CSCI 2980

Master's Project Report

How to Reason about Correctness of Programs Designed for Non-Volatile Memory?

Submitted in partial fulfillment of
the requirements for the award of the degree of

Master of Science
in
Computer Science

Submitted by

Kartik Singhal

kartik@cs.brown.edu

Under the guidance of

Maurice Herlihy

Department of Computer Science
Brown University
Providence, RI, USA – 02912

Spring 2017

Abstract

Traditional storage stack necessitates a separate data format for the persistence of in-memory data structures, requires additional code for conversion to that data format and wastes a lot of CPU time. Upcoming byte-addressable non-volatile memory (NVM) technologies such as memristors or phase change memory offer an opportunity to rethink how code interacts with persistent data. Researchers have come up with a variety of programming models to make effective use of NVM but, unfortunately, it is considered hard to reason about the safety properties provided by these models.

In this report, we look at existing work in a somewhat related field of formal reasoning about the correctness of concurrent software and discuss whether those techniques can be applied to software designed for NVM or persistent memory.

We also document our design of a concurrent graph data structure for non-volatile memory which offers crash-resilience with the help of Atlas programming model.

Contents

Chapter 1

Introduction

An ongoing development in the world of storage is the proposed non-volatile memory (NVRAM) that combines the characteristics of storage hierarchy involving DRAM (latency) and disk based storage (persistence). NVRAM offers higher density, lower power and comparable latency as DRAM while providing persistence with byte addressability as opposed to block-based storage offered by solid state disks and hard disks. A clear advantage is being able to store data structures in a single data format instead of serializing them to disk and eliminate both time and effort required to write and maintain code that keeps the two formats consistent.

There is a lot of ongoing research to define the semantics of how to program for persistent memory. A widely accepted system model consists of traditional DRAM (transient) and NVRAM (persistent) being used together as primary storage with the address space divided between the two. CPU registers and the cache hierarchy remain transient. This hierarchy with persistence available at a lower latency than traditional secondary storage comes with its own interesting problems.

To illustrate, we borrow an example from the Atlas paper[CBB14]:

```
1: t = pmalloc(...); // allocate persistent memory
2: *t = ...;         // initialize
3: l.lock(); ptr_to_persistent->x = t; l.unlock(); // publish
```

In this code, if the program crashes when store to persistent memory in line 3 has been written to NVRAM but the store to persistent memory in line 2 is still in cache, then the data structure in NVRAM will be in inconsistent state. On recovery, the program will see uninitialized data on dereference of `x`. To prevent this problem, it is easy to see that updates to NVRAM should be done atomically and in the correct order with respect to other updates. Unfortunately, optimizations implemented in compilers and hardware neither ensure atomicity nor exact program ordering. This is the same problem that programmers writing scalable concurrent algorithms face.

Programming models such as Mnemosyne, NV-Heaps, Atlas, NVML, NVL-C offer trade-offs between safety guarantees, performance and familiarity of API for ease of use[Bal16]. Soon, the state of art is expected to reach a consensus and some model built on top of these may become the foundation for writing future persistent programs. But no work has been attempted in our knowledge to formally reason about the correctness of any of these programming models. If these foundations are not strong enough, we may face a deluge of bugs in the applications written for persistent memory reminiscent of the bugs that are found in existing software especially in lower level code that requires similar subtle reasoning such as racy concurrent software.

We focus on the Atlas[CBB14] programming model which exposes C/C++ APIs for writing persistent programs and provides an all or nothing guarantee around critical sections. In other words, with minimal changes it is possible to transform multi-threaded code to crash-resilient code. However, Atlas cannot provide such guarantees for lock-free programs. To get familiar with Atlas, we designed a concurrent graph data structure optimized for Atlas, ie, limiting ourselves to using lock based synchronization.

In the rest of this report, we present the design of our concurrent graph data structure in the next chapter and then move on to the discussion about existing work in reasoning about concurrent software and our understanding of what needs to be done to apply formal methods for proving correctness of persistent programs.

Chapter 2

A Concurrent Graph Data Structure for Non-Volatile Memory

2.1 Problem Overview

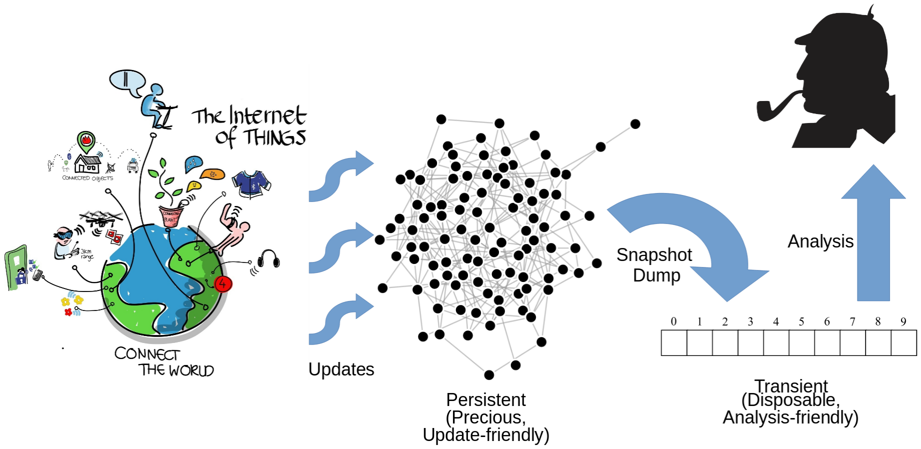

Figure 2.1: Overview

Imagine a large analytics system (see Figure 2.1) that receives input from all over the world in the form of data collected from a variety of Internet of Things (IoT) devices. To store such data, a graph data structure offers the most flexibility and as such this system uses a graph as its primary data store. This graph continuously keeps on changing as the updates keep coming in from various input sources. Doing any analysis on this graph is a hard problem, doing a real time analysis even harder. Existing parallel graph analysis frameworks such as the Parallel Boost Graph Library, Galois, or Ligra either do not support analyses on dynamically changing graphs, or do so inefficiently.

A reasonable current practice involves taking read-only snapshot for analysis. However, there are certain problems associated with this practice: 1) taking a snapshot of such a large graph dataset can take a long time, which can slow down analysts, 2) a consistent snapshot requires taking a global lock on the complete graph which prevents any updates to be persisted on the graph, ie, updates need to be throttled, 3) there is no feasible way to record and replay large input streams during snapshot, 4) in case, a crash occurs during any operation in the system storing the central graph data structure, there is no resilience-mechanism.

In this work, we propose a way to make this sensible practice of taking snapshots faster while ensuring crash-resilience with the help of non-volatile (persistent) memory. We design our system to make use of the Atlas programming model that provides durability guarantees for persistent memory.

2.2 Design and Implementation

2.2.1 System Architecture

Consider the Figure 2.1 again. Our design involves storing the central graph data structure in persistent memory because it is precious and keep it update-friendly so that there is minimal cost involved in persisting incoming inputs to the graph. The snapshots taken from this central graph may be stored on transient memory (DRAM) as they are disposable, read-only and optimized for analysis. We use Compressed Sparse Row (CSR) format for snapshots, which preserves locality information on the graph (useful for running most analyses) and is time-and-space efficient. This approach allows concurrent updates to the graph while a snapshot is in progress.

Atlas runtime ensures crash resilience for the persistent graph, ie, recover to a consistent state in case of system failure.

The challenge is in obtaining efficient and consistent snapshot dumps concurrently without throttling updates to the persistent graph store.

2.2.2 Data Structure Design

Refer Figure 2.2 which describes the design of the central persistent graph data structure. The algorithm that operates on this data structure involves three global states – NORMAL, SNAPSHOT and CLEANUP. This state is stored

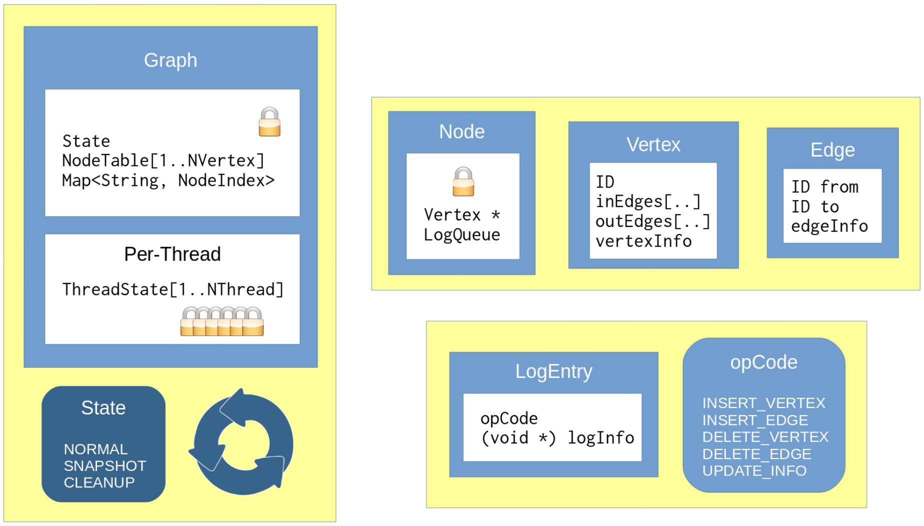

Figure 2.2: Data Structure Design

both in a global `State` variable which is protected by a mutex lock and is replicated in per-thread `ThreadState` variable. Further, the nodes are stored in a globally accessible array. Each entry in the array represents a node in the graph and points to two types of information related to a node: 1) primary node information (called `Vertex` in the diagram), such as metadata stored at that node and the incoming and outgoing edges, and 2) a log queue that stores changes on that node that are done during the SNAPSHOT phase to be lazily merged to `Vertex` later. There is a single lock per `Node` that guards either the `Vertex` or the `LogQueue` depending on what phase the algorithm is currently operating in.

The system operates in NORMAL mode where there is no snapshot in progress, switches to SNAPSHOT phase when a snapshot is requested and goes to CLEANUP after the snapshot is over; the cycle resumes after CLEANUP.

Operations performed in each phase:

- NORMAL phase – each worker thread takes lock on required node and does the update
- SNAPSHOT phase – snapshot thread waits for each worker thread to acknowledge that they are switching to SNAPSHOT phase before starting to linearly copy the primary vertex data from each node (without any other lock acquisition). Each worker thread after switching to SNAPSHOT phase gives up on writing to a vertex directly and instead appends to the log associated with each node. The node lock previously protecting the vertex data now protects the logQueues for each node. Since there is no thread writing to the vertex data (just the snapshot thread reading it),

there is no need of a lock for vertices any more.

- CLEANUP phase – When snapshot thread is done dumping the snapshot, it switches the global state to CLEANUP. The worker threads need to do the cleanup before going back to NORMAL mode. They do this by reading the logQueue associated with each node and merging the log with the primary vertex data.

2.3 Evaluation and Discussion

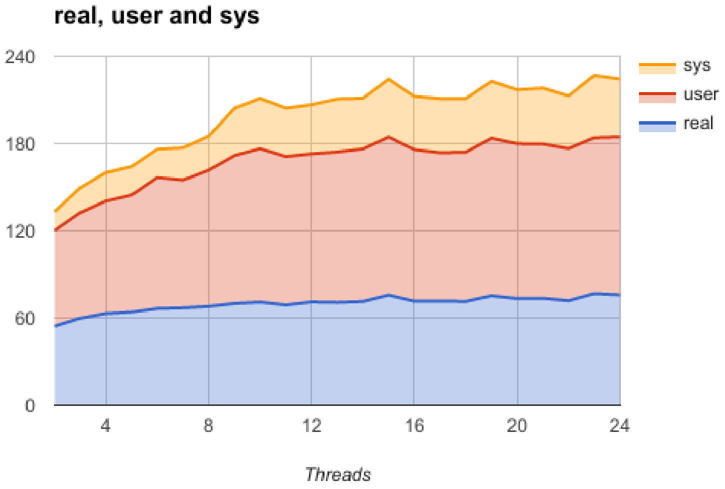

Figure 2.3: Time for V=1M, E=~7.6M inserts

We tested our prototype transient version of the implementation available at https://github.com/k4rtik/concurrent-graph for 1M vertices and 7.6M edges for 1) inserts and 2) inserts with periodic snapshots (Figures 2.3 - 2.5). The initial prototype included use of a STL map which severely affected performance, but we think our approach is promising and a second iteration on the implementation will provide a better picture.

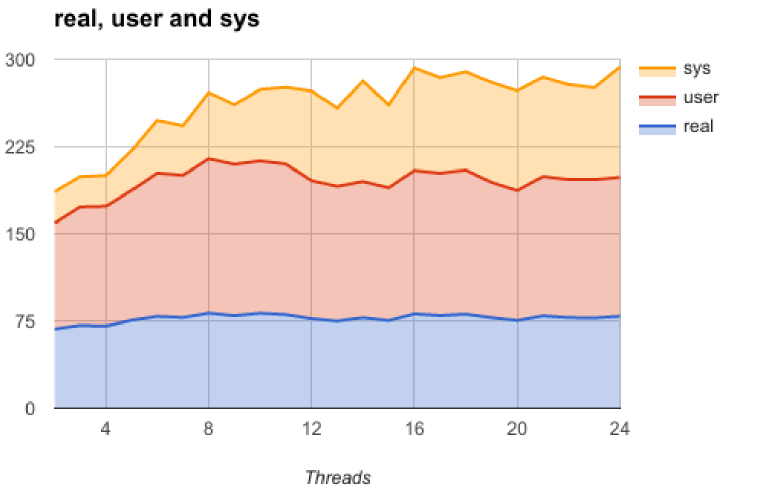

Figure 2.4: Time for V=1M, E=~7.6M inserts with periodic snapshots (1 per second)

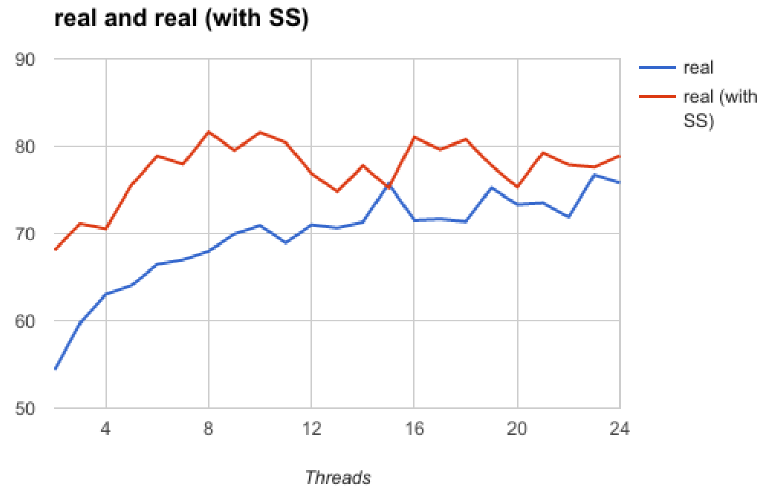

Figure 2.5: Comparison of wall clock time for just inputs vs inputs with periodic snapshots

Chapter 3

Formal Reasoning about Correctness of Persistent Programs

3.1 Why Reason about Correctness of Programs?

Most software written today, either free or commercial, comes with a license agreement disclaiming any warranty that the product is bug-free or will perform exactly as it is supposed to. While a civil engineer as a professional provides specific guarantees about the efficacy and failure modes of a bridge, a software engineer for a software product, does not. Computer Science as a scientific discipline has come a long way, but as an engineering discipline, it has a long way to go.

Adam Chlipala begins his excellent book on formal logical reasoning about correctness of programs and Coq proof assistant[coq] (a tool for machine-checked mathematical theorem proving), *Formal Reasoning About Programs*[Chl17, Ch. 1], as follows:

> **Why Prove the Correctness of Programs?**
>
> The classic engineering disciplines all have their standard mathematical techniques that are applied to the design of any artifact, before it is deployed, to gain confidence about its safety, suitability for some purpose, and so on. The engineers in a discipline more or less agree on what are "the rules" to be followed in vetting a design. Those rules are specified with a high degree of rigor, so that it isn't

> a matter of opinion whether a design is safe. Why doesn't software engineering have a corresponding agreed-upon standard, whereby programmers convince themselves that their systems are safe, secure, and correct? The concepts and tools may not quite be ready yet for broad adoption, but they have been under development for decades.

As humanity becomes increasingly dependent on software, software that keeps on becoming more and more complex, increasingly difficult to understand, and involving increasingly greater risk when it fails, there is an urgent need to change this status quo by improving our tools and techniques to produce correct, reliable software systems.

A high-level idea of the current state of the art in formal methods for software correctness can be gained from a recent NIST report (*Dramatically Reducing Software Vulnerabilities: Report to the White House Office of Science and Technology Policy*)[BBGF16, §2.1]:

> In the early days of programming, some practitioners proved the correctness of their programs. That is, given language semantics, they logically proved that their program had certain properties or gave certain results. As the use of software exploded and programs grew so large that purely manual proofs were infeasible, formal correctness arguments lost favor. In recent decades, developments, such as the breathtaking increase in processing capacity predicted by Moore's law, multi-core processors and cloud computing, made orders of magnitude more computing power readily available. Advances in algorithms for solving Boolean Satisfiability (SAT) problems, satisfiability modulo theories (SMT), decision procedures (e.g., ordered binary decision diagrams - OBDD) and reasoning models (e.g., abstract interpretation and separation logic) dramatically slashed resources required to answer questions about software.
>
> . . .
>
> By the 1990s, formal methods had developed a reputation as taking far too long, in machine time, person years and project time, and requiring a PhD in computer science and mathematics to use them. This is no longer the case. Formal methods are widely used today. For instance, compilers use SAT solvers to allocate registers and optimize code. Operating systems use algorithms formally guaranteed to avoid deadlock.

The report[BBGF16, §2.1.9] continues about the potential impact of formal methods:

> **Rationale for Potential Impact**
>
> The greatest potential impact is likely in costs avoided for components that, over time, become heavily relied upon. The Heartbleed debacle is an example of a modest code base with outsized importance: a

> judicious use of formal methods might have avoided the problem in the first place. Generally, higher quality software, such as can be produced using formal methods, can be used to lower long-term maintenance and replacement costs of software components. Unlike physical systems that wear out and eventually fail, software systems suffer failures when they are incorrect and the flaws are triggered by environmental factors, such as, particular sequences or combinations of inputs.

As mentioned in the introductory chapter, various programming models have been proposed for writing persistent programs, however, none has been widely adopted yet. We believe there is a lot to be gained in terms of assurance if formal reasoning techniques are adopted to ensure correctness of persistent programs right from the start.

In this report, we focus on separation logic and its variants as they apply to reasoning about correctness of concurrent programs and possibly for persistent programs.

3.2 Current State of the Art in Reasoning about Concurrent Programs

We enlist increasingly advanced mathematical logics that help us reason about correctness of programs, from simplest single-threaded programs with no support for linked data structures to the most complicated racy concurrent ones.

Hoare Logic is used for proving correctness properties for imperative programs. It introduces Hoare Triples: $\{P\}\ c\ \{Q\}$, where P and Q are *precondition* and *postcondition* (*assertions*) and c is *command*. The assertions are formulae in predicate logic. The logic consists of axioms and inference rules for each construct of the imperative language. Basic intuition behind Hoare triples is that if we start in a state where P is true, then execute c, we will end up in a state where Q is true. A major limitation of Hoare Logic is that is not capable of reasoning about linked data structures. Some example Hoare triples:

$$\{x = a\}\ if\ (x < 0)\ then\ x := -x\ \{x = |a|\}$$

$$\{false\}\ x := 3\ \{x = 8\}$$

Separation Logic (SL) is an extension of Hoare logic that supports reasoning about programs that involve pointer manipulation and allows localized reasoning about portions of the heap (as opposed to global state as a whole). It introduced the idea of transfer of ownership of portions of heap where ownership refers to the notion that a code fragment can access only those portions of the heap which it owns. It also introduced a novel logical primitive: $*$ (called *star* or *separating conjunction*). Intuitively, $\{P\} * \{Q\}$ partitions the overall heap into two subheaps each of which respectively satisfy one of P and Q. Though a significant improvement over previous state of the art in reasoning about sharing of mutable memory across libraries and data structures in sequential programs, SL was not sufficient for reasoning about most concurrent programs.

Concurrent Separation Logic (CSL) is a variant of SL that allows independent reasoning about threads that access separate pieces of memory. This was done with the introduction of Parallel Composition Rule[BO16] that allowed using the separating conjunction with multiple threads owning separate portions of memory. It supports dynamic ownership transfer of memory and modular reasoning. This was a major step forward in being able to reason about concurrent programs and the inventors were awarded 2016 Gödel Prize for that, which cites: "*For the last thirty years experts have regarded pointer manipulation as an unsolved challenge for program verification and shared-memory concurrency as an even greater challenge. Now, thanks to CSL, both of these problems have been elegantly and efficiently solved; and they have the same solution.*" It facilitated automation of proofs in practice because of its simplicity and similarity to common programming idioms. Most concurrent program logics in the past decade derive from CSL (a recent restrospective paper[BO16] on CSL by the inventors has more context). The original paper[O'H07] on CSL by O'Hearn describes several worked examples to gain intuition.

With most recent derivatives of CSL, it is possible to reason about even the most subtle racy concurrent programs, ie, those utilizing non-blocking techniques and those that require reasoning with the help of weak/relaxed memory models. GPS[TVD14] (Ghost State, Protocols, Separation Logic) was the first logic to support structured reasoning about weak memory. While GPS depended on protocols, Iris 1.0[JSS+15] proposed that monoids and invariants are all that are needed to reason about most concurrent programs, with partial commutative monoids (PCMs) used for expressing protocols on shared state and invariants used for enforcing them. Iris 2.0[JKBD16] extended Iris to support higher-order ghost state by generalizing PCMs to resource algebras. An interesting aspect of all of these logics is that each of these come with Coq implementations available which makes them quite promising for building over them.

3.3 Connection between Reasoning about Concurrency and Persistence

So far we have not explicitly mentioned why the work on correctness of concurrent programs may be relevant to the same for persistent programs, we discuss that here.

Aaron Turon, in his dissertation on "Understanding and Expressing Scalable Concurrency"[Tur13, §2], writes (emphasis theirs):

> The fundamental problem of concurrency, in our view, follows from the inherently shared-state, nondeterministic nature of expressive interaction:
>
> *Concurrent programming is the management of sharing and timing.*

Here 'timing' refers to when synchronized or unsynchronized (racy) access to shared state is allowed while ensuring global progress. Common mechanisms to control timing include, but are not limited to, locking, waiting on conditions, software transactional memory (STM) and optimistic mechanisms such as a retry loop, each with their own trade-offs.

Coming to persistence, the Atlas programming model recognizes that visibility (threading) and persistency (fail safety) critical sections are similar and that persistence depends on atomicity and ordering[CBB14, page 2]:

> In this paper we argue that for multithreaded lock-based programs. . . , the locking operations usually give us enough information to infer NVRAM atomicity and ordering requirements, and that it is useful to extend locking primitives with failure-atomicity semantics. A section of code is failure-atomic if the effects of either all or none of the enclosing updates are visible in NVRAM.

Further, we realize how relaxing memory consistency models allows greater performance to be achieved by (racy) concurrent programs. Similarly, there are several proposals for "memory persistency models" which define the semantics of pushing writes to the persistent (NVM) storage. Izraelevitz et al[IMS16, §3.2] write:

> The semantics of instructions controlling the ordering and timing under which cached values are pushed to persistent memory comprise a memory persistency model. Since any machine with bounded caches must sometimes evict and write back a line without program intervention, the principal challenge for designers of persistent objects is to ensure that a newer write does not persist before an older write (to some other location) when correctness after a crash requires the locations to be mutually consistent.

Many of these persistency models propose two primitive hardware instructions (often using different terminology): 1) *pfence* or *persistence fence*, which ensures ordering of persistent writes without confirmation from the memory device (asynchronous), and 2) *psync* that waits for the hardware buffer of ordered writes to drain (synchronous). *pfence* is especially quite similar to the *memory fence* instruction that enforces ordering constraints on memory operations and allows implementation of low-level high-performance concurrent programs. Programs written using fences available in the C11 memory model can already be verified using a recently proposed program logic called *Fenced Separation Logic*[DV16]. We think it it should be promising to apply similar reasoning to the persistency related primitives.

Apart from these similarities, it is clear that without incorporating concurrency, reasoning about persistent programs may be worthless as most interesting software that utilizes persistent memory is inherently concurrent.

But concurrency alone is not sufficient as we will see in the next section.

3.4 What about Crashes and Recovery?

The problem with reasoning about correctness of persistent programs is that when a system crash occurs it is not clear whether the data that was in registers/cache got persisted to NVM. A basic requirement of a correct persistent program is that when a crash occurs for any reason and the system reboots, the system should recover to a consistent program state. Concurrent Separation Logic and its variants though quite powerful even for reasoning about highly subtle racy programs are insufficient to reason about crashes.

Fortunately, **Crash Hoare Logic (CHL)**[CZC+15] deals with exactly this problem, albeit for disk consistency, by extending traditional Hoare Logic with crash conditions and recovery semantics and uses Separation Logic to represent parts of disk (as opposed to memory):

Crash conditions[CZC+15, §3.2]:

> To reason about the behavior of a procedure in the presence of crashes, CHL allows a developer to capture both the state at the end of the procedure's crash-free execution and the intermediate states during the procedure's execution in which a crash could occur.

Recovery execution semantics[CZC+15, §3.2] (given `log_recover` is the recovery procedure, emphasis theirs):

> To state that `log_recover` must run after a crash, CHL provides a recovery execution semantics. In contrast to CHL's regular execution semantics, which talks about a procedure producing either a failure (accessing an invalid disk block), a crash, or a finished state, the recovery semantics talks about *two* procedures executing (a normal procedure and a recovery procedure) and producing either a failure, a *completed* state (after finishing the normal procedure), or a *recovered* state (after finishing the recovery procedure). This regime models the notion that the normal procedure tries to execute and reach a completed state, but if the system crashes, it starts running the recovery procedure (perhaps multiple times if there are crashes during recovery), which produces a recovered state.

Though not exactly the same as disk consistency, reasoning about failure scenarios in persistent programs has a lot in common with what CHL tries to solve.

3.5 The Road Ahead

We have looked at a lot of work that takes us closer to proving correctness of both persistent programs and programming models being proposed to write them. But there are still some pieces missing that may lead to a new logic that can be used to model persistent memory and its various properties, most likely building on some CSL variant and CHL.

Currently, the specifications of the basic persistency primitives are still being finalized and we are yet to reach a consensus on which persistent programming model is the most flexible and suitable. But the work we need to do does not require waiting on standardization efforts as a recent abstract framework[IMS16] for reasoning about correctness of persistent programs demonstrates. It proposes theoretical safety conditions such as *durable linearizability* and *buffered durable linearizability* that can be used to reason about safety of persistent objects at a higher-level. The problem with this framework is that it builds over linearizability, which is the most well accepted theoretical safety condition for concurrent objects, but that has been found difficult to incorporate in mechanical proof systems; to quote Turon[Tur13, §3.4.2] again (emphasis theirs):

> Linearizability, . . . , is defined in terms of quite abstract "histories," seemingly without reference to any particular language. But to actually prove linearizability for specific examples, or to benefit formally from it as a client, *some* connection is needed to a language. In particular, there must be some (language-dependent) way to extract the possible histories of a given concurrent data structure—giving a kind of "history semantics" for the language.
>
> . . .
>
> If linearizability is a proof technique for refinement, its soundness is a kind of "context lemma" saying the observable behavior of a data structure with hidden state can be understood entirely in terms of (concurrent) invocations of its operations; the particular contents of the heap can be ignored. The problem is that the behavior of its operations—from which its histories are generated—*is* dependent on the heap. Any proof method that uses linearizability as a component must reason, at some level, about the heap. Moreover, linearizability is defined by quantifying over *all* histories, a quantification that cannot be straightforwardly tackled through induction. Practical ways of proving linearizability require additional technical machinery, the validity of which must be separately proved.

This suggests that a potential new logic compatible with these correctness conditions will still need a machinery like separation logic to reason about the heap.

Finally, we propose two ways that this work can be taken forward with our current understanding, both of which may be easier if attempted with sequential programs first:

- Bottom-up: We start with certain well-defined specifications (operational semantics) of basic persistency primitives such as *pfence* and *psync*; model them in a variant of a logic such as FSL; come up with well-defined safety properties that can be encoded suitably in this logic; and iterate over the design of this new logic by trying to prove these properties and specifications of higher-level data structures.

- Top-down: Choose a particular persistent programming model such as Atlas and treat the guarantees ensured by this model as a trusted base; pick a specific example program that uses this model to focus on (e.g. the code shown in chapter 1) and prove it correct with a variant of CHL; iterate over the design of this new logic with more examples, possibly by integrating properties of a powerful CSL variant such as Iris; and then start looking for ways to prove the design on the programming model correct.

Chapter 4

Conclusion and Future Work

We described our design of a concurrent graph data structure for non-volatile memory which offers crash-resilience with the help of Atlas programming model. Our current transient implementation seems promising but can be improved before attempting a persistent version.

We described the landscape of current research into proving the correctness of persistent programs and related work in the field of reasoning about concurrency (variants of CSLs) and failure scenarios (CHL). We plan to come up with a new logic and a (Coq-based) machine-checkable proof system for reasoning about the correctness of software written for persistent memory. The goal is to provide usable tools to programmers and designers of programming models working in the area of persistent memory to validate their assumptions. Crash Hoare Logic for crash scenarios and variants of Separation Logic seem to have the most essential foundations that seem necessary and that we expect to be able to build upon.

Acknowledgements

I am extremely grateful to my advisor Maurice Herlihy for his guidance and support during the course of this work. I am also thankful to my mentors at HP Labs: Dhruva Chakrabarti, who saw a research potential in me, and both him and Terence Kelly who guided me throughout the summer on my concurrent graph data structure project. I would also like to thank researchers from the verification community who helped me think through the problem of proving correctness of persistent programs and provided various insights – Derek Dreyer at MPI-SWS, Lars Birkedal and Aleš Bizjak, both at Aarhus University.

Kartik Singhal

May 2017
Brown University

References

[Bal16] Piotr Balcer. Persistent Memory Semantics in Programming Languages – Overview of the Ongoing Research, October 2016. LinuxCon+ContainerCon Europe 2016 (talk slides): http://events.linuxfoundation.org/sites/events/files/slides/linuxcon_pmem_lang_ext_v2_0.pdf.

[BBGF16] Paul E. Black, Mark L. Badger, Barbara Guttman, and Elizabeth N. Fong. Dramatically Reducing Software Vulnerabilities: Report to the White House Office of Science and Technology Policy. NIST Interagency/Internal Report (NISTIR) 8151, National Institute of Standards and Technology, December 2016. Available at: https://dx.doi.org/10.6028/NIST.IR.8151.

[BO16] Stephen Brookes and Peter W. O'Hearn. Concurrent Separation Logic. *ACM SIGLOG News*, 3(3):47–65, August 2016.

[CBB14] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[Chl17] Adam Chlipala. Formal Reasoning About Programs. MIT, Cambridge, MA, USA, 2017. Book webpage: http://adam.chlipala.net/frap/.

[coq] The Coq Proof Assistant. https://coq.inria.fr/.

[CZC+15] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM.

[DV16] Marko Doko and Viktor Vafeiadis. *A Program Logic for C11 Memory Fences*, pages 413–430. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[IMS16] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. *Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model*, pages 313–327. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[JKBD16] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order Ghost State. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 256–269, New York, NY, USA, 2016. ACM.

[JSS$^+$15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 637–650, New York, NY, USA, 2015. ACM.

[O'H07] Peter W. O'Hearn. Resources, Concurrency, and Local Reasoning. *Theoretical Computer Science*, 375(1):271 – 307, 2007.

[Tur13] Aaron Turon. *Understanding and Expressing Scalable Concurrency.* PhD thesis, College of Computer and Information Science, Northeastern University, Boston, Massachusetts, April 2013. Available at https://people.mpi-sws.org/~turon/turon-thesis.pdf.

[TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 691–707, New York, NY, USA, 2014. ACM.