Università della Svizzera italiana

Faculty of Informatics

Research Prospectus

July 24, 2014

Declarative Data Consistency

Nosheen Zaza

Abstract

The utilization of relaxed data consistency models has increased in recent years, especially in distributed applications, where maintaining strong consistency comes on the cost of availability and performance. Developing applications with correct consistency properties is difficult, because current development tools do not provide abstractions that enable programmers to write their code in a replicated, asynchronous environment naturally. In this research prospectus, we survey various state of the art techniques and tools proposed to facilitate consistency management, and describe our vision of a data-centric, asynchronous framework wen intend to develop to facilitate distributed computing with replicated data.

Research Advisor
Prof. Nate Nystrom

Academic Advisor
Prof. Nate Nystrom

Review Committee
Prof. Committee Member1, Prof. Committee Member2

Research Advisor's approval (Prof. Nate Nystrom): Date: .....

PhD Director's approval (Prof. Stefan Wolf): Date: .....

1 Introduction and Problem Domain

Relaxed data consistency models are being successfully employed in frameworks on which various types of applications are built. The internet has brought many use-cases where consistency can be relaxed with little compromise. As a motivating example, consider a shopping application that replicates product data. When a client requests a list of all products that have 5-star ratings, it is not necessary to query all replicas in order to get the most up-to-date rating data. In fact, requiring acknowledgments from all replicas would cause unnecessary system load for a task that is inherently subjective. Real-world use-cases include metrics collection and product recommendations at eBay [26], and inbox search at facebook [19]. Both employ Cassandra [19], an eventually-consistent [31] storage backend.

Relaxed consistency is rarely sufficient on its own for the requirements of a complete application. Typically, components requiring different consistency policies co-exist and operate. Great care must be taken to ensure correct program semantics in this case. Continuing with the previous example, consider when the client adds an item from the list of items with 5-star ratings to his shopping cart. Because the list was not constructed by querying all replicas, it is possible that it includes some items that are no longer in stock. If the programmer forgets to query all replicas to ensure that the selected item is in stock prior to check out. The client will end up buying a non-existing item.

The problem of maintaining data consistency is not exclusive to distributed systems with explicit replication and application-level weak consistency. It is well known in concurrent programming literature. On the lowest level, data is replicated under the hood among main memory, caches and registers, and many common CPU architectures, such ARM-based processors, used in many handheld devices provide only a weak memory consistency model [9], on which applications with typically strong consistency requirements are built. Yet maintaining data consistency does not happen under the hood as well, on the contrary it is the heavy responsibility of the programmer, who must ensure that:

1. The program does not violate consistency.
2. The compiler will not reorder the program in a way that violates consistency.
3. The CPU will not reorder the program in a way that violates consistency.

Programming languages and software frameworks designers have provided many tools to facilitate consistency management, many of which can be adapted to a distributed environment with arguably more potential of success, because replication and consistency requirements are explicit and on application-level. Furthermore, the overhead of some techniques is better tolerated in a distributed environment. We have been investigating a declarative, data-centric approach to consistency management, originally proposed by Vaziri, Tip and Dolby [30], and we believe that it has the potential to be adapted and enriched to provide a powerful framework for programmers to declaratively express not only atomicity as originally proposed, but also various consistency constraints. Based on these constraints, it is possible and delegate many parts of consistency management logic to the compiler and runtime. In this research prospectus, I start by describing the core ideas of this approach, then show how it can be adapted to support multiple consistency levels to naturally express common distributed computing use cases. Then I will list related work on consistency constraints and program properties that we believe are useful for programmers to express declaratively. After that I will overview our research directions and challenges, and how we plan to conduct our work.

2 Data-Centric Constraints

When programming in a shared-memory environment, programmers must ensure that concurrent accesses to the same memory location do not lead to data races and inconsistent results. This is traditionally achieved by ensuring that instructions accessing shared data acquire locks, or are within a transaction. Such approaches have an operational, control centric flavor. The problem is that data races may occur if the programmer forgets synchronize even a single control flow path.

2.1 Atomic Sets

Vaziri et al. outlined a solution in 2006 [30], that associates synchronization constraints with data, and based on these constraints, the compiler generates synchronization code for all operations that access this data. Figure 1(counter example from the paper) shows an example taken from that paper written in Java. It involves incrementing a counter concurrently. An atomic set is declared, which is a memory region to which all accesses must be synchronized, then class fields that need to be protected are added to this set. The compiler then synchronizes all public class methods that interact with the atomic set. The system also allows atomic sets that span multiple objects, as well as other features.

2.2 Adding Replicated Data Consistency Constraints

In many replicated key-value storage systems, such as Cassandra [19], Dynamo [16], and Riak [27], clients can tune consistency by specifying the number of replicas on which an operation must succeed before returning an acknowledgment to the client application. In Cassandra, consistency is configured on the level of cluster, data center, or individual I/O operations. Going back to our online shopping example, instead of relying solely on the programmer to choose the correct consistency level every time he writes code that interacts with the contents of the shopping cart, we can attach a policy to shopping cart data region, stating that all operations must have the consistency level ALL, meaning all replicas must acknowledge the successful completion of the operation. We gain two benefits following this data-centric approach: it is now possible to infer the correct consistency level for all operations dealing with the shopping cart data, and if the programmer specifies a lower consistency level, such as ONE, which only requires an acknowledgment from one replica, it is possible now to show an error/warning stating that this is probably not what is meant to be done. Furthermore, Since the list or 5-star products was read with consistency level ONE, it represents a memory region that is not strongly consistent. It will not be possible to assign an item form this region to the shopping cart, as it represents a forbidden information flow path.

3 Research Prospectives

Inspired by the above mentioned work, we aim to present a programming environment that enables developing distributed applications with better data abstractions. We have identified the following aspects that we need to investigate:

- Defining data region abstractions.
- Grouping and classifying operations on data.
- Identifying data and operation properties and constraints that would be useful to infer/express.

Our aim is to create a development framework that enables programmers to code in the presence of asynchronous operations and data replication in more intuitive ways than currently offered.

In the following sections, we describe each research problem, and show related work.

3.1 Data Region Abstractions

We need to determine which data region abstraction(s) to employ, because choosing the right abstractions eases defining constraints on data. Many data abstraction in various systems come with baked-in consistency constraints. Having good default definitions of such constraints often relieves the programmer from having to define them himself. We also believe that having replication defined as an explicit property of data makes reasoning about it easier for the programmer.The following list contains various data abstractions and built-in consistency guarantees.

1. The Jave memory model [25] ensures atomic semantics for one-word primitive types, such as `int`. `long` and `double` are not required to be atomic by the language specification.
2. In many database systems, row updates are atomic. For instance, In MySQL [23], the MYISAM storage engine locks tables when an update is issued, while InnoDB engine locks the row being updated. In Cassandra, rows are atomic units.
3. Objects in object oriented languages span over multiple memory locations, and group together data and operations [28]. Objects can be arbitrarily nested and passed around to form complex graphs, and accesses to object data are typically unordered and synchronous. Objects present a difficult case for constraint attachment, because many threads can have references to different parts of an object and update them concurrently, possibly making temporary invariant violation visible among threads. Even if each object in the graph is properly synchronized, the collection of objects might still have incorrect semantics [4]. Furthermore, it is hard to reason about nesting of objects bearing conflicting constraints. There is a large body of ongoing research on ownership types and alias management that aim to restrict arbitrary referencing of objects [22, 10, 14].
4. In the current implementation of atomic sets [11], Vaziri el al. employ a simple ownership type system that ensures objects are only accessed through their owners, without this constraint, it would be hard to perform static analysis. Atomic sets presented in this work are static data containers that may span the data of multiple objects. data within an atomic set share the same lock. However atomic sets only express “shallow-locking”, if nested objects must be locked, they need to be explicitly added to the containing object’s atomic set. It is also not possible to dynamically grow or shrink atomic sets.

5. X10 [8] supports places [12] a distribution-friendly data abstraction than encapsulate the binding of activities and globally addressable memory. An activity may synchronously access data items only at the place in which it is running. It may atomically update one or more data items, but only in the current place. If it becomes necessary to read or modify an object at some other place, a place-shifting operation can be used. This approach ensures that expensive operations (e.g., those which require communication) are readily visible in the code. The implicit constraint in this system is the locality of data.

6. DPJ [6] regions are hierarchal heap partitions used to disambiguate accesses to distinct objects, as well as distinct parts of the same object. A type checker then ensures that there are no conflicting accesses to overlapping memory regions between concurrent tasks.

7. The actor model [1] differs in two fundamental ways from the object model: all accesses to actor state are serial and asynchronous. This model is a better fit for a networked environment, and is gaining popularity with languages such as Erlang [3] and Scala [24] employing it successfully. Typically, actors must not expose their internal data by any means other than message passing. In practice, systems such as such as Akka [7] do not enforce that, and rely on the programmer to maintain actor encapsulation. Because of their properties, actors are easily distributable, however it is hard to introduce concurrency within an actor. Habanero Java and Habanero Scala [18] enable controlled concurrency within an actor.

8. Coqa [20] is a programming language that supports different kinds of objects with properties similar to traditional objects or actors.

3.2 Grouping and Classifying Operations

As there are wrappers to data, there are wrappers to operations. A database transaction, or a method body are two commonly used wrappers. Investigating such wrappers is an important aspect to our work, because we need to attach constraints to operations as well, and sometimes it is more convenient to infer the consistency of a memory region from the constraints of the operation interacting with it or from its properties.

1. In databases, isolation levels can be attached to transactions [5]. The consistency of dataset resulting from a transaction would depend of the specified isolation level.

2. A problem with object-oriented systems is that operations on data are grouped as functions, function bindings in many systems are determined at runtime. Functions can perform arbitrary reads and writes, and because they are dynamically bound, it is hard to tell at compile time which pieces of data are affected by a function. Effect systems allow the programmer to declaratively attach effects to methods, and some of them can infer the effects partially [21]. Cormac and Shaz propose an effect system that declares atomicity as a property of methods [13].

3. Software transactional memory [29] borrows the ideas of database transactions and apply them to generic programs. Such systems are known for they high overhead because of the delay of creating operation logs and performing roll-backs. Not all software systems can tolerate such overheads. However transactions are easier to reason about than locking, and they are composable.

In conclusion, if the coding environment is aware of operation effects, this can allow for more concurrency. This of course comes with the overhead of encoding the effects by the programmer.

3.3 Consistency Constraints

We have already presented some constraints that can be attached to data or operations, such as read/write effects, isolation levels and atomicity. We now list some other constraints that enable better control of consistency by making some program properties explicit to the programmer or the development environment.

1. Monotonicity: this property means that the final order or content of input will never cause any earlier output to be “revoked” once it has been generated. This means that any order of operation generates the same consistent result. Bloom is a distributed language that exploits this property by providing programmers with order-independent abstractions, which encourages a programming style that minimizes coordination requirements and guarantees consistency. [2]

2. Latency and failure: X10, Scala and other languages supports futures [15]. These constructs materialize latency and failure as datatypes and operation effects. Their advantage is that they make these code properties

explicit, In distributed environment, failure of operations should always be considered. Multicore programers need not be concerned with a CPU core going down or instructions getting lost between system buses, but network developers always face node failure and message loss. Futures make failure handling a first-class action in a distributed language. (@Nate this is related but not so much related to consistency directly, keep it or drop it?)

3. Locality: Loci [32] is a pluggable type system that enables programmers to make thread locality of data explicit and statically checkable. This has many advantages: programmers need not to worry about the consistency of data proved to be local, and compilers can perform more optimizations with such data.

4. Commutativity and Inverses: DPJ and the transactional boosting method [17] allow programmers to specify commutative and inverse methods. Commutativity permits re-ordering of method calls while ensuring deterministic semantics, and inverses allow creating more compact rollback logs for transactional memory.

4 Research Directions

Current development tools need to be extended in order to support programming patterns of distributed systems. The declarative approach of programming is the new trend, and programming languages such as Scala facilitate, and encourage programmers to create declarative, domain specific languages tailored for the needs of a class of applications. An asynchronous model of computing is favored in distributed applications, and programming languages are starting to natively support this model. We plan to build a declarative programming language constructs that enable data-centric, global reasoning of consistency, that naturally supports replication, asynchronous computing and materializes latency, failure and streaming. Because the object model is still very commonly used, we also plan to focus on implementing our model in an object-oriented environment. This will require us to build ownership and effect systems to tame this model and make it suitable for use under our system. We also plan to further investigate code properties and constraints, other than the ones we mentioned earlier to enrich our system and make it easier to use.

5 Achievements

So far I have achieved the following:

- Implemented atomic sets [11] in Scala.
- Attended a workshop on software correctness and reliability at ETH Zurich.
- Attended the following courses: Bugs 2013 (4 ECTS) and Introduction to Ph.D. studies (2 ECTS)
- Attended ECOOP 2014 with co-located workshops and summer schools.
- Served as a TA for the following courses: Programming Fundamentals 3 (4 ECTS) and Automata and Formal Languages (2 ECTS).

References

[1] G. A. Agha. Actors: a model of concurrent computation in distributed systems. 1985.

[2] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260. Citeseer, 2011.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent programming in erlang. 1993.

[4] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[5] P. A. Bernstein. Sql isolation levels. In *Encyclopedia of Database Systems*, pages 2761–2762. Springer, 2009.

[6] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 4–4. USENIX Association, 2009.

[7] J. Boner. Akka. `http://akka.io/`, 2014.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[9] N. Chong and S. Ishtiaq. Reasoning about the arm weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: Held in Conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, MSPC '08, pages 16–19, New York, NY, USA, 2008. ACM.

[10] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. *Minimal ownership for active objects*. Springer, 2008.

[11] J. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek. A data-centric approach to synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):4, 2012.

[12] V. S. et al. X10 language specification. `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`, 2014.

[13] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *ACM SIGPLAN Notices*, volume 38, pages 338–349. ACM, 2003.

[14] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP 2010–Object-Oriented Programming*, pages 354–378. Springer, 2010.

[15] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic. Futures and promises. *URL http://docs. scala-lang. org/overviews/core/futures. html*.

[16] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: AmazonâĂŹs highly available key-value store. In *In Proc. SOSP*. Citeseer, 2007.

[17] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 2008.

[18] S. M. Imam and V. Sarkar. Integrating task parallelism with actors. In *ACM SIGPLAN Notices*, volume 47, pages 753–772. ACM, 2012.

[19] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[20] Y. D. Liu, X. Lu, and S. F. Smith. Coqa: Concurrent objects with quantized atomicity. In *Compiler Construction*, pages 260–275. Springer, 2008.

[21] D. Marino and T. Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, pages 39–50. ACM, 2009.

[22] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In *Programming Languages and Fundamentals of Programming*, volume 263. Fernuniversität Hagen, 1999.

[23] MySQL. Mysql. `http://dev.mysql.com/`, 2014.

[24] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, 2004.

[25] Oracle. Threads and locks. `http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4`, 2014.

[26] F. Qu and A. Jambhekar. Cassandra at ebay scale. `http://www.slideshare.net/planetcassandra/5-feng-qu`, 2013.

[27] E. Redmond and J. R. Wilson. *Seven databases in seven weeks: a guide to modern databases and the NoSQL movement*. Pragmatic Bookshelf, 2012.

[28] T. Rentsch. Object oriented programming. *ACM Sigplan Notices*, 17(9):51–57, 1982.

[29] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[30] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. *SIGPLAN Not.*, 41(1):334–345, Jan. 2006.

[31] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[32] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for java. In *ECOOP 2009–Object-Oriented Programming*, pages 445–469. Springer, 2009.