University of California
Los Angeles

Program Analyses for Cloud Computations

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Sai Deep Tetali

2015

Abstract of the Dissertation

Program Analyses for Cloud Computations

by

Sai Deep Tetali
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2015
Professor Todd D. Millstein, Chair

Cloud computing has become an essential part of our computing infrastructure. In this model, data and programs are hosted in (often third-party) data centers that provide APIs for data access and running large-scale computations. It is used in almost all major internet services companies and increasingly being considered by other organizations to host their data and run analytics. However several challenges lie in its full-scale adoption, chief among them being security, performance and correctness. Security is important as both client data and computations need to be sent to third-party data centers. Performance is important as cloud computing involves several development iterations, each running on large-scale data. Correctness is critical as cloud frameworks are complex distributed systems serving billions of users every day.

In this dissertation, I argue that program analysis techniques can help address the above key challenges of cloud computing. I describe three projects that illustrate different aspects of the solution space: MrCrypt is a system that uses static analysis to guarantee data confidentiality in cloud computations by using homomorphic encryption schemes. Vega is a library that significantly improves incremental performance by rewriting modified workflows to use previously computed results. Kuai is a distributed, enumerative model checker that verifies correctness in Software Defined Networks, the networking layer used by many data centers.

The dissertation of Sai Deep Tetali is approved.

Tyson Condie

Miryung Kim

Paulo Tabuada

Todd D. Millstein, Committee Chair

University of California, Los Angeles

2015

Table of Contents

List of Figures

List of Tables

Acknowledgments

None of this would be possible without my advisor, Todd Millstein. After "inheriting" me from Rupak Majumdar, my previous advisor, he took me under his wing and provided an amazing amount of support. He has taught me so much about doing good research and communicating ideas. It is a learning experience just observing him do his thing.

Rupak was just as influential in my PhD journey. He is a bundle of brilliant ideas. But just as important to me, he was flexible and guided me even when I changed topics.

I thank my committee, Tyson Condie, Miryung Kim, Paulo Tabuada and Jens Palsberg, for their valuable feedback and being flexible with my constraints.

I owe a huge debt of gratitude (and a few lunches) to Steve Arbuckle, Craig Jessen, Edna Todd and Freda Robinson who took care of a lot of red-tape for me.

I would also like to thank my many academic collaborators: Amit Goel, Sava Krstić, Mohsen Lesani, Zilong Wang and Matteo Interlandi (among others) for the many fruitful discussions I have had with them. They have influenced me in more ways than they could imagine.

I have had just as many "fruitful" discussions with my non-academic collaborators (a.k.a. friends) here in Los Angeles who have been with me since the beginning of this journey. To name a few: Matt Wang, Jacob Mathew, Sarah Promnitz and Sharath Gopal. I would especially like to thank Siddharth Joshi, who, while not in LA, has spend just as much (if not more!) time talking to me as anyone in LA. You guys are the best!

I had the distinct fortune to work with Robert Englund. Thanks Bob for teaching me so much through stimulating conversations and for all the faculty center lunches! Thanks for treating me like family.

Speaking of family, thanks Sruthi for being a constant support and especially for enduring my crankiness while writing this dissertation. Thanks Sai Roop for being an amazing brother

and never losing faith in me. Thanks mom and dad for, well, just about everything. Somehow they figured out exactly when to push it and when to let me be.

MrCrypt (presented in chapter 2) benefitted immensely from discussions with Todd Millstein and Rupak Majumdar who helped craft the narrative presented here. Mohsen Lesani, with guidance from Todd Millstein, developed its formal semantics. Special thanks to anonymous reviewers of our OOPSLA 2013 paper for suggesting the large-scale benchmark suite we used to run performance experiments.

The idea of performing incremental re-computation in the face of code changes (presented in chapter 3) was inspired by Tyson Condie. Subsequently, I collaborated with Todd Millstein, Matteo Interlandi and Tyson Condie on Vega, the project presented in this dissertation. Special thanks to Todd Millstein for his detailed feedback on the chapter drafts.

Kuai (presented in chapter 4) is a collaboration with Zilong Wang and Rupak Majumdar. Zilong Wang developed the formal semantics and produced the proofs of correctness for various optimizations used by the tool.

Vita

2007	Teaching Assistant, DA-IICT, Gandhinagar, India. Taught sections of Computer Organization and Programming course under the direction of Professor Asim Banerjee.
2008	Intern, Microsoft Research, Bangalore, India.
2008	B. Tech in Information and Communication Technology, DA-IICT, Gandhinagar, India.
2008–2009	Research Software Developer, Microsoft Research, Bangalore, India.
2009–2015	Graduate Student Researcher, Computer Science Department, UCLA.
2010 & 2011	Intern, Intel Corporation, Hillsboro, Oregon.
2011 & 2013	Intern, Max Planck Institute for Software Systems, Kaiserslautern, Germany.
2012	Teaching Assistant, Computer Science Department, UCLA. Taught sections of Introduction to Programming (CS 31) course under the direction of Professor David Smallberg.
2012	M.S in Computer Science, UCLA.
2013	Intern, LogicBlox, Atlanta, Georgia.
2014	Teaching Assistant, Computer Science Department, UCLA. Taught sections of Programming Languages (CS 131) course under the direction of Professor Todd Millstein.
2014	Intern, Microsoft Research, Redmond.

Publications

Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani and Sai Deep Tetali. *Compositional May-Must Program Analysis: Unleashing The Power of Alternation.* In Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010).

Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, Sai Deep Tetali and Aditya V. Thakur. *Proofs from Tests.* In 2010 IEEE Transactions on Software Engineering (TSE).

Aditya V. Nori, Sriram K. Rajamani, Sai Deep Tetali and Aditya V. Thakur. *The Yogi Project: Software Property Checking via Static Analysis and Testing.* In Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009).

Amit Goel, Sava Krstić, Rupak Majumdar and Sai Deep Tetali. *Quantified Interpolation for SMT.* In the 9th International Workshop on Satisfiability Modulo Theories (SMT 2011).

Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar and Todd Millstein. *MrCrypt: Static Analysis for Secure Cloud Computations.* In Proceedings of 2013 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA).

Rupak Majumdar, Sai Deep Tetali and Zilong Wang. *Kuai: A model checker for software-defined networks.* In Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design, (FMCAD 2014).

CHAPTER 1

Introduction

Cloud computing has become an essential part of our computing infrastructure. It provides a utility model where computing resources are "rented out" to clients who upload data and programs to (often third-party) data centers. The main draw towards this model is that it can provide massive scalability, on-demand, without upfront investment in computing resources. It is used in almost all major internet services companies and increasingly being considered by other organizations to host their data and run analytics. However several challenges lie in its full-scale adoption, chief among them being security, performance and correctness.

In this dissertation, I argue that program analysis techniques can help in improving several aspects of cloud computing, including security, performance and correctness.

Security

Security concerns arise as potentially sensitive data and computations need to be uploaded to external data centers. Additionally cloud providers maximize utilization by having several client computations run on the same physical machine. Bugs in cloud infrastructure, malicious system administrators or lack of proper security measures by the clients can leak data across client computations. Data confidentiality, an important part of security, has become increasingly important in the cloud due to several publicized breaches. Examples include private photos of many users (including celebrities) that were leaked from Apple's iCloud photo service and private data and photos leaked from Snapchat — a cloud based messaging service.

However, traditional encryption methods are not a good fit for cloud computations. These methods aim to protect data when it is stored (in the file system or database) but force applications to decrypt it on-the-fly to perform any computation. However, when a program is running in the cloud, we cannot assume that the data decrypted for computation will not be leaked.

In the next chapter of the dissertation, I use program analysis techniques to achieve data confidentiality. Traditional systems security relied on either the data or computation to be under user's control. However, in a common use case for cloud computing, clients upload data and computation to servers that are managed by a third-party infrastructure provider. MrCrypt is a system that provides data confidentiality in this setting by executing client computations using homomorphic encryption schemes. These schemes allow computations run directly on encrypted data thus removing the need for decryption and achieving confidentiality. MrCrypt statically analyzes a program to identify the set of operations on each input data column, in order to select an appropriate homomorphic encryption scheme for that column, and then transforms the program to operate over encrypted data. The encrypted data and transformed program are uploaded to the server and executed as usual, and the result of the computation is decrypted on the client side. I have implemented MrCrypt for Java and illustrate its practicality on three standard benchmark suites for the Hadoop MapReduce framework [Whi12].

Performance

I consider improving the performance of incremental computation in the third chapter of the dissertation. Big data applications (i.e. those that use datasets that are beyond the capability of traditional databases to store and mange [MCB11]) are a major part of cloud computing and a large part of a big data pipeline includes ingesting data from several sources, transforming them into a common format and doing several exploratory computations on the data to understand its structure. In many such exploratory scenarios, programmers start out with an initial workflow, observe the output and iteratively improve the workflow

by adding new transformations or modifying existing ones until the output is in the desired form. Every change is usually run anew, discarding all work done in the previous iterations. The immense scale of the data typical in cloud computations makes these kinds of iterations very time consuming. It is not uncommon for data scientists to wait for 3+ hours, only to find that they should have filtered out some obvious outliers. Typically users select a small sample from the dataset and perform explorations on that set. This approach is incomplete in most cases. Existing frameworks such as Percolator [PD10] and Naiad [MMI13] allow computations to be run incrementally when there are small changes in data, but they do not deal with changes in code.

I describe Vega, a library for incremental computation that handles code changes. It handles them in two different ways: Vega pushes the changed transformations towards the end of the workflow as much as possible by using commutativity. This allows the reuse of as much of the original work as possible. And finally the changes in the code are transformed into changes in data by taking a difference of the old and new transformers' outputs that are then pushed downstream similar to Naiad. To achieve this, Vega introduces an API of commutative operators on top of the default ones. Vega has been implemented on top of the Spark framework and I demonstrate significant performance benefits in various ETL [Vas09] (extract, transform, load) use cases.

Correctness

Finally, cloud frameworks (for computation, communication, authentication etc.) have to be distributed to be able to scale across many thousands of nodes that are present in data centers. This increases the complexity of the system tremendously, with many actors performing tasks concurrently, and makes testing and debugging especially difficult. Since cloud programs often serve millions of users, ensuring correctness of the underlying stack on which they are built is especially important. High profile bugs include the authentication layer of Dropbox, a cloud based file synchronization service, that let users log into any account without requiring a password.

Many such frameworks have a master-workers architecture where there is a single logical master and many workers. The master usually has complicated (and often changing) logic for scheduling tasks and responding to events. Workers usually have the same (relatively simple) program to run commands given by the master. Frameworks such as Google's MapReduce [DG08], Hadoop [Whi12], Dryad [IBY07], Spark [ZCD12] etc. are built using this architecture. A major challenge in verifying correctness of such systems involves dealing with both program logic of the master and unbounded communication with workers.

Software Defined Networks (SDN) [FRZ13] is the application of the master-worker architecture to large-scale networks. In software-defined networking, a software controller manages a distributed collection of switches by installing and uninstalling packet-forwarding rules in the switches. SDNs allow flexible implementations for expressive and sophisticated network management policies. They are increasingly becoming the networking layer of choice for data centers and the correctness of their policies determine the correctness of several mission-critical cloud applications.

I describe Kuai, a distributed enumerative model checker for SDNs. Kuai takes as input a controller implementation written in Murphi [Dil96], a description of the network topology (switches and connections), and a safety property, and performs a distributed enumerative reachability analysis on a cluster of machines. Kuai uses a set of partial order reduction techniques specific to the SDN domain that help reduce the statespace dramatically. In addition, Kuai performs an automatic abstraction to handle unboundedly many packets traversing the network at a given time and unboundedly many control messages between the controller and the switches. I demonstrate the scalability and coverage of Kuai on standard SDN benchmarks. I show that the set of partial order reduction techniques significantly reduces the state spaces of these benchmarks by many orders of magnitude. Another novelty of Kuai is in demonstrating that current large-scale systems architecture can provide excellent execution platforms for software verification.

CHAPTER 2

MrCrypt

2.1 Introduction

A common use case for cloud computing involves clients uploading data and computation to servers managed by third-party infrastructure providers. Since the data and programs are no longer in an environment controlled by the client, private client data may be exposed to adversarial clients in the cloud server, either by accidental misconfigurations or through malicious intent. Publicized incidents involving the loss of confidentiality or integrity of customer data [Kow08] only heighten these concerns. The threat of potential violations to the confidentiality and integrity of customer data is a key barrier to the adoption of cloud computing based on third-party infrastructure providers.

One way to alleviate these concerns is to store encrypted data on the cloud and decrypt it as needed during the cloud computation. However, this approach is insufficient to protect against adversaries who can potentially view the memory contents of the server, for example a curious cloud administrator or a malicious client running on the same machine. Therefore, all computations must be performed on the client side [LKM04, MSL10], which severely reduces the attractiveness of the cloud model. Theoretically, fully homomorphic encryption schemes [RAD78, Gen09] offer the possibility of uploading and storing encrypted data on the cloud and performing arbitrary operations on the encrypted data. Unfortunately, current implementations of fully homomorphic encryption schemes are still prohibitively expensive [Gen10, GH11].

In this chapter I present MrCrypt, a system that automatically transforms programs in

order to enforce data confidentiality.

MrCrypt is based on the key insight that many useful cloud computations only perform a small number of operations on each column of the data. While fully homomorphic encryption is expensive, there are efficient encryption schemes that support common subsets of operations. Thus, instead of encoding each column using a fully homomorphic encryption scheme, one can encrypt it using a more efficient scheme that supports only the necessary operations. For example, suppose that the client program sums all the elements of a column. Instead of a fully homomorphic encryption scheme, one can encrypt the column using the Paillier cryptosystem [Pai99], which guarantees that

$$\mathsf{Paillier}(x) \cdot \mathsf{Paillier}(y) = \mathsf{Paillier}(x + y)$$

for any x, y, where $\mathsf{Paillier}(x)$ denotes the encryption of x using the scheme, and the multiplication of the codewords on the left is modulo a public key. Thus to compute on the encrypted data, the program must simply take the product of all codewords. When the result is decrypted on the client side, the sum of all the numbers is recovered. Similarly, the El Gamal cryptosystem [ElG85] is homomorphic for multiplication:

$$\mathsf{ElGamal}(x) \cdot \mathsf{ElGamal}(y) = \mathsf{ElGamal}(x \cdot y)$$

Similar schemes exist for performing equality checks and range comparisons. In this way, MrCrypt reduces the problem of securing cloud computations to that of identifying the subset of primitive operations (such as addition, multiplication, equality checks, and order comparisons) performed on each column of the input, in order to determine the most efficient encryption scheme that can be used for the column. MrCrypt uses a static analysis to perform the inference task, called *encryption scheme inference*, on imperative programs. Given the results of encryption scheme inference, it is straightforward to produce the translated program that will be sent to the cloud along with the encrypted data.

2.2 Background

2.2.1 Homomorphic Encryption Schemes

A (public-key) encryption scheme consists of three algorithms (K, E, D) for key-generation, encryption, and decryption. The key-generation procedure K is a randomized algorithm that takes a security parameter λ as input and outputs a secret key sk and public key pk. The encryption procedure E is a randomized algorithm that takes pk and a plaintext m as input and outputs a ciphertext ψ. The decryption procedure D takes sk and ψ as input and outputs the plaintext m, i.e., $D(sk, E(pk, m)) = m$. The computational complexity of all of these algorithms must be polynomial in λ.

Given a binary operation f, an encryption scheme is *homomorphic for* f if there exists a (possibly randomized) polynomial-time algorithm $Eval_f$, which takes as input the public key pk and a pair of ciphertexts (ψ_1, ψ_2) such that

$$D(sk, Eval_f(pk, \psi_1, \psi_2)) = f(D(, \psi_1), D(, \psi_2))$$

Informally, if ψ_1 and ψ_2 are respectively encryptions of plaintexts m_1 and m_2 under pk, then $Eval_f(pk, \psi_1, \psi_2)$ is an encryption of $f(m_1, m_2)$ under pk. For a set of operations F, an encryption scheme is homomorphic for F if it is homomorphic for each $f \in F$. An encryption scheme is said to be *fully homomorphic* if it is homomorphic for $\{+, \times\}$. It is easy to see that in this case, any polynomial-sized arithmetic circuit can be evaluated purely on the ciphertext.

In addition to homomorphic encryption schemes, I shall also consider encryption schemes with a related property in which the result of an operation can be computed (in clear text) directly on the ciphertext:

$$Eval_f(pk, \psi_1, \psi_2) = f(D(, \psi_1), D(, \psi_2))$$

For example, using a deterministic encryption scheme, one can check if two values are equal simply by comparing the ciphertexts for equality, without requiring any information about

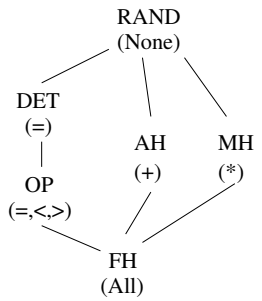

Figure 2.1: A lattice of encryption schemes.

the original values. In the following, I abuse notation and call such encryption schemes "homomorphic" as well.

Given a set of operations $\mathcal{F}$, one can arrange encryption schemes in a partial order: an encryption scheme $\mathcal{E}_1$ is "less than" a scheme $\mathcal{E}_2$ if $\mathcal{E}_1$ is homomorphic for $F_1 \subseteq \mathcal{F}$, $\mathcal{E}_2$ is homomorphic for $F_2 \subseteq \mathcal{F}$, and $F_2 \subseteq F_1$. A fully homomorphic encryption scheme is the unique minimal element in this ordering, and a random encryption scheme is the maximal element (it is not homomorphic for any operation). Typically, one expects that encryption schemes "higher" in the ordering (i.e., supporting fewer operations) will have more efficient implementations.

MrCrypt's implementation is parameterized by a lattice of encryption schemes. The tool employs several forms of encryption, which are shown as a lattice in Figure 2.1 along with the set of operations that each scheme supports.*RAND* (random) supports no homomorphic operations [Sch96]; *DET* (deterministic) supports equality testing [Sch96]; *OP* (order-preserving) supports comparisons [BCL09, BCO11]; *AH* (additive homomorphic) supports addition [Pai99]; *MH* (multiplicative homomorphic) supports multiplication [ElG85]; *FH* (fully homomorphic) supports all operations [Gen09]. The *DET* and *OP* schemes produce their results in clear text, while the other schemes are homomorphic in the strict sense. Because fully homomorphic encryption is not currently practical, MrCrypt does not include an implementation of it, but I show that it is rarely required in the benchmark programs.

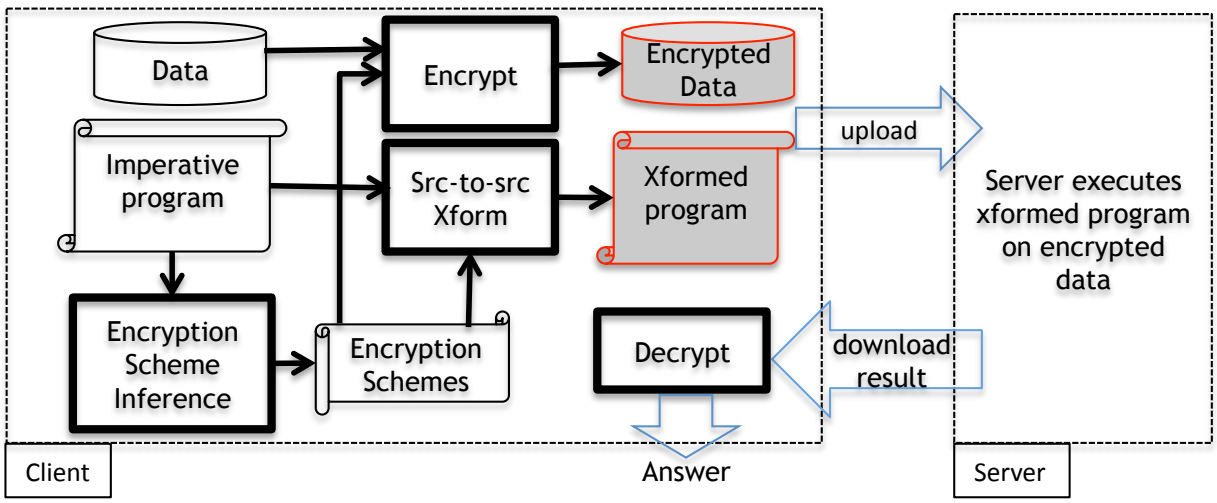

Figure 2.2: Architecture of MrCrypt; solid boxes show implemented components .

2.2.2 MapReduce

MapReduce [DG10] is a popular distributed programming model introduced by Google for processing large data sets on clusters. In this model, the computation is divided into three stages. The map stage invokes a user-defined *map* function in parallel over the data and produces a list of intermediate key/value pairs. A *shuffle* stage in the MapReduce framework then sorts all the resulting records based on the keys and groups together all values associated with the same key. Finally, the reduce stage invokes a user-defined *reduce* function in parallel to combine the values associated with each key in some fashion, typically producing just zero or one final values per key. Hadoop MapReduce is an open-source implementation of MapReduce that is widely used by both researchers and major corporations (e.g., Facebook, Twitter) to perform large-scale distributed computations.

2.3 Approach

The architecture of MrCrypt is shown in Figure 2.2. Given a Java program and a data set, MrCrypt performs static analysis on the program to determine an encryption scheme for each input column, program variable, and program constant such that each program operation can be performed homomorphically on encrypted data (*encryption scheme inference*). The analysis first generates constraints based on how operations in the program are used, and then it solves the constraints to determine the most efficient (i.e., highest in the lattice) encryption scheme to use for each part of the program.

Next the results of the encryption scheme inference are used to transform the program. Specifically, each call to a primitive operation f in the program is replaced by a call to $Eval_f^{\mathcal{E}}$, where $\mathcal{E}$ is the encryption scheme inferred for the arguments to f. Similarly, each program constant c is replaced by its encrypted value $E^{\mathcal{E}}(c)$, where $\mathcal{E}$ is the encryption scheme inferred for c. The data set is also encrypted according to the inferred encryption schemes on the client side.

Finally, the encrypted data and the transformed program are uploaded to the (untrusted) server, where the program is executed. The result is sent back to the client, where it is decrypted.

We assume a passive (honest-but-curious) adversary model. That is, the adversary can view all the data uploaded to the server, the program that is uploaded, as well as the entire execution trace of the program. However, we assume that the adversary does not change the data, the program, or the result of the program (i.e., no integrity attacks). Please refer [TLM13] for details about the formalism and security guarantees.

2.3.1 Example

The program in listing 2.1 is inspired by wireless fitness trackers. Users continually upload fitness information such as the number of calories burnt during a workout to the cloud. This program uses the MapReduce programming model [DG08] to compute the total number of

```
Integer map(Integer entryDate, Integer entryMonth, Integer entryYear, Integer caloriesBurnt) {
   if (entryYear > 2012)
      return caloriesBurnt;
   else
      return 0;
}
Integer reduce(List<Integer> caloriesBurntList) {
   Integer sum = 0;
   for (Integer caloriesBurnt : caloriesBurntList)
      sum += caloriesBurnt;
   return sum;
}
```

Listing 2.1: Fitness Program

calories burnt by a user since the beginning of the year 2013. This result can be used further to compute statistics such as the average calories burnt per day. Every record includes the number of calories burnt and the date associated with the event (given by year, month and day fields). For expository purposes some implementation details of MapReduce frameworks are omitted, for example the need to parse input data from a file and to produce key-value pairs as results. However, the example is illustrative of common MapReduce use cases.

The user-defined map function is executed on each row of the data, and it has the effect of producing all entries from the *caloriesBurnt* column for which the associated entry year is greater than 2012. The MapReduce framework collects the values returned by the map function invocations and passes the resulting list to the user-defined reduce function, which sums the calories.

Encryption Inference For a variable x, let $\sigma(x)$ denote the encryption scheme for x, and similarly for a constant c. When necessary to disambiguate, variables or constants are subscripted with the name of the function in which they appear. From line 2, it is concluded that $\sigma(\text{entryYear}) = \sigma(2012)$ and that $\sigma(\text{entryYear})$ should support at least $>$. From lines 3 and 5, $\sigma(\text{caloriesBurnt}_{\text{map}}) = \sigma(0_{\text{map}})$. From line 8, $\sigma(\text{sum}) = \sigma(0_{\text{reduce}})$. From line 10, $\sigma(\text{sum}) = \sigma(\text{caloriesBurnt}_{\text{reduce}})$ and $\sigma(\text{sum})$ should support at least $+$. Finally, the semantics of the MapReduce framework requires that the result from the map function must

```
AH_Integer map(RAND_Integer entryDate, RAND_Integer entryMonth, OP_Integer entryYear,
AH_Integer caloriesBurnt) {
   if (OP_GT(entryYear, [[OP_E(2012)]])
      return caloriesBurnt;
   else
      return [[AH_E(0)]];
}
AH_Integer reduce(List<AH_Integer> caloriesBurntList) {
  AH_Integer sum = [[AH_E(0)]];
   for (AH_Integer caloriesBurnt : caloriesBurntList)
      sum = AH_PLUS(sum, caloriesBurnt);
   return sum;
}
```

Listing 2.2: Encrypted Fitness Program

use the same encryption scheme as the data items in the argument list to the reduce function. Given the lattice of encryption schemes from Figure 2.1, the best solution to these constraints maps σ(entryYear) and σ(2012) to OP, σ(entryDate) and σ(entryMonth) to $RAND$ (since there are no constraints on these variables), and everything else to AH.

The translated program for the example is shown in listing 2.2. First, the primitive $>$ function is replaced by the corresponding operation in the order-preserving encryption scheme, which is denoted by `OP_GT`, and similarly $+$ is replaced by `AH_PLUS`. Second, each constant is replaced by an appropriately encrypted version of that constant. For example, `[[OP_E(2012)]]` is used to denote the encrypted value of the constant 2012 under the order-preserving encryption scheme. Note that this value is computed statically and inserted into the transformed program in place of the original constant.

2.4 Implementation

I have implemented the encryption scheme inference and transformation algorithms for Java programs.

2.4.1 Encryption Schemes

I briefly describe the encryption schemes that are currently supported in MrCrypt. Since there is no efficient scheme for FH currently, MrCrypt throws an exception if FH is required. (The experimental evaluation shows that this is rarely the case.) In general, I follow the security parameters from prior work [PRZ11].

RAND is a probabilistic encryption scheme that guarantees IND-CPA but which does not support any operations on the encrypted data. *RAND* is implemented using Blowfish [Sch94] for 32-bit integers and AES [DR02] for strings in CBC mode and with a random initialization vector. Blowfish produces a 64-bit ciphertext and AES outputs ciphertext as 128-bit blocks.

DET is a deterministic encryption scheme: the same plaintext generates the same ciphertext. Thus, *DET* allows checking for equality on the encrypted values. I make the standard assumption that Blowfish and AES block ciphers are pseudorandom permutations and use these encryption schemes in ECB mode. For values up to 64 and 128 bits, I use Blowfish and AES respectively after padding smaller plaintexts to at least 64 bits. For longer strings, I use AES with a variant of CMC mode [HR03] with a zero initial vector, as is done in CryptDB [PRZ11].

OP is an order-preserving encryption scheme that allows checking order relations between encrypted data items. I use the implementation of *OP* in CryptDB, which follows the algorithm in [BCL09]. Since MrCrypt only does order operations on 32-bit integers, a ciphertext size of 64 bits for each value is used.

AH allows performing addition on encrypted data. I use CryptDB's implementation of the Paillier cryptosystem [Pai99] to support *AH*. MrCrypt generates 512 bits of ciphertext for each 32-bit value.

MH allows performing multiplication on encrypted data. I use the El Gamal cryptosystem [ElG85] to support *MH*. MrCrypt generates 1024-bit ciphertext for each 32-bit integer.

2.4.2 Encryption Scheme Inference

MrCrypt is built as an extension to the Polyglot compiler framework [NCM03]. Polyglot is designed to allow language extensions and analysis tools to be written on top of a base compiler for Java. MrCrypt is written in Scala and interfaces with Polyglot's intermediate representation of the Java bytecode. The tool takes as input a Java program and an encryption-scheme lattice and outputs a translated Java program which runs on the encrypted domain. It uses Polyglot compiler's dataflow framework and soundly handles imperative updates, aliasing, and arbitrary Java control flow in the standard way.

The inference algorithm has been extended in several ways to handle Java programs that employ the Hadoop MapReduce framework; most of these extensions would also be useful in conjunction with other cloud computing frameworks. First, the user-defined map function in Hadoop is given a portion of a file representing the input data and must perform custom processing based on the file format to parse the data into columns. MrCrypt requires programmers to annotate the parsing code so it can understand which variables get values from which columns, which are identified by number. For example, the user should annotate the following statement, which gets the fifth field in a line of input, with `@getColumn(5)`:

```
x = Library.splitLine(input, ' ').get(5);
```

Similarly, the statement that outputs `x` as the sixth field in a record should be annotated with `@putColumn(6,x)`.

Second, the inference algorithm has been extended to handle common data structures. The map function returns a list of key-value pairs and the reduce function accepts a list of values as an argument. Further, programmers often use container data structures such as hashmaps and hashsets to remove duplicates, order elements, etc. The implementation recognizes these data structures by type and encrypts their elements rather than the data structures themselves. In general the implementation uses a single logical variable for the purpose of encryption-scheme inference for a data structure's elements, ensuring that all elements are encrypted with the same scheme. However, I introduce two logical variables to

handle lists of key-value pairs, so that keys and values can use different encryption schemes from one another.

MrCrypt also requires data structures to be annotated with the operations they perform on their elements, in order to preserve these operations in the encrypted domain. Specifically, the standard Java hashmap and hashset classes are annotated to require the equality operation on elements.

Third, the shuffle phase of MapReduce sorts the intermediate keys produced by the map phase, thereby requiring support for order comparisons. However, in many cases the final output does not depend on the keys being sorted, instead just requiring that intermediate values be grouped by their key. Therefore, MrCrypt allows programmers to annotate that sorting is not required for correctness of the program, allowing it to choose deterministic encryption for the keys (which preserves equality, necessary for grouping values by key) rather than order-preserving encryption. The shuffle phase will be performed as usual by the Hadoop framework but will no longer guarantee that the underlying plaintext keys are in sorted order.

2.4.3 Optimizations

In order to scale to large datasets, I implemented a number of optimizations to the translator and runtime which can be categorized as follows:

Data serialization. Textual formats are very commonly used for MapReduce programs, with numbers represented as decimal strings. This encoding is highly inefficient for the mostly binary ciphertext data. Hence MrCrypt uses a binary serialization system, Avro[1], to store ciphertext.

Tuning Hadoop framework parameters. I tune Hadoop framework parameters such as the number of simultaneous map and reduce tasks, heap size, RAM used for shuffle phase, total number of reduce tasks, block size for the distributed file system, etc. based on the

[1] http://avro.apache.org

hardware on which they run. This is a manual process which depends on the program, data size as well as the cluster resources. These optimizations apply equally well to both the plaintext and ciphertext programs as they have similar data access patterns.

Efficient encoding of constants. I implemented a simple optimization for the case when the map function emits constant integer values. For example, in the standard MapReduce implementation of word count the map function emits the tuple $\langle w, 1 \rangle$ for every word w in the input, and the reduce function sums up the numbers in the second component of each tuple. While this is efficient in the plaintext, the *AH* ciphertext for the number 1 in the translated program is 512 bits long. This causes significant slowdowns as the map's output is saved to the disk and read back and the entire data is kept in memory while sorting.

The tool applies an optimization whenever either a constant integer value or a `final` variable initialized to an constant integer value is emitted by the map function. The optimization creates a dictionary in the translated program which associates symbols (represented by integers) with their ciphertexts. In the word count example, the plaintext map function contains `@putColumn(col0, word)` and `@putColumn(col1, 1)` for every word and the translated map function contains `@putColumn(col1, S)` and the reduce function has access to the dictionary which maps S to the *AH* ciphertext of 1.

2.5 Evaluation

This section describes the experimental evaluation of MrCrypt. I have applied encryption scheme inference to all programs in three MapReduce benchmark suites, in order to illustrate the applicability of the approach. I have also executed programs from one of the three suites on a cluster at scale to determine the run-time overhead of executing on encrypted data. Finally, I have used a set of microbenchmarks to isolate the client-side and server-side costs of encryption.

2.5.1 Benchmark Programs

The three benchmark suites are respectively listed in Tables 2.1, 2.2 and 2.3. For each benchmark, the number of source lines of code determined by the SLOCCount tool [Whe15] are listed.

PIGMIX2[2] is a set of 17 benchmark programs written for the Pig framework, which provides a high-level language for writing large-scale data analysis programs called Pig Latin [ORS08a]. The framework compiles Pig Latin scripts into MapReduce programs and the runtime manages the evaluation of these programs. The PIGMIX2 benchmarks each come with Pig Latin scripts as well as hand-written MapReduce programs which the authors believe are efficient ways to execute the scripts. The programs run on a dataset primarily comprised of two tables: the PageViews table has 9 columns and the Users table has 6 columns.

Pavlo *et al.* [PPR09] compare the performance of parallel databases that accept SQL queries with equivalent MapReduce programs. Their evaluation employs a standard word-search task [DG10] along with five other MapReduce benchmarks that perform various analytics queries,which I hereafter refer to as "the Brown suite."

The Purdue MapReduce Benchmarks (PUMA) Suite [ALT12] contains 13 diverse MapReduce programs dealing with different computational and data patterns. In addition to performing encryption scheme inference, I also run these benchmarks on the large datasets that are provided with the benchmarks: 50GB Wikipedia documents for the Word Count, Grep, Inverted Index, Term Vector, and Sequence Count benchmarks; 27GB movies data for the Histogram Movies and Histogram Ratings benchmarks; and upwards of 28GB of synthetic data for the rest of the benchmarks.

A few modifications to the benchmarks are made to work around current limitations of MrCrypt:

- The supported encryption schemes do not handle floating-point numbers, so all bench-

[2]https://cwiki.apache.org/PIG/pigmix.html

marks that use floating-point numbers have been converted to use integers.

- The implementation of *OP* supports comparisons for integers but not for strings, necessitating modifications to three benchmarks. First, I modified the Aggregate Variant benchmark in the Brown suite to represent an IP address as four integers rather than a single string. Second, Self Join in the PUMA suite takes as input alphanumerically sorted text consisting of the string "entryNum" followed by 10 digits. I modified the input dataset to only include the numbers. Finally, Tera Sort in the PUMA suite sorts a column for which the input data consists of 10 random characters. I restrict the input data for this column to be populated by numeric characters.

- Three benchmarks in PIGMIX2 — L8, L15, and L17 — compute an average over some columns, which requires support for division. I modified these benchmarks to instead return a pair of the sum and the element count.

2.5.2 Experimental Setup

The experiments were run on the compute cluster at Max Planck Institute for Software Systems. The MapReduce computations were run on two Dell R910 rack servers each with 4 Intel Xeon X7550 2GHz processors, 64 x 16GB Quad Rank RDIMMs memory and 174GB storage. The experiments ran on a total of 64 cores and had access to 1TB of RAM and 348GB of permanent storage. The machines were a shared resource and were under light load from other research projects. The Hadoop framework was configured to run 60 map and reduce tasks in parallel across the 64 available computational units.

In addition I used four Dell R910 rack servers (each with 2 Intel Xeon X5650 2.66GHz processors, 48GB RAM and 1TB hard disks) to host the distributed file system. No MapReduce computations were run on these machines and they were only used to serve input data and to store results. These machines were also a shared resource under regular load from other researchers.

2.5.3 Experimental Results

We are interested in three key metrics:

1. Annotation burden: How much extra work must the programmer do to make the existing MapReduce programs run securely?

2. Inference effectiveness: Does MrCrypt find the most efficient encryption scheme? How often is fully homomorphic encryption required?

3. Time and space overhead: How much runtime and storage cost does encrypted execution incur?

2.5.3.1 Annotation Burden

As mentioned in the implementation section, MrCrypt requires programmers to annotate parsing code to correlate variables with the input columns from which they are read. The simple `getColumn` and `putColumn` annotations were sufficient to cover all of the file formats used in the benchmarks.

The encryption inference can otherwise be accomplished without any user annotations. However, as mentioned earlier, MrCrypt allows users to annotate the fact that keys in a MapReduce program's output need not be in sorted order. I found that sorting is unnecessary in 29 of the 36 benchmarks because the specification does not require sorted output, so I included the associated annotation for these programs.

On average I added 12 annotations to each benchmark, which amounts to 7% of the lines of code.

2.5.3.2 Encryption Scheme Inference

Since *FH* is inefficient in practice, the utility of the tool depends on whether it is able to find efficient encryption schemes for real-world MapReduce programs. I present the results

for the three benchmark suites in Tables 2.1, 2.2 and 2.3. For each benchmark, I measure the source lines of code by using the SLOCCount tool [Whe15] along with the encryption schemes inferred for the input columns. For each encryption scheme the number of columns for which that scheme was inferred is mentioned in parenthesis. For each benchmark, the analysis time was less than 1 second, and the entire compilation time (including analysis and translation) was less than 5 seconds.

On 24 of 36 benchmarks, MrCrypt can identify encryption schemes to support the necessary functionality without requiring fully homomorphic encryption. Hence 66.7% of the programs can be executed securely through the system. I also manually analyzed each benchmark to verify the correctness of these results.

In the four cases of the PIGMIX2 suite where *FH* is required, the programs perform both equality and addition on the same column of data, for example to obtain a sum of all distinct values in the column. One of the benchmarks in the Brown suite (UDF) invokes performs string operations that MrCrypt does not support, one (Search) requires regular expression evaluation, and the other benchmark (Join) performs a sort on data obtained by computing a sum over some column. I am not aware of any homomorphic encryption scheme other than *FH* supporting both order comparisons and addition. In the PUMA suite, *FH* is required for regular-expression evaluation (Grep) and for computing cosine similarity (K-means and Classification).

Finally, MrCrypt determines that two benchmarks in the PUMA suite require *FH* for intermediate data produced by the map function. First, Term Vector counts all occurrences of words in documents and sort them by their frequency. This is implemented by using the map function to output ⟨doc-name, word, 1⟩ for every word, and the reduce to sum up all the 1s for each word in a document and then sort the words using the sums. Hence the numbers are both summed up and compared for order which results in *FH* for that variable. However, since we need to use DET to encrypt the input words to preserve equality, the number of occurrences of each (encrypted) word is already being leaked to the adversary. Hence leaving the integers in plaintext would not entail any extra loss of confidentiality, so

in fact the benchmark can be executed securely without *FH*.

Second, Histogram Movies uses the map function to calculate the average rating of each movie rounded to the nearest 0.5. The reduce function then counts the number of movies with the same average rating. This functionality requires addition, division, and rounding operations and hence requires *FH*. However, I observe that we can refactor the benchmark into two different MapReduce programs to avoid *FH*. I refer to these two programs as Histogram Movies 1 and Histogram Movies 2, and they are also listed in Table 2.3. Histogram Movies 1 performs just the map phase of the original benchmark, with a trivial reduce, outputting the sum of all ratings of each movie along with their count. Histogram Movies 2 takes as input the average rating of each movie and performs just the reduce phase of the original benchmark, counting the number of movies with each average rating. MrCrypt infers encryption schemes for each of these benchmarks that allows them to execute securely without requiring *FH*.

To achieve the functionality of the original Histogram Movies benchmark, the client must decrypt the *AH* ciphertext output from Histogram Movies 1, re-encrypt it to use *DET* after computing the average and rounding it to the nearest 0.5 (and then doubling it to make it an integer), and provide the resulting ciphertext as input to Histogram Movies 2. While the client must perform some extra work, it does so on a small amount of data. On the input dataset, Histogram Movies 1 operates on 27GB of movie-rating data while Histogram Movies 2 only operates on 4MB of data that results from summing those ratings per movie (Table 2.4).

2.5.3.3 Time and Space Overhead

The approach incurs two main sources of performance overhead, which we evaluate separately.

Client-side Overhead The client-side overhead consists of the need to encrypt the input data before sending it to the cloud and decrypt the output data from the computation. I

evaluated this cost by measuring the time taken for encrypting and decrypting 500 random 32-bit integers. The OP, AH, and MH schemes take an average of 10ms, 4ms, and 1.5ms to encrypt each integer, respectively, and less than 0.5ms per decryption. Blowfish (the basis for $RAND$ and DET) has much less overhead of 200ns for each encryption and decryption operation. Thus, for example, encrypting one million data items with AH requires a bit more than one hour. However, in the target application domains the encryption can be performed incrementally as data is generated, and the encryption cost is amortized across multiple runs of the cloud computations.

Server-side Overhead The server-side overhead consists of the need to perform homomorphic operations on encrypted data rather than the original operations on the plaintext data. To isolate this overhead I developed a set of microbenchmarks, each of which performs a single operation one million times on the input data. For each operation I have one version of the microbenchmark that accepts plaintext integers and another version that uses the appropriate homomorphic encryption scheme to operate on ciphertext. I use a corpus of 10,000 32-bit integers and their corresponding ciphertexts as the input data. The performance overhead for encrypted execution is significant: slowdowns of 2× for DET, 4× for OP, 500× for AH, and 75× for MH.

Fortunately, the overheads on real MapReduce benchmarks are much lower, since the homomorphic operations contribute only a small percentage of the overall time. To evaluate the overhead of encryption on real-world data, I ran the PUMA benchmarks at scale on large data on a cluster. For each benchmark, I report the runtime for the original program, and the runtime for the transformed program. I also report the plaintext size and ciphertext size of the input data. I tabulate the results in Table 2.4.

The homomorphic operations add an insignificant overhead and the size of the ciphertext is the main factor in determining the runtime of the translated programs. On average the translated programs take 2.61× as long to execute as the original programs. However, Histogram Movies 1 is an outlier due to the need for AH, which uses 512 bits of ciphertext for each 32-bit integer, on a large amount of data. Excluding this benchmark the translated

programs take an average of 1.57× as long to execute as the original programs.

In the three benchmarks where the program operating on ciphertext runs faster than the plaintext program (Adjacency List, Self Join, and Tera Sort), the speedup is due to using binary formats for encoding the encrypted numbers while the plaintext input uses a particularly inefficient textual format to encode numbers. In these benchmarks, numbers are padded with zeros to keep the length of each column the same so as to make use of the built-in sorting algorithm in the shuffle phase. Hence the number 1 would be represented as 0000000001. This approach uses 10 bytes to encode the range of 32-bit integers while the encrypted data uses at most 8 bytes to store the resulting 64-bit OP ciphertext. The binary format uses variable-length encoding and hence might use fewer than 8 bytes in some cases.

2.5.4 Discussion

Space Efficiency. Encryption schemes like AH require a significant blowup in space, which has a direct impact on execution time as well. We can reduce the overhead for Pallier encryption (the implementation of AH) using a packing optimization [GZ07].

Avoiding Fully Homomorphic Encryption. I showed earlier how refactoring the Histogram Movies benchmark can make it amenable to this approach, and believe there are additional opportunities along these lines. For example, four benchmarks that currently require FH require a "sum of distinct elements" functionality, which typically looks as follows:

```
Integer f(List<Integer> revenues) {
  HashSet<Integer> hs = new HashSet<Integer>();
  for (Integer r: revenues) hs.add(r);
  int sum = 0;
  for(Integer r: hs) sum += r.intValue();
  return new Integer(sum);
}
```

The revenues column has two operations performed on it: equality (from the hashset) and addition. Hence the tool infers FH in this case. However, this program can be run securely by keeping two copies of the revenues column, one for equality and the other for addition, and keeping a correspondence between them (I use the class `P2` for pairs, along with associated

Benchmark	Lines Of Code	Encryption Schemes Inferred for Inputs
L1	137	DET(2), RAND(7)
L2	148	DET(1), RAND(8)
L3	185	AH(1), DET(1), RAND(7)
L4	141	DET(2), RAND(7)
L5	169	DET(1), RAND(8)
L6	139	DET(3), FH(1), RAND(5)
L7	158	DET(1), OP(1), RAND(7)
L8	170	AH(2), RAND(7)
L9	196	OP(1), RAND(8)
L10	245	OP(3), RAND(6)
L11	184	DET(1), RAND(8)
L12	218	AH(1), DET(3), OP(1), RAND(4)
L13	182	DET(1), RAND(8)
L14	183	DET(1), RAND(8)
L15	188	DET(2), FH(2), RAND(5)
L16	134	DET(1), FH(1), RAND(7)
L17	259	FH(5), OP(20)

Table 2.1: Inference results on the PIGMIX2 benchmarks.

operations, from the Java library `fj`[3]):

```
Integer f(List<Integer> erevenues,
          List<Integer> arevenues) {
  HashSet<Integer> hs = new HashSet<Integer>();
  List<Integer> distincts = list();
  for (P2<Integer, Integer> p:
       erevenues.zip(arevenues)) {
    if (!hs.contains(p._1())) {
      hs.add(p._1());
      distincts.cons(p._2());
    }
  }
  int sum = 0;
  for(Integer r: distincts)
```

[3]http://functionaljava.org/

Benchmark	Lines of Code	Encryption Schemes Inferred for Inputs
Search	109	FH(1)
Select	71	OP(1), RAND(2)
Aggregate	99	AH(1), DET(1), RAND(7)
Aggregate Variant	162	AH(1), DET(3), RAND(7)
Join	518	AH(1), DET(2), FH(1), OP(1), RAND(4)
UDF	58	AH(1), DET(1), FH(1), RAND(6)

Table 2.2: Inference results on benchmarks from the Brown suite.

```
            sum += r.intValue();
  return new Integer(sum);
}
```

It would be interesting to explore performing such preprocessing automatically in order to extend the applicability of the current approach.

In the next chapter, I use automatic rewrites to do performance optimizations on cloud programs. Cloud programs are generally split into small functions with well-defined semantics (like the map and reduce functions presented here) which makes a number of analyses amenable to these applications. In this chapter we took advantage of the usually simple dataflow that exists between map and reduce functions to add constraints in the inference algorithm. In the next chapter, we will take advantage of commutativity across such functions to optimize programs for incremental performance when the code in the workflow changes.

Benchmark	Lines Of Code	Encryption Schemes Inferred for Inputs
Word Count	88	DET(1)
Inverted Index	126	DET(1)
Term Vector*	187	DET(1)
Self Join	136	OP(1)
Adjacency List	157	OP(2)
K-Means	428	DET(1), FH(1), OP(1)
Classification	228	DET(1), FH(1), OP(1)
Histogram Movies*	132	AH(1), RAND(2)
Histogram Movies 1	113	AH(1), RAND(2)
Histogram Movies 2	98	AH(1), DET(1)
Histogram Ratings	115	DET(1), RAND(2)
Sequence Count	124	DET(1)
Ranked Inverted Index	127	DET(4), OP(1)
Tera Sort	192	OP(1), RAND(1)
Grep	55	FH(1)

Table 2.3: Inference results on the PUMA benchmark suite.

Benchmark	Original Program Runtime (sec)	Transformed Program Runtime (sec)	Plaintext Size (GB)	Ciphertext Size (GB)
Word Count	528	1064	50	79
Inverted Index	395	658	50	79
Term Vector	556	1114	50	79
Self Join	252	234	28	26.1
Adjacency List	823	769	28	26.5
Histogram Movies 1	138	1801	27	388
Histogram Movies 2	22	32	0.004	0.067
Histogram Ratings	214	427	27	36
Sequence Count	492	1006	50	79
Ranked Inverted Index	305	525	37.8	60.3
Tera Sort	1080	1062	28	26.9

Table 2.4: Performance results on the PUMA benchmark suite

CHAPTER 3

Vega

3.1 Introduction

Data analysts report spending a majority of their time writing code to ingest data from several sources, transform them into a common format and perform several exploratory computations on the data to understand its structure [KPH12]. A difficult problem as it is, this is exacerbated by the immense sizes of data increasingly being collected by organizations. The scale of the data (ranging from several hundred gigabytes to petabytes) led to the coining of the term "Big Data Applications". These programs typically run on the cloud and aim to extract patterns from several semi-structured data sources.

Extracting such insights from data is usually an iterative process. Programmers start out with an initial workflow, observe the output and iteratively improve the workflow by adding new transformations or modifying existing ones until the output is in the desired form.

Prior work ([MMI13], [PD10]) has addressed incremental re-computation in the face of changes to the input. However, these approaches are not applicable when the code changes. Existing frameworks run each development iteration anew, discarding all work done in the previous iterations. The immense scale of the data typical in cloud computations makes these kinds of iterations very time consuming. It is not uncommon for data scientists to wait for 3+ hours, only to find that they should have filtered out some obvious outliers. To alleviate this, users typically select a small sample from the dataset and perform explorations on that set. This approach is incomplete in most cases and many times leads to additional time spent trying to figure out what was missing in the sample that was present in the original

dataset.

In this chapter I describe Vega, a library that performs incremental re-computation in the face of code changes.

3.2 Problem Definition and Approach

To make the problem concrete, define a workflow to be a list of transforms. A transform is usually a data-parallel function such as map, filter, reduce etc. Please see the previous chapter for semantics of map and reduce functions. Let us consider consider a workflow $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$. I.e the input to T_i is the output of T_{i-1}. We consider the output of a transform to be a multiset. In our setting, a user has already run the workflow, let us call the output produced by transform T_i as O_i. The user observes the final output, O_n and decides to add a new transform T_α after T_k. The new workflow now becomes $T_1 \rightarrow \ldots T_k \rightarrow T_\alpha \rightarrow T_{k+1} \ldots \rightarrow T_n$. Let O'_i be the output produced by transform T_i in the changed workflow. Clearly $(1 <= j <= k) \Rightarrow O_j = O'_j$ and hence a trivial re-computation solution is to execute $O_k \rightarrow T_\alpha \rightarrow T_{k+1} \rightarrow \ldots \rightarrow T_n$. Our goal is to do better than this.

We represent a transform by its type and an anonymous function that determines the action taken by the transform. We use Scala[1] syntax for specifying the functions. For example `Map (x => x + 1)` is a map transform that increments every input number by 1, `Reduce ((c1, c2) => c1 + c2)` is a reduce transform that sums all the values.

A function $f : A \rightarrow B$ is invertible if there exists a function $f_{inv} : B \rightarrow A$ such that $\forall x \in A.\, f_{inv}(f(x)) = x$. If a function is invertible, then it is one-to-one, i.e., $\forall x_1, x_2 \in A.\, f(x_1) = f(x_2) \Leftrightarrow x_1 = x_2$. We use $\circ$ to denote function composition: $\forall x \in A.\, (g \circ f)(x) = g(f(x))$.

As a running example, let us take the standard word count workflow used to test several big data frameworks. We assume the input is a list of words. The workflow is as follows:

```
Input -> Map (x => (x, 1)) -> Shuffle -> Reduce ((c1, c2) => c1 + c2)
```

[1] `http://www.scala-lang.org`

The map transform maps every word to a tuple. The shuffle transform aggregates all the tuples with the same key (first element). The reduce transform sums all the 1's corresponding to each word and outputs the count. Figure 3.1 illustrates this example on a small sample of data.

A first step in cleaning many datasets is to remove stop-words. Inspired by that, we look at a case where the user decides to include a filter at the beginning of the workflow to remove all c's. In this scenario, the user has already run the above workflow and now wants to incrementally run the following workflow:

```
Input -> Filter (x => x != 'c')
      -> Map (x => (x, 1)) -> Shuffle -> Reduce ((c1, c2) => c1 + c2)
```

Here T_α is `Filter (x => x != 'c')` .

3.2.1 First Strategy

We first take notice of existing frameworks like Naiad [MMI13] and Percolator [PD10] that handle small changes in data. Once a computation is run on some input, these frameworks can execute the same computation on small changes to input very quickly. Usually they can get results in time proportional to the changes in the inputs. Vega deals with a slightly different problem where the code changes in each iteration. However Vega can still leverage their insights by transforming changes in code into changes in data. To do this, we rewrite the entire workflow so that instead of computing the result directly, it computes the changes (or Δs) that should be made to the old results to produce the new result.

To present it formally, consider the changed workflow, $T_1 \to \dots T_k \to T_\alpha \to T_{k+1} \dots \to T_n$, where T_α was added. Using this approach, this workflow is rewritten as follows: $O_k \to \delta T_\alpha \to \Delta T_{k+1} \dots \to \Delta T_n$. Each ΔT_m outputs 2 multisets, Δ_{m+} and Δ_{m-} such that $O'_m = (O_m \cup \Delta_{m+}) \setminus \Delta_{m-}$. For our convenience, we denote an element $e \in \Delta_{m+}$ as $+(e)$ and an element $e \in \Delta_{m-}$ as $-(e)$. Let $O_\alpha = T_\alpha(O_k)$. δT_α starts off the delta computation by producing the Δ sets as follows: $\Delta_{\alpha+} = O_\alpha \setminus O_k$ and $\Delta_{\alpha-} = O_k \setminus O_\alpha$.

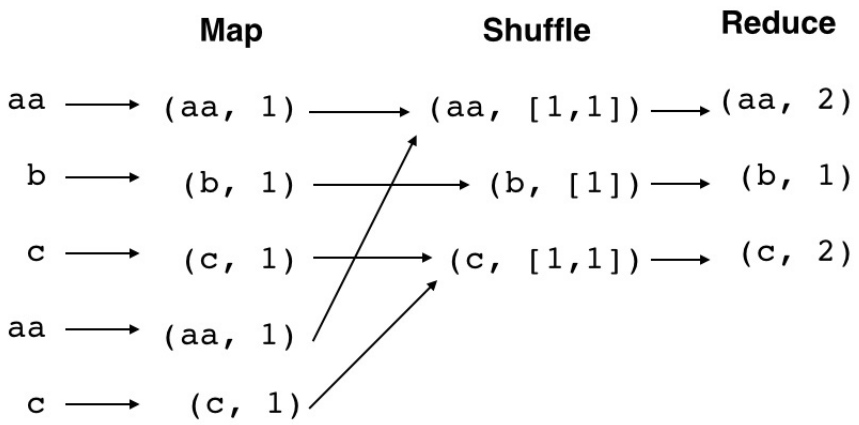

Figure 3.1: Word count workflow

In the word count example, the original workflow is now rewritten to use transforms on changes in data. For example δfilter is: `(x => if (x == 'c') then -(x))`. This indicates that the value x whenever it matches the letter 'c' should no longer be included downstream. Map and shuffle transforms are defined similarly. Reduce transform aggregates all the Δs and then inverts their values to *subtract* it from the previously computed result. Figure 3.2 shows the input and outputs of Δ transforms. From the figure, we can see that the incremental computation only uses the two inputs that are different from the previous result.

As I show in section 3.4.1.1, this performs very well for small to medium data sizes. However, for large data, this approach takes as much time as rerunning the entire computation. After further investigation, it became clear that there are a couple of reasons for the degradation in performance:

Space: Since many delta operators require previous outputs to be cached in memory, as the data size increases, more intermediate data of the original computation has to be kept in memory along with the Δs of the incremental computation.

Time: In most workflows, the time to shuffle data dominates the runtime. As the data size increases, the size of the Δs that need to be shuffled increases, causing a significant amount of time spent in the shuffle phase.

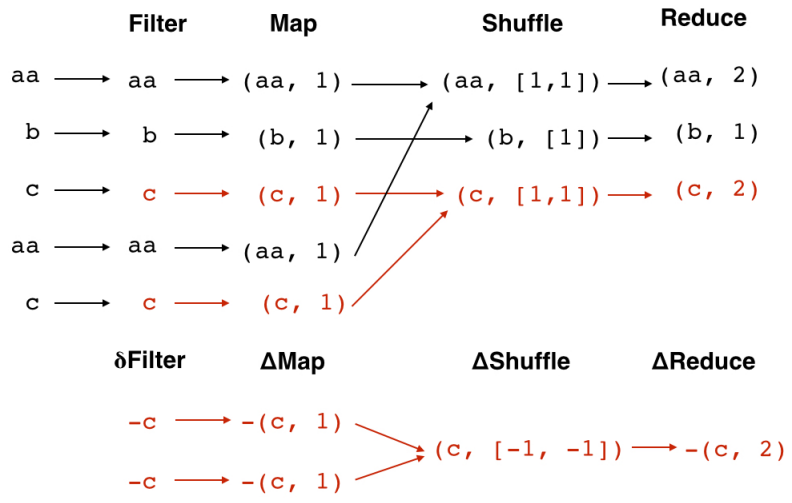

Figure 3.2: Word count workflow using Δ transforms. The changes are colored red

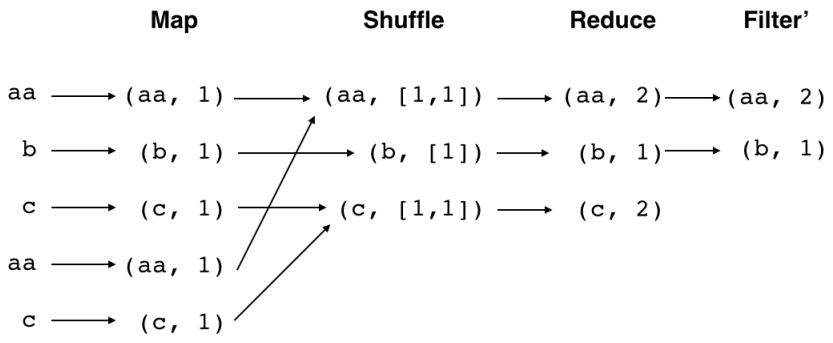

Figure 3.3: Word count workflow with filter pushed to the end

This leads to the insight that the farther downstream a new operator is placed, the less the amount of data that needs to be kept in memory while running the incremental computation. In addition, placing the new operator past a shuffle will also save significant time.

3.2.2 Commutative-rewrites

The insight leads to the idea of optimizing the introduced transform so it can be placed farther downstream. The key idea is to see if we can rewrite T_α into T'_α so that $O_k \to T_{k+1} \to T'_\alpha \to \ldots T_n$ produces the same output as the existing workflow $O_k \to T_\alpha \to T_{k+1} \to \ldots T_n$. This saves us from recomputing T_{k+1}. This process is repeated until we reach the end of the workflow or we encounter a non-commutative transform.

Going back to our word count example, we first notice that the map function is invertible. The inverse function is given by $m_{inv}((w, c)) : w$. That is, the inverse just picks the first component on the tuple. Using this information, we can rewrite the filter function so it can use the map's output as follows resulting in the new workflow:

```
Input -> Map (w => (w, 1)) -> Filter' (tup => minv(tup) != 'c')
      -> Shuffle -> Reduce ((c1, c2) => c1 + c2)
```

We then notice shuffle transform aggregates tuples by the keys and our filter is on the keys. So we can apply the filter after the shuffle step is done. Finally the reduce transform aggregates the values and does not change the keys. So we can move the filter past this transform as well. This results in the final workflow:

```
Input -> Map (x => (x, 1)) -> Shuffle
      -> Reduce ((c1, c2) => c1 + c2) -> Filter' (tup => minv(tup) != 'c')
```

Using commutative-rewrites, we have pushed the transform to the end of the workflow. The final execution strategy takes the cached output of the original workflow and runs it through the optimized filter. The optimized workflows performs significantly better for all

sizes of data as presented in section 3.4.1.1. The reason for the scalability is the the incremental computation now runs in time proportional to the output of the initial computation. The output of the word count workflow depends on the number of unique words present in the data and this number does not increase significantly on many real world data sets.

3.2.3 Combining the strategies

Commutative-rewrites are useful when pushed past transforms that produce significantly less output relative to their inputs, such as after a reduce. Δ computation is more general while commutative-rewrites might need inverses of functions to be available. Thus Vega attempts to push a new operator as far past shuffles in the workflow as possible first and then computes the changes in data brought about by the new transform. The rest of the workflow proceeds using Δ computation. This gives us the Δs (i.e. changes) from the final result of the original computation. The incremental result is computed by integrating these changes in the old result.

3.3 Design and Implementation

3.3.1 Vega API

Vega public API is sketched in listing 3.1. Programmers create workflows by instantiating the `WorkFlow` class and add transforms using the `inject` method. This allows transforms to be injected at any place in the workflow. Then they call the `run` method to execute it and get the results. After observing the results, they are able to introduce new transforms to the workflow by using the `inject` method again.

The library provides several versions of map and filter transforms. Each map transform takes a function that performs the map along with its inverse. If the inverse is `null`, then the function is assumed to be noninvertible. Vega introduces new transforms (MapKey, MapValue, FilerKey, FilterValue) on top of default map and filter transforms as transforms

```
//transforms
class Map(f: (A => B), finv: (B => A)) extends Transform[A, B]
class Filter(f: (A => Boolean)): Transform[A, A]

//pairwise transforms
class MapKey(f: (K => K'), finv: (K' => K)) extends Transform[(K, V), (K', V)]
class MapValue(f: (V => V'), finv: (V' => V)) extends Transform[(K, V), (K, V')]
class FilterKey(f: (K => Boolean)): Transform[(K, V), (K, V)]
class FilterValue(f: (V => Boolean)): Transform[(K, V), (K, V)]
class Reduce(f ((V, V) => V'), inv: (V => V), fzero: (V => Boolean))
          : Transform[(K, V), (K, V')]

class Workflow() {
  //For injecting transform 't' in the workflow immediately past the transform 'after'
  def inject(t: Transform[B, C], after: Transform[A, B]): Unit

  //Run the workflow and return the result
  def run(): Array[A]
}
```

Listing 3.1: Vega API

over disjoint fields can be commuted past each other even if they are not invertible. Vega also provides a suite of functions that can be used in transforms that includes several invertible and distributive functions.

3.3.2 Implementation

Vega is implemented as a library on top of Apache Spark framework. The basic unit of a Vega workflow is the class `Transform` (see listing 3.2). This associates each transform with a Spark RDD, the basic datastructure used by Spark for execution. It is required to support one function, `t.commuteWith(u)` that decides if two transforms, t and u, adjacent to each other in the workflow can be commuted. `commuteWith` returns `None` if they cannot be commuted, or if they can, it returns a new transformer that can be inserted after u. In addition, each transform keeps track of the fields read and written by it. This is used in the implementation of `commuteWith` function to optimize transforms acting on disjoint fields (see section 3.3.2.1).

```
abstract class Field
case object Row extends Field
case object Column extends Field

//transform from A to B
abstract class Transform[A, B] {
  val readSet: Set[Field] = Set(Row, Column)
  val writeSet: Set[Field] = Set(Row, Column)
  val rdd: RDD[B] = null //Spark RDD corresponding to this transform

  def commuteWith(other: Transform[B, C]): Option[Transform[C, C]]
}
```

Listing 3.2: Vega API

The `WorkFlow` class provides two ways to chain transforms together. All transforms added to the `WorkFlow` object (using the `inject` method) before calling `run` are executed regularly. After the initial run, `inject` method adds transforms by performing rewrites to run them incrementally.

3.3.2.1 Commutative-rewrites

Currently we only support maps and filters for commutative-rewrites. The `commuteWith` function is defined by the rules in tables 3.1 and 3.2. $t.f$ refers to the filter function if t is a filter transform, $t.m$ refers to the map function if t is a map transform and $t.r$ the reduce function if t is reduce transform. f^{-1} refers to the inverse function of f.

The table matches transforms of type u the columns with that of type of t in the rows. In particular, the cell for t/u contains the code t' such that $t \circ u$ is equivalent to $u \circ t'$. For example, if t is a transform that maps only keys and u is a filter only on values, then t can be safely commuted with u and no rewrite is necessary. Hence the table cell corresponding to that case only has t. On the other hand, if t is a map on values, then the map has to be inverted before applying u filter. This is given by $t.f \circ u.m^{-1}$.

Similar case arises when t and u are maps on the same field (key or value). In that case, u's map first needs to be inverted, then t applied (because t logically occurs before u in the

$u \rightarrow$	Filter		Map	
$t \downarrow$	Key	Value	Key	Value
Filter (Key)	t	t	$t.f \circ u.m^{-1}$	t
Filter (Value)	t	t	t	$t.f \circ u.m^{-1}$
Map (Key)	None	t	$u.m \circ t.m \circ u.m^{-1}$	t
Map (Value)	t	None	t	$u.m \circ t.m \circ u.m^{-1}$

Table 3.1: Rewrite rules 1

workflow) and finally u's map needs to be applied again. This is given by $u.m \circ t.m \circ u.m^{-1}$.

If an invertible map is applied to keys, then it is one-to-one and hence preserves the grouping of keys done by shuffle. Hence shuffle and reduce transforms can be safely commuted with such maps. If a map is not invertible, it could potentially require another stage of shuffling to re-group the values. However, this shuffle is faster as it operates only on the inputs that are remapped.

Finally, a reduce function cannot commute with an arbitrary map on values. This is because the reduce function has aggregated the values and there is no way, in general, to reapply the map on the aggregation. However, in the case where the map function distributes over the reduce function, we can apply the map after the reduce aggregation is completed. The distributive property is defined as follows: $u.r(t.m(a), t.m(b)) = t.m(u.r(a, b))$. This property implies that the map function can be applied to the aggregate value computed by reduce function. For example if a map function is doubling its inputs, it can be pushed after a reduce function that adds all the values (since $2x + 2y = 2(x + y)$). We present an interesting example of this case in section 3.4.2.

When a new transform is introduced, the commutative-rewrite algorithm works as follows: It starts with the initial placement of the transform and successively refines it by iteratively calling `commuteWith` function on the immediate downstream transform until the function returns `None` or we have pushed the transform all the way to do end.

$u \rightarrow$ $t \downarrow$	Shuffle	Reduce	Join
Filter (Key)	t	t	t
Filter (Value)	t	None	$t.f(res_1)$
Map (Key)	t if invertible or extra shuffle	t if invertible or extra shuffle	None
Map (Value)	t	t if $t.f$ distributes over $u.r$	$t.f(res_1)$

Table 3.2: Rewrite rules 2

It is left for future work to verify or construct inverses and distributive annotations. In addition to reduce, Spark also provides a "combine" transform that aggregates values on the machine executing the map transform before sending them to the reduce transform. Vega supports these combine transforms in a similar way to reduce transforms.

3.3.2.2 Δ Computation

For Δ computation, Vega follows a strategy similar to Naiad. A general transform t can be converted into a Δ transform Δt as follows: let m^p_- and m^p_+ be changes propagated from previous Δ transform in the workflow and let O^p be its previous output. Now generate the new input for the transform by integrating the changes: $I^t = (O^p \cup m^p_+) \setminus m^p_-$. Construct $\Delta t = (m^t_+, m^t_-)$ where $m^t_+ = (t(I^t) \setminus O^t)$, $m^t_- = (O^t \setminus t(I^t))$ and O^t is the previous output of t. This runs in the order of the input to t and is clearly inefficient. However, for many known transforms, the output changes can be computed efficiently.

For example: a map transform with function m can be defined to work on Δ data as follows: $\Delta m(+x) = +m(x)$, $\Delta m(-x) = -m(x)$. Please refer to [MMI13] for details on how transforms can be optimized to work on Δ data.

3.3.3 Changing Transforms

While we have discussed addition of transforms in this section, deletion and changing of existing transforms follows similar strategies. For Δ computation, it is just a matter of taking the "diff" of the workflow before and after the changes and propagating them. For commutative-rewrites on maps, deletion has the same effect of applying the inverse, while changing a map is simulated by applying the inverse of the old map, followed by the new map. Removing a filter or replacing it cannot be handled with commutative rewrites and we use Δ computation.

3.4 Evaluation

I evaluate Vega on 3 experiments based on existing workflows on large-scale data. We would like to understand the expressiveness of the library and its performance characteristics in each case.

The experiments were carried out on a cluster containing 16 machines with i7 processors, each running at 3.40GHz and equipped with 4 cores (2 hyper-threads per core), 32GB of RAM and 1TB of disk capacity. The operating system is 64bit Ubuntu version 12.04. The datasets were all stored in HDFS version 1.0.4 with a replication factor of one. Vega uses Spark 1.2.1 as the execution engine for running the workflows.

In each experiment, I run an initial workflow and then make a change to it and run the changed workflow incrementally. This is compared against running the changed workflow from scratch. Each experiment is run 3 times and the average time is reported.

I take real queries from existing work and either make incremental versions of them or add my own incremental queries on top of existing ones. Word count in section 3.4.1 is an existing workflow that I have made incremental by adding a filter, map, filter over map. They are inspired by real-world use cases but simplified for this example. All queries are run on existing data. All the queries in WikiReverse website [Fai15] are made incremental in

section 3.4.2. The incremental portion of query 3 is my addition on the existing query. All its queries are also run on existing data. Both initial and incremental queries in section 3.4.3 are from [LDY13] but they do not provide any data, so I just show that the optimizations work in this scenario.

3.4.1 Word Count

This experiment is based on workflow discussed in the motivating example. As mentioned before word count is widely used to test big data frameworks and it is representative of many data analytic workflows. We run the tests on two datasets. One based on [IPW11] (which we call "Word Bag") that is composed of 8000 words in Zipf distribution, and another taken from Wikipedia[2]. The results for both datasets are shown in table 3.3.

The initial workflow is:

```
File("...")
-> FlatMap(line => line.split(" "))
-> Map(word => (word, 1))
-> Reduce((count1, count2) => count1 + count2)}
```

That is, we split every line into words, associate the number 1 with every word, aggregate all the 1s corresponding to each word and sum them up.

In the first query, we add a filter to only include words that have at least 3 characters. The new workflow becomes:

```
File("...")
-> FlatMap(line => line.split(" "))
-> Filter(word => word.length > 2) //added
-> Map(word => (word, 1))
-> Reduce((count1, count2) => count1 + count2)
```

[2] https://www.wikipedia.org

In the second query, we start out with the original workflow of the previous test and add a map to add suffix to every word in the dataset. This could be useful to convert the output into comma separate values (CSV) for example. The new workflow becomes:

```
File("...")
-> FlatMap(line => line.split(" "))
-> Map(word => (word, 1))
-> MapKey(word => word + suffix) //added
-> Reduce((count1, count2) => count1 + count2)
```

Similar to the previous case, Vega pushes the map all the way to the end and computes the result.

In the third query, we start out with the previous workflow with the map and add the filter from the first test to get:

```
File("...")
-> FlatMap(line => line.split(" "))
-> Filter(word => word.length > 2) //added
-> Map(word => (word, 1))
-> MapKey(word => word + suffix)
-> Reduce((count1, count2) => count1 + count2)
```

In this case Vega has to also invert the second map, by removing its suffix, to push the filter towards the end.

From the table we can see that Vega usually computes incremental results an order to two magnitude faster than rerunning the computation again. The output of any word count workflow depends on the number of unique words present in the data. This results in two observations:

1. There aren't many unique new words added when we go from 50 GB to 100 GB in

	Filter		Map		Filter over Map	
	Regular	Incremental	Regular	Incremental	Regular	Incremental
50 GB Word Bag	69.3	0.5	76.7	0.6	76.4	0.3
100 GB Word Bag	59.3	0.9	67.5	0.7	70.7	0.5
50 GB Wikipedia	209.4	24.4	358.1	27.6	163.2	45.2
100 GB Wikipedia	981.2	26.2	1067.4	29.8	1045.4	47.5

Table 3.3: Results on word count workflow. Time in seconds.

either data sets. Hence, Vega performs incremental computations in roughly the same time even when the input size has doubled.

2. Since Word Bag only has 8000 unique words, its output is significantly smaller than Wikipedia set and hence Vega performs significantly better on Word Bag set.

This experiment reiterates an important feature of Vega— it computes incremental results in time proportional to the output of the initial workflow. Many big data workflows start out with huge quantities of input data but reduce it down to significantly smaller sizes after processing. Hence Vega performs especially well on such workflows.

3.4.1.1 Comparing Two Strategies

We use the first query to compare the performance of vanilla Δ computation and that with commutative-rewrites. In the first test, we run the filter in-place using Δs. The result on Word Bag dataset is plotted in figure 3.4. We notice that while the incremental computation performs well for small data, it slows down as data size increases, eventually taking as long as rerunning the changed computation from scratch. As discussed earlier, this is due to the space required to keep old outputs in cache along with the data required for incremental computation and time required to shuffle the Δs.

In the next test, we optimize it with commutative-rewrites first. As discussed in the example, Vega pushes the filter all the way to the end and computes the result. The results

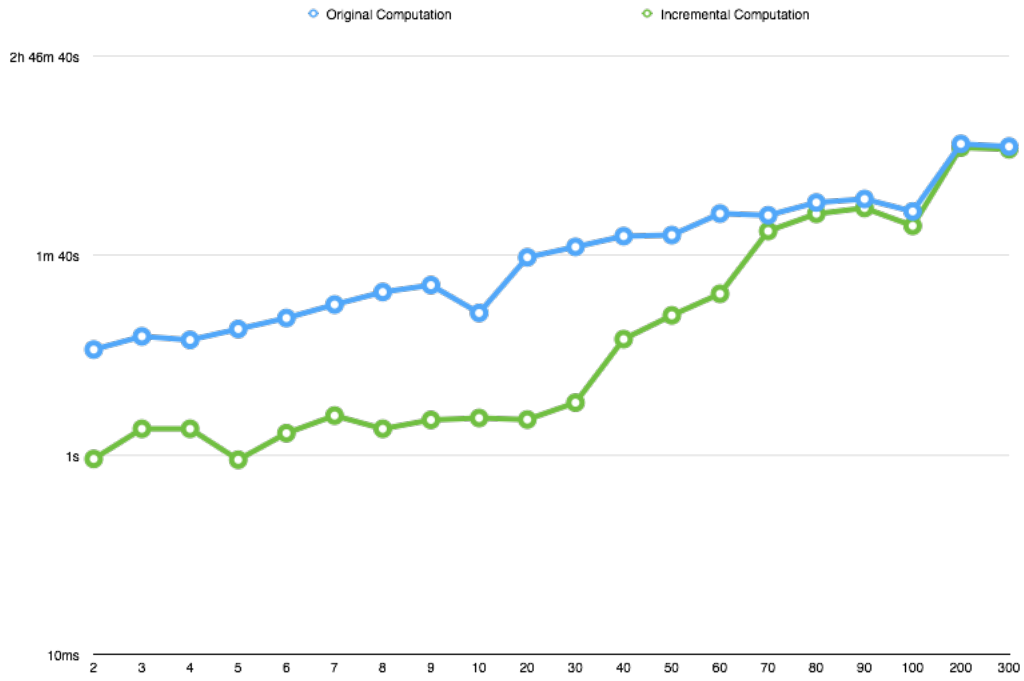

Figure 3.4: Δ Computation Results of query 1 (log-scale used for Y-axis)

on the same Word Bag dataset are plotted in figure 3.5. We notice that the incremental computation scales really well for all sizes of input data.

3.4.2 Wiki Reverse

The WikiReverse project [Fai15] aims to understand how people use Wikipedia on the web. It does so by computing a number of statistics such as the number of links incoming to `wikipedia.org` domain, the number of links to individual Wikipedia language, the number of popular websites linking to Wikipedia, and many others. The dataset used by WikiReverse is Common Crawl Dataset [3]. Common Crawl is a non-profit foundation that collects data from web pages using a crawler and publishes massive corpus of data. This experiment uses the same data and all the queries used by WikiReverse project (with query 3 below slightly augmented) and runs them incrementally.

Query 1: We start out with a scenario where an analyst want to compute how many

[3] `https://commoncrawl.org`

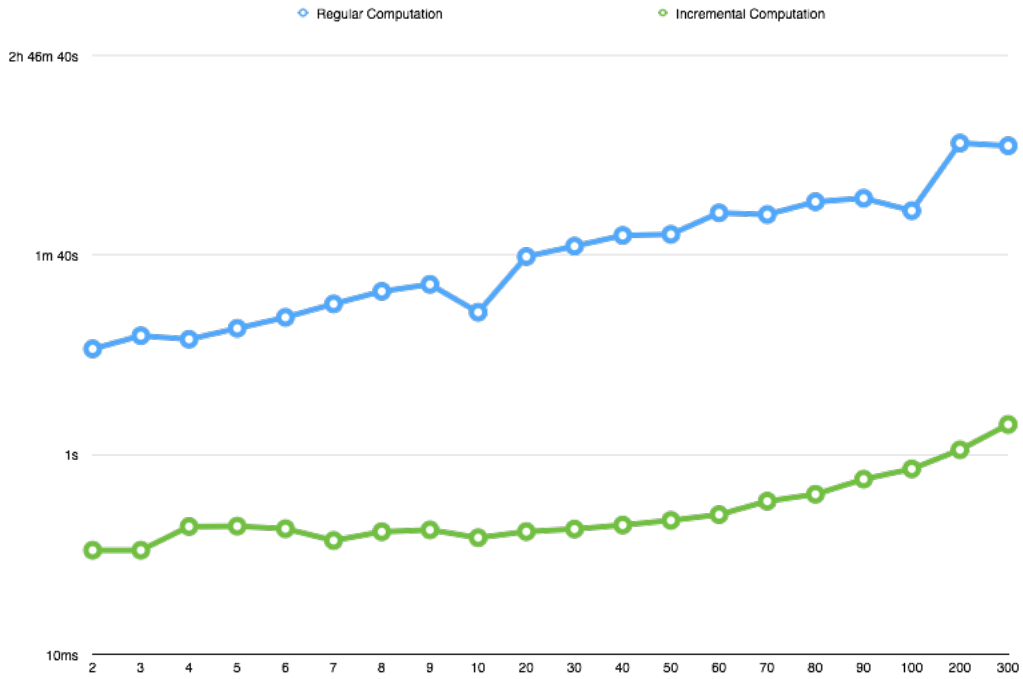

Figure 3.5: Rewritten Computation Results of query 1 (log-scale used for Y-axis)

links in the Common Crawl dataset point to Wikipedia.

```
File("...")
-> Map(extractLinks)
-> Map(link => (link, 1))
-> MapKey(getDomain)
-> MapKey(stringToURL)
-> Reduce((count1, count2) => count1 + count2)
```

The workflow first extracts links from each crawled page and associates each link with 1 in order to count them later. Domains are extracted from the links and then converted into Java URL objects. The final reduce groups domains together and counts the number of links pointing to them.

After running the workflow, the analyst realizes that she has made a slight error here — she forgot to filter the data to only include wikipedia links. This workflow is measuring the

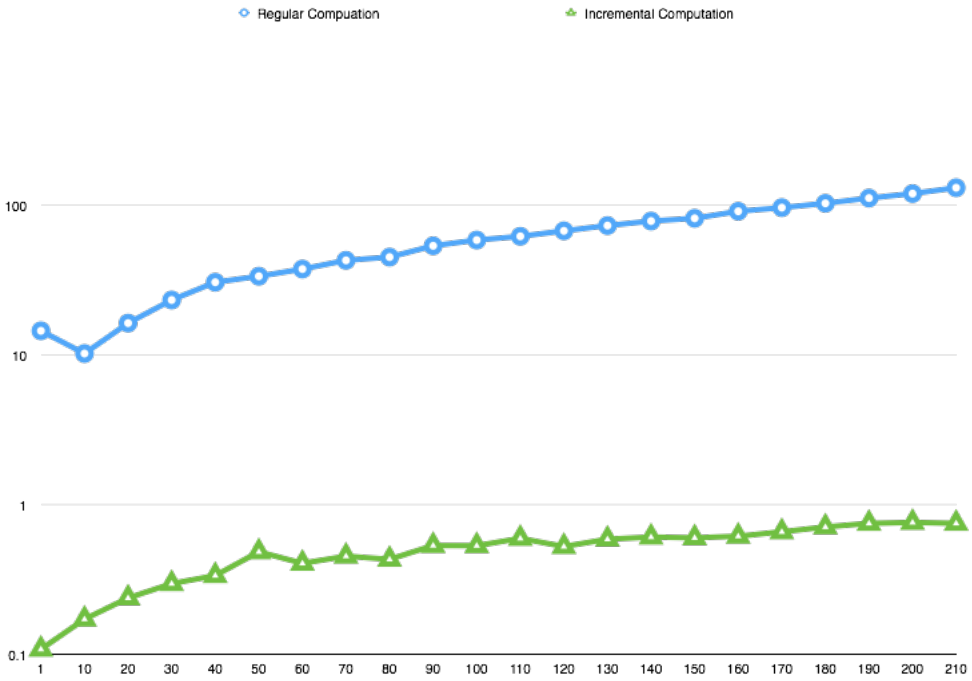

Figure 3.6: Query 1 Results (log-scale used for Y-axis)

total number of links present in the Common Crawl dataset instead of measuring just the Wikipedia links. These kinds of errors are very common in exploratory analysis.

The analyst fixes the error by adding a filter to only include Wikipedia domain. The new workflow is as follows:

```
File("...")
-> Map(extractLinks)
-> Map(link => (link, 1))
-> MapKey(getDomain)
-> FilterKey(domain => domain.contains("wikipedia.org")) //added
-> MapKey(stringToURL)
-> Reduce((count1, count2) => count1 + count2)
```

The response time for this query for various input data sizes is presented in figure 3.6. Vega is able push the filter to the end and compute the result quickly.

Query 2: Now let us assume that the analyst is only interested in English language articles. She could add another filter to the workflow as follows:

```
File("...")
-> Map(extractLinks)
-> Map(link => (link, 1))
-> MapKey(getDomain)
-> FilterKey(domain => domain.contains("wikipedia.org"))
-> FilterKey(domain => domain.contains("en.")) //added
-> MapKey(stringToURL)
-> Reduce((count1, count2) => count1 + count2)
```

This only includes domains like `en.wikipedia.org` or `en.m.wikipedia.org` — that point to English language articles. The results are plotted in figure 3.7. Vega pushes this filter to the end as well and computes the result. In this way, the analyst is able to compute various analytics such as the most popular languages in Wikipedia (English followed by Spanish) very quickly using incremental computation.

Query 3: In this test, an analyst would like to measure the incoming traffic generated by popular sites to Wikipedia. As a first step, she writes the following workflow to calculate the incoming links from popular sites:

```
File("...")
-> Map(page => extractPageURL(page), extractLinks(page))
-> FilterValue(link => link.contains("wikipedia.org"))
-> Map((page-url, links) => (page-url, links.length))
-> MapKey(getDomain)
-> MapKey(stringToURL)
-> Reduce((count1, count2) => count1 + count2)
-> Join(popularDomainsRDD)
```

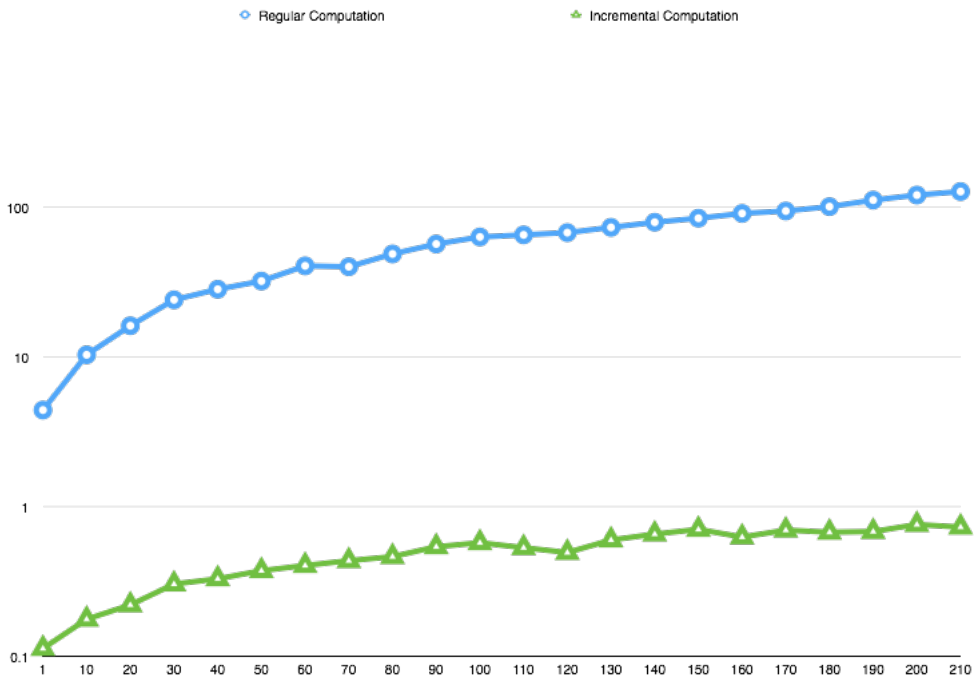

Figure 3.7: Query 2 Results (log-scale used for Y-axis)

This query assumes that every incoming link generates the same amount of traffic. However, in practice, different sites produce different kinds of traffic. Let us make a simplifying assumption sites with `.com` domains generate twice as much traffic as other domains on average. She now adds a weighted map to the workflow as follows:

```
File("...")
-> Map(page => (extractPageURL(page), extractLinks(page)))
-> FilterValue(link => link.contains("wikipedia.org"))
-> Map((page-url, links) => (page-url, links.length))
-> MapKey(getDomain)
-> Map((domain, link-count) => //added
    if (domain.contains(".com")) (domain, (link-count * 2))
    else (domain, link-count))
-> MapKey(stringToURL)
-> Reduce((count1, count2) => count1 + count2)
```

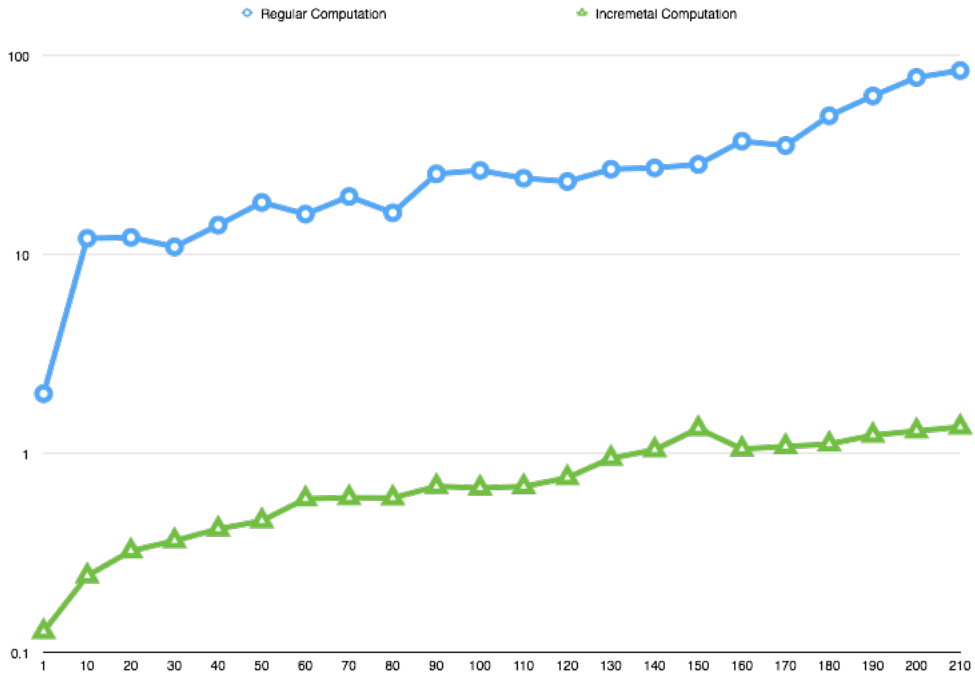

Figure 3.8: Query 3 Results (log-scale used for Y-axis)

```
-> Join(popularDomainsRDD)
```

Vega takes advantage of the distributive property of times over plus and is able to push past the reduce and eventually past join, hence reusing the entire output of the previous computation. We plot the results in figure 3.8.

Query 4: In this test, the analyst would like to understand which topics in Wikipedia are linked to the most. She starts out with this initial workflow:

```
File("...")
-> Map(page => extractLinks(page))
-> Map(link => (link, 1))
-> FilterKey(link => link.contains("wikipedia.org"))
//extractTopic returns the topic part of the URL
//e.g: returns 'United_States' from 'https://en.wikipedia.org/wiki/United_States'
-> MapKey(link => extractTopic(link))
-> Reduce((count1, count2) => count1 + count2)
```

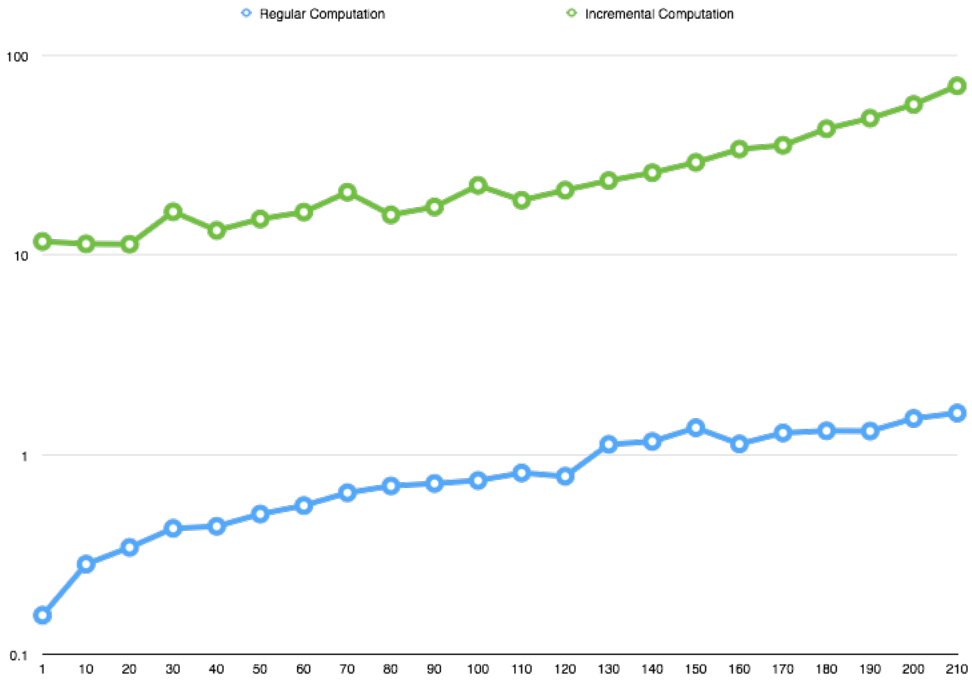

Figure 3.9: Query 4 Results (log-scale used for Y-axis)

While this gives the results that she wants, she realizes that the topics extracted from URLs have underscores separating them and not spaces. Since it is natural to use spaces to separate words (and since many word processing libraries assume space as the separator), she decides to update the workflow to convert underscores to spaces:

```
File("...")
-> Map(page => extractLinks(page))
-> Map(link => (link, 1))
-> FilterKey(link => link.contains("wikipedia.org"))
//extractTopic returns the topic part of the URL
//e.g: returns 'United_States' from 'https://en.wikipedia.org/wiki/United_States'
-> MapKey(link => extractTopic(link))
//change 'United_States' to 'United States'
-> MapKey(topic => topic.replace('_', ' ')) //added
-> Reduce((count1, count2) => count1 + count2)
```

Since this is an invertible map (the inverse just replaces spaces with underscores), Vega is able to push it past the reduce and compute the result quickly. The results are plotted in figure 3.9.

Discussion: Similar to the word count example, we notice that the output of the initial computation is not very large, even when the initial computation runs on large data. This makes Vega very effective on such workflows as it runs in time proportional to the output of the initial computation. This experiment also shows that Vega is applicable to larger workflows.

3.4.3 Telecom Workflow

This example is taken from [LDY13]. In this scenario, a telecom operator is planning to run some analytics on user-downloads data. The input is tuples of type `(user-id, device, bytes-downloaded)`. Our analyst is only interested in non-tethered data, so the initial workflow (adapted from figure 1 in [LDY13]) is:

```
input = File("...")
//total traffic per user
joinData =
  input
  -> Map((user, device, data) => (user, data))
  -> Reduce((data1, data2) => data1 + data2)

workflow =
  input
  -> Map((user, device, data) => (user, device))
  -> FilterValue(user => !device.contains("desktop"))
  -> Join(joinData)
  -> Map((user, device, total-data) => (user, total-data))
```

After running the workflow, the analyst realizes that the filter does not remove all the cases of tethering. She need to also remove devices that contain "badAgent" in them. So she refines the workflow to handle that case:

```
workflow =
  input
  -> Map((user, device, data) => (user, device))
  -> FilterValue(device => !device.contains("desktop"))
  -> FilterValue(device => !device.contains("badAgent")) //added
  -> Join(joinData)
  -> Map((user, (device, total-data)) => (user, total-data))
```

Vega is able to propagate the filter through the join and re-execute the computation quickly.

Similarly the analyst might want to refine a related workflow (also from figure 1 in [LDY13]) that computed downloads per device:

```
input = File("...")
//total traffic per device
joinData =
  input
  -> Map((user, device, data) => (device, data))
  -> Reduce((data1, data2) => data1 + data2)


workflow =
  input
  -> Map((user, device, data) => (device, user))
  -> FilterKey(device => !device.contains("desktop"))
  -> Join(joinData)
  -> Map((device, (user, total-data)) => (device, total-data))
```

The new workflow includes the additional filter as mentioned above. Vega also re-computes this query by pushing the filter to the end.

3.5 Discussion

Vega's main use-case is to take programs that can be decomposed into functional operators and provide interactive performance on small incremental code changes in them. As I have shown in section 3.4, Vega is applicable to several large-scale exploratory use-cases and works on structured, semi-structured and unstructured data. Vega's commutative and Δ rewrites are general enough to run on any execution engine that is at least as expressive as Hadoop (e.g., Spark, Dryad). Vega provides the same fault-tolerance guarantees as the underlying execution engine.

Vega critically depends on invertible map and reduce functions (for commutative-rewrites and Δ computation respectively). For transforms with no such structure, the commutative-rewrites can still be applied if there is lineage information that keeps track of the inputs to each transform. That is, to invert a map's output, we can simply consult the lineage to get its inputs.

Vega trades space for time by caching previous computations to avoid re-computation. The amount of memory required to perform incremental re-computation depends on how much farther downstream the introduced operator can be pushed. In the general case, Δ computation requires keeping the old outputs of all downstream operators in memory along with data required for incremental computation. This is usually significantly larger than the input data. The general trend in data centers has been to increase the RAM in machines and also to use SSDs for storage. SSDs provide significant speed over magnetic hard disks and hence can work well as Vega caches.

In the next chapter, we tackle another key challenge in cloud computation, namely correctness. For this I use a different analysis technique called enumerative model checking. In this analysis, I simulate all possible behaviors of a program and check each one to see

if there is any error. While this can get prohibitive for general programs, I develop several domain-specific optimizations in order to make it tractable for Software Defined Networks, the networking layer of choice for many cloud applications.

CHAPTER 4

Kuai

4.1 Introduction

Software-defined networking (SDN) is a novel networking architecture in which a centralized software controller dynamically updates the packet processing policies in network switches based on observing the flow of packets in the network [JKM13, FRZ13]. SDNs have been used to implement sophisticated packet processing policies in networks, and there is increasing industrial adoption [MAB08, JKM13]. Since cloud applications are inherently network based, it is critical for the correctness of the applications to have a bug-free networking layer.

This chapter considers the problem of verifying that an SDN satisfies a network-wide safety property. Since the controller code in an SDN can dynamically change how packets flow in the network, a bug in the controller code can lead to hard-to-analyze network errors at run time. Many cloud application frameworks have a similar architecture to SDNs where there is a single logical master and many workers. The master usually has complicated (and often changing) logic for scheduling tasks and responding to events. Workers usually have the same (relatively simple) program to run commands given by the master. Hence the techniques for verifying SDNs could potentially be useful for other layers in the software stack as well.

I describe the design of Kuai, a distributed enumerative model checker for SDNs. The input to Kuai is a model of an SDN consisting of two parts. The first part is the controller, written in a simplified guarded-command language similar to Murphi [Dil96]. The second part is the description of a network, consisting of a fixed finite set of switches, a fixed set of

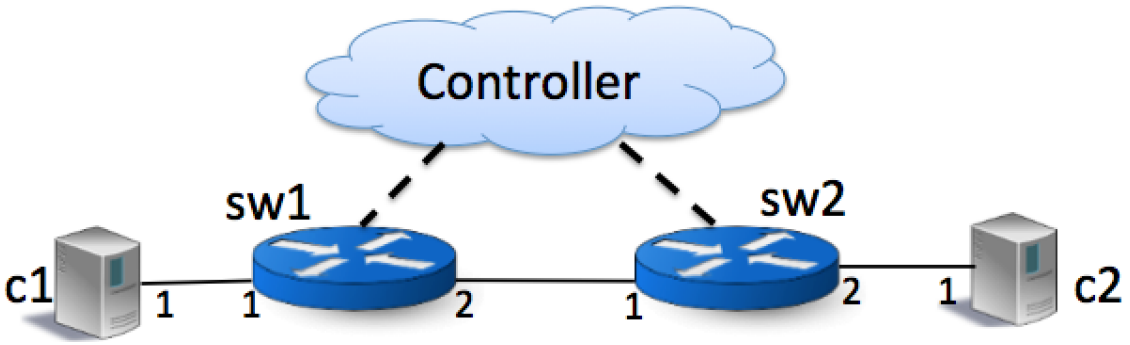

Figure 4.1: SSH Example

```
1  def pktIn(pkt)
2    (sw,pt) = pkt.loc
3    if pkt.prot = SSH:
4      drop(pkt)
5    else:
6      dest = 2 if pt = 1 else 1
7      fwd(pkt, [|dest|], sw)
8    rule r1 = (5,{prot=SSH},[||])
9    rule r2 = (1,{port=1},[|2|])
10   rule r3 = (1,{port=2},[|1|])
11   message cm1 = add(r1)
12   message cm2 = add(r2)
13   message cm3 = add(r3)
14   for sw in [sw1, sw2]:
15     send_message(cm1, sw)
16     send_message(cm2, sw)
17     send_message(cm3, sw)
```

Listing 4.1: Controller for SSH

client nodes, and the topology of the network (i.e., the connections between the ports of the clients and the switches). Given a safety property of the network, Kuai explores the state space of the SDN to check if the property holds on all executions.

4.2 Example

Figure 4.1 shows a simple SDN. It consists of two switches sw_1 and sw_2 connected to two clients c_1 and c_2. Each client has a port and each switch has two ports to send and receive packets, and the figure shows how the ports are connected to each other. Each connection

between ports represents a bi-directional communication channel that may reorder packets. Moreover, the switches are connected to a controller through dedicated links. Packets are routed in the network using *flow tables* in switches. A flow table is a collection of prioritized forwarding *rules*. A rule consists of a priority, a pattern on packet headers, and a list of ports. A switch processes an incoming packet based on its flow table. It looks at the highest priority rule whose pattern matches the packet and forwards the packet to the list of ports specified in the rule, and drops the packet if the list of ports in the rule is empty. In case no rule matches a packet, the switch forwards the packet to the controller using a request queue and waits for a reply from the controller on a forward queue. The controller replies with a list of ports to which the packet should be forwarded, and optionally sends *control messages* to the control queue of one or more switches to update their flow tables. A control message can add or delete a rule in a switch.

By specifying the rules to be added or deleted, a controller can dynamically control the behaviors of all switches in an SDN network. For example, suppose we want to implement the policy that all SSH packets are dropped. The controller can update the switches with a rule that states that no SSH packets are forwarded, and another that states all non-SSH packets are forwarded. List 4.1 shows a possible controller that implements this policy.

The controller's `pktIn` function gets called when a switch encounters a packet for which no rules match. We present the implementation in Python-like syntax. The controller first gets the switch and port corresponding to the packet (line 1). It then checks to see if the packet's protocol is SSH. If so, it instructs the switch to drop the packet (line 4), otherwise it instructs the switch to forward it through the port other than the one it received the packet (lines 6 and 7). It then creates rules corresponding to the actions described above so the switches don't have to consult the controller for SSH packets again. Each rule is a tuple consisting of an integer priority, match rule (dictionary of type field and value to match (in line 8 `prot=SSH` corresponds to protocol field and SSH value)) and a list of ports to forward the matched packets. If no ports are given, then the packet is dropped (e.g. line 8). The controller specifies that these rules should be added (lines 11, 12, 13). Alternatively it can

also instruct switches to remove rules. Finally it instructs the switches to install the rules (lines 15, 16, 17).

Essentially, the controller adds three rules on the switches: $r1$ to drop SSH packets, $r2$ to forward packets from port 1 to port 2, and $r3$ to forward packets from port 2 to port 1. Since dropping SSH packets (rule $r1$) has higher priority, it will match SSH packets, and rules $r2$ and $r3$ will only match (and forward) non-SSH packets. The controller has a subtle bug. It turns out that a switch can implement rules in arbitrary order. Thus, the switches may end up adding rules $r2$ and $r3$ before adding $r1$, thus violating the policy. Kuai confirms the bug A possible fix in this case is to implement a *barrier* after line 15, to ensure that rule $r1$ is added before the other rules. Kuai confirms the policy holds in the fixed version.

The verification of SDNs is challenging due to several reasons. First, even when the topology is fixed with a finite set of clients and switches, the state space is still unbounded, as clients may generate unboundedly many packets and these packets could be simultaneously progressing through the network. For example, client c_1 may send a packet to sw_1 at any point, and an unbounded number of packets can be in the network before sw_1 processes them. Similarly, there may be an unbounded number of control messages (i.e., messages sent from the controller to a switch) between the controller and the switches. While there may be a physical limit on the number of packets and control messages imposed by packet buffers in the switches, the sizes of these buffers can be large (of the order of megabytes) and precise modeling of buffers will blow up the state space.

Second, the packets may be processed in arbitrary interleaved orders, and the processing of one packet may influence the processing of subsequent ones because the controller may update flow tables based on the first packet. Similarly, control messages between the controller and the switches may be processed in arbitrary order and this may lead to potential bugs, including the bug pointed to above.

4.3 Approach

Kuai handles these challenges in the following way. First, instead of modeling unbounded multisets for packet queues, Kuai implements a *counter abstraction* where it tracks, for each possible packet, whether zero or arbitrarily many instances of the packet are waiting in a multiset. This abstraction enables us to apply finite-state enumerative model checking approaches.

A finite-state enumerative model checker starts with the initial state of the program and constructs a graph of all reachable states that arise from it. Every time a node is added to the reachability graph, the checker checks to see the state of the program corresponding to that node satisfies the desired property. If any state does not satisfy the property, then we have found a bug. Otherwise, the program satisfies the property.

In addition, Kuai includes a set of partial-order reduction techniques that are specific to the SDN domain (discussed in section 4.5). I empirically demonstrate that our set of partial order reduction techniques significantly reduces the state spaces of SDN benchmarks, often by many orders of magnitude. For the simple SSH example, the number of explored states is approximately 2 million without partial order reductions, but only 13 with reductions!

4.4 Formally Modeling Software-defined Networks

Preliminaries. A *multiset* m over a set Σ is a function $\Sigma \to \mathbb{N}$ with finite support (i.e., $m(\sigma) \neq 0$ for finitely many $\sigma \in \Sigma$). By $\mathbb{M}[\Sigma]$ we denote the set of all multisets over Σ. We shall write $m = [\![\sigma_1^2, \sigma_3]\!]$ for the multiset $m \in \mathbb{M}[\{\sigma_1,\sigma_2,\sigma_3\}]$ with $m(\sigma_1) = 2, m(\sigma_2) = 0$, and $m(\sigma_3) = 1$. We write $\emptyset$ for an empty multiset, mapping each $\sigma \in \Sigma$ to 0. We write $\{\}$ for an empty set. Two multisets are ordered by $m_1 \leq m_2$ if for all $\sigma \in \Sigma$, we have $m_1(\sigma) \leq m_2(\sigma)$. Let $m_1 \oplus m_2$ (resp. $m_1 \ominus m_2$) be the multiset that maps every element $\sigma \in \Sigma$ to $m_1(\sigma) + m_2(\sigma)$ (resp. $\max\{0, m_1(\sigma) - m_2(\sigma)\}$).

Given a set of states, a *(guarded) action* α is a pair (g, c) where g is a *guard* that evaluates

the states to a boolean and c is a *command*. A action α is *enabled* in a state s if the guard of α evaluates s to true. If α is enabled in s, the command of α can execute and lead to a new state s', denoted by $s \xrightarrow{\alpha} s'$. We write $\alpha(s) = s'$ if $s \xrightarrow{\alpha} s'$. A *transition system* TS is a tuple $(S, A, \rightarrow, s_0, AP, L)$ where S is a set of states, A is a set of actions, $\rightarrow \subseteq S \times A \times S$ is a transition relation, $s_0 \in S$ is the initial state, AP is a set of atomic propositions, and $L : S \rightarrow 2^{AP}$ is a labeling function. We write $\rightarrow^*$ for the reflexive transitive closure of $\rightarrow$. A state s' is *reachable* from s if $s \rightarrow^* s'$. We write $s \rightarrow^+ s'$ if there is a state t such that $s \rightarrow t \rightarrow^* s'$. For a state s, let $A(s)$ be the set of actions enabled in s; we assume $A(s) \neq \emptyset$ for each $s \in S$. The *trace* of an infinite execution $\rho = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$ is defined as $trace(\rho) = L(s)L(s_1)\ldots$. The trace of a finite execution $\rho = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$ is defined as $trace(\rho) = L(s)L(s_1)\ldots L(s_n)$. An execution is *initial* if it starts in s_0. Let $Traces(TS)$ be the set of traces of initial executions in TS. We define invariants and invariant satisfaction in the usual way.

Syntax of Software-defined Networks We model an SDN as a network consisting of *nodes*, *connections*, and a *controller* program. Nodes come from a finite set *Clients* of *clients* and a (disjoint) finite set *Switches* of *switches*. Each node n has a finite set of *ports* $Port(n) \subseteq \mathbb{N}$ which are connected to ports of other nodes. A *location* (n, pt) is a pair of a node and a port $pt \in Port(n)$. Let *Loc* be the set of locations. A connection is a pair of locations. A network is well-formed if there is a bijective function $\lambda : Loc \rightarrow Loc$, called the *topology function*, such that $\{((n, pt), \lambda(n, pt)) \mid (n, pt) \in Loc\}$ is the set of connections and no two clients are connected directly.

We model a *packet* *pkt* in the network as a tuple $(a_1, \ldots, a_k, loc)$, where $(a_1, \ldots, a_k) \in \{0, 1\}^k$ models an abstraction of the packet data and $loc \in Loc$ indicates the location of *pkt*. Let *Packet* be the set of all packets.

Each switch contains a set of rules that determine how packets are forwarded. A *rule* is a tuple $(priority, pattern, ports)$, where $priority \in \mathbb{N}$ determines the priority of the rule, *pattern* is a proposition over *Packet*, and *ports* is a multiset of ports. We write *Rule* to denote the set of all rules. Intuitively, a packet matches a rule if it satisfies *pattern*. A

```
type client {
  Port : set of nat
  pq : multiset of packets
}
rule "send(c, pkt)"
  true ==> send(c, pkt)
end
rule "recv(c,pkt,pkts)"
  exist(pkt:c.pq, true) ==> recv(c,pkt,pkts)
end
```

Listing 4.2: Client

switch forwards a packet along *ports* for the highest priority rule that matches.

Rules are added or deleted on a switch by the controller through a set of *control messages* $CM = \{\mathsf{add}(r), \mathsf{del}(r) \mid r \in Rule\}$. Additionally, the controller uses a *barrier* message b to synchronize.

A client $c \in Clients$ is modeled as in List 4.2. It consists of a finite set *Port* of ports and a *packet queue* $pq \in \mathbb{M}[Packet]$ containing a multiset of packets which have arrived at the client. We use (guarded) actions to model behaviors of clients. An action is written as "**rule** *name guard* $\Longrightarrow$ *command* **end**." Predicate $exist(i : X, \varphi)$ asserts that there is an element i in the set (or multiset) X such that the predicate φ holds. Additionally, if $exist(i : X, \varphi)$ holds, then the variable i is bound to an element of X that satisfies φ and can be used later in the command part. In each step, a client c can (1) send a non-deterministically chosen packet *pkt* along some ports (rule send), or (2) receive a packet *pkt* from its packet queue and (optionally) send a multiset of packets *pkts* on some ports (rule recv).

A switch *sw* is modeled as in List 4.3. It consists of a set of ports, a *flow table* $ft \subseteq Rule$, a packet queue *pq* containing packets arriving from neighboring nodes, a *control queue cq* containing control messages or barriers from the controller, a *forward queue fq* consisting of at most one pair $(pkt, ports)$ through which the controller tells the switch to forward packet *pkt* along the ports *ports*, and a boolean variable *wait*. Predicate $noBarrier(sw)$ asserts $sw.cq$ does not contain a barrier. Predicate $bestmatch(sw, r, pkt)$ asserts that r is the highest priority rule whose pattern matches the packet *pkt* in switch *sw*'s flow table.

```
type switch {
  Port : set of nat
  ft : set of rules
  pq : multiset of packets
  cq : list of barriers and
       multisets of control messages
  fq : set of forward messages
  wait : boolean
}
rule "match(sw,pkt,r)"
  !sw.wait & noBarrier(sw) &
  exist(pkt:sw.pq,
    exist(r:sw.ft, bestmatch(sw,r,pkt))) ==>
  match(sw,pkt,r)
end
rule "nomatch(sw,pkt)"
  !sw.wait & noBarrier(sw) & !RqFull(controller) &
  exist(pkt:sw.pq,
    !exist(r:sw.ft,bestmatch(sw,r,pkt))) ==>
  nomatch(sw,pkt)
end
rule "add(sw,r)"
  !sw.wait & noBarrier(sw) &
  exist(add(r):sw.cq[0],true) ==>
  add(sw,r)
end
rule "delete(sw,r)"
  !sw.wait & noBarrier(sw) &
  exist(del(r):sw.cq[0],true) ==>
  delete(sw,r)
end
rule "fwd(sw,pkt,pts)"
  sw.wait & noBarrier(sw) &
  exist((pkt,pts):fq, true) ==>
  fwd(sw,pkt,pts)
end
rule "barrier(sw)"
  !noBarrier(sw) ==>
  barrier(sw)
end
```

Listing 4.3: Switch

Intuitively, a switch has a normal mode and a waiting mode determined by the *wait* variable. When the switch is in the normal mode, as long as there is no barrier in its control queue, it can either attempt to forward a packet from its packet queue based on its flow table,

```
type controller {
  CS : set of control states
  cs0 : CS
  cs : CS
  rq : set of packets
  κ : ℕ+
  pktIn : function
}
rule "ctrl(pkt,cs)"
  exist(pkt:controller.rq, true) ==>
  ctrl(pkt,controller.cs)
end
```

Listing 4.4: Controller

or update its flow table according to a control message in its control queue. When the switch cannot find a matching rule in its flow table for a packet, it can initiate a request to the controller, change to the waiting mode, and wait for a forward message from the controller telling it how to forward the packet. Once it receives a forward message (pkt, pts) and there is no barrier in the control queue, it forwards the pending packet pkt to the ports in pts, and changes back to the normal mode. If the control queue contains one or more barriers, the switch dequeues all control messages up to the first barrier from its control queue and updates its flow table.

A controller *controller* is modeled as in List 4.4. It is a tuple $(CS, cs_0, cs, rq, \kappa, pktIn)$ where CS is a finite set of *control states*, $cs_0 \in CS$ is the *initial control state*, cs is the *current control state*, rq is a finite *request queue* of size $\kappa \geq 1$ consisting of packets forwarded to the controller from switches, and $pktIn$ is a function that takes a packet pkt and a control state cs_1, and returns a tuple $(\eta, (pkt, pts), cs_2)$ where η is a function from *Switches* to $(\mathbb{M}[CM] \cup \{b\})^*$, (pkt, pts) is a forward message, and cs_2 is a control state. Intuitively, in each step, the controller removes a packet pkt from rq and executes $pktIn(pkt, controller.cs)$. Based on the result $(\eta, (pkt, pts), cs')$, it sends back to the source of the packet the forward message (pkt, pts) that specifies pkt should be forwarded along pts, and goes to a new control state cs'. Further, for each switch sw in the network it appends $\eta(sw)$ to sw's control queue.

Semantics of Software-defined Networks The semantics of an SDN is given as a

transition system. Let $\mathcal{N} = (\mathit{Clients}, \mathit{Switches}, \lambda, \mathit{Packet}, \mathit{Rule}, \mathit{controller})$ be an SDN, where each component is as defined above.

A *state* s of the SDN $\mathcal{N}$ is a quadruple (π, δ, cs, rq), where π is a function mapping each client $c \in \mathit{Clients}$ to its packet queue pq and δ is a function mapping each switch $sw \in \mathit{Switches}$ to a tuple $(pq, cq, fq, ft, wait)$ consisting of its packet queue, control queue, forward queue, flow table, and the wait variable.

The semantics of an SDN $\mathcal{N}$ is given by a transition system $TS(\mathcal{N}) = (S, A, \rightarrow, s_0, AP, L)$. Here, S is the set of states, $s_0 = (\pi_0, \delta_0, cs_0, \{\})$ is the initial state, and $A = \mathit{Send} \cup \mathit{Recv} \cup \mathit{Match} \cup \mathit{NoMatch} \cup \mathit{Add} \cup \mathit{Del} \cup \mathit{Forward} \cup \mathit{Barrier} \cup \mathit{Ctrl}$. The actions are informally defined as follows:

1. $send(c, pkt)$ enqueues pkt in the packet queue of switch connected to the destination port in the packet.

2. $recv(c, pkt, pkts)$ dequeues pkt from the packet queue for the client.

3. $match(sw, pkt, r)$ enqueues pkt in the packet queue of switch or client connected to the port identified by the rule.

4. $nomatch(sw, pkt)$ sets the wait variable and enqueues the packet to the controller's packet queue.

5. $add(sw, r)$ adds rule r to the flow table of switch sw.

6. $del(sw, r)$ removes rule r from the flow table of switch sw.

7. $fwd(sw, pkt, pts)$ enqueues pkt in the packet queue of the client or switch connected to the destination port in the packet. (The port set by the controller.) Finally, unsets the wait variable.

8. $barrier(sw)$ dequeues rules in the control queue of switch sw and processes them until one barrier is removed from the control queue.

9. *ctrl*(pkt, cs) calls the *pktIn* function of the controller *cs* with using packet *pkt* as argument. The controller can send messages to any switch in the topology to add or remove rules. Eventually it should send *fwd* message to the switch from which the packet has originated.

An atomic proposition $p \in AP$ is an assertion over packet fields or over control states. Define an SDN specification as a safety property $\Box\phi$ where ϕ is a formula over AP and $\Box$ is the "globally" operator of linear-temporal logic. The *model checking problem for an SDN* asks, given an SDN $\mathcal{N}$ and an SDN specification $\Box\phi$, if $TS(\mathcal{N})$ satisfies $\Box\phi$. For example, blocking SSH packets can be specified as $\Box \bigwedge_{pkt \in Packet}(pkt.loc.n \in Clients \wedge pkt.src \in Clients \wedge pkt.loc.n \neq pkt.src \Rightarrow pkt.prot \neq \text{SSH})$.

4.5 Optimizations

I now describe partial-order reduction and abstraction techniques that reduce the state space. These techniques use the structure of SDNs and, as I demonstrate empirically, are crucial in making the model checking scale to non-trivial examples. The formalization of the optimizations including their correctness theorems are stated in [MTW14]; the proofs are in [MTW].

Partial Order Reduction Let $TS = (S, A, \rightarrow, s_0, AP, L)$ be an action-deterministic transition system, i.e., $s \xrightarrow{\alpha} s'$ and $s \xrightarrow{\alpha} s''$ implies $s' = s''$. Given two actions $\alpha, \beta \in A$ with $\alpha \neq \beta$, α and β are *independent* if for any $s \in S$ with $\alpha, \beta \in A(s)$, $\beta \in A(\alpha(s))$, $\alpha \in A(\beta(s)))$, and $\alpha(\beta(s)) = \beta(\alpha(s))$. The actions α and β are dependent if α and β are not independent. An action $\alpha \in A$ is a *stutter action* if for each transition $s \xrightarrow{\alpha} s'$ in TS, we have $L(s) = L(s')$.

For $i \in \{1, 2\}$, let $TS_i = (S_i, A_i, \rightarrow_i, s_0^i, AP, L_i)$ be transition systems. Infinite executions ρ_1 of TS_1 and ρ_2 of TS_2 are *stutter-equivalent*, denoted $\rho_1 \triangleq \rho_2$, if there is an infinite sequence $A_0A_1A_2\ldots$ with $A_i \subseteq AP$, and natural numbers $n_0, n_1, n_2, \ldots, m_0, m_1, m_2, \ldots \geq 1$ such that

$$trace(\rho_1) = \underbrace{A_0 \ldots A_0}_{n_0 \text{ times}} \underbrace{A_1 \ldots A_1}_{n_1 \text{ times}} \underbrace{A_2 \ldots A_2}_{n_2 \text{ times}} \ldots$$

$$trace(\rho_2) = \underbrace{A_0 \ldots A_0}_{m_0 \text{ times}} \underbrace{A_1 \ldots A_1}_{m_1 \text{ times}} \underbrace{A_2 \ldots A_2}_{m_2 \text{ times}} \ldots$$

TS_1 and TS_2 are *stutter equivalent*, denoted $TS_1 \triangleq TS_2$, if $TS_1 \trianglelefteq TS_2$ and $TS_2 \trianglelefteq TS_1$, where $\trianglelefteq$ is defined by: $TS_1 \trianglelefteq TS_2$ iff for all $\rho_1 \in Traces(TS_1)$. $\exists \rho_2 \in Traces(TS_2)$. $\rho_1 \triangleq \rho_2$.

Let $\Box\phi$ an arbitrary safety property. If $TS_1 \triangleq TS_2$ then TS_1 satisfies $\Box\phi$ if and only if TS_2 satisfies $\Box\phi$. If $TS_1 \trianglelefteq TS_2$ then TS_2 satisfies $\Box\phi$ implies TS_1 satisfies $\Box\phi$.

4.5.1 Barrier Optimization

This is based on an observation that when a barrier is present in the control queue, the only action the switch can fire is the barrier action (all other rules are guarded by no barrier predicate). Hence we can always flush out control queues of switches until there are no barriers in them. This implies that after a control action is executed, one can immediately update flow tables of switches whose control queue has barriers added by the controller. Hence a control action and successive barrier actions can be merged. I present details about how we can prove this below. These details are elided for the rest of the optimizations.

Its correctness is shown by viewing it as an instance of partial order reduction.

For an SDN $\mathcal{N}$, note that $TS(\mathcal{N})$ is not action-deterministic due to barrier actions. With different fetching orders, $barrier(sw)$ may lead to multiple states. Define $b(s, sw)$ as the number of transitions of the form $s \xrightarrow{barrier(sw)} s'$. Note that a barrier action from any s leads to at most $2^{|Rule|}$ states. Hence for each transition $s \xrightarrow{barrier(sw)} s_i$ where $1 \leq i \leq b(s, sw)$, we can append the action with the index i, i.e., $s \xrightarrow{barrier(sw)_i} s_i$. In the following, we redefine the set $Barrier = \{barrier(sw)_i \mid sw \in Switches \wedge 1 \leq i \leq 2^{|Rule|}\}$, and assume that $TS(\mathcal{N})$ is action-deterministic by renaming barrier actions.

A switch sw has a barrier iff there is a barrier in sw's control queue. A state s has a barrier, denoted $hasb(s)$, iff some switch $sw \in Switches$ has a barrier in s. Define the *ample set* for every state s in $TS(\mathcal{N})$ as follows: if s has a barrier, then $ample(s) = \{barrier(sw)_i \mid 1 \leq i \leq b(s, sw) \wedge sw$ has a barrier in $s\}$, that is, all barrier actions enabled in s. If s does not have a barrier, then $ample(s) = A(s)$.

Given $TS(\mathcal{N})$, we now define a transition system $\widehat{TS} = (\hat{S}, A, \Rightarrow, s_0, AP, L)$ where $\hat{S} = S$ is the set of states, and the transition relation $\Rightarrow$ is defined as: if $s \xrightarrow{\alpha} s'$ and $\alpha \in \mathit{ample}(s)$, then $s \overset{\alpha}{\Rightarrow} s'$.

We claim that $TS(\mathcal{N}) \triangleq \widehat{TS}$.

Intuitively, it holds because any barrier action is independent of other actions and is a stutter action. Hence for an infinite execution $s \xrightarrow{\alpha_1} s_1 \ldots \xrightarrow{\alpha_n} s_n \xrightarrow{\mathit{barrier}(sw)} t$ in $TS(\mathcal{N})$ where s has a barrier and α_i is not a barrier action for all $1 \leq i \leq n$, we can permute $\mathit{barrier}(sw)$ forward until s and obtain a stutter-equivalent execution in $\widehat{TS}$.

Hence, we can merge a control action and successive barrier actions into a single transition. This optimization does not introduce any false positives.

4.5.2 Client Optimization

Given transition system $TS_2 = (S_2, A_2, \rightarrow_2, s_0, AP_2, L_2)$, we further reduce the state space by observing that any receive action of a client is a stutter action and is independent of other actions. Formally, we define $\mathit{ample}(s)$ for each state $s \in S_2$ as follows: if there is a client in s such that its packet queue is not empty, then $\mathit{ample}(s) = \{\mathit{recv}(c, \mathit{pkt}, \mathit{pkts}) \mid \mathit{pkt}$ is in $c.pq$ at $s\}$, that is, all receive actions enabled in s. Otherwise, $\mathit{ample}(s) = A(s)$. We now define a transition system $TS_3 = (S_3, A_3, \rightarrow_3, s_0, AP_3, L_3)$ where $S_3 = S_2$, $A_3 = A_2$, $AP_3 = AP_2$, $L_3 = L_2$, and where the transition relation $\rightarrow_3$ is defined as: if $s \xrightarrow{\alpha}_2 s'$ and $\alpha \in \mathit{ample}(s)$, then $s \xrightarrow{\alpha}_3 s'$.

The intuition here is that the invariants we want to check are on controller state and packet queues and not on the clients themselves. Hence, the internal actions of the clients are independent of the system. In addition, since the queues already simulate arbitrary packet delays, we can fire all the client receive rules at once instead of simulating all actions of the system for every individual client receive. This optimization does not introduce any false positives.

4.5.3 $(0, \infty)$ Abstraction

The $(0, \infty)$ abstraction bounds the size of packet queues and the multiset in each control queue. The idea is as follows. One can regard a multiset as a counter that counts the number of elements in it exactly. Instead, $(0, \infty)$ abstraction abstracts a multiset so that for each element e, it either does not contain e (i.e. 0) or contains unboundedly many copies of e (i.e. ∞). Then the size of an abstracted multiset is bounded. Note that for any state in the system, any switch's control queue contains exactly one multiset. Hence, the abstraction bounds the length of control queues.

This gives us an abstract state transition system that has significantly fewer states than the concrete SDN system. The abstraction is sound in that if any safety property is proven in the abstract system, then it can be proven in the concrete system. However, this is an overabstraction and can introduce false positives: a bug present in the abstract transition system may not be present in the concrete system.

4.5.4 All Packets in One Shot Abstraction

So far, a switch processes a single packet at a time. We can further reduce the reachable state space by forcing a switch to process all packets matched by some rule at a time. The intermediate states produced by successive match actions in a switch are removed. Intuitively, all the matched packets are just forwarded to their output ports. Since the output ports already simulate arbitrary delays, there are no extra behaviors to be lost by pushing all the matched rules in one go instead of pushing them one at a time. This is a sound overabstraction, i.e. any safety property satisfied by the optimized transition system will be satisfied by the original system. However bugs found in the optimized system may not be present in the original system.

4.5.5 Controller Optimization

We consider a restricted class of SDNs in which the size κ of the controller's request queue is one. Under this restriction, we can define a new transition system that is stutter equivalent to the previous system and has fewer reachable states. The idea is to observe that since the controller queue has length 1, only one switch can talk to the controller at any time and the other switches will have to wait until the controller is done processing. Hence the non-determinism we need to simulate is the order in which the controller processes the waiting switches. This way we can merge the no-match and control actions. This abstraction is restricted to certain classes of controllers and does not introduce any false positives.

4.6 Implementation and Evaluation

Kuai[1] is implemented on top of PReach [BBP10], a distributed enumerative model checker built on Murphi. This allows Kuai to distribute the model checking over a number of nodes in a cluster and enables Kuai to scale to large state spaces. We model switches, clients, and the controller as concurrent Murphi processes which communicate using message passing, with the queues modeled as multisets. I manually abstract IP packets using predicates used in the controller. I implement $(0, \infty)$-counter abstraction as a library on top of Murphi multisets.

Kuai takes as input topology information such as the number of switches, clients, and their connections, (manually) abstracted packets, and the controller code written as a Murphi process, and invariants written in Murphi syntax. We found it fairly straightforward to port POX [POX] controllers due to the imperative features of Murphi. Murphi allows arbitrary first order logic formulas as invariants and it is easy to specify safety properties. Kuai compiles them into a single Murphi file and the model checking effort is then distributed across several machines using PReach. Finally the output of the tool is an error trace if the program invariant fails, or *success* otherwise.

I have evaluated Kuai on a number of real world OpenFlow benchmarks. The experiments

[1]The tool can be downloaded at https://github.com/t-saideep/kuai

Program	Bytes/ state	w/o optimizations		w/ optimizations	
		States	Time	States	Time
SSH 2×2	304	2,283,527	23.52s	13	6.40s
ML 3×3	320	9,109,456	89.99s	5308	6.39s
ML 6×3	748			23,926,202	604.07s
ML 9×2	1276			18,615,767	793.84s
FW(S) 1×2	332	2,110,986	26.89s	3645	5.45s
FW(M) 2×4	448			45,507	8.03s
FW(M) 3×4	560			512,439	55.06s
FW(M) 4×4	676			5,360,871	475.54s
RS 4×4	764			4998	6.60s
RS 4×5	764			590,570	82.82s
RS 4×6	764			5,112,013	327.39s
SIM 5×6	632			167	6.23s
SIM 5×8	632			167	6.34s
SIM 5×12	1108			167	6.85s

Table 4.1: Kuai Experimental results

were performed on a cluster of 5 Dell R910 rack servers each with 4 Intel Xeon X7550 2GHz processors, 64 x 16GB Quad Rank RDIMMs memory and 174GB storage. Our experiments had access to a total of 150 cores and had access to 4TB of RAM.

Table 4.1 shows a summary of experimental results and compares against model checking without the optimizations from Section 4.5. Empty rows indicate model checking did not terminate in 1 hour or ran out of memory. The number X×Y in the Program column means that there are X switches and Y clients in the example. Figure 4.2 shows the scalability of model checking with increasing distribution on the three largest examples. We noticed that the performance of the distributed model checker plateaued around 70 Erlang processes on these and other large examples. Thus, times (in table 4.1) are provided for configurations that use 70 Erlang processes. As we introduced abstractions, it is possible that we get false positives. I verified the existence of all bugs reported by Kuai manually and there were no false positives.

Besides the table, I plot the MAC learning example in Figure 4.3, which shows how significantly our optimization techniques reduce the state space. Though we still suffer from the state-space explosion problem, our optimizations delay it and enable us to verify SDNs with much larger configurations.

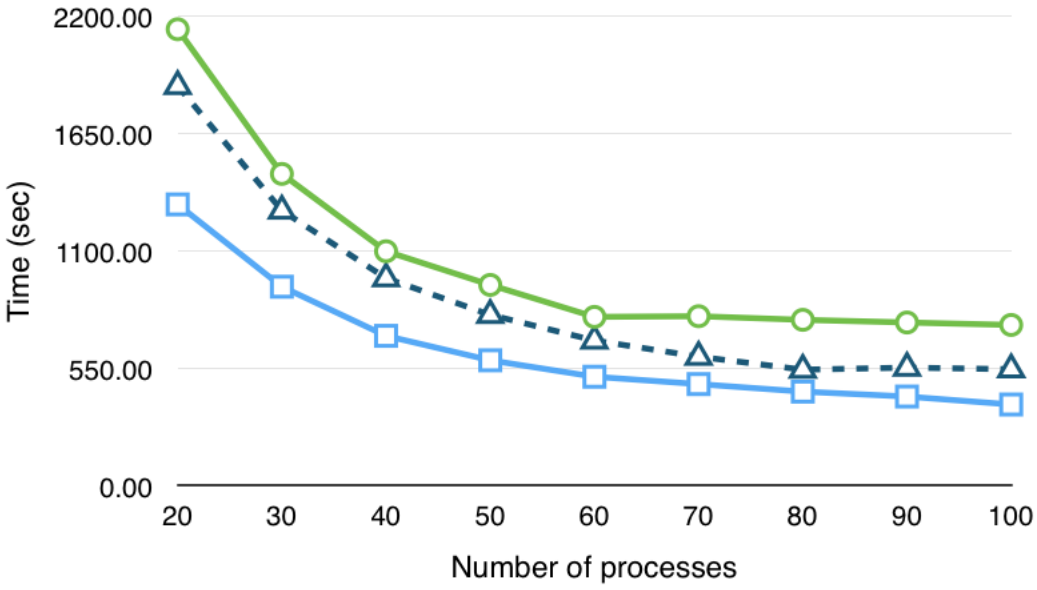

Figure 4.2: Verification time vs processes ∘ ML 9×2 Δ ML 6×3 □ FW(M) 4×4

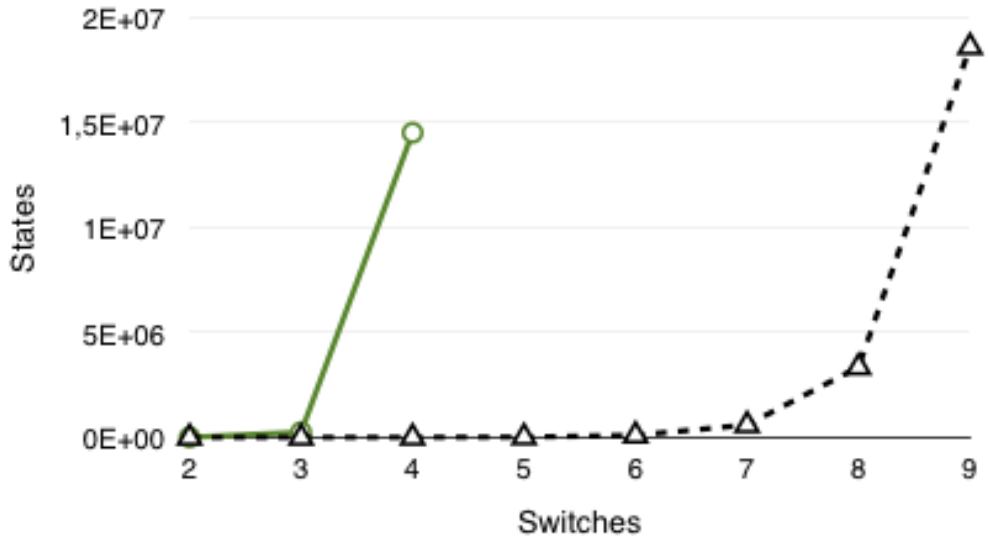

Figure 4.3: State space of MAC learning controller: Δ: optimized, ∘ unoptimized

I now describe the benchmarks in detail.

SSH I run Kuai on the SSH controller from Listing 4.1. It finds the control message reordering bug in 0.1 seconds. By adding a barrier after line 15, Kuai proves the correctness in 6.4 seconds by exploring 13 states. In contrast, the unoptimized version explores over 2 million states.

MAC Learning Controller (ML) This is based on the POX [POX] implementation of the standard ethernet discovery protocol. We check there are no forwarding loops (similar to [SNM13]), i.e., a packet should not reach a switch more than once. Packets are augmented with a bit for each switch which gets set when the switch processes that packet. The invariant is specified using these visit-bits (called *reached*): $\Box\ \forall sw \in Switches.\ \forall pkt \in sw.pq.\ (\neg pkt.reached(sw))$.

A cycle in the topology will lead to forwarding loops as the controller does not compute the minimum spanning tree. We discover the bug in a cyclic topology of 3 switches 3 clients in 0.47 seconds. We re-ran the example on a topology containing the minimum spanning tree of the original cyclic topology and the tool is able to prove that there were no forwarding loops in 6.39 seconds. We scale the example by adding more switches. We notice that while the verification on topology with 9 switches and 2 clients has fewer states than the one with 6 switches and 3 clients, each state in the latter case is bigger than the former and hence the memory and communication overheads are higher.

Single Switch Firewall (FW(S)) This is based on an advanced GENI assignment [GEN] on building an OpenFlow based firewall. The controller takes as input a simple configuration file which is a list of tuples of the form (client1, port1, client2, port2). This specifies that packets originating from client1 on port1 can be forwarded to client2 on port2. We abbreviate the tuples as $(client1 : port1 \rightarrow client2 : port2)$. Any flow not explicitly allowed is forbidden. The flows are uni-directional and the above flow will reject traffic initiated by client2 on port2 towards client1 on port1. However, once client1 initiates a flow, the firewall should allow client2 to reply back, making the flow bi-directional until client1 closes the connection.

The naive implementation of the controller is as follows: on receiving a packet ($c1\colon p1 \rightarrow c2\colon p2$), check if there is a tuple matching the flow in the policy. If it does, add rules ($c1\colon p1 \rightarrow c2\colon p2$) and ($c2\colon p2 \rightarrow c1\colon p1$) and forward the packet to $c2$. Otherwise add a rule to drop packets of the form ($c1\colon p1 \rightarrow c2\colon p2$). The invariant to verify here is to ensure the policy of the firewall, i.e., a packet from $c1\colon p1$ should be forwarded to $c2\colon p2$ if and only if ($c1\colon p2 \rightarrow c2\colon p2$) exists in the firewall policy or if ($c2\colon p2 \rightarrow c1\colon p1$) exists in the policy and $c2$ has already initiated the corresponding flow. The following formula specifies that allowed packets should not be dropped: $\Box\forall p \in \mathit{Packet}.\ \mathit{on\_dropped}(p) \Rightarrow \neg \mathit{flows}[p.src][\mathit{packet.src\_port}][\mathit{packet.dest}][p.\mathit{dest\_port}]$, where *on_dropped(p)* is set if a packet-drop transition is fired on packet p (and reset at the beginning of every transition). *flows* is an auxiliary variable in the controller which keeps track of allowed flows based on the firewall policy and initiating client.

I ran the experiment on a topology with 2 clients and a firewall and found an interesting bug in its implementation which is caused by not assigning proper priorities to rules. For example, when ($c1\colon p1 \rightarrow c2\colon p2$) is present in the policy but not ($c2\colon p2 \rightarrow c1\colon p1$), the rule to drop flows should have a lower priority than the rules to allow flows. Otherwise, the following bug would occur. If $c2$ initiates the flow ($c2\colon p2 \rightarrow c1\colon p1$) then the controller adds a rule to drop packets matching that flow. Later on, if $c1$ initiates ($c1\colon p1 \rightarrow c2\colon p2$) and the controller adds the corresponding rules to allow the flow on both directions, the switch now has two conflicting rules of the same priority. One to allow and the other to drop ($c2\colon p2 \rightarrow c1\colon p1$). The switch may non-deterministically choose to drop the packet. Once we fixed the bug, the tool could prove the invariant in 5.45 seconds.

Multiple Switch Firewalls (FW(M)) I extend the above example to include multiple replicated firewalls for load balancing. We now allow the clients to send packets to all of these firewalls. I augment the implementation of the single switch controller to add the same rules on all firewalls. However, this implementation no longer ensures the invariant in the multi-switch setting.

Consider the case with two firewalls, $f1$ and $f2$. The tool reports the following bug: $c1$

initiates ($c1\colon p1 \rightarrow c2\colon p2$) on firewall $f1$. The controller adds the corresponding rules to allow flows in both directions to $f1$ and $f2$ but only sends a barrier to $f1$. Now $f2$ delays the installation of ($c2\colon p2 \rightarrow c1\colon p1$) and $c2$ replies back to $c1$ through $f2$ which forwards the packet to the controller. The controller then drops the packet.

The fix here is to add the rules along with barriers on all switches and not just the switch from which the packet originates. With this fix the tool is able to prove the property in 8 seconds. In order to test the scalability, we tested the tool on increasing number of firewalls in the topology.

Resonance (RS) Resonance [NRF09] is a system for ensuring security in large networks using OpenFlow switches. When a new client enters the network, it is assigned *registration* state and is only allowed to communicate with a web portal. The portal either authenticates a client by sending a signal to the controller (and the controller assigns the client an *authenticated* state), or sets the client to *quarantined* state. In the authenticated state, the client is only allowed to communicate with a scanner. The scanner ensures that the client is not infected and sends a signal to the controller and lets the controller assign it an *operational* state. If an infection is detected, it is assigned a *quarantined* state. The clients in operational state are periodically scanned and moved to the quarantined state if they are infected. Quarantined clients cannot communicate with other clients.

In our model, the web portal non-deterministically chooses to authenticate or quarantine a client and the scanner non-deterministically marks a client operational or quarantined. We check the invariant that packets from quarantined clients should not be forwarded: $\Box \forall p \in \mathit{Packet}.\ \mathit{on\_forward}(p) \Rightarrow (\mathit{state}(p.\mathit{src}) \neq \mathit{Quarantined})$. Similar to *on_drop*, *on_forward* is set when packet-forward transition is fired and reset before the beginning of every transition. The controller follows the Resonance algorithm [NRF09].

I ran the experiment on a topology of two clients, one portal, one scanner and four switches. The topology is the same as in Figure 2 of [NRF09] without DHCP and DNS clients. Kuai proves the invariant in 6.6 seconds. I scale up the example by increasing the

number of clients.

Simple (SIM) Simple [QTC13] is a policy enforcement layer built on top of OpenFlow to ensure efficient middlebox traffic steering. In many network settings, traffic is routed through several middleboxes, such as firewalls, loggers, proxies, etc., before reaching the final destination. Simple takes a middlebox policy as input and translates this to forwarding rules to ensure the policy holds. The invariant ensures that all source packets to a client will be received and forwarded by the middleboxes specified in a given policy before the packet reaches its destination.

I ran the experiment on a topology of two clients, two firewalls, one IDS, one proxy and five switches (see Figure 1 of [QTC13]). Kuai can prove the invariant in 6.48 seconds.

We scale up the example by fixing the destination client and increasing the number of source clients that can send packets to it. Because of our "all packets in one shot" optimization (section 4.5.4), no matter how many packets get queued initially, they are all forwarded in lock-step as the controller forwarding rule applies to all incoming packets.

CHAPTER 5

Related Work

5.1 MrCrypt (Chapter 2)

Computing over encrypted data. The problem of (fully) homomorphic encryption was posed by Rivest, Adleman, and Dertouzos [RAD78], and the first fully homomorphic scheme was discovered by Gentry [Gen09]. Implementations of Gentry's construction remains prohibitively expensive [GH11]. A more efficient encryption scheme [NLV11] can perform unbounded additions but only a bounded number of multiplications. Cryptographically secure multi-party computations are also theoretically possible for general circuit evaluation [Yao86, SYY99]. Homomorphic encryption schemes have been proposed to protect data security in several applications including secure financial transactions [BRR09], secure voting [HS00], and sensor networks [CMT05].

The work closest to our own is the CryptDB project [PRZ11], which uses homomorphic encryption to run queries securely on relational databases. CryptDB encrypts the data in all possible encryption schemes, layered on top of each other in a structure resembling our lattice. A trusted proxy stands between clients and the database system, analyzes the SQL queries on the fly, and decrypts the relevant columns to the right encryption layers so that the query can be executed. The key difference between these two efforts is that MrCrypt performs static analysis of imperative Java programs while CryptDB performs analysis on database queries and so is limited to computations that are expressible in pure SQL (i.e., no user-defined functions). Further, because MrCrypt has up-front access to the programs, it can statically determine the best encryption schemes to use, avoiding the need to encrypt

data with multiple schemes and to employ a trusted proxy. However, encrypting data with multiple schemes allows some queries to be executed using CryptDB that cannot be handled by our system. Finally, we have formalized our approach and proven its correctness and security guarantees, while CryptDB provides only informal guarantees.

Other work in the database community has used homomorphic encryption for particular kinds of queries. For example, SADS [RVB09] allows encrypted text search and other work uses additive homomorphic schemes to support sum and average queries [GZ07]. These systems do not support general imperative computations.

Cryptographic schemes have been used to provide privacy and integrity in systems running on untrusted servers [LKM04, MSL10]. However, these systems have so far required application logic to be executed purely on the client. Our goal, on the other hand, is to enable computations to run directly on untrusted servers. It may be possible to incorporate ideas from these systems in order to augment our approach to guarantee integrity in addition to confidentiality.

Mitchell *et al.* formalize a domain-specific language (DSL) whose type system ensures that programs can be translated to run securely using either FH or secure multiparty computation [MSS12]. They also describe an implementation of their DSL embedded in Haskell. This approach can potentially be more expressive than ours but requires programmers to write programs in a specialized language, while MrCrypt handles existing Java programs with minimal code annotations. Finally, Mitchell *et al.* do not consider the use of partially homomorphic encryption schemes.

Static and dynamic analysis for security. There is a large body of work on static and dynamic techniques for enforcing security policies or for finding security vulnerabilities. Most language-based approaches to enforcing confidentiality are based on the notion of *secure information flow* [SM03]. These approaches are less applicable to the setting of cloud computing, where the adversary can have direct access to the machine on which a computation is being performed. For example, a common threat model in the context of secure

information flow assumes the adversary has access only to the public inputs and outputs of a computation. Researchers have augmented traditional information-flow type systems to reason about confidentiality in the presence of cryptographic operations [Vau11, FPR11], but these approaches require programmers to manually employ cryptography in their programs.

MrCrypt also leverages static analysis techniques, but for a different purpose — to identify the most efficient encryption schemes to use for each input column of data. As described in our formalism, this analysis is similar to techniques for flow-insensitive type qualifier inference [OP97, FJK06].

Computing in untrusted environments. The Excalibur system [SRG12] uses trusted platform modules (TPMs) to guarantee that privileged cloud administrators cannot inspect or tamper with the contents of a VM. While this approach provides the same security guarantees as MrCrypt, it requires additional investment from the cloud companies to install special TPM chips on each node in the cloud and for managing keys. CLAMP [PMW09] prevents web servers from leaking sensitive user data by isolating code running on behalf of one user from that of other users. However, CLAMP does not protect user confidentiality against honest-but-curious cloud administrators. Finally, work on differential privacy for MapReduce (e.g., [RSK10]) is dual to our concern: in that setting the server is trusted but information exposed to clients is minimized.

5.2 Vega (Chapter 3)

Program transformations for cloud. Several programs transformations have been applied for cloud programs. Apache Pig [ORS08b] is a source-to-source transformer that translates high-level PigLatin code to low-level MapReduce code. In the process it applies several optimizations to improve performance. However, Pig system does not have any support for incremental code changes. Other transformations include improving performance by adding support for additional hardware [CSR10]. Other programs transformations have been successfully applied to areas such as security ([TLM13], [PRZ11], [MSS12]) but these cannot be

applied to improve performance.

Incremental view maintenance. Incremental view maintenance is a well studied problem ([GMS93], [CW91], [NJL12]) where changes in input data are propagated through a set of relations (or views) on input data. Naiad [MMI13] is an application of these techniques to large scale data with additional support for nested control flow. Most of the work in this area deals with data changes and a few restricted code changes (such as addition of relations to a union of relations). In addition, the ideas are best suited for structured data. Vega is specifically built to handle more general code changes and can handle incremental computation in both structured and unstructured settings.

Database query optimization. Traditional database query planning [Cha98] involves two kinds of optimizations: logical plan optimization that rewrites the query to an equivalent and hopefully more performant one; and physical plan optimization that chooses the best implementation for the physical operators (such as joins) based on the query being executed and input data. Vega's optimizations are closer in spirit to the logical plan optimizations. Traditional query planning usually does not take into account results of the old queries and does not have a system to cache results for future use. [AXL15] is an application of database query planning to large scale workflows. It compiles high-level dataframe API into low level Spark operators. Vega optimizes Spark workflows directly without the need for another high-level language. However Vega's performance optimizations serve a very different goal: for example [AXL15] optimizations will try to push the filters as close to the source of data as possible while Vega tries to push the filters toward the end of the workflow if it can reuse existing results.

ETL frameworks. Several frameworks exist for extract, transform, load (ETL) workflows in databases (see [Vas09] for a comprehensive survey). The main focus of these projects is to provide general rules for cleaning up and homogenizing structured data. Some frameworks attempt to automatically derive rules by observing patterns in data. While Vega can be applied to ETL use-cases, it solves a general data exploration problem. Vega transforms

can be arbitrary functions, not necessarily tied to a specific domain. Most of the ETL frameworks' rules critically depend on structured data and relational constraints such as uniqueness and functional dependencies. Vega works on unstructured data and is not tied to any specific constraint structure.

Fast approximate answers. Frameworks such as [AGP00], [BCD03], [OBE09] can provide interactive response times by providing approximate answers to ad-hoc queries. [AMP13] provides approximates answers in large-scale setting. The key idea behind them is to sample enough data upfront, so they can answer significant number of queries by extrapolating from the samples. Vega performs no sampling, computes directly on the base data and provides exact answers.

5.3 Kuai (Chapter 4)

There is a lot of systems and networking interest in SDNs [JKM13, FRZ13] and standards such as Openflow [MAB08]. From the formal methods perspective, research has focused on verified programming language frameworks for writing SDN controllers [FHF11, GRF13]. Here, verification refers to correct compilation from Frenetic to executable code, or to checking composability of programs, not the correctness of invariants.

Previous model checking attempts for SDNs mostly focused either on proving a static snapshot of the network [KVM12] or on model checking or symbolic simulation techniques for a fixed number of packets [CVP12, NGD13]. Recent work extended to controller updates and arbitrary number of packets [SNM13], but used a manual process to add non-interference lemmas. In contrast, our technique automatically deals with unboundedly many packets and, thanks to the partial-order techniques, scales to much larger configurations than reported in [SNM13]. Program verification for SDN controllers using loop invariants and SMT solving has been proposed recently [BBG14]. While the invariants can quantify over the network (and therefore not limited to finite topologies), the model of the network ignores asynchronous interleavings of packet and control message processing that we handle here.

Our work builds on top of distributed enumerative model checking and the PReach tool [BBP10]. Our contribution is identifying domain specific state space reduction heuristics that enable us to explore large configurations.

CHAPTER 6

Conclusion

Programming language design has already played an important role in cloud computing by providing restricted but sufficiently expressive programming models to deal with issues such as resource contention and fault-tolerance.

In this dissertation, I have provided evidence that the complimentary aspect to design, program analysis, can bring just as many advantages to cloud system design and have laid groundwork for further adoption. I have presented three projects that tackle key challenges in cloud computing.

MrCrypt is a framework that ensures data confidentiality when both code and data are present in a third-party datacenter. It achieves this by automatically transforming Java programs to use homomorphic encryption schemes. MrCrypt is able to securely execute majority of the cloud benchmarks with modest encryption overheads. The project can be extended in several directions. It would be very useful to ensure integrity of computations — i.e. to make sure the result returned by the cloud is correct. It would also be useful to expand the tool to handle different classes of programs, especially in machine learning domain (see [BPT14] for some interesting developments in this area).

Vega is a library that performs incremental re-computation in the face of code changes. It achieves this by rewriting Big Data workflows to reuse previous results. Vega achieves one to two orders magnitude performance gains in several real-world workflows. Currently Vega provides libraries for various functions and their inverses that are commonly used by programmers. For future work, we would like to automatically infer the inverses or at least check that the inverse of a function given by the user is correct. The "algebraic" approach of

Vega (i.e. using inversion and distribution) can also be extended to other important domains such as machine-learning (for example by using techniques presented in [Izb13]).

Kuai is a software model checking tool that verifies safety properties of Software Defined Networks. It achieves this by using abstractions to make state-space finite, then applying partial order reduction to reduce it significantly and finally using a distributed checker to parallelize the task across many nodes. Kuai is able to verify correctness in many SDN benchmarks. Possible extensions include adding support for liveness properties and extending it to other critical areas such as the authentication layer. This would involve developing additional domain-specific partial order reductions.

An overarching goal of this dissertation is to push expert domain knowledge into the compiler level as much as possible so that many (non-expert) programmers can benefit from it. MrCrypt frees the programmers from knowing and correctly using specialized encryption algorithms, Vega alleviates the programmer's burden when it comes to performance tuning and Kuai frees the programmer from coming up with manual proofs to ensure correctness of their network policies. The area of program analysis provides rich tools to achieve this in a sound and efficient way.

References

[AGP00] Swarup Acharya, Phillip B Gibbons, and Viswanath Poosala. "Congressional samples for approximate answering of group-by queries." In *ACM SIGMOD Record*, volume 29, pp. 487–498. ACM, 2000.

[ALT12] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. "PUMA: Purdue MapReduce Benchmarks Suite." Technical Report TR-ECE-12-11, School of Electrical and Computer Engineering, Purdue University, 2012.

[AMP13] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. "BlinkDB: queries with bounded errors and bounded response times on very large data." In *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 29–42. ACM, 2013.

[AXL15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. "Spark SQL: Relational data processing in Spark." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394. ACM, 2015.

[BBG14] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. "VeriCon: Towards Verifying Controller Programs in Software-defined Networks." PLDI '14, pp. 282–293, 2014.

[BBP10] B. Bingham, J. Bingham, F.M. de Paula, J. Erickson, G. Singh, and M. Reitblatt. "Industrial Strength Distributed Explicit State Model Checking." In *PDMC-HIBI*, pp. 28–36, Sept 2010.

[BCD03] Brian Babcock, Surajit Chaudhuri, and Gautam Das. "Dynamic sample selection for approximate query processing." In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 539–550. ACM, 2003.

[BCL09] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. "Order-Preserving Symmetric Encryption." In *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pp. 224–241. Springer, 2009.

[BCO11] A. Boldyreva, N. Chenette, and A. O'Neill. "Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions." In *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pp. 578–595. Springer, 2011.

[BPT14] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. "Machine learning classification over encrypted data." *Crypto ePrint Archive*, 2014.

[BRR09] M. Bellare, T. Ristenpart, P. Rogaway, and T. Stegers. "Format-Preserving Encryption." In *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pp. 295–312. Springer, 2009.

[Cha98] Surajit Chaudhuri. "An overview of query optimization in relational systems." In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 34–43. ACM, 1998.

[CMT05] C. Castelluccia, E. Mykletun, and G. Tsudik. "Efficient Aggregation of encrypted data in Wireless Sensor Networks." In *Proceedings of the The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, MOBIQUITOUS '05, pp. 109–117, Washington, DC, USA, 2005. IEEE Computer Society.

[CSR10] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. "Customizable domain-specific computing." *IEEE Design & Test of Computers*, (2):6–15, 2010.

[CVP12] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. "A NICE Way to Test Openflow Applications." NSDI'12, pp. 127–140, 2012.

[CW91] Stefano Ceri and Jennifer Widom. "Deriving production rules for incremental view maintenance." 1991.

[DG08] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM*, **51**(1):107–113, 2008.

[DG10] J. Dean and S. Ghemawat. "MapReduce: a flexible data processing tool." *Commun. ACM*, **53**(1):72–77, 2010.

[Dil96] David L Dill. "The Mur ϕ verification system." In *Computer Aided Verification*, pp. 390–393. Springer, 1996.

[DR02] J. Daemen and V. Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.

[ElG85] T. ElGamal. "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms." *IEEE Transactions on Information Theory*, **31**(4):469–472, 1985.

[Fai15] Ross Fairbanks. "WikiReverse." `https://wikireverse.org`, 2015.

[FHF11] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. "Frenetic: A Network Programming Language." ICFP '11, pp. 279–291, New York, NY, USA, 2011. ACM.

[FJK06] J.S. Foster, R. Johnson, J. Kodumal, and A. Aiken. "Flow-insensitive type qualifiers." *ACM Trans. Program. Lang. Syst.*, **28**(6):1035–1087, November 2006.

[FPR11] Cédric Fournet, Jérémy Planul, and Tamara Rezk. "Information-flow types for homomorphic encryptions." In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pp. 351–360. ACM, 2011.

[FRZ13] Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The Road to SDN." *Queue*, **11**(12):20:20–20:40, December 2013.

[GEN] GENI Assignment. `http://groups.geni.net/geni/wiki/GENIEducation/SampleAssignments/OpenFlowFirewallAssignment/ExerciseLayout/Execute`.

[Gen09] C. Gentry. "Fully homomorphic encryption using ideal lattices." In *STOC 09: Symposium on Theory of Computing*. ACM, 2009.

[Gen10] C. Gentry. "Computing arbitrary functions of encrypted data." *Commun. ACM*, **53**(3):97–105, 2010.

[GH11] C. Gentry and S. Halevi. "Implementing Gentry's Fully-Homomorphic Encryption Scheme." In *EUROCRYPT 11*, volume 6632 of *Lecture Notes in Computer Science*, pp. 129–148. Springer, 2011.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. "Maintaining views incrementally." *ACM SIGMOD Record*, **22**(2):157–166, 1993.

[GRF13] Arjun Guha, Mark Reitblatt, and Nate Foster. "Machine-verified Network Controllers." PLDI '13, pp. 483–494, New York,USA, 2013. ACM.

[GZ07] T. Ge and S. Zdonik. "Answering aggregation queries in a secure system model." In *Proceedings of the 33rd international conference on Very large data bases*, pp. 519–530. VLDB Endowment, 2007.

[HR03] S. Halevi and P. Rogaway. "A tweakable enciphering mode." *Advances in Cryptology-CRYPTO 2003*, pp. 482–499, 2003.

[HS00] M. Hirt and K. Sako. "Efficient receipt-free voting based on homomorphic encryption." In *Proceedings of the 19th international conference on Theory and application of cryptographic techniques*, EUROCRYPT'00, pp. 539–556, Berlin, Heidelberg, 2000. Springer-Verlag.

[IBY07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks." In *ACM SIGOPS Operating Systems Review*, volume 41, pp. 59–72. ACM, 2007.

[IPW11] Robert Ikeda, Hyunjung Park, and Jennifer Widom. "Provenance for generalized map and reduce workflows." 2011.

[Izb13] Michael Izbicki. "Algebraic classifiers: a generic approach to fast cross-validation, online training, and parallel training." In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 648–656, 2013.

[JKM13] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. "B4: Experience with a Globally-deployed Software Defined Wan." SIGCOMM13, pp. 3–14, New York, NY, USA, 2013.

[Kow08] E. Kowalski. "Insider Threat Study: Illicit Cyber Activity in the Information Technology and Telecommunications Sector." Technical report, Technical report, U.S. Secret Service and CMU, 2008.

[KPH12] Sean Kandel, Andreas Paepcke, Joseph M Hellerstein, and Jeffrey Heer. "Enterprise data analysis and visualization: An interview study." *Visualization and Computer Graphics, IEEE Transactions on*, **18**(12):2917–2926, 2012.

[KVM12] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks." In *NSDI*, pp. 113–126, 2012.

[LDY13] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. "Scalable lineage capture for debugging DISC analytics." In *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 17. ACM, 2013.

[LKM04] J. Li, M. Krohn, D. Mazières, and D. Shasha. "Secure untrusted data repository (SUNDR)." In *OSDI 04: Operating Systems Design and Implementation*, pp. 91–106. ACM, 2004.

[MAB08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: Enabling Innovation in Campus Networks." *SIGCOMM*, **38**(2):69–74, March 2008.

[MCB11] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. "Big data: The next frontier for innovation, competition, and productivity." 2011.

[MMI13] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. "Differential dataflow." In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[MSL10] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. "Depot: Cloud storage with minimal trust." In *OSDI 10: Operating Systems Design and Implementation*. ACM, 2010.

[MSS12] John C Mitchell, Rahul Sharma, Dumitru Stefan, and Jeramy Zimmerman. "Information-flow control for programming on encrypted data." In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pp. 45–60. IEEE, 2012.

[MTW] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. "Kuai: A Model Checker for Software-defined Networks. Technical report." `http://web.cs.ucla.edu/~saideep/TR_FMCAD2014.pdf`.

[MTW14] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. "Kuai: A model checker for software-defined networks." In *Formal Methods in Computer-Aided Design (FMCAD), 2014*, pp. 163–170. IEEE, 2014.

[NCM03] N. Nystrom, M. Clarkson, and A. Myers. "Polyglot: An extensible compiler framework for Java." In *Compiler Construction*, pp. 138–152. Springer, 2003.

[NGD13] Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. "A Balance of Power: Expressive, Analyzable Controller Programming." HotSDN '13, pp. 79–84, New York, NY, USA, 2013. ACM.

[NJL12] Vivek Nigam, Limin Jia, Boon Thau Loo, and Andre Scedrov. "Maintaining distributed logic programs incrementally." *Computer Languages, Systems & Structures*, **38**(2):158–180, 2012.

[NLV11] M. Naehrig, K. Lauter, and V. Vaikuntanathan. "Can homomorphic encryption be practical?" In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pp. 113–124, New York, NY, USA, 2011. ACM.

[NRF09] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. "Resonance: Dynamic Access Control for Enterprise Networks." WREN '09, pp. 11–18, New York, NY, USA, 2009. ACM.

[OBE09] Christopher Olston, Edward Bortnikov, Khaled Elmeleegy, Flavio Junqueira, and Benjamin Reed. "Interactive Analysis of Web-Scale Data." In *CIDR*, 2009.

[OP97] Peter Ørbæk and Jens Palsberg. "Trust in the λ-calculus." *Journal of Functional Programming*, **7**(6):557–591, November 1997.

[ORS08a] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. "Pig latin: a not-so-foreign language for data processing." In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pp. 1099–1110, New York, NY, USA, 2008. ACM.

[ORS08b] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. "Pig latin: a not-so-foreign language for data processing." In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1099–1110. ACM, 2008.

[Pai99] P. Paillier. "Public-key cryptosystems based on composite degree residuosity classes." In *EUROCRYPT 99: Theory and Applications of Cryptographic Techniques*, 1999.

[PD10] Daniel Peng and Frank Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications." In *OSDI*, volume 10, pp. 1–15, 2010.

[PMW09] B. Parno, J.M. McCune, D. Wendlandt, D.G. Andersen, and A. Perrig. "CLAMP: Practical Prevention of Large-Scale Data Leaks." In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pp. 154–169, Washington, DC, USA, 2009. IEEE Computer Society.

[POX] POX. `http://www.noxrepo.org/pox/about-pox/`.

[PPR09] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, and M. Stonebraker. "A comparison of approaches to large-scale data analysis." In *Proceedings of the 35th SIGMOD international conference on Management of data*, pp. 165–178. ACM, 2009.

[PRZ11] R.A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. "CryptDB: protecting confidentiality with encrypted query processing." In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 85–100. ACM, 2011.

[QTC13] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. "SIMPLE-fying Middlebox Policy Enforcement Using SDN." SIGCOMM13, pp. 27–38, New York, NY, USA, 2013. ACM.

[RAD78] R. Rivest, L. Adleman, and M.L. Dertouzos. "On Data Banks and Privacy Homomorphisms." In *Foundations of Secure Computation*, pp. 169–179. Academic Press, 1978.

[RSK10] I. Roy, S.T.V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. "Airavat: Security and Privacy for MapReduce." In *NSDI*, pp. 297–312. USENIX, 2010.

[RVB09] M. Raykova, B. Vo, S.M. Bellovin, and T. Malkin. "Secure anonymous database search." In *CCSW 09: Cloud Computing Security Workshop*, pp. 115–126. ACM, 2009.

[Sch94] B. Schneier. "Description of a new variable-length key, 64-bit block cipher (Blowfish)." In *Fast Software Encryption*, pp. 191–204. Springer, 1994.

[Sch96] B. Schneier. *Applied cryptography*. Wiley, 2nd edition, 1996.

[SM03] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security." *IEEE Journal on Selected Areas in Communications*, **21**(1):5–19, 2003.

[SNM13] D. Sethi, S. Narayana, and S. Malik. "Abstractions for model checking SDN controllers." In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pp. 145–148, Oct 2013.

[SRG12] N. Santos, R. Rodrigues, K.P. Gummadi, and S. Saroiu. "Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services." In *Usenix Security Symposium*. USENIX Association, 2012.

[SYY99] T. Sander, A. Young, and M. Yung. "Non-Interactive CryptoComputing for NC^1." In *FOCS 99: Foundations of Computer Science*. IEEE, 1999.

[TLM13] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. "MrCrypt: Static analysis for secure cloud computations." In *ACM SIGPLAN Notices*, volume 48, pp. 271–286. ACM, 2013.

[Vas09] Panos Vassiliadis. "A survey of extract-transform-load technology." *IJDWM*, **5**(3):1–27, 2009.

[Vau11] J.A. Vaughan. "AuraConf: a unified approach to authorization and confidentiality." In *Proceedings of the 7th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '11, pp. 45–58, New York, NY, USA, 2011. ACM.

[Whe15] David A. Wheeler. "SLOCCount." `http://www.dwheeler.com/sloccount/`, 2015.

[Whi12] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[Yao86] A. Yao. "How to generate and exchange secrets." In *FOCS 86: Foundations of Computer Science*, pp. 162–167. IEEE, 1986.

[ZCD12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2. USENIX Association, 2012.