RESEARCH STATEMENT
ANDREW QUINN

Analytical mistakes are as old as civilization; the oldest known examples of script writing, Sumerian proto-cuneiform tablets from around 3000 BC, contain familiar errors, including incorrect arithmetic and pre-computer typos (i.e., 'lapsus') [3]. Today, software developers regularly make similar mistakes. Yet, instead of miscalculating barley quantities, software errors result in devastating financial cost, massive power outages, and even loss of life.

Placing the blame for software errors entirely on developers is easy, but overly simplistic. Today's software developers face a myriad of complex requirements, including functional correctness, fast performance, and strong security and privacy requirements. As a result, modern software projects are incredible feats of engineering that manage dozens of concurrent execution tasks, are comprised of millions of lines of code, and are written by hundreds of engineers. The complexity of our software means that simple solutions (e.g., code reviews, unit/integration tests) are insufficient for reliability. Instead, developers require better techniques, tools, and systems to reduce, detect, and resolve errors in software systems.

My research applies techniques from data-intensive computing (i.e., 'big-data') to build systems that detect, prevent, and resolve software errors. The software development life cycle and open source community produce data that has gone underutilized for reliability, including source code modifications, bug reports, program executions, and alternative solutions to the same and/or similar problems. Data-intensive techniques can process and gather insights from software data, which, when used in conjunction with techniques from programming languages, systems, and databases, can better track, understand, and predict software behavior. My research investigates how the resulting **data-intensive software reliability** techniques can solve fundamental problems in software reliability, including how we resolve subtle bugs in complex software, how we provide safety properties for execution behavior, and how we predict and react to potentially buggy code.

DISSERTATION RESEARCH

The systems in my dissertation employ two data-intensive techniques, large-scale parallelization and relational query models, to make it easier for developers to resolve their software errors. Today, developers resolve errors using roughly the same approach employed 50 years ago: by instrumenting their software to observe execution states and reproducing their errors. Instrumentation, whether performed by a tool or manually, must be meticulously minimized, since frequent observations hinder performance and may conceal bugs. Data-intensive techniques allow developers to perform significantly more observations without encountering these undesirable side-effects. **In effect, my dissertation enables a fundamentally more powerful debugging model, in which developers can easily express and perform expensive debugging tasks so that these tasks can be used during traditional interactive debugging sessions.**

The key idea behind my research is to use deterministic record and replay to treat an execution of a program as a massive immutable data object consisting of all program states (i.e., the memory and register values) reached by the execution. In this conceptual framework, debugging tasks (e.g., tracking the value of a variable) and dynamic analyses (e.g., data-race detection) can be expressed as queries that process data from an execution data object. My work accelerates this processing by introducing cluster-fueled analyses, which parallelize work across large-scale compute clusters, to accelerate highly sequential dynamic analyses (**JetStream** [4]) and general-purpose debugging tasks (**SledgeHammer** [5]). At cluster-scale, developers can interactively perform expensive debugging tasks, even tasks that were previously infeasible due to performance overhead (e.g., identifying the instruction that corrupts a large in-memory data-structure). In addition, my work simplifies the processing of execution data objects by introducing a relational query model, inspired by the database community, to express queries over an execution data object (**SteamDrill** [6]). This relational model allows developers to succinctly query entire program executions and has the potential to fundamentally change the way that developers approach debugging.

JETSTREAM—CLUSTER-SCALE PARALLELIZATION OF INFORMATION FLOW [4]

JetStream accelerates dynamic information flow tracking (DIFT), an important tool for understanding and troubleshooting program behavior across a variety of domains including security, debugging, security forensics, data provenance, and configuration troubleshooting. Also called taint-tracking, DIFT instruments an application binary to track data and/or control flow from global sources (e.g., program inputs) to global sinks (e.g., program outputs). Dynamic information flow has an overhead of two orders of magnitude, which precludes interactive uses in which a developer refines DIFT parameters (e.g., the sources and/or sinks) to understand software behavior. Parallelizing DIFT is a compelling idea, but prior approaches have struggled to even scale across multiprocessors because information flow tracks many sequential dependencies; DIFT has even been called "embarrassingly sequential" [7]. In contrast, JetStream scales to hundreds of cores by mitigating the scalability bottleneck of sequential dependencies through the use of two forms of parallelism. **By introducing a parallelization technique for sequential dynamic analyses, JetStream identifies a path towards interactivity for many such analyses, including memory error detection (e.g., AddressSantizier), data-race detection (e.g., ThreadSanitizer), and dynamic slicing.**

JetStream calculates DIFT in two phases and uses a separate form of parallelism in each phase. First, JetStream employs epoch parallelism, which time-slices an executable data object into chunks of execution called epochs. It calculates DIFT over each epoch in parallel by treating the starting and final memory states of an epoch as sources and sinks, respectively. In the second phase, JetStream merges data across epochs. Traditional approaches to merge data (i.e., tree-like techniques that recursively perform a pairwise merge) are insufficient for resolving cross-epoch dependencies, because many of the dependencies resolved between epochs do not derive from a source and never lead to a sink. So, JetStream uses a stream-processing technique that organizes epochs into a chain in the order of program execution. The system passes sources forward through the chain and sinks backward through the chain using a pipeline-style parallel algorithm in which multiple items are passed concurrently, similar to instruction pipelining in a processor. The pipeline-style algorithm allows JetStream to retain parallelism and pass a small amount of sequential information to drastically reduce wasted work. When executed on 128 cores, JetStream calculates DIFT even faster than the original execution of the program; the scalability limitations of dynamic instrumentation tools (e.g., Intel Pin) prevent JetStream from scaling beyond about 128 cores.

SLEDGEHAMMER—CLUSTER-FUELED DEBUGGING [5]

There are many powerful debugging techniques that are useful but too expensive to be interactive, including detailed logging and frequent checking of complex invariants (e.g., "is my btree balanced?"). Cluster-fueled debugging proposes the use of large compute clusters to accelerate these expensive debugging tasks. **By enabling a many order of magnitude increase in the number of compute resources that can be applied to debugging, cluster-fueled debugging allows developers to interactively perform heavyweight debugging tasks, even tasks which previously were infeasibly expensive.** Cluster-fueled debugging accelerates existing tools, such as retro-logging, which shows developers the output of modified logging code in a previously-recorded execution. In addition, cluster-fueled debugging provides enough compute power to enable new tools, such as continuous function evaluation, which allows developers to logically validate a complex invariant over the state of their software after every executed instruction.

Whereas prior debugging tools are designed to reduce runtime overhead, realizing a cluster-fueled debugger requires introducing scalability as a first-class design constraint. SledgeHammer, the first cluster-fueled debugger is replay-based (i.e., developers debug using a prior execution) and mirrors current debugging workflows, in which a developer adds logging and/or invariant checking instrumentation, and analyzes the output of the additional logic. SledgeHammer parallelizes debugging using a two phases. First, SledgeHammer employs epoch parallelism to instrument and inspect an execution using developer-specified debugging code in parallel. Existing instrumentation tools (e.g., Intel Pin) do not scale to thousands of cores, so SledgeHammer introduces a scalable approach that uses an undo-log to preserve deterministic replay. Next, SledgeHammer provides two frameworks for parallel analysis of debugging output. For tasks with sequential dependencies (e.g., tracking acquires/releases of locks), SledgeHammer provides stream analysis, which organizes developer-specified analysis in a chain and passes information forward and backward through the chain. Stream analysis becomes a bottleneck for non-sequential tasks (e.g., counting the accesses of a variable), so

SledgeHammer provides tree analysis, which organizes developer-specified analysis in a tree and passes information upwards through progressively smaller levels of the tree. SledgeHammer scales to thousands of nodes and returns results in a few seconds, even for extremely expensive tasks such as identifying the precise instruction that leads to corruption in a gigabyte-sized in-memory data-structure.

THE OMNITABLE MODEL—SIMPLIFIED DEBUGGING [6]

Most debugging tools are instrumentation based; they allow powerful access to program state at a single point in time, but are fundamentally stuck in the execution stream of the program. When using these tools to debug stateful errors, such as use-after-free bugs, a developer must implement complex logic to track the history of program state. In contrast, the OmniTable model exposes the entire history of an execution through a relational table abstraction and supports debugging queries through a SQL interface. By decoupling the specification of a debugging task from the program, the OmniTable query model also enables automated performance optimizations. **By allowing developers to easily and quickly query entire executions, the OmniTable model has the potential to fundamentally change the way that developers approach debugging.**

At the heart of the OmniTable query model is an OmniTable, a relational table containing the entire history of an execution. As a program executes, all user-level state (registers, memory values, etc.) is extracted after every instruction to populate an OmniTable. To aid with inspecting such a large table, my prototype, SteamDrill, supports SQL queries over an OmniTable and repurposes database concepts to simplify debugging tasks. For example, database-style views label events in an execution, such as the functions executed or the network packets received. SteamDrill resolves developer queries in parallel. The system decomposes a query into statements that observe execution state at a single point in time, which it converts into SledgeHammer debugging code and evaluates in parallel, and statements that aggregate such observations, which it executes on cluster-computing frameworks (e.g., Spark). In addition, SteamDrill accelerates queries using traditional database optimizations, such as predicate push-down, and a multiple-round replay approach that uses data that is inexpensive to observe (e.g., data about function invocations) to reduce the tracing of data that is expensive to observe (e.g., data about executed instruction).

The OmniTable model offers a trade-off between simplicity and expressibility: OmniTable queries are simpler than imperative (e.g., SledgeHammer) queries, but provide a more limited SQL interface compared to the flexibility of imperative interfaces. Nonetheless, I find that common debugging tasks can be succinctly expressed in the simple OmniTable model, requiring significantly fewer lines of code than imperative scripts for existing debugging tools.

FUTURE RESEARCH

In the future, I am interested in exploring software reliability techniques that detect and/or prevent errors by efficiently guaranteeing strong runtime safety properties without requiring large developer effort. Existing low-effort techniques (e.g., safe languages) provide generic safety guarantees (e.g., type safety) that have been useful for building reliable software. With the adoption of non-traditional hardware (e.g., persistent memory, FPGAs), kernel bypass interfaces, and legal regulations (e.g., European privacy laws), developers require systems that efficiently track and enforce strong, potentially application-specific, properties. Data-intensive techniques will be useful for scaling static analyses and optimizing for the common case, but, will need to be carefully employed as overzealous adoption can lead to safety violations. I am interested in exploring useful guarantees across a variety of domains and am currently investigating two classes of properties.

STATISTICAL SAFETY PROPERTIES

Statistical safety properties provide explicit statistical bounds for program correctness, such as "90% confidence that request parsing code will not crash". Today, developers attempt to obtain similar assurances through imprecise approaches, such as testing code coverage, or expensive-to-develop techniques, such as formal methods. In contrast, data-intensive techniques over prior executions, issue trackers, testing data, and source code modifications can provide low-effort statistically significant guarantees. These guarantees will allow developers to safely target the use of additional testing and/or tracing resources to the code regions that are most problematic.

DEVELOPER-SPECIFIED SAFETY PROPERTIES

I am exploring developer-specified safety properties, which allow developers to specify custom properties that are enforced at runtime. For example, a developer could enforce privacy requirements for compliance with European laws (i.e., GDPR) by specifying the acceptable information flows in their software, which a system would enforce at runtime. There are many questions surrounding custom properties that depend upon the application domain, for example: Can useful properties be supported? Do all properties require runtime tracing? When a violation is detected, should a system crash or can a runtime mitigation be performed? How will a system deal with imprecision? Data-intensive techniques over past executions, bug reports, and code modifications will be useful for scaling static analyses, optimizing for efficiency, and generating candidate runtime mitigations. I am mentoring several graduate students on projects that apply developer-specified safety properties to improve the reliability of emerging hardware deployments.

TOWARDS RELIABLE PERSISTENT MEMORY APPLICATIONS. Many storage systems have adopted persistent memory (PM), also called non-volatile memory, to accelerate durable storage. Current systems provide low-level imperative interfaces to encode logic required for crash-consistency. These interfaces are error-prone, since the crash-consistency logic is tightly coupled with application code and is restated many times throughout the program. I am mentoring a student on projects that use developer-specified safety properties to provide fundamentally more powerful PM interfaces. We first investigated the bugs in current interfaces by adopting symbolic execution to automatically find bug in PM applications [2] and building a system that provides non-regression guarantees for automatically fixing these bugs (i.e., provably, the system does not introduce new bugs) [1]. Currently, we are working on a declarative language for specifying application-specific crash-consistency safety properties which will be enforced at runtime. Data-intensive software reliability techniques will allow us to scale sound static analyses and optimize runtime tracing for paths that are most likely to be executed. I have worked on an NSF grant proposal based on this research.

TOWARDS RELIABLE HETEROGENEOUS SYSTEMS. Heterogeneous deployments, in which an application uses multiple types of compute resources (e.g., CPU+FPGA, CPU+GPU), have been adopted to accelerate applications in a post Moore's Law world. Today, components in a heterogeneous system that execute on different compute resources are developed independently, which makes it difficult to reason about system behavior across components. I am mentoring students on holistic debugging tools that use developer-specified safety properties to provide a unified cross-component interface for reasoning about heterogeneous system behavior. To achieve efficiency, holistic tools will benefit from data-intensive techniques over software data, such as past executions and testing data. To begin, we are investigating the security and correctness implications of current development approaches for heterogeneous systems.

References

[1] Ian Neal, **Andrew Quinn**, and Baris Kasikci. HIPPOCRATES: Healing persistent memory bugs without doing any harm. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 21)*, April 2021.

[2] Ian Neal, Ben Reeves, Ben Stoler, **Andrew Quinn**, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI 20)*, October 2020.

[3] H.J. Nissen, P. Damerow, R.K. Englund, R.K. Englund, R.K. Larsen, and P. Larsen. *Archaic Bookkeeping: Early Writing and Techniques of Economic Administration in the Ancient Near East*. University of Chicago Press, 1993.

[4] **Andrew Quinn**, David Devecsery, Peter M Chen, and Jason Flinn. Jetstream: Cluster-scale parallelization of information flow queries. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, November 2016.

[5] **Andrew Quinn**, Jason Flinn, and Michael Cafarella. Sledgehammer: Cluster-fueled debugging. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, October 2018.

[6] **Andrew Quinn**, Jason Flinn, and Michael Cafarella. You can't debug what you can't see: Expanding observability with the omnitable. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS 19)*, Bertinoro, Italy, May 2019.

[7] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA 08)*, Munich, Germany, June 2008.