Research Statement

Caleb Stanford

Fall 2021

Today, data is generated with higher velocity and higher volume than can be feasibly stored, raising the need for new algorithms, software abstractions, and systems. This requirement for *distributed online data processing* has been recognized across application domains including Internet of Things (IoT) systems, cloud computing, microservices, and data analytics. It has also prompted industry investment and an ever-increasing number of major software projects designed to serve these needs.[1] Despite the growing interest, current programming support is not well-adapted to the online setting.

To address this gap, my research investigates foundations, software abstractions, and verification tools for online data processing.

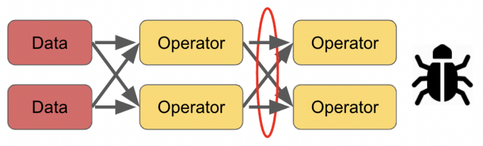
Emerging online systems are a fundamentally new paradigm (Figure 1). First, online data movement is highly time- and order-dependent, leading to new challenges in concurrency. For example, online aggregators (e.g., sum of data over the last 1000 events per ID) are usually applied out-of-order and in parallel, but doing so correctly requires ordering assumptions which are neither programmatically articulated nor reliably enforced. Second, compared to offline software, failures in online systems are catastrophic, as they can lead to service outages with irreversible repercussions (i.e., permanently dropped data and requests). Widely known examples of such outages are the October 2021 Facebook outage and the 2020 Google outages. Third, in the online setting, standards for efficiency are more rigid, often involving near-constant worst-case time and space bounds. For example, IoT medical monitoring applications must run using a predictable amount of memory with a minimal impact on battery life. Finally, when verification is deployed at runtime in an online system, verification strategies must operate on tighter resource constraints and make compromises in order to achieve performance. In fact, almost all generic verification tools (e.g., model checkers and automatic theorem provers) are resultingly completely out of scope: they cannot be run in the critical online loop.

My research applies programming languages and formal methods techniques to the above challenges (Figure 2). I have worked on ensuring *safe order-aware concurrency* to prevent subtle bugs for programs written in online data processing systems such as Apache Flink and Apache Storm [1, 2, 3, 4]. In this line of research, we introduce the idea of modeling data as *partially ordered* and we annotate these partial orders with a domain-specific type system. Related to this work, we have leveraged annotated partial orders for *automatic distributed execution* of complex stateful programs [5, 6]. In the performance direction, my work proposes *streaming models with formal performance guarantees* [7, 8, 9] with applications in medical devices and runtime monitoring. Both safe ordering and performance guarantees are aspects of preventing failures in online applications. Finally towards enabling incremental verification technology, I have proposed *new online techniques in automated theorem proving*, implemented in Microsoft's state-of-the-art satisfiability modulo theories solver Z3 [10, 11]. Our techniques improve on the state-of-the-art in the two areas of solving string constraints and state space exploration. Besides these directions, I am interested in exploring online ideas in other areas, including ongoing collaborations in testing for programmable networks. My research has been published in the premier conferences in programming languages, including PLDI, POPL, OOPSLA, and PPoPP [1, 2, 5, 7, 10].

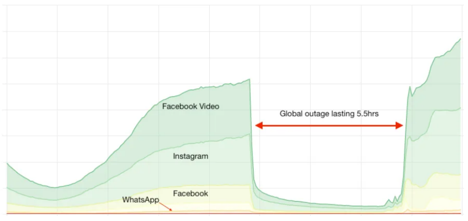
If successful, new programming languages and formal methods research could revolutionize online systems development. This research could enable applications that are more semantically robust, bug-free, and verified both at development time and during live deployment. On the technical side, there are a wealth of research problems to tackle that require combining new ideas and understanding with existing paradigms from data management, distributed systems, networks, programming languages, and algorithms.

Research philosophy. My research spans the spectrum from theory to practice. On the theory side, I am especially excited about identifying *where theory is missing* and developing new foundations and abstractions which are more directly applicable to today's data and technology landscape [3, 6, 7, 8, 9, 10, 11]. On the practice side, I spend most of my time applying foundations and abstractions to system design and implementation [1, 2, 4, 5, 10, 11]. I believe that the best theory demonstrably applies to software engineering and systems practice, and the best practice has well-motivated design choices rooted in theoretical foundations.

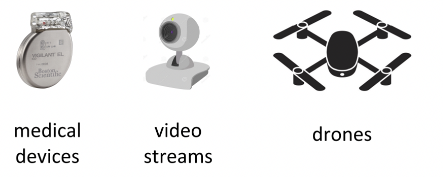
[1] E.g.: Kinesis from Amazon, Timely and Differential Dataflow from Materialize, and a range of open-source frameworks including Spark, Storm, Flink, Kafka, Samza, Heron, and Calcite from the Apache Software Foundation.



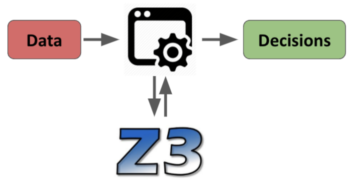
(a) Concurrent data movement



(b) Catastrophic failures



(c) Rigid performance requirements



(d) Incremental verification tools

Figure 1: Facets of the emerging online systems paradigm. *(a) Stateful processing is highly time- and order-dependent. (b) Failures in cloud services induce outages. (c) Time- and space-usage require near-constant bounds. (d) Verification tools must work online and provide utilizable incremental feedback.*

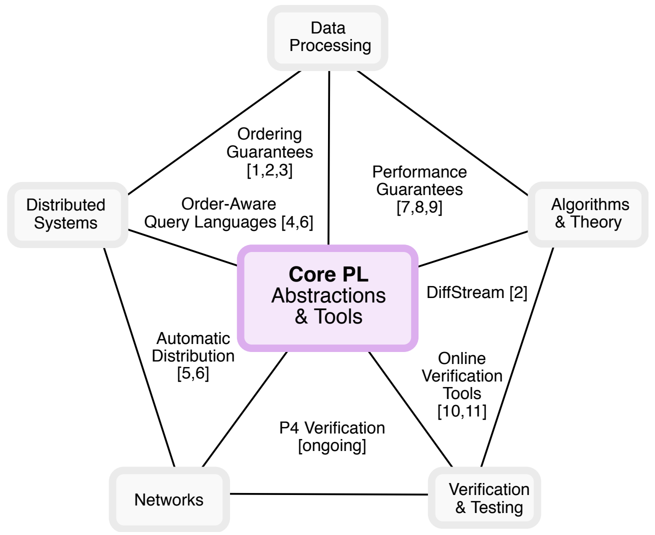

Data Processing
Distributed Systems
Algorithms & Theory
Networks
Verification & Testing
Core PL Abstractions & Tools
Ordering Guarantees [1,2,3]
Performance Guarantees [7,8,9]
Order-Aware Query Languages [4,6]
DiffStream [2]
Automatic Distribution [5,6]
Online Verification Tools [10,11]
P4 Verification [ongoing]

Figure 2: Research areas overview. (Each triangle represents the intersection of its three vertices.)

Research directions

Safe order-aware concurrency. One of the most fundamental problems that arises in developing online distributed applications is modeling order. Traditionally, online data processing applications have been developed using two orthogonal models: in the *relational* model, all data is assumed to be unordered, and in the *sequential* model, data comes in ordered streams. However, real systems involve a combination of sequential and relational data, as well as more complex partially ordered data. Notable examples of these orders include partitioned streams and relations synchronized by watermark events, which are commonplace in almost any high-throughput online system. These data are best modeled as *partially ordered sets*, an idea first proposed in our work, to abstract a general ordering relationship between events in a system.

Leveraging the partially ordered set approach, my coauthors and I built tool support on top of Apache Storm [1] and Apache Flink [2] for testing and verification to prevent bugs due to data parallelism. My work in Apache Storm shows that, by manipulating data streams with extra type information, programmers can statically ensure that their code parallelizes correctly. My 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. This research spans theory (semantics of computations over partial orders) to practice (tools utilizing partial-order type information). On the semantics side, we have explored the foundations of monotone transformations on partially ordered streams [3, 6].

Taking the partial-order approach further, I have investigated new programming models which enable *automatic distribution* guided by the partial ordering type information. Fully automatic distribution has been a long-term goal in programming languages and distributed systems. Today, aggressively-parallel programming frameworks in the style of MapReduce are one of the most popular solutions, and have also inspired most work on parallelization in online data processing. However, many workloads can't be done in an aggressively-parallel fashion and do require some *synchronization* across different tasks. I proposed and developed a new programming model, leveraging a distribution-independent specification of the required synchronization to distribute streaming applications automatically, and worked on the development and evaluation of a prototype stream processing system based on this idea [5]. We demonstrate that our system, in contrast to aggressively-parallel systems, can enable true automatic distribution for complex workloads with synchronization requirements without regard to the number of parallel instances or the partitioning of data.

Performance guarantees. Why do online services fail? Besides fault-tolerance and application logic errors, one understudied reason is that they are inherently performance-sensitive: if given too much input data, rather than simply taking a long time to complete, they may start dropping requests or crash entirely. Performance bounds on online applications are also very hard to determine, as (unlike programs running directly on hardware) they are typically deployed in a distributed or cloud computing setting with dynamic resources. Existing efforts to optimize and improve system performance do not offer *formally guaranteed performance*, so they do not demonstrate a solution to this problem.

Rooted in logic and automata theory, my research has proposed to address this by developing abstractions which rethink performance. Specifically, I have developed abstractions where we can know exactly how much time and how much space a set of operations will take to complete.[2] I designed a new intermediate representation (IR) for streaming programs [7] for monitoring data streams online, and showed how these can be used to compile expressive quantitative temporal queries with provably *quadratic* (in the query) bounds on the performance of the compiled representation . Other theoretical results include exponential succinctness of the model over competing approaches, making it suitable for streaming. In fact, I showed that this IR can express *all* streaming programs in a certain sense, where the streaming programs are expressed as loop-free code. My open-source Rust implementation of the IR could easily be incorporated into general-purpose stream processing frameworks or used for specialized IoT applications. I have also formally studied the benefits of related program representations [8, 9].

Online verification tools. Traditional full-featured verification tools often require seconds to minutes to respond to verification queries, making them infeasible for use in online applications in the main feedback loop. For example, a query to a satisfiability modulo theories (SMT) solver is unlikely to give results in time for a service request which requires near-instantaneous latency; and in the case that it times out, it is unlikely to give *incremental* partial feedback. Verification queries in online applications are especially useful in emerging cloud security applications (here, security requires verifying properties of requests and access dynamically at runtime) and are strictly necessary in all cases where program correctness information is not fully known offline. If we can redesign SMT to reduce response times and always provide such incremental feedback, that will open the door to new use cases and applications for verification technology.

I have worked on the internals of SMT solvers to improve their performance in an incremental setting; my research is currently incorporated in the state-of-the-art solver Z3 through ongoing collaborations at Microsoft. First, I have focused on solving string and regular expression constraints, which are used to verify the security of string-manipulating programs. The security applications of string solving technology is also closely related to work I did with industry collaborators at Amazon AWS (summer 2019). Traditional techniques involve a preprocessing cost where all regular expressions are first converted to automata, resulting in a baseline-minimum latency even for rather simple string constraints. My evaluation demonstrated that by leveraging algebraic rewrite techniques on the regular expression constraints themselves, we can bypass this latency and improve performance over competing approaches. This research is theoretically grounded in a new extension of *symbolic regular expressions* and *Antimirov-style derivatives*, which enable the algebraic rewrites [10]. Concretely, for the typical case of Boolean combinations of constraints, I showed a 50% speedup over the best competing solver, and a 6x speedup over the previous competing version of Z3.

In the course of my research on string solving, I recognized a generic problem that has far-reaching use cases throughout past and current verification tools: *online state space exploration.* In general, SMT solvers can benefit

[2]These bounds are given mathematically in terms of number of variables (for space) and number of variable updates (for time), independent of the size of the input stream.

from re-usable highly efficient data structures (see, e.g., e-graphs); I have proposed and implemented *guided incremental digraphs* as a data structure for online state space exploration [11] (in submission; implemented in C++ as part of Z3, and a more developed implementation in Rust). This data structure was initially inspired by the theory of online graph algorithms (specifically, Tarjan's strong connected component maintenance); however, in typical SMT use cases, edges in the graph are not added in a worst-case pattern, but rather in a pattern that is guided by the solver. A *guided* incremental digraph is one where the edges arrive in a pattern typical of use cases, and I developed a more specialized algorithm for this case. While a worst-case analysis is elusive, I conjecture that it enjoys better complexity bounds, and demonstrated that it beats the best-known online graph algorithms in practice in our implementation.

More generally, I am interested in applications of online verification technology and runtime monitoring, and I have active work on verification and testing for programmable networks with collaborators at Penn. We are currently investigating fuzz testing in the dataplane for stateful P4 programs.

Future work

I am interested in all aspects of distributed and data processing systems in the online setting. I will highlight three major opportunities in this space. Most research is best done in collaboration; I am interested in collaborating with faculty in databases, distributed systems, networks, security, machine learning, and streaming algorithms. This area is well-funded by government and industrial agencies; my research has been supported by a 1.2M grant from the NSF (Award CCF 1763514).

Verified dataflow programming. Today's online dataflow programming frameworks make it too easy for software engineers to write incorrect code. To prevent this, I am interested in designing a library for *verified* dataflow. This effort would build on my work in correctness and performance guarantees for online applications, but would require extending to other families of guarantees (including fault tolerance) and to check the guarantees statically through an SMT or proof assistant back-end. The desired library has to (1) formally specify the required semantics (incorporating partial order requirements, faults, and performance bounds) and (2) expose a usable API with minimal proof burden on the end user. Similar to existing dataflow APIs, a program would be built by chaining together operators in sequence and in parallel (or even connected in cycles), but each of these operators should be annotated with formal requirements on its behavior, and these requirements should compose. One challenge is how to ensure that the formal requirements are met by the operator logic for user-defined, stateful operators: one approach would be to generate external *verification conditions* based on the formal requirements which verify the user's code is satisfactory.

Privacy and security. Researchers are only beginning to explore the question of what privacy and security mean for data-intensive online applications. In cases where most data is aggregated, compressed, or thrown away after an initial pass, what are the appropriate definitions of privacy, including differential privacy, and how can they be ensured? In particular, the differential privacy of streaming algorithms with constant or near-constant space usage should be investigated and quantified. This research would have a direct impact on real-world privacy guarantees if implemented in today's software. In the security of online applications, assumptions need to be articulated on input data (e.g., well-formed input), data rates, and node failures in order to provide security guarantees. I am also interested in the design and implementation of lightweight *specification languages* for privacy and security in online and cloud applications. All of these questions should be investigated from the spectrum from theory to practice: developing new theories and formal abstractions that are missing in this space, and demonstrating their implementation and utility through practical tools.

Query languages. Decades ago, systems and databases researchers envisioned a world where users implement application logic with simple queries over online data (e.g. using SQL and its variants). The increasing complexity of application logic has moved us steadily away from that goal; in practice, most online applications require very specialized development by distributed systems experts, and even programs written in higher-level frameworks often involve custom logic in the form of stateful functions. Today, we need higher-level abstractions which serve the same purpose but are *intuitive*, have *safe semantics* and are not arbitrary user-defined code. Allowing users to describe queries in *English* would be an even more ambitious goal interdisciplinary with natural language processing. I am currently investigating the design of query languages built on top of stream processing systems to make online applications such as distributed health monitoring systems simply a matter of writing a query and passing it to an online system.

References

[1] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G Ives, and Val Tannen. Data-trace types for distributed stream processing systems. *Programming Language Design and Implementation (PLDI)*, 2019.

[2] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Diffstream: differential output testing for stream processing programs. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2020.

[3] Rajeev Alur, Konstantinos Mamouras, Caleb Stanford, and Val Tannen. Interfaces for stream processing systems. *Invited paper, Principles of Modeling: Festschrift Symposium in honor of Edward A. Lee*, 2017.

[4] Caleb Stanford, Konstantinos Kallas, and Rajeev Alur. Correctness in stream processing: Challenges and opportunities. *Conference on Innovative Data Systems Research (CIDR)*, 2022.

[5] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Stream processing with dependency-guided synchronization. *Principles and Practice of Parallel Programming (PPoPP)*, 2022.

[6] Rajeev Alur, Phillip Hilliard, Zachary G Ives, Konstantinos Kallas, Konstantinos Mamouras, Filip Niksic, Caleb Stanford, Val Tannen, and Anton Xue. Synchronization schemas. *Invited paper, Principles of Database Systems (PODS)*, 2021.

[7] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Modular quantitative monitoring. *Principles of Programming Languages (POPL)*, 2019.

[8] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Automata-based stream processing. *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2017.

[9] Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. Streamable regular transductions. *Theoretical Computer Science (TCS)*, 2020.

[10] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. Symbolic boolean derivatives for efficiently solving extended regular expression constraints. *Programming Language Design and Implementation (PLDI)*, 2021.

[11] Caleb Stanford and Margus Veanes. Guided incremental dead state detection. *In submission*, 2021.