Research Statement

Caleb Stanford

March 2021

Overview: Programming Solutions for Data-Driven Computing

Data is increasingly at the center of society, and most data is now generated in higher velocity and higher volume than can be feasibly stored. For these reasons, programmers and industrial companies are adopting new software abstractions and building specialized systems for processing data more efficiently. Some core motives for this new software are (1) the emergence of *Internet of Things (IoT)* devices and data, which have caused a multiplicative increase in data being generated; (2) the requirement to *scale* services and applications across many computers and ideally many data centers; and (3) the requirement for *minimal response times* when accessing services and analyzing data over the cloud. My research brings two fields of computer science to attack these practical problems: programming languages, which studies how to design the languages programers use to write software, and formal methods, which studies how to use mathematical and logical reasoning to ensure that programs are reliable and free of bugs and vulnerabilities. In sum, **my research lies in applying programming languages and formal methods techniques to emerging data-driven computing applications.**

I have particularly focused on the design and implementation of *distributed stream processing systems*, which are specialized software platforms for processing streaming data. Today's most popular examples include Apache Storm, Apache Spark, Apache Flink, and Timely Dataflow. My research aims to answer questions such as the following:

1. How can users best specify computations over streaming data?
2. What tools and techniques are available – and lacking – for ensuring such computations are correct?
3. What tools and techniques can be used to enable more efficient implementations of such computations?

My PhD research [2, 1, 3, 5, 4] focuses in two domains: first, improving the *reliability* of programming such applications through better guarantees about correctness, particularly due to the distributed setting. Second, improving the *performance* of programming such applications through optimized intermediate layers that can be used by software and compilers. I will begin by describing the landscape of data that needs to be processed in representative applications. Then I will discuss my research in the above two directions, as well as other research I have done in my time at Penn and future directions.

The Data Landscape and Key Applications

IoT. The future will include a massive amount of data from Internet of Things devices, including health monitoring implants, drones, and regular smartphones. Many emerging applications rely on streaming analytics of such data in real time. For video streaming devices, minimizing bandwidth use by pushing computation to the edge devices is a primary concern. For health devices and smartphones, often a primary concern is battery conservation, due to the high battery used by applications that are constantly running.

Cloud services. Software companies increasingly provide cloud services to individual buyers, rather than companies and individuals running their own servers. This business model has led to many data streams in the cloud, for example, a stream of web-clicks, a stream of financial transactions, or a stream of social media events such as responses and events on Twitter.

Runtime Monitoring. Large-scale software projects often change too often and too dynamically to do full-scale formal verification of correctness. As a result, it is popular in industry to instead monitor traces of existing systems to find correctness and performance bugs, in order to catch problems when they occur. For example, large tech companies monitor streams of transactions and web requests to identify if services are down and if they are responding correctly to the requests.

Reliability

Stream processing applications regularly fail. Failures can be catastrophic as the system typically needs to then be taken down and restarted again. One reason is that nodes in a distributed system can fail, but a more unique problem to the streaming domain is that *data streams* can be unpredictable in a number of ways: in the dynamics of their volume over time, and in the fact that data is fundamentally out-of-order. I believe that these challenges can be addressed at the language level: through better software libraries and programming language tools, such as type systems.

In this vein, we have built tool support on top of Apache Storm [5] and Apache Flink [4] for testing and verification to prevent bugs due to data parallelism. Our work in Apache Storm shows that, by manipulating data streams with extra type information, programmers can statically ensure that their code parallelizes correctly. Our work in Apache Flink focuses on testing, showing that programmers can use similar annotations to automatically test applications for bugs due to parallelism, without having to modify the application source code.

Performance

Performance in stream processing refers to (1) how many bytes of data can be processed per second by the system (throughput)? (2) what is the response time from input data to output data by the system (latency)? Existing systems are heavily engineered but sometimes make compromises in order to expose a convenient API to the end user. An attack surface on this problem is given by classical techniques in programming language compilers: how can we design *representation layers* (called intermediate representations) for the program that can be used by the software to compile and optimize the code efficiently?

To address these challenges, for high-level query languages incorporating user-defined stateful and quantitative computations, we show that a new intermediate representation can be used to achieve efficient compilation with static bounds on performance [3], and formally study the benefits of related program representations [2, 1]. Our recent and ongoing work (in submission) focuses on distributed compilation of stream processing applications in the internet of things domain: we have built a prototype stream processing system (in submission) for this problem which safely distributes the computation over many devices, while minimizing network load.

Other Research

In addition to my research at Penn, I have explored fruitful collaborations with external researchers through doing research internships at Amazon Web Services (Summer 2019) and Microsoft Research (Summer 2020). Both of these internships were related to applying formal methods to computer security applications. At AWS, I developed tools to automate the security review process. Specifically, I analyzed the permissions configurations of cloud resources in conjunction with other account data to more easily detect AWS account configurations deviating from security best practice. At Microsoft, I worked with Margus Veanes and Nikolaj Bjorner on the Z3 SMT solver (satisfiability modulo theories) for solving regular expression constraints on strings, based on a formal mathematical technique called symbolic derivatives. Our solver supports typical regular expression constraints arising in target security applications more efficiently than competing solvers. This work has been conditionally accepted to PLDI [6], with follow-up work in submission.

Future Work

Data processing is fundamental to both computers and society, and I believe that software applications will increasingly adopt a data-centric view. More broadly, I am interested in the interaction between Programming Languages, Data Processing, and Distributed Systems. There are a number of future opportunities in stream processing along the directions of reliability and performance. For reliability, there is no good approach that handles both out-of-order data and node failures, which is an important source of crashes in practice. I also believe that tools such as ours could be better integrated into existing software infrastructure. For performance, I am interested in formal models of performance: can we mathematically predict how many resources it will take a stream processing program to execute, including throughput and latency? Finally, what abstractions are missing to bridge the gap between software and hardware in this space?

References

[1] Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. Streamable regular transductions. *Theoretical Computer Science*, 807:15–41, 2020.

[2] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Automata-based stream processing. In *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[3] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Modular quantitative monitoring. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–31, 2019.

[4] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Diffstream: differential output testing for stream processing programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[5] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G Ives, and Val Tannen. Data-trace types for distributed stream processing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 670–685, 2019.

[6] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. Symbolic boolean derivatives for efficiently solving extended regular expression constraints. Technical report, Technical Report MSR-TR-2020-25, 2020.