Research Statement
Srivatsan Ravi

September 6, 2016

Almost every computing system nowadays is *distributed*, ranging from multicore desktops/laptops 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. Fundamental issues that arise in reliable and efficient distributed systems include developing adequate methods for modeling *failures* and *synchrony* assumptions, and capturing the trade-off between consistency and efficiency for implementations of distributed algorithms. 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 and practical interest.

My research is primarily concerned with the theory and practice of distributed computing. Designing distributed programs though 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 [58, 29], among failure-prone processes in an asynchronous environment. Alternatively phrased, this result implies that it is impossible to achieve *consistency*, *availability* and *partition-tolerance* in an asynchronous environment [33], 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 in tackling a question concerning a distributed computation: (i) What is the model of computation? Answering this question requires identifying the *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 stop their computation, computing and memory capacity of the processes, etc. (ii) Given a computational model, is it (im)possible to build a distributed application for a specific choice of consistency and availability? (iii) What are the complexity metrics that characterize the cost of the distributed computation and can we derive bounds that identify the theoretical cost of implementing a distributed application? (iv) What are the existing hardware and programming tools available that help realize a working implementation of the distributed application and finally (v) How can we *verify* that the resulting implementation conforms to its intended semantics?

1 Previous work

A significant portion of my past studies concern synchronization algorithms for today's multicore CPU architectures whose execution can be 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 [46, 69] 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.

The following capsules provide an overview of the questions I have worked on since the start of my graduate studies.

1.1 Safety for Transactional Memory(TM) [9, 8, 11]

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 [64, 4, 59] 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* [41]. 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 [49, 25] and verify if they are indeed safety properties.

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* [13] and *disjoint-access parallelism* [50]. 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 [53, 56]. 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 [40], 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 [41] 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* [5, 12] (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* [7, 61], 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 [54]. 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 [44, 48] 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 [45, 60, 69, 70, 31] 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 [24, 23, 28]. 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 [14] in obstruction-free TMs is impossible, (ii) A read operation in a n-process obstruction-free TM implementation incurs $\Omega(n)$ *memory stalls* [27, 6]. (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 [55]. 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 [39]. (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.

1.3 Inherent limitations of Hybrid transactional memory [3, 2]

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 [46]. Most popular TM designs, subsequent to the original proposal in [46] have implemented all the functionality in software [21, 69, 45, 60, 31]. More recently, CPUs have included hardware extensions to support small transactions [65, 1, 63]. 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?

We make the following contributions which help determine the cost of instrumentation in HyTMs. (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.

1.4 Search for concurrency-optimal data structures [38, 35, 36, 37]

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.

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 [47] and the Lazy list-based sets [42], 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 [30] on Power8, SPARC and x86-64 architectures. Our algorithm is formally proved correct and concurrency-optimal.

2 Current and future work

As such, my short-term research agenda primarily involves extending some of the open questions left pending from my own prior research. However, I also plan to tackle some new classes of distributed computing problems in the immediate future and I am always motivated in introducing techniques from distributed computing to new domains.

2.1 Ongoing projects

Programming scalable cloud services [66]. 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 *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 fine-grained *elasticity* enabling the programmer to transparently migrate individual actors without violating atomicity or entailing significant performance overheads.

We have implemented a highly available and fault-tolerant prototype of *AEON* in `C++`. Extensive experiments show several complex cloud applications build 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 [20] and Orleans [16]). And the elasticity of *AEON* guarantees service quality with minimal cost compared to any static setup.

Cost of concurrency in Hybrid transactional memory. State-of-the-art *Software Transactional Memory (TM)* implementations achieve good performance by carefully avoid 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.

Synchronization protocols for software-defined networking (SDN). *Software-defined networking (SDN)* out-sources the control over the network to a logically centralized software, called the control plane. The ability of the *control plane* to "program" the network (the data plane) opens new interesting opportunities to network operators and system designers. The advent of SDN has opened up several applications ranging from traffic engineering, load balancing for application servers, inter-domain routing, security and access policy, network virtualization, all of which are vital to existing network infrastructures [15].

While the perspective of a centralized controller simplifies network management, it comes with the usual drawbacks: single point of failure, scalability bottleneck, etc. A fully centralized system may not adequately or cost-effectively provide the required levels of availability, responsiveness and scalability. This raises the question: *What kind of a distributed control plane for SDN is needed* [52]?

To treat this problem formally, the interaction between the data plane and a distributed control plane (consisting of a collection of fault-prone controllers) has been studied formally [17, 18, 51]. Thus, balancing consistency, efficiency and availability in network management becomes a classical problem of distributed computing. However, there remain several unanswered questions; for *e.g.*, how do we ensure the *consistency of data plane* when several controllers in a concurrent fashion update *network policies* (specifies the intended network behavior) on the switches? Indeed, concurrent operation on the same data plane by distributed controllers may lead to conflicts and thus, inconsistent network behaviour. How do we build scalable synchronization protocols that ensure the consistency of data plane updates while remaining resilient to potentially malicious and fault-prone controllers? To answer these questions, I investigate how the efficiency of a SDN protocol is affected by the following factors: (i) Message complexity of communication among controllers? (ii) Communication channel between controllers and the data plane elements: is it *in-band* or *off-band*? (iii) Failure model for both the controller and data plane elements? What if control elements are malicious?

Scheduling for big-data processing frameworks [67]. 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.

2.2 Expected future work

Evidently, it is far easier to explain work which I have already been done than answer questions I yet plan to do. Nonetheless, these are some of the questions I have thought about, but have not managed to write interesting papers yet.

(i) **Randomized non-blocking data structures**: As mentioned in the introduction, one of the seminal results in distributed computing, extended to the *shared memory* model, is that, in an asynchronous environment where failure-prone processes communicate by only reading and writing, two processes cannot *deterministically* achieve *consensus*: both processes initially propose a value and must eventually agree on one of the proposed values [58, 29]. The *consensus hierarchy* [43] classifies data types by a *consensus number*: a data type τ has consensus number n if it is possible to deterministically achieve *wait-free* consensus among n processes using τ and *atomic registers*, but not among $n+1$ processes. For example, *queues*, *stacks* and *test-and-set* have consensus number 2 and thus cannot be wait-free implemented from atomic registers in an asynchronous fault-prone environment. Similarly, *compare-and-swap* can solve consensus among any number of objects and thus cannot be wait-free implemented from data types like queues and stacks.

One successful approach to circumvent these impossibilities is to introduce *randomization*, thus allowing a non-faulty process to terminate only with probability *one*. There are several open questions concering the complexity bounds for wait-free (and more generally *non-blocking*) randomized implementations of popular data structures. Specifically, I am working on complexity bounds for concurrent data structures assuming scheduling models that place constraints on the ability of the *adversary* to observe the value of memory locations.

(ii) **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. Our work will hopefully provide nontrivial answers to these questions by deriving complexity bounds and implementing provably correct algorithms for *non-blocking* data structures in NVMs.

(iii) **Byzantine algorithms for distributed crypto-currencies**: *Cryptocurrencies* like *Bitcoin* [62] and *Ripple* [68] 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 [19], 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* [26]: a cryptographic puzzle that essentially allows the Bitcoin to tradeoff computation for communication complexity. However, several questions remain unanswered over the scalability and inherent complexity bounds for such protocols.

(iv) **Relaxed consistency models for geo-distributed systems**: Distributed protocols for building stateful fault-tolerant applications in data centers typically strive to provide strong consistency à la *linearizability* [43]. However, the complexity of achieving consistent replication of state while ensuring availability makes it hard to justify its deployment in *geo-distributed data centers*. The increasing popularity of replicating online services across diverse data centers emphasizes the need for low-latency protocols that cater to failure patterns specific to such a setting while still maintaining some *relaxed* form of *cross-site consistency*. One of my goals is constructing modular protocols for state-machine replication in such geo-distributed settings that are tunable to specific consistency levels and failure patterns.

(v) **Computing in oblivious shared memory**: We consider the problem of computing in an asynchronous shared memory that is *oblivious* [34]: 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 [32]. One of my goals is formalizing the notion of obliviousness in the distributed setting as well as deriving complexity bounds that characterize the complexity of building oblivious *linearizable* and *non-blocking* distributed objects.

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 [57], quantum computing [22] and in general big-data analysis, I envision introducing techniques from distributed computing to other scientific fields.

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

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] Dan Alistarh, Justin Kopinsky, Petr Kuznetsov, Srivatsan Ravi, and Nir Shavit. Inherent limitations of hybrid transactional memory. *6th Workshop on the Theory of Transactional Memory, Paris, France*, 2014.

[3] Dan Alistarh, Justin Kopinsky, Petr Kuznetsov, Srivatsan Ravi, and Nir 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. Extended version of results from [2].

[4] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

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

[6] Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4), 2009.

[7] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498, 2011.

[8] Hagit Attiya, Sandeep Hans, Petr Kuznetsov, and Srivatsan Ravi. What is safe in transactional memory. *4th Workshop on the Theory of Transactional Memory, Madeira, Portugal*, 2012.

[9] Hagit Attiya, Sandeep Hans, Petr Kuznetsov, and Srivatsan Ravi. Safety of deferred update in transactional memory. In *ICDCS*, pages 601–610, 2013.

[10] Hagit Attiya, Sandeep Hans, Petr Kuznetsov, and Srivatsan Ravi. Safety of deferred update in transactional memory. *2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)*, 0:601–610, 2013.

[11] Hagit Attiya, Sandeep Hans, Petr Kuznetsov, and Srivatsan Ravi. Safety and deferred update in transactional memory. In Rachid Guerraoui and Paolo Romano, editors, *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, volume 8913 of *Lecture Notes in Computer Science*, pages 50–71. Springer International Publishing, 2015. Extended version of results from [10] and [8].

[12] Hagit Attiya, Danny Hendler, and Philipp 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.

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

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

[15] Wolfgang Braun and Michael Menth. Software-defined networking using openflow: Protocols, applications and architectural design choices. *Future Internet*, 6(2):302–336, 2014.

[16] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. page 16, 2011.

[17] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. Software transactional networking: concurrent and consistent policy composition. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013*, pages 1–6, 2013.

[18] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A distributed sdn control plane for consistent network updates. In *Proc. 34th IEEE Conference on Computer Communications (INFOCOM)*, 2015.

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

[20] Wei-Chiu Chuang, Bo Sang, Sunghwan Yoo, Rui Gu, Milind Kulkarni, and Charles Edwin Killian. EventWave: Programming Model and Runtime Support for Tightly-coupled Elastic Cloud Applications. page 21, 2013.

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

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

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

[24] Dave Dice and Nir Shavit. What really makes transactions fast? In *Transact*, 2006.

[25] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.*, 25(5):769–799, 2013.

[26] Cynthia Dwork and Moni 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.

[27] Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012.

[28] Robert Ennals. Software transactional memory should not be obstruction-free. 2005.

[29] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[30] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC*, pages 50–59, 2004.

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

[32] Craig 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.

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

[34] 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.

[35] Vincent Gramoli, Petr Kuznetsov, and Srivatsan Ravi. From sequential to concurrent: correctness and relative efficiency (brief announcement). In *Principles of Distributed Computing (PODC)*, pages 241–242, 2012.

[36] Vincent Gramoli, Petr Kuznetsov, and Srivatsan Ravi. Sharing a sequential data structure: correctness definition and concurrency analysis. *4th Workshop on the Theory of Transactional Memory, Madeira, Portugal*, 2012.

[37] Vincent Gramoli, Petr Kuznetsov, and Srivatsan Ravi. In the search of optimal concurrency. *To appear in the Proceedings of the International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2016.

[38] Vincent Gramoli, Petr Kuznetsov, Srivatsan Ravi, and Di Shang. A concurrency-optimal list-based set (brief announcement). In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9*, 2015.

[39] Rachid Guerraoui and Michal 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.

[40] Rachid Guerraoui and Michal Kapalka. The semantics of progress in lock-based transactional memory. *SIGPLAN Not.*, 44(1):404–415, January 2009.

[41] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

[42] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2006.

[43] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, 1991.

[44] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

[45] Maurice Herlihy, Victor Luchangco, Mark Moir, and William 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.

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

[47] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[48] Maurice Herlihy and Nir Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.

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

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

[51] Naga Praveen Katta, Haoyu Zhang, Michael J. Freedman, and Jennifer Rexford. Ravana: controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Santa Clara, California, USA, June 17-18, 2015*, pages 4:1–4:12, 2015.

[52] Petr Kuznetsov. Synchronization and Concurrency in Software-Defined Networking.

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

[54] Petr Kuznetsov and Srivatsan 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.

[55] Petr Kuznetsov and Srivatsan 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.

[56] Petr Kuznetsov and Srivatsan 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.

[57] Jeff W. Lichtman, Hanspeter Pfister, and Nir Shavit. The big data challenges of connectomics. *Nature Neuroscience*, 17(11):1448–1454, October 2014.

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

[59] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[60] Virendra J. Marathe, William N. Scherer Iii, and Michael L. Scott. Adaptive software transactional memory. In *In Proc. of the 19th Intl. Symp. on Distributed Computing*, pages 354–368, 2005.

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

[62] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2009.

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

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

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

[66] Bo Sang, Gustavo Petri, Masoud Saeida Ardekani, Srivatsan Ravi, and Patrick Eugster. Programming scalable cloud services with aeon. *To appear in the Proceedings of the ACM Middleware*, 2016.

[67] Derek Schatzlein, Srivatsan Ravi, Youngtae Noh, Masoud Saeida Ardekani, and Patrick Eugster. The misbelief in delay scheduling. *To appear in proceedings of International Workshop on Distributed Cloud Computing (DCC)*, 2016.

[68] David Schwartz, Noah Youngs, and Arthur Britto. The ripple consensus protocol. 2014.

[69] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[70] Fuad Tabba, Mark Moir, James R. Goodman, Andrew W. Hay, and Cong 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.