openjdk version 1.8.0_171 (OpenJDK 64-Bit Server VM build 25.171-b10, mixed mode). All nodes are connected to a Mellanox 18 port InfiniBand switch. Reported numbers are the arithmetic means of results from five repetitions.

Conflict and Dependency Analysis. The concur and independence conditions for Counter, NNCounter, Register and BankAccount use-cases all fall in the quantifier-free fragment of the theory of linear arithmetic. The conditions for CSet, GSet, FDSet and 2PSet all fall in the quantifier-free fragment of the theory of sets. However, the Auction

Usecase	$\#\mathsf{M}^1$	$\#\mathcal{I}^2$	$\mathcal{P}^3$	$\mathcal{S}^4$	Indep	Total
Bank	3	1	284	695	595	1574
Auction	3	2	405	921	571	1897
Courseware	5	4	950	3256	2597	6803
NNCounter	3	1	283	598	470	1351
Tournament	9	5	3482	25615	24146	53603

[1] The number of methods

[2] The number of invariants

[3] $\mathcal{P}$-concure time (ms)

[4] $\mathcal{S}$-commute time (ms)

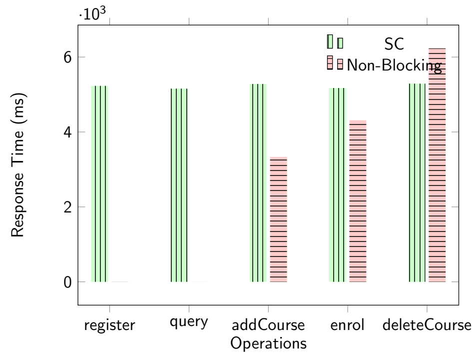
Figure 2.11: Analysis time

use-case uses the max function. We specified the following two axioms for max and CVC4 could use them to decide the validity of the conditions. $\mathcal{A}_1 : \forall s, i.\ \ i \in s \rightarrow \mathsf{max}(s) \geq i$ and $\mathcal{A}_2 : \forall s.\ \ s \neq \emptyset \rightarrow \mathsf{max}(s) \in s$. The integrity properties of the relational uses-cases are encoded using quantifiers as presented in Fig. 2.10. The reason is that the current theory of sets in CVC4 does not support a complete set of relational operators. A set of operators is called complete if any relational algebra expression can be expressed by a combination of them. Selection (σ), projection (π), renaming (ρ), union ($\cup$), difference ($\setminus$) and product ($\times$) are a complete set of operators. For example, a referential integrity $\mathsf{refIntegrity}(R, a, R', a')$ can be written as $\pi_a R \setminus \pi_{a'} R' = \emptyset$ using projection and difference and as $\mathsf{Car}(R \bowtie_{a=a'} R') \geq \mathsf{Car}(R)$ using join and cardinality. CVC4 supports difference and join but not projection and cardinality is a planned feature [343]. Despite using quantifiers, CVC4 can decide the validity of all conditions in our relational use-cases in less than a minute. We

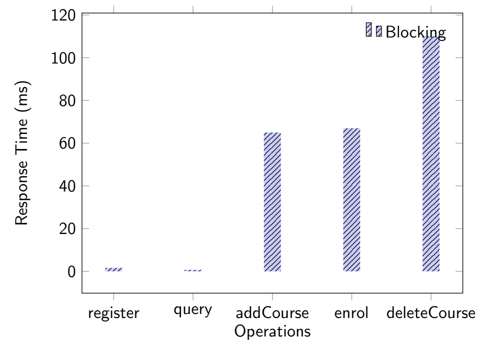
measured the time that Hamsaz takes to calculate the conditions and represent the results in Fig. 2.11. For each use-case, the table lists the number of methods, the number of invariants, the time to calculate $\mathcal{P}$-concur and $\mathcal{S}$-commute for the conflict relation, the time to calculate the independence relation and the total time.

Results. In this section, we compare the response time of our protocols with each other and the sequentially consistent (SC) baseline. The response time for a call is the time spent between the request and the response of the call. We conduct two experiments on the courseware use-case that we saw in Fig. 2.1 and the bank account use-case. In the bank account use-case, the withdraw method conflicts with itself and is dependent on deposit. The deposit and balance methods are conflict-free and independent. In the first experiment, we compare the response time of methods using different protocols. In the second one, we measure the effect of increasing the workload on the response time. In both experiments, we execute 500 calls evenly distributed on the methods.

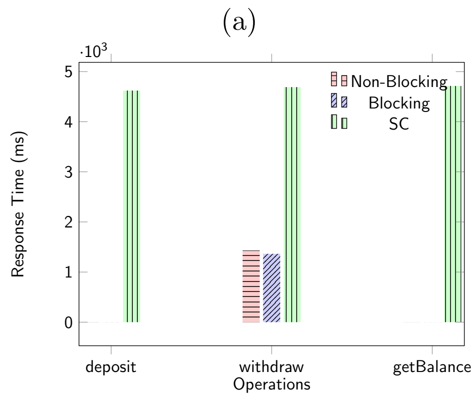
In the first experiment, we issue one call per millisecond and measure the average response time of the calls on each method. The results for the non-blocking and the SC protocols are shown in Fig. 2.12.(a) and for the blocking protocol are shown in Fig. 2.12.(b). We make this separation because the latter is two orders of magnitude more responsive than the former. The response time for SC is the same across methods as all methods use the same TOB instance. In the non-blocking protocol, the response time of the register and query methods is around a millisecond. The response time of these two methods is significantly less than that of the other methods because they can execute without coordination. The response time of the deleteCourse method is about two times that of addCourse and enroll methods



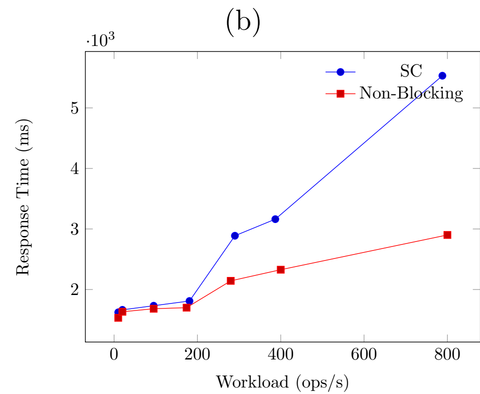
(a)



(b)



(c)



(d)

Figure 2.12: (a) Response time for Courseware with the Non-Blocking and SC protocols. (b) Response time for Courseware with the Blocking protocol. (c) Response time for BankAccount. (d) The effect of workload on response time for Courseware.

because a deleteCourse call has to be ordered by two TOB instances while an addCourse or enroll call needs to be ordered by only one TOB instance. The enroll method is less responsive than the addCourse method because enroll has dependencies and needs to wait for them and addCourse does not. Both the SC and non-blocking protocols synchronize by TOB instances that rely on consensus. On the other hand, the blocking protocol avoids using TOB for

the courseware use-case. Fig. 2.12.(b) shows that this avoidance significantly improves the response time. The two methods register and query execute without coordination. Calls on the deleteCourse method coordinate to block addCourse and enroll methods. Therefore, deleteCourse is less responsive than the other methods. Calls on addCourse and enroll methods may be blocked but when they are not, they execute without coordination.

We applied the first experiment to the bank account use-case as well. The results are shown in Fig. 2.12.(c). Similar to the courseware use-case, the SC protocol is the least responsive and uniform across all methods. In the other two protocols, the deposit and balance methods can execute without coordination in around a millisecond response time. Interestedly, the withdraw method exhibits almost the same response time in the blocking and non-blocking protocols. The reason is that withdraw conflicts with itself, and the blocking protocol uses a TOB to order withdraw calls (at R_6 in Fig. 2.9). Thus, both the blocking and non-blocking protocols use TOB for the bank account use-case and the synchronization by the TOB dominates the execution time.

In the second experiment, we increase the workload from 10 to 800 calls per second and measure the average response time over all the calls. The results for the non-blocking and the SC protocols on the courseware use-case are shown in Fig. 2.12.(d). Similar to the first experiment, we make this separation because the latter is orders of magnitude more responsive than the former. As we increase the workload, the network transmits more messages and the protocol states grow that in turn affect the responsiveness. All the protocols get less responsive as the workload increases. The response time of the SC protocol grows faster as every operation goes through synchronization.

We observe that coordination specially synchronization can adversely affect the response time. The experiments suggest that our protocols can effectively avoid coordination to reduce the response time. They exhibit considerable improvement over the SC protocol. In particular, the blocking protocol is more responsive than the other protocols especially when no method conflicts with itself. However, as mentioned before, the blocking protocol may not progress in case of node crashes. On the other hand, the non-blocking protocol is less responsive but maintains progress.

2.9 Related Works

$\mathcal{I}$-confluence [35] is a sufficient condition for invariant preservation of state-based replicated objects [424]. It states that if user operations and the merge operation are invariant preserving, then every execution is invariant preserving. In contrast to $\mathcal{I}$-confluence, well-coordination is a correctness condition for operation-based replicated objects [424]. Further, in addition to coordination-avoiding operations, it supports and reduces coordination for conflicting operations. A follow-up work, Blazes [24], applies a technique called sealing to replicated stream processing. It calculates deterministic results in the presence of non-deterministic reordering of messages. The idea is to split messages to windows and apply aggregate operations on them. Both the technique and its applications are distinct from ours.

Warranties [309] delay update operations for a limited time to preserve a state assertion on the distributed state. Thus, local computations can count on the assertion without coordination. However, in contrast to our approach, warranties are not automatically

inferred and specifically improve the efficiency of read-dominated applications. Further, they preserve strong consistency rather than exploiting weak consistency. Homeostasis [409] targets invariants that span nodes of partially-replicated distributed stores. Each node maintains a condition called treaty on its local state and relies on the validity of other node treaties. The idea is that a change in the state of a node may preserve its treaty and not observationally change the execution of transactions in other nodes. Thus, coordination can be avoided. However, if a node violates its treaty, it synchronizes with other nodes and a new set of treaties are calculated and installed. In contrast, well-coordination targets fully-replicated stores, exploits weak consistency and guarantees convergence. Further, the analysis is static and the protocols do not calculate conditions at runtime.

Sieve [301, 299] defines a consistency model called RedBlue and applies static and dynamic analysis to determine whether an operation can be executed under causal consistency (blue) or needs strong consistency (red) to preserve the invariant. However, the analysis does not check that the result will indeed validate the invariant. In contrast, we prove the sufficiency of well-coordination. Further, causal consistency is the weakest possible notion in the RedBlue model while our model allows operations to execute with no synchronization and dependency.

Quelea [431] lets the programmer declare consistency contracts for operations of a replicated object using primitive consistency relations such as visibility and session orders. It defines axiomatic semantics for consistency notions using the same primitives. It then automatically maps a contract to the weakest consistency notion that satisfies the contract. However, these contracts are lower-level than integrity invariants and translating

invariants to contracts is non-trivial. Inspired by weak memory models, a similar work [103, 59] presented a framework for specification of weak consistency models that have atomic visibility and defined dynamic and static checks for serializability of applications that choose to use weak consistency. Later, [84] defined a generalization of conflict serializability to be used together with weak consistency notions. It presented a dynamic checker to determine whether an execution of an application that uses weak consistency is serializable. In contrast, our approach requires applications to specify only higher-level integrity properties and automatically finds the coordination needed to preserve them.

Indigo [45, 46] lets the user introduce application-specific predicates and define invariants and method post-conditions in terms of those predicates. It identifies operations that violate the invariant if executed concurrently and either prevents or repairs their concurrent execution. For the former, it applies reservation techniques to enhance coordination efficiency and for the latter, it provides a library of restoring operations. In contrast, well-coordination does not require user-defined predicates and annotations. In addition, the well-coordination conditions are formally defined and their sufficiency is proved. Further, well-coordination guarantees convergence in addition to invariant preservation. Besides, the implementation of Indigo is dependent on causal consistency of a lower-level store while we defined and implemented standalone protocols.

CISE [189, 356] lets the user specify the invariant of the object, associate each method with tags and define conflicts between tags. It presents a rely-guarantee style proof technique for invariant preservation. The proof technique allows conditions to be associated with tags and requires that each operation guarantees the conditions of its tags relying on

the conditions of non-conflicting tags. In contrast to well-coordination, the proof rule is fundamentally dependent on causal consistency and hence causal consistency is the weakest possible notion in the model. Further, our approach does not require the user to provide the conflict relation and automatically calculates it. In addition, we present protocols that provide the required coordination.

For database transactions, [323] presented correctness conditions of different isolation levels and an algorithm to find the lowest isolation level for transactions of an application to be semantically correct. Later, [159, 160] presented an algorithm to determine whether executions of a transaction are serializable under snapshot isolation. Recently, AloneTogether [247] presented a program logic that enables compositional verification of invariant preservation for weakly isolated transactions. These works consider isolation levels on shared memory [367, 241, 27, 361, 358, 223, 274] databases. For instance, in the AloneTogether model, all updates of a transaction become visible to all threads in an indivisible step. In contrast, we consider weak consistency for replicated state.

IPA [224] presents a type system ensuring that values from weakly consistent operations cannot flow into strongly consistency operations without explicit user endorsement. IPA stores adapt consistency to system load within the user-specified bound. Similarly, MixT [344] is a language that allows transactions to access different stores with varying consistency guarantees and applies information flow analysis to prevent less-consistent data from influencing more-consistent data. In contrast to our approach that infers the required synchronization and dependency, IPA and MixT require the user to explicitly associate

consistency conditions with objects and stores. Further, they are concerned with consistency flow rather than integrity preservation.

Epsilon-serializability [396] and TACT [489] reduce coordination by bounding the staleness of replicated state. PBS [41] reduces coordination to a partial rather than a complete quorum and statistically bounds staleness. In contrast to bounding staleness, we focus on preserving invariants efficiently. Rationing [266] and Pileus [446] dynamically adjust consistency based on temporal load statistics and Correctables [193] incrementally makes the result more consistent to enhance responsiveness. In contrast, we presented a static approach to avoid coordination.

We note that our commutativity definitions are similar to Lipton's [306] moverness in nature. However, they are defined for replicated rather than shared state. In addition, they are weaker conditions. In particular, $\mathcal{P}$-commutativity does not require the same final state and return value after the move as far as the guard and the invariant continue to hold.

Chapter 3

RDMA-Enabled Well-Coordination

3.1 Introduction

Data centers equipped with RDMA network interfaces are pervasive. These network interfaces support Remote Direct Memory Access (RDMA) [5, 250] from one node to another without going through the network and operating system stack or requiring CPU cycles from the other node. RDMAs mark the advent of a new model for distributed computing that combines the traditionally separate models of shared memory and message-passing, and have motivated new protocol designs [410, 16, 15]. This technology has been used to enable microsecond-scale [54] replicated services whose availability and low-latency are critical in applications such as finance and control.

RDMAs have been utilized to accelerate key-value stores [141, 249] and transactions [251, 474, 473]. In particular, they have been used to implement State Machine Replication (SMR) [420]. At its core, an SMR is a consensus or atomic broadcast protocol that executes requests in the same total order across replicas, and provides strong consistency. From the

long-lasting class of SMR protocols and systems, RDMA-accelerated SMRs have gained recent attention in projects such as DARE [382], APUS [467], Derecho [233], HovercRaft [262], Mu [17], Hermes [255] and Kite [179]. In contrast to traditional message-passing SMRs whose latencies are hundreds of microseconds, RDMA SMRs exhibit latencies that are less than a dozen microseconds. To maintain the low latency, it is crucial to avoid overloading the system. Therefore, the throughput of RDMA replicated systems is an important factor for their responsiveness as well [178].

In the message-passing model, SMR protocols such as Viewstamp [362], Paxos [279], Raft [364] and Spanner [123] provide strong consistency and have been the de facto standard for replication. However, practitioners [459, 370, 275, 122, 6] soon realized that SMR does not provide enough throughput, responsiveness and availability [9, 80, 81] for industrial applications, and opted for relaxed notations of consistency. In fact, deployments of SMR are often limited to small cluster sizes [123, 91, 229]. The large class of relaxed consistency notions [423] can be more efficiently provided [374, 280]. However, these notions forgo the total order of operations across replicas. Therefore, an immediate question is the safety of these notions for replication. Convergent and Commutative Replicated Data Types (CRDTs) [425, 424] and similar notions [25, 406] formally define replicated data types that converge under relaxed consistency. In addition to convergence, RA-linearizability [466] and ACC [304] define specifications for the functional correctness of these types. The definition of these types and their specifications led to projects on their composition [114, 342, 475, 475], and verification [78, 89, 148, 353, 498, 312, 187]. They were later followed by more expressive convergent types such as cloud, mergeable and reactive types [88, 248, 87, 348]. Convergence

might be enough for simple objects such as counters. However, relaxed consistency can further violate the integrity [37] of objects (such as maintaining a non-negative balance for a bank account). Thus, replicated data types that preserve integrity under relaxed consistency [35, 476, 355] followed.

However, not all operations can preserve convergence and integrity under relaxed consistency. Some operations do need strong consistency. Therefore, several projects considered hybrid models where each operation is executed under either relaxed or strong consistency based on its semantics. These projects include IPA [44], Sieve [301, 299, 300], Indigo [45, 46], CISE [189], Quelea [431], Carol [298] Hamsaz [226] and ECRO [133]. Hamsaz analyzes the given object to find the conflicting and dependent pairs of methods. It then synthesizes well-coordinated replicated objects that synchronize for conflicting, and preserve dependencies for dependent method calls. Well-coordinated replicated data types (WRDTs) guarantee convergence and integrity. Hampa [303] later added recency guarantees. Other projects tested and verified [246, 392, 354, 85, 56, 69], and repaired [393] replicated objects in hybrid models. Yet, others [224, 344, 263] considered the flow between relaxed and strong consistency.

However, the distributed system model that was considered for CRDTs and WRDTs was always the traditional message-passing network model. What is the semantics of CRDTs and WRDTs in the RDMA network model? What are the efficient coordination protocols that can leverage RDMAs to implement CRDTs and WRDTs?

RDMA offers two classes of communication primitives: two-sided and one-sided. Two-sided communication has similar semantics to the traditional message-passing model. A

node can execute a send operation to communicate a message to another node. The other node should execute an explicit receive operation to deliver and process the message. On the other hand, one-sided communication has similar semantics to the traditional shared memory model. A node directly performs a write or read operation on the memory of another node. The access is performed without involving the CPU of the other node. The new class, one-sided communication, tends to deliver lower response time since it bypasses the network and operating system stack and does not interrupt the CPU of the other node. How can well-coordination be efficiently implemented by one-sided communication?

This paper presents a novel operational semantics for RDMA WRDTs. The semantics divides methods of a given object into three categories, reducible, irreducible conflict-free, and conflicting, and declares distinct coordination requirements for each. The semantics does not perform any message-passing. In particular, reducible method calls can be performed with a single one-sided write operation that can be executed in parallel on the replicas. Similarly, the coordination for the other two categories is a sequence of local operations followed by one-sided remote operations. Further, we define an abstract operational semantics for WRDTs that captures well-coordination conditions. We prove that the concrete semantics of RDMA WRDTs refines the abstract semantics of WRDTs. Therefore, any execution of an RDMA WRDT is well-coordinated. Since (op-based) CRDTs are a special instance of WRDTs, each of the above two WRDT semantics subsume the semantics of CRDTs.

The operational semantics of RDMA WRDTs serves as a specification for their implementation and runtime system. We implement RDMA WRDT on top of consensus and reliable broadcast abstractions for the RDMA network model. We adopted and implemented

several CRDTs, and WRDTs. We evaluated our implementations by comparing them to both message-based and SMR-based implementations. The results show that on average, WRDTs exhibit more than 17x and 2.7x higher throughput respectively with almost the same response time. In summary, we make the following contributions:

- It introduces RDMA WRDTs, the first hybrid replicated data types for the RDMA network model.
- It presents novel operational semantics for RDMA WRDTs that is based solely on one-sided communication. It divides methods to three categories and defines the required coordination for each.
- It captures the notion of well-coordination as the abstract WRDT operational semantics, and formally proves that the RDMA WRDT semantics refine the abstract WRDT semantics, and preserve integrity and convergence.
- It efficiently implements the coordination protocols for RDMA WRDTs using only one-sided communication.
- It empirically shows that the RDMA WRDTs outperform the throughput of the existing message-based and SMR-based implementations.

3.2 Overview

We now illustrate RDMA replication with the familiar bank account example.

Example. As Fig. 3.1.(a) shows, an object of the *Account* class stores the balance state b, with the integrity property (or invariant) $\mathcal{I}$ that requires the balance to stay

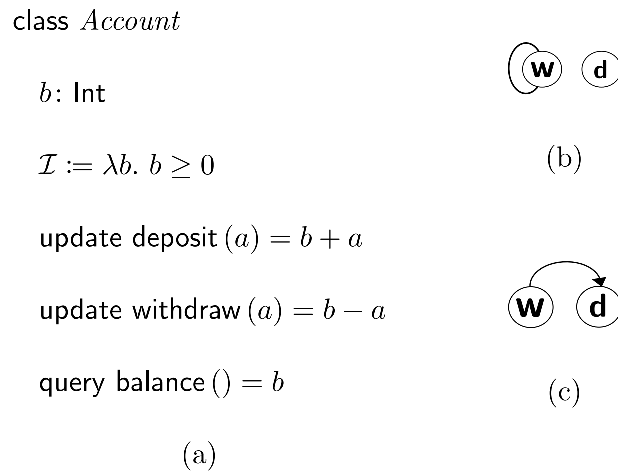


Figure 3.1: Bank Account. (a) The user specification, (b) The conflict graph, and (c) The dependency graph

non-negative. The class exposes the two update methods deposit and withdraw, which return the updated balance state, and the query method balance that returns the current balance.

The goal is to replicate an object on the given set of host processes such that the processes can issue requests to call update and query methods on the object. The processes should coordinate the calls so that the *integrity* property is always preserved, and the states of the processes eventually *converge*.

Well-coordination. A withdraw call issued at a process should be *locally permissible*: it should not overdraft the account, *i.e.*, not violate the invariant. Further, preserving integrity and convergence requires enforcing certain orders between calls across processes. For example, consider two withdraw calls, that each zeros the balance, are concurrently executed at two processes, and are propagated and applied to the other process in the opposite order. The second withdraw call in each process overdrafts the account and

violates the integrity property. Although it was locally permissible in its issuing process, it becomes impermissible in the other process. We say that two withdraw calls *permissible-conflict*. Further, consider a withdraw call that zeros the balance is executed right after a deposit call in a process. If the withdraw call is propagated and applied to other processes before the deposit call, then the withdraw call overdraft the account in the other processes. We say that the withdraw call is *dependent* on the deposit call. Similarly, for a set object, if an *add* and a *remove* call on the same element concurrently execute on two processes, and are applied to the other process in the opposite order, the state of the set object diverges. We say that the two calls *add* and *remove state-conflict*.

A replicated execution is well-coordinated if it is (1) locally permissible: every call should be permissible in the issuing process, (2) conflict-synchronizing: any pair of conflicting calls should have the same order across processes, and (3) dependency-preserving: a received call should be applied locally only if all the calls that it succeeded in the issuing process and is dependent on are already applied.

RDMA Coordination. RDMA allows a process to directly access the memory of other processes. Direct reads and writes are considerably faster than reading and writing by message-passing through the network stack. How can RDMA-enabled processes provide well-coordinated replicated objects? How can direct memory accesses accelerate the required coordination? Coordination mechanisms that can be captured as *local accesses and then a sequence of independent remote accesses* can execute efficiently. In these mechanisms, an access does not need to wait for a round-trip to receive the result of the previous access. In Fig. 5.2, we showcase the coordination of RDMA replicated objects for our account example

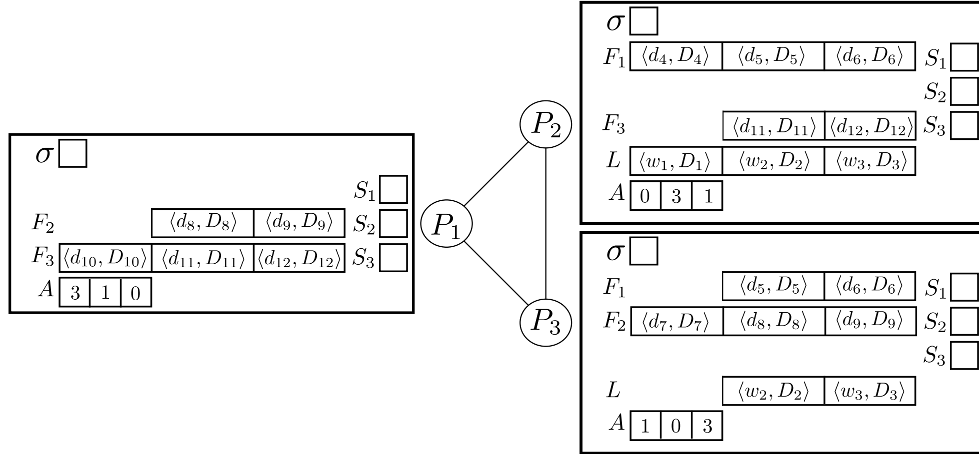


Figure 3.2: RDMA Replicated Bank Account

with three processes. Each process stores the state σ of the object: in our example, the balance for the account. It further stores other pieces of state that we visit in turn.

Conflicting methods. The conflict relation between the methods of an object induces the conflict graph. As Fig. 3.1.(b) shows, in our account example, the conflict graph has a loop on the withdraw method, and the deposit method is conflict-free. Every pair of calls on adjacent methods of the conflict graph need to be synchronized to have the same order in all the processes. We call a connected component of the conflict graph a *synchronization group*. Each synchronization group will have a leader process. Every other process in the group is a follower. Each follower process stores a *buffer* L for each synchronization group that stores pending calls on the methods of that group. In the account example, there is a group for withdraw method calls. In Fig. 5.2, the leader for this group is p_1 and each follower process keeps a buffer L for the withdraw calls. The leader checks local permissibility, orders and locally applies calls on the methods of the group, and remotely appends the ordered

calls to the buffers of the followers. A follower process periodically traverses its buffer and applies the pending calls to the state σ.

Follower processes receive updates without actively listening for and receiving messages through the network stack. The buffer for a group at each follower process is written by only the leader of the group and is read by only that local process. Therefore, the leader itself maintains the tail pointer of the buffer, and the coordination operation of the leader is locally reading and updating the tail pointer, and then remotely writing the call. We will see more details about how the buffers are managed in § 3.4.

Dependencies. As we saw above, the dependencies of calls should be respected. Therefore, each process keeps a mapping A from each process p and method u to the number of calls on u from p that are locally applied. When a call is shipped to be appended to a remote buffer, it is shipped with an account of its dependencies. The *dependency map* D of a call on a method u is the projection of the *applied map* A of the issuing process over the methods that u is dependent on. To respect the dependencies, when a process traverses a buffer, it applies a call only if the local applied map A is *point-wise greater than* the dependency map D that accompanies the call. When a call is applied, the local applied map A is advanced for the issuing process. As Fig. 3.1.(c) shows, in our account example, the dependencies of the withdraw method is the singleton set containing the deposit method. On the other hand, the deposit method is dependence-free. Thus, the applied and dependency maps are reduced to arrays indexed by process identifiers that store the number of deposit calls issued by each process.

Conflict-free methods. Calls to conflict-free methods such as the deposit method can avoid synchronization. Each process can autonomously propagate its conflict-free methods to other processes. Each process stores a *buffer* F for conflict-free calls from each other process. A process p that issues a conflict-free call checks that it is locally permissible, applies it locally, and remotely appends it to the buffers that other processes store for p. For example, in Fig. 5.2, the buffer F_2 in process p_1 stores the deposit calls issued by the process p_2. Similarly, the process p_3 stores a buffer F_2 for deposit calls issued by p_2. Similar to a conflicting call, a conflict-free call is accompanied by its dependency map D and is applied only if the local applied map A is ahead of its dependency map.

A process stores a buffer of conflict-free calls for each other process. The other process is the only writer of the buffer and the local process is the only reader. Therefore, the other process can perform the update by locally reading and writing the buffer tail, and then remotely writing the call (and its dependencies). Sharing buffers would require synchronization across processes. RDMA does provide compare-and-swap operations; however, they are more expensive than reads and writes and we avoid them with a *single-writer* design.

Reducible methods. We saw that conflict-free calls can avoid synchronization. The issuing process can simply propagate them to other processes by remote writes. An important observation is that for some conflict-free calls, even processing the calls can be done at the issuing process, and the other processes can receive the updates with no effort. In our account example, two deposit calls can be *summarized* to a single deposit with an amount equal to the sum of the two amounts. Therefore, the issuing process can summarize its deposit calls into a single deposit call, and remotely write only that call. As Fig. 5.2

shows, each buffer of conflict-free calls F can be replaced with a single summary call S. Each process stores a summary call for itself and each other process. When a process issues a new deposit call, it first calculates the summary of its current summary and the new call. It then overwrites the summary locally for itself and remotely for each other process. It further advances the applied map for the current process both locally and remotely. In contrast to buffers, each process keeps its own summary since the process needs its own summary to recalculate it.

Up to this point, the query method balance would simply return the stored state σ. However, in the presence of summarized calls, it should apply all the locally stored summary calls to σ to calculate the current state. In our example, it should apply the calls S_1, S_2 and S_3 to σ. Since the summary calls are conflict-free, they can be applied in any order. In a more elaborate object, it might be possible to summarize only separate subsets of methods which we call *summarization groups*. Each process stores a summary call from each process per summarization group.

Method categories. Summarization can accelerate the propagation and processing of calls. The summary is locally recalculated and is propagated by a *single remote write*. The caveat is that not all summarizable methods can be propagated as above. The method needs to be not only conflict-free but also dependence-free. If a method call has dependencies and it is summarized and remotely written for another process, the other process might not have applied the dependencies yet. Therefore, we consider three categories of methods. We say that a method is ***reducible*** as described above only if it is conflict-free, dependence-free and summarizable. We call a method ***irreducible conflict-free*** if it is

conflict-free but either not summarizable or not dependence-free. For example, in a grow-only set that has a contains and an add method (to add an element but not a set), the method add is conflict-free but is not summarizable. On the other hand, if the set object has an add method to add a set, then the add method is summarizable. As another example, consider a bank that is represented as a map that associates accounts to their balances, and in addition to deposit and withdraw, exposes the open method to open accounts. The deposit method is conflict-free but is dependent on the open method. Irreducible conflict-free methods use the conflict-free buffers that we saw above. Finally, ***conflicting*** calls use the conflicting buffers. We will consider these categories in more detail and the semantics of RDMA replicated objects in the next section.

3.3 Replicated Data Types

In this section, we first present how the high-level specifications of object data types can be captured. We then present a core operational semantics for well-coordinated replicated data types (WRDTs), and prove that it guarantees integrity and convergence. This abstract semantics will serve as the specification for replicated data types. We then present the semantics of RDMA replicated data types. It divides the methods of an object into three categories, and declares separate coordination requirements for each. We prove that this concrete semantics refines the earlier abstract semantics of WRDTs, and therefore, guarantees integrity and convergence.

3.3.1 Object Data Types

As Fig. 3.3 shows, a class of objects is a tuple $\langle \Sigma, \mathcal{I}, \overline{u \coloneqq d}, \overline{q \coloneqq d} \rangle$ that defines the state type Σ, the invariant (or integrity property) $\mathcal{I}$ on the state, and the definitions of update methods u and query methods q. The invariant (or integrity) $\mathcal{I}$ is a predicate on the state. The definition of an update method is a function from the parameter and the pre-state σ to the post-state. Similarly, the definition of a query method is a function from the parameter and the pre-state σ to the return value. The object is replicated on the set of processes P. Any process p can accept and issue update calls $u(v)$ or query calls $q(v)$. The calls have unique request identifiers r. An update call is decorated with the issuing process p and the request identifier r. We elide these decorations when they are not needed or are evident from the context.

σ	$:$	Σ	State
$\mathcal{I}$			Invariant
v	$:$	V	Value
u	$:$	U	Update Method
q	$:$	Q	Query Method
d	$:$	$\lambda x, \sigma.\ e$	Definition
e			Expression
o	$\coloneqq$	$\langle \Sigma, \mathcal{I}, \overline{u \coloneqq d}, \overline{q \coloneqq d} \rangle$	Object
p	$:$	P	Process
r	$:$	R	Request Identifier
$c : C$	$\coloneqq$	$u(v)_{p,r}$	Update Call
a	$\coloneqq$	$q(v)$	Query Call
ℓ	$\coloneqq$	$(p, u(v)_r) \mid (p, q(v))$	Label
τ	$\coloneqq$	ℓ^*	Trace

Figure 3.3: Basic Syntax

Clients can request method calls at every process, and the processes coordinate these calls. A label for a call request is the pair of the issuing process and a call, and a trace is a sequence of labels.

3.3.2 Semantics of Well-Coordinated Replicated Data Types

We now present the operational semantics for well-coordinated replicated data types (WRDTs). We first see the replicated state and the coordination conditions that the transition rules use.

Replicated State. The state of the given object is replicated across processes. The replicated state $\mathbf{ss}$ is a mapping from each process p to its states σ. The execution history x of a process is modeled as a permutation of a set of calls. Since query calls do not mutate the state, an execution history only keeps update calls. We write $c \in x$ to denote that the call c is in the history x. An execution history x defines a total order on its calls: we write $c \prec_{\mathsf{x}} c'$ iff the call c precedes the call c' in the execution history x. A replicated execution xs is a function from each process to its execution history. The state W of our operational semantics is the pair of the replicated state $\mathbf{ss}$ and the replicated execution xs. In the initial state W_0, the state of each process is the same state σ_0 that satisfies the invariant $\mathcal{I}$, and the history of each process is an empty list.

Coordination Conditions. We now define the coordination conditions in steps. For the sake of brevity, we elide separate definition environments.

State-conflict. A replicated execution is convergent if the state of the processes is the same after all the calls are propagated to all processes. Out of order delivery of calls at

$$
\begin{array}{llll}
\textsf{ss} & : & P \mapsto \Sigma & \text{Replicated State} \\
\textsf{xs} & : & P \mapsto \mathsf{List}(C) & \text{Replicated Execution} \\
W & := & \langle \textsf{ss}, \textsf{xs} \rangle & \text{World} \\
W_0 & := & \langle \overline{[p \mapsto \sigma_0]}_{p \in P}, \overline{[p \mapsto \emptyset]}_{p \in P} \rangle & \text{Initial World}
\end{array}
$$

Figure 3.4: WRDT State

different processes can lead to divergence of their states. For example, for the set data type, according to the execution order of the two calls add and remove of the same element, there are two possible resulting states. Therefore, they should synchronize. Two method calls c_1 and c_2 $\mathcal{S}$-commute, written as $c_1 \leftrightarrows_{\mathcal{S}} c_2$, iff $c \circ c' = c' \circ c$ (where $\circ$ is function composition). Otherwise, they $\mathcal{S}$-conflict, written as $c_1 \bowtie_{\mathcal{S}} c_2$.

Integrity and Permissibility. In the execution history of a process, the post-state of a call is the pre-state of the next call. The body of each method can assume and rely on the invariant in the pre-state; it should then preserve it in the post-state for the next call. The notion of permissibility requires the invariant to hold in the post-state. A method call c is permissible in a state σ, written as $\mathcal{P}(\sigma, c)$, iff $\mathcal{I}(c(\sigma))$. The initial state is assumed to satisfy the invariant. Therefore, if it is ensured that every call is permissible in its pre-state, then by induction, every call enjoys integrity in its pre-state. Permissibility leads to integrity.

Invariant-sufficiency. There are calls (*e.g.*, deposit on a bank account) that are always permissible (*i.e.*, a deposit never overdrafts) as far as they are applied to a state that has integrity. Thus, when they are broadcast and executed on another process, in order to

be permissible, they only need the pre-state to have integrity. A call c is invariant-sufficient iff for every state σ, if $\mathcal{I}(\sigma)$ then $\mathcal{P}(\sigma, c)$.

Permissible-Right-Commutativity. However, not all calls are invariant-sufficient. For example, a withdraw call may be permissible in a process but may become impermissible in another where it is executed after a racing withdraw call. A call c_1 $\mathcal{P}$-R-commutes with another c_2, written as $c_1 \rightarrow_{\mathcal{P}} c_2$, iff for every state σ, if $\mathcal{P}(\sigma, c_1)$ then $\mathcal{P}(c_2(\sigma), c_1)$. *i.e.*, permissibility holds even after c_1 is pushed right after c_2.

Permissible-conflict. We say that c_1 $\mathcal{P}$-concurs with a call c_2, if c_1 is invariant-sufficient or $c_1 \rightarrow_{\mathcal{P}} c_2$. Otherwise, c_1 $\mathcal{P}$-conflicts with c_2 and needs to synchronize with it.

Conflict. We say that two calls c_1 and c_2 *concur* iff they both $\mathcal{S}$-commute and $\mathcal{P}$-concur with each other. Otherwise, we say they *conflict* written as $c_1 \bowtie c_2$. Conflicting calls need synchronization. A call is conflict-free if it does not conflict with any other call.

Permissible-Left-Commutativity. We saw above that invariant-sufficient calls always preserve the invariant. However, there are calls whose preservation of the invariant is dependent on the calls that precede them. For example, a withdraw call may be dependent on the money deposited by a preceding deposit call in the issuing process; after propagation to another process, if the withdraw moves to the left of the deposit, the withdraw call can overdraft. A call c_2 $\mathcal{P}$-L-commutes a call c_1, written as $c_2 \leftarrow_{\mathcal{P}} c_1$ iff for every state σ, if c_2 is permissible in the post-state of the call c_1 on σ, *i.e.*, $\mathcal{P}(c_1(\sigma), c_2)$, then c_2 is permissible in σ, *i.e.*, $\mathcal{P}(\sigma, c_2)$, as well.

$$\textsc{Call}\quad \frac{c = u(v)_{p,r} \quad \mathcal{P}(\sigma, c) \quad \mathsf{CallConfSync}(\mathsf{xs}, p, c) \quad \mathsf{xs}' = \mathsf{xs}\,[p \mapsto (\mathsf{xs}(p) ::: c)] \quad \sigma' = u(v)(\sigma)}{\langle \mathbf{ss}[p \mapsto \sigma], \mathsf{xs}\rangle \xrightarrow{p,\ u(v)_r} \langle \mathbf{ss}[p \mapsto \sigma'], \mathsf{xs}'\rangle}$$

$$\textsc{Prop}\quad \frac{c = u(v)_{p',r} \quad c \in \mathsf{xs}(p') \setminus \mathsf{xs}(p) \quad \mathsf{PropConfSync}(\mathsf{xs}, p, c) \quad \mathsf{PropDepPres}(\mathsf{xs}, p', p, c) \quad \mathsf{xs}' = \mathsf{xs}\,[p \mapsto (\mathsf{xs}(p) ::: c)] \quad \sigma' = u(v)(\sigma)}{\langle \mathbf{ss}[p \mapsto \sigma], \mathsf{xs}\rangle \rightarrow \langle \mathbf{ss}[p \mapsto \sigma'], \mathsf{xs}'\rangle}$$

$$\textsc{Query}\quad \frac{v' = q(v)(\sigma)}{\langle \mathbf{ss}[p \mapsto \sigma], _\rangle \xrightarrow{p,\ q(v):v'} \langle \mathbf{ss}[p \mapsto \sigma], _\rangle}$$

$$\mathsf{CallConfSync}\,(\mathsf{xs}, p, c) \quad := \quad \forall c', p'. \quad c' \in \mathsf{xs}(p') \,\wedge\, c \bowtie c' \,\rightarrow\, c' \in \mathsf{xs}(p)$$

$$\mathsf{PropConfSync}\,(\mathsf{xs}, p, c) \quad := \quad \forall c', p'. \quad c' \prec_{\mathsf{xs}(p')} c \,\wedge\, c \bowtie c' \,\rightarrow\, c' \in \mathsf{xs}(p)$$

$$\mathsf{PropDepPres}\,(\mathsf{xs}, p', p, c) \quad := \quad \forall c'. \quad c' \prec_{\mathsf{xs}(p')} c \,\wedge\, c \not\perp\!\!\!\perp c' \,\rightarrow\, c' \in \mathsf{xs}(p)$$

Figure 3.5: WRDTs Semantics

Dependency. A call c_2 is independent of c_1, written as $c_2 \perp\!\!\!\perp c_1$, if c_2 is invariant-sufficient or $c_2 \leftarrow_{\mathcal{P}} c_1$. Otherwise, c_2 is dependent on c_1, written as $c_2 \not\perp\!\!\!\perp c_1$. If c_1 is executed before c_2 in the issuing process of c_2, and $c_2 \not\perp\!\!\!\perp c_1$, then c_2 can be applied to another process only if c_1 is already applied.

Given the integrity properties, the representation and automated checking and inference of conflict and dependency relations [189, 226, 45, 355] is a topic of active research.

Transitions Rules. The transition rules of the operational semantics are presented in Fig. 3.5. The rule Call accepts and executes an update method call c at the process p. It first checks that the call is locally permissible $\mathcal{P}(\sigma, c)$. It then checks that if

the new call c conflicts with a call c' that another process p' has executed, then c' should have been already executed at the current process p. Thus, executing the call keeps the execution conflict-synchronizing. A replicated execution is conflict-synchronizing if every pair of conflicting calls have the same order across processes.

The rule PROP propagates a call c from another process p' to the current process p. Similar to the previous rule, this rule makes sure that executing c keeps the execution conflict-synchronizing. It checks that if a call c' that conflicts with the new call c is executed before c in any other process, then c' is already executed at the current process p. The rule also makes sure that executing c keeps the execution dependency-preserving. A replicated execution is dependency-preserving if for every call, its preceding dependencies in its issuing process precede it in the other processes as well. The rule checks that if a call c' is executed before c in p', and c is dependent on c', then c' is already executed at the current process p.

The rule QUERY executes a query call $q(v)$ at a process p. The return value v' is the result of applying the call to the current state σ of p.

We note that (op-based) CRDTs (Convergent and Commutative Replicated Data Types) [424] are a special case of WRDTs where it is assumed that all method calls state-commute with each other, and the integrity predicate is simply the assertion `true`. The conflict-synchronization and dependency-preservation conditions in the above transition rules are trivially satisfied. Therefore, the rules are always enabled and calls can propagate without coordination.

We also note that linearizable data types are a special case of WRDTs where the conflict relation is complete. The executions of WRDTs are conflict-synchronizing. Therefore,

all the calls are totally ordered across processes. The execution histories $\mathsf{xs}(p)$ of processes p are the prefixes of the total order. Further, the real-time preservation property is maintained by the CALL rule as (1) a call c in process p returns only after adding c to $\mathsf{xs}(p)$, and (2) the condition $\mathsf{CallConfSync}$ ensures that a process p' executes a call c' only after every call c that is in $\mathsf{xs}(p)$ is also in $\mathsf{xs}(p')$.

Guarantees. Every well-coordinated execution enjoys integrity and convergence.

Integrity is the safety property that the invariant predicate holds for all reachable states of a process.

Lemma 24 (Integrity) *For all* $\mathbf{ss}$ *and* p*, if* $W_0 \rightarrow^* \langle \mathbf{ss}, _ \rangle$ *then* $\mathcal{I}(\mathbf{ss}(p))$*.*

Convergence is the safety property that states that processes which have applied the same set of calls have the same state. We say that two histories x and x' are equivalent $x \sim x'$ if they have the same set of calls.

Lemma 25 (Convergence) *For all* $\mathbf{ss}$*,* xs*,* p *and* p'*, if* $W_0 \rightarrow^* \langle \mathbf{ss}, \mathsf{xs} \rangle$ *and* $\mathsf{xs}(p) \sim \mathsf{xs}(p')$ *then* $\mathbf{ss}(p) = \mathbf{ss}(p')$*.*

3.3.3 RDMA Replicated Data Types

We now present the operational semantics of RDMA replicated data types. The semantics divides methods into three categories, reducible, irreducible conflict-free, and conflicting, and presents dedicated coordination requirements for each. We prove that this concrete semantics refines the abstract semantics of WRDTs that we saw in the previous subsection. This concrete semantics captures the core of our runtime system that we will see in § 3.4.

Method Categories. A pair of methods u and u' *conflict* if there are arguments v and v' such that the calls $u(v)$ and $u'(v')$ conflict. We say that a method is *conflicting* if there is a method that it conflicts with, and say that it is *conflict-free* otherwise. Similarly a method is *dependent* on another method if there is a call on the former that is dependent on a call on the latter. We write the set of methods that a method u is dependent on as $\mathsf{Dep}(u)$. We say that a method is *dependence-free* if its set of dependencies is empty.

As we saw in the semantics of WRDTs, calls to a pair of conflicting methods should preserve the same order across processes. The conflict relation on methods induces an undirected graph that we call the conflict graph. The *synchronization* group $\mathsf{SyncGroup}(u)$ of a method u is the connected component of the method in the conflict graph. Methods of a synchronization group synchronize with each other.

The summary of two calls c and c', written as $\mathsf{Summarize}(c, c')$, is a call c'' iff for all states σ, $c \circ c'(\sigma) = c''(\sigma)$. For example, the summary of $\mathsf{deposit}(3)$ and $\mathsf{deposit}(4)$ is $\mathsf{deposit}(7)$. We say that a set of methods g are a *summarization group* if calls on g are closed under summarization. A sequence of calls from a summarization group can be successively summarized into a single call. We say that a method u is *summarizable* if it is a member of a summarization group, that we write as $\mathsf{SumGroup}(u)$. Otherwise, it is not summarizable, that we write as $\mathsf{SumGroup}(u) = \bot$. We say that a method is *reducible* if it is conflict-free, dependence-free and summarizable. Otherwise, we say that it is *irreducible*.

The semantics utilizes the remote write feature of RDMAs to directly communicate updates from one process to another. In order to tolerate faults and also have low latency for query methods, each process keeps a local replica of the state, and performs remote writes

but no remote reads. The semantics distinguishes between three categories of methods: (1) conflicting methods, (2) irreducible conflict-free methods, *i.e.*, methods that are conflict-free but are either not dependence-free or not summarizable, and (3) reducible methods. We consider the coordination for each category in turn.

For conflicting methods, each synchronization group is assigned a leader process, and each process replicates a buffer of calls for each group. As we will see in the transition rules, the leader process of the group orders the calls on the group and then remotely appends them to the buffer of each other process. The other processes periodically traverse their own buffers and locally apply the calls. The leader is the single remote writer of all buffers, and each process is the single reader of its own local buffer.

Conflict-free methods do not need synchronization and processes autonomously issue and propagate them. Each process replicates a buffer of calls for all the irreducible conflict-free calls of each other process. When a process p issues an irreducible conflict-free call, it remotely appends it to the buffers that each other process stores for p. The other processes periodically traverse their buffer, locally apply the calls and then discard them. The process p is the single remote writer of these buffers, and each process is the single reader of its own local buffer.

Reducible methods are remarkable: the issuing process can reduce them together locally and then remotely write them for other processes (*i.e.*, using RDMA one-sided communication). Therefore, for each pair of a summarization group of methods and a process, each process replicates a single call rather than a buffer of calls. This not only saves space but also time as it eliminates the buffer traversals by the target processes. Thanks to direct

$$
\begin{array}{rcll}
g : G & = & \mathsf{Set}(U) & \text{Method Group} \\
A : \mathcal{A} & = & P \rightarrow U \rightarrow \mathbb{N} & \text{Applied calls} \\
D : \mathcal{D} & = & P \rightarrow U \rightarrow \mathbb{N} & \text{Dependencies} \\
S : \mathcal{S} & = & G \rightarrow P \rightarrow C & \text{Summarized calls} \\
F : \mathcal{F} & = & P \rightarrow \mathsf{List}\ (C \times \mathcal{D}) & \text{Conflict-free buffers} \\
L : \mathcal{L} & = & G \rightarrow \mathsf{List}\ (C \times \mathcal{D}) & \text{Conflicting buffers} \\
K : \mathcal{K} & = & P \rightarrow \Sigma \times \mathcal{A} \times \mathcal{S} \times \mathcal{F} \times \mathcal{L} & \text{Configuration}
\end{array}
$$

Figure 3.6: WRDT RDMA State

RDMA writes, other processes obliviously receive updates without receiving and traversing messages. Similar to the above buffers, each summarization call is written by only a single remote process and is only read by the local process.

Replicated State. Fig. 3.6 shows the state of the operational semantics. A configuration K is a mapping from processes p to tuples $\langle \sigma, A, S, F, L \rangle$. The stored state σ represents the result of applying calls at process p. These calls are either conflicting or irreducible conflict-free. The applied calls A is a mapping that maps a process p' and an update method u to the number of calls on u from p' that are applied in the current process. The summarized calls S is a mapping that maps a summarization group g of update methods and a process p' to a call c that summarizes calls on methods in g from p'. The conflict-free calls F is a mapping that maps a process p' to the list of irreducible conflict-free calls received from p'; each call c is coupled with its dependencies D. The conflicting calls L is a mapping

that maps a synchronization group g of update methods to the list of calls on methods of g; as before, each call c comes with its dependencies D.

Given a summarized map of calls S, the state $\mathsf{Apply}(S)(\sigma)$ is the result of applying the calls in the range of S to σ. Since the calls in the summarized map are conflict-free, they can be applied in any order. An applied map A satisfies a dependency map D, written as $D \leq A$, iff for all p and u, $D(p, u) \leq A(p, u)$.

We note that the semantics explicitly models the leader of each synchronization group that orders the calls on that group. The map L stores a copy of the total order at each process. The leader updates the orders stored at other processes by remote writes. Since these orders are the same, this model is a refinement of an abstract leaderless model where a configuration stores a single copy of the order.

Transition rules. Transition rules are presented in Fig. 3.7 and Fig. 3.8. The rule REDUCE presents the transition for a reducible method call $u(v)$ by a process p_j. If u is conflict-free (*i.e.*, $\mathsf{SyncGroup}(u) = \bot$), is dependence-free (*i.e.*, $\mathsf{Dep}(u) = \emptyset$), and is summarizable (*i.e.*, $\mathsf{SumGroup}(u) = g$), then the call $u(v)$ can be reduced. The rule first checks that the call is locally permissible. The current state σ of this process p_j is calculated by applying the summarized calls that it has received S_j to its stored state σ_j. The post-state that results from applying $u(v)$ to σ should preserve the integrity property $\mathcal{I}$. The current summarizing call $u'(v')$ for the group g and the new call $u(v)$ are summarized as the call $u''(v'')$. The new summary call is stored at all the processes p_i, both locally at the current process p_j and remotely at other processes. Consequently, the number of applied calls on

Reduce

$$\frac{\begin{array}{c}\mathsf{SyncGroup}(u) = \bot \qquad \mathsf{Dep}(u) = \emptyset \qquad \mathsf{SumGroup}(u) = g \\ \sigma = \mathsf{Apply}(S_j)(\sigma_j) \qquad \mathcal{I}(u(v)(\sigma)) \qquad S_j(g, p_j) = u'(v') \qquad \mathsf{Summarize}(u'(v'), u(v)) = u''(v'') \\ \overline{S_i' = S_i[(g, p_i) \mapsto u''(v'')]}_{i \in \{1..|P|\}} \qquad n = A_j(p_j, u) + 1 \qquad \overline{A_i' = A_i[(p_i, u) \mapsto n]}_{i \in \{1..|P|\}}\end{array}}{\begin{array}{c}\overline{[p_i \mapsto \sigma_i, A_i, S_i, _, _]}_{i \in \{1..|P|\}} \xrightarrow{p_j,\ u(v)} \\ \overline{[p_i \mapsto \sigma_i, A_i', S_i', _, _]}_{i \in \{1..|P|\}}\end{array}}$$

Free

$$\frac{\begin{array}{c}\mathsf{SyncGroup}(u) = \bot \\ \mathsf{Dep}(u) \neq \emptyset \vee \mathsf{SumGroup}(u) = \bot \qquad \sigma_j' = u(v)(\sigma_j) \qquad \sigma' = \mathsf{Apply}(S_j)(\sigma_j') \qquad \mathcal{I}(\sigma') \\ A_j' = A_j[(p_j, u) \mapsto A_j(p_j, u) + 1] \\ \mathsf{Dep}(u) = \{\overline{u'}\} \qquad \overline{F_i' = F_i[p_j \mapsto F_i(p_j) ::: \langle u(v), A_j|\{\overline{u'}\}\rangle]}_{i \in \{1..|P|\} \setminus \{j\}}\end{array}}{\begin{array}{c}[p_j \mapsto \sigma_j, A_j, _, _, _]\overline{[p_i \mapsto _, _, _, F_i, _]}_{i \in \{1..|P|\} \setminus \{j\}} \xrightarrow{p_j,\ u(v)} \\ [p_j \mapsto \sigma_j', A_j', _, _, _]\overline{[p_i \mapsto _, _, _, F_i', _]}_{i \in \{1..|P|\} \setminus \{j\}}\end{array}}$$

Conf

$$\frac{\begin{array}{c}\mathsf{SyncGroup}(u) = g \qquad \mathsf{Leader}(g) = p_j \\ \sigma_j' = u(v)(\sigma_j) \qquad \sigma' = \mathsf{Apply}(S_j)(\sigma_j') \qquad \mathcal{I}(\sigma') \\ A_j' = A_j[(p_j, u) \mapsto A_j(p_j, u) + 1] \\ \mathsf{Dep}(u) = \{\overline{u'}\} \qquad \overline{L_i' = L_i[g \mapsto L_i(g) ::: \langle u(v), A_j|\{\overline{u'}\}\rangle]}_{i \in \{1..|P|\} \setminus \{j\}}\end{array}}{\begin{array}{c}[p_j \mapsto \sigma_j, A_j, _, _, _]\overline{[p_i \mapsto _, _, _, _, L_i]}_{i \in \{1..|P|\} \setminus \{j\}} \xrightarrow{p_j,\ u(v)} \\ [p_j \mapsto \sigma_j', A_j', _, _, _]\overline{[p_i \mapsto _, _, _, _, L_i']}_{i \in \{1..|P|\} \setminus \{j\}}\end{array}}$$

Free-App

$$\frac{D \leq A \qquad \sigma' = u(v)(\sigma) \qquad A' = A[(p', u) \mapsto A(p', u) + 1]}{\begin{array}{c}K[p \mapsto \sigma, A, _, F[p' \mapsto \langle u(v), D\rangle :: l], _] \rightarrow \\ K[p \mapsto \sigma', A', _, F[p' \mapsto l], _]\end{array}}$$

Figure 3.7: RDMA WRDTs Semantics

CONF-APP

$$\frac{D \leq A \qquad \sigma' = u(v)(\sigma) \qquad \mathsf{Leader}(g) = p' \qquad A' = A[(p', u) \mapsto A(p', u) + 1]}{\begin{array}{c} K[p \mapsto \sigma, A, _, _, L[g \mapsto \langle u(v), D\rangle :: l]] \rightarrow \\ K[p \mapsto \sigma', A', _, _, L[g \mapsto l]] \end{array}}$$

QUERY

$$\frac{\sigma' = \mathsf{Apply}(S)(\sigma) \qquad v' = q(v)(\sigma')}{K[p \mapsto \sigma, _, S, _, _] \xrightarrow{p,\ q(v):v'} K[p \mapsto \sigma, _, S, _, _]}$$

Figure 3.8: More RDMA WRDTs Semantics

method u from p_j is incremented locally and stored both locally and remotely. The two remote writes are independent and can be issued concurrently.

The rule FREE presents the transition for a conflict-free but irreducible (*i.e.*, dependent or not summarizable) method call $u(v)$ by a process p_j. If u is conflict-free (i.e, $\mathsf{SyncGroup}(u) = \perp$), but either dependent (*i.e.*, $\mathsf{Dep}(u) \neq \emptyset$) or not summarizable (*i.e.*, $\mathsf{SumGroup}(u) = \perp$), then the call $u(v)$ cannot be reduced but can avoid synchronization. As before, the call is first checked to be locally permissible. Then, the call is locally applied and the number of applied calls on u is incremented. It is then remotely written for each other process p_i: it is appended to the list $F_i(p_j)$ that stores at p_i the conflict-free calls issued from p_j. Let the dependencies $\mathsf{Dep}(u)$ of u be the set of methods $\{\overline{u'}\}$. The call $u(v)$ is accompanied with a record of applied calls that $u(v)$ is dependent on, *i.e.*, $A_j|\{\overline{u'}\}$.

The rule CONF presents the transition for a conflicting method call $u(v)$ by a process p_j. The process p_j is the leader for the method u. As before, the call is first checked to be locally permissible. Then, the call is locally applied and the number of applied calls

is advanced. Let the synchronization group of u be g (*i.e.*, $\mathsf{SyncGroup}(u) = g$). The call is remotely written for each other process p_i: the call is appended to the list $L_i(g)$ that stores at p_i the calls from the synchronization group g. As before, the call is accompanied with a record of its dependencies. The function Leader uniquely maps each synchronization group to a process. As we will see in the next section, changing the leader of a synchronization group from one process to another preserves this uniqueness.

The rule FREE-APP presents an internal transition by a process p to apply a call from its conflict-free buffers F. A call from the buffer can be applied only if the record of the already applied calls A at p is ahead of the dependencies D of the call. The stored state σ is updated and the record of applied calls A is advanced. The rule CONF-APP is similar except that it applies a call from the conflicting buffers L.

Finally, the rule QUERY presents the transition of a query $q(v)$ by a process p. The return value v' results from applying $q(v)$ to the current state σ', which is calculated by applying the summarized calls S to the stored state σ.

We will see an implementation of this semantics in the next section.

Correctness. The RDMA WRDT semantics (Fig. 3.7 and Fig. 3.8) refines the WRDT semantics (Fig. 3.5): any trace that is observed from the former can be observed from the latter.

Lemma 26 (Refinement) *For all K and τ, if $K_0 \xrightarrow{\tau} K$, then there exists W, such that $W_0 \xrightarrow{\tau} W$.*

The refinement relation and the proof are available in the supplemental material.

The immediate corollaries are that executions of RDMA WRDTs enjoy integrity and convergence.

All the reachable states of each process satisfy the integrity property. The state of a process is the result of applying the summarized calls S to the stored state σ.

Corollary 27 (Integrity) *For all* $i \in \{1..|P|\}$,
if $K_0 \rightarrow^* \overline{[p_i \mapsto \sigma_i, _, S_i, _, _]}_{i \in \{1..|P|\}}$ *then* $\mathcal{I}(\mathsf{Apply}(S_i)(\sigma_i))$.

When all the buffers F and L are applied, the states of the processes converge.

Corollary 28 (Convergence) *For all* $i, j \in \{1..|P|\}$,
if $K_0 \rightarrow^* \overline{[p_i \mapsto \sigma_i, _, S_i, F_i, L_i]}_{i \in \{1..|P|\}}$ *and* $F_i = F_j = \emptyset$ *and* $L_i = L_j = \emptyset$ *then* $\mathsf{Apply}(S_i)(\sigma_i) = \mathsf{Apply}(S_j)(\sigma_j)$.

3.4 Implementation

Hamband[1] is implemented on top of RDMA's Reliable Connection (RC) model using ibverbs library over Infiniband [3] in 1430 lines of code. First, we briefly explain the metadata stored at each node and then describe how Hamband propagates calls. In particular, we describe the reliable broadcast protocol that we use to broadcast conflict-free calls.

Meta-data. As we saw in the semantics, each node stores a location S for each reduction group, and two separate set of buffers: conflicting calls L, and irreducible conflict-free calls F. Each buffer has a head that is locally stored at the host node and a tail

[1] https://github.com/fhoushmand/Hamband.git

that is remotely stored at the single writer node. The buffers store pairs of calls c and their dependencies D. The dependency map that we saw in the semantics is efficiently represented as an array per node where each cell represents the number of calls on a method. Since the number of dependencies of methods is not necessarily the same, the dependency arrays are variable-sized. When the pairs of a buffer are traversed, the size of dependency arrays in the second element is decided based on the identifier of the method in the first element. Each node also keeps the number of applied calls A from each node as an integer array that is indexed by method identifiers. Further, each node keeps the following coordination analysis results: the list of synchronization groups, and a mapping from each method to the set of methods that it is dependent on.

Processing requests. Upon receiving a call request from the client, there are four possibilities based on the category of the method. First, if the call is a query, it is executed locally and the result is returned back to the client. Second, if the call is reducible, it is reduced with the local summary, and the result is remotely overwritten to the remote summary locations. Third, if the call is irreducible conflict-free, it is executed locally and written to the remote buffers F. To guarantee convergence, the above two propagations are done using the reliable broadcast abstraction. Before propagation, a call is assigned a unique id, paired with its dependency arrays and is serialized into a byte stream. Fourth, we instantiate a Mu [17] consensus instance for each synchronization group. If the call belongs to a synchronization group, it is sent to the corresponding consensus instance to be ordered in the L buffers.

In each node, two threads traverse and process the calls of F and L buffers if their dependencies are already satisfied. Each buffer has a head that is locally stored at the receiver node, and a tail that is remotely stored at the single writer node. Each call in the buffer contains a canary bit as the last bit. To check whether the buffer is not empty and the call is not concurrently being written, the receiver checks the canary bit. If the check fails, then the periodical traversal of the buffer will retry later. Even if a call is missed in a traversal, it will be processed in the next one. After a successful read, the head pointer is advanced to the next location. The calls at locations before the head are already executed. To avoid memory overflow, these locations are reused.

RDMA Reliable Broadcast. The reliable broadcast abstraction guarantees the following agreement property: if a message m is delivered by some correct node, then m is eventually delivered by every correct node. The best-effort broadcast abstraction on RDMAs can simply write the message remotely for all nodes. However, the source node may crash in the middle of the remote writes and violate the agreement property. To provide agreement, the source node keeps a shared memory location, and gives other nodes read access to the location. The source locally writes in the shared location before remotely writing it for others. It clears the location afterwards. The shared location acts as a backup. Each node has a heartbeat thread that periodically updates a local counter. This counter is periodically read by other nodes to determine whether that node is still alive or not. If other nodes detect that the source has failed, they remotely read the shared location, obtain any pending message, and check if they have received it. If not, they deliver the message.

Synchronization. We adopt Mu consensus protocol [17] to serialize calls in the L buffers. Under normal execution, only a designated leader has the permission to write to the follower buffers. As we described above, nodes have a heartbeat mechanism to let others detect when they fail. If a follower suspects that the leader has failed, it requests others to accept it as the leader and waits for a majority of them to acknowledge. At any time, each node recognizes only one node as the leader and grants it the write permission. A node revokes permission from the previous leader before granting it to the next. Therefore, only one node can be recognized as the leader by a majority and write to L buffers.

3.5 Experimental Results

We now evaluate the RDMA WRDTs of Hamband; we compare them with message-passing CRDTs and RDMA-enabled SMRs. We observe that Hamband outperforms message-passing CRDTs by 17.7× and 23× in terms of throughput and response time respectively. Further, it provides 2.7× higher throughput than the state-of-the-art SMR system, Mu, with almost the same response time.

Mu [17] is a low-latency leader-based SMR system, such as ZAB [229] that is used in the industry. However, we note that RDMA WRTDs are independent of any leader-based SMR protocol. They modularly use an SMR system (*i.e.*, consensus) for the conflicting category of methods. On the other hand, for the two conflict-free categories of methods, they avoid the synchronization cost and use more efficient broadcast protocols or just single RDMA writes. Therefore, they improve performance over the SMR baseline.

Questions. In our experiments, we aim to answer the following question in terms of both throughput and response time: How do RDMA WRDTs compare to RDMA-based SMRs? We further investigate the following more detailed questions for RDMA WRDTs. (1) What is the effect of separate synchronization groups for conflicting methods? (2) What is the impact of failures?

Platform and setup. We performed the experiments on a 7-node cluster, each with 8 AMD opteron 6376 cores and 50GB memory. The nodes are connected via 40Gbps Infiniband network, and run CentOS 7.4 Linux x86_64 kernel version 3.10. All programs are compiled with gcc-7.4.0.

All the experiments are done with 4M operations unless stated otherwise. We randomly generate method calls and uniformly distribute update calls between updated methods. The calls on conflicting methods are automatically redirected to the corresponding leader node(s). All the other calls including conflict-free and query calls are divided equally between the nodes.

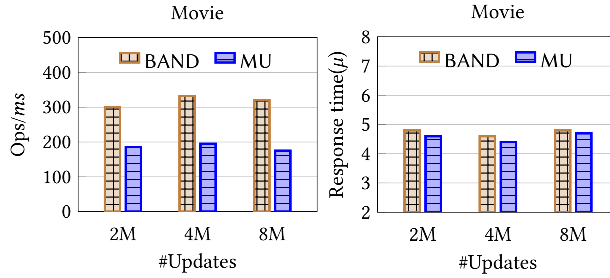
The throughput is calculated by dividing the total number of calls by the time that it takes for all the update calls to be replicated on all the nodes. The response time is calculated as the average response time over all the calls. We repeat each experiment 3 times and report the average.

Experiments and findings. We perform experiment on the RDMA WRDT of a database schema that has all three categories of methods, reducible, irreducible conflict-free, and conflicting. The results indicate that synchronization avoidance and remote buffering improve the throughput of the database schema by up to 21% compared to the Mu SMR.

Further, we inject failures into the RDMA WRDT of a database schema. The experiment shows that it is able to tolerate leader or follower failure with minimum overhead for both throughput and response time of conflict-free operations.

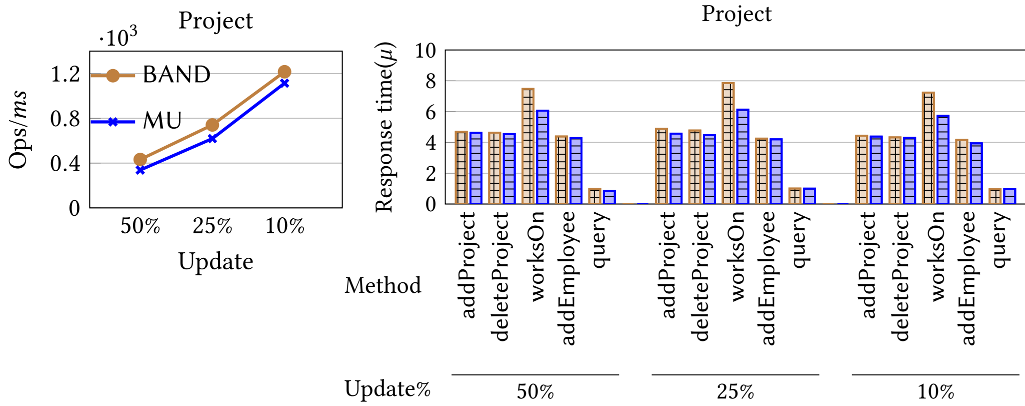
Use-cases and benchmarks. We adopted [226, 368] three relational schemata: project management, courseware, and movie. The project management class has five methods, namely, addProject, deleteProject, worksOn, addEmployee, and query. The methods addProject, deleteProject, and worksOn belong to a synchronization group and the worksOn method depends on addProject and addEmployee due to the foreign-key constraint. The movie class has four methods addCustomer, deleteCustomer, addMovie, and deleteMovie operating on two separate relations; therefore, forming two synchronization groups. There is no dependency in this class. The Courseware class has five methods, namely, addCourse, deleteCourse, enroll, registerStudent, and query. Conflict analysis shows that there is one synchronization group that includes addCourse, deleteCourse and enroll. The enroll method depends on both addCourse and registerStudent.

Effect of synchronization groups. To study the effect of synchronization groups, we compared Hamband and Mu on the movie use-case whose methods form two distinct synchronization groups. We perform experiments that execute 2, 4, and 8M update operations on four nodes. Fig. 3.9(a) and (b) show the throughput and response time respectively. We observe that the Hamband exhibits 1.4× to 1.8× higher throughput than Mu. This is due to the fact that Hamband is able to utilize two separate leaders to order requests while Mu uses a single leader. Hamband's throughput gain is close to the theoretical limit of 2×.



(a) Throughput **(b)** Response time

Figure 3.9: The effect of synchronization groups



(a) Throughput **(b)** Response time

Figure 3.10: Project management use-case

The difference in the response times is statistically negligible because the synchronization operations that a leader performs for a call is independent of the number of leaders.

Mix of categories. In this subsection we experiment with the project management database scheme that has methods in all the three categories. Fig. 3.10(a) compares the throughput of Hamband and Mu with 50%, 25%, and 10% update calls on four nodes. Hamband provides up to 21% higher throughput than Mu. Fig. 3.10(b) compares the

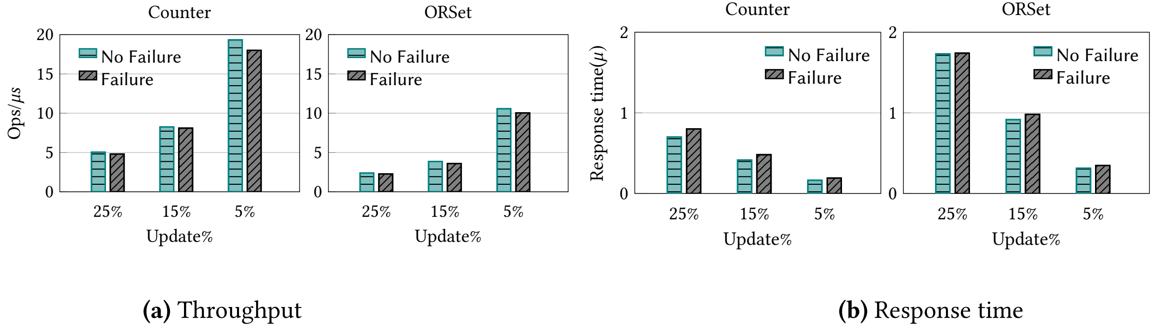
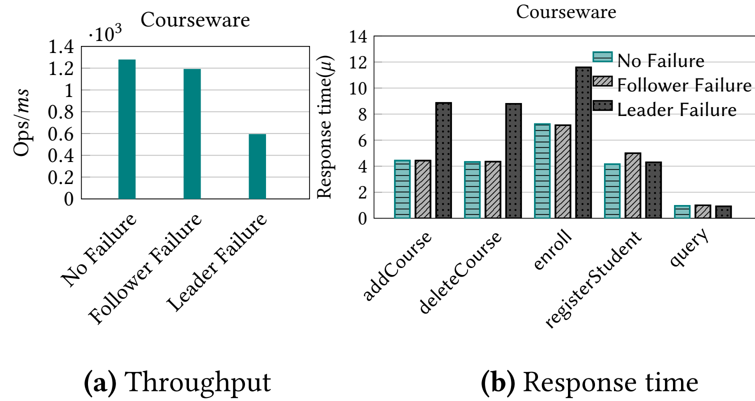


Figure 3.11: Effect of failure on the Counter and ORSet use-cases.

response times for each method. The response times for all methods except WorksOn stay almost the same. The response time for WorksOn calls is higher since they are dependent on addProject and addEmployee calls and have to wait for them to be delivered.

Fault tolerance. We now study the effect of failure on throughput and response time. All the failure experiments are done on 4 nodes. We first experiment on two CRDTs in Fig. 3.11 to study the effect of failure where there is no conflicting method. The methods of these use-cases are all in the two conflict-free categories. Therefore they use the reliable broadcast protocol or the single RDMA writes and do not use Mu. Moreover, we report results for the more elaborate courseware WRDT that has methods in all the three categories in Fig. 3.12. We inject failures into a node by suspending its heartbeat thread which make other nodes suspect that node. After a failure, all the requests of the failed node are redirected to the next available node. In the case of leader failure, the conflicting calls have to wait until the leader-change protocol elects the new leader.

Fig. 3.11(a) and (b) show the throughput and the response time of the Counter and ORSet respectively with different update ratios. We observe that the throughput of the



(a) Throughput **(b)** Response time

Figure 3.12: The effect of failure on the courseware use-case.

Counter and ORSet decrease by only an average of 5% and 5%, while the average response time increases by 15% and 5% respectively. Therefore, Hamband can smoothly withstand failures for conflict-free use-cases.

Fig. 3.12(a) and (b) show the throughput and average response time of the courseware use-case for three scenarios: the normal execution without failures as the baseline, failure of a follower, and finally, failure of the leader. Fig. 3.12(a) shows that Hamband can gracefully tolerate follower failure with only 6% impact on the throughput. However, since the leader change protocol is involved, when the leader of the synchronization group fails, the decrease in throughput is 53%. Fig. 3.12(b) shows the response time per method. The response time of the conflict-free registerStudent method experiences little to no change even in the leader failure scenario. This is because calls on this method do not need to be synchronized by the leader; therefore, they can be easily redirected to follower nodes without much impact on their response time. However, the response time of the conflicting methods such as addCourse, deleteCourse, and enroll almost doubles when the leader fails. They need to wait for the leader-change protocol to install the next leader.

3.6 Related Works

RDMA and hardware-aided replication. A few replication systems have been recently designed for RDMA [382, 467, 233, 17, 255] but they all implement an SMR and provide strong consistency. In contrast, this paper considers the semantics of methods, and avoids synchronization when possible.

DARE [382], the first RDMA-based SMR, presented a wait-free protocol that uses RDMA direct accesses and permissions, and applies it to implement a strongly consistent key-value store. Subsequently, APUS [467] improved the throughput of the SMR protocol. However, it showed higher response times, since the protocol requires the followers assist the leader during replication. Derecho [233], supports both an in-memory and a persistent SMR with high throughput. It uses an RDMA multicast protocol (RDMC) to move the data in high-rate flows, and uses a distributed shared memory (SST) to exchange control messages that determine when it is safe to deliver the data. Mu [17] reaches consensus with a single one-sided RDMA operation in the common case. It uses remote reads to detect failures and uses permissions to prevent concurrent leaders in the case of failure. Our synchronization mechanism for conflicting methods is similar. Hermes [255] uses logical timestamps to decentralize write operations, and locally establish a total order for a key-value store. Therefore, it is similar to the last-write-wins register CRDT that we implemented as well. In contrast, this paper offers general semantics and protocols for WRDTs that subsume CRDTs. Odyssey [178] presents a taxonomy and a comparison of these replicated systems.

Kite [179] adopts the release consistency (RC) model from shared-memory concurrency where threads use release and acquire synchronization primitives, and offers these

primitives in a high throughput key-value store abstraction (similar to a distributed shared memory). The key value store is implemented on top of eventually and strongly consistent protocols that benefit from RDMA acceleration, and provides the well-understood SC for DRF guarantee. On the other hand, GRAFStakes a high-level data type with no distribution details together with integrity properties. The convergence and integrity requirements lead to the inference of conflict and dependency relations between methods. According to these relations, GRAFScategorizes methods into three classes based on their coordination requirements that are separately and efficiently implemented on top of reliable and total-order broadcast protocols on RDMA. In particular, conflict-free methods calls can be executed under eventual instead of sequential consistency. Further, the reducible class of method calls can be implemented as single RDMA remote writes.

NetChain [238] uses programmable switches to store data and process queries in the network data plane. This eliminates the query processing at coordination servers and reduces the response time. HovercRaft [262] extends the Raft protocol to separate request replication from ordering, and integrates it with a transport protocol on a P4 [138] ASIC that supports load-balancing by updating the destination IP of RPC requests.

Hybrid replication models. Several projects have recently considered hybrid consistency models. However, all of them assumed the traditional message-passing network model; none addressed replication on the RDMA network model and its one-sided communication mechanism.

IPA [44] presents a static analysis the identifies the conflicting operations that can violate the integrity properties, and modifies them such that the invariants are maintained.

Sieve [301, 299, 300] applies static and dynamic analysis to determine whether an operation can be executed under causal consistency (blue class) or needs strong consistency (red class) in order to preserve the invariants. Quelea [431] and similarly the follow-up works [103, 59] define axiomatic semantics for consistency notions based on primitive consistency relations such as visibility and session orders. They capture user-defined consistency contracts for methods using the same primitives. They then automatically map a contract to the weakest consistency notion that satisfies the contract. Indigo [45, 46] captures invariants and post-conditions of methods in terms of user-defined predicates. It then identifies conflicting methods and either prevents or repairs their concurrent executions. CISE [189, 356] allows the user associate tags with methods and define conflicts between tags, and presents a rely-guarantee style proof technique for invariant preservation. Hamsaz [226] presents an axiomatic definition of well-coordination; in contrast, this paper presents an abstract operational semantics for general WRDTs, and further a concrete operational semantics for RDMA WRDTs, and proves a refinement between them. Carol [298] lets users declare required guard predicates on the current and remote view of the data, and automatically infers the required coordination. In order to reduce coordination, ECRO [133] reorders conflicting operations locally when possible.

Chapter 4

Graph Analytics Fusion and Synthesis

4.1 Introduction

Large-scale *graph analytics* has recently gained popularity due to its growing applicability across various important domains including social networks, market influencer analysis, bioinformatics, criminology, and machine learning and data mining. Several large-scale graph processing systems [188, 428, 508, 509, 330, 408, 505, 339, 462, 338] have been developed to enable efficient graph analysis across shared memory and distributed platforms. Their programming models often require graph analysis problems to be expressed in terms of low-level kernel functions over vertices and edges. However, analyses over graphs are best expressed using *higher-level abstractions* such as reduction over paths in the graph. For instance, shortest path, reachability and connected component problems are fundamentally formulated in terms of paths. Further, elaborate graph analysis problems that involve multiple reductions over paths or vertices are difficult to correctly implement using the offered low-level programming models. More importantly, manual *optimizations* such as

merging multiple iterations can be time-consuming and error-prone. In particular, showing *correctness and termination* properties requires reasoning about the flow of values between vertices across multiple iterations that emulate values for paths.

This project regards the interface of the graph processing frameworks as the instruction set for graph analytics, and introduces GraFS, a graph analytics language and synthesizer. The GraFS language is a *high-level declarative specification language* that provides features for common graph processing idioms such as reduction over paths. We show that the declarative language can easily and concisely capture the common graph analysis problems. Given a specification, the GraFS synthesizer *automatically synthesizes code* for five graph processing frameworks: Ligra [428], GridGraph [509], PowerGraph [188], Gemini [508], and GraphIt [505].

To synthesize efficient implementations, GraFS optimizes specifications by syntactic *fusion transformations* that fuse similar operations to be executed together. We formalize the syntax and the semantics of the GraFS language and the fusion rules, and prove that the fusion transformations are semantics-preserving. Effectively, fusion reduces specifications to the sequence of three primitives: reduction over paths, mapping over vertices and reduction over vertices.

Graph analytics frameworks offer *iterative programming models* to calculate reduction over paths. The values for vertices or edges are calculated iteratively based on the values of neighbors. Influenced by their runtime systems, these frameworks differ on how values are propagated between iterations. Some allow computations to both pull and push values to neighbors [188] whereas others only allow push [509] and others support a hybrid

[508, 428, 505]. Not only the propagation methods, but also *system-specific* nuances of the frameworks make their implementation of the same analysis problem subtly different. For example, they follow different protocols for atomicity of updates.

We formalize a comprehensive set of *iterative models* that given certain kernel functions, calculate path-based reductions. For each model, we present *correctness and termination conditions* on candidate kernel functions. Given a path-based reduction, the GraFS synthesizer enumerates candidate kernel functions and uses the correctness conditions as specifications to automatically *synthesize the kernel functions.* After fusion reduces specifications to the three primitives, reduction over paths, mapping over vertices and reduction over vertices, the synthesizer reduces reductions over paths to iterative calculations. Thus, graph analysis is reduced to *iteration-map-reduce primitives.* GraFS translates each of these primitives to implementations in each of the five target frameworks.

We apply GraFS to common graph analysis use-cases and generate code for each of the five frameworks. We note that graph processing frameworks often offer a more flexible and expressive API. However, we show that GraFS can express a large collection of common use-cases. The experimental results show that GraFS concisely captures use-cases, efficiently analyzes and synthesizes code, and its fusion brings up to 4× and in average 2.4× speedup.

In summary, this paper makes the following contributions. It provides the high-level declarative language GraFS and its semantics for large-scale graph analytics. It captures the iteration-map-reduce graph processing primitives that implement graph computations as structured let terms. GraFS presents semantics-preserving fusion transformations that are aware of these primitives: they fuse computation into and maintain this structure. This

paper formally models and proves the formal correctness and termination conditions for a comprehensive set of iterative models. Further, it combines type-directed enumerative synthesis and constrained-based synthesis to automatically synthesize the iterative kernel functions. The resulting tool can target five different graph processing frameworks and is evaluated on multiple standard benchmarks. The experiments show that fusion can accelerate execution. GraFS showcases that declarative languages and fusion transformations are effective for large-scale analytics.

In the following sections will present (1) Graph analytics specification language GraFS and its semantics (§ 4.3 and § 4.5.1), (2) Semantics-preserving and platform-independent fusion transformations (§ 4.5.2), (3) The formalization of iterative graph computation models (§ 4.4), their correctness and termination conditions (§ 4.6.1), and synthesis of their kernel functions (§ 4.6.2), and (4) The synthesis tool that generates code for five graph processing frameworks and its experimental results (§ 4.7).

4.2 Overview

Fig. 4.1 shows the overview of GraFS. The user writes her graph analytics as a declarative specification. Then, the fusion transformations optimize the input specification and translates it to iteration-map-reduce primitives. Subsequently, the synthesis process generates iterative kernel functions for the optimized specification. Finally, the code generation backend translates the kernels to five target graph processing frameworks. In this section, we present an example use-case in the GraFS specification language, and then show how that specification can be fused into an equivalent more efficient and canonical specification.

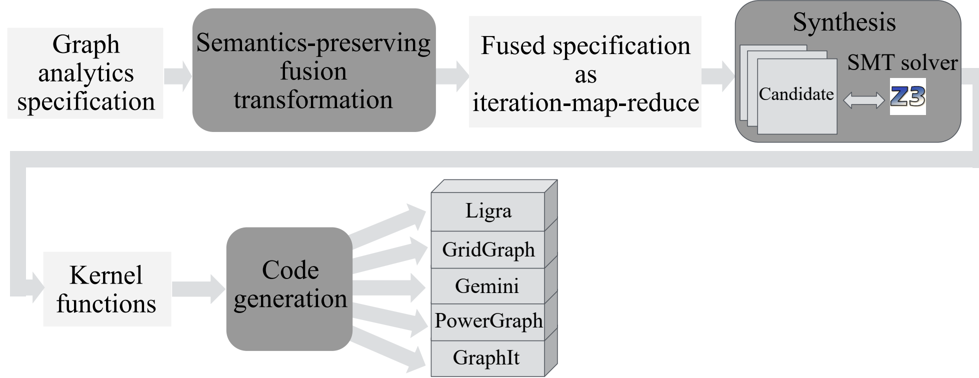


Figure 4.1: Workflow of GRAFS (Graph Analytics Fusion and Synthesis)

Then, we illustrate iterative reduction models (i.e., algorithms). Next, we consider the iteration-map-reduce primitives, and see both fused and unfused implementations of our use-case based on them. Finally, we see a glimpse of the correctness conditions of iterative reductions and how a synthesizer can use the conditions to generate the iterative kernel functions.

Specification. The GRAFS language allows declarative and concise specification of graph analysis computations. For example, Fig. 4.2, Eq. 4.1 represents the specification of the RADIUS use-case. The radius of a graph is the minimum eccentricity over its vertices. The eccentricity of a source vertex s is the longest of the shortest paths from s to any other vertex. The inner-most reduction of the RADIUS use-case is a *path-based reduction* that calculates the shortest path from a source vertex s to a destination vertex v. It applies the minimum reduction function min to the result of applying the weight function weight to all paths p in $\mathsf{Paths}(s, v)$, that is the set of paths from s to v. Then, it specifies the eccentricity of s as a nesting *vertex-based reduction* with the reduction function max to find the longest

of the shortest paths over all destination vertices v. Finally, it specifies the radius as the minimum of the eccentricity of the sample sources s_1 and s_2.

Fusion. A naive execution of specifications may execute path-based and vertex-based reductions multiple times. We show that multiple such reductions can be fused into a single reduction, and specifications can be represented as a common *triple-let form* with separate terms for path-based reduction, mapping over vertices and vertex-based reduction.

For example, the RADIUS use-case includes multiple path-based reductions one per source that can be fused together. Further, the path-based reductions are enclosed by vertex-based reductions that can be fused together as well. We illustrate this fusion in Fig. 4.2. We consider the fusion steps in turn. The specification of RADIUS is represented in Eq. 4.1. In Eq. 4.2, the outer min function over the two sources is unrolled. In Eq. 4.3, we restate each of the two reductions in a triple-let form. GRAFS features a triple-let term that separates path-based reductions, mapping over vertices and vertex-based reductions, and thus, facilitates fusion. The term $\max\limits_{v \in \mathsf{V}} \min\limits_{p \in \mathsf{Paths}(s_1, v)} \mathsf{length}(p)$ is rewritten as the following three lets. The first let, $\mathsf{ilet}\ x := \min\limits_{s_1} \mathsf{length}$, calculates a path-based reduction. For each vertex, it calculates the shortest length over the paths from the source s_1, and binds the result to x. The second let applies a map function in each vertex on the results of the path-based reduction. In this case, there is only one path-based reduction; therefore, the map function in the second let, $\mathsf{mlet}\ x' := x$, is simply the identity function, and the result is bound to x'. (In use-cases with an expression on multiple path-based reductions, the map in the second let captures the expression.) The third let calculates a reduction over all vertices. In this example, the third let, $\mathsf{rlet}\ x'' := \max\ x'$, calculates the maximum value over all vertices and

binds the result to x''. In Eq. 4.3, a similar transformation is applied for the other source s_2 as well.

Next, in Eq. 4.4, the two triple-let terms are fused into one by pairing the operations of the corresponding lets, and the outer min is applied to the two final results x'' and y''. In the next two steps, the paired path-based and vertex-based reductions are fused. In Eq. 4.5, the two path-based reductions of the first let, ilet, are fused into one. The fused reduction calculates the pair of the two values simultaneously. (The two sources s_1 and s_2 are used to initialize the first and second elements of the pairs respectively.) The fused path function $\mathcal{F}$ returns the pair of the results of the two path functions. Similarly, the fused reduction function R applies the two reduction functions to the first and second elements of the input pairs respectively. Finally, in Eq. 4.6, the pair of vertex-based reductions of the third let, rlet, are fused into one. The fused reduction function R' applies the two reduction functions to the first and second elements of the input pairs respectively. The original RADIUS specification executes two rounds of path-based and vertex-based reductions; however, the fused version computes one path-based and one vertex-based reduction on tuples of two elements at the same time. We will see the formal fusion rules including rules for more elaborate terms such as nested path-based reductions (§ 4.5.2 and § 4.5.3).

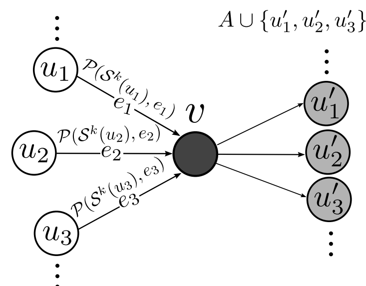
The final term represents the specification of RADIUS as an equivalent sequence of one path-based reduction, one map in each vertex, and one reduction over all vertices. We will next see that path-based reductions are calculated iteratively. Thus, fusion reduces GRAFS specifications to three primitives: *Iteration-Map-Reduce*: iteration for iterative path-based reduction, map for mapping over vertices and reduce for reduction over vertices. Map and

$$\begin{aligned}
\textsc{Radius} &= \min_{s \in \{s_1, s_2\}} \max_{v \in \mathsf{V}} \min_{p \in \mathsf{Paths}(s,v)} \mathsf{length}(p) && (4.1)\\
&= \min\left(\max_{v \in \mathsf{V}} \min_{p \in \mathsf{Paths}(s_1,v)} \mathsf{length}(p), \quad \max_{v \in \mathsf{V}} \min_{p \in \mathsf{Paths}(s_2,v)} \mathsf{length}(p) \right) && (4.2)\\
&= \min\left(\begin{pmatrix} \mathsf{ilet}\ x \coloneqq \min_{s_1}\ \mathsf{length}\ \mathsf{in} \\ \mathsf{mlet}\ x' \coloneqq x\ \mathsf{in} \\ \mathsf{rlet}\ x'' \coloneqq \max\ x'\ \mathsf{in} \\ x'' \end{pmatrix}, \begin{pmatrix} \mathsf{ilet}\ y \coloneqq \min_{s_2}\ \mathsf{length}\ \mathsf{in} \\ \mathsf{mlet}\ y' \coloneqq y\ \mathsf{in} \\ \mathsf{rlet}\ y'' \coloneqq \max\ y'\ \mathsf{in} \\ y'' \end{pmatrix} \right) && (4.3)\\
&= \begin{pmatrix} \mathsf{ilet}\ \langle x, y\rangle \coloneqq \langle \min_{s_1}\ \mathsf{length},\ \min_{s_2}\ \mathsf{length}\rangle\ \mathsf{in} \\ \mathsf{mlet}\ \langle x', y'\rangle \coloneqq \langle x, y\rangle\ \mathsf{in} \\ \mathsf{rlet}\ \langle x'', y''\rangle \coloneqq \langle \max\ x',\ \max\ y'\rangle\ \mathsf{in} \\ \min(x'', y'') \end{pmatrix} && (4.4)\\
&= \begin{pmatrix} \mathsf{ilet}\ \langle x, y\rangle \coloneqq \underset{\langle s_1, s_2\rangle}{\mathsf{R}}\ \mathcal{F}\ \mathsf{in} \\ \mathsf{mlet}\ \langle x', y'\rangle \coloneqq \langle x, y\rangle\ \mathsf{in} \\ \mathsf{rlet}\ \langle x'', y''\rangle \coloneqq \langle \max\ x',\ \max\ y'\rangle\ \mathsf{in} \\ \min(x'', y'') \end{pmatrix} \quad \text{where} \quad \begin{array}{l} \mathcal{F} \coloneqq \lambda p.\ \langle \mathsf{length}(p), \mathsf{length}(p)\rangle \\ \mathsf{R}\,(\langle a, b\rangle, \langle a', b'\rangle) \coloneqq \\ \quad \langle \min(a, a'), \min(b, b')\rangle \end{array} && (4.5)\\
&= \begin{pmatrix} \mathsf{ilet}\ \langle x, y\rangle \coloneqq \underset{\langle s_1, s_2\rangle}{\mathsf{R}}\ \mathcal{F}\ \mathsf{in} \\ \mathsf{mlet}\ \langle x', y'\rangle \coloneqq \langle x, y\rangle\ \mathsf{in} \\ \mathsf{rlet}\ \langle x'', y''\rangle \coloneqq \mathsf{R}'\ \langle x', y'\rangle\ \mathsf{in} \\ \min(x'', y'') \end{pmatrix} \quad \text{where} \quad \begin{array}{l} \mathsf{R}'\,(\langle a, b\rangle, \langle a', b'\rangle) \coloneqq \\ \quad \langle \max(a, a'), \max(b, b')\rangle \end{array} && (4.6)
\end{aligned}$$

Figure 4.2: Fusion of the Radius Use-case.

```
def Iteration-Pull (𝓘, 𝓟, R)
  foreach (v ∈ V)
    𝓢(v) ← 𝓘(v)
    if (𝓢(v) ≠ ⊥) A ← A ∪ {v}
  while (A ≠ ∅)
    foreach (v ∈ A)
      gv ← ⊥
      foreach (⟨u, v⟩ ∈ in-edges(v))
        gv ← R(gv, 𝓟(𝓢(u), ⟨u, v⟩)
      nv ← R(𝓢(v), gv)
      if (nv ≠ 𝓢(v))
        𝓢(v) ← nv
        foreach (⟨v, u′⟩ ∈ out-edges(v))
          A′ ← A′ ∪ {u′}
    A, A′ ← A′, ∅
```

(a)



(b)

```
def Map (f)
  foreach (v ∈ V)
    𝓢(v) ← f(𝓢(v)))
```

```
def Reduce (R)
  val ← ⊥
  foreach (v ∈ V)
    val ← R(val, 𝓢(v))
  return val
```

(c)

Figure 4.3: Pull Iterative Reduction.

reduce over vertices can be directly implemented; next, we consider iterative path-based reductions.

Iterative Path-based Reduction. Calculating path-based reductions by explicit enumeration of paths is prohibitively inefficient. Instead, path-based reductions are calculated iteratively by local updates on the value of vertices based on the values of their neighbors. As an example, we consider the pull iterative model for idempotent reduction

functions. Let us consider the shortest path use-case $\text{SSSP}(s)(v) = \min_{p \in \mathsf{Paths}(s,v)} \mathsf{weight}(p)$. It specifies a path-based reduction from the source s where the reduction function R is min and the path function $\mathcal{F}$ is weight. (We saw a similar reduction, the shortest length, as the innermost reduction of RADIUS.)

The pull-based iterative reduction is presented in Fig. 4.3a and illustrated in Fig. 4.3b. Each vertex stores a value $\mathcal{S}(v)$; we denote the value of a vertex v in the iteration k as $\mathcal{S}^k(v)$. The iterative calculation is based on three kernel function: the initialization function $\mathcal{I}$, the propagation function $\mathcal{P}$ and the reduction function R. The function $\mathcal{I}$ is a function from vertices to their initial value. For the SSSP use-case, $\mathcal{I}$ is $\lambda v.\ \mathsf{if}\ \ (v = s)\ 0\ \mathsf{else}\ \ \bot$ that initializes the value of the source s to zero (the some value of zero to be more precise) and the other vertices to none $\bot$. In each iteration, if the value of a vertex changes, its successors are added to the active set A for the next iteration. Fig. 4.3b shows the calculations for the active vertex v that is shown in black. In an iteration $k + 1$, an active vertex v pulls the value $\mathcal{S}^k(u)$ of each of its predecessors u. For each predecessor u, it applies the propagation function $\mathcal{P}$ to the value $\mathcal{S}^k(u)$ and the edge $\langle u, v \rangle$. For the SSSP use-case, the function $\mathcal{P}$ is $\lambda n, e.\ n + \mathsf{weight}(e)$ that adds the value of the predecessor to the weight of the edge from it. It then applies R (that is min in SSSP) to reduce the propagated values together and with the current value $\mathcal{S}^k(v)$ of v. The result is the new value $\mathcal{S}^{k+1}(v)$ of v. If the value of v changes, the successors of v that are marked as gray are active in the next iteration. The calculation stops when the values of vertices stay unchanged in two consecutive iterations. We will formalize a comprehensive set of iterative models (§ 4.4).

Iteration-Map-Reduce. After the fusion transformation, the specification reduces to *iteration-map-reduce primitives*: reduction over paths, mapping over vertices and reduction over vertices. We saw in Fig. 4.3a and b that reductions over paths can be computed using the iterative models. A sketch of both Map and Reduce operations is shown in Fig. 4.3c. The Map operation simply goes over all the vertices in the graph and applies the input function f to the value of each vertex. Similarly, the Reduce operation reduces vertex values with the input reduction function R. Fig. 4.4 shows sketches for both unfused and fused implementation of the RADIUS use-case. Fig. 4.4a shows the translation of the original unfused specification of the RADIUS use-case that we saw in Fig. 4.2, Eq. 4.1. In contrast, Fig. 4.4b shows the translation of the final fused specification of the RADIUS use-case that we saw in Fig. 4.2, Eq. 4.6. (We note that since the map is the identity function for the RADIUS use-case, it is elided in these implementations.) The unfused version in Fig. 4.4a executes two rounds of iteration-map-reduce. The first and the second rounds perform calculations for the first source s_1 and the second source s_2 respectively. Each round first performs an iteration to calculate the shortest path from the source to each vertex. Then, it calculates the eccentricity of the source by applying the max reduction function on the values of all vertices. Finally, the radius of the graph is minimum of the two eccentricity values. The fused version in Fig. 4.4b performs one round of iteration-map-reduce. The fused iteration stores and performs operations on a pair of values: shortest paths to each of the two sources. The initialization function initializes the values of the two sources, and the propagation and reduce functions propagate and reduce the shortest path values to them at the same time. Similarly, the subsequent reduction over all vertices calculates the eccentricity of the

Radius (Unfused):

$\mathsf{Iteration}(\lambda v.\ \mathsf{if}\ \ (v = s_1)\ 0\ \mathsf{else}\ \ \bot,$

$\quad \lambda n, e.\ n + 1,$

$\quad \min)$

$ecc_1 \leftarrow \mathsf{Reduce}(\max)$

$\mathsf{Iteration}(\lambda v.\ \mathsf{if}\ \ (v = s_2)\ 0\ \mathsf{else}\ \ \bot,$

$\quad \lambda n, e.\ n + 1,$

$\quad \min)$

$ecc_2 \leftarrow \mathsf{Reduce}(\max)$

$radius \leftarrow \min(ecc_1, ecc_2)$

(a)

Radius (Fused):

$\mathsf{Iteration}(\lambda v.\ \langle \mathsf{if}\ \ (v = s_1)\ 0\ \mathsf{else}\ \ \bot,$

$\quad \mathsf{if}\ \ (v = s_2)\ 0\ \mathsf{else}\ \ \bot \rangle,$

$\quad \lambda n, e.\ \langle n + 1,\ n + 1 \rangle,$

$\quad \lambda \langle a, b \rangle, \langle a', b' \rangle.$

$\quad \langle \min(a, a),\ \min(b, b') \rangle)$

$\langle ecc_1, ecc_2 \rangle \leftarrow \mathsf{Reduce}(\lambda \langle a, b \rangle, \langle a', b' \rangle.$

$\quad \langle \max(a, a'),\ \max(b, b') \rangle)$

$radius \leftarrow \min(ecc_1, ecc_2)$

(b)

Figure 4.4: Unfused and fused implementations of Radius as iteration-map-reduce rounds.

two sources at the same time. As we will see in the experiments (§ 4.7), this reduces the computation load by a factor of two.

Correctness and Synthesis. We formalize *correctness and termination conditions* for calculation of path-based reductions based on the iterative models. The conditions are parametric in terms of the kernel initialization and propagation functions and are used to automatically synthesize the kernel functions (§ 4.6.1 and § 4.6.2).

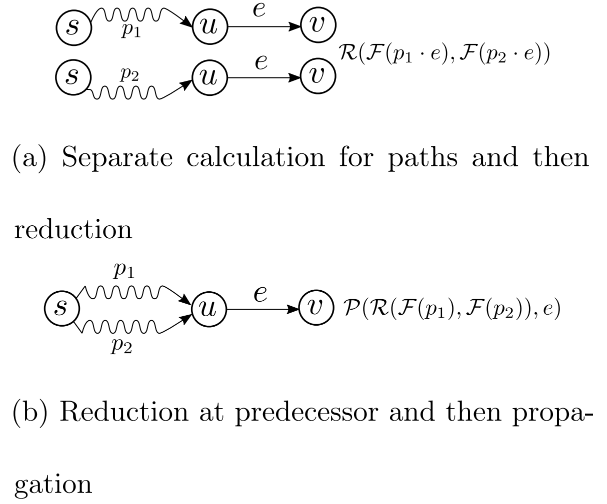
We present and prove sufficient conditions for a comprehensive set of iterative models (§ 4.6.1). As an example, we consider the pull model and illustrate one of the correctness conditions on the propagation function $\mathcal{P}$ in Fig. 4.5a and Fig. 4.5b. Consider a vertex v and a predecessor u of v. Consider calculating the reduction over all the paths

to v that go through u. Fig. 4.5a shows the direct calculation where the value of the path function for each path to v is separately calculated, and then the results are reduced. On the other hand, Fig. 4.5b shows a calculation using the propagation function $\mathcal{P}$ where first, the values of the path function for the paths to the predecessor u are calculated and reduced, and then, the result is propagated by $\mathcal{P}$ to v. In order to correctly calculate path-based reductions by local updates, the result of the above two calculations should be the same. Intuitively, local propagations from predecessors should be equivalent to global reductions over paths. Further, to reason about termination, we formalize the termination conditions for iterative models. Iterations incrementally consider longer paths. Cycles of a graph generate an infinite number of paths and can cause divergence. However, under certain conditions on the reduction and path functions R and $\mathcal{F}$, adding longer paths has no effect on the vertex values. For example, for the shortest path use-case SSSP (with non-negative edges), after a certain number of iterations, all the simple paths of the graph are already considered, and longer cyclic paths cannot improve the shortest path. Therefore, the calculation eventually terminates. We will see a formal definition of this condition and prove that it is sufficient for termination.

We use the correctness conditions to *synthesize correct kernel functions* (§ 4.6.2). In particular, we apply type-guided enumerative synthesis to find candidates, and automatic solvers to check the validity of the correctness conditions for each candidate. The result is correct-by-construction kernel functions that can iteratively calculate path-based reductions. We translate the synthesized functions to code in five high-performance graph processing frameworks (§ 4.7).

4.3 Declarative Graph Analytics

The GRAFS language declaratively and concisely captures *mathematical specifications of graph analysis computations*. The language design is guided by common idioms in graph processing use-cases. It supports reduction over values of paths to a vertex, and mapping and reduction over those values. Fig. 4.6 presents example use-cases.



(a) Separate calculation for paths and then reduction

(b) Reduction at predecessor and then propagation

Figure 4.5: The Correctness of the Pull Model. The path $p \cdot e$ denotes the extension of path p with edge e.

Path-based Reductions. The use-case SSSP specifies the weight of the shortest path from the source vertex s to each vertex v. The set of paths from a source vertex s to a destination vertex v is denoted by $\mathsf{Paths}(s, v)$. The specification applies the minimum reduction function min to the result of applying the weight function weight to all paths p in $\mathsf{Paths}(s, v)$. The specification of connected component (for undirected graphs) CC takes the smallest identifier of the vertices in a component as the representative identifier of that component. The set of all paths (from any source vertex) to a destination vertex v is denoted by $\mathsf{Paths}(v)$. The specification CC defines the connected component of each vertex v as the minimum identifier of the head vertices of the paths $\mathsf{Paths}(v)$. The above two specifications apply a reduction function R to the result of a path function $\mathcal{F}$ for a set of paths. We call these reductions *path-based reductions*. Similarly, the breadth-first-search use-case BFS calculates the parent for each

$$\mathrm{SSSP}(s)(v) = \min_{p \in \mathsf{Paths}(s,v)} \mathsf{weight}(p) \qquad \text{Single Source Shortest Path}$$

$$\mathsf{CC}(v) = \min_{p \in \mathsf{Paths}(v)} \mathsf{head}(p) \qquad \text{Connected Components}$$

$$\mathrm{BFS}(s)(v) = \mathsf{penultimate}(\operatorname*{arg\,min}_{p \in \mathsf{Paths}(s,v)} \mathsf{length}(p)) \qquad \text{Breadth-First Search}$$

$$\mathrm{WSP}(s)(v) = \mathsf{let}\ P \coloneqq \operatorname*{args\,min}_{p \in \mathsf{Paths}(s,v)} \mathsf{length}(p)\ \mathsf{in}\ \max_{p \in P} \mathsf{capacity}(p) \qquad \text{Widest Shortest Paths}$$

$$\mathrm{NSP}(s)(v) = \left| \operatorname*{args\,min}_{p \in \mathsf{Paths}(s,v)} \mathsf{weight}(p) \right| \qquad \text{Number of Shortest Paths}$$

$$\mathrm{NWR}(s)(v) = \frac{\min_{p \in \mathsf{Paths}(s,v)} \mathsf{capacity}(p)}{\max_{p \in \mathsf{Paths}(s,v)} \mathsf{capacity}(p)} \qquad \text{Narrowest to Widest Path Ratio}$$

$$\textsc{Trust}(v) = \max_{s \in \overline{s}} \left(\frac{\max_{p \in \mathsf{Paths}(s,v)} \mathsf{capacity}(p)}{\min_{p \in \mathsf{Paths}(s,v)} \mathsf{length}(p)} \right) \qquad \text{Trust from users } \{\overline{s}\}$$

$$\textsc{Radius} = \min_{s \in \{\overline{s}\}} \max_{v \in \mathsf{V}} \min_{p \in \mathsf{Paths}(s,v)} \mathsf{length}(p) \qquad \text{Radius Sampled on vertices } \{\overline{s}\}$$

$$\mathrm{DRR} = \frac{\max_{s \in \{\overline{v}\}} \max_{v \in \mathsf{V}} \min_{p \in \mathsf{Paths}(s,v)} \mathsf{length}(p)}{\min_{s \in \{\overline{v}\}} \max_{v \in \mathsf{V}} \min_{p \in \mathsf{Paths}(s,v)} \mathsf{length}(p)} \qquad \text{Diameter to Radius Ratio}$$

$$\mathrm{DS}(s) = \bigcup_{v \in \mathsf{V} \wedge \left(\min_{p \in \mathsf{Paths}(s,v)} \mathsf{weight}(p) \right) > 7} \{v\} \qquad \text{Vertices with minimum distance of 7}$$

$$\mathrm{L}\textsc{Trust}(s) = \mathsf{let}\ \mathrm{SSSP} \coloneqq \lambda s, v.\ \min_{p \in \mathsf{Paths}(s,v)} \mathsf{weight}(p)\ \mathsf{in}$$
$$\mathsf{let}\ \mathrm{WP} \coloneqq \lambda s, v.\ \max_{p \in \mathsf{Paths}(s,v)} \mathsf{capacity}(p)\ \mathsf{in}$$
$$\min_{v \in \mathsf{V} \wedge \mathrm{SSSP}(s,v) < \textsc{Radius}} \mathrm{WP}(s,v) \qquad \text{Least trust in the radius}$$

Figure 4.6: A Subset of Use-cases in GraFS

vertex in the BFS tree rooted at a source vertex s. For each vertex v, it specifies a path-based reduction to find the shortest-length path from s to v, and returns the penultimate of that path. The penultimate of a path is the vertex before the last in the path. The specification uses the reduction function $\arg\min$ to get the path with the minimum length rather than the minimum length itself, and then applies the $\mathsf{penultimate}$ function to the path. (A simpler specification can simply apply $\min$ instead of $\arg\min$ and return the minimum path length, i.e., the depth of the vertex in the breadth-first-search tree.)

Nested Path-based Reductions. Path-based reductions can be nested. The use-case WSP specifies the widest shortest path from a source s to each vertex v. We use the let syntactic sugar to enhance readability. WSP has a nested reduction (with the reduction function $\mathrm{args}\min$) to find the shortest paths, and then a nesting reduction to find the widest capacity in those paths. WSP is used as a metric of the trust of a user to other users in social networks where the capacity of each edge is the local trust rating of the source user to the sink user [186]. Intuitively, users with wider (stronger trust ratings) and shorter (closer) paths are more trustworthy sources of information. Similarly, the use-case NSP specifies the number of shortest paths from a source s to each vertex v. It uses a nested reduction to find the shortest paths and then applies the cardinality operator to the resulting set. (We will see in § 4.5.3 that cardinality is a syntactic sugar for a path-based reduction with the sum $\sum$ function.)

Mapping over Vertices. Mathematical operators can be applied to path-based reductions. The use-case NWR specifies the narrowest to widest path ratio from a source to each vertex. At each vertex, it divides the result of two path-based reductions. Similarly,

the use-case Trust is the result of division and maximum operations between path-based reductions. It specifies the trust from a set of users $\{\overline{s}\}$ to each other user v. As before, wider and shorter paths are favored.

Vertex-based Reductions. The values of vertices calculated by a path-based reduction can be subsequently reduced by a *vertex-based reduction*. For example, the eccentricity of a source vertex s is the longest of the shortest paths from s to any other vertex. The radius of a graph is the minimum eccentricity over its vertices. The Radius use-case specifies eccentricity as a vertex-based reduction with the reduction function max to find the longest of the shortest paths over all vertices. It then specifies the radius as the minimum of the eccentricity of a set of sample sources $\{\overline{s}\}$. Similar to path-based reductions, mathematical operators can be applied to vertex-based reductions. As the set of sampled sources $\{\overline{s}\}$ is finite, the outer min function can be unrolled to an infix operator between vertex-based reductions. Similarly, the use-case DRR, that is the ratio of the diameter over the radius of the graph, is specified as maximum and minimum operations between vertex-based reductions, and a subsequent division.

The use-case DS specifies the set of vertices with the distance of at least 7 from the source s. The union $\cup$ vertex-based reduction is used to calculate the set. The set of vertices that it is applied to are constrained by a nested path-based reduction to specify the distance. (In § 4.5.3, we show that *constrained vertex-based reductions* can be desugared to standard vertex-based reductions that are applied to path-based reductions on pairs of values.) The next use-case, LTrust, represents a measure of the least amount of trust from a user to her neighbourhood in a social network. Similar to DS, the use-case LTrust is

specified as a constrained vertex-based reduction. Given a source s, it calculates the widest path to each vertex within the radius of s (i.e., k-hop neighbourhood of s where k is the radius of the graph), and then returns the narrowest of those.

4.4 Iterative Models

We formalize four canonical models for iterative graph computations: the pull and push models with idempotent and non-idempotent reduction. Graph computation frameworks [188, 428, 508, 509, 330, 408] implement variants of these models. Later in § 4.6, we use these models to implement path-based reductions and present their correctness conditions.

In these models, each vertex is first initialized. Then, the value of each vertex is iteratively updated based on the values of its predecessors. In each iteration, the vertex pulls the values of its predecessors or each predecessor pushes its value to the vertex. Then, the values of the predecessors and the current value of the vertex are reduced to calculate the new value of the vertex. Before assigning the reduced value to the vertex, a final function may be applied to it. The iteration stops when the value of no vertex changes. The models are parametrized by four kernel functions: $\mathcal{I}$, $\mathcal{P}$, R and $\mathcal{E}$. The *initialization* function $\mathcal{I}$ defines the initial value for each vertex. The *propagation* function $\mathcal{P}$, given a value n and an edge $\langle u, v \rangle$ where n is the value of u, defines the value that is propagated to v. The commutative and associative *reduction* function R defines how the propagated values are aggregated. The *epilogue* function $\mathcal{E}$ defines the final update.

We present a high-level language to specify the kernel functions. We compile kernels specified in this language to executable programs in five graph processing frameworks. The grammar for the bodies of the kernel functions is presented in Fig. 4.7a. (Later in § 4.6.2, the same grammar is used by the synthesis process; given higher-level specifications, it automatically generates the kernel functions in this language.) Fig. 4.7b shows the iterative kernel functions for two example use-cases: the shortest path SSSP and the page-rank PageRank (PR). For the shortest path SSSP use-case, the initialization function $\mathcal{I}$ initializes the source vertex s to (some value of) 0 and the other vertices to none $\bot$. The propagation function $\mathcal{P}$ adds the value v of the predecessor to the weight of the edge e. The reduction function R is the minimum (that is idempotent) and the epilogue function $\mathcal{E}$ is the identity function. For the page-rank use-case PR, $\mathcal{I}$ divides the value 1 between the number of vertices $|V|$. The function $\mathcal{P}$ divides the value v of the predecessor between its successors. The function R is sum (that is non-idempotent). The function $\mathcal{E}$ multiplies the sum with the damping factor γ and adds a constant.

Pull Model. The characteristic of the pull model is that vertices pull the values of their predecessors to calculate their new values. We consider the pull model for idempotent and non-idempotent reduction functions in turn.

Pull model with idempotent reduction ($\mathsf{pull+}$*).* The pull model for idempotent reduction is represented in Fig. 4.8, Theorem 29. The value of the vertex v in the iteration k is represented as $\mathcal{S}^k_{\mathsf{pull+}}(v)$. In the beginning when $k = 0$, vertices have no value $\bot$. In the first iteration $k = 1$, they are initialized by the initialization function $\mathcal{I}$. In subsequent iterations $k + 1,\ k \geq 1$, each vertex v pulls values of its predecessors. For each predecessor u,

$$
\begin{array}{lcll}
e & ::= & n \mid v & \text{Body Exp} \\
 & \mid & e + e \mid e - e \mid -e \mid \langle e, e \rangle & \\
 & \mid & e \times e \mid e \,/\, e \mid e = e \mid e < e & \\
 & \mid & \min(e, e) \mid \max(e, e) & \\
 & \mid & \mathsf{if}\ (e)\ \mathsf{then}\ e\ \mathsf{else}\ e & \\
 & \mid & \mathsf{weight}(e) \mid \mathsf{capacity}(e) & \\
 & \mid & \mathsf{indeg}(e) \mid \mathsf{outdeg}(e) & \\
 & \mid & \mathsf{src}(e) \mid \mathsf{dst}(e) & \\
 & \mid & |V| & \text{Graph Order} \\
n & ::= & 0 \mid 1 \mid .. \mid \mathbb{T} \mid \mathsf{False} & \text{Literal} \\
v & & & \text{Variable}
\end{array}
$$

(a) Grammar

SSSP

$$
\begin{array}{lcl}
\mathcal{I} & := & \lambda v.\ \mathsf{if}\ (v = s)\ 0\ \mathsf{else}\ \bot \\
\mathcal{P} & := & \lambda v, e.\ v + \mathsf{weight}(e) \\
\mathsf{R} & := & \lambda v, v'.\ \min(v, v') \\
\mathcal{E} & := & \lambda v.\ v
\end{array}
$$

PageRank (PR)

$$
\begin{array}{lcl}
\mathcal{I} & := & \lambda v.\ 1 \,/\, |V| \\
\mathcal{P} & := & \lambda v, e.\ v \,/\, \mathsf{outdeg}(\mathsf{src}(e)) \\
\mathsf{R} & := & \lambda v, v'.\ v + v' \\
\mathcal{E} & := & \lambda v.\ \gamma \times v \ + \ (1 - \gamma) \,/\, |V|
\end{array}
$$

(b) Examples

Figure 4.7: (a) Grammar for Kernel Functions (b) Example Kernel Functions. (min and + filter none values $\bot$.)

the propagation function $\mathcal{P}$ is applied to the value $\mathcal{S}^k_{\mathsf{pull+}}(u)$ of u (from the previous iteration k) and the connecting edge $\langle u, v \rangle$. Then, as illustrated in Fig. 4.3b, all the propagated values are reduced by R with each other and then with the previous value $\mathcal{S}^k_{\mathsf{pull+}}(v)$ of v. Finally, applying the epilogue function to the reduced value results in the new value $\mathcal{S}^{k+1}_{\mathsf{pull+}}(v)$ of v. As an optimization, the above update is performed only if the set of predecessors of v whose value have changed in the previous iteration $\mathsf{CPreds}^k(v)$ is non-empty.

Pull model with non-idempotent reduction ($\mathsf{pull-}$*).* The pull model for non-idempotent reduction is represented in Fig. 4.8, Theorem 30. The value of the vertex v in the iteration k is represented as $\mathcal{S}^k_{\mathsf{pull-}}(v)$. Similar to the previous model, the values

from predecessors are propagated and reduced. The difference is that after reducing the propagated values, the result is not reduced with the previous value of the vertex. The reason is to avoid duplicate reduction with the non-idempotent reduction function. Consider a vertex v and a predecessor u of v. Assume that the value of u represents the reduction of a set S of values. After the value of u is propagated to v, the value of v includes the reduced and propagated values of S. Assume that the value of u is updated again to represent the reduction of more values. If the new value of u is propagated to v and reduced with the current value of v, then the set S is included in the value of v twice.

Push Model. In the pull model above, each vertex itself pulls values from its predecessors. In contrast, in the push model, the predecessors push values to the vertex when they are updated. We consider the push model for idempotent and non-idempotent reduction functions in turn.

*Push model with idempotent reduction (*push+*).* The push model for idempotent reduction is represented in Fig. 4.8, Theorem 31. The value of the vertex v in the iteration k is represented as $\mathcal{S}^k_{\mathsf{push+}}(v)$. The iterations 0 and 1 are similar to the previous models. In subsequent iterations $k+1,\ k \geq 1$, for each vertex v, the predecessors $\{u_0, .., u_{n-1}\}$ that have been changed in the previous iteration independently propagate their values and reduce it with the current value of v. Since the reduction function is commutative and associative, the predecessors can apply their updates in any order. In each iteration, the initial value S_0 of v is its value in the previous iteration k. For each changed predecessor u_i, the propagation function $\mathcal{P}$ is applied to the value $\mathcal{S}^k_{\mathsf{push+}}(u_i)$ of u_i (from the previous iteration k) and the connecting edge $\langle u_i, v\rangle$. The result is then reduced with the current value S_i of v to calculate

$$\mathsf{CPreds}^k(v) = \big\{u \mid u \in \mathsf{preds}(v) \wedge \mathcal{S}^k(u) \neq \mathcal{S}^{k-1}(u)\big\}$$

Definition 29 (Pull (idempotent))

$$\mathcal{S}^0_{\mathsf{pull}+}(v) \coloneqq \bot$$

$$\mathcal{S}^1_{\mathsf{pull}+}(v) \coloneqq \mathcal{I}(v)$$

$$\mathcal{S}^{k+1}_{\mathsf{pull}+}(v) \coloneqq \begin{cases} \mathcal{S}^k_{\mathsf{pull}+}(v) & \textit{if } \mathsf{CPreds}^k(v) = \emptyset \\ \mathcal{E}\left[\mathsf{R}\Big(\mathcal{S}^k_{\mathsf{pull}+}(v),\ \mathsf{R}_{u \in \mathsf{preds}(v)}\, \mathcal{P}\Big(\mathcal{S}^k_{\mathsf{pull}+}(u), \langle u, v\rangle\Big)\Big)\right] & \textit{else} \end{cases} \qquad k \geq 1$$

Definition 30 (Pull (non-idempotent))

$$\mathcal{S}^0_{\mathsf{pull}-}(v) \coloneqq \bot$$

$$\mathcal{S}^1_{\mathsf{pull}-}(v) \coloneqq \mathcal{I}(v)$$

$$\mathcal{S}^{k+1}_{\mathsf{pull}-}(v) \coloneqq \begin{cases} \mathcal{S}^k_{\mathsf{pull}-}(v) & \textit{if } \mathsf{CPreds}^k(v) = \emptyset \\ \mathcal{E}\left[\mathsf{R}_{u \in \mathsf{preds}(v)}\, \mathcal{P}\Big(\mathcal{S}^k_{\mathsf{pull}-}(u), \langle u, v\rangle\Big)\right] & \textit{else} \end{cases} \qquad k \geq 1$$

Definition 31 (Push (idempotent))

$$\mathcal{S}^0_{\mathsf{push}+}(v) \coloneqq \bot$$

$$\mathcal{S}^1_{\mathsf{push}+}(v) \coloneqq \mathcal{I}(v)$$

$$\mathcal{S}^{k+1}_{\mathsf{push}+}(v) \coloneqq \mathcal{E}(S_n), \quad k \geq 1 \quad \textit{where}$$

$$\mathsf{let}\ \{u_0, .., u_{n-1}\} \coloneqq \mathsf{CPreds}^k(v)\ \mathsf{in}$$

$$S_0 \coloneqq \mathcal{S}^k_{\mathsf{push}+}(v)$$

$$S_{i+1} \coloneqq \mathsf{R}\Big(S_i,\ \mathcal{P}\Big(\mathcal{S}^k_{\mathsf{push}+}(u_i), \langle u_i, v\rangle\Big)\Big)$$

Definition 32 (Push (non-idempotent))

$$\mathcal{S}^0_{\mathsf{push}-}(v) \coloneqq \bot$$

$$\mathcal{S}^1_{\mathsf{push}-}(v) \coloneqq \mathcal{I}(v)$$

$$\mathcal{S}^{k+1}_{\mathsf{push}-}(v) \coloneqq \mathcal{E}(S_n), \quad k \geq 1 \quad \textit{where}$$

$$\mathsf{let}\ \{u_0, .., u_{n-1}\} \coloneqq \mathsf{preds}(v)\ \mathsf{in}$$

$$S_0 \coloneqq \bot$$

$$S_{i+1} \coloneqq \mathsf{R}\Big(S_i,\ \mathcal{P}\Big(\mathcal{S}^k_{\mathsf{push}-}(u_i), \langle u_i, v\rangle\Big)\Big)$$

Figure 4.8: Four Iterative Reduction Methods. $\mathsf{CPreds}^k(v)$: The predecessors of the vertex v that changed in the iteration k

its new value S_{i+1}. Propagation and reduction by the last changed predecessor u_{n-1} results in the value S_n. The final value of v is the result of applying the epilogue $\mathcal{E}$ to S_n.

Push model with non-idempotent reduction (push−). This model works for non-idempotent (in addition to idempotent) reduction functions. The iterative model is represented in Fig. 4.8, Theorem 32. Let the value of the vertex v in the iteration k be represented as $\mathcal{S}^k_{\mathsf{push}-}(v)$. Since the reduction function may not be idempotent, in contrast to the previous model, vertices start from the none value $S_0 = \bot$, and all the predecessors u_i propagate their values in each iteration. For each predecessor u_i, the propagate function $\mathcal{P}$ is applied to the latest value $\mathcal{S}^k_{\mathsf{push}-}(u_i)$ of u_i and the connecting edge $\langle u_i, v\rangle$. The resulting value is reduced with the current value S_i of v. We note that this variant makes all vertices active during an iteration; GraFS also incorporates another variant where only the vertices whose values change are active and propagate their values. In this variant, an active predecessor u_i first rollbacks its previous update before applying its new update.

The iterative models that we saw here are *synchronous*. In the synchronous model, vertices store their previous in addition to their new value to propagate their previous value. In the *asynchronous* model, however, each vertex stores one value, and vertices can propagate intermediate values.

4.5 Specification and Fusion

In this section, we define the core specification language, its denotational semantics, and the semantics-preserving fusion transformations.

$$
\begin{array}{rcll}
r & := & \underset{\mathsf{v}}{\mathsf{R}}\; m \mid & \text{Vertex-based Red.} \\
 & & r \oplus r \mid x & \\
 & & \mathsf{ilet}\ X := M\ \mathsf{in} & \\
 & & \mathsf{mlet}\ X := E\ \mathsf{in} & \\
 & & \mathsf{rlet}\ X := R\ \mathsf{in}\ e & \\
m & := & \underset{p \in P}{\mathsf{R}}\, \mathcal{F}(p) \mid & \text{Path-based Red.} \\
 & & m \oplus m \mid x \mid & \\
 & & \mathsf{ilet}\ X := M\ \mathsf{in}\ e & \\
P & := & \mathsf{Paths} \mid \underset{p \in P}{\mathsf{argsR}}\ \mathcal{F}(p) & \text{Paths} \\
\mathsf{R} & := & \min \mid \max \mid & \text{Reduction Fun.} \\
 & & \vee \mid \wedge \mid \sum & \\
\mathcal{F} & := & \mathsf{length} \mid \mathsf{weight} \mid & \text{Path Fun.} \\
 & & \mathsf{capacity} & \\
\oplus & := & \min \mid \max \mid & \text{Operation} \\
 & & \wedge \mid \vee \mid & \\
 & & + \mid - \mid \times \mid / \mid & \\
 & & = \mid < \mid > & \\
p & & & \text{Path Variable} \\
x & & & \text{Variable} \\
e & := & e \oplus e \mid x &
\end{array}
$$

$$
\begin{array}{rcll}
X & := & \langle X, X \rangle \mid x & \\
M & := & \langle M, M \rangle \mid \mathsf{R}\ \mathcal{F} & \\
R & := & \langle R, R \rangle \mid \mathsf{R}\ \langle \overline{x} \rangle \mid \mathsf{R}\ \langle \overline{d} \rangle & \\
E & := & \langle E, E \rangle \mid e & \\
\mathbb{R} & := & [\,] \mid \mathbb{R} \oplus r \mid r \oplus \mathbb{R} & r\ \text{Context} \\
\mathbb{M} & := & [\,] \mid \underset{\mathsf{v}}{\mathsf{R}}\ \mathbb{M} \mid & m\ \text{Context} \\
 & & \mathbb{M} \oplus m \mid m \oplus \mathbb{M} & \\
\mathbb{M}\mathsf{s} & := & [\,] \mid \langle \mathbb{M}\mathsf{s}, M \rangle \mid \langle M, \mathbb{M}\mathsf{s} \rangle \mid & M\ \text{Context} \\
 & & \mathsf{ilet}\ X := \mathbb{M}\mathsf{s}\ \mathsf{in}\ e \mid & \\
 & & \mathsf{ilet}\ X := \mathbb{M}\mathsf{s}\ \mathsf{in} & \\
 & & \mathsf{mlet}\ X := E\ \mathsf{in} & \\
 & & \mathsf{rlet}\ X := R\ \mathsf{in}\ e & \\
\mathbb{R}\mathsf{s} & := & [\,] \mid \langle \mathbb{R}\mathsf{s}, R \rangle \mid \langle R, \mathbb{R}\mathsf{s} \rangle \mid & R\ \text{Context} \\
 & & \mathsf{ilet}\ X := M\ \mathsf{in} & \\
 & & \mathsf{mlet}\ X := E\ \mathsf{in} & \\
 & & \mathsf{rlet}\ X := \mathbb{R}\mathsf{s}\ \mathsf{in}\ e & \\
\mathsf{n} & & & \text{Value} \\
\mathsf{v} & & & \text{Vertex Value} \\
d : \mathcal{D}_m & := & (\mathsf{V}(g) \mapsto \mathbb{N}) \cup \{\bot\} & m\ \text{Sem. Dom.} \\
\mathsf{n}_\bot : \mathcal{D}_r & := & \mathbb{N} \cup \{\bot\} & r\ \text{Sem. Dom.}
\end{array}
$$

Figure 4.9: Core Specification Language

4.5.1 Core Specification Language

To present the crux of the fusion transformations, we define a core specification language in Fig. 5.3. It features both reduction over paths and reduction over vertices. A computation can be specified as a reduction r over the values of vertices. The value of vertices, in turn, can be specified as a nested reduction m over the paths to each vertex. More elaborate computations can be specified by nested path-based computations, and applying operations between multiple path-based and vertex-based computations. We will visit each term type in turn.

Vertex-based and path-based reductions. A vertex-based reduction $\underset{\mathsf{V}}{\mathsf{R}}\ m$ applies a reduction function R to the result of path-based reductions m over all vertices V. The function R is a commutative and associative function such as min, max, $\vee$, $\wedge$ and $\sum$. Larger vertex-based reductions $r \oplus r'$ can be constructed using the operators $\oplus$. A path-based reduction $\underset{p \in P}{\mathsf{R}} \mathcal{F}(p)$ applies a reduction function R to the results of applying the function $\mathcal{F}$ to each path p in set of paths P. Similar to vertex-based reductions, larger path-based reductions $m \oplus m'$ can be constructed using the operators $\oplus$. The path function $\mathcal{F}$ is the length, weight, or $\mathsf{capacity}$ of the path. The set of paths P can be either Paths that denotes all the paths to each vertex, or the restricted paths $\underset{p \in P}{\mathsf{args}\,\mathsf{R}}\ \mathcal{F}(p)$ where $\mathsf{R} \in \{\min, \max\}$ that denotes the paths in P whose $\mathcal{F}$ value is the extremum. (The r and m terms can be also variables x that can be substituted with a value n or a map value d from vertices to values.)

Let forms. Let terms factor different reductions. Factored reductions are conducive to fusion. As shown in Fig. 5.3, the terms m and r both have let forms. The m term constructor $\mathsf{ilet}\ X := M\ \mathsf{in}\ e$ binds variables X to factored path-based reductions

M for the expression e. The expression e can apply operators $\oplus$ to the variables X. Both the variables X and reductions M can be inductively constructed as pairs. A single path-based reduction M is simply represented as $\mathsf{R}\,\mathcal{F}$ where R is the reduction function and $\mathcal{F}$ is the path function. Similarly, the triple-let r constructor $\mathsf{ilet}\ X \coloneqq M\ \mathsf{in}\ \ \mathsf{mlet}\ X' \coloneqq E\ \mathsf{in}$ $\mathsf{rlet}\ X'' \coloneqq R\ \mathsf{in}\ e$ binds variables X to factored path-based reductions M, binds variables X' to expressions E (on X), and binds variables X'' to factored vertex-based reductions R (on X'). A triple-let term represents an r term as a sequence: path-based reductions, mappings on the results, and finally vertex-based reductions on the results. We will see that this form enables fusion (§ 4.5.2) and can be directly implemented (§ 4.7). Similar to M, the factored vertex-based reductions R can be inductively constructed as pairs. A single vertex-based reduction R is $\mathsf{R}\ \langle\overline{x}\rangle$ that is a reduction over tuples of variables $\langle\overline{x}\rangle$ (or is $\mathsf{R}\ \langle\overline{d}\rangle$ after the variables are substituted with map values d from vertices to values). To concisely represent the fusion rules, we define the context $\mathbb{R}$ to abstract the surrounding term where a term r appears. Similarly, we define the contexts $\mathbb{M}$, $\mathbb{M}\mathsf{s}$, and $\mathbb{R}\mathsf{s}$ for the terms m, M and R.

Semantics and Compositionality. We define the denotational semantics of the specification language (presented in Fig. 5.3). For brevity, we showcase the semantics $[\![\ ]\!]$ of a subset of the term constructors in Fig. 4.10. The semantics of an undefined or stuck computation is represented as $\perp$. In each rule, it is assumed that the semantics of subterms are not undefined; otherwise, the semantics of the whole term is undefined as well. The domain $\mathcal{D}_m$ of a path-based computation m on a graph g is a finite map from each vertex of g to natural numbers $\mathsf{V}(g) \mapsto \mathbb{N}$, and $\perp$ (for undefined computation). The domain $\mathcal{D}_r$ of a

SPRed

$$\left[\!\!\left[\underset{p \in P}{\mathsf{R}} \mathcal{F}(p) \right]\!\!\right](g) \ = \ \overline{[\mathsf{v} \mapsto \mathsf{R}\ \{\mathcal{F}(p) \mid p \in [\![P]\!](g)(\mathsf{v})\}]}_{\mathsf{v} \in \mathsf{V}(g)}$$

SMBin

$$[\![m \oplus m']\!](g) \ = \ [\![m]\!](g) \ \oplus \ [\![m']\!](g)$$

SVRed

$$\left[\!\!\left[\underset{\mathsf{V}}{\mathsf{R}}\ m \right]\!\!\right](g) \ = \ \mathsf{R}\ \left\{ \overline{[\![m]\!](g)(\mathsf{v})}\ _{\mathsf{v} \in \mathsf{V}(g)} \right\}$$

SRLet

$$\left[\!\!\left[\begin{array}{l} \mathsf{ilet}\ X := M\ \mathsf{in} \\ \mathsf{mlet}\ X' := E\ \mathsf{in} \\ \mathsf{rlet}\ X'' := R\ \mathsf{in} \\ e \end{array} \right]\!\!\right](g) \ = \ [\![e\,[X'' := [\![R\,[X' := [\![E\,[X := [\![M]\!](g)]]\!]]]\!](g)]]\!]$$

SMPair

$$[\![\langle M,\ M' \rangle]\!](g) = \langle [\![M]\!](g),\ [\![M']\!](g) \rangle$$

SRPair

$$[\![\langle R,\ R' \rangle]\!](g) = \langle [\![R]\!](g),\ [\![R']\!](g) \rangle$$

SMM

$$[\![\mathsf{R}\ \mathcal{F}]\!] = \left[\!\!\left[\underset{p \in \mathsf{Paths}}{\mathsf{R}} \mathcal{F}(p) \right]\!\!\right]$$

SRR

$$\left[\!\!\left[\mathsf{R}\ \left\langle \overline{[\mathsf{v} \mapsto n_{\mathsf{v}}]}_{\mathsf{v} \in \mathsf{V}(g)}, .., \overline{[\mathsf{v} \mapsto n'_{\mathsf{v}}]}_{\mathsf{v} \in \mathsf{V}(g)} \right\rangle \right]\!\!\right] = \left[\!\!\left[\underset{\mathsf{V}}{\mathsf{R}}\ \left(\overline{[\mathsf{v} \mapsto \langle n_{\mathsf{v}},\ ..,\ n'_{\mathsf{v}} \rangle]}_{\mathsf{v} \in \mathsf{V}(g)} \right) \right]\!\!\right]$$

Figure 4.10: Denotational Semantics of the Specification Language

vertex-based computation r is the natural numbers $\mathbb{N}$ and $\perp$. We use the notation $\overline{[k_i \mapsto v_i]}_i$ for a finite map that maps each key k_i to value v_i over the range i.

The rule SPRed defines the semantics of the path-based reduction $\underset{p \in P}{\mathsf{R}} \mathcal{F}(p)$. It uses the (elided) semantics of paths P that is a map from each vertex v to the set of paths to v. For each vertex v, the rule SPRed applies the function $\mathcal{F}$ to each path to v, and then applies the reduction function R to the resulting values. (Since the reduction functions R are commutative and associative, they can be applied to the values in any order.) The rule SMBin defines the semantics of $m \oplus m'$ as the result of the operator $\oplus$ on the semantics of m and m'. The operators R and $\oplus$ are in the syntactic and semantic domains when they are

on the left- and right-hand side of the rules respectively. The operator $\oplus$ is simply lifted to maps by pointwise application to the values of each key.

The rule SVRED defines the semantics of the vertex-based reduction $\underset{\mathsf{V}}{\mathsf{R}}\ m$ using the map resulted from the semantics of m; it reduces the values of the map for all vertices. The rule SRLET defines the semantics of triple-let terms by three subsequent substitutions: the substitution of the variables X with the semantics of M in E, the substitution of X' with the semantics of E in R, and finally the substitution of X'' with the semantics of R in e. (The semantics of e and E are elided.)

The rules SMPAIR and SRPAIR define the semantics of pairs of factored reductions M and R inductively. The two rules SMM and SRR reduce the semantics of single factored reductions to expanded reductions. The rule SMM defines the semantics of the factored path-based reduction $\mathsf{R}\ \mathcal{F}$ as a path-based reduction on the paths Paths. The rule SRR defines the semantics of a factored vertex-based reduction. It merges the tuple of the factored maps $\overline{[\mathsf{v} \mapsto n_{\mathsf{v}}]}_{\mathsf{v} \in \mathsf{V}(g)}, .., \overline{[\mathsf{v} \mapsto n'_{\mathsf{v}}]}_{\mathsf{v} \in \mathsf{V}(g)}$ into a map from vertices v to tuples $\langle n_{\mathsf{v}}, .., n'_{\mathsf{v}} \rangle$, and then applies the vertex-based reduction.

We prove that the semantics is compositional. If two terms are semantically equivalent, replacing one with the other in any context is semantics-preserving. Compositionality of the semantics is used to prove that the fusion transformations are semantic-preserving. The following theorem states the compositionality for r.

Lemma 33 (Compositionality) *For all r, r' and $\mathbb{R}$, if $[\![\, r \,]\!] = [\![\, r' \,]\!]$ then $[\![\, \mathbb{R}[r] \,]\!] = [\![\, \mathbb{R}[r'] \,]\!]$.*

4.5.2 Fusion

We now present the fusion transformations. Fusion reduces computation time by combining separate reductions into a single reduction. The transformations have three main forms: fusion of nested path-based reductions, fusion of pairs of path-based reductions, and fusion of pairs of vertex-based reductions. The result of fusion is an equivalent specification in the triple-let form with separate terms for path-based reduction, mapping over vertices and vertex-based reduction.

The fusion rules are presented in Fig. 4.11 and Fig. 4.12. The top-level fusion relation $\Rightarrow_r$ is called r-fusion and transforms an r term to another. The other fusion relations $\Rightarrow_m$, $\Rightarrow_M$, and $\Rightarrow_R$ which transform m, M and R terms are called m-fusion, M-fusion and R-fusion. We consider m-fusions first. The rule FMInR states that m-fusions can be applied to m terms that appear in the context of r terms. (Both $\mathbb{M}[m_1]$ and $\mathbb{M}[m_2]$ in this rule are r terms.) The rule FMInM states that m-fusions can be applied to m terms in the context of other m terms.

Fusing nested path-based reductions. The rule FPNest m-fuses nested path-based reductions to flat reductions. Consider the nested path-based reduction $\underset{p' \in P'}{\mathsf{R}} \mathcal{F}(p)$ where the set of paths P' is another path-based reduction $\underset{p \in P}{\mathsf{args}\,\mathsf{R}'}\ \mathcal{F}'(p)$ where R' is min or max. Let us assume that R' is min. A straightforward calculation computes $\mathcal{F}'$ on the paths P and finds the subset of paths P' with the minimum value, and then computes $\mathcal{F}$ on the paths P' and reduces them by R. An optimized calculation can compute both $\mathcal{F}'$ and $\mathcal{F}$ on the paths P simultaneously and only consider the pairs with the minimum first element to calculate the reduction R over the second elements. To calculate the values of the path functions $\mathcal{F}$ and

$\mathcal{F}'$, this approach enumerates paths only once instead of twice. Therefore, the two reductions can be fused into one reduction as $\mathsf{ilet}\ \langle x, x'\rangle \coloneqq \underset{p' \in P}{\mathsf{R}''}\ \mathcal{F}''(p')\ \mathsf{in}\ x'$. The new path function $\mathcal{F}''$ returns the pair of values $\mathcal{F}'(p')$ and $\mathcal{F}(p')$ for an input path p'. The new reduction function R'' considers the first element of the two input pairs and if the first element of one input is (strictly) smaller than the other, that input is returned. That input takes over because the set of paths for the reduction R are only those with the minimum value for $\mathcal{F}'$. On the other

FMinR

$$\frac{m_1 \quad \Rightarrow_m \quad m_2}{\mathbb{M}[m_1] \quad \Rightarrow_r \quad \mathbb{M}[m_2]}$$

FPNest

$$\underset{p' \in \mathsf{args}\,\underset{p \in P}{\mathsf{R}'}\ \mathcal{F}'(p)}{\mathsf{R}} \mathcal{F}(p') \quad \Rightarrow_m \quad \mathsf{ilet}\ \langle x, x'\rangle \coloneqq \underset{p \in P}{\mathsf{R}''}\ \mathcal{F}''(p)\ \mathsf{in}\ x'$$

where $\mathsf{R}' \in \{\min, \max\}$

$\mathcal{F}'' \coloneqq \lambda p.\ \langle \mathcal{F}'(p), \mathcal{F}(p)\rangle$

$\mathsf{R}''(\langle a, b\rangle, \langle a', b'\rangle) \coloneqq$

$\mathsf{if}\ \ (a = a')\ \mathsf{then}\ \langle a, \mathsf{R}(b, b')\rangle$

$\mathsf{else}\ \ \mathsf{if}\ \ (\mathsf{R}'(a, a') = a)\ \mathsf{then}\ \langle a, b\rangle\ \mathsf{else}\ \ \langle a', b'\rangle$

FMinM

$$\frac{m_1 \quad \Rightarrow_m \quad m_2}{\mathbb{M}[m_1] \quad \Rightarrow_m \quad \mathbb{M}[m_2]}$$

FPRed

$$\underset{p \in \mathsf{Paths}}{\mathsf{R}} \mathcal{F}(p) \quad \Rightarrow_m \quad \mathsf{ilet}\ x \coloneqq \mathsf{R}\ \mathcal{F}\ \mathsf{in}\ x$$

FILetBin

$$(\mathsf{ilet}\ X_1 \coloneqq M_1\ \mathsf{in}\ e_1)\ \oplus\ (\mathsf{ilet}\ X_2 \coloneqq M_2\ \mathsf{in}\ e_2) \quad \Rightarrow_m \quad \mathsf{ilet}\ \langle X_1, X_2\rangle \coloneqq \langle M_1, M_2\rangle\ \mathsf{in}\ e_1 \oplus e_2$$

if $\mathsf{free}(e_1)\ \cap\ X_2 = \emptyset$, $\mathsf{free}(e_2)\ \cap\ X_1 = \emptyset$

FVRed

$$\underset{\mathsf{V}}{\mathsf{R}}\ (\mathsf{ilet}\ X \coloneqq \mathsf{R}'\ \mathcal{F}\ \mathsf{in}\ e) \quad \Rightarrow_r \quad \mathsf{ilet}\ X \coloneqq \mathsf{R}'\ \mathcal{F}\ \mathsf{in}\ \ \mathsf{mlet}\ x \coloneqq e\ \mathsf{in}\ \ \mathsf{rlet}\ x' \coloneqq \mathsf{R}\ x\ \mathsf{in}\ x'$$

FMinILet

$$\frac{M_1 \quad \Rightarrow_M \quad M_2}{\mathsf{ilet}\ X \coloneqq \mathbb{M}\mathsf{s}[M_1]\ \mathsf{in}\ e \quad \Rightarrow_m \quad \mathsf{ilet}\ X \coloneqq \mathbb{M}\mathsf{s}[M_2]\ \mathsf{in}\ e}$$

FMPair

$$\big\langle \mathsf{R}\ \mathcal{F}, \mathsf{R}'\ \mathcal{F}' \big\rangle \quad \Rightarrow_M \quad \mathsf{R}''\ \mathcal{F}''$$

where $\mathcal{F}'' \coloneqq \lambda p.\ \langle \mathcal{F}(p), \mathcal{F}'(p)\rangle$

$\mathsf{R}''(\langle a, b\rangle, \langle a', b'\rangle) \coloneqq \big\langle \mathsf{R}(a, a'), \mathsf{R}'(b, b')\big\rangle$

Figure 4.11: Fusion Rules

FLetsBin

$$\begin{pmatrix} \mathsf{ilet}\ X_1 \coloneqq M_1\ \mathsf{in} \\ \mathsf{mlet}\ X_1' \coloneqq E_1\ \mathsf{in} \\ \mathsf{rlet}\ X_1'' \coloneqq R_1\ \mathsf{in} \\ e_1 \end{pmatrix} \oplus \begin{pmatrix} \mathsf{ilet}\ X_2 \coloneqq M_2\ \mathsf{in} \\ \mathsf{mlet}\ X_2' \coloneqq E_2\ \mathsf{in} \\ \mathsf{rlet}\ X_2'' \coloneqq R_2\ \mathsf{in} \\ e_2 \end{pmatrix} \Rightarrow_r \begin{pmatrix} \mathsf{ilet}\ \langle X_1, X_2\rangle \coloneqq \langle M_1, M_2\rangle\ \mathsf{in} \\ \mathsf{mlet}\ \langle X_1', X_2'\rangle \coloneqq \langle E_1, E_2\rangle\ \mathsf{in} \\ \mathsf{rlet}\ \langle X_1'', X_2''\rangle \coloneqq \langle R_1, R_2\rangle\ \mathsf{in} \\ e_1 \oplus e_2 \end{pmatrix} \text{ if}$$

$$\mathsf{free}(E_1) \cap X_2 = \mathsf{free}(E_2) \cap X_1 = \mathsf{free}(R_1) \cap X_2' = \mathsf{free}(R_2) \cap X_1' = \mathsf{free}(e_1) \cap X_2'' = \mathsf{free}(e_2) \cap X_1'' = \emptyset$$

FMInLets

$$\frac{M_1 \Rightarrow_M M_2}{\begin{pmatrix} \mathsf{ilet}\ X \coloneqq \mathbb{M}\mathsf{s}[M_1]\ \mathsf{in} \\ \mathsf{mlet}\ X' \coloneqq E\ \mathsf{in} \\ \mathsf{rlet}\ X'' \coloneqq R\ \mathsf{in}\ e \end{pmatrix} \Rightarrow_r \begin{pmatrix} \mathsf{ilet}\ X \coloneqq \mathbb{M}\mathsf{s}[M_2]\ \mathsf{in} \\ \mathsf{mlet}\ X' \coloneqq E\ \mathsf{in} \\ \mathsf{rlet}\ X'' \coloneqq R\ \mathsf{in}\ e \end{pmatrix}}$$

FRPair

$$\langle \mathsf{R}_1\ x_1, \mathsf{R}_2\ x_2\rangle \Rightarrow_R \mathsf{R}_3\ \langle x_1, x_2\rangle$$

where $\mathsf{R}_3(\langle a, b\rangle, \langle a', b'\rangle) \coloneqq \langle \mathsf{R}_1(a, a'), \mathsf{R}_2(b, b')\rangle$

FRInLets

$$\frac{R_1 \Rightarrow_R R_2}{\begin{pmatrix} \mathsf{ilet}\ X \coloneqq M\ \mathsf{in} \\ \mathsf{mlet}\ X' \coloneqq E\ \mathsf{in} \\ \mathsf{rlet}\ X'' \coloneqq \mathbb{R}\mathsf{s}[R_1]\ \mathsf{in}\ e \end{pmatrix} \Rightarrow_r \begin{pmatrix} \mathsf{ilet}\ X \coloneqq M\ \mathsf{in} \\ \mathsf{mlet}\ X' \coloneqq E\ \mathsf{in} \\ \mathsf{rlet}\ X'' \coloneqq \mathbb{R}\mathsf{s}[R_2]\ \mathsf{in}\ e \end{pmatrix}}$$

Figure 4.12: More Fusion Rules

hand, if the first elements of the inputs are equal, their second elements are reduced by R to make the second element of the output pair. The rule FPNest can be repeatedly applied to a deeply nested path-based reduction to flatten it to a reduction over the basic paths term Paths.

Factoring, pairing and fusing path-based reductions. The rule FPRed factors out a flat reduction to an equivalent let form. The rule FILetBin fuses an operation between two let terms to a single let term. It pairs the factored reductions M_1 and M_2 of the two let terms. The condition of the rule prevents the free variables of the expression of one term from clashing with the bound variables of another. The rule FMInILet allows the factored reductions M in the context of a let term to be fused. The rule FMPair M-fuses a pair of factored reductions $\left\langle \mathsf{R}\ \mathcal{F}, \mathsf{R}'\ \mathcal{F}' \right\rangle$ to a single reduction $\mathsf{R}''\ \mathcal{F}''$ that calculates the two reductions simultaneously. The path function $\mathcal{F}''$ returns the pair of the results of $\mathcal{F}$ and $\mathcal{F}'$. Similarly, the reduction function R'' returns a pair: the reduction of the first elements by R and the second elements by R'.

Factoring into, pairing and fusing triple-let terms. The rules above can factor all path-based reductions m to the let form, and fuse factored reductions to a single one. The next rule FVRed transforms vertex-based reductions that are applied to these path-based reductions to an equivalent triple-let form. The triple-let form factors path-based and vertex-based reductions in separate let parts. The rule FLetsBin fuses an operation between two triple-let terms to a single triple-let term. It pairs the factored path-based reductions M, expression E, and vertex-based reduction R of the two terms. The rules FMInLets and FRinLets allow the factored reductions M and R in the context of a triple-let term to be fused.

Fusing vertex-based reductions. The rule FRPair presents R-fusions. It fuses a pair of factored vertex-based reductions $\langle \mathsf{R}_1\ x_1, \mathsf{R}_2\ x_2 \rangle$ to a single reduction $\mathsf{R}_3\ \langle x_1, x_2 \rangle$.

Given two pairs, R_3 returns a pair: the reduction of the first elements by R_1 and the second elements by R_2.

We saw an example fusion in Fig. 4.2. The fusion transformation presented above is semantic-preserving: terms are only fused into other terms with the same semantics. The following theorem states the semantics-preservation property of fusion.

Theorem 34 (Semantics-preserving Fusion) *For all* r_1 *and* r_2*, if* $r_1 \Rightarrow_r r_2$ *then* $[\![r_1]\!] = [\![r_2]\!]$.

4.5.3 Extensions

We now consider extensions to the core syntax and the fusion rules.

Common Operation Elimination. Fusion factors path-based reductions, vertex-based map operations and vertex-based reductions into the triple-let form. This form facilitates common operation elimination. For example, if a path-based reduction appears twice in the first let and assigned to two sets of variables, one can be eliminated and the result of the other can be assigned to both sets of variables.

Domain. The scalar semantic domain of the core language was confined to the natural numbers. The domain are simply extended to booleans, vertex identifiers and also sets of values. Thus, the reduction operations are extended with union $\cup$ and intersection $\cap$, and the path functions are extended with head and penultimate. The function head returns the identifier of the head vertex of the path, and the function penultimate returns the identifier of the penultimate (that is the vertex before the last) of the path.

Unary operations and Literals. The path-based reductions m and vertex-based reductions r and their fusion rules can be simply extended with unary operations and literals.

Vertex Variables. We extend the core syntax with path terms $\mathsf{Paths}(v, v')$ and $\mathsf{Paths}(v)$ that can specify vertex variables as source and destination. The term $\mathsf{Paths}(v, v')$ specifies the set of paths from the source v to the destination v', and the term $\mathsf{Paths}(v)$ specifies the set of paths from any source to the destination v. Thus, the source s of a path-based reduction can be either a vertex v or none $\perp$. A factored path-based reduction $\underset{c}{\mathsf{R}}\ \mathcal{F}$ carries its configuration c, that is either a source, or a pair of other configurations. We also extend the syntax with vertex-based reductions $\underset{v \in \mathsf{V}}{\mathsf{R}}\ m$ that can bind the vertex variable v.

Syntactic Sugar. Syntactic sugar enables concise specifications. For example, the term $\mathcal{F}(\underset{p \in P}{\mathsf{arg\,R}}\ \mathcal{F}'(p))$ where R is either min or max first finds a path p in P with the minimum or maximum value for the function $\mathcal{F}'$, and then returns the result of applying $\mathcal{F}$ to p. It is used to specify the BFS use-case. The following rule expands this term to a path-based reduction in the let form. The path function $\mathcal{F}''$ returns the pair of the results of $\mathcal{F}'$ and $\mathcal{F}$. The reduction function R' returns the input pair with the minimum or maximum first element.

FMRed

$$\mathcal{F}\,(\underset{p \in P}{\mathsf{arg\,R}}\ \mathcal{F}'(p)) \quad := \quad \mathsf{ilet}\ \langle x, x' \rangle := \underset{p \in P}{\mathsf{R}'}\ \mathcal{F}''(p)\ \mathsf{in}\ x' \quad \text{where} \quad \mathsf{R} \in \{\min, \max\}$$
$$\mathcal{F}'' \ := \ \lambda p.\ \langle \mathcal{F}'(p), \mathcal{F}(p) \rangle \qquad \mathsf{R}'(\langle a, b \rangle, \langle a', b' \rangle) \ := \ \mathsf{if}\ \ (\mathsf{R}(a, a') = a)\ \mathsf{then}\ \langle a, b \rangle\ \mathsf{else}\ \ \langle a', b' \rangle \tag{4.7}$$

As an example DS is fused as follows:

$$
\begin{aligned}
\mathrm{DS}(s) &= \bigcup_{v \in \mathsf{V} \,\wedge\, \left(\min\limits_{p \in \mathsf{Paths}(s,v)} \mathsf{weight}(p)\right) > 7} \{v\} \\
&= \mathop{\mathsf{R}}_{v \in \mathsf{V}} \left\langle \left(\min_{p \in \mathsf{Paths}(s,v)} \mathsf{weight}(p)\right) > 7, \{v\} \right\rangle \quad \text{where} \quad
\begin{array}{l}
\mathsf{R}(\langle a, b\rangle, \langle a', b'\rangle) \coloneqq \\
\quad \mathsf{if}\ \ (a \wedge a')\ \mathsf{then}\ \langle a, b\ \cup\ b'\rangle \\
\quad \mathsf{else}\ (a)\ \mathsf{then}\ \langle a, b\rangle \\
\quad \mathsf{else}\ \ \langle a', b'\rangle
\end{array} \\
&= \mathop{\mathsf{R}}_{v \in \mathsf{V}} \langle \left(\mathsf{ilet}\ x \coloneqq \min_{s}\ \mathsf{weight}\ \mathsf{in}\ x\right) > \mathsf{ilet}\ x' \coloneqq \bot\ \mathsf{in}\ 7,\ \ \mathsf{ilet}\ x'' \coloneqq \bot\ \mathsf{in}\ \{v\}\rangle \\
&= \mathop{\mathsf{R}}_{v \in \mathsf{V}} \left(\mathsf{ilet}\ \langle\langle x, x'\rangle, x''\rangle \coloneqq \langle\langle \min_{s}\ \mathsf{weight}, \bot\rangle, \bot\rangle\ \mathsf{in}\ \ \langle x > 7, \{v\}\rangle\right) \\
&= \mathop{\mathsf{R}}_{v \in \mathsf{V}} \left(\mathsf{ilet}\ x \coloneqq \min_{s}\ \mathsf{weight}\ \mathsf{in}\ \ \langle x > 7, \{v\}\rangle\right) \\
&= \left(
\begin{array}{l}
\mathsf{ilet}\ x \coloneqq \min\limits_{s}\ \mathsf{weight}\ \mathsf{in} \\
\mathsf{mlet}\ x' \coloneqq \langle x > 7, \{v\}\rangle\ \ \mathsf{in} \\
\mathsf{rlet}\ x'' \coloneqq \mathsf{R}\ x'\ \mathsf{in} \\
x''
\end{array}
\right)
\end{aligned}
$$

Nested Triple-lets. The core syntax supports expressions that can be fused to a single iteration-map-reduce triple-let term. We extend the core syntax to support nested vertex-based reductions, and extend the fusion rules to fuse them. For example, the use-case LTrust that we saw in Fig. 4.6 uses the vertex-based reduction Radius as a nested term. Nested triple-let terms can be translated to a sequence of iteration-map-reduce rounds on the graph.

4.6 Mapping Specification to Iteration-Map-Reduce

As we saw in the final term of Fig. 4.2, fusion results in the triple-let form shown in Fig. 4.13. The three let parts can be directly mapped to three computation primitives: iteration, map and reduce. Each vertex stores the variables X and X'. The first let is mapped to an iterative calculation for the path-based reduction $\underset{c}{\mathsf{R}}\ \mathcal{F}$ that results in values for the variables X in each vertex. The second let is mapped to a map operation over vertices: given the values of the variables X in each vertex, the map operation calculates the values of the expressions E, and stores the results in the variables X' for the vertex. The third let is mapped to a reduction operation over vertices: given the values of the variables $\overline{x'}$ in X' in each vertex, the reduction operation $\mathsf{R}'\ \langle\overline{x'}\rangle$ reduces the values of $\langle\overline{x'}\rangle$ for all vertices, and stores the results in the global variables X''. Finally, the single expression e is calculated based on the values of X''.

$$\begin{array}{l}\mathsf{ilet}\ X := \underset{c}{\mathsf{R}}\ \mathcal{F}\ \mathsf{in}\\ \mathsf{mlet}\ X' := E\ \mathsf{in}\\ \mathsf{rlet}\ X'' := \mathsf{R}'\ \langle\overline{x'}\rangle\ \mathsf{in}\ \ e\end{array}$$

Figure 4.13: Triple-let Form

The two latter primitives, vertex-based mapping and reduction, can be implemented by a traversal over vertices. Since the mapping and the reduction both traverse the vertices, a simple optimization is to perform them in the same pass. Now, we consider how path-based reductions can be implemented. We saw the iterative computation models in § 4.4. In the next subsections, we present how they can be instantiated to implement path-based reductions. We first present the correctness conditions of the iterative models to calculate path-based reductions (§ 4.6.1), and then present the synthesis of iteration kernel functions based on the correctness conditions (§ 4.6.2).

4.6.1 The Correctness of Iterative Path-Based Reduction

This subsection presents the iterative calculation of path-based reductions. We consider both the pull and push models with both idempotent and non-idempotent reduction. For each model, we present correctness and termination conditions.

Specification. Factored path-based reductions in the triple-let specifications have the form $\underset{c}{\mathsf{R}}\, \mathcal{F}$. Considering a general single reduction, c is either none $\perp$ or a source vertex s. The factored reduction for the former (with no source) is simply unrolled to $\mathsf{R}_{p \in \mathsf{Paths}(v)}\ \mathcal{F}(p)$ and the latter (with the source s) is unrolled to $\mathsf{R}_{p \in \{p \,|\, p \in \mathsf{Paths}(v) \,\wedge\, \mathsf{head}(p) = s\}}\ \mathcal{F}(p)$. Both of these reductions can be captured as the following general specification where the condition $C(p)$ is $\mathbb{T}$ for the former and is $\mathsf{head}(p) = s$ for the latter.

Definition 35 (Specification) $\mathcal{S}pec(v) = R_{p \in \{p \,|\, p \in \mathsf{Paths}(v) \,\wedge\, C(p)\}}\ \mathcal{F}(p)$

The reduction function R is associative and commutative. It returns $\perp$ on an empty set, and returns the single element on a singleton set.

Model Instantiation. The iterative models (we saw in § 4.4) are parametric in terms of the kernel functions $\mathcal{I}$, $\mathcal{P}$, R, and $\mathcal{E}$. We consider the correctness conditions on the kernel functions such that the iterative models calculate the specified path-based reduction. We will see in § 4.6.2 that these conditions will guide automatic synthesis of the kernel functions $\mathcal{I}$, $\mathcal{P}$, and R for a given path-based reduction.

Correctness. The iterative models calculate the value $\mathcal{S}^k(v)$ of each vertex v in iterations k by propagating the values of its neighbor vertices. The iteration stops when the value of no vertex changes. The values of the vertices $\mathcal{S}^k(v)$ are expected to converge to the specification $\mathcal{S}pec(v)$. We show the correctness in two steps. (1) First, we show that under

certain conditions, at the end of each iteration k, the value $\mathcal{S}^k(v)$ of each vertex v is equal to the iteration specification $\mathcal{S}pec^k(v)$ for the iteration k. The specification $\mathcal{S}pec^k(v)$ is defined as the result of reduction over paths of length less than k.

Definition 36 $\quad \mathcal{S}pec^k(v) = \mathsf{R}_{p \in \{p \mid p \in \mathsf{Paths}(v) \wedge \mathsf{C}(p) \wedge \mathsf{length}(p) < k\}} \mathcal{F}(p)$

(2) Second, we show that under certain conditions, there is an index k where $\mathcal{S}pec^k(v)$ and $\mathcal{S}pec^{k+1}(v)$ are equal with each other and $\mathcal{S}pec(v)$ as well. These two steps together show that the values of vertices $\mathcal{S}^k(v)$ eventually converge to $\mathcal{S}pec(v)$. We now consider the four variants of the iterative models.

Pull Model. We consider the correctness of the pull model to calculate path-based reductions. We look at idempotent and non-idempotent reduction functions in turn.

The correctness of the pull models is dependent on the conditions $\mathbb{C}_1$ - $\mathbb{C}_9$ presented in Fig. 4.14. (A) The conditions $\mathbb{C}_1$ and $\mathbb{C}_2$ require the correctness of initialization function $\mathcal{I}$. If the path condition C holds on the simple initial path $\langle v, v \rangle$, the value of the initialization function $\mathcal{I}$ should be $\mathcal{F}(\langle v, v \rangle)$; otherwise, it should be none $\perp$. (B) The conditions $\mathbb{C}_3$ - $\mathbb{C}_5$ state the requirements for the propagation function $\mathcal{P}$. The condition $\mathbb{C}_3$: It simply states that if the value of the vertex is none $\perp$, its propagated value should be none $\perp$ as well. The condition $\mathbb{C}_4$: We saw an illustration for $\mathbb{C}_4$ in Fig. 4.5a and Fig. 4.5b. For a path p, we call the value of $\mathcal{F}$ on p, the path value of p. The path $p \cdot e$ denotes the extension of the path p at the end with the edge e. Consider two paths p_1 and p_2 that end in a vertex u and there is an edge $\langle u, v \rangle$ from u to another vertex v. Reducing the path values of the two extended paths $p_1 \cdot \langle u, v \rangle$ and $p_2 \cdot \langle u, v \rangle$. should be the same as reducing the path values of p_1 and

p_2, and then propagating the result with $\mathcal{P}$ through $\langle u, v \rangle$. Intuitively, this condition states that the local reduction and propagation effectively calculate reduction over paths. The condition $\mathbb{C}_5$: Vertices that have only a single incoming path do not receive multiple values to be reduced. For such vertices, $\mathbb{C}_5$ states that the propagation of the path value of p over an outgoing edge e is equal to the path value of the extended path $p \cdot e$. (C) The conditions $\mathbb{C}_6$ - $\mathbb{C}_9$ state the required properties of the reduction function R. The none value $\perp$ should be the identity element, and R should be commutative, associative and idempotent. (The epilogue function $\mathcal{E}$ is instantiated to identity.) For example, given the factored path-based reduction $\min\limits_{s} \mathsf{length}$ for the shortest path use-case SSSP, the kernel functions that we saw in Fig. 4.7 satisfy the conditions above.

Pull model with idempotent reduction. The following theorem states that if the conditions above hold, then the value $\mathcal{S}^k_{\mathsf{pull+}}(v)$ that the pull model with idempotent reduction (Theorem 29) calculates complies with the specification $\mathcal{S}pec^k(v)$.

Theorem 37 (Correctness of Pull (idempotent reduction)) *For all R, $\mathcal{F}$, C, $\mathcal{I}$, $\mathcal{P}$, and $k \geq 1$, if the conditions $\mathbb{C}_1$ - $\mathbb{C}_9$ hold, then $\mathcal{S}^k_{\mathsf{pull+}}(v) = \mathcal{S}pec^k(v)$.*

The proof is by induction on the iteration k. At the iteration $k = 1$, the specification $\mathcal{S}pec^1(v)$ requires reduction on only the paths of length zero to each vertex. Therefore, by the conditions $\mathbb{C}_1$ - $\mathbb{C}_2$, the initialization function $\mathcal{I}$ properly initializes each vertex v to $\mathcal{S}pec^1(v)$. In each iteration $k + 1$, if there is any predecessor of the vertex v whose value is changed in the previous iteration k, then their new values are propagated by $\mathcal{P}$ and reduced together by R, and then reduced with the current value of v. By the conditions $\mathbb{C}_7$ and $\mathbb{C}_8$, the reduction function R is commutative and associative, and can be applied to the

propagated values in any order. By the induction hypothesis, the value of each predecessor u is the reduction of the paths to u of length l, $0 \leq l < k$. The predecessors that have no paths and store $\perp$ are ignored by the conditions $\mathbb{C}_3$ and $\mathbb{C}_6$. By the conditions $\mathbb{C}_4$ and $\mathbb{C}_5$, the propagation of the value of a predecessor u of the vertex v is equal to the reduction over the paths to v that pass through u. Since these paths include the edge (from u to v), their length l is $0 < l < k + 1$. The previous value of v itself is the reduction over paths to v of length l, $0 \leq l < k$. Since, the reduction function R is idempotent, reducing these two values absorbs the values of the repeated paths, and results in the reduction over all paths of length l, $0 \leq l < k + 1$ that the specification requires.

Pull model with non-idempotent reduction. The pull model with non-idempotent reduction $\mathcal{S}^k_{\mathsf{pull-}}(v)$ is defined in Theorem 30. We show that it can correctly calculate path-based reductions with non-idempotent (in addition to idempotent) reduction functions. For instance, consider the factored path-based reduction $\sum_s 1$ that counts the number of paths from the source s; the reduction function sum $\sum$ is non-idempotent. The initialization function is instantiated to $\mathcal{I} = \lambda v.\ 1$ and the propagation function is instantiated to $\mathcal{P} = \lambda n, e.\ n$ that simply propagates the value of the predecessor.

The following theorem states that if the conditions above except idempotency hold and the source vertex is not on any cycle then the pull model with non-idempotent reduction $\mathcal{S}^k_{\mathsf{pull-}}(v)$ complies with the specification $\mathcal{S}pec^k(v)$.

Theorem 38 (Correctness of Pull (non-idempotent reduction)) *For all* R, $\mathcal{F}$, $\mathcal{I}$, $\mathcal{P}$, $k \geq 1$, *and* s, *let* $\mathsf{C}(p) = (\mathsf{head}(p) = s)$, *if* $\mathbb{C}_1$ - $\mathbb{C}_8$ *hold, and* s *is not on any cycle,* $\mathcal{S}^k_{\mathsf{pull-}}(v) = \mathcal{S}pec^k(v)$.

A. Initialization:

$\mathbb{C}_1$ (Init1): $\forall v.\ \mathsf{C}(\langle v, v\rangle) \to \mathcal{I}(v) = \mathcal{F}(\langle v, v\rangle)$

$\mathbb{C}_2$ (Init2): $\forall v.\ \neg\mathsf{C}(\langle v, v\rangle) \to \mathcal{I}(v) = \bot$

B. Propagation:

$\mathbb{C}_3$ (None Propagation): $\forall e.\ \mathcal{P}(\bot, e) = \bot$

$\mathbb{C}_4$ (Aggregate Propagation):

$\forall p_1, p_2, v.$

$\mathsf{tail}(p_1) = \mathsf{tail}(p_2) \to$

$\mathsf{let}\ u := \mathsf{tail}(p_1)\ \mathsf{in}$

$\mathcal{P}\,[\mathsf{R}(\mathcal{F}(p_1), \mathcal{F}(p_2)),\ \langle u, v\rangle] = \mathsf{R}\,[\mathcal{F}(p_1 \cdot \langle u, v\rangle),\ \mathcal{F}(p_2 \cdot \langle u, v\rangle)]$

$\mathbb{C}_5$ (Single Path): $\forall p, e.\ \mathcal{P}(\mathcal{F}(p),\ e) = \mathcal{F}(p \cdot e)$

C. Reduction:

$\mathbb{C}_6$ (Identity): $\forall n.\ \mathsf{R}(n, \bot) = n$

$\mathbb{C}_7$ (Commutativity): $\forall n, n'.\ \mathsf{R}(n, n') = \mathsf{R}(n', n)$

$\mathbb{C}_8$ (Associativity): $\forall n, n', n''.\ \mathsf{R}(\mathsf{R}(n, n'), n'') = \mathsf{R}(n, \mathsf{R}(n', n''))$

$\mathbb{C}_9$ (Idempotency): $\forall n.\ \mathsf{R}(n, n) = n$

Termination:

$\mathbb{C}_{10}$ (Simple Path): $\forall p.\ \mathsf{R}(\mathcal{F}(p), \mathcal{F}(\mathsf{simple}(p))) = \mathcal{F}(\mathsf{simple}(p))$

Figure 4.14: Correctness and Termination Conditions

Compared to the previous model, the reduction with the current value is avoided. However, no path is missed. The difference is only the paths of length 0. The vertices other than the source s do not have a path of length 0 from s. The source s itself is also correctly

initialized to the value of $\mathcal{F}$ on the zero-length path $\langle s, s \rangle$, and since s is not on any cycle, its correct value is never overwritten.

Push Model. We now consider the correctness of the push model to calculate path-based reductions.

Push model with idempotent reduction. The following theorem states that if the conditions $\mathbb{C}_1$ - $\mathbb{C}_9$ hold, the value $\mathcal{S}^k_{\mathsf{push}+}(v)$ calculated by the push model with idempotent reduction, (Theorem 31) complies with the specification $\mathcal{S}pec^k(v)$.

Theorem 39 (Correctness of push (idempotent reduction)) *For all R, $\mathcal{F}$, C, $\mathcal{I}$, $\mathcal{P}$, and $k \geq 1$, if the conditions $\mathbb{C}_1$ - $\mathbb{C}_9$ hold, $\mathcal{S}^k_{\mathsf{push}+}(v) = \mathcal{S}pec^k(v)$.*

Push model with non-idempotent reduction. Similarly, the following theorem states the correctness of the push model with non-idempotent reduction (Theorem 32).

Theorem 40 (Correctness of Push (non-idempotent reduction)) *For all R, $\mathcal{F}$, $\mathcal{I}$, $\mathcal{P}$, $k \geq 1$, and s, let $C(p) = (\mathit{head}(p) = s)$, if $\mathbb{C}_1$ - $\mathbb{C}_8$ hold, and s is not on any cycle, $\mathcal{S}^k_{\mathsf{push}-}(v) = \mathcal{S}pec^k(v)$*

Termination. We show that under certain conditions, there exists an iteration k where $\mathcal{S}pec^k(v)$ (Theorem 36) stays unchanged and converges to the original specification $\mathcal{S}pec(v)$ (Theorem 35). Iterations incrementally consider longer paths; however, longer paths do not necessarily yield new information. For example, in the shortest path use-case SSSP, after considering all the simple paths, the longer paths (that are cyclic) cannot lead to shorter paths (in graphs with non-negative edges). Given a path p, we call the path that results from removing its cycles the simplification $\mathsf{simple}(p)$ of p. In the shortest path use-case SSSP, the

reduction function R is min and the path function $\mathcal{F}$ is weight. Reducing the $\mathcal{F}$ value of $\mathsf{simple}(p)$ with the $\mathcal{F}$ value of p results in the former. Therefore, simplified paths are enough to arrive at the same result for the reduction, and longer paths do not change the result. We capture this property as the condition $\mathbb{C}_{10}$ in Fig. 4.14 and prove convergence.

Theorem 41 (Termination) *For all R, $\mathcal{F}$, and C, if the graph is acyclic or the condition $\mathbb{C}_{10}$ holds, then there exists k such that for every $k' \geq k$, $\mathcal{S}pec^{k'}(v) = \mathcal{S}pec(v)$.*

Let l be the length of the longest simple path to the vertex v. After the iteration $k = l + 1$, the value of $\mathcal{S}pec^{k}(v)$ stays unchanged. This is because the reduction with the paths of length greater than l does not change the value of $\mathcal{S}pec^{k}(v)$. If a path p that is longer than l exists, then p is not simple, i.e., it includes a cycle. This is refuted if the graph is acyclic. Otherwise, the simplification of p, $\mathsf{simple}(p)$, is already in the set of paths of length less than $l + 1$ and by the condition $\mathbb{C}_{10}$, reducing the path value of p with the path value of $\mathsf{simple}(p)$ results in the path value of $\mathsf{simple}(p)$.

An immediate corollary of the above two theorems is that iteration eventually terminates and converges to the specification $\mathcal{S}pec(v)$ (if the corresponding conditions in Theorem 37 to Theorem 40 hold). The final iteration is simply the maximum value of k from Theorem 41 for all vertices. For example, the corollary for the pull model for idempotent reduction functions is the following.

Corollary 42 (Termination for pull model with idempotent reduction) *For all R, $\mathcal{F}$, C, $\mathcal{I}$, and $\mathcal{P}$, if the conditions $\mathbb{C}_1$ - $\mathbb{C}_9$ hold, and the graph is acyclic or the condition $\mathbb{C}_{10}$ holds, then there exists an iteration k such that $\mathcal{S}^{k}_{\mathsf{pull+}}(v) = \mathcal{S}pec(v)$.*

We state the correctness conditions and prove similar theorems for all the models.

```
def SynthP (F, R)
  Let T be the return type of F.
  memoize variable v for type T and size 1
  memoize variable l for type Edge and size 1
  foreach (literal l_i with type T_i)
    memoize l_i for T_i and size 1
  size ← 1
  while (true)
    E ← Candidates (T, size)
    foreach (e ∈ E)
      if F; R; Γ ⊢ (ℂ4 ∧ ℂ5)[P := (λv, l. e)]
        return (λv, l. e)
    size ← size + 1
```

(a)

```
P := List[V],
eweight: ⟨V, V⟩ → ℕ
weight: P → ℕ

∀p. if (p = ⊥)
      weight(p) = 0
    else
      let v := head(p), p' := tail(p) in
      if (p' = ⊥) weight(p) = 0
      else
        let v' := head(p') in
        weight(p) = weight(p') + eweight(⟨v', v⟩)
```

(b)

Figure 4.15: (a) Synthesis of the Propagation Function $\mathcal{P}$. (b) Context assertions Example.

4.6.2 Synthesis of Iterative Reduction

Given a path-based reduction $\underset{c}{\mathsf{R}}\ \mathcal{F}$, we now use the correctness conditions presented in the previous subsection to automatically synthesize correct-by-construction kernel functions.

For example, consider the push iterative model with idempotent reduction that we saw in Fig. 4.8, Theorem 31. By Theorem 39, we need to find the functions $\mathcal{I}$, $\mathcal{P}'$ and R' such that the conditions $\mathbb{C}_1$ - $\mathbb{C}_{10}$ (presented in Fig. 4.14) hold. We use these conditions to

synthesize the functions $\mathcal{I}$, $\mathcal{P}'$ and R'. In particular, (1) we use the initialization conditions $\mathbb{C}_1$ - $\mathbb{C}_2$ to synthesize $\mathcal{I}$, (2) we use the propagation conditions $\mathbb{C}_4$ and $\mathbb{C}_5$ to synthesize $\mathcal{P}$ and then wrap it in the following function $\mathcal{P}'$ to handle none $\perp$ values and satisfy the condition $\mathbb{C}_3$. The some value of v is denoted as $[v]$.

$$\mathcal{P}' \coloneqq \lambda n, e.\ \mathsf{if}\ \ (n = \perp)\ \mathsf{return}\ \ \perp\ \mathsf{else}\ \ \mathsf{return}\ \ [\mathcal{P}(n, e)]$$

and (3) we check the conditions $\mathbb{C}_7$ - $\mathbb{C}_9$ for the reduction function R, and then wrap R in the following reduction function R' to handle none $\perp$ values so that the condition $\mathbb{C}_6$ is satisfied. If the conditions $\mathbb{C}_7$ - $\mathbb{C}_9$ hold for R, they hold for R' as well.

$$\begin{aligned}\mathsf{R}' \coloneqq \lambda a, b.\ \mathsf{if}\ \ (a = \perp)\ \mathsf{return}\ \ b\ \mathsf{else}\ \ \mathsf{if}\ \ (b = \perp)\ \mathsf{return}\ \ a \\ \mathsf{else}\ \ \mathsf{return}\ \ [\mathsf{R}(a, b)]\end{aligned}$$

To find candidate expressions for the body of $\mathcal{I}$ and $\mathcal{P}$, we apply a type-guided enumerative search. It enumerates expressions from the grammar that we saw in Fig. 4.7a in the order of increasing size. To support overloaded operators, the expression constructors have union types. To synthesize an expression of the given type, the search only considers expression constructors that return that type. Given the parameter types of the constructor, it then recursively searches for the arguments, and uses memoization to avoid redundant enumeration.

The procedure $\mathsf{Synth}\mathcal{P}$ that synthesizes $\mathcal{P}$ is shown in Fig. 4.15a. (The synthesis of the other kernel functions is similar.) It starts by memoizing expressions of size one, literals and variables, to make them available for the synthesis of the body of $\mathcal{P}$. Let T be the return type of $\mathcal{F}$; thus, vertices store values of type T. The propagation function $\mathcal{P}$ takes a value stored at a vertex (of type of T) and an edge (of type Edge) and returns a vertex value (of

type T). Thus, the two input parameters of type T and Edge are memoized as available expressions. Then, candidate bodies for $\mathcal{P}$ (of type T) of increasing sizes are obtained.

A candidate is correct if the condition $\mathbb{C}_4$ and $\mathbb{C}_5$ are valid when $\mathcal{P}$ is replaced with the candidate. To check the validity of an assertion, we use off-the-shelf SMT solvers to check the satisfiability of its negation. The context of the validity check $\mathcal{F}; \mathsf{R}; \Gamma$ is the definition of the functions $\mathcal{F}$ and R from the given path-based reduction, and a set of assertions Γ that define basic graph functions and relations. We model paths P as lists of vertices V, and define graph functions and relations including the path functions length, weight, $\mathsf{punultimate}$ and $\mathsf{capacity}$ in the combination of the quantified uninterpreted functions and list theories. Fig. 4.15b showcases the axiomatization of the weight function in Γ. The edge-weight $\mathsf{eweight}$ is a function on pairs of vertices $\langle \mathsf{V}, \mathsf{V} \rangle$, and the path weight weight is a function on paths P to natural numbers $\mathbb{N}$. If the list for the path is empty or has a single vertex, the weight of the path is trivially zero; otherwise, the weight of the path is recursively the sum of the weight of the path without the last edge, and the edge-weight of the last edge.

For termination, we check a stronger condition than $\mathbb{C}_{10}$. We remove an edge instead of a cycle: for every path p and edge e, if reducing the $\mathcal{F}$ value of p with the $\mathcal{F}$ value of $p \cdot e$ results in the former, then the reduction is terminating.

4.7 Experimental Results

Implementation. We implemented the GraFS synthesis tool in three parts: fusion, synthesis and backends. The fusion phase closely follows the fusion rules (of § 4.5.2) using the visitor pattern. The synthesis phase uses the Z3 SMT solver to check the validity

of the correctness conditions. GRAFS incorporates a dedicated backend for each framework. Each backend generates a framework-specific C++ file containing the initialization $\mathcal{I}$, propagation $\mathcal{P}$, (if needed rollback $\mathbb{B}$) and reduction function R.

Platform and Benchmarks. We performed the experiments on a 4-node cluster, each with 32 cores and 64GB memory. The experiments for frameworks that are exclusively for shared memory are performed on one of these nodes. The nodes are connected via 40Gbps InfiniBand network, and they run CentOS 7.4 Linux x86_64 kernel version 3.10. All programs are compiled with gcc-5.1.0 (for Ligra, GridGraph and GraphIt) and mpich-3.2.1 and openmpi-3.0 (for PowerGraph and Gemini, respectively). We used social network graphs of various sizes: LiveJournal (LJ, 1.1GB), Twitter (TW, 23GB), TwitterMPI (TM, 28GB), and Friendster (FR, 31GB). We report the average of 5 executions.

Results Summary. To evaluate the GRAFS synthesis tool, we study the effect of fusion on performance. Then, we study the effect of different fusion types separately. The experiments show that fusion can lead up to 4× and in average 2.4× faster execution time compared to the unfused codes. We then report the number of lines of code for the specifications and their synthesis time. Compared to existing frameworks, GRAFS allows significantly more concise specifications, and can efficiently generate code in less than two minutes. Finally, we compare the synthesized code with handwritten versions for use-cases available in the frameworks. Experimental results show that the synthesized code either matches or outperforms handwritten code. We then show that synthesized programs scale similar to handwritten programs.

Table 4.1: Execution times (in seconds). H: Handwritten, S: Synthesized, R: the ratio H/S.

Prog.	Input	Ligra			GridGraph			Gemini			PowerGraph (Push)			PowerGraph (Pull)			GraphIt (Push)		
		H	S	**R**	H	S	**R**	H	S	**R**	H	S	**R**	H	S	**R**	H	S	**R**
DRR	LJ	1.03	0.33	**3.1**	15.3	3.8	**4**	1.2	0.4	**3**	20.4	6.4	**3.2**	36	10	**3.6**	0.63	0.21	**3**
	TW	-	-	-	82	23	**3.6**	7.2	3.7	**2**	120	48	**2.5**	292	81	**3.6**	16.3	5.4	**3**
	TM	-	-	-	141	44	**3.3**	8.9	3.8	**2.3**	166	50	**3.3**	462	86	**2.9**	19	6	**3.1**
	FR	-	-	-	265	73	**3.6**	13	4.9	**2.6**	247	86	**2.9**	522	154	**3.4**	30	9.3	**3.2**
Trust	LJ	1	0.36	**2.7**	14.1	4.4	**3.2**	1.18	0.62	**1.9**	20.5	7.5	**2.7**	37	10	**3.7**	0.61	0.29	**2.1**
	TW	-	-	-	85	28	**3.1**	6.2	5	**1.2**	122	50	**2.4**	293	99	**2.9**	16.1	8.3	**1.9**
	TM	-	-	-	122	40	**3**	7.5	5.6	**1.3**	157	71	**2.2**	455	129	**3.5**	17.2	6.9	**2.5**
	FR	-	-	-	218	117	**2**	10.1	6.7	**1.5**	252	98	**2.6**	526	173	**3**	28	13	**2.1**
LTrust	LJ	1.02	0.63	**1.6**	11.5	6	**1.9**	1.13	0.86	**1.3**	23	15.1	**1.5**	41	28	**1.4**	0.59	0.39	**1.5**
	TW	-	-	-	74	47	**1.6**	6.1	4.9	**1.2**	130	108	**1.2**	-	-	**-**	16	11.4	**1.4**
	TM	-	-	-	142	82	**1.7**	7.3	6	**1.2**	200	134	**1.5**	-	-	**-**	10	20.1	**2**
	FR	-	-	-	210	175	**1.2**	10.2	8.8	**1.1**	286	198	**1.5**	-	-	**-**	34	24.9	**1.3**

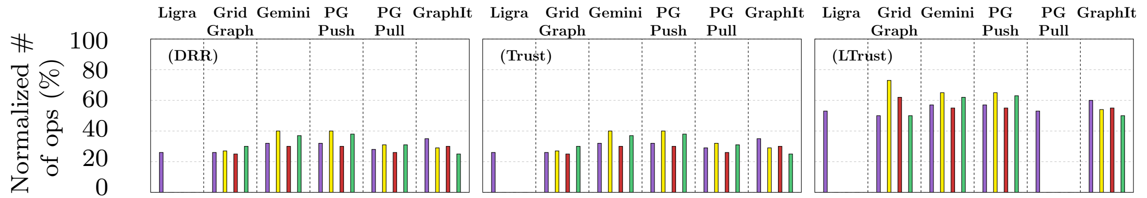


Figure 4.16: Edge-work Ratio: Normalized # of edges processed by the fused over the unfused version. Missing bars correspond to programs not successfully running on input graphs.

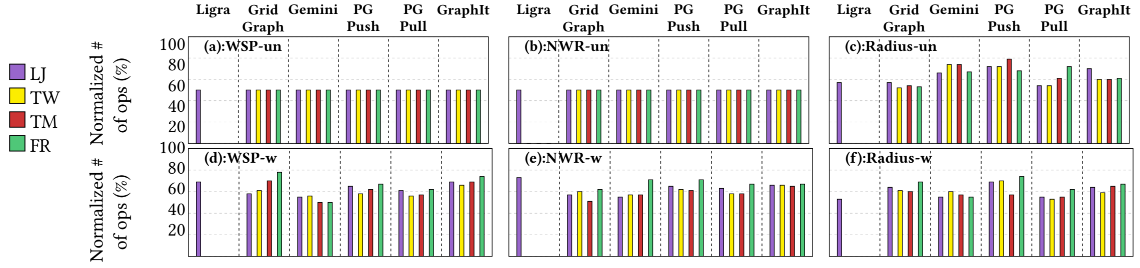


Figure 4.17: Edge-work ratio: # of edges processed by the fused over the unfused version. Missing bars correspond to programs not successfully running on inputs.

The Effect of Fusion. We study the performance benefits of fusion on the more elaborate use-cases: Trust, DRR, and LTrust (presented in Fig. 4.6). The absolute execution times and edge-work ratio are presented in Table 4.1 and Fig. 4.16 respectively. The goal of these measurement is to compare handwritten and synthesized programs. Therefore, for each pair, we fix the framework, its configuration and the iterative model. These measurements are not meant to compare frameworks with each other, as fine-tuning configurations is framework-specific and orthogonal to the goal of these experiments. The number of edges processed by a program indicates the number of times that propagation happens across edges; thus, it represents the amount of computation performed throughout the execution. The edge-work ratio is the number of edges processed by the synthesized (fused) programs normalized w.r.t. that by the unfused versions. The experimental results show that fusion reduces the edge-work ratio up to a quarter and leads to up to 4× speedup. These use-cases benefit from fusion rules for path-based and vertex-based reductions, common operation elimination and factoring of nested vertex-based reductions.

DRR. DRR calculates the ratio of the diameter over radius sampled over two sources. In addition to the rules FMPair, FRPair and FLetsBin which fuse path-based and vertex-based reductions, common operation elimination factors redundant path-based computations in diameter and radius. Therefore, instead of 4 reductions, GraFS fuses and calculates 1 reduction. In Fig. 4.16, we observe that the edge-work ratio is 25-40%. This translates to 2-4× speedup in Table 4.1.

Trust. Trust specifies the trust from a given set of sources to other vertices. It applies division and maximum operator between path-based reductions: the widest and shortest paths. The rules FILetBin and FMPair fuse the 4 path-based reductions to 1. As Fig. 4.16 shows, the edge-work ratio is 25-40%, and as Table 4.1 shows, the speedup is 1.2-3.7×. We note that the theoretical bound on the edge-work ratio for both DRR and Trust is 25%, which happens when the path-based computations for the two sources fully overlap.

LTrust. Given a source s, LTrust calculates the narrowest of the widest paths to vertices within the distance Radius from s. LTrust has a nested reduction for Radius that is factored and then fused. Moreover, the two path-based reductions, the narrowest and shortest paths, are fused by the rules FILetBin and FMPair. This results in a sequence of two iteration-map-reduce rounds. The unfused and fused programs perform four and two sequences of iteration-map-reduce rounds respectively. The theoretical bound for the edge-work ratio is 50% . Fig. 4.16 shows that the edge-work ratio is 57-85% which translates to 1.1-2× speedup in Table 4.1.

Fusion Types. In order to study the performance benefits of different types of fusion rules, we compare the unfused and the fused implementations of three representative use-cases: WSP, NWR and RADIUS (presented in Fig. 4.6). Fig. 4.17 shows the edge-work ratio. We visit the use-cases and the applied fusion rules in turn.

WSP. The unfused program for WSP consists of two computation phases over the edges of the input graph, one after the other. The first calculates the shortest paths from the given source to all the vertices, and the second computes the capacity of the widest path across the shortest paths. WSP is fused by the rule FPNEST that fuses nested path-based reductions. The fused program executes the two computations above in one pass over a pair of values.

Assessment. Fig. 4.17a and Fig. 4.17d show the edge-work ratio of WSP for unweighted and weighted graphs respectively. In unweighted graphs, the fused program processes half the number of edges processed by the unfused program (50% ratio). The ratio is 50-70% for the weighted graphs. When graphs are unweighted, each edge represents a unit cost (either weight or capacity). In each iteration, the set of edges that contribute to the weight and capacity values of a vertex are the same. The fused program exploits this overlap by simultaneously propagating the two values across each edge. However, for weighted graphs, the two values can be propagated to the vertex in different iterations resulting in different shortest and widest paths. Hence, the fused program exploits the partial overlap between the processed edges.

NWR. The unfused version of NWR calculates the narrowest and the widest paths separately. The two are fused by the rule FMPAIR that fuses multiple path-based

reductions into one. It fuses the two reduction functions to one reduction function that operates on pairs. The fused propagation function passes the narrowest and widest values over an edge at the same time.

Assessment. Fig. 4.17b and Fig. 4.17e show the edge-work ratio for NWR for unweighted and weighted graphs respectively. Similar to WSP, the fused program reduces the number of processed edges to 50% for unweighted graphs and to 51-73% for weighted graphs.

RADIUS. RADIUS computes eccentricity (i.e. the maximum shortest distance) by sampling over two sources. The unfused version computes eccentricity separately for each source. However, RADIUS is fused by the rule FMPAIR (that we considered above) and the rule FRPAIR which fuses multiple vertex-based reductions into a single reduction.

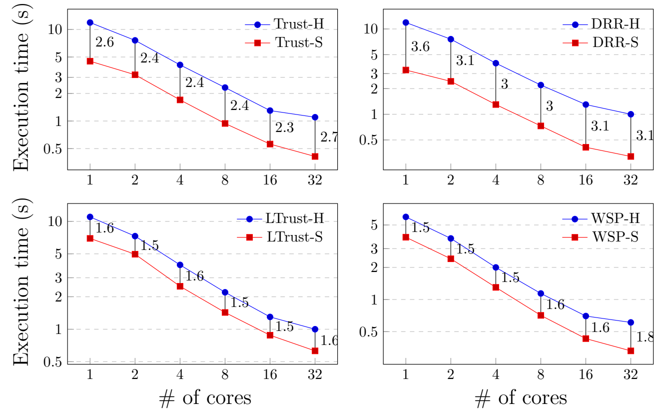
Assessment. Fig. 4.17c and Fig. 4.17f show the edge-work ratio for RADIUS for unweighted and weighted graphs respectively. We observe that on unweighted graphs, the edge-work ratio is 52-78%. This ratio is 53-74% on weighted graphs. Even though fusion enables computation of multiple eccentricity values at the same time, contrary to WSP and NWR, we do not observe the 50% reduction. This is because eccentricity computations across different sources can occur via non-overlaping paths. The fused version exploits the partial overlaps.

We observe that the reduction in edge computations is different across different frameworks as well. For example, the edge-work ratio is 52-68% in GridGraph, whereas 54-78% in PowerGraph. This is because of the difference in the scheduling strategies across these different frameworks, that lead to different overlaps in edge computations.

Usecase	#PBR[1]	T[2]	GF [3]	L[4]	GG[5]	G[6]	PG[7]	GI[8]
BFS	1	25	1	32	100	185	280	130
CC	1	1	1	47	60	117	210	130
SSSP	1	24	1	66	73	117	280	133
WP	1	29	1	66	73	117	280	133
WSP	2	44	2	85	95	197	294	40
NWR	2	58	2	80	95	222	294	40
Radius	2	49	2	38	65	222	294	40
DRR	4	50	3	65	110	213	561	111
Trust	4	105	3	105	145	300	481	96
LTrust	4	102	4	115	148	302	481	107

[1] # of path-based reductions [2] Synthesis time (s) [3] GF: LoC in GraFS [4] L: LoC in Ligra [5] GG: LoC in GridGraph [6] G: LoC in Gemini [7] PG: LoC in PowerGraph [8] GI: LoC in GraphIt

(a)



(b)

Figure 4.18: (a): Synthesis time and the number of lines of code, (b): Scalability on Ligra. X-axis: # of cores. Y-axis: time is logarithmic scale. H: Handwritten. S: Synthesized.

Synthesis Time and LoC. Fig. 4.18a presents the synthesis time for the use-cases. Synthesis is done in less than 2 minutes and often less. It also compares the lines of code (LoC) that user should write in GraFS and the other five frameworks. For each use-case, it reports the number of lines of code of the functions or struct definitions where a change is needed for that use-case. We observe that the GraFS specifications are significantly smaller.

Synthesized Matching Handwritten. We compared the performance of the synthesized programs and their equivalent handwritten programs on five use-cases BFS, CC, SSSP, WP (widest path) and PR. We adopted the handwritten implementations of BFS, CC, SSSP and PR that are available in the frameworks, and developed WP based on SSSP

Table 4.2: Execution Times (in seconds). H: Handwritten, S: Synthesized, R: the ratio H/S, ER: edge-work ratio. Missing cells are due to either missing handwritten use-cases (PR) or not successfully running on an input

Prog.	Input	Ligra				GridGraph				Gemini				PowerGraph (Push)				PowerGraph (Pull)				GraphIt (Push)			
		H	S	**R**	**ER**	H	S	**R**	**ER**	H	S	**R**	**ER**	H	S	**R**	**ER**	H	S	**R**	**ER**	H	S	**R**	**ER**
BFS	LJ	0.38	0.37	**1.02**	**1**	1.56	1.56	**1**	**1**	0.38	0.39	**0.99**	**1**	5.9	5.6	**1.04**	**1**	10.1	9.3	**1.09**	**1**	0.16	0.15	**1.06**	**1**
	TW	8.6	8.7	**0.98**	**1**	210	195	**1.07**	**1**	2.9	2.9	**1**	**1**	33.7	30.8	**1.1**	**1**	69.6	64.2	**1.08**	**1**	4.6	3.8	**1.2**	**1**
	TM	7.1	7	**1.01**	**1**	487	472	**1.03**	**1**	3.1	3.2	**0.96**	**1**	48.8	43.4	**1.12**	**1**	108.9	106.3	**1.02**	**1**	4.1	4.1	**1**	**1**
	FR	-	-	-	-	521	532	**0.97**	**1**	3.7	3.9	**0.96**	**1**	69.9	64.8	**1.07**	**1**	131	118	**1.11**	**1**	7.8	8.2	**0.95**	**1**
CC	LJ	0.36	0.38	**0.94**	**1**	2.21	2.22	**0.99**	**1**	0.77	0.77	**1**	**1**	13	10	**1.3**	**0.45**	19.6	18.9	**1.03**	**1**	0.18	0.19	**0.94**	**1**
	TW	21	20	**1.05**	**1**	230	214	**1.07**	**1**	4.8	4.9	**0.98**	**1**	84.4	69.6	**1.21**	**0.33**	122.6	119.1	**1.02**	**1**	6.3	7.5	**0.84**	**1**
	TM	13	15	**0.86**	**1**	432	423	**1.02**	**1**	7.5	7.6	**0.98**	**1**	160.3	129.7	**1.23**	**0.45**	262.7	241.4	**1.08**	**1**	6.1	6.1	**1**	**1**
	FR	-	-	-	-	606	599	**1.01**	**1**	14	14.3	**0.98**	**1**	259.1	200.7	**1.3**	**0.43**	292.3	293	**0.99**	**1**	12.2	11.7	**1.04**	**1**
SSSP	LJ	0.54	0.57	**0.94**	**1**	2.42	2.1	**1.15**	**1**	0.45	0.49	**0.9**	**1**	7.3	7.3	**1**	**1**	13.3	13.1	**1.01**	**1**	0.2	0.22	**0.9**	**1**
	TW	-	-	-	-	201	205	**0.98**	**1**	2.8	2.8	**1**	**1**	36.1	35	**1.03**	**1**	87.6	84.6	**1.03**	**1**	4.4	4.7	**0.93**	**1**
	TM	-	-	-	-	490	487	**1**	**1**	2.8	3	**0.94**	**1**	47.5	48.8	**0.97**	**1**	137.4	125.3	**1.09**	**1**	5	5.3	**0.94**	**1**
	FR	-	-	-	-	572	570	**1**	**1**	5	5.4	**0.92**	**1**	96.1	90.9	**1.05**	**1**	176.9	182.9	**0.96**	**1**	12.6	14	**0.9**	**1**
WP	LJ	0.61	0.64	**1.04**	**1**	3.46	3.2	**1.08**	**1**	0.45	0.47	**0.97**	**1**	7.8	7.9	**0.98**	**1**	15.1	14.7	**1.02**	**1**	0.25	0.2	**1.25**	**1**
	TW	-	-	-	-	245	242	**1.01**	**1**	3	3	**1**	**1**	36.4	36.4	**1**	**1**	93.2	91.68	**1.01**	**1**	5.5	5	**1.1**	**1**
	TM	-	-	-	-	479	498	**0.96**	**1**	3.2	3.2	**1**	**1**	57.6	54	**1.06**	**1**	175.7	160.5	**1.09**	**1**	8.2	7.5	**1.09**	**1**
	FR	-	-	-	-	551	545	**1.01**	**1**	5.5	5.8	**0.95**	**1**	86.3	97.2	**0.88**	**1**	198.4	225.8	**0.87**	**1**	10.9	9.6	**1.13**	**1**
PR	LJ	19.5	19	**1.01**	**1**	44	37	**1.1**	**1**	21	21	**1**	**1**	-	-	-	-	80	80	**1**	**1**	11.8	11.4	**1.03**	**1**
	TW	673	664	**1**	**1**	1000	908	**1.1**	**1**	282	400	**0.7**	**1**	-	-	-	-	1128	1041	**1.08**	**1**	319	331	**0.96**	**1**
	TM	597	646	**0.92**	**1**	1399	1441	**0.97**	**1**	880	860	**1.02**	**1**	-	-	-	-	1157	1078	**1.07**	**1**	596	613	**0.97**	**1**
	FR	-	-	-	-	1023	995	**1.02**	**1**	590	577	**1.02**	**1**	-	-	-	-	601	548	**1.09**	**1**	260	280	**0.93**	**1**

by updating the path function. Table 4.2 shows the execution times of the handwritten programs (H) and synthesized programs (S), and their relative ratio (R), i.e., former divided by the latter. It also reports the edge-work ratio (ER) that is the number of edges processed by the synthesized programs divided by that processed by the handwritten versions. (The PR use-case was run until convergence.) Although the execution time is primarily dependent on the number of processed edges, it is also dependent on the efficiency of the kernel functions,

which is influenced by the number of vertex and edge variables and atomic operations. To have a more precise comparison, in Table 4.3, we further compare the number of atomic operations per edge computation, and the state maintained per vertex and edge, which are two key factors for efficiency of graph computations.

Assessment. We observe in Table 4.2 that the synthesized programs process the same number of edges compared to handwritten programs in Ligra, GridGraph and Gemini. On PowerGraph (for the use-case CC in the push model), the synthesized program process fewer edges. This is due to unnecessary processing of all the edges in the first iteration in the handwritten program which the synthesized version avoids. We also observe that the execution time is closely related to the number of processed edges. The performance of the handwritten and synthesized code is similar in most cases. The synthesized CC for PowerGraph in the push model performs 28% faster. We observe in Table 4.3 that the number of atomic operations per edge is exactly the same as that in the handwritten programs, and the size of the state per vertex and edge is minimal.

Table 4.3: Metrics for Comparing Handwritten and Synthesized Code. H: Handwritten, S: Synthesized. (PowerGraph does not require the user to write atomic operations.)

Prog.	Vertex Data Size (bytes) :: Edge Data Size (bytes)										# Atomics Per Edge									
	Ligra		GridGraph		Gemini		PowerGraph		GraphIt		Ligra		GridGraph		Gemini		PowerGraph		GraphIt	
	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S
BFS	8::0	8::0	8::0	8::0	8::0	8::0	12::0	12::0	4::0	8::0	1	1	1	1	1	1	0	0	1	1
CC	4::0	4::0	4::0	4::0	4::0	4::0	8::0	8::0	4::0	4::0	1	1	1	1	1	1	0	0	1	1
SSSP	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	1	1	1	1	1	1	0	0	1	1
WP	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	1	1	1	1	1	1	0	0	1	1
PR	4::0	4::0	4::0	4::0	8::0	8::0	8::0	8::0	4::0	4::0	1	1	1	1	1	1	0	0	0	0

Scalability. Fig. 4.18b shows the scalability of both the handwritten and synthesized code on the Ligra framework and LJ input graph for four use-cases: TRUST, DRR, LTRUST, and WSP. The speedup remains steady around 2.5×, 3.1×, 1.5× and 1.5× respectively. As we saw in § 4.5, fusion preserves, and furthermore increases, the parallelism of specifications. Moreover, the synthesized codes retain the edge- and vertex-level parallelism offered by the frameworks. They never rely on major synchronization bottlenecks (e.g., locking multiple edges or vertices at the same time). Thus, synthesized programs scale similar to handwritten programs.

4.8 Related Work

Graph Processing Frameworks. Graph processing systems provide interfaces to hide the implementation details such as parallelism, synchronization and communication in scalable runtimes. At the heart of graph computations are operations over vertex and edge values and scheduling policies to determine the order in which operations are performed. Parallelism is often extracted at the vertex and edge level, and hence, most interfaces allow computations to be directly expressed as vertex-level and edge-level operations [330, 320, 319, 188, 428, 508, 190, 509, 408, 505, 360, 220, 130, 463, 339, 462, 338]. Certain DSLs raise the abstraction level by expressing the operation in the form of sequential programs or datalog queries, in order to simplify development of graph algorithms [505, 225, 12, 405, 454, 421]. Others [110, 426, 185] focus on generating implementations of graph algorithms for different architectures such as GPUs. Unlike our synthesis process that generates codes for multiple

graph processing frameworks, these systems generate implementations that are tied to their runtime specifics. Moreover, name synthesizes the kernel functions.

Declarative Graph Processing DSLs. Fregel [151] is a domain-specific language that allows graph computations to be expressed as a higher-order function that is applied at every vertex. Its latest version compiles code to the Giraph and Pregel+. Similar to name, Fregel is declarative, models termination conditions, and applies optimizing transformations (such as tupling). Following Fregel, Palgol [504] extends Fregel's functional interface with remote data access. Similarly, s6graph [118] is a graph processing framework with a functional interface and dedicated runtime. In addition to a vertex-centric intermediate language, name presents a higher-level language for path-based computations and its semantics, formally models a comprehensive set of the common iterative models and proves the formal correctness and termination conditions for them, captures the canonical iteration-map-reduce primitives as a let form and presents several fusion optimization types that transform specifications into these primitives, combines type-directed enumerative and constrained-based synthesis to generate the iterative kernel functions, and generates implementations in five graph processing frameworks.

Elixir [386, 387] captures a graph computation as an operator on a graph neighborhood that is iteratively applied to the graph non-deterministically. It allows declarative constraints for scheduling, implementation selection, and synchronization insertion into the operators and applies automated planning to find multiple implementations. LM [126] and CLM [125] present a logic programming language for programming over graph structures and algorithms. Similar to Elixir, CLM supports declarative specification of scheduling

and partitioning policies that allows programmers to add logical rules for optimization. In contrast, name offers a more high-level specification language for path-based computations, applies fusion optimizations, formalizes correctness and termination conditions for iterative computations, and uses them to automatically synthesize the iterative kernel functions.

To simplify constructing and reasoning about programs, declarative programming [404] is applied to many domains such as compiler optimization [305], parallel programming [126] and configuration generation [219].

Program Synthesis. Program synthesis has always been an area of interest for computer scientists. Previous works have employed enumeration [452, 231], variants of syntax-guided synthesis [22] and type-guided synthesis [366, 383] to synthesize protocol snippets [452] and Excel macros [195, 197]. name's synthesis process enumerates graph processing kernel functions based on a syntax grammar for local computations.

Previous works have also used constraint solving to fill holes in program sketches [438, 437] including architectural kernel functions [481], and to synthesize control structures, imperative programs [441, 161] and program templates [51], and to compose APIs [234, 427]. The name synthesis tool applies SMT solvers to check that the candidate kernel functions satisfy the correctness conditions of the iterative models. Built on top of Fregel, [350] uses SMT solvers to optimize kernel functions. In contrast, name automatically synthesizes the kernel functions.

Superoptimization is another thread of synthesis which applies stochastic search methods to synthesize programs [340, 242, 243, 49, 419]. Moreover, Souper [416] took a step further by synthesizing superoptimizers. In contrary to superoptimization which focuses

on optimizing machine-level code, name fusion rules optimize high-level graph processing specifications.

Distributed and concurrent program synthesis. Bigλ [433] synthesizes map-reduce-style distributed programs and SCYTHE [468] synthesizes SQL queries based on the programming-by-example approach. Hamsaz [227] minimizes and synthesizes coordination between replicas in a distributed system. Transit [452] describes a distributed protocol as both symbolic and concrete execution fragments called concolic snippets, and applies solvers and user feedback to interactively generate the implementation. All three works above have different synthesis domains than name.

Previous works have synthesized concurrent programs either by inferring atomic sections and inserting synchronization primitives [72, 111, 127, 208, 456, 457], or by following semantic preserving rules to transform sequential to concurrent programs [101, 102, 100].

Fusion. Fusion is a versatile optimization technique. Loop fusion [129, 256, 390, 76] merges the bodies of loops on regular structures such as arrays and hence reduces the number of memory accesses and improves locality. Fusion also has been applied to tree structures [395, 394, 414, 415] to combine multiple phases of traversal or fuse different stages of data processing pipelines [412] to enhance data locality. Deforestation of functional programs [465, 184, 112, 239] combines a sequence of function applications into a single application and eliminates intermediate values. However, deforestation is oblivious to the primitives of graph computation. Graph computations use three fundamental primitives; thus, name structures these primitives as the triple-let term. The fusion rules transform computations to this structure and maintain it during fusion.

Chapter 5

Verified Tensor Graph Rewrite

5.1 Introduction

Deep learning (DL) is pervasive in modern day machine learning (ML) workloads. Researchers and practitioners use different topologies of neural networks such as convolutional neural networks [268], graph neural networks [258] and transformers [455] to perform various prediction tasks such as image classification [215], language modeling [136], etc. They usually express these ML computations using a tensor language, such as TensorFlow [10], PyTorch [372], and JAX [173].

Programmers usually express these computations using tensor operations (e.g., convolution, matrix multiplication, non-linear activation, etc) that are supported by modern ML frameworks. These ML frameworks then optimize the tensor computation graphs specified from the users' programs. One of the important optimizations is graph rewrite. XLA [444] is a production tensor compiler used by TesnsorFlow and JAX to generate code for different hardware backend, such as CPUs, GPUs, and TPUs [245, 244]. In the XLA

compiler, the graph rewrite optimization is known as the algebraic simplification pass[1]. It rewrites parts of the tensor computation expressed as a computation graph into an equivalent but potentially faster tensor computation. For example, under a certain precondition, an expensive dot operation may be decomposed into a simpler composition of element-wise multiplication and broadcast operations. Usually, these rewrites are algebraic in nature, and production compilers include hundreds of such rewrite patterns. For instance, in XLA's algebraic simplifier, there are more than 130 tensor rewrite rules spanning more than 7,000 lines of C++ code. XLA always executes this pass when compiling any program regardless of the backend target. Thus, it is important that we have confidence that the rewrite rules are correct.

The goal of this work is to formally prove the correctness of tensor rewrite rules that are present in a production quality compiler such as XLA. There has been multiple works on verifying tensor rewrite rules. However, they fall short in realizing our goal due to multiple reasons. TASO [235] proves that certain tensor rewrite rules are correct, but due to its axiomatic proof technique, it requires new manually-written axioms to verify the majority of the XLA tensor rewrite rules. PET [469] can represent and prove expressive tensor rewrites without relying on axioms, but its proof methodology works on only linear operations. Equally importantly, these existing systems cannot fully support verifying rules for tensors with unbounded dimension sizes and ranks. The ability to verify unbounded tensors is crucial because most algebraic simplification rules in XLA are applicable to unbounded tensors.

[1] `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/compiler/xla/service/algebraic_simplifier.cc`

We present TensorRight a verified tensor rewrite system that is able to both represent and prove the majority of the XLA tensor rewrite rules with *unbounded tensor ranks and dimension sizes*. TensorRight provides the first formalism of XLA tensor operators in a purely functional manner using the denotational semantics (Section 5.4). We introduce TensorRight DSL for implementing rewrite rules. The DSL consists of a core language specification for the XLA tensor operators and additional constructs to specify operations on dimensions, which aid in verifying the rewrites. We use the denotational semantics of operators to generate correctness verification conditions for the rewrite rules in a rank and dimension size polymorphic manner. To achieve this, we embed the semantics of each operator as an executable specification in TensorRight DSL. We generate a verification condition of a rule by symbolically executing the LHS and RHS expressions and asserting their equality. We introduce new dimension type constructs to handle unbounded dimension sizes and ranks. Overall, TensorRight is the first system to both formally specify a production compiler's tensor operators and prove rewrite rules for unbounded tensors.

We show that TensorRight is able to specify and prove the majority of the XLA rewrite rules containing complicated preconditions and tensor operator combinations (Section 5.7). Further, we generalize some of the existing rules in XLA by relaxing some of their preconditions. Note that most of these preconditions were put in by the compiler developers to simplify some of the intricate index and dimension size calculations necessary to make the rule correct in a general case. With the aid of TensorRight we believe in the future, compiler engineers can quickly iterate through complicated rules to get feedback on

their correctness leading to productivity increases and more complicated rewrite rules inside the compiler.

In summary, this paper makes the following contributions.

- We present TensorRight DSL that consists of XLA operators and other constructs on dimensions and accesses that allows users to write complex tensor rewrite rules with preconditions.

- We provide the first formal semantics of tensor operators used in the production XLA compiler.

- We introduce the first verification methodology to verify tensor rewrite rules on tensors with unbounded ranks and dimension sizes.

- We show that our tensor rewrite system can represent and prove 104 out of 123 rules of interest in the XLA's algebraic simplifier. Additionally, we show evidence on how TensorRight can be used for generalizing overly constrained XLA rewrite rules.

5.2 Motivation

The algebraic simplifer of the XLA compiler, contains hundreds of rewrite rules that get executed during compilation of programs. Currently, neither the rewrite rules nor the rewrite engine is verified. The compiler developers check the correctness of those rules via unit tests and often limit the generality of the rules to alleviate concerns of introducing compiler bugs.

Tensor rewrite rules deployed inside the XLA compiler (1) support rules that involve complex XLA operations, and (2) work with tensors of arbitrary ranks and dimension sizes. The first property is intrinsic, while the second property is important since the compiler should not miscompile on some dimension sizes. Existing verified tensor rewrite systems either cannot express or prove the tensor rewrites satisfying the above properties. We present TensorRight a verified rewrite system that can both *represent* and *prove* the rewrites used in a production compiler such as XLA for tensors with unbounded ranks and dimension sizes.

5.2.1 Example Rewrite Rule

To illustrate how TensorRight achieves its goals, let's consider the following simple rewrite rule extracted from XLA's algebraic simplifier that uses the dot (einsum) operation. Note that we have omitted some operands and details of shapes for simplicity.

$$\texttt{dot(A,B)} \Longrightarrow \mathsf{binary}(\mathsf{broadcast}\texttt{(A)}, \mathsf{broadcast}\texttt{(B)}, *) \tag{5.1}$$

The dot operation performs a sum reduction along the contracting dimensions of element-wise multiplied tensor values across tensors A and B. The output tensor has union of all non-contracting dimensions from both A and B. For the 2-dimensional case with one contracting dimension, it simply becomes the general matrix multiplication. The general N-dimensional case can have any number of contracting dimensions. The above DotToMultiply rewrite replaces an expensive dot operation with a composition of element-wise binary multiplication and broadcast operations. This rewrite is generally not correct,

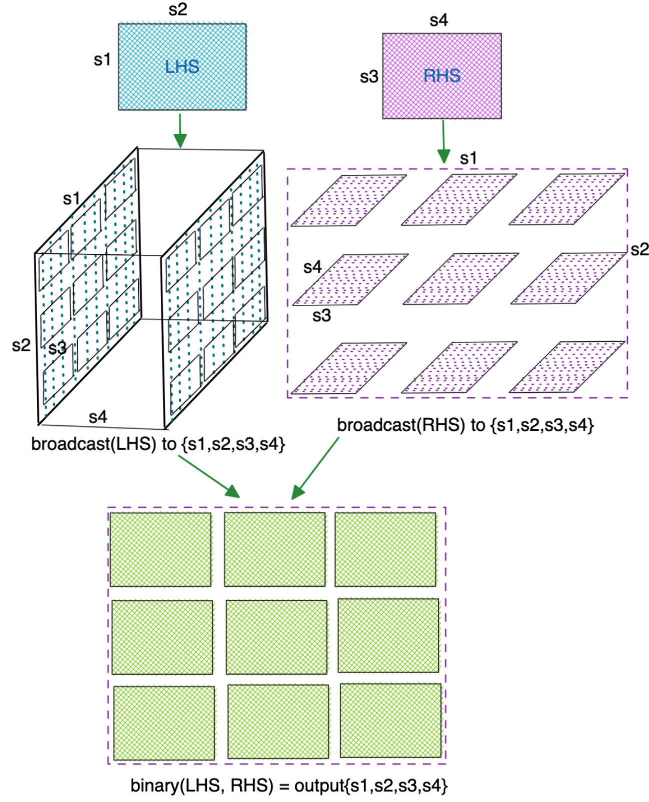
unless A and B have no contracting dimensions. Fig. 5.1a illustrate this computation on 2-dimensional input tensors. Since there are no contracting dimensions, there is no need for reduction, so this RHS tensor expression is equivalent to the LHS expression.

5.2.2 Representation and Proof

Here, we illustrate the challenges in representing and proving the DotToMultiply rewrite rule in Eq. 5.1 — to be correct for tensors A and B of unbounded dimension sizes and ranks — and show how our design of TensorRight overcomes these challenges.

Challenges in Representation First, the rewrite system should allow specifying the rule with arbitrary dimension sizes and ranks for tensors A and B. Second, it should allow specifying operations on dimensions themselves (e.g., computing the output dimensions of `dot`). Third, it should model tensor operations covering the expressivity and parameterizations in XLA. Last, it should allow defining a precondition of a rule (e.g., no contracting dimension).

TensorRight Representation TensorRight DSL models XLA operations and allows the developers to specify the DotToMultiply rule with the parametric form shown in Fig. 5.1b. Instead of storing dimension sizes as an array or a list, we use *dimension maps*. This allows us to perform operations supported by *map*s to operate on dimensions (key) and their sizes (value). If the map is left uninterpreted and represented using meta variables, the rule representation becomes both rank and dimension size polymorphic. In Fig. 5.1b, s, s', s'' are shapes of tensors A, B, and the output of the `dot` operator respectively, while d, d', d'' denote their dimension maps. The rule uses `Dims` function to retrieve the dimension maps



(a) Illustration of the DotToMultiply rule

DotToMultiply :

let $d'' = \mathsf{Dims}(s'')$ in

$$\mathsf{dot}_{s''}(A_s, B_{s'}, \emptyset, \emptyset) \Longrightarrow \mathsf{binary}(\mathsf{broadcast}(A, \mathsf{Id}, d''), \mathsf{broadcast}(B, \mathsf{Id}, d''), *)$$

FoldConvInputPad :

let $m_{ol} = m_l + m_{lp}$ in

let $m_{oh} = m_h + m_{hp}$ in

$$\mathsf{convolution}(\mathsf{pad}(t, 0, m_{lp}, m_{hp}, m_{ip}), t', i_b, i_{if}, i_{of}, m_l, m_h, m_i, m_i')$$

$$\Longrightarrow_{m_{ip}=0 \land m_i=1}$$

$$\mathsf{convolution}(t, t', i_b, i_{if}, i_{of}, m_{ol}, m_{oh}, m_{oi}, m_i')$$

(b) Example Rewrite Rules in TensorRight DSL

Figure 5.1: Motivating examples

from their corresponding shapes. The rule's RHS broadcasts both tensors A and B with d'' to have the same shape as the output of the dot. Note that we supply the empty set of dimensions as the final two arguments to dot in the LHS, indicating the precondition that there must be no contracting dimensions.

TensorRight supports much richer rewrites with complex preconditions. Consider the FoldConvInputPad shown in Fig. 5.1b. The idea behind the rule is simple: to fold the padding operator into the convolution operator. However, computing how this external padding composes with existing padding in the convolution operator is non-trivial. Since TensorRight DSL uses dimension maps, we can express these padding sizes (m_{ol}, m_{oh}) using a series of map operations. Further, we can represent and prove even a more general version of this rule with interior padding (Section 5.7).

Challenges in Proving Soundness A desired verified tensor rewrite system should be able to prove all or at least a large proportion of rules. Most widely adopted approach in tensor graph rewrite proofs is an axiomatic approach, where the rewrite language designer comes up with a sufficient set of lemmas about the tensor operators, which are then used to prove increasingly complex rules. However, this approach requires the designer to update the core set of lemmas when new rules are introduced and can quickly become cumbersome.

TensorRight Proof Strategy We introduce a new approach based on denotational semantics of tensor operators to prove rules with unbounded rank and dimension sizes for input tensors, which we describe briefly in Section 5.5 before diving into details later. Typically, the number of tensor operators is much fewer than the number rewrite rules,

making our approach more scalable than a purely axiomatic approach. As shown in § 5.7, we can prove a majority of the rules in XLA's algebraic simplification pass for unbounded ranks and dimension sizes.

5.3 Overview

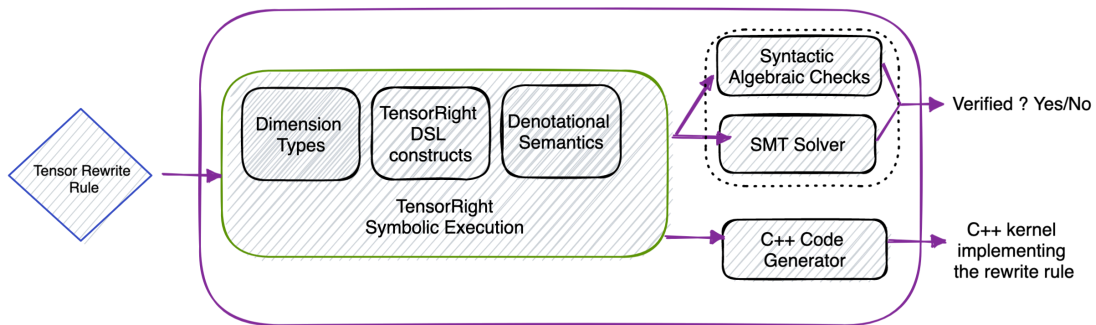


Figure 5.2: TensorRight overview and workflow

Fig. 5.2 shows the overview of TensorRight components. The denotational semantics of operators (§ 5.4) are implemented as executable specifications, embeded in TensorRight DSL. Rewrite rules written in TensorRight DSL are programs that symbolically execute the operators' specifications and generates symbolic representations of the LHS and RHS expressions. The verification condition of a rule is simply the assertion of equality between the LHS and RHS expressions (§ 5.5). The dimension types in TensorRight DSL allow the proof system to support unbounded tensor dimension sizes and ranks.

Our system design provides additional compelling features. First, the user can use TensorRight to concretely execute tensor expressions to intuitively understand what preconditions may be needed to prove a rule. This increases user productivity. Second, since the rules are represented in a declarative form, we can automatically generate imperative C++

code that matches and rewrites patterns in computational graphs using this rule specification. We have built a prototype C++ code generator to showcase this functionality. This allows TensorRight to automatically translate new rewrite rules written in TensorRight DSL to C++ code, which can be integrated (literally copied) into the XLA compiler itself.

5.3.1 TensorRight Rewrite Rules

Similar to many other tensor rewrite systems, TensorRight rewrites are modeled as rewriting a LHS tensor expression to a RHS tensor expression, subject to certain preconditions, which the LHS expression should satisfy. The users use the constructs provided by TensorRight DSL to write both tensor expressions and the preconditions. Preconditions are predicates that operate on the LHS tensor expression. We use the notation $t_{LHS} \Rightarrow_C t_{RHS}$ to represent a generic tensor rewrite, where t_{LHS} and t_{RHS} are the LHS and RHS tensor expressions respectively, and C are the set of preconditions under which the rewrite rule is supposedly correct, which is verified by our system.

5.3.2 TensorRight Language Constructs

The TensorRight DSL provides constructs to define input tensors, dimensions, accesses and perform operations on them. Here, we look at some of the constructs briefly before we present more thorough definitions in Section 5.5.

Dimensions. TensorRight represents dimensions as mapping from dimension names (shown with i) to their sizes. For example, consider a 3-dimensional tensor t with three dimensions i_0, i_1 and i_2 with sizes of 3, 4, and 5, respectively. We represent dimension map of t as $\{i_0 \mapsto 3, i_1 \mapsto 4, i_2 \mapsto 5\}$. Dimension maps remain uninterpreted during our definition

of denotational semantics. During symbolic execution, we introduce the concept of *aggregate dimensions* (Section 5.5.1) to reason about a collection of dimensions simultaneously and use symbolic variables to represent dimension sizes. This way we support verification of rewrites on unbounded tensor rank and dimension sizes.

Accesses. Accesses allow us to retrieve an element from a tensor at a particular location. We model accesses as a map a from dimension names to access value in each dimension. For example, let's consider the same 3-dimensional tensor t with dimension map $\{i_0 \mapsto 3, i_1 \mapsto 4, i_2 \mapsto 5\}$. An access map of $a = \{i_0 \mapsto 2, i_1 \mapsto 3, i_2 \mapsto 4\}$ retrieves the element at the second location in i_0, third location in i_1, and forth location in i_2.

Tensors and Operations. We model tensors as a mapping $\overline{a \mapsto v}$ from accesses a to values v of type integer or real. We then model tensor operations in the core TensorRight DSL langauge and define the denotational semantics for the language in § 5.4.

5.3.3 Proving Correctness

We define the denotational semantics of TensorRight DSL in § 5.4. Given the semantic rules, TensorRight generates the verification conditions required to prove the correctness of the rewrite rule. Given the rule $t_{LHS} \Rightarrow_C t_{RHS}$, our proof obligation states that if the precondition C is satisfied, the semantics of t_{LHS} and t_{RHS} must be equal. More specifically, TensorRight symbolic execution engine generates the verification conditions by interpreting the LHS and RHS tensor expressions under a general access with symbolic indices chosen from the domain of accesses of the two expressions. Then, TensorRight asserts the equality of the interpreted results from LHS and RHS as a verification condition. For some rules that require reasoning about reductions, we develop proof extensions via a

few lemmas to make the verification feasible as detailed in Section 5.5. Note that, a vast majority of rules are proven without using these lemmas.

5.4 TensorRight DSL & Semantics

This section first presents the core syntax of TensorRight DSL — which closely models the core syntax of XLA operators — and then the denotational semantics of the language.

5.4.1 Core Syntax

We capture the core syntax of TensorRight DSL in Fig. 5.3. Values that tensors store can be of either Integer, Real, or Boolean types τ. We denote primitive literals as $\mathbf{n}$ and we use the bold font to represent literals of different types. An integer term n can be an integer literal, or an application of a binary operator to two integers. We use i to represent dimension indices or names. The indices are merely labels for dimensions, and they don't enforce particular ordering of dimensions. We note that this representation of dimension differs from that of XLA, where the dimensions of the tensors are represented as an ordered list.

Maps m are an essential part of the language. A dimension map d maps a dimension index i to the size of the dimension. An access map a specifies access values for each dimension inside a tensor. We represent maps m (also denoted as d for dimension maps and a for access maps) as a set of finite mappings $\overline{i \mapsto e}$ of indices i to expressions e. The map $m[i \mapsto e]$ updates the value of m at index i to the new value e. Other operations on maps include

τ	$:=$	$\mathsf{Int} \mid \mathsf{Real} \mid \mathsf{Bool}$	Type
$\mathbf{n} : \mathcal{N}$			Primitive Lit
n	$:=$	$\mathbf{n} \mid n \oplus n$	Integer
i			Index
$\mathbf{m} : \mathcal{M}$	$=$	$\{i \,..\, i\} \to \mathcal{N}$	Map Lit
m, d, a	$:=$	$\mathbf{m}$	Map
		$\mid \overline{i \mapsto e} \mid m[i \mapsto e]$	
		$\mid \ \odot m \mid m\vert_{\bar{i}} \mid m^{-1}$	
		$\mid m \circ m$	
		$\mid m \oplus m \mid m \oplus n$	
		$\mid m \cup m \mid m \setminus m$	
		$\mid \mathsf{Dims}(s)$	
$\mathbf{s}$	$:=$	$\langle \mathbf{d},\ \tau \rangle$	Shape Lit
s	$:=$	$\langle d,\ \tau \rangle$	Shape
$\mathbf{t}_\mathbf{s}$	$:=$	$\mathsf{Const}_\mathbf{s}(\mathbf{n})$	Tensor Lit
$t_\mathbf{s}$	$:=$	$\mathbf{t}_\mathbf{s} \mid o$	Tensor Expr
f	$=$	$a \to a$	Access
			Transformer

e	$:=$	$t_\mathbf{s}$	Expr
		$\mid n \mid i \mid m(i)$	
		$\mid \bar{e} \mid t_\mathbf{s}[a]$	
		$\mid \mathsf{dom}(m) \mid \mathsf{range}(m)$	
o	$:=$	$\mathsf{binary\text{-}op}(t_\mathbf{s}, t_\mathbf{s}, \oplus)$	Operation
		$\mid \mathsf{transpose}_\mathbf{s}(t_\mathbf{s}, m)$	
		$\mid \mathsf{expand}(t_\mathbf{s}, d)$	
		$\mid \mathsf{broadcast}(t_\mathbf{s}, m, d)$	
		$\mid \mathsf{slice}_\mathbf{s}(t_\mathbf{s}, m, m, m)$	
		$\mid \mathsf{dy\text{-}slice}_\mathbf{s}(t_\mathbf{s}, m, m)$	
		$\mid \mathsf{dy\text{-}up\text{-}slice}(t_\mathbf{s}, t_\mathbf{s}, m)$	
		$\mid \mathsf{pad}_\mathbf{s}(t_\mathbf{s}, e, m, m, m)$	
		$\mid \mathsf{iota}(d, i)$	
		$\mid \mathsf{reduce}_\mathbf{s}(t_\mathbf{s}, i, e, \oplus)$	
		$\mid \mathsf{reshape}(t_\mathbf{s}, d, f)$	
		$\mid \mathsf{concat}(t_\mathbf{s}, t_\mathbf{s}, i)$	
		$\mid \mathsf{conv}(t_\mathbf{s}, t_\mathbf{s}, i, i)$	
		$\mid \mathsf{dot}(t_\mathbf{s}, t_\mathbf{s}, \bar{i}, \bar{i})$	

Figure 5.3: Core Specification Language

unary and point-wise binary operation on values of indices. A map m can also be the result of applying an binary operation to a map and an integer term. The two operations inverse (m^{-1}) and composition ($m \circ m$) can be applied to maps whose values are indices as well. As an example, the composition of m with its inverse m^{-1} is simply the identity Id map. The projection $m|_{\bar{i}}$ restricts the domain of m to the set of indices $\bar{i}$. Adding new elements to a map is simply done by getting the union of the two maps. We note that for a union

operation to be valid, our semantics require that the added keys must be fresh indices that are not already present in the map. Removing from a map is done with the set minus ($\setminus$) operator. Finally, $\mathsf{Dims}(s)$ retrieves the dimension map from the shape s. We define the shape of a tensor s to be the pair of its dimension map d and the type elements that it stores. We decorate language terms with their shapes s as a subscript.

A tensor expression t is either a constant tensor replicating value $\mathbf{n}$, or the result of an operation o. We will describe operations and their semantics in the next subsection. Finally, an expression e is either a tensor t, a number n, an index i, an indexing $m(i)$ of a map m at an index i, an accessing $t[a]$ of a tensor t with access map a, a set of expressions $\overline{e}$, or the domain $\mathsf{dom}(m)$ or range $\mathsf{range}(m)$ of a map m. Finally, we define f as an access transformer function which defines how an input access map is transformed to another access map. This transformer function is used in the reshape operator which we will visit later.

5.4.2 Denotational Semantics

We define the denotational semantics of the specification language in Fig. 5.4. A map $\overline{i \mapsto v}$ maps indices i to values v (that can be integers or indices). Indices can be general names that include integers. A tensor $\overline{a \mapsto v}$ with dimension map d, is an unbounded finite map from accesses a of d to values v (that can be integers, reals or booleans). A dimension map is a mapping from a set of indices, called dimensions, to their sizes. The rank of a tensor is the size of its dimension map. Given a dimension map d, an access is a map from each dimension in d to a number less than the size of that dimension. For a dimension map d, we denote the set of accesses as $\mathsf{A}(d)$. The dimension map of a tensor t is denoted as

$$\textsc{BinOp}\quad \frac{t = [\![e]\!] \qquad t' = [\![e']\!] \qquad \mathsf{D}(t) = \mathsf{D}(t') \qquad d = \mathsf{D}(t)}{[\![\mathsf{binary}(e, e', \oplus)]\!] = \{a \mapsto t[a] \oplus t'[a] \mid a \in \mathsf{A}(d)\}}$$

$$\textsc{Trans}\quad \frac{t = [\![e]\!] \qquad d = \mathsf{D}(t) \qquad d' = d \circ m^{-1}}{[\![\mathsf{transpose}(e, m)]\!] = \{a \mapsto t[a \circ m] \mid a \in \mathsf{A}(d')\}}$$

$$\textsc{Expand}\quad \frac{t = [\![e]\!] \qquad d = \mathsf{D}(t) \qquad \bar{i} = \mathsf{dom}(d') \setminus \mathsf{dom}(d) \qquad d \subseteq d'}{[\![\mathsf{expand}(e, d')]\!] = \{a' \mapsto t[a' \setminus a'|_{\bar{i}}] \mid a' \in \mathsf{A}(d')\}}$$

$$\textsc{Slice}\quad \frac{t = [\![e]\!] \qquad d = \mathsf{D}(t) \qquad m_s \geq 0 \qquad m_e < d \qquad d_o = \left\lceil \frac{m_e - m_s}{m_p} \right\rceil}{[\![\mathsf{slice}(e, m_s, m_e, m_p)]\!] = \{a \mapsto t[m_s + a \times m_p] \mid a \in \mathsf{A}(d_o)\}}$$

$$\textsc{Reduce}\quad \frac{t = [\![e]\!]}{[\![\mathsf{reduce}(e, \bar{i}, \oplus)]\!] = \mathsf{Reduction}(t, \bar{i}, \oplus)}$$

$$\textsc{Iota}\quad [\![\mathsf{iota}(d, i)]\!] = \{a \mapsto a(i) \mid a \in \mathsf{A}(d)\}$$

$$\textsc{Pad}\quad \frac{\begin{array}{c} t = [\![e]\!] \qquad d = \mathsf{D}(t) \qquad n = [\![e_v]\!] \\ d_i = d + (m_i \times (d-1)) \qquad d_l = d_i + m_l \qquad d_o = d_l + m_h \\ \textit{int-pad} = \lambda a. \bigvee_{j \in \mathsf{dom}(a)} (a|_j - m_l|_j) \ \%\ (m_i|_j + 1) \ \wedge\ m_i|_j \neq 0 \end{array}}{\begin{array}{c} [\![\mathsf{pad}(e, e_v, m_l, m_i, m_h)]\!] = \\ \{a \mapsto n \mid a \in \mathsf{A}(d_o) \wedge \bigvee_{i \in \mathsf{dom}(d)} a|_i < m_l|_i\} \cup \\ \{a \mapsto t[\frac{a - m_l}{m_i + 1}] \mid a \in \mathsf{A}(d_o) \wedge \neg \textit{int-pad}(a)\} \cup \\ \{a \mapsto n \mid a \in \mathsf{A}(d_o) \wedge \textit{int-pad}(a)\} \cup \\ \{a \mapsto n \mid a \in \mathsf{A}(d_o) \wedge \bigvee_{i \in \mathsf{dom}(d)} a|_i \geq d_l|_i\} \end{array}}$$

$$\textsc{Reshape}\quad \frac{t = [\![e]\!] \qquad d = \mathsf{D}(t) \qquad k = |d| \qquad k' = |d'| \qquad \Pi_{i=0}^{k-1} d(i) = \Pi_{i=0}^{k'-1}\ d'(i)}{[\![\mathsf{reshape}(e, d', f)]\!] = \{a \mapsto t[f(a)] \mid a \in \mathsf{A}(d')\}}$$

$\mathsf{D}(t)$. Finally, given a tensor t, its dimension map $\mathsf{D}(t)$ can be calculated as follows. Each dimension i is mapped to the largest value for i in all accesses of t.

Concat

$$\frac{\begin{array}{c} t = [\![e]\!] \qquad t' = [\![e']\!] \\ d = \mathsf{D}(t) \qquad d' = \mathsf{D}(t') \\ d \setminus d|_i = d' \setminus d'|_i \qquad \overline{i'} = \mathsf{dom}(d') \setminus i \\ d_o = (d \setminus d|_i) \cup (d|_i + d'|_i) \end{array}}{\begin{array}{c} [\![\mathsf{concatenate}(e, e', i)]\!] \ = \\ \{a \mapsto t[a] \mid a \in \mathsf{A}(d_o) \ \wedge a|_i < d|_i\} \cup \\ \{a \mapsto t'[a - (d|_i \cup \overline{i' \mapsto 0})] \mid a \in \mathsf{A}(d_o) \ \wedge a|_i \geq d|_i\} \end{array}}$$

DyUpdateSlice

$$\frac{\begin{array}{c} t = [\![e]\!] \qquad t' = [\![e']\!] \\ d = \mathsf{D}(t) \qquad d' = \mathsf{D}(t') \qquad m_s \geq 0 \end{array}}{\begin{array}{c} [\![\mathsf{dy\text{-}up\text{-}slice}(e, e', m_s)]\!] \ = \\ \{a \mapsto t[a] \mid a \in \mathsf{A}(d) \wedge a < m_s\} \cup \\ \{a \mapsto t'[a - m_s] \mid a \in \mathsf{A}(d) \wedge m_s \leq a < d'\} \cup \\ \{a \mapsto t[a] \mid a \in \mathsf{A}(d) \wedge m_s + d' \leq a\} \end{array}}$$

Dot

$$\frac{\begin{array}{c} t_1 = [\![e_1]\!] \qquad d_1 = \mathsf{D}(t_1) \qquad t_2 = [\![e_2]\!] \qquad d_2 = \mathsf{D}(t_2) \\ d_1|_{\bar{i}} = d_2|_{\bar{i}} \qquad d_1' = d_1 \setminus d_1|_{\bar{i}} \qquad d_2' = d_2 \setminus d'|_{\bar{i}} \qquad d_o = d_1' \cup d_2' \cup d_1|_{\bar{i}} \\ t_1' = [\![\mathsf{broadcast}(t_1, \mathsf{Id}, d_o)]\!] \qquad t_2' = [\![\mathsf{broadcast}(t_2, \mathsf{Id}, d_o)]\!] \end{array}}{[\![\mathsf{dot}(e_1, e_2, \bar{i})]\!] \ = \ \mathsf{Reduction}([\![\mathsf{binary}(t_1', t_2', \times)]\!], \bar{i}, +)}$$

Conv

$$\frac{\begin{array}{c} t_1 = [\![\mathsf{pad}(e_1, 0, m_h, m_l, m_i)]\!] \qquad t_2 = [\![\mathsf{pad}(e_2, 0, \overline{_ \mapsto 0}, \overline{_ \mapsto 0}, m_i')]\!] \\ d_1 = \mathsf{D}(t_1) \qquad d_1' = d_1 \setminus d_1|_{\{i_b, i_{if}\}} \qquad d_2 = \mathsf{D}(t_1') \qquad d_2' = d_2 \setminus d_2|_{\{i_{if}, i_{of}\}} \\ d_o = \{i_b \mapsto d_1(i_b), i_{if} \mapsto d_2(i_{if}), i_{of} \mapsto d_2(i_{of})\} \cup (d_1' - d_2' + 1) \\ t_1' = [\![\mathsf{broadcast}(t_1, \mathsf{Id}, d_1 \cup \{i_{of} \mapsto d_2(i_{of})\})]\!] \qquad t_2' = [\![\mathsf{broadcast}(t_2, \mathsf{Id}, d_2 \cup \{i_b \mapsto d_1(i_b)\})]\!] \\ window = \lambda a.\ [\![\mathsf{slice}(t_1', a, a + t_2')]\!] \qquad mul = \lambda a.\ [\![\mathsf{binary}(window(a), t_2', \times)]\!] \\ reduced = \{a \mapsto \mathsf{Reduction}(mul(a), \mathsf{dom}(d_2'), +) \mid a \in \mathsf{A}(d_o)\} \end{array}}{[\![\mathsf{convolution}(e_1, e_2, i_b, i_{if}, i_{of}, m_h, m_l, m_i, m_i')]\!] \ = \ \mathsf{Reduction}(reduced, i_{if}, +)}$$

Figure 5.4: Denotational semantics of the core language.

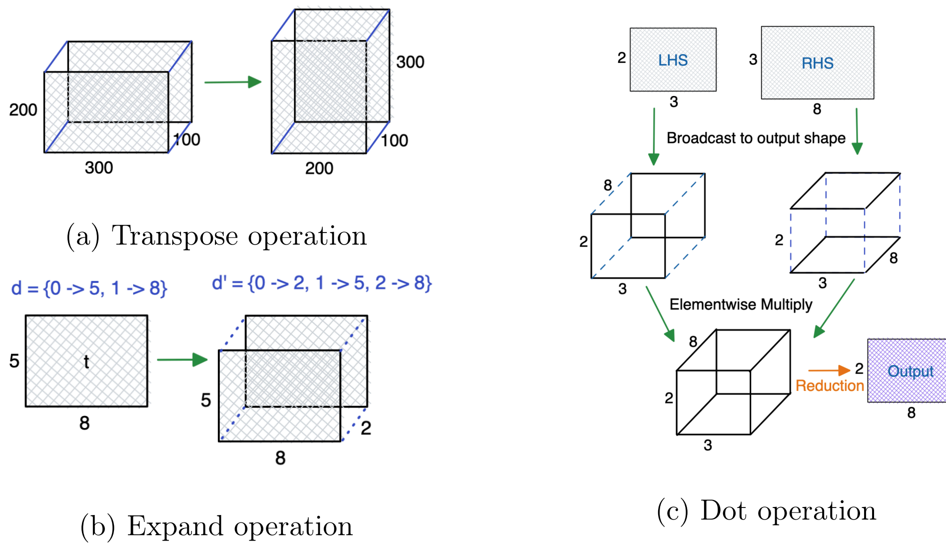
The semantics of maps and numbers are standard. The operator $\oplus$ is simply lifted to two maps by point-wise application. The operation $m \oplus n$ on the map m and an integer n, first lifts n into a map of the same domain as m, before point-wise application of $\oplus$.

Tensor Operations. Next, we explain the semantics of each operation:

- BinOp. The semantics of the binary operator $\oplus$ applied to t and t' is straightforward and is valid only if the operands have the same dimension maps. For each accesses a, the result is the pair-wise application of $\oplus$ to the value of t and t' at a. As evident in this rule, our semantics is rank and size polymorphic.

- Trans. The transpose operation is a general multi-dimensional version of the matrix transpose operation. The argument to this operation is a permutation map m which specifies the mapping from the input dimensions d to the output dimensions. More specifically, m maps every dimension i in the input to some index in the output. Therefore, to calculate the output dimension map d', the rule composes d with the inverse of the permutation map, namely, m^{-1}. The rationale is that, for every index of d', the size is found by looking up that index in d. Similarly, an access a of the output tensor is mapped to the value in t at the access $a \circ m$, which composes a with m. We note that for a transpose operation to be valid, the composition operation $d \circ m^{-1}$ must be well-defined, i.e, the domain of m must match the domain of d. To understand the transpose operation, let's consider the example tensor t with dimension map $d = \{i_0 \mapsto 100, i_1 \mapsto 200, i_2 \mapsto 300\}$. Assume dimensions i_0, i_1, and i_2 are depth, height, and width of t, in order. Fig. 5.5a shows the result of $\mathsf{transpose}(t, \{i_0 \mapsto i_0, i_1 \mapsto i_2, i_2 \mapsto i_1\})$.

- Expand. Given the new dimension map d', the expand operation adds the dimensions with indices $\bar{i}$ to the existing dimension map d such that the dimensions $\bar{i}$ contain a copy of the data in the input tensor. For the expanded output tensor, an access a takes the value of the input tensor t at the access $a \setminus a|_{\bar{i}}$ which is the result of removing the added mappings with domain $\bar{i}$ from a. Fig. 5.5b depicts an example of the expand operation $\mathsf{expand}(t, \{i_0\}, \{i_0 \mapsto 2, i_1 \mapsto 5, i_2 \mapsto 8\})$ on the 2D input tensor t with the dimension map $d = \{i_1 \mapsto 5, i_2 \mapsto 8\}$. For the sake of presentation, assume that indices i_1 and i_2 represent the row and column of t respectively. We note that in general, dimensions may not be not be a complete range of numbers, and may not be ordered. Also, assume that the new index i_0 will be the depth of the output tensor. As shown in Fig. 5.5b, the output is a 3D tensor constructed by copying the 2D input tensor t across the depth.



(a) Transpose operation

(b) Expand operation

(c) Dot operation

Figure 5.5: Illustration of the (a)expand, (b)$\mathsf{transpose}$, and (c)dot operations.

- Broadcast. The core operations introduced above allow us to capture the broadcast operation $\mathsf{broadcast}(t, m, d')$ as a syntactic sugar: $\mathsf{transpose}(\mathsf{expand}(t, d'), m)$, a

transpose applied to an expand operation. It first expands the given tensor to the new dimension mapping d'. Then, it transposes the dimensions based on the given permutation map m.

- SLICE. The slice operation $\mathsf{slice}(e, m_s, m_e, m_p)$ extracts a sub-tensor from the input tensor t. The sub-tensor is of the same rank as t and contains the values inside a bounding box within t. The start indices of the bounding box is given by the map m_s. Similarly, the end indices are given by the map m_e. Finally, the stride map, m_p, specifies which elements to pick within the bounding box. For example, if the value of the stride map is 1 for a dimension, no element is skipped in that dimension; if the value is 2, every other element is skipped in that dimension. Therefore, each access a of the output tensor, is mapped to the access $m_s + a \times m_p$ in the input tensor t.

- DYSLICE. The slice operation introduced above allow us to capture the dynamic slice operation $\mathsf{dy\text{-}slice}(t, m, m')$ as a syntactic sugar: $\mathsf{slice}(t_s, m, m + m', \overline{\mathsf{dom}(d) \mapsto 1})$ where $d = \mathsf{dom}(\mathsf{Dims}(s))$. It performs a slice operation with m as start indices, $m + m'$ as the end indices, and the stride 1 for all the dimensions.

- IOTA. The iota operation creates a tensor with the given dimension map d as the output dimension. The values of the output tensor along the dimension i start at zero and increment by one, which is given by $a(i)$, the value of the access a at dimension i.

- PAD. The pad operation increases individual dimension sizes of the given tensor. Consider a tensor t with dimension map d. To specify the amount of padding for each dimension, this operation takes three maps m_l, m_h, and m_i representing the amount of low padding, high padding and interior padding, respectively. We note that the interior padding

logically occurs *before* low and high padding. The total number of elements after the interior padding is performed, namely d_i is equal to $d + m_i \times (d-1)$. Similarly, d_l and d' calculate the dimension maps after the low and high padding are performed. Given an access a in the output tensor, *int-pad* function checks if the accessed element is part of the interior padded elements. For an access a in the output dimension map d_o, there are the following possibilities: If a is less than m_l or greater than or equal to d_l, the result is the pad value n. If a falls within m_l and d_l ($m_l \leq a < d_l$) and is part of the interior padding, the output is the pad value n. Otherwise, the output is the element at $\frac{a-m_l}{m_i+1}$ in t.

- Reshape. The reshape operation $\mathsf{reshape}(t, d', f)$ reshapes the input tensor t to the given dimension map d'. The operation gets an access transformer function f which transforms accesses of the output tensor to their corresponding accesses in the input tensor t.

Reduction

$$\frac{d = \mathsf{D}(t) \qquad d_o = d \setminus d|_{\bar{i}} \qquad \bar{i} = \{i_1, \ldots, i_k\} \qquad elem = \lambda a, \overline{n}.\ a \cup \overline{i \mapsto n}}{\mathsf{Reduction}(t, \bar{i}, \oplus) \ = \ \{a \mapsto \bigoplus_{j_1=0}^{d(i_1)-1} \cdots \bigoplus_{j_k=0}^{d(i_k)-1} t[elem(a, j_1 \ldots j_k)] \mid a \in \mathsf{A}(d_o)\}}$$

Figure 5.6: Reduction

- Concat. The concatenate operation constructs the output tensor by concatenating its operands t and t' along the dimension i. The operation is valid only if the rank of the operands are the same and all the dimensions except i have the same sizes, i.e., $d \setminus d|_i = d' \setminus d'|_i$. The output dimension map d_o is the union of the non-concatenating dimensions $d \setminus d|_i$, with the new concatenated dimension, $d|_i + d'|_i$. Therefore, accesses a of the output are mapped to t or t' based on the value of the a at i.

- DYUPDATESLICE. The dynamic update slice operation $\mathsf{dy\text{-}up\text{-}slice}(t, t', m_s)$ updates a slice of the input tensor t with the update tensor t' where m_s specifies the starting point of the slice to be updated. This operation is valid only if m_s is greater than or equal to zero. Accesses of the output tensor that fall outside t', i.e, accesses less than m_s or beyond t' boundaries, are mapped to the same elements in t. Otherwise, accesses are mapped to elements of t'.

- REDUCE. The semantics of the $\mathsf{reduce}(t, \bar{i}, \oplus)$ operation is represented as the Reduction function defined in Fig. 5.6. It applies the reduction function $\oplus$ to all the elements of the tensor t along the reduction dimensions $\bar{i}$. Therefore, the dimension map of the output d_o will not contain the reduced dimensions $\bar{i}$. Concretely, every access a in d_o is mapped to the result of applying $\oplus$ operations to elements of t that are accessed by all extensions of a.

- DOT. The dot operation $\mathsf{dot}(t_1, t_2, \bar{i})$ performs a sum reduction of the point-wise multiplication of tensors t_1 and t_2 over the specified contracting dimensions $\bar{i}$. (We note that the XLA's implementation of the dot operation accepts two sets of contracting dimensions for t_1 and t_2. For simplicity, we assume that the dimension names representing contracting dimensions in t_1 and t_2 are the same. This assumption can be easily lifted.) Let d_1 and d_2 be the dimension maps of t_1 and t_2. The condition $d_1|_{\bar{i}} = d_2|_{\bar{i}}$ checks that the contracting dimensions have the same sizes. The spatial dimensions, i.e. the dimensions that are left unchanged, of t_1 and t_2 are represented as the dimension maps d'_1 and d'_2 respectively. The output dimension map d_o is the union of d'_1 and d'_2, and the contracting dimension map $d|_{\bar{i}}$. In order to make the sizes of the two tensors compatible, the tensors t'_1 and t'_2 are calculated as the broadcast of t_1 and t_2 with the output dimension map d_o (and simply, the identity

function Id as the permutation map). Finally, the result of the dot operation is calculated as first, the binary multiplication of t'_1 and t'_2, and then reduction over the dimensions $\bar{i}$.

- CONV. We now consider a convolution operation $\mathsf{convolution}(t_1, t_2, i_b, i_{if}, i_{of}, m_h, m_l, m_i, m'_i)$. A convolution can be thought as a kernel tensor that is moved across and applied to an activation tensor. The activation tensor t_1 has multiple batches of input at the dimension i_b. For each batch, the values for the input features to be processed are at dimension i_{if}. The kernel tensor t_2 contains the window tensor at dimension i_{if}. It further includes output features at dimension i_{of} that we explain below. A point-wise computation is performed on the overlapping parts of t_1 and t_2, followed by a reduction. Finally, the resulting tensor is reduced over the input feature dimension i_{if}.

The maps m_l, m_h, and m_i specify the amount of low padding, high padding and interior padding to be applied to t_1 respectively, while m'_i specifies the amount of interior padding for t_2. Note that the kernel t_2 does not need low and high padding. The pad value is always zero. The rule applies the pad operation with the given arguments to t_1 and t_2 to obtain the padded tensors.

Next, the spatial dimension map of t_1 and t_2, namely, d'_1 and d'_2, are calculated by removing i_b and i_{if} from d_1, and removing i_{if} and i_{of} from d_2, respectively. The output tensor has a spatial dimension map of $d'_1 - d'_2 + 1$, which is calculated by sliding the kernel over the activation. (We note that the XLA's convolution operation accepts a stride map for the kernel, which specifies the gap while sliding the kernel). The complete dimension map of the output d_o is calculated by inserting i_b, i_{if}, and i_{of} into the spatial dimension map. Similar to the dot operation, the broadcast is performed to make operands compatible. For

every access a in d_o, we first calculate mul, the binary multiplication of the broadcast kernel t'_2, and a window of t'_1 starting at a, over the spatial dimensions of the kernel $\mathsf{dom}(d')$. We then calculate the *reduced* tensor as the result of the sum reduction of mul over the domain of d'_2. The final result is calculated by applying a sum reduction to the *reduced* tensor over i_{if}.

5.5 Verification of Rewrite Rules

This section describes how TensorRight verifies equivalence of tensor expressions. We specify the rules to be verified in a solver-aided DSL that we implemented in Rosette [448, 450]. The DSL implements the tensor semantics as well as normalization lemmas for tensor expressions containing reduction operators. Symbolic execution of the rule proves some properties directly during symbolic execution, and generates the verification condition of remaining properties to be solved by an SMT solver.

We first describe the DSL, focusing on dimension types which model tensors with unbounded ranks. Next, we describe the steps of the verification condition generation process driven by symbolic execution. Finally, we describe how we prove the normalization lemmas for expressions with reduction operators.

5.5.1 Symbolic Dimension Types

To model tensors of unknown and unbounded ranks, our tensor semantics in § 5.4 relies on maps of unbounded sizes. In principle, we could implement the unbounded rank using a theory of maps in SMT. For practical reasons, we implement custom maps tailored

to symbolic execution in order to reason about the unbounded nature of these maps entirely during the symbolic execution, instead of deferring the reasoning to the SML solver. These maps are presented to rule authors as symbolic dimension types, described below. In order to simplify explanation, we use Rosette s-expression notations.

Dimension Types. We use two types of dimensions to represent tensors with unknown ranks. `SDim` is a single dimension of a tensor, whereas `ADim` is an aggregate dimension that contains a set of `SDim`s. The composition of dimensions in an `ADim` can be left unspecified, in which case the `ADim` stands for an unspecified number of dimensions. Tensors with such dimensions are rank-polymorphic, and proofs involving such tensors are valid for all possible instantiations of the `ADim`. Dimensions are uniquely named (e.g., `(SDim "d1")`, `(ADim "d2")`) and tensor operations check that argument tensors have valid dimensions. For example, the element-wise binary operation `(*` t_1 t_2`)` will check that t_1 and t_2 include the same set of dimensions.

Accesses. To prove that two tensors are equivalent, we verify that all possible accesses to the tensors produce the same value. An access structure maps a dimension to a symbolic variable. For example, to index into the dimension `(SDim "d1")` using symbolic index `i`, we create the access structure `(Acc (SDim "d1") i)`. During the symbolic access to the tensor, we assert the index to be within the range of its dimension size.

Tensors. We model tensors as uninterpreted functions from symbolic accesses to symbolic integer or real values ($T : Acc \mapsto Int/Real/Bool$). We use a custom implementation of uninterpreted functions, instantiating new values lazily for each symbolically different access to the tensor.

Tensor Operators. We implement TensorRight DSL operators as Rosette functions that accept symbolic tensors and output a new symbolic tensor that represents the result of the operation. We compose tensor operators by composing functions.

5.5.2 The Verification Approach

During the symbolic execution of the LHS and RHS expressions of a rule, TensorRight decomposes the verification into three kinds of checks.

Checks performed during symbolic execution Each tensor operator constructs the dimension map of the output tensor from the dimension maps of the input tensor(s). TensorRight then checks that the final LHS and RHS tensors have the same dimensions. This check is performed entirely by the symbolic execution engine rather than by the solver.

Normalization of reduction expressions The reduction operator is uninterpreted in our denotational semantics to avoid delegating reduction proofs to the SMT solver, which is infeasible when dimension sizes are unbounded. When proving equivalence of expressions involving reductions, TensorRight attempts to rewrite the LHS and RHS expressions into the same syntactic form. The strategy is to move binary multiplications to the inside of sum reductions and binary additions to the outside of sum reductions. We established a few lemmas for this purpose, shown in Figure 5.7.

Checks delegated to the SMT solver During symbolic execution, TensorRight collects assertions which are then sent to the SMT as a verification condition. These assertions check that accesses fall within dimension sizes; that the dimensions sizes for

LHS and RHS are the same; and that the values stored in tensor expressions are the same. The following are types of assertions emitted by TensorRight illustrated with simplified examples.

- Assertions related to dimension sizes: e.g. `(= (dim-size lhs dim) (+ end (- start))` and `(= (dim-size rhs dim) (+ end (- start))))`; solver can now reason about the relationship between dimension sizes of operations in LHS and RHS.

- Assertions related to access ranges: These assertions are needed to establish that LHS and RHS should fail or succeed at the same time (e.g. `(&& (<= 0 i) (< i (dims rhs init-dims)))`). Also, for complex operations such as `pad` only certain expressions are valid for certain ranges, and by qualifying these ranges, the solver can do case analysis to prove that the final symbolic expressions are equivalent for all ranges.

- Assertions related to final tensor expressions: These are assertions on the final symbolic tensor expressions. In general, there can be arithmetic expressions that needs SMT reasoning. A simple example rule for this case is `(= (+ A B) (+ B A))`, where `A` and `B` are symbolic values of two tensors obtained from uninterpreted functions that model these tensors.

- Assumptions for rules with preconditions: Consider a rule with two input tensors (A and B) and the precondition stating that they should be of the same size. This condition is asserted by the rewrite rule writer.

If both the syntactic checks and the SMT based checks succeed, we deem that the rule is verified.

5.5.3 Verifying expressions with reduction operators

As mentioned in 5.5.1, TensorRight verifies expressions with reduction operators by normalizing lemmas described in Fig. 5.7. Here we describe how these lemmas are verified with a proof approach tailored to reductions. The proof approach described here is also useful for proving expressions where our normalization did not succeed to transform the LHS and RHS into the same syntactic form.

Automatically verifying expressions with reduction operators is challenging for the following reasons.

- The sizes of reduced dimensions are unbounded.
- A reduction of a tensor may be performed in several steps, e.g., due to blocking of the tensor into smaller tiles or distributing reduction over concatenation. For instance, we may want to prove `reduce(concat(A,B)) = reduce(concat(reduce(A),reduce(B)))`.
- The equivalence of two expressions may rely on tensor-level distributivity property, as shown in the lemma CanonBinRed2 in Figure 5.7.

The key idea of verifying expressions with reductions is to make the proof specific to values typically produced by reductions. In particular, we assume that reductions produce tensors whose elements are scalar values of the form $v = \sum x_i y_j$ where x_i, y_j are symbolic scalar values originating in the input tensors. Proving the equivalence of two scalar values $v = \sum x_i y_j$ and $v' = \sum x_{i'} y_{j'}$ boils down to showing that for each term $x_i y_j$ from v, there exists i' and j' such that $x_{i'} y_{j'}$ from v' can be proven to be equivalent to $x_i y_j$. We also need to prove this property in the opposite direction. To show that values v and v' sum

CanonNestRed

$$\text{Reduction}(\text{Reduction}(t, \overline{i'}, \oplus), \bar{i}, \oplus)) \Rightarrow \text{Reduction}(t, \bar{i} \cup \overline{i'}, \oplus)$$

CanonBinRed

$$\text{Reduction}(\text{binary}(t, t', +), \bar{i}, +)) \Rightarrow \text{binary}(\text{Reduction}(t, \bar{i}, +), \text{Reduction}(t', \bar{i}, +), +)$$

CanonNoRed

$$\text{Reduction}(t, \emptyset, \oplus)) \Rightarrow t$$

CanonBinRed2

$$\frac{d = \text{D}(t) \qquad d' = \text{D}(t') \qquad d|_{i'} = \emptyset \qquad d'|_{i} = \emptyset \qquad d_o = d \cup d'}{\text{binary}(\text{Reduction}(t, \bar{i}, +), \text{Reduction}(t', \overline{i'}, +), \times) \Rightarrow \text{Reduction}(\text{binary}(\text{broadcast}(t, \text{Id}, d_o), \text{broadcast}(t', \text{Id}, d_o), \times), \bar{i} \cup \overline{i'}, +)}$$

Figure 5.7: Normalization lemmas for reduction operator.

the same set of terms, it remains to show that the set of terms in v and v' do not contain duplicates; i.e., that no term $x_i y_j$ is added to the result more than once. This is guaranteed by construction of our expressions; this property does not hold, for example, in the expression `reduce(X * concat(Y,Y))` which duplicates Y, allowing scalar values y_j to appear in the result multiple times. To prove equivalence of tensors, we prove equivalence of scalar values in each element of the tensors.

5.6 Implementation

We have implemented a verification backend to prove the TensorRight rewrite rules are correct and a C++ code generator to convert the declarative rewrite rules written in TensorRight DSL to imperative C++ code that can be integrated inside the XLA compiler. The C++ code generator is a tool that we built to showcase the versatility of having verified declarative tensor rewrite rules.

5.6.1 Rosette based Verification

We implemented the operator defintions in the solver-aided programming language Rosette to leverage its symbolic execution capabilities. Given a tensor rewrite rule written in Rosette and a dimension map, we use symbolic execution to generate verification conditions according to Section 5.5. Here, we show an example operator definition and a rewrite rule to illustrate our implementation.

Operator Definition Example

We present how we encode the semantics of a transpose operator. Listing 5.1 shows the implementation of the transpose operator where the function `transpose` takes a tensor and returns tensor. This tensor can then be accessed using a well-formed access (`Acc`). Line 6 checks whether the provided access is valid according to the dimensions that were defined. Lines 8-9 asserts that the tensor symbolic value is defined only when the access indices are within bounds. In our implementation, we modeled the permutation maps as two ordered lists (`in-dims` and `out-dims`) that we use to compute the `in-accs` for the input tensor in line 3. Lines 12-19 establishes the relationship between the input and output tensor dimension sizes, where we assert they should remain unchanged. These internal assertions become part of the verification condition for any rule that uses `transpose` operator.

Rewrite Rule Example

We describe the anatomy of a rewrite rule specification in Rosette taking ReorderTransposeOfSlice rule (Figure 5.8) as an example. Listing 5.2 shows the complete implementation with simplifications for brevity. We instantiate the dimension maps with

symbolic dimension sizes and symbolically execute the tensors under symbolic indices. `(setup-tensor ..)` in Line 4 sets the dimension map keys to `init-dims` and instantiates the sizes to be symbolic variables. Arguments to the `transpose` operator signifies that `init-dims` is replaced by `trans-dims` (Line 13). We implement the permutation map as a list permutation as discussed during `transpose` operator implementation. The start and end indices to the `slice` operators are represented as `Acc` for ease of implementation (Lines 6-10). As shown in the semantics, we use the start and end indices to calculate both the dimension

ReorderTransposeOfSlice :

let $m'_s = m_s \circ m^{-1}$ in

let $m'_l = m_l \circ m^{-1}$ in

let $m'_p = m_p \circ m^{-1}$ in

$\mathsf{slice}(\mathsf{transpose}(t, m), m_s, m_l, m_p) \Longrightarrow \mathsf{transpose}(\mathsf{slice}(t, m'_s, m'_l, m'_p), m)$

Figure 5.8: Reorder transpose and slice.

map sizes and access index values. Finally, we create a general access with symbolic index i (Line 20) and evaluate both LHS and RHS tensor expressions. The implementation checks for composition of the dimensions of the output tensors, while creating verification conditions for the dimension sizes. Then, we use Rosette's `verify` construct to feed the generated verification conditions regarding both the dimension sizes and tensor values to the SMT solver to check for satisfiability. If the SMT solver, returns that there are no counter examples, we deem that the rewrite rule is correct. In this case, indeed the SMT solver returns `(unsat)` proving that the ReorderTransposeOfSlice rule is correct.

Listing 5.1: Transpose Operator Implementation in Rosette

```
1 (define (transpose in-tensor in-dims out-dims)
2  (define (out-transpose accs)
3   (let ([in-accs (..# code to compute accs for input)])
4    (assert (is-access-valid? out-transpose accs))
5    (assert (full-range out-transpose accs))
6    (assert (full-range in-tensor in-accs))
7    (in-tensor in-accs)))
8   (for/list ([in-dim in-dims])
9     (let ([out-dim (..# find corresponding dim)])
10      (assert (equal? (dims out-transpose out-dim)
11                            (dims in-tensor in-dim)))))
12  out-transpose)
```

Generated Verification Conditions

We present few examples of assertions emitted during symbolic execution that becomes part of the verification conditions.

- Assertions related to dimension sizes: e.g. `(= (dims slice_rhs init-dims) (+ end (- start))` and `(= (dims slice_lhs) (+ end (- start))))`
- Assertions related to accesses with in bound: e.g. `(&& (<= 0 i) (< i (dims transpose_rhs init-dims)))`.
- Assertions related to final tensor expressions: In the example, Rosette can itself deduce that the tensor expressions are the same syntactically without going to the solver.

Listing 5.2: Specification of the rule ReorderTransposeOfSlice

```
1 (define-tensor A)
2 (define init-dims (ADim "Root" (list)))
3 (setup-tensor A (list init-dims))
4 (define trans-dims (ADim "Trans" (list)))
5 (define-symbolic start end str integer?)
6 (define start-lhs (list (Acc trans-dims start)))
7 (define end-lhs (list (Acc trans-dims end)))
8 (define str-lhs (list (Acc trans-dims str)))
9 (define start-rhs (list (Acc init-dims start)))
10 # similar definition for end-rhs, str-rhs
11 (define rhs-expr
12   (slice (transpose A (list init-dims) (list trans-dims))
13          start-lhs end-lhs str-lhs))
14 (define lhs-expr
15    (transpose (slice A start-rhs end-rhs str-rhs)
16               (list init-dims) (list trans-dims)))
17 (define-symbolic i integer?)
18 (define access (list (Acc trans-dims i)))
19 (define lhs-value (rhs-expr access))
20 (define rhs-value (lhs-expr access))
21 (verify (assert (equal? lhs-value rhs-value)))
```

5.7 Evaluation

This section evaluates the TensorRight verification framework on the following aspects:

- **Q1:** How expressive is TensorRight compared to a production rewrite engine and existing verified rewrite engines? (Section 5.7.1)
- **Q2:** Is TensorRight a useful tool for compiler developers? (Section 5.7.2)
- **Q3:** Can TensorRight be used to generalize complicated rules? (Section 5.7.3)

To answer these questions, we select a set of existing rules in the XLA compiler to study. We consider XLA's algebraic simplifier (AS) pass for our evaluation. The AS rewrite pass replaces the expressions of certain form with their equivalent expressions. The decision of what rewrite rules to add depends on number of factors. Certain rewrite rules are present to speedup execution. Other rules allow further optimizations such as fusion in other passes of the compiler. Hence, we evaluate TensorRight on all the rules from XLA's target-independent AS pass. We filter out rules that are rarely or never applied by considering only rules that are fired when compiling programs from XLA benchmark suite and MLPerf workloads. In total, there are 123 such rules, which we study in details.

5.7.1 Expressiveness

We show that TensorRight can support a wide variety of rewrite rules present in the XLA compiler. Next, we compare expressivity of TensorRight with popular tensor rewrite tools, TASO [235] and PET [469], on both representation and proof capabilities. We compare both on the implemented rules as well as rules that can be easily supported in these systems just requiring engineering effort.

Coverage. Table 5.1 categorizes the 123 XLA rules into five disjoint classes and summarizes how many rules TensorRight supports. Among 123 rules, 50 rules are arithmetic operations lifted to element-wise tensor expressions. We categorize these operations into simple (39 rules) and advanced (11 rules) classes. The simple element-wise class consists of operations that easily expressible in SMT such as add, subtract, multiplication, division, max, min and modulo. The advanced class either requires additional theory such as IEEE floating point or include operations that cannot be easily modeled via SMT, such as sqrt and log. TensorRight can easily verify 9 simple element-wise rules in less than a second. TensorRight can also support the rest of the rules in the simple element-wise class but they have not been implemented and verified at the time of the submission. In the advanced element-wise class, TensorRight has capability to support a few by unrolling an advanced operation to simple operations. More can be supported but require more development effort and may increase the verification time significantly.

On more interesting rules that involve non-element-wise operations, we categorize them into three classes: requiring operations on dimensions, requiring physical dimension order reasoning, and basic (the rest). 22 rules requires expressing operations on tensor dimensions. For example, the rule in Eq. 5.1 requires a computing d'' to use in the RHS. TensorRight is capable of expressing and proving all of the rules in this category, in which we have implemented and verified 8 of them so far. The rules that we can already support in this category are very important; as shown in Table 5.3, the first two rules, which get fired very often, requires operations on dimensions. The next category contains 40 basic non-element-wise rules. These rules involve similar operations as in the previous category,

but the rules are simpler because they do not require expressing operations on dimensions. TensorRight is also capable of expressing and proving all the rules in this category, in which we have verified 10 of them. The last category, consisting of the remaining 11 rules, requires reasoning about the physical order of dimensions, which is not supported by any of the prior systems and ours. In total, TensorRight is capable of expressing and proving 104 out of 123 rules (85%).

Limitations. Next, we investigate the rules that we have not yet implemented, some of which can be easily supported, but some cannot because of TensorRight́s limitations. Table 5.2 summarizes the different reasons. Note that a rule may fall into multiple reasons as they are not completely disjointed. Among the rules we study, we have not yet supported 16 of them because we have not implemented the semantics of the involving operations (e.g., convert, select, iota, gather, and scatter); and 4 of them because they require additional canonocalization lemmas for reduction operations. We cannot support some of the rules due to more fundamental limitations. TensorRight cannot model 7 rules in SMT via Rosette using boolean, integer, and real number theory. For example, Rosette does not support power, exp, log, and sqrt operations. We also cannot reason about the precision of floating point expressions. For example, $\text{binary}(\text{binary}(\texttt{A},\texttt{const},*),\text{binary}(\texttt{B},\texttt{const},*),+) \Rightarrow \text{binary}(\text{binary}(\texttt{A, B},+),\texttt{const},*)$ is only valid for floating point operands when `const` is $\frac{1}{2}^k$ where $k \in \mathbb{Z}$. To prove this rule, we will have to use SMT's IEEE float point theory, which is not currently supported by Rosette.

Besides, TensorRight has not yet supported reasoning about the physical order of tensor dimensions, i.e., the layout of the tensor in physical memory. Hence, rewrite rules

effecting physical ordering changes, e.g., using an XLA `copy` operation that changes the physical ordering of the tensor data or an XLA `bitcast` that requires reasoning that the logical and physical ordering of the tensor data is unchanged, are not handled by TensorRight .

Comparison to prior work. We compare TensorRight with existing systems that can verify tensor graph rewrites. The existing systems use two different proof mechanisms. One is axiomatic proof system, which requires axioms (or called properties or lemmas) to prove equivalence of tensor expressions. The well-known tensor rewrite engine that uses this mechanism is TASO. Another approach is using statistical proof mechanism, which purely relies on testing. This approach is used by PET. Table 5.1 summarizes the number of XLA rules that TASO and PET can support (and have already implemented). We can see that TensorRight can support significantly more rules than TASO and PET.

Even for the rules that TASO and PET can support, both systems cannot prove the correctness of those rules for fully unbounded tensors. While TASO supports verifying rules involving tensors of arbitrary sizes, its axioms are verified only for tensors of up to 4 elements in each dimension. Additionally, TASO cannot verify rules with arbitrary ranked tensors for some operands (e.g., convolution kernel); TASO specializes convolution into `conv1d`, `conv2d`, etc. While PET can verify rules for arbitrary sizes and ranks, some dimension sizes (e.g., convolution kernel's) dictates the number of regular boxes, in which the statistical testing must be performed. As a result, it cannot not verify that a rule is correct for some unbounded dimensions without enumerating the test on all possible sizes. The ability to handle unbounded tensors is crucial in a production compiler; most of the 123 rules we study are implemented in XLA for unbounded tensors.

Next, we discuss other limitations of axiomatic and statistical proof techniques beyond the unboundedness aspect. Table 5.2 breaks down the number of rules and the reasons different approaches cannot support them. The majority of XLA operations — including complex ones such as dot and slice — are not implemented in TASO and PET, so most of the 123 rules are not yet supported. The convolution operation supported in TASO and PET is much simpler that XLA's as it does not support interior padding and allows only a few sizes for exterior padding. Additionally, an axiomatic approach like TASO requires new axioms to support new rules; among the XLA rules we study, 113 rules need new axioms in TASO. Despite the developmental effort to implement new operations (and axioms), both TASO and PET do not support rules that require reasoning about the physical order of dimensions, similar to TensorRight. This is because all the systems only check the correctness of the logical values so far. Note that while TASO's and PET's transpose and reshape operations change physical dimension orders and/or sizes, it is unclear how they will handle a rule such as reshape(A) $\Rightarrow$ bitcast(A) that requires a certain constraint on the physical dimension order of A.

Unlike TensorRight TASO and PET cannot express rules that require operations on dimensions and preconditions. This precludes them from supporting 22 and 52 XLA rules, respectively. To support these rules, the systems need an expressive language to manipulate operations on dimensions in a rewrite rule, and specify conditions of an input tensor regarding its values and shape as well as rule's parameters. Once these features are expressible, a statistical approach like PET should be able to prove the rule's correctness by generating test

inputs according to the precondition. However, it is cumbersome for an axiomatic approach like TASO as it requires adding more axioms with preconditions.

The statistical approach has additional unique benefits and drawbacks. The major advantage is that it does not need to generate verification conditions to be proven by a theorem prover such as an SMT solver. This process is complex and difficult to get right. Furthermore, the capability of the system is not limited by the capability of the solver. For example, the approach can potentially handle the operations that are not easily modelled in a theorem prover, such as log and sqrt. However, a statistical approach has a major limitation. In particular, PET's approach can verify an equivalence of expressions only if they are linear. This restriction is in fact more limited than operations supported by a theorem prover like an SMT solver. We can see that PET cannot verify 38 XLA rules because of the linearity restriction; whereas, TensorRight and TASO cannot verify 7 rules due to the translation to SMT formulas.

5.7.2 TensorRight Deployment

This section evaluates the importance of the rewrite rules that TensorRight can verify for XLA.

Rewrite rules used in XLA benchmark suite Our benchmarks comprise of 114 ML models (both inference and training) from the XLA regression suite for Google's Tensor Processing Units (TPUs). These benchmarks contain computation graphs with 100–56,000 tensor operations. Table 5.3 highlights a few of TensorRight verified-rules that are

Category (disjoint)	Number of Supported Rules			
	XLA	TensorRight	TASO	PET
Elementwise: simple	39	39 (9)	33 (5)	14 (10)
Elementwise: advanced	11	3 (0)	3 (0)	0 (0)
Non-elementwise: basic	40	40 (10)	16 (3)	16 (3)
Non-elementwise: operations on dimensions	22	22 (8)	0 (0)	0 (0)
Non-elementwise: physical dimension order	11	0 (0)	0 (0)	0 (0)
Total	123	104 (27)	52 (8)	30 (13)

Table 5.1: Number of supported rules per disjoint category. The numbers in parentheses indicate rules that have been implemented and verified (and we continue to implement more); some of the not-yet-implemented rules require defining new operations and/or new lemmas.

Reason for the lack of support	Number of Unsupported Rules		
	TensorRight	TASO	PET
Unsupported ops	16	104	104
Need lemmas	4	113	0
Cannot model in SMT	7	7	0
Physical dimension order	12	12	12
Operations on dimensions	0	22	22
Precondition	0	52	52
Non-linear	0	0	38

Table 5.2: Number of unsupported rules per each reason. There may be overlaps between reasons. The first reason is not a fundamental limitation because new operations can be added.

exercised heavily for these 114 XLA benchmarks. These rules include sophisticated rewrites on `Dot`, e.g., (a) rewrite a `Dot` as a product of the `Broadcast`-ed `Transpose`-ed inputs, and

	Rule	Occurs
1.	Rewrite a `Dot` as a product of the broadcasted transposed variants of the inputs	313
2.	Rewrite a `Dot` as a reduction on the product of broadcasted and transposed inputs.	5317
3.	Reverse the ordering of a `slice` and a `broadcast`	335

Table 5.3: Table shows the number of instances a TensorRightverified rule was instantiated for 114 XLA regression benchmarks compiled for multiple generations of TPUs.

	Rule	Occurs
1.	Reverse the ordering of a `dynamic-slice` and `transpose`	7626
2.	Replace a `broadcast` with a `transpose` op	6945
3.	Remove a non-permuting `transpose` operation	28218

Table 5.4: Table shows the number of instances a TensorRightgenerated-C++ rule was instantiated for the `MLPerf` suite of benchmarks compiled for multiple generations of TPUs.

(b) rewrite a `Dot` as a reduction on the product of the `Broadcast`-ed `Transpose`-ed inputs. Such sophisticated rewrites occur thousands of times and hence, are deemed critical to be verified, which is able to be done by TensorRight .

TensorRight generated `C++` code in `MLPerf`. In **??**, we present the details on the code generator that translates rules written in TensorRight DSL to C++ implementation, which can be integrated into XLA as part of the algebraic simplification pass. Table 5.4 highlights three rewrite rules that are integrated with XLA and applied extensively when compiling `MLPerf` benchmark suite, a benchmark suite consisting of benchmarks jointly developed by industry and academia [401].

5.7.3 Generalizing XLA Rewrite Rules

Using TensorRight we found that some XLA rewrite rules are overly constrained. This is done for practical reasons, where the compiler engineers intentionally avoid reasoning in cases where a spurious bug may be introduced. We use TensorRight to generalize the following rule by relaxing its precondition.

FoldConvInputPad Rule Fig. 5.9 (top) presents the rule as it exists in XLA using the TensorRight DSL notation. The goal of the rule is to fold the pad operation into the operand arguments of the convolution itself (XLA convolutions support padding as operands). The XLA rule[2] does not support internal padding in the input tensor and gives up if this constraint is violated. This is largely because verifying the correctness of internal padding is a challenging task, and writing code to convert internal padding into dilation is tedious. TensorRight can generalize this rule by removing this constraint as shown in Figure 5.9 (bottom). The differences are put in boxes. The key to generalizing the rule is to calculate the padding arguments that get fed into the convolution operator. This is a function of the `pad` operators interior, high and low padding as well as padding that may already exist in the `convolution` operator. Figure 5.9 (bottom) shows how to calculate padding maps m_{ol}, m_{oh}, m_{oi} for the rule to be general. These maps are more complicated than the non-general version, but this allows the compiler writer to get rid of the precondition of the rule shown in Figure 5.9 (bottom). It is not clear immediately why this formulation might be correct. Therefore, we encode it in our Rosette implementation and successfully prove that the generalized rule with these calculations are sound.

[2]`https://github.com/tensorflow/tensorflow/blob/116adfcc644ed297fe2cfd7c0756e6470dd6a2ba/tensorflow/compiler/xla/service/algebraic_simplifier.cc#L6653`

5.8 Related Work

Tensor Graph Rewrite Optimizations

There are multiple existing systems [236, 235, 469, 485, 153, 507] that optimize tensor computation graphs by applying rewrite rules. Apart from applying only existing rewrite rules, TASO [235] and PET [469] can synthesize new rules and provide mechanisms to prove their correctness. TASO checks the correctness in two phases: test then verify. First, TASO tests the LHS and RHS expressions on random inputs with a small tensor shape. If the LHS and RHS expressions have the same output, TASO then verifies the rule via an SMT solver by providing axioms, which capture mathematical properties of operators. In contrast, PET verifies the correctness of a rule via a statistical approach. PET symbolically infers the bounding boxes of the output tensor, where each box contains elements that can be represented by the same linear expression of its input elements. Leveraging the linear property, PET statistically verifies the equivalence of the corresponding boxes of the LHS and RHS

FoldConvInputPad(XLA) :

let $m_{ol} = m_l + m_{lp}$ in

let $m_{oh} = m_h + m_{hp}$ in

convolution(pad$(t, 0, m_{lp}, m_{hp}, m_{ip}), t'$,
$(i_b, i_{if}, i_{of},$
$m_l, m_h, m_i, m'_i)$

$\Longrightarrow_{m_{ip}=0 \land m_i=1}$

convolution$(t, t', i_b, i_{if}, i_{of},$
$m_{ol}, m_{oh}, m_{oi}, m'_i)$

FoldConvInputPad(Generalized) :

let $m_{ol} = m_l + (m_i + 1) \times m_{lp}$ in

let $m_{oh} = m_h + (m_i + 1) \times m_{hp}$ in

let $m_{oi} = m_i + m_{pi} + (m_i \times m_{pi})$ in

convolution(pad$(t, 0, m_{lp}, m_{hp}, m_{ip}), t'$,
$(i_b, i_{if}, i_{of},$
$m_l, m_h, m_i, m'_i)$

$\Longrightarrow$

convolution$(t, t', i_{ib}, i_{ob}, i_{if}, i_{of},$
$m_{ol}, m_{oh}, m_{oi}, m'_i)$

Figure 5.9: Fold input pad into convolution.

outputs by checking $m+1$ specific positions in the box, where m is the number of dimensions of the output tensor. As discussed in § 5.7, both an axiomatic approach like TASO and a statistical approach like PET cannot prove many algebraic simplification rules present in XLA.

Many other tensor and stencil compilers [442, 206, 174, 434] not only use rewrite rules for optimizations but also generate low-level code, where a rule can map from high-level constructs to low-level ones. However, these compilers assume the correctness of these rewrite rules and do not attempt to verify them. TensorRight could be useful for providing correctness guarantees in these systems.

Verified Tensor Program Optimizations. A handful of exiting works support an end-to-end correctness verification of generated code from high-level tensor programs. ATL [308] guarantees the correctness of the generated code by applying only manually-verified rewrite rules via Coq theorem prover. Unlike TensorRight that aims to automate the correctness checking of rewrite rules, ATL aims to verify the correctness of the final generated code. Similar to ATL, Halide translation validation [117] verifies the equivalence of the low-level generated code and the high-level specification. It formalizes both the high-level and the low-level program representations, and associates the high-level constructs to low-level code. This technique can be used to validate the correctness of the compiled code for a given program. Comparatively, TensorRight verifies the correctness of the optimization rules for any input sizes. Unlike the above two systems, Halide term rewriting synthesis [359] presents an automatic verification tool for soundness and termination of Halide's rewriting system, rather than providing the end-to-end program correctness validation. The term rewriting

synthesis generates and verifies rules for expressions on tensor indices' bounds, so the verified expressions are on scalar values.

Verified Optimizations in General-Purpose Compilers. There are many related work on verified compilers for general-purpose programs. The most relevant work in this area is Alive [317], an automatic verification tool for LLVM's peephole optimization rewrite rules. AliveInLean [284] implements Alive using Lean. Alive2 [316], employs translation validation to prove correctness of generated code from given programs. It defines and introduces semantics for new previously not formalized constructs in LLVM and is able to catch subtle bugs. However, it does not verify that the LLVM transformations are correct. Further, most of these systems work on scalar or vector code and does not formalize computations on higher dimensional tensors that require reasoning about dimensions.

Formal languages and semantics of Tensor IRs. There are several works on formalizing Tensor IRs. Lean-MLIR [365] provides a formalism of MLIR [283] that has been used for defining different tensor dialects. Axon [119] gives a formal language to represent operations on dimensions. The language is expressive, but unlike TensorRight it does not provide semantics.

Chapter 6

Conclusion and Future Work

This thesis explores the application of program synthesis to construct reliable and efficient distributed systems within the rapidly expanding technological landscape. Specifically, our research delves into synthesis of coordination in replicated systems in both message-passing and RDMA network adaptors. Additionally, we employ program synthesis to automatically generate computations in the domain of graph analytics, aiming to aid programmers in the implementation and optimization of complex graph analytics problems. Lastly, our work examines the use of formal verification in the field of tensor graph optimization, highlighting the benefits that a verified tensor language can offer to compiler programmers to specify and develop new rewrite rules.

In particular, Chapter two introduces Hamsaz, a framework for replication coordination analysis and synthesis. We define the notion of well-coordination, a sufficient condition for integrity and convergence of the replicated data types. Hamsaz uses off-the-shelf SMT solvers to determine pairs of conflicting and dependent methods in an object specification.

The result of this analysis is then used to instantiate protocols for generating replicated objects with minimum synchronization. We evaluated Hamsaz on various use-cases such as bank accounts, auctions, payroll, and tournaments, and the results of the experiments indicated that the synthesized replicated objects were much more responsive than the strongly consistent baseline.

In chapter three, We saw well-coordinated replicated data types (WRDTs) for the RDMA network model. We saw operational semantics for both abstract WRDTs and concrete RDMA WRDTs. The abstract semantics captures the well-coordination conditions and serves as a specification for the concrete semantics. The concrete semantics of RDMA WRDTs divides methods into three categories based on their conflict, dependency, and summarization properties, and captures their coordination requirements based on one-sided communication. It is formally proved that concrete semantics refines the abstract semantics and preserves convergence and integrity. We saw the protocols that efficiently implement the semantics, and the empirical evaluation that shows their high throughput.

Chapter four presents GraFS, declarative graph analytics language and synthesizer. The GraFS language provides a high-level declarative specification for graph analytics, along with a set of semantics-preserving fusion transformations to optimize the specification. The fusion rules reduce the specification to three primitives of graph analytics, namely, reduction over paths, mapping over vertices, and reduction over vertices. GraFS formally presents the correctness and termination conditions of graph analytics iterative models, which allows for the synthesis of kernel functions based on these conditions. The experimental results demonstrate that the synthesized code is on par with or outperforms handwritten code,

and that fusion accelerates execution. Overall, the GraFS language and synthesizer are powerful tools that can greatly accelerate the development and deployment of graph analytics applications, ultimately benefiting a wide range of fields and applications.

In chapter five we presented TensorRight. A verified tensor rewrite system for specification and verification of the XLA rewrite rules. We formally define the specification language which closely models the XLA's tensor operators. The denotational semantics of TensorRight is then used to generate the verification conditions required to prove the correctness of the rewrites. TensorRight can help identify errors and potential vulnerabilities early in development, leading to higher-quality software.

The XLA compiler, as a crucial element of the TensorFlow machine learning framework, requires continuous improvement to meet the increasing demands of the field. One area of potential future work is the synthesis of new rewrite rules for the compiler. To accomplish this, program synthesis can play an important role in the development of the XLA compiler. Writing preconditions for a rule can be a challenging and time-consuming task, as it requires the developer to consider all possible scenarios that could arise during execution. However, synthesis can help alleviate this burden by automatically generating preconditions that satisfy the desired behavior. This approach can significantly reduce the development time of new rewrite rules while ensuring their correctness and effectiveness. By program synthesis, the development of the XLA compiler can continue to progress, leading to more efficient and effective machine learning systems.

Bibliography

[1] The coq proof assistant. https://coq.inria.fr/.

[2] Elc: Spacex lessons learned. https://lwn.net/Articles/540368/.

[3] Infiniband userspace verbs access. https://www.kernel.org/doc/html/latest/infiniband/user_verbs.html.

[4] LinkedIn's Voldemort. http://www.project-voldemort.com/.

[5] Mellanox technologies. rdma aware networks programming user manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[6] Memcached. http://memcached.org/.

[7] Pvs specification and verification system. http://pvs.csl.sri.com/.

[8] grammer-v4. https://github.com/antlr/grammars-v4, 2017.

[9] Daniel Abadi. Consistency tradeoffs in modern distributed database system design. *Computer*, 45(2), 2012.

[10] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[11] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 59–74, New York, NY, USA, 2005. ACM.

[12] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):20, 2017.

[13] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002.

[14] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[15] Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 51–60, 2018.

[16] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of rdma on agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 409–418, 2019.

[17] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 599–616, 2020.

[18] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995.

[19] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Protocol-aware recovery for consensus-based distributed storage. *ACM Transactions on Storage (TOS)*, 14(3):1–30, 2018.

[20] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language cat. *arXiv preprint arXiv:1608.07531*, 2016.

[21] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[22] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2013.

[23] Peter Alvaro, Tyson Condie, Neil Conway, Joseph M. Hellerstein, and Russell Sears. I do declare: Consensus in a logic language. *SIGOPS Oper. Syst. Rev.*, 43(4):25–30, January 2010.

[24] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination analysis and placement for distributed programs. *ACM Trans. Database Syst.*, 42(4):23:1–23:31, October 2017.

[25] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: A calm and collected approach. In *In Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260, 2011.

[26] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 731–744, 2019.

[27] Zachary R. Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 98–109, 2009.

[28] Apache. Apache giraph. `https://giraph.apache.org/`, 2020.

[29] Michael Armbrust, Kristal Curtis, Tim Kraska, Armando Fox, Michael J Franklin, and David A Patterson. Piql: Success-tolerant query processing in the cloud. *Proceedings of the VLDB Endowment*, 5(3):181–192, 2011.

[30] Joe Armstrong. The development of erlang. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 196–203, New York, NY, USA, 1997. ACM.

[31] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and Padmanabhan Pillai. Meld: A declarative approach to programming ensembles. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2794–2800, Oct 2007.

[32] Michael P. Ashley-Rollman, Peter Lee, Seth Copen Goldstein, Padmanabhan Pillai, and Jason D. Campbell. A language for large ensembles of independently executing nodes. In *Proceedings of the 25th International Conference on Logic Programming*, ICLP '09, pages 265–280, Berlin, Heidelberg, 2009. Springer-Verlag.

[33] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2), 1994.

[34] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quema, and Marko Vukolić. The next 700 bft protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, January 2015.

[35] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014.

[36] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1327–1342. ACM, 2015.

[37] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1327–1342, 2015.

[38] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 22. ACM, 2012.

[39] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[40] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proc. SIGMOD*, 2013.

[41] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[42] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[43] Alex Bain, John Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. A domain-specific language for computing on encrypted data (invited talk). In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

[44] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguica. Ipa: Invariant-preserving applications for weakly consistent replicated databases. *Proceedings of the VLDB Endowment*, 12(4).

[45] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguica, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, New York, NY, USA, 2015. ACM.

[46] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguica, Mahsa Najafzadeh, and Marc Shapiro. Towards fast invariant preservation in geo-replicated systems. *SIGOPS Oper. Syst. Rev.*, 49(1):121–125, January 2015.

[47] Valter Balegas, Nuno Preguica, Sérgio Duarte, Carla Ferreira, and Rodrigo Rodrigues. Ipa: invariant-preserving applications for weakly-consistent replicated databases. *arXiv preprint arXiv:1802.08474*, 2018.

[48] Kshitij Bansal, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. A new decision procedure for finite sets and cardinality constraints in smt. In *International Joint Conference on Automated Reasoning*, pages 82–98. Springer, 2016.

[49] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *ACM Sigplan Notices*, volume 41, pages 394–403. ACM, 2006.

[50] Daniel Barbará-Millá and Hector Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, 1994.

[51] Shaon Barman, Rastislav Bodik, Satish Chandra, Emina Torlak, Arka Bhattacharya, and David Culler. Toward tool support for interactive synthesis. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 121–136. ACM, 2015.

[52] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[53] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[54] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[55] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. From relational verification to simd loop synthesis. In *ACM SIGPLAN Notices*, volume 48, pages 123–134. ACM, 2013.

[56] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. Checking robustness between weak transactional consistency models. *Programming Languages and Systems*, 12648:87, 2021.

[57] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *Proc. NSDI*, 2006.

[58] John Bender, Mohsen Lesani, and Jens Palsberg. Declarative fence insertion. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 367–385, New York, NY, USA, 2015. ACM.

[59] Giovanni Bernardi and Alexey Gotsman. Robustness against consistency models with atomic visibility. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 59. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[60] A. Bessani and P. Sousa. Smart — high-performance byzantine-fault-tolerant state machine replication. `http://code.google.com/p/bft-smart/`, 2009.

[61] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 169–180, Berkeley, CA, USA, 2013. USENIX Association.

[62] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. Depspace: A byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 163–176, New York, NY, USA, 2008. ACM.

[63] Mark Bickford. Component specification using event classes. In *Component-Based Software Engineering*, volume 5582 of *Lecture Notes in Computer Science*. 2009.

[64] M. Biely, P. Delgado, Z. Milosevic, and A. Schiper. Distal: A framework for implementing fault-tolerant distributed algorithms. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–8, June 2013.

[65] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. In *Proc. SOSP*, 1985.

[66] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1), 1987.

[67] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proc. POPL*, 2006.

[68] Ranadeep Biswas, Michael Emmi, and Constantin Enea. On the complexity of checking consistency for replicated data types. In *International Conference on Computer Aided Verification*, pages 324–343. Springer, 2019.

[69] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. Monkeydb: effectively testing correctness under weak isolation levels. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, 2021.

[70] S. Biswas, J. Huang, Sengupta A., , and Bond M. D. Doublechecker: Efficient sound and precise atomicity checking. In *Proc. PLDI*, 2014.

[71] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proc. OOPSLA*, 2006.

[72] Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Ausserlechner, and Raphael Spörk. Synthesis of synchronization using uninterpreted

functions. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 11:35–11:42, Austin, TX, 2014. FMCAD Inc.

[73] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proc. ICSE*, 2011.

[74] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 141–154, Berkeley, CA, USA, 2011. USENIX Association.

[75] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. Automated conflict-free distributed implementation of component-based models. In *International Symposium on Industrial Embedded System (SIES)*, pages 108–117. IEEE, 2010.

[76] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[77] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *ACM SIGPLAN Notices*, volume 51, pages 775–788. ACM, 2016.

[78] A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In *Proc. POPL*, 2014.

[79] Ahmed Bouajjani, Constantin Enea, Madhavan Mukund, Gautham Shenoy, and SP Suresh. Formalizing and checking multilevel consistency. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 379–400. Springer, 2020.

[80] Eric Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[81] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

[82] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, September 1973.

[83] Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, August 2016.

[84] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. Serializability for eventual consistency: Criterion, analysis, and applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 458–472, New York, NY, USA, 2017. ACM.

[85] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. Serializability for eventual consistency: criterion, analysis, and applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 458–472, 2017.

[86] Sebastian Burckhardt. *Principles of Eventual Consistency.* Foundations and Trends in Programming Languages. 2014.

[87] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 691–707, 2010.

[88] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming*, pages 283–307. Springer, 2012.

[89] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *Proc. POPL*, 2014.

[90] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, 2008.

[91] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[92] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

[93] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proc. CCS*, 2006.

[94] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam Mckelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Marvin Mcnett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *In SOSP '11*, pages 143–157, 2011.

[95] Domenico Cantone, Eugenio Omodeo, and Alberto Policriti. *Set theory for computing: from decision procedures to declarative programming with sets.* Springer Science & Business Media, 2013.

[96] Domenico Cantone and Calogero G Zarba. A new fast tableau-based decision procedure for an unquantified fragment of set theory. In *Automated Deduction in Classical and Non-Classical Logics*, pages 126–136. Springer, 2000.

[97] Nuno Carvalho and et. al. Appia framework. `http://appia.di.fc.ul.pt/wiki/index.php?title=Main_Page`, 2011. Accessed: 2018-06-23.

[98] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[100] Pavol Cerny, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. From non-preemptive to preemptive scheduling using synchronization synthesis. *Form. Methods Syst. Des.*, 50(2-3):97–139, June 2017.

[101] Pavol Cerny, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 951–967, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[102] Pavol Cerny, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Regression-free synthesis for concurrency. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 568–584, Cham, 2014. Springer International Publishing.

[103] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 42. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[104] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[105] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[106] Feng Chen and Grigore Rosu. Parametric and sliced causality. In *Proc. CAV*, 2007.

[107] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jpredictor: A predictive runtime analysis tool for java. In *Proc. ICSE*, 2008.

[108] Qichang Chen, Liqiang Wang, Zijiang Yang, and ScottD. Stoller. Have: Detecting atomicity violations via integrated dynamic and static analysis. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering*, volume 5503 of *LNCS*, pages 425–439. Springer Berlin Heidelberg, 2009.

[109] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 579–594, USA, 2018. USENIX Association.

[110] Unnikrishnan Cheramangalath, Rupesh Nasre, and Y N. Srikant. Dh-falcon: A language for large-scale graph processing on distributed heterogeneous systems. pages 439–450, 09 2017.

[111] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. *ACM SIGPLAN Notices*, 43(6):304–315, 2008.

[112] Wei-Ngan Chin. Safe fusion of functional expressions. In *ACM SIGPLAN Lisp Pointers*, number 1, pages 11–20. ACM, 1992.

[113] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 234–245, New York, NY, USA, 2011. ACM.

[114] Kevin Clancy and Heather Miller. Monotonicity types for distributed dataflow. In *Proceedings of the Programming Models and Languages for Distributed Computing*, page 2. ACM, 2017.

[115] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[116] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[117] Basile Clément and Albert Cohen. End-to-end translation validation for the halide language. 6(OOPSLA1), apr 2022.

[118] Onofre Coll Ruiz, Kiminori Matsuzaki, and Shigeyuki Sato. s6raph: vertex-centric graph processing framework with functional interface. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, pages 58–64, 2016.

[119] Alexander Collins and Vinod Grover. Axon: A Language for Dynamic Shapes in Deep Learning Graphs. *arXiv e-prints*, page arXiv:2210.02374, October 2022.

[120] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[121] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2), 2008.

[122] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[123] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[124] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[125] Flavio Cruz, Ricardo Rocha, and Seth Copen Goldstein. Declarative coordination of graph-based parallel programs. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2016.

[126] Flavio Cruz, Ricardo Rocha, Seth Copen Goldstein, and Frank Pfenning. A linear logic programming language for concurrent programming over graph structures. *Theory and Practice of Logic Programming*, 14(4-5):493–507, 2014.

[127] Dave Cunningham, Khilan Gudka, and Susan Eisenbach. Keep off the grass: Locking the right path for atomicity. In *International Conference on Compiler Construction*, pages 276–290. Springer, 2008.

[128] Andrei Damian, Cezara Drăgoi, Alexandru Militaru, and Josef Widder. Communication-closed asynchronous protocols. In *International Conference on Computer Aided Verification*, pages 344–363. Springer, 2019.

[129] Alain Darte. On the complexity of loop fusion. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00425)*, pages 149–157. IEEE, 1999.

[130] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 752–768, New York, NY, USA, 2018. ACM.

[131] Marc A De Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 475–486, 2012.

[132] Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[133] Kevin De Porre, Carla Ferreira, Nuno Preguica, and Elisa Gonzalez Boix. Ecros: building global scale systems from sequential code. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[134] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, 2007.

[135] Yves Deville and Kung-Kiu Lau. Logic program synthesis. *The Journal of Logic Programming*, 19:321–350, 1994.

[136] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[137] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. Ettm: A scalable fault tolerant network manager. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 85–98, Berkeley, CA, USA, 2011. USENIX Association.

[138] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. Petr4: formal foundations for p4 data planes. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–32, 2021.

[139] Cezara Dragoi, Thomas A Henzinger, and Damien Zufferey. PSYNC : A partially synchronous language for fault-tolerant distributed algorithms. *Popl*, pages 1–16, 2016.

[140] Cezara Drăgoi, Thomas A Henzinger, and Damien Zufferey. Psync: a partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices*, 51(1):400–415, 2016.

[141] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.

[142] K. Driscoll, B. Hall, M. Paulitsch, P. Zumsteg, and H. Sivencrona. The real byzantine generals. In *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, volume 2, pages 6.D.4–61–11 Vol.2, Oct 2004.

[143] Kevin Driscoll, Brendan Hall, Hrakan Sivencrona, and Phil Zumsteg. *Byzantine Fault Tolerance, from Theory to Reality*, pages 235–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[144] Kevin R Driscoll. *Murphy Was an Optimist*, pages 481–482. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[145] Bruno Dutertre. Yices 2.2. In *Proc. CAV*, 2014.

[146] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. Diesel: Dsl for linear algebra and neural net computations on gpus. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 42–51, 2018.

[147] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 73–84, Oct 2005.

[148] Michael Emmi and Constantin Enea. Monitoring weak consistency. In *Proc. CAV*, 2018.

[149] Michael Emmi and Constantin Enea. Monitoring weak consistency. In *International Conference on Computer Aided Verification*, pages 487–506. Springer, 2018.

[150] Michael Emmi and Constantin Enea. Weak-consistency specification via visibility relaxation. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[151] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, Akimasa Morihata, and Hideya Iwasaki. Think like a vertex, behave like a function! a functional dsl for vertex-centric big graph processing. *ACM SIGPLAN Notices*, 51:200–213, 09 2016.

[152] Mahdi Eslamimehr and Jens Palsberg. Race directed scheduling of concurrent programs. In *Proc. PPoPP*, 2014.

[153] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. Optimizing DNN Computation Graph Using Graph Substitutions. *Proc. VLDB Endow.*, 13(12):2734–2746, July 2020.

[154] Azadeh Farzan and P. Madhusudan. Causal atomicity. In Thomas Ball and RobertB. Jones, editors, *Computer Aided Verification*, volume 4144 of *LNCS*, pages 315–328. Springer Berlin Heidelberg, 2006.

[155] Azadeh Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Proc. CAV*, 2008.

[156] Azadeh Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *Proc. TACAS*, 2009.

[157] Azadeh Farzan, P. Madhusudan, and Francesco Sorrentino. Meta-analysis for atomicity violations under nested locking. In *Proc. CAV*, 2009.

[158] Grigory Fedyukovich and Rastislav Bodík. Accelerating syntax-guided invariant synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I 24*, pages 251–269. Springer, 2018.

[159] Alan Fekete. Allocating isolation levels to transactions. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 206–215, New York, NY, USA, 2005. ACM.

[160] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[161] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. Component-based synthesis for complex apis. *ACM SIGPLAN Notices*, 52(1):599–612, 2017.

[162] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1), 1988.

[163] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. Technical report, Massachusetts Inst of Tech Cambridge lab for Computer Science, 1982.

[164] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[165] Cormac Flanagan. Verifying commit-atomicity using model-checking. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, volume 2989 of *LNCS*, pages 252–266. Springer Berlin Heidelberg, 2004.

[166] Cormac Flanagan and Stephen N Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. POPL*, 2004.

[167] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, August 2008.

[168] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Proc. PLDI*, 2008.

[169] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proc. PLDI*, 2003.

[170] Mitch Fletcher. Progression of an open architecture: from orion to altair and lss. Technical Report White paper S65- 5000-20-0, Honeywell International, Glendale, 2009.

[171] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *International School on Advanced Functional Programming*, pages 129–158. Springer, 2002.

[172] Eddy Fromentin, Michel Raynal, and Frederic Tronel. On classes of problems in asynchronous distributed systems with process crashes. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No. 99CB37003)*, pages 470–477. IEEE, 1999.

[173] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. In *2nd SysML Conference*, 2019.

[174] Rongxiao Fu, Xueying Qin, Ornela Dardha, and Michel Steuwer. Row-polymorphic types for strategic rewriting. *CoRR*, abs/2103.13390, 2021.

[175] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663, 2014.

[176] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguica. Efficient middleware for byzantine fault tolerant database replication. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 107–122, New York, NY, USA, 2011. ACM.

[177] Adria Gascón and Ashish Tiwari. Synthesis of a simple self-stabilizing system. In *Proc. 3rd Workshop on Synthesis (SYNT)*, 2014.

[178] Vasilis Gavrielatos, Antonios Katsarakis, and Vijay Nagarajan. Odyssey: The impact of modern hardware on strongly-consistent replication protocols. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 245–260, 2021.

[179] Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan, Boris Grot, and Arpit Joshi. Kite: Efficient and available release consistency for the datacenter. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–16, 2020.

[180] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 21–21, Berkeley, CA, USA, 2007. USENIX Association.

[181] David K Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM, 1979.

[182] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), June 2002.

[183] Seth Gilbert and Nancy A. Lynch. Perspectives on the CAP theorem. *IEEE Computer*, 45(2):30–36, 2012.

[184] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[185] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. Abelian: A compiler for graph analytics on distributed, heterogeneous platforms. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 249–264, Cham, 2018. Springer International Publishing.

[186] Jennifer Ann Golbeck. *Computing and applying trust in web-based social networks*. PhD thesis, 2005.

[187] Victor BF Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[188] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[189] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, New York, NY, USA, 2016. ACM.

[190] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. Making pull-based graph processing performant. In *ACM SIGPLAN Notices*, volume 53, pages 246–260. ACM, 2018.

[191] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 595–608, New York, NY, USA, 2015. ACM.

[192] Matthias Güdemann, Gwen Salaün, and Meriem Ouederni. Counterexample guided synthesis of monitors for realizability enforcement. In *Automated Technology for Verification and Analysis: 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings 10*, pages 238–253. Springer, 2012.

[193] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *OSDI*, pages 169–184, 2016.

[194] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 169–184, 2016.

[195] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[196] Sumit Gulwani. Programming by examples. *Dependable Software Systems Engineering*, 45(137):3–15, 2016.

[197] Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[198] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46(6):62–73, 2011.

[199] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, volume 11, pages 62–73, 2011.

[200] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *ACM SIGPLAN Notices*, volume 46, pages 50–61. ACM, 2011.

[201] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[202] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: replication at the speed of multi-core. In *Proc. Eurosys*, 2014.

[203] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Peter Bodik, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure recovery: When the cure is worse than the disease. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 8–8, Berkeley, CA, USA, 2013. USENIX Association.

[204] Vassos Hadzilacos and Sam Toueg. Reliable broadcast and related problems. *Distributed Systems*, 26:97–145, 1993.

[205] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. Fireiron: A scheduling language for high-performance linear algebra on gpus. *arXiv preprint arXiv:2003.06324*, 2020.

[206] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, pages 100–112, 2018.

[207] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, February 2009.

[208] Richard L Halpert, Christopher JF Pickett, and Clark Verbrugge. Component-based lock allocation. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 353–364. IEEE, 2007.

[209] Md. E. Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proc. ASPLOS*, 2015.

[210] John Hatcliff, Robby, and MatthewB. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 175–190. Springer Berlin Heidelberg, 2004.

[211] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[212] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[213] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 165–181, Berkeley, CA, USA, 2014. USENIX Association.

[214] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In *Proc. ESOP*, 1986.

[215] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[216] Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. Counterexample-guided control. In *Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30–July 4, 2003 Proceedings*, pages 886–902. Springer, 2003.

[217] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA*, 1993.

[218] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[219] John A Hewson, Paul Anderson, and Andrew D Gordon. A declarative approach to automated configuration. In *LISA*, volume 12, pages 51–66, 2012.

[220] Loc Hoang, Matteo Pontecorvi, Roshan Dathathri, Gurbinder Gill, Bozhi You, Keshav Pingali, and Vijaya Ramachandran. A round-efficient distributed betweenness centrality algorithm. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 272–286, 2019.

[221] Loc Hoang, Matteo Pontecorvi, Roshan Dathathri, Gurbinder Gill, Bozhi You, Keshav Pingali, and Vijaya Ramachandran. A round-efficient distributed betweenness centrality algorithm. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 272–286, New York, NY, USA, 2019. ACM.

[222] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[223] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 124–133. IEEE Computer Society, 2013.

[224] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. Disciplined inconsistency with consistency types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 279–293, New York, NY, USA, 2016. ACM.

[225] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: a dsl for easy and efficient graph analysis. *ACM SIGARCH Computer Architecture News*, 40(1):349–362, 2012.

[226] Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication coordination analysis and synthesis. In *Proceedings of Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '19, New York, NY, USA, 2019. ACM.

[227] Farzin Houshmand and Mohsen Lesani. Hamsaz: replication coordination analysis and synthesis. *Proceedings of the ACM on Programming Languages*, 3(POPL):74, 2019.

[228] Qinheping Hu and Loris D'Antoni. Syntax-guided synthesis with quantitative syntactic objectives. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30*, pages 386–403. Springer, 2018.

[229] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[230] Phillip W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *ICDCS*, 1990.

[231] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. A simple inductive synthesis methodology and its applications. In *ACM Sigplan Notices*, volume 45, pages 36–46. ACM, 2010.

[232] Chamikara Jayalath and Patrick Eugster. Efficient geo-distributed data processing with rout. In *Proc. ICDCS*, 2013.

[233] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems (TOCS)*, 36(2):1–49, 2019.

[234] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224. ACM, 2010.

[235] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[236] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of MLSys Conference*, 2019.

[237] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soule, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018.

[238] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 35–49, 2018.

[239] Patricia Johann and Eelco Visser. Warm fusion in stratego: A case study in generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 29(1):1–34, 2000.

[240] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Proc. FMCAD*, 2013.

[241] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

[242] Rajeev Joshi, Greg Nelson, and Keith Randall. *Denali: a goal-directed superoptimizer*, volume 37. ACM, 2002.

[243] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. Denali: A practical algorithm for generating optimal code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):967–989, 2006.

[244] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM*, 63(7):67–78, June 2020.

[245] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 2017.

[246] Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.

[247] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. Alone together: Compositional reasoning and inference for weak isolation. *Proc. ACM Program. Lang.*, 2(POPL):27:1–27:34, December 2017.

[248] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[249] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[250] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance {RDMA} systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 437–450, 2016.

[251] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided ({RDMA}) datagram rpcs. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 185–201, 2016.

[252] Nikolaos D Kallimanis and Eleni Kanellou. Wait-free concurrent graph objects with dynamic traversals. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[253] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*, 2018.

[254] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308, New York, NY, USA, 2012. ACM.

[255] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 201–217, 2020.

[256] Ken Kennedy and Kathryn S McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320. Springer, 1993.

[257] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[258] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[259] J. Kirsch, S. Goose, Y. Amir, D. Wei, and P. Skare. Survivable scada via intrusion-tolerant replication. *IEEE Transactions on Smart Grid*, 5(1):60–70, Jan 2014.

[260] Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *Approaches and Applications of Inductive Programming: Third International Workshop, AAIP 2009, Edinburgh, UK, September 4, 2009. Revised Papers 3*, pages 50–73. Springer, 2010.

[261] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, February 2014.

[262] Marios Kogias and Edouard Bugnion. Hovercraft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[263] Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. Rethinking safe consistency in distributed object-oriented programming. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[264] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 351–364, Berkeley, CA, USA, 2010. USENIX Association.

[265] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[266] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, August 2009.

[267] Sudha Krishnamurthy, William H Sanders, and Michel Cukier. An adaptive quality of service aware middleware for replicated services. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1112–1125, 2003.

[268] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[269] Philipp Kufner, Uwe Nestmann, and Christina Rickmann. Formal verification of distributed algorithms. In *Theoretical Computer Science*, volume 7604 of *LNCS*. 2012.

[270] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Software synthesis procedures. *Communications of the ACM*, 55(2):103–111, 2012.

[271] Viktor Kuncak, Huu Hai Nguyen, and Martin Rinard. Deciding boolean algebra with presburger arithmetic. *Journal of Automated Reasoning*, 36(3):213–239, 2006.

[272] Viktor Kuncak and Martin Rinard. Towards efficient satisfiability checking for boolean algebra with presburger arithmetic. In *International Conference on Automated Deduction*, pages 215–230. Springer, 2007.

[273] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4), 1992.

[274] Ori Lahav and Viktor Vafeiadis. Owicki-gries reasoning for weak memory models. In *International Colloquium on Automata, Languages, and Programming*, pages 311–323. Springer, 2015.

[275] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[276] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), 2010.

[277] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[278] Leslie Lamport. Introduction to TLA. Technical report, 1994.

[279] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

[280] Leslie Lamport. Generalized consensus and paxos. 2004.

[281] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[282] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[283] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Arnaud Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *CGO 2021*, 2021.

[284] Juneyoung Lee, Chung-Kil Hur, and Nuno P. Lopes. Aliveinlean: A verified llvm peephole optimization verifier. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 445–455, Cham, 2019. Springer International Publishing.

[285] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[286] Mohsen Lesani. Tm testing tool source code. `http://people.csail.mit.edu/lesani/companion/disc13/index.html`, 2013.

[287] Mohsen Lesani. Pvs proofs of tl2 transactional memory algorithm based on sol logic. `http://people.csail.mit.edu/lesani/companion/dissertation/`, 2014.

[288] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 357–370, New York, NY, USA, 2016. ACM.

[289] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory*, CONCUR'12, pages 516–530, Berlin, Heidelberg, 2012. Springer-Verlag.

[290] Mohsen Lesani, Victor Luchangco, and Mark Moir. Pvs framework for transactional memory verification. `http://people.csail.mit.edu/lesani/companion/concur12/index.html`, 2014.

[291] Mohsen Lesani, Todd Millstein, and Jens Palsberg. Chapar verification framework source code. `http://people.csail.mit.edu/lesani/companion/popl16/artifact/index.html`, 2014.

[292] Mohsen Lesani, Todd Millstein, and Jens Palsberg. Snowflake verification tool source code. `http://people.csail.mit.edu/lesani/companion/cav14/`, 2014.

[293] Mohsen Lesani, Todd D. Millstein, and Jens Palsberg. Automatic atomicity verification for clients of concurrent data structures. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 550–567, 2014.

[294] Mohsen Lesani and Jens Palsberg. Communicating memory transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 157–168, New York, NY, USA, 2011. ACM.

[295] Mohsen Lesani and Jens Palsberg. *Proving Non-opacity*, pages 106–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[296] Mohsen Lesani and Jens Palsberg. *Decomposing Opacity*, pages 391–405. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[297] Nicholas V Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Cerny. Conflict-aware replicated data types. *arXiv preprint arXiv:1802.08733*, 2018.

[298] Nicholas V Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Cerny. Sequential programming for replicated data stores. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–28, 2019.

[299] Cheng Li, João Leitão, Allen Clement, Nuno Preguica, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

[300] Cheng Li, João Leitão, Allen Clement, Nuno Preguica, and Rodrigo Rodrigues. Minimizing coordination in replicated systems. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 8. ACM, 2015.

[301] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguica, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[302] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 10–10, Berkeley, CA, USA, 2007. USENIX Association.

[303] Xiao Li, Farzin Houshmand, and Mohsen Lesani. Hampa: Solver-aided recency-aware replication. In *International Conference on Computer Aided Verification*, pages 324–349. Springer, 2020.

[304] Hongjin Liang and Xinyu Feng. Abstraction for conflict-free replicated data types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 636–650, 2021.

[305] Christian Lindig and Norman Ramsey. Declarative composition of stack frames. In *International Conference on Compiler Construction*, pages 298–312. Springer, 2004.

[306] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975.

[307] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the harp file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 226–238, New York, NY, USA, 1991. ACM.

[308] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. Verified tensor-program optimization via high-level scheduling rewrites. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[309] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 503–517, Berkeley, CA, USA, 2014. USENIX Association.

[310] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3s: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 423–437, Berkeley, CA, USA, 2008. USENIX Association.

[311] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. From clarity to efficiency for distributed algorithms. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 395–410, New York, NY, USA, 2012. ACM.

[312] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. Verifying replicated data types with typeclass refinements in liquid haskell. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[313] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. SOSP*, 2011.

[314] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. NSDI*, 2013.

[315] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual consistency. *Communications of the ACM*, 57(5):61–68, 2014.

[316] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 65–79, New York, NY, USA, 2021. Association for Computing Machinery.

[317] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. PLDI '15, page 22–32, New York, NY, USA, 2015. Association for Computing Machinery.

[318] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The smart way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 103–115, New York, NY, USA, 2006. ACM.

[319] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[320] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[321] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. ASPLOS*, 2008.

[322] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proc. ASPLOS*, 2006.

[323] Shiyong Lu, Arthur Bernstein, and Philip Lewis. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1070–1081, 2004.

[324] Thibaut Lutz and Vinod Grover. Lambdajit: a dynamic compiler for heterogeneous optimizations of stl algorithms. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, pages 99–108, 2014.

[325] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2, 1989.

[326] P Madhusudan and P Thiagarajan. Distributed controller synthesis for local specifications. volume 2076, pages 396–407, 07 2001.

[327] P. Madhusudan and P.S. Thiagarajan. Distributed controller synthesis for local specifications. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 396–407, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[328] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, The University of Texas at Austin, 2011.

[329] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in expressos. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 293–304, New York, NY, USA, 2013. ACM.

[330] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[331] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[332] Zohar Manna and Richard Waldinger. Synthesis: dreams→ programs. *IEEE Transactions on Software Engineering*, (4):294–328, 1979.

[333] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

[334] Zohar Manna and Richard J Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[335] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[336] Parisa Jalili Marandi, Samuel Benz, Fernando Pedone, and Kenneth P. Birman. The performance of Paxos in the cloud. In *Proc. SRDS*, 2014.

[337] Parisa Jalili Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 527–536, June 2010.

[338] Mugilan Mariappan, Joanna Che, and Keval Vora. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys '21)*, pages 1–16, 2021.

[339] Mugilan Mariappan and Keval Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys '19)*, pages 1–16, 2019.

[340] Harry Massalin. Superoptimizer – a look at the smallest program. *Palo Alto, California*, 1987.

[341] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, 1989.

[342] Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 184–195, 2015.

[343] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. Relational constraint solving in smt. In *International Conference on Automated Deduction*, pages 148–165. Springer, 2017.

[344] Matthew Milano and Andrew C Myers. Mixt: A language for mixing consistency in geodistributed transactions. 2018.

[345] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992.

[346] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Inf. Comput.*, 100(1):41–77, September 1992.

[347] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using {One-Sided}{RDMA} reads to build a fast,{CPU-Efficient}{Key-Value} store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.

[348] Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proceedings of the ACM on Programming Languages*, (OOPSLA):1–29, 2019.

[349] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[350] Akimasa Morihata, Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Hideya Iwasaki. Optimizing declarative parallel distributed graph processing by using constraint solvers. In John P. Gallagher and Martin Sulzmann, editors, *Functional and Logic Programming*, pages 166–181, Cham, 2018. Springer International Publishing.

[351] Akimasa Morihata, Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Hideya Iwasaki. Optimizing declarative parallel distributed graph processing by using constraint solvers. In *International Symposium on Functional and Logic Programming*, pages 166–181. Springer, 2018.

[352] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proc. NSDI*, 2004.

[353] Kartik Nagar and Suresh Jagannathan. Automated parameterized verification of crdts. In *International Conference on Computer Aided Verification*, pages 459–477. Springer, 2019.

[354] Kartik Nagar, Prasita Mukherjee, and Suresh Jagannathan. Semantics, specification, and bounded verification of concurrent libraries in replicated systems. In *International Conference on Computer Aided Verification*, pages 251–274. Springer, 2020.

[355] Sreeja Nair, Gustavo Petri, and Marc Shapiro. Proving the safety of highly-available distributed objects. In *ESOP 2020-29th European Symposium on Programming*, 2020.

[356] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The cise tool: Proving weakly-consistent applications correct. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, pages 2:1–2:3, New York, NY, USA, 2016. ACM.

[357] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System.

[358] Francesco Zappa Nardelli, Peter Sewell, Jaroslav Sevcik, Susmit Sarkar, Scott Owens, Luc Maranget, Mark Batty, and Jade Alglave. Relaxed memory models must be rigorous. In *Exploiting Concurrency Efficiently and Correctly Workshop*, 2009.

[359] Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. Verifying and improving halide's term rewriting system with program synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[360] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[361] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[362] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[363] Chris Olston and Jennifer Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. Technical report, Stanford, 2000.

[364] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[365] online url. Lean-mlir, 2022.

[366] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, 50(6):619–630, 2015.

[367] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, 1976.

[368] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*, volume 2. Springer, 2020.

[369] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proc. FSE*, 2008.

[370] Seo Jin Park and John Ousterhout. Exploiting commutativity for practical fast replication. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 47–64, 2019.

[371] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[372] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[373] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 207–218, New York, NY, USA, 2013. ACM.

[374] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

[375] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. SOSP*, 1997.

[376] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.

[377] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. PLDI*, 1988.

[378] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proc. LICS*, 2009.

[379] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. *ACM SIGPLAN Notices*, 49(6):396–407, 2014.

[380] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 297–310. ACM, 2016.

[381] Gordon D. Plotkin. The origins of structural operational semantics. In *Proc. Journal of Logic and Algebraic Programming*, pages 60–61, 2004.

[382] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118, 2015.

[383] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *ACM SIGPLAN Notices*, volume 51, pages 522–538. ACM, 2016.

[384] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

[385] Mathias Preiner, Aina Niemetz, and Armin Biere. Counterexample-guided model synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I 23*, pages 264–280. Springer, 2017.

[386] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Elixir: A system for synthesizing concurrent graph programs. In *ACM SIGPLAN Notices*, volume 47, pages 375–394. ACM, 2012.

[387] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Synthesizing parallel graph programs via automated planning. In *ACM SIGPLAN Notices*, volume 50, pages 533–544. ACM, 2015.

[388] Yewen Pu, Rastislav Bodik, and Saurabh Srivastava. Synthesis of first-order dynamic programming algorithms. *ACM SIGPLAN Notices*, 46(10):83–98, 2011.

[389] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[390] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 249–258, New York, NY, USA, 2006. ACM.

[391] Vincent Rahli. Interfacing with proof assistants for domain specific programming using EventML. 10th International Workshop on User Interfaces for Theorem Provers, 2012.

[392] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. Clotho: directed test generation for weakly consistent database systems. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, 2019.

[393] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. Repairing serializability bugs in distributed database programs via automated schema refactoring. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 32–47, 2021.

[394] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, Robert J Harrison, and Ponnuswamy Sadayappan. A domain-specific compiler for a parallel multiresolution adaptive numerical simulation environment. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 40. IEEE Press, 2016.

[395] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J Harrison, and Ponnuswamy Sadayappan. On fusing recursive traversals of kd trees. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 152–162. ACM, 2016.

[396] Krithi Ramamritham and Calton Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):997–1007, 1995.

[397] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4(4):243–254, January 2011.

[398] Mahesh Ravishankar, Paulius Micikevicius, and Vinod Grover. Fusing convolution kernels through tiling. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pages 43–48, 2015.

[399] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Mahesh Ravishankar, Vinod Grover, Atanas Rountev, Louis-Noël Pouchet, and P Sadayappan. Domain-specific optimization and generation of high-performance gpu code for stencil computations. *Proceedings of the IEEE*, 106(11):1902–1920, 2018.

[400] Michel Raynal and André Schiper. From causal consistency to sequential consistency in shared memory systems. volume 1026 of *LNCS*. 1995.

[401] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton

Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark. ISCA '20, page 446–459. IEEE Press, 2020.

[402] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Automating formal proofs for reactive systems. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 452–462, New York, NY, USA, 2014. ACM.

[403] Tom Ridge. Verifying distributed systems: the operational approach. In *Proc. POPL*, 2009.

[404] Ricardo Rocha and John Launchbury. *Practical Aspects of Declarative Languages: 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*, volume 6539. Springer, 2011.

[405] Marko A Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10. ACM, 2015.

[406] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[407] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3), 2011.

[408] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[409] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1311–1326, New York, NY, USA, 2015. ACM.

[410] Signe Rüsch, Ines Messadi, and Rüdiger Kapitza. Towards low-latency byzantine agreement protocols using rdma. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 146–151. IEEE, 2018.

[411] John Rushby. *Bus Architectures for Safety-Critical Embedded Systems*, pages 306–323. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[412] Olli Saarikivi, Margus Veanes, Todd Mytkowicz, and Madan Musuvathi. Fusing effectful comprehensions. *SIGPLAN Not.*, 52(6):17–32, June 2017.

[413] Caitlin Sadowski, Stephen N. Freund, and Cormac Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *Proc. ESOP*, 2009.

[414] Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. Treefuser: a framework for analyzing and fusing general recursive tree traversals. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):76, 2017.

[415] Laith Sakka, Kirshanthan Sundararajah, Ryan R Newton, and Milind Kulkarni. Sound, fine-grained traversal fusion for heterogeneous trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 830–844. ACM, 2019.

[416] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422*, 2017.

[417] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[418] N. Schiper, V. Rahli, R. van Renesse, M. Bickford, and R.L. Constable. Developing correctly replicated databases using formal tools. In *Proc. DSN*, 2014.

[419] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGPLAN Notices*, volume 48, pages 305–316. ACM, 2013.

[420] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[421] Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. Using domain-specific languages for analytic graph databases. *Proceedings of the VLDB Endowment*, 9(13):1257–1268, 2016.

[422] Peter Sewell, James J Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. *Journal of Functional Programming*, 17(4-5):547–612, 2007.

[423] Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. *Consistency in 3D*. PhD thesis, Institut National de la Recherche en Informatique et Automatique (Inria), 2016.

[424] Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, INRIA, 2011.

[425] Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[426] G Shashidhar and Rupesh Nasre. Lighthouse: An automatic code generator for graph algorithms on gpus. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 235–249. Springer, 2016.

[427] Kensen Shi, Jacob Steinhardt, and Percy Liang. Frangel: Component-based synthesis with control structures. *Proc. ACM Program. Lang.*, 3(POPL):73:1–73:29, January 2019.

[428] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[429] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 515–527, 2018.

[430] A Sinha, S. Malik, Chao Wang, and A Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In *MEMOCODE*, 2011.

[431] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 413–424, New York, NY, USA, 2015. ACM.

[432] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 413–424, New York, NY, USA, 2015. ACM.

[433] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. *ACM SIGPLAN Notices*, 51(6):326–340, 2016.

[434] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2021, page 21–31, New York, NY, USA, 2021. Association for Computing Machinery.

[435] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15:475–495, 2013.

[436] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 167–178, 2007.

[437] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Notices*, volume 40, pages 281–294. ACM, 2005.

[438] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM Sigplan Notices*, 41(11):404–415, 2006.

[439] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proc. FSE*, 2010.

[440] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[441] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. From program verification to program synthesis. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 313–326, 2010.

[442] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, page 205–217, New York, NY, USA, 2015. Association for Computing Machinery.

[443] Philippe Suter, Robin Steiger, and Viktor Kuncak. Sets with cardinality constraints in satisfiability modulo theories. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 403–418. Springer, 2011.

[444] TensorFlow. XLA: Optimizing Compiler for TensorFlow, 2022. [Online; accessed 10-November-2022].

[445] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP*, 1995.

[446] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[447] Robert H Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, 1979.

[448] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery.

[449] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices*, 49(6):530–541, 2014.

[450] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 530–541, New York, NY, USA, 2014. Association for Computing Machinery.

[451] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A New Algorithm for Generating All the Maximal Independent Sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[452] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 48(6):287–296, 2013.

[453] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In DavidA. Watt, editor, *Compiler Construction*, volume 1781 of *LNCS*, pages 18–34. Springer Berlin Heidelberg, 2000.

[454] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 7. ACM, 2016.

[455] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[456] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. *ACM SIGPLAN Notices*, 43(6):125–135, 2008.

[457] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *ACM Sigplan Notices*, volume 45, pages 327–338. ACM, 2010.

[458] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, Jan 2013.

[459] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6), 2008.

[460] C. von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2004.

[461] Keval Vora. *Exploiting Asynchrony for Performance and Fault Tolerance in Distributed Graph Processing*. PhD thesis, University of California, Riverside, 2017.

[462] Keval Vora. Lumos: Dependency-Driven Disk-based Graph Processing. In *USENIX Annual Technical Conference (USENIX ATC '19)*, pages 429–442, 2019.

[463] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. pages 237–251, 2017.

[464] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *ASPLOS*, pages 237–251, 2017.

[465] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co.

[466] Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. Replication-aware linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 980–993, 2019.

[467] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107, 2017.

[468] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466, 2017.

[469] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54. USENIX Association, July 2021.

[470] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proc. PPoPP*, 2006.

[471] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, February 2006.

[472] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[473] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 233–251, 2018.

[474] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104, 2015.

[475] Matthew Weidner, Heather Miller, and Christopher Meiklejohn. Composing and decomposing op-based crdts with semidirect products. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–27, 2020.

[476] Michael Whittaker and Joseph M Hellerstein. Interactive checks for coordination avoidance. *Proceedings of the VLDB Endowment*, 12(1):14–27, 2018.

[477] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed system. In *Proc. PLDI*, 2015.

[478] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical bft execution. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 123–138, New York, NY, USA, 2011. ACM.

[479] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Proc. PLDI*, 2005.

[480] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A Acar, et al. Quartz: superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 625–640, 2022.

[481] Zhilei Xu, Shoaib Kamil, and Armando Solar-Lezama. Msl: A synthesis enabled language for distributed implementations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 311–322, Piscataway, NJ, USA, 2014. IEEE Press.

[482] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proc. NSDI*, 2009.

[483] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Type-and content-driven synthesis of sql queries from natural language. *arXiv preprint arXiv:1702.01168*, 2017.

[484] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proc. NSDI*, 2009.

[485] Yichen Yang, Mangpo Phitchaya Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. In *Proceedings of MLSys Conference*, 2021.

[486] Y. C. Yeh. Safety critical avionics for the 777 primary flight controls system. In *Digital Avionics Systems, 2001. DASC. 20th Conference*, volume 1, pages 1C2/1–1C2/11 vol.1, Oct 2001.

[487] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. Sidetrack: Generalizing dynamic atomicity analysis. In *Proc. PADTAD*, 2009.

[488] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 253–267, New York, NY, USA, 2003. ACM.

[489] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation-Volume 4*, page 21. USENIX Association, 2000.

[490] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation-Volume 4*, page 21. USENIX Association, 2000.

[491] Haifeng Yu and Amin Vahdat. Efficient numerical error bounding for replicated network services. In *VLDB*, pages 123–133. Citeseer, 2000.

[492] Haifeng Yu and Amin Vahdat. Combining generality and practicality in a conit-based continuous consistency model for wide-area replication. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 429–438. IEEE, 2001.

[493] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 29–42. ACM, 2001.

[494] Haifeng Yu and Amin Vahdat. Minimal replication cost for availability. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 98–107. ACM, 2002.

[495] Haifeng Yu, Amin Vahdat, et al. Efficient numerical error bounding for replicated network services. In *VLDB*, pages 123–133. Citeseer, 2000.

[496] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proc. OSDI*, 2014.

[497] Pamela Zave. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, March 2012.

[498] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of crdts. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 33–48. Springer, 2014.

[499] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of CRDTs. volume 8461 of *LNCS*. 2014.

[500] Rachid Zennou, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. Checking causal consistency of distributed databases. In *International Conference on Networked Systems*, pages 35–51. Springer, 2019.

[501] Junpeng Zha, Hongjin Liang, and Xinyu Feng. Verifying optimizations of concurrent programs in the promising semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 903–917, 2022.

[502] Chi Zhang and Zheng Zhang. Trading replication consistency for performance and availability: an adaptive approach. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 687–695. IEEE, 2003.

[503] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proceedings of the 3rd workshop on Scientific Cloud Computing*, pages 13–22. ACM, 2012.

[504] Yongzhe Zhang, Hsiang-Shang Ko, and Zhenjiang Hu. Palgol: A high-level dsl for vertex-centric graph processing with remote data access. In *Asian Symposium on Programming Languages and Systems*, pages 301–320. Springer, 2017.

[505] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA):121:1–121:30, October 2018.

[506] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[507] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shizhi Tang, Lei Xie, Kezhao Huang, and Zhihao Jia. Ollie: Derivation-based tensor program optimizer, 2022.

[508] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 301–316, 2016.

[509] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 375–386, 2015.