Research Statement
Srivatsan Ravi

Almost every computing system nowadays is *distributed*, ranging from multicore CPUs prevalent in everyday desktops/laptops/Internet of Things (IoTs) to the very Internet itself. Thus, understanding the foundations of distributed computing is important for the design of efficient computational techniques across all scientific fields. Moreover, many problems that are trivial to solve sequentially are impossible or infeasible to solve in a distributed fashion, thus presenting us with problems of deep intellectual yet practical interest.

My research is primarily concerned with the theory and practice of distributed computing. Designing provably correct distributed programs requires overcoming some nontrivial challenges, the most important of which is achieving efficient *synchronization* among *processes* of computation. When there are several processes that attempt to *concurrently* access the same data, they will need to coordinate their actions to ensure correct program behaviour, thus motivating the search for efficient synchronization techniques. Synchronization though is hard due to *failures* and the *asynchrony* pervasive in distributed systems. In fact, one of the seminal results in distributed computing is the impossibility of *deterministically* achieving *consensus*: processes initially propose a value and must *eventually agree* on one of the proposed values [61, 32], among failure-prone processes in an asynchronous environment. Alternatively phrased, this result implies that it is impossible to achieve *consistency*, *availability* (à la *progress*) and *partition-tolerance* in an asynchronous environment [36], the so called *CAP* theorem. Intuitively, this result implies that programmers of distributed applications have to compromise on simultaneously providing *strong* variants of all three features of consistency, availability and the ability to cope with failures or circumvent the impossibility by adopting *weaker* models of computation.

Consequently, I find myself asking the following unsurprising questions about a distributed computation:

- What is the *model* of computation? Answering this question requires identifying the communication model: via *shared memory* or *message passing* or variants thereof; *timing* assumption: *synchrony vs. asynchrony* which specifies the relative speeds with which processes take steps in the computation; the *failure pattern* which identifies the ways in which some subset of processes may become faulty; computing and memory capacity of the processes, etc.

- Given a computational model, is it (im)possible to build a distributed application for a specific choice of consistency and availability?

- What are the complexity metrics that characterize the cost of the computation and can we derive bounds that identify the implementation's theoretical cost?

- What are the available hardware and programming abstractions that help application programmers realize a working implementation with largely *sequential* semantics in mind and finally

- How can we *verify* that the resulting implementation conforms to its intended *sequential* semantics?

Generally speaking, when reasoning about distributed computations in any context; multicore machines or graph algorithms for routing in today's Internet network architectures or information sharing in social peer-to-peer networks, all the above questions apply. But there are also other practical considerations: Is there a bound on the number of participating computational entities? Are we seeking *deterministic* or *randomized* algorithms? Is there an a priori agreed naming convention for identifying the computational entities? Lastly, how easy or hard is the distributed *programming model*? This latter question is especially important from a practical standpoint since it is the simplicity of the programming model that determines whether ordinary programmers may choose to adopt it. *Reasoning about the correctness of a distributed computation is a science onto itself* and the programming model must ideally enable

programmers to build distributed applications with largely their sequential semantics in mind and without worrying about synchronization problems that may arise.

1 Prior and ongoing work

Broadly speaking, I have worked extensively on algorithms, formal semantics, lower bounds and programming models for multicore CPUs as well as *cloud* infrastructures, implementations of concurrent data structures and efficient protocols for distributed *cryptocurrencies*. The following sections give a flavor of the kind of distributed computation models I have studied and hopefully, a hint of how distributed computing techniques have become all-pervasive; directing emerging hardware trends to deploying applications on the cloud platform and how my own research has contributed in expanding the problem and solution space.

1.1 Towards safe in-memory transactions

A significant portion of my past studies concern synchronization algorithms for today's multicore CPU architectures. This is typically modelled as a *shared memory* over which processes communicate using the CPU's instruction set. Traditional solutions for synchronization in shared memory like *locking* that provide *mutual exclusion*, *i.e.*, restricting data access to at most one process at a time, come with limitations. *Coarse-grained* locking typically serializes access to a large amount of data and does not fully exploit hardware concurrency. Program-specific *fine-grained* locking, on the other hand, is a dark art to most programmers and trusted to the wisdom of a few computing experts. Thus, it is appealing to seek a middle ground between these two extremes: a synchronization mechanism that relieves the programmer of the overhead of reasoning about the *conflicts* that may arise from concurrent operations without severely limiting the program's performance. The *Transactional memory (TM)* abstraction [48, 75] is such a mechanism: it combines an easy-to-use programming interface with an efficient utilization of the concurrent-computing abilities provided by multicore architectures.

Transactional memory allows the user to declare sequences of instructions as speculative *transactions* that can either *commit* or *abort*. If a transaction commits, it appears to be executed sequentially, so that the committed transactions constitute a correct sequential execution. If a transaction aborts, none of its instructions can affect other transactions. The TM implementation endeavors to execute these instructions in a manner that efficiently utilizes the concurrent computing facilities provided by multicore architectures.

1.1.1 Safety for Transactional Memory [11, 10, 12]

We formalize the semantics of a *safe* TM: every transaction, including aborted and incomplete ones, must observe a view that is *consistent* with some sequential execution. This is important, since if the intermediate view is not consistent with *any* sequential execution, the application may experience a fatal irrevocable error or enter an infinite loop. Additionally, the response of a transaction's read should not depend on an ongoing transaction that has not started committing yet. This restriction, referred to as *deferred-update semantics* appears desirable, since the ongoing transaction may still abort, thus rendering the read inconsistent. We define the notion of deferred-update semantics formally and apply it to several TM consistency criteria proposed in literature. We then verify if the resulting TM consistency criterion is a *safety* property [68, 6, 62] in the formal sense, *i.e.*, the set of *histories* (interleavings of invocations and responses of transactional operations) is *prefix-closed* and *limit-closed*.

We first consider the popular consistency criterion of *opacity* [44]. Opacity requires the states observed by all transactions, included uncommitted ones, to be consistent with a global *serialization*, *i.e.*, a serial execution constituted by committed transactions. Moreover, the serialization should respect the *real-time order*: a transaction that completed before (in real time) another transaction started should appear first in the serialization.

One may notice that the intended safety semantics does not require, as opacity does, that all transactions observe the same serial execution. As long as committed transactions constitute a serial execution and every transaction witnesses a consistent state, the execution can be considered "safe": no run-time error that cannot occur in a serial execution can happen. We undertake a comprehensive study of definitions in literature that have adopted this approach [51, 28] and verify if they are indeed safety properties.

1.1.2 Complexity of transactional memory

One may observe that a TM implementation that aborts or never commits any transaction is trivially safe, but not very useful. Thus, the TM implementation must satisfy some nontrivial *liveness* property specifying the conditions under which the transactional operations must return some response and a *progress* property specifying the conditions under which the transaction is allowed to abort.

Two properties considered important for TM performance are *read invisibility* [14] and *disjoint-access parallelism* [52]. Read invisibility may boost the concurrency of a TM implementation by ensuring that no reading transaction can cause any other transaction to abort. The idea of disjoint-access parallelism is to allow transactions that do not access the same data item to proceed independently of each other without memory contention.

We investigate the inherent complexities in terms of time and memory resources associated with implementing safe TMs that provide strong liveness and progress properties, possibly combined with attractive requirements like read invisibility and disjoint-access parallelism. Which classes of TM implementations are (im)possible to solve?

Blocking TMs [55, 58]. We begin by studying TM implementations that are *blocking*, in the sense that, a transaction may be delayed or aborted due to concurrent transactions. (i) We prove that, even inherently *sequential* TMs, that allow a transaction to be aborted due to a concurrent transaction, incur significant complexity costs when combined with read invisibility and disjoint-access parallelism. (ii) We prove that, *progressive* TMs, that allow a transaction to be aborted only if it encounters a read-write or write-write conflict with a concurrent transaction [43], may need to exclusively control a linear number of data items at some point in the execution. (iii) We then turn our focus to *strongly progressive* TMs [44] that, in addition to progressiveness, ensures that *not all* concurrent transactions conflicting over a single data item abort. We prove that in any strongly progressive TM implementation that accesses the shared memory with *read*, *write* and *conditional* primitives, such as compare-and-swap, the total number of *remote memory references* [7, 13] (RMRs) that take place in an execution in which n concurrent processes perform transactions on a single data item might reach $\Omega(n \log n)$ in the worst-case. (iv) We show that, with respect to the amount of *expensive synchronization* patterns like compare-and-swap instructions and *memory barriers* [9, 64], progressive implementations are asymptotically optimal. We use this result to establish a linear (in the transaction's data set size) separation between the worst-case transaction expensive synchronization complexity of progressive TMs and *permissive* TMs that allow a transaction to abort only if committing it would violate opacity.

Non-blocking TMs [56]. Next, we focus on TMs that avoid using locks and rely on non-blocking synchronization: a prematurely halted transaction cannot not prevent other transactions from committing. Possibly the weakest non-blocking progress condition is obstruction-freedom [46, 50] stipulating that every transaction running in the absence of *step contention*, *i.e.*, not encountering steps of concurrent transactions, must commit. In fact, several early TM implementations [47, 63, 75, 76, 34] satisfied obstruction-freedom. However, circa. 2005, several papers presented the case for a shift from TMs that provide obstruction-free TM-progress to lock-based progressive TMs [27, 26, 31]. They argued that lock-based TMs tend to outperform obstruction-free ones by allowing for simpler algorithms with lower complexity overheads. We prove the following lower bounds for obstruction-free TMs. (i) Combining invisible reads with even *weak* forms of disjoint-access parallelism [15] in obstruction-free TMs is impossible, (ii) A read operation in a n-process obstruction-free TM implementation incurs $\Omega(n)$ *memory stalls* [30, 8]. (iii) A *read-only* transaction may need to perform a linear (in n) number of expensive synchronization patterns. We then present a progressive TM implementation that beats all of these lower bounds, thus suggesting that the course correction from non-blocking (obstruction-free) TMs to blocking (progressive) TMs was indeed justified.

Partially non-blocking TMs [57]. Lastly, we explore the costs of providing non-blocking progress to only a *subset* of transactions. Specifically, we require *read-only* transactions to commit *wait-free*, *i.e.*, every transaction commits within a finite number of its steps, but *updating* transactions are guaranteed to commit only if they run in the absence of concurrency. We show that combining this kind of *partial* wait-freedom with read invisibility *or* disjoint-access parallelism comes with inherent costs. Specifically, we establish the following lower bounds for TMs that provide this kind of partial wait-freedom. (i) This kind of partial wait-freedom equipped with invisible reads results in maintaining unbounded sets of *versions* for every data item. (ii) It is impossible to implement a *strict* form of disjoint-access parallelism [42]. (iii) Combining with the weak form of disjoint-access parallelism means that a read-only transaction (with an arbitrarily large read set) must sometimes perform at least one expensive synchronization pattern per read operation in some executions.

1.1.3 Complexity of Hybrid transactional memory

The TM abstraction, in its original manifestation, augmented the processor's *cache-coherence protocol* and extended the CPU's instruction set with instructions to indicate which memory accesses must be transactional [48]. Most popular TM designs, subsequent to the original proposal in [48] have implemented all the functionality in software [24, 75, 47, 63, 34]. More recently, CPUs have included hardware extensions to support small transactions [70, 1, 67]. Hardware transactions may be spuriously aborted due to several reasons: cache capacity overflow, interrupts *etc*. This has led to proposals for *best-effort* HyTMs in which the fast, but potentially unreliable hardware transactions are complemented with slower, but more reliable software transactions. However, the fundamental limitations of building a HyTM with nontrivial concurrency between hardware and software transactions are not well understood. Typically, hardware transactions usually employ *code instrumentation* techniques to detect concurrency scenarios and abort in the case of contention. But are there inherent instrumentation costs of implementing a HyTM, and what are the trade-offs between these costs ands provided concurrency, *i.e.*, the ability of the HyTM to execute hardware and software transactions in parallel?

Cost of instrumentation in Hybrid transactional memory [4, 3, 5]. (i) We propose a general model for HyTM implementations, which captures the notion of *cached* accesses as performed by hardware transactions, and precisely defines instrumentation costs in a quantifiable way. (ii) We derive lower and upper bounds in this model, which capture for the first time, an inherent trade-off on the degree of concurrency allowed between hardware and software transactions and the *instrumentation* overhead introduced on the hardware.

Cost of concurrency in Hybrid transactional memory [17, 18]. State-of-the-art *Software Transactional Memory (TM)* implementations achieve good performance by carefully avoiding the overhead of *incremental validation*, i.e., re-reading previously read data items to preclude inconsistent executions. Hardware TMs promise even better performance. However hardware transactions offer no progress guarantees since they may abort for *spurious* reasons and *conflict aborts*. Thus, they must be combined with software TMs, thus leading to the advent of *hybrid* TMs (HyTM). To allow hardware transactions in a HyTM to detect conflicts with software transactions, hardware transactions must be *instrumented* to perform additional metadata accesses. This instrumentation introduces overhead.

We first show that, unlike in software TMs, software transactions in HyTMs cannot avoid incremental validation. Specifically, we establish that *progressive* HyTMs in which a transaction may be aborted only due to a *data* conflict with a concurrent transaction, must necessarily incur a validation cost that is *linear* in the size of the transaction's read set. This is in stark contrast to progressive software TMs which can achieve $O(1)$ complexity operations. Secondly, we present two provably *opaque* HyTM algorithms in which both hardware and software transactions perform an optimal number of metadata accesses. The first algorithm is *progressive* and the second algorithm is progressive only for *read-only* software transactions. We show how some of the metadata accesses in these algorithms can be performed *non-speculatively* without violating opacity. We evaluate implementations of these algorithms on Intel Haswell, which does not support non-speculative accesses inside a hardware transaction, and IBM Power8, which does.

1.2 Designing concurrency-optimal data structures

To be pessimistic or optimistic? [38, 39, 40]. Lock-based implementations are conventionally *pessimistic* in nature: the operations invoked by processes are not "abortable" and return only after they are successfully completed. The TM abstraction is a realization of *optimistic* concurrency control: speculatively execute transactions, abort and roll back on dynamically detected conflicts. But are optimistic implementations fundamentally better equipped to exploit concurrency than pessimistic ones?

We compare the *amount of concurrency* one can obtain by converting a sequential implementation of a data abstraction into a concurrent one using optimistic or pessimistic synchronization techniques. To establish fair comparison of such implementations, we introduce a new correctness criterion for concurrent implementations, called *locally serializable linearizability*, defined independently of the synchronization techniques they use.

We treat an implementation's concurrency as its ability to accept *schedules* of sequential operations from different processes. More specifically, we assume an external scheduler that defines which processes execute which steps of the corresponding sequential implementation in a dynamic and unpredictable fashion. This allows us to define concurrency provided by an implementation as the set of interleavings of steps of sequential operations (or schedules) it *accepts*, *i.e.*, is able to effectively process. Then, the more schedules the implementation would accept without hampering correctness, the more concurrent it would be.

Our work makes the following contributions: (i) We provide a framework to analytically capture the inherent concurrency provided by two broad classes of synchronization techniques: pessimistic implementations that implement some form of mutual exclusion and optimistic implementations based on speculative executions. (ii) We explore the concurrency properties of *search* data structures which can be represented in the form of directed acyclic graphs exporting insert, delete and search operations. We prove, for the first time, that *pessimistic* (*e.g.*, based on conservative locking) and *optimistic serializable* (*e.g.*, based on serializable transactional memory) implementations of search data-structures are incomparable in terms of concurrency. Specifically, there exist simple interleavings of sequential code that cannot be accepted by *any* pessimistic (and *resp.*, serializable optimistic) implementation, but accepted by a serializable optimistic one (and *resp.*, pessimistic). Thus, neither of these two implementation classes is concurrency-optimal. Our results suggest that "semantics-aware" optimistic implementations may be better suited to exploiting concurrency than their pessimistic counterparts.

Concurrency-optimal list and binary search tree [41, 2]. We propose the first provably concurrency-optimal data structure, the Versioned list. We show that the previous most efficient lists are not concurrency-optimal in that they reject schedules of memory accesses that would not violate consistency. To this end, we consider the classic set implementation as an example and show that, unlike the Harris-Michael [49] and the Lazy list-based sets [45], the Versioned list is concurrency-optimal in that it accepts all correct schedules.

In addition, the Versioned list is probably the fastest list algorithm to date. It builds upon a new *pre-locking validation* technique that exploits versioning and try-locks to achieve high performance of update operations and to reduce the overhead of read-only operations. We implement our algorithm in Java 8, using the new StampedLock, and in C11, exploiting the stdatomic intrinsics, and show that it outperforms the Harris-Michael algorithm, the Lazy list algorithm and the Selfish optimization of Fomitchev and Ruppert's algorithm [33] on Power8, SPARC and x86-64 architectures.

We also present the first concurrency-optimal implementation of a binary search tree (BST). The implementation is based on a standard sequential implementation of an internal tree, and it ensures that every *schedule*, i.e., interleaving of steps of the sequential code, unless linearizability is violated. To ensure this property, we use an novel read-write locking scheme that protects tree *edges* in addition to nodes. Our implementation outperforms the state-of-the art BSTs on most basic workloads, which suggests that optimizing the set of accepted schedules of the sequential code can be an adequate design principle for efficient concurrent data structures.

1.3 Scheduling for big-data processing frameworks [72]

Often used in multi-user environments, big-data processing frameworks like Hadoop and Spark have struggled to achieve a balance between the full utilization of cluster resources and fairness between users. In particular, data locality becomes a concern, as enforcing fairness policies may cause poor placement of tasks in relation to the data on which they operate. To combat this, the schedulers in many frameworks use a heuristic called delay scheduling, which involves waiting for a short, constant interval for data-local task slots to become free if none are available; however, a fixed delay interval is inefficient, as the ideal time to delay varies depending on input data size, network conditions, and other factors.

We propose an adaptive solution (Dynamic Delay Scheduling), which uses a simple feedback metric from finished tasks to adapt the delay scheduling interval for subsequent tasks at runtime. We present a dynamic delay implementation in Spark, and show that it outperforms a fixed delay in TPC-H benchmarks. Our preliminary experiments confirm our intuition that job latency in batch-processing scheduling can be improved using simple adaptive techniques with almost no extra state overhead.

1.4 Distributed transactions for programming scalable cloud services

Distributed programming model for cloud services [71]. Designing distributed Internet-facing applications that are adaptable to unpredictable workloads and efficiently utilize modern cloud computing platforms is hard. The *actor* model is a popular paradigm that can be used to develop distributed applications: actors encapsulate state and communicate with each other by sending events. Consistency is guaranteed if each event only accesses a single actor, thus eliminating potential data races and deadlocks. However it is nontrivial to provide consistency for concurrent events spanning across multiple actors.

We address this problem by introducing the Atomic Events and Ownership Network (AEON): a protocol for *strongly consistent* and truly *scalable* cloud applications across distributed actors. Concretely AEON provides the

following properties: (i) *Programmability*: programmers need only reason about sequential semantics when reasoning about concurrency resulting from multi-actor events; (ii) *Scalability*: its runtime protocol guarantees *serializable* and *starvation-free* execution of multi-actor events, while maximizing parallel execution; (iii) *Elasticity*: supports *elasticity* enabling the programmer to transparently migrate individual actors without violating atomicity or entailing significant performance overheads.

We implemented a highly available and fault-tolerant prototype of AEON in `C++`. We present formal operational semantics which proves serializability and the absence of deadlocks. Extensive experiments show several complex cloud applications built atop AEON significantly outperform others built using existing state-of-the-art distributed cloud programming protocols. According to the experiments, AEON is about 3x faster than similar programming models (EventWave [22] and Orleans [20]). And the elasticity of AEON guarantees service quality with minimal cost compared to any static setup.

Programmable Elasticity for Actor-based Cloud Applications [16]. For many applications, it is pertinent for programmers to reason about *distributed state* resulting from message composition across multiple actors. Building a cost-effective and scalable cloud application requires fine-grained scale-adjustment among the distributed application state and cloud resources. However, there is no efficient solution which could manage application elasticity automatically during runtime without completely disrupting ongoing and application events invoked by the clients.

In this work, we propose the idea of "programmable elasticity", which allows programmers to define elasticity rules about actors with our programming model. The runtime endeavors to implement the elasticity rules while relieving the application programmer from dealing with the management of distributed state and efficient utilization of cloud resources. Unlike existing solutions, ours is designed for stateful actor-based applications, and the runtime conducts elasticity management at the level of actors.

1.5 Towards Byzantine Generalized Paxos [65]

The *Paxos* [59] protocol for reaching agreement in a distributed system has made its way to the core of the implementation of the services that are used by millions of people over the Internet, in particular since Paxos-based state machine replication is the key component of Google's Chubby lock service [19], or the open source ZooKeeper project [54], used by Yahoo! among others.

One of the most recent members of the Paxos family of protocols is *Generalized* Paxos. This variant of Paxos has the characteristic that it departs from the original specification of consensus, allowing for a weaker safety condition where different processes can have a different views on a sequence being agreed upon. However, much like the original Paxos counterpart, Generalized Paxos does not have a simple implementation. Furthermore, with the recent practical adoption of *Byzantine* [73] fault tolerant protocols in the context of blockchain protocols, it is timely and important to understand how Generalized Paxos can be implemented in the Byzantine model. In this work, we make two main contributions. First, we attempt to provide a simpler description of Generalized Paxos, based on a simpler specification and the pseudocode for a solution that can be readily implemented. Second, we extend the protocol to the Byzantine fault model, and provide the respective correctness proof.

1.6 Concurrency and privacy for cryptocurrencies [69]

Cryptocurrencies like *Bitcoin* [66] and *Ripple* [74] have grown as a possible avenue for *secure* decentralized online payments and credit exchange between arbitrary pairs of processes in a distributed system. It is expected that any cryptocurrency synchronization protocol needed to execute secure financial transactions provide resilience against *Byzantine* adversaries, *i.e.*, computing entities exhibiting malicious behavior. Unlike traditional protocols for *Byzantine agreement* typically require knowledge of the set of participating processes [21], protocols for cryptocurrencies are expected to work in a peer-to-peer setting in which several processes may join or leave the network arbitrarily. In such cryptocurrencies, processes continuously extend a distributed data structure called a *blockchain* that maintains the list of transactions issued by processes over time. Protocols like Bitcoin achieve agreement over the blockchain data structure by forcing processes to solve a *proof-of-work* [29]: a cryptographic puzzle that essentially allows the Bitcoin to tradeoff computation for communication complexity.

However, *permissionless* blockchains based on global consensus such as Bitcoin are inherently limited in transaction throughput and latency. Current efforts to address this key issue focus on off-chain payment channels that can be combined in a Payment-Channel Network (PCN) to enable an unlimited number of payments without requiring to access the blockchain other than to register the initial and final capacity of each channel. While this approach paves the

way for low latency and high throughput of payments, its deployment in practice raises several privacy issues as well as technical challenges related to the inherently concurrent nature of payments, such as race conditions and deadlocks, that have been understudied so far.

In this work, we lay the foundations for privacy and concurrency in PCNs, presenting a formal definition in the Universal Composability framework as well as practical and provably secure enforcement solutions. In particular, we present *Fulgor* and *Rayo*. Fulgor is the first payment protocol for PCNs that provides provable privacy guarantees for PCN and is fully compatible with current Bitcoin. Nevertheless, Fulgor is a blocking protocol and therefore prone to deadlocks of concurrent payments as in currently available PCNs. Instead, Rayo is the first protocol for PCNs that enforces non-blocking progress (i.e., at least one of the concurrent payments terminates). We show through a new impossibility result that the latter property necessarily comes at the cost of breaking anonymity. At the core of Fulgor and Rayo is Multi-Hop HTLC, a new smart contract, compatible with Bitcoin, that provides conditional payments while reducing running time and communication overhead with respect to previous approaches. Our performance evaluation of Fulgor and Rayo shows that a payment with 10 intermediate users takes as few as 5 seconds and requires to communicate 17 MB, thereby demonstrating their feasibility to be deployed in practice.

2 Expected future work

These are some of the questions I have thought about, but have not managed to write interesting papers (yet). In general, I often find that my research is motivated and shaped by emerging new hardware trends that require a new abstract computation model or via introduction of techniques from distributed computing to domains where the sequential implementation continues to be state-of-the-art.

2.1 Concurrent data structures for non-volatile memory (NVM)

It is expected that current *volatile* memory based on DRAM will be augmented by *storage-class memories (SCM)* that are *non-volatile* and *byte-addressable*. The primary advantage of this hardware development is that it removes the need for two distinct file formats: the in-memory object formatsand the persistent file format for the block-oriented traditional persistent storage à la NAND flash that is prevalent in today's solid-state devices. Yet, whether the data structure is designed directly on NVM or via a combination of DRAM plus NVM, *i.e.*, DRAM with a NVM backup on account of the DRAM crash failure, there remain several open questions concerning the design of efficient *persistent* concurrent data structures. Firstly, the data structure *state* must be constantly updated in the non-volatile memory so that in the event of a crash failure, the computation may re-start from the *most recent consistent state* of the data structure. This write-back to the NVM must be *atomic* so that the recovered data structure state is *consistent*. Secondly, this raises the following question: what must be representation of the data structure in the NVM? For *e.g.*, in a *sorted linked-list-based set*, it may be sufficient to store the set of values contained in the set, as opposed to an *unsorted* one since the pointer references can not be deterministically re-created during the *re-start* procedure invoked after the crash-recovery. Deriving complexity bounds and implementing provably correct algorithms for *non-blocking* data structures in NVMs is something I have been thinking about.

2.2 Secure computing platform for Internet-of-Things (IoT)

Computing today has evolved well beyond traditional computing processors to generic physical devices connected to the Internet, the so called IoT. Traditional distributed computing techniques for synchronization have typically focussed on relatively *homogenous* computing devices. However, the ubiquitous nature of IoT devices employing multiple heterogenous embedded computing devices (e.g., Arduino, Raspberry Pi, Smart Phones etc.), communication technologies (LTE, WiFi, RFID, etc.) and sensor networks (RFID tag, Actuators, etc.) requires new distributed algorithms that allow IoT devices to shared information and coordinate decisions in a secure and privacy-preserving manner. This opens several unique distributed computing challenges that I expect to be working towards, including modelling the IoT computing platform and designing correct-by-construction algorithms that can deal with its sheer scale and heterogeneity.

2.3 Distributed algorithms using trusted computing modules

Recent distributed computing research has focussed on implementing secure multi-part computation protocols that assumes each computational entity has access to a *trusted* hardware module [53]. Intuitively, this trusted hardware module prevents a malicious adversary from sending conflicting messages to different honest processes. Implementing distributed algorithms using trusted computing modules has become more relevant since Intel released Software Guard Extensions (SGX) [23] that enables secure remote computation among other use-cases. Yet, what sort of trusted computing modules must be provided by hardware manaufacturers that can benefit a maximal set of distributed applications? This is a problem space I have made some preliminary progress and intend to continue working on.

2.4 Computing in oblivious shared memory

We consider the problem of computing in an asynchronous shared memory that is *oblivious* [37]: no information is divulged to an adversary about the *access patterns* of the processes to the shared memory. Besides the fact that this a nontrivial problem to formulate and solve, it had wide ranging applications especially in the context of today's cloud computing services. Consider a server hosting a shared memory for client processes to access by invoking operations of a distributed algorithm. However, clients would wish that the server or the other clients be oblivious to their individual access patterns. Specifically, a data structure implementation is said to be oblivious if any two equal-length sequences of operations invoked are *computationally indistinguishable* to anyone but the client, a solution that typically involves employing *homomorphic encryption* schemes [35]. One of my goals is formalizing the notion of obliviousness in the distributed setting as well as deriving algorithms and complexity bounds for implementing oblivious non-blocking cloud applications.

3 Concluding remarks

In his wonderfully sarcastic critique of the scientific community in *His Master's Voice* [1], the great Polish writer Stanisław Lem refers to a *specialist* as a *barbarian whose ignorance is not well-rounded*. While I have attempted at becoming a specialist on all things distributed, I am culturedly not totally ignorant of exciting innovations that are needed in both the problem and solution space of distributed systems and their application across the STEM fields. Indeed, given the ubiquity of computational algorithms across other scientific disciplines ranging from neurosciences [60], fault-tolerant hardware design, quantum computing [25] and in general analytics for the data sciences, I envision introducing techniques from distributed computing to other scientific fields.

References

[1] Advanced Synchronization Facility Proposed Architectural Specification, March 2009. `http://developer.amd.com/wordpress/media/2013/09/45432-ASF_Spec_2.1.pdf`.

[2] V. Aksenov, V. Gramoli, P. Kuznetsov, A. Malova, and S. Ravi. A concurrency-optimal binary search tree. In *23rd International European Conference on Parallel and Distributed Computing (EURO-PAR), Spain*, 2017.

[3] D. Alistarh, J. Kopinsky, P. Kuznetsov, S. Ravi, and N. Shavit. Inherent limitations of hybrid transactional memory. *6th Workshop on the Theory of Transactional Memory, Paris, France*, 2014.

[4] D. Alistarh, J. Kopinsky, P. Kuznetsov, S. Ravi, and N. Shavit. Inherent limitations of hybrid transactional memory. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 185–199, 2015.

[5] D. Alistarh, J. Kopinsky, P. Kuznetsov, S. Ravi, and N. Shavit. Inherent limitations of hybrid transactional memory. *Distributed Computing*, 2017.

[6] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, Oct. 1985.

[7] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[8] H. Attiya, R. Guerraoui, D. Hendler, and P. Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4), 2009.

[1] Stanislaw Lem, His Master's Voice, Harvest Books, 1984, ISBN 0-15-640300-5

[9] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498, 2011.

[10] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. What is safe in transactional memory. *4th Workshop on the Theory of Transactional Memory, Madeira, Portugal*, 2012.

[11] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of deferred update in transactional memory. In *ICDCS*, pages 601–610, 2013.

[12] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety and deferred update in transactional memory. In R. Guerraoui and P. Romano, editors, *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, volume 8913 of *Lecture Notes in Computer Science*, pages 50–71. Springer International Publishing, 2015.

[13] H. Attiya, D. Hendler, and P. Woelfel. Tight rmr lower bounds for mutual exclusion and other problems. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 447–447, New York, NY, USA, 2008. ACM.

[14] H. Attiya and E. Hillel. The cost of privatization in software transactional memory. *IEEE Trans. Computers*, 62(12):2531–2543, 2013.

[15] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4):698–719, 2011.

[16] G. P. M. A. N. Z. M. P. E. Bo Sang, Srivatsan Ravi. Programmable elasticity for actor-based cloud applications. In *9th Workshop on Programming Languages and Operating Systems (PLOS 2017)*, 2017.

[17] T. Brown and S. Ravi. Cost of concurrency in hybrid transactional memory. *In Workshop on Transactional Computing (Transact), Austin Texas*, 2017.

[18] T. Brown and S. Ravi. Cost of concurrency in hybrid transactional memory. In *Distributed Computing - 31th International Symposium, DISC*, 2017.

[19] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350. USENIX Association, 2006.

[20] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. page 16, 2011.

[21] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, Feb. 1999.

[22] W. Chuang, B. Sang, S. Yoo, R. Gu, M. Kulkarni, and C. E. Killian. EventWave: Programming Model and Runtime Support for Tightly-coupled Elastic Cloud Applications. page 21, 2013.

[23] V. Costan, I. A. Lebedev, and S. Devadas. Secure processors part I: background, taxonomy for secure enclaves and intel SGX architecture. *Foundations and Trends in Electronic Design Automation*, 11(1-2):1–248, 2017.

[24] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. *SIGPLAN Not.*, 45(5):67–78, Jan. 2010.

[25] V. S. Denchev and G. Pandurangan. Distributed quantum computing: A new frontier in distributed systems or science fiction? *SIGACT News*, 39(3):77–95, Sept. 2008.

[26] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[27] D. Dice and N. Shavit. What really makes transactions fast? In *Transact*, 2006.

[28] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.*, 25(5):769–799, 2013.

[29] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, pages 139–147, London, UK, UK, 1993. Springer-Verlag.

[30] F. Ellen, D. Hendler, and N. Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012.

[31] R. Ennals. Software transactional memory should not be obstruction-free. 2005.

[32] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[33] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *PODC*, pages 50–59, 2004.

[34] K. Fraser. Practical lock-freedom. Technical report, Cambridge University Computer Laborotory, 2003.

[35] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[36] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[37] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 182–194, New York, NY, USA, 1987. ACM.

[38] V. Gramoli, P. Kuznetsov, and S. Ravi. From sequential to concurrent: correctness and relative efficiency (short paper). In *Principles of Distributed Computing (PODC)*, pages 241–242, 2012.

[39] V. Gramoli, P. Kuznetsov, and S. Ravi. Sharing a sequential data structure: correctness definition and concurrency analysis. *4th Workshop on the Theory of Transactional Memory, Madeira, Portugal*, 2012.

[40] V. Gramoli, P. Kuznetsov, and S. Ravi. In the search for optimal concurrency. In *Structural Information and Communication Complexity - 23rd International Colloquium, SIROCCO 2016, Helsinki, Finland, July 19-21, 2016, Revised Selected Papers*, pages 143–158, 2016.

[41] V. Gramoli, P. Kuznetsov, S. Ravi, and D. Shang. A concurrency-optimal list-based set (short paper). In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9*, 2015.

[42] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 304–313, New York, NY, USA, 2008. ACM.

[43] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. *SIGPLAN Not.*, 44(1):404–415, Jan. 2009.

[44] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

[45] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2006.

[46] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

[47] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[48] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[49] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[50] M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.

[51] D. Imbs and M. Raynal. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theor. Comput. Sci.*, 444, July 2012.

[52] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC*, pages 151–160, 1994.

[53] A. Jaffe, T. Moscibroda, and S. Sen. On the price of equivocation in byzantine agreement. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 309–318, 2012.

[54] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, DSN '11, pages 245–256.

[55] P. Kuznetsov and S. Ravi. On the cost of concurrency in transactional memory. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 112–127, 2011.

[56] P. Kuznetsov and S. Ravi. Grasping the gap between blocking and non-blocking transactional memories. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 232–247, 2015.

[57] P. Kuznetsov and S. Ravi. On partial wait-freedom in transactional memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, page 10, 2015.

[58] P. Kuznetsov and S. Ravi. Progressive transactional memory in time and space. In *Parallel Computing Technologies - 13th International Conference, PaCT 2015, Petrozavodsk, Russia, August 31 - September 4, 2015, Proceedings*, pages 410–425, 2015.

[59] L. Lamport. The Part-Time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[60] J. W. Lichtman, H. Pfister, and N. Shavit. The big data challenges of connectomics. *Nature Neuroscience*, 17(11):1448–1454, Oct. 2014.

[61] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[62] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[63] V. J. Marathe, W. N. S. Iii, and M. L. Scott. Adaptive software transactional memory. In *In Proc. of the 19th Intl. Symp. on Distributed Computing*, pages 354–368, 2005.

[64] P. E. McKenney. Memory barriers: a hardware view for software hackers. Linux Technology Center, IBM Beaverton, June 2010.

[65] R. R. Miguel Pires, Srivatsan Ravi. Generalized paxos made byzantine (and less complex). In *19th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2017)*, 2017.

[66] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2009.

[67] M. Ohmacht. Memory Speculation of the Blue Gene/Q Compute Chip, 2011. `http://wands.cse.lehigh.edu/IBM_BQC_PACT2011.ppt`.

[68] S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.

[69] A. K. M. M. S. R. Pedro Moreno-Sanchez, Giulio Malavolta. Concurrency and privacy with payment-channel networks. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[70] J. Reinders. Transactional Synchronization in Haswell, 2012. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`.

[71] B. Sang, G. Petri, M. S. Ardekani, S. Ravi, and P. T. Eugster. Programming scalable cloud services with AEON. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*, page 16, 2016.

[72] D. Schatzlein, S. Ravi, Y. Noh, M. S. Ardekani, and P. Eugster. The misbelief in delay scheduling. In *Proceedings of the 4th Workshop on Distributed Cloud Computing*, DCC '16, pages 9:1–9:6, New York, NY, USA, 2016. ACM.

[73] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, May 1984.

[74] D. Schwartz, N. Youngs, and A. Britto. The ripple consensus protocol. 2014.

[75] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[76] F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. Nztm: Nonblocking zero-indirection transactional memory. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 204–213, New York, NY, USA, 2009. ACM.