Programming Technologies for Highly-Scalable Data-Intensive Applications

Gowtham Kaki
Purdue University gkaki@purdue.edu

In the formal study of computer programming, concurrency is inevitably associated with language-level abstractions such as threads, shared variables, and mutex locks. It is generally assumed that when programmers write concurrent programs, they reason about concurrency in terms of one or several such abstractions. The contemporary reality, however, doesnâĂŹt quite fit this world-view. Everyday around the world, millions of programmers write concurrent programs unwittingly – without manifestly using concurrency abstractions, and without reasoning explicitly about concurrency-related issues. In fact, the common-case concurrent programming today is done by professional software engineers building web applications that rely on a (possibly distributed) database system to manage large amounts of data constituting the application state. Such data-centric applications frequently serve hundreds to millions of clients simultaneously, yet they are not considered concurrent programs in formal sense as they are expected to delegate concurrency control to the underlying database. The database in return is expected to provide high-level assurances, such as Transaction Serializability and Sequential Consistency, that let application developers safely ignore concurrency-related concerns, while still enjoying its benefits. Such clear separation of concerns indeed leads to an elegant software design that has been widely adopted, and is still taught in the classrooms today.

In the recent years however, the radical evolution of system architectures coupled with ever-increasing expectations of scalability and availability has left databases playing the catchup game with no choice but to gradually weaken the high-level assurances. The result is a possible spillover of concurrency effects into application space, and unsuspecting programmers being made party to a difficult tradeoff between performance and ease of reasoning. Making an uninformed trade-off under this circumstance may lead to unpleasant consequences in form of concurrency bugs that, at best, impede application usability and, at worst, introduce serious security vulnerabilities that threaten the existence of the software enterprise. The threat is aggravated in the context of distributed applications with replicated state, as concurrent executions under asynchronous state replication are known to yield counterintuitive program behaviors. The lack of suitable programming models and a non-existent tool support has made it hard to guarantee the correctness of even the simplest of data-centric applications.

Research Highlights

My research has focused on solving the aforementioned problem by studying data-centric applications as concurrent programs in their own right. I realized that the formal models of concurrency used conventionally in the programming language research community are either too simple to

adequately address the challenges faced by the data-centric applications, or too complex to not take advantage of their specific characteristics that make them amenable to automated reasoning. Consequently, the conventional reasoning, verification and synthesis techniques are either not applicable, or not nearly as effective on these applications as they are on the ordinary concurrent programs. To address this shortcoming, I have developed new formal models, proof systems, automation techniques and verification tools that collectively push the envelope in concurrent and distributed program verification to reach contemporary data-centric applications. In particular, my research has resulted in the development of (a). Formal theoretical underpinnings of non-serializable weakly-isolated transactions, which enabled mechanical verification of complex database applications under concurrent execution, (b). Succinct operational models of weakly-consistent distributed data stores, which resulted in novel verification and modelchecking techniques for geo-distributed applications, and (c). Foundational principles that enable systematic derivation of distributed data structures from sequential data structures, which let ordinary language libraries to be reused safely in the context of a distributed application. The relevance of my research to the theory and practice of contemporary data-centric software engineering has been acknowledged by the software industry, in particular Google, who funded my later work in this area via a generous research fellowship. In the following sections, I elaborate on the research efforts that I mentioned briefly above, and present my vision for the future of computing that would inform my research for the years to come.

Full-Functional Verification of Database-Backed Applications

A vast majority of data-centric applications today are supported by off-the-shelf relational databases, such as MySQL and PostgreSQL, which do not enforce the serializability of transactions, either by default or ever. Instead, to accentuate performance, such databases employ weakly-isolated transactions âĂŞ transactions that are not fully isolated from the effects of concurrent transactions, resulting in concurrency anomalies. However not all concurrency anomalies manifest as safety violations, and when they donâĂŹt, it is beneficial to admit them, for it lets the application scale using non-serializable transactions. Knowing if a database-induced concurrency anomaly is harmful or innocuous in the context of an application is therefore a problem facing every application developer. The problem is further complicated due to the arbitrary stratification of weak isolation into various isolation levels defined only by their implementation details. Facilitating rigorous, yet automated, reasoning about program correctness under weak isolation is therefore challenging.

My research proposed a language-based solution to the problem. As a first step, I demonstrate that the semantics of various seemingly-unrelated weak isolation levels implemented on commercial databases can be captured as simple logical assertions over the states visited by an abstract database machine executing a transactional program. This observation crucially lets me lift the specification vocabulary of weak isolation from that of abstract low-level traces, as was standard then, to that of concrete high-level program states visible to programmers and reasoning tools. Based on this novel construction, I define a Rely-Guarantee program logic to reason about database transactions operationally, akin to ordinary concurrent programs, while reasoning about their isolation properties declaratively. In concrete terms, this means that a developer does not have to understand how weak isolation is implemented, nor does she need to understand how concurrency anomalies manifest in her application in order to determine whether the application runs as intended under weak isolation. All she needs is a declarative specification of weak isolation, thus

making the verification effort commensurate with that required for serializable transactions, i.e., ordinary concurrent programs. I have exploited this observation to build a tool called ACIDifier that can automatically analyze database applications with weakly-isolated transactions, detecting vulnerabilities caused by weak isolation, if any, and certifying the application correct otherwise. As we demonstrate in [**?**], ACIDifier works with succinct specifications of MySQL and PostgreSQL (as against their mind-numbingly large implementations) to not only demonstrate the vulnerabilities that real-world database applications exhibit at weaker isolation configurations, but also infer the weakest (hence most scalable) isolation configurations at which it is safe to run these applications.

Automated Bounded Verification of Geo-Distributed Applications

ACIDifier assumes a typical relational database with weakly-isolated transactions and strongly (or sequentially) consistent reads and writes. Strong consistency is however impossible in a geo-distributed setting where application state is asynchronously replicated across multiple geographical locations to decrease latency and improve service availability. In such conditions, application has to choose between various levels of weak consistency to assign to various operations so that application works as intended even as it accrues the benefits of replication. Considering the complexity of the problem, support for automated reasoning is an imperative.

In [**?**], I propose a novel symbolic execution technique, called Q9, that can automatically analyze distributed âĂIJNoSQLâĂİ applications under weak consistency to find concurrency bugs, and suggest repairs by determining the weakest consistency configuration needed to eliminate them. Q9 is based on a novel operational model of a replicated data store that is simple, yet highly expressive âĂŞ It can emulate the behavior of an application on the store under a range of consistency models used in practice. Like in ACIDifier, the reasoning is operational for application semantics and declarative for consistency semantics; we have previously shown that such declarative reasoning does scale in practice, and it even facilitates highly efficient execution of large distributed applications on top of off-the-shelf NoSQL data stores [2]. Through iterative strengthening of consistency Q9 can progressively repair programs and eventually verify them under a finite concurrency bound. The concurrency bound roughly translates to the number of messages in flight at any point during the exection. Thus, by using Q9 under a sufficiently large concurrency bound, one can verify geo-distributed applications at essentially zero cost. The experiments reported in [**?**] demonstrate the efficacy of Q9 over a range of distributed applications, including the TPC-C/E benchmarks ported to NoSQL.

Principled Derivation of Distributed Data Structures

In [1] I propose a dependent type system with decidable type checking, called CATALYST, that is tailor-made for verifying data structure transformations such as sorting, tree balancing etc. At the center of CATALYST is the observation that any data structure can be represented canonically using just sets and relations. For instance, the canonical representation of a list contains (a). A unary membership relation that defines the set of its elements, and (b). A binary order relation that captures the left-to-right order of elements in the list. By decomposing to relations, one can reason about the relationship between a pair of structures, e.g., a binary tree and a list, uniformly in the decidable relational domain. In [1], I show that this approach generalizes elegantly to polymorphic data structures and higher-order functions without losing decidability, thus yielding an automatic

reasoning framework for verifying complex data structure transformations of the kind often found in compilers. The relevance to compiler correctness has also been acknowledged by DARPA, which funded the SafeDocs proposal to apply CATALYST-style reasoning at scale to verify parsers for document formats, such as JSON and XML.

In [**?**], I extend the utility of relational decomposition beyond sequential data structure verification to distributed data structure synthesis. I propose a new class of distributed data types based on the premise that the ability to merge concurrent instances of the type is what distinguishes a distributed data type from an ordinary type. A distributed data type is therefore a mergeable type, i.e., a data type extended with a merge function. In [**?**], I demonstrate that such merge functions can be derived systematically from first principles using CATALYST-style relational decompositions, thus paving the way for (mostly-) automated synthesis of distributed data structures from sequential ones. Using the relational decomposition approach, we were able to automatically verify a number of functional data structure libraries in [1], and derive their correct-by-construction distributed variants in [**?**].

Future Work

There are a number of avenues of future research I would like to pursue, which, I believe, will become increasingly relevant as the nature of computing evolves together with its relationship to the society.

Empirically certified formal models of complex software systems: In [**?**] and [**?**], I demonstrate the utility of employing simple operational models of (distributed) databases in lieu of their complex implementations to reason about application safety. Constructing formal models of complex software systems is a challenging task considering that such models have to abstract away low-level phenomena, such as race conditions and network partitions, to simplify reasoning, yet remain faithful to the original implementation. The conventional language-based approach to ensure fidelity of a software model is to co-develop the model and the implementation, along with a proof that the latter is simply a refinement of the former (e.g., CompCert TSO and IronFleet). This approach is however infeasible for complex real-world software that accrue code and optimizations resulting from decades of engineering effort. There is a need for alternative approaches that do not require formal reasoning about the implementation, yet ensure high level of confidence in the fidelity of the model. Inspiration can be drawn from the world of weak memory models where fidelity is ensured through âĂIJlitmus testsâĂİ, however it is far from clear if the idea of litmus tests can be generalized, or if it is possible to quantify the confidence gained by employing a suite of litmus tests. Considering that the complexity of real-world computational processes is increasingly rivaling that of physical processes, there is a need to conceive scientific approaches for building and empirically evaluating models of computation that facilitate precise yet tractable reasoning. This shall be an area of focus for my future research.

Intent-defined portable data-centric applications: Relational databases trace their origins to E.F. CoddâĂŹs insight that human data is best organized as finite relations (i.e., sets of tuples), and queries on such data are best understood in terms of an algebra over such relations. It should therefore come as no surprise that I found monadic set logic to be an ideal vehicle to reason about transactions on relational databases in [**?**]. Reduction to a relational logic has indeed been a recurring trope in my work, both in theory and in practice, as I built several reasoning frameworks and verification tools to establish the safety of data-centric computations of various kinds [1**? ? ?**].

The indispensability of relational logic to reasoning about data-centric computations leads me to believe that perhaps the intent behind such computations is expressible, devoid of any implementation concerns, in a relational language. It should be possible subsequently to concretize the intent in the context of a specific machine semantics as an efficient implementation for that machine. In particular, I am interested in evaluating datalog, or a variant thereof, as a means of capturing the intent behind a data-centric computation in form of an executable specification. I conjecture that there exist a series of intent-preserving transformations, parameterized on database semantics, that can compile a datalog specification into an ordinary program that runs efficiently on the given database. Successful demonstration of this conjecture would let developers build portable data-centric applications that do not make assumptions about the storage layerâĂŹs API and semantics, hence are not locked into a specific vendorâĂŹs (e.g., AmazonâĂŹs or MicrosoftâĂŹs) database or cloud ecosystem. Realizing such an intent-defined data-centric application stack is going to be another focus of my future research.

Programming models for fully-decentralized and privacy-conscious internet: While initially conceived as a decentralized network of participating servers, internet has slowly morphed into a (virtually) centralized network in control of few public and private entities. In the recent years, ambitious initiatives have sought to counteract this development by promoting full decentralization in application and network protocol design (e.g., Blockchain and IPFS), and by putting people are in charge of their data in their own premises. Well-intended as they may be, such initiatives nonetheless have systemic issues for there are currently no languages and tools that can bridge the wide chasm between the limited human cognition of the programmer, and planet-scale workings of the underlying system. The result is a new class of distributed applications that are fragile, vulnerable, and highly inefficient. This is a problem to which I am interested in finding language-based solutions. In [**?**], I sketch a programming model based on distributed version control systems (e.g., Git), which I show can significantly reduce the cognitive burden involved in building highly decentralized applications. In the near future, my focus would be on laying the theoretical foundations of this programming model to help determine its guarantees and limitations. Imperative to this goal is the task of placing Git-style version control systems on a sound formal footing, whose benefits, I believe, would extend well beyond data-centric applications.

References

[1] Gowtham Kaki and Suresh Jagannathan. A Relational Framework for Higher-order Shape Analysis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 311–324, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628159. URL http://doi.acm.org/10.1145/2628136.2628159.

[2] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 413–424, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737981. URL http://doi.acm.org/10.1145/2737924.2737981.