Distributed Computing Column 58
Maurice Herlihy's 60th Birthday Celebration

Jennifer L. Welch
Department of Computer Science and Engineering
Texas A&M University, College Station, TX 77843-3112, USA
`welch@cse.tamu.edu`

Maurice Herlihy is one of the most renowned members of the Distributed Computing community. He is currently a professor in the Computer Science Department at Brown University. He has an A.B. in Mathematics from Harvard University, and a Ph.D. in Computer Science from M.I.T. He has served on the faculty of Carnegie Mellon University and on the staff of DEC Cambridge Research Lab. He is the recipient of the 2003 Dijkstra Prize in Distributed Computing, the 2004 Gödel Prize in theoretical computer science, the 2008 ISCA influential paper award, the 2012 Edsger W. Dijkstra Prize, and the 2013 Wallace McDowell award. He received a 2012 Fulbright Distinguished Chair in the Natural Sciences and Engineering Lecturing Fellowship, and he is a fellow of the ACM, a fellow of the National Academy of Inventors, and a member of the National Academy of Engineering and the American Academy of Arts and Sciences.

On the occasion of his 60th birthday, the SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), which was held in Paris, France in July 2014, hosted a celebration which included several technical presentations about Maurice's work by colleagues and friends. This column includes a summary of some of these presentations, written by the speakers themselves. In the first article, Vassos Hadzilacos overviews and highlights the impact of Maurice's seminal paper on wait-free synchronization. Then, Tim Harris provides a perspective on hardware trends and their impact on distributed computing, mentioning several interesting open problems and making connections to Maurice's work. Finally, Michael Scott gives a concise retrospective on transactional memory, another area where Maurice has been a leader. This is a joint column with the Distributed Computing Column of the Bulletin of the European Association for Theoretical Computer Science (June 2015 issue), edited by Panagiota Fatourou. Many thanks to Vassos, Tim, and Michael!

Call for contributions: I welcome suggestions for material to include in this column, including news, reviews, open problems, tutorials and surveys, either exposing the community to new and interesting topics, or providing new insight on well-studied topics by organizing them in new ways.

A Quarter-Century of Wait-Free Synchronization[1]

Vassos Hadzilacos
Department of Computer Science
University of Toronto, Canada
vassos@cs.toronto.edu

It is an honour and a pleasure to have the opportunity to speak about what in my opinion is Maurice Herlihy's most influential paper, and indeed one of the most significant papers in the theory of distributed computing. I am referring to his work on wait-free synchronization, which appeared in preliminary form in PODC 1988 [8] and in its final form in *TOPLAS* three years later [10]. I will first review the key contributions of this paper and then I will discuss its impact.

1 Review of the key contributions

The context for this work is a distributed system in which processes take steps ***asynchronously*** and communicate by accessing ***shared objects***. Here asynchrony means that between successive steps of a process other processes may take an arbitrary number of steps. Processes are subject to crash failures, meaning that they may stop taking steps altogether, even though they have not reached the end of their computation. For convenience, we assume that a process is designed to take steps (perhaps no-ops) forever, and so we can define a process to have crashed if it takes only finitely many steps in an infinite execution. Minimally, the shared objects that the processes use to communicate are registers accessible via separate but atomic write and read operations. The shared objects can also include registers with additional operations such as fetch-and-add, whereby a process atomically increments the value of the register by a specified amount and reads the value of the register before it was incremented; or even other types of shared objects, such as queues or stacks.

The key question that animates the paper is the following:

> "For given object types A and B, in a system with n processes, can we implement an object of type A using objects of type B and registers?"

[1]Remarks on the occasion of Maurice Herlihy's 60th birthday in PODC 2014. Based on the transparencies used in an oral presentation on July 15, 2014, Paris, France. I have tried to preserve the informal tone of that oral presentation here. Supported in part by a grant from the Natural Sciences and Engineering Council of Canada.

In what follows, we will take registers (with atomic write and read operations) for granted. So, the above question will be simplified to "in a system of n processes, can we implement an object of type A using objects of type B?"

Here are some specific instances of this question:

- Can we implement a queue shared by two processes using only registers? Herlihy showed that the answer to this question is negative.
- Can we implement a register with a fetch-and-add operation, shared by five processes, using registers with a compare-and-swap operation?[1] Herlihy showed that the answer to this question is affirmative.

What is significant about this paper is not so much the answer to these specific questions, but the tools that it gave us to answer such questions in general.

Having set the general context for the paper, I will now describe its main contributions.

Contribution 1: Model of computation

The type of an object specifies what operations can be applied to an object (of that type) and how the object is supposed to behave ***when operations are applied to it sequentially***. For example, the type queue tells us that we can access the object via enqueue and dequeue operations only, and that in a ***sequence*** of such operations items are dequeued in the order in which they were enqueued. But how should a shared queue behave if operations are applied to it by processes concurrently?

More generally, what exactly are the properties that an implementation of an object of type A (shared by n processes) should have? Herlihy requires two properties of such an implementation: linearisability and wait freedom.[2]

Linearisability: The implemented object should behave as if each operation took effect instantaneously, at some point between its invocation and its response.

Wait freedom: An operation on the implemented object invoked by a nonfaulty process eventually terminates, regardless of whether other processes are fast, slow, or even crash.

Note that the requirement of wait freedom implies that implementations

(a) may not use locks: otherwise, a process that crashes while holding a lock could prevent all others from terminating, and

(b) must be starvation free: not only must the system as a whole make progress but each individual nonfaulty process must complete all its operations.

The first important contribution of the paper was the articulation of ***a compelling, elegant, and pragmatic model of computation***.

[1]A compare-and-swap operation applied to register X takes two parameters, values *old* and *new*, and has the following effect (atomically): If the present value of X is equal to *old* then X is assigned the value *new* and the value "success" is returned; otherwise, the value of X is not changed and the value "failure" is returned.

[2]In my oral presentation, I referred to linearisability as a safety property, and to wait freedom as a liveness property. Rachid Guerraoui, who was in the audience, brought to my attention a paper of his with Eric Ruppert in which they show that this is not quite right [6]: There are types with infinite non-determinism for which linearisability is not a safety property; for types with bounded non-determinism, however, linearisability is indeed a safety property.

Contribution 2: Comparing the power of object types

Recall the basic question the paper addresses: In a system of n processes, can we implement an object of type A using objects of type B? An affirmative answer to such a question presents no methodological difficulties: One presents an implementation and a proof that it satisfies the two requisite properties. But what if the answer is negative? How can we prove that A cannot be implemented using B? One way to do so is to show that there is some task C that A can do, and that B cannot do. So, task C is a "yardstick" that can be used to compare A and B. Another key contribution of the paper is ***the identification of the right kind of yardstick*** to compare types, namely, solving the ***consensus problem***. This problem, which under various forms and guises had been studied extensively in fault-tolerant distributed computing before Herlihy's paper, can be described as follows:

- Each process starts with an input.
- Each nonfaulty process produces an output.
- The output of any process is the input of some process (validity), and is no different than the output of any other process (agreement).

Note that we are interested in ***wait-free*** solutions for this problem. Let us examine some examples of the use of this "yardstick" to prove non-implementability results.

Example 1: To show that in a system of two processes we cannot implement a queue using registers we prove that

(1) using queues we can solve consensus for two processes; and

(2) using registers we cannot solve consensus for two processes.

From (1) and (2) we conclude that we cannot implement queues using only registers: For, if we could, we would combine such an implementation with (1) to obtain an algorithm that uses registers and solves consensus for two processes, contradicting (2).

Example 2: To show that in a system of three processes we cannot implement a register with the compare-and-swap operation using registers with a fetch-and-add operation we prove that

(1) using registers with compare-and-swap we can solve consensus for three processes; and

(2) using registers with fetch-and-add we cannot solve consensus for three processes.

Using similar reasoning as in Example 1, from (1) and (2) we conclude that we cannot implement compare-and-swap using only fetch-and-add.

So, to capture the "power" of an object type A, Herlihy attaches to A a ***consensus number***, namely the unique integer n such that:

- using objects of type A we can solve consensus for n processes, and
- using objects of type A we cannot solve consensus for $n+1$ processes.

If no such integer n exists, the consensus number of A is ∞. The following, methodologically very useful, theorem follows immediately from this definition.

Theorem 1.1 ([8, 10])**.** *If type A has consensus number n and type B has consensus number $m < n$, then A cannot be implemented from B in a system with more than m processes.*

This leads us to Herlihy's ***consensus hierarchy*** of object types: A type A is at level n of the consensus hierarchy if and only if its consensus number is n — i.e., if and only if A solves consensus for n, but not for $n + 1$, processes. Thus, by Theorem 1.1, "stronger" types are at higher levels of this hierarchy.

Figure 1 illustrates the consensus hierarchy. I now briefly explain the types mentioned in that figure that I have not already defined.

- The test-and-set type (at level 2) refers to a register initialised to 0, with an operation that atomically sets the register to 1 and returns the value of the register before the operation. (So, the operation that is linearised first returns 0, and all others return 1.)
- The n-consensus type (at level n) refers to an object with a PROPOSE(v) operation, where v is an arbitrary value (say a natural number); the object returns the value proposed by the first operation to the first n PROPOSE operations applied to it, and returns an arbitrary value to subsequent operations. (Thus, it is an object designed to solve consensus for n processes.)
- The n-peekable queue type (also at level n) refers to a kind of queue to which a maximum of n values can be enqueued (any values subsequently enqueued are lost) and which allows a process to "peek" at the first value enqueued without dequeuing it.
- The n-assignment type (at level $2n - 2$) allows a process to atomically assign n specified values to n specified registers.
- The consensus type (at level ∞) is similar to n consensus, except that it returns to all PROPOSE operations (not only to the first n) the value proposed by the first one.
- Finally, the memory-to-memory swap type (also at level ∞) allows a process to atomically swap the values of two specified registers.

Contribution 3: Universality of consensus

We have seen how Herlihy used consensus as a "yardstick" to compare the relative power of object types. But why is consensus the ***right*** yardstick? In principle, we could have taken any task and used it as a yardstick. For example, consider the ***leader election*** problem:

- Each nonfaulty process outputs "winner" or "loser".
- At most one process outputs "winner".
- Some process outputs "winner" or crashes after taking at least one step.

We could define the "leader election number" of type A to be the maximum number of processes for which A can solve the leader election problem — by analogy to the definition of the consensus number, but using a different problem as the yardstick. There is nothing in principle wrong with this, except that the resulting "leader election hierarchy" would not be very interesting: it would consist of just two levels: all types in levels two to infinity of the consensus hierarchy would

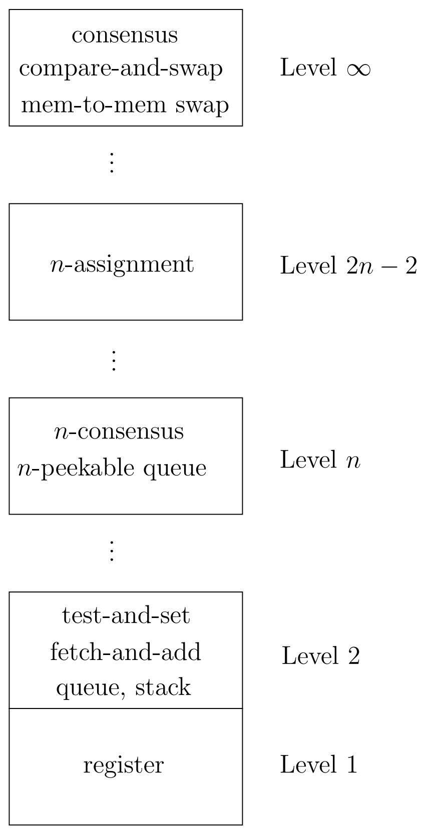

Figure 1: The consensus hierarchy

coalesce into a single level! In other words, unlike consensus, the leader election yardstick is not a very discriminating one. So, what is special about consensus that makes it the right yardstick? The answer lies in the following important fact:

Theorem 1.2 ([8, 10])**.** *Any object type B with consensus number n is* ***universal*** *for n processes: it can implement an object of* ***any*** *type A, shared by n processes.*

The proof of this theorem is through an intricate algorithm that has come to be known as ***Herlihy's universal construction***. Given a function that defines the sequential behaviour of an arbitrary type A, this construction shows how to implement an object of type A shared by n processes using only registers and n-consensus objects. So, given any object of type B with consensus number n, we can solve the consensus problem for n processes (by definition of consensus number), and therefore we can implement n-consensus objects. Then, using Herlihy's universal construction, we can implement an object of type A shared by n processes.

At a very high level, the intuition behind this theorem is simple: Processes use consensus to agree on the order in which to apply their operations on the object they implement. Between this intuition and an actual working algorithm that satisfies wait freedom, however, there is a significant gap. Herlihy's universal construction is an algorithm well worth studying carefully, and returning to every now and then!

2 Impact

The impact of the paper is accurately reflected by its citation count. A Google Scholar search conducted in July 2014 showed over 1400 citations for [10] and over 200 for [8]. Let us look beyond the numbers, however, into the specific ways in which Herlihy's paper on wait-free synchronisation has influenced the field of distributed computing.

Impact 1: The model

The model of asynchronous processes communicating via linearisable, wait-free shared objects that was articulated in a complete form in this paper has been a very influential one. As noted earlier, it is mathematically elegant but also pragmatic. It is certainly true that different aspects of this model appeared earlier, but I believe that this was the first paper that presented the complete package. It is nevertheless useful to trace the heritage.

Shared memory: The asynchronous shared memory model goes back to Dijkstra's seminal paper on mutual exclusion [3].

Wait freedom: The concept of wait-free implementations (though not under this name) originated in Lamport's and Peterson's work on implementations of shared registers [15, 19, 16, 17].

Linearisability: The concept of linearisability as the correctness criterion for the behaviour of shared objects was introduced by Herlihy and Wing [12, 13].

Impact 2: Lock-free data structures

The idea of synchronising access to data structures without relying on locks has had a significant impact on the practice of concurrent programming. Although locking is still (and may well remain) the predominant mechanism employed to coordinate access to data structures by multiple processes, Herlihy's paper helped highlight some of its shortcomings (potential for deadlock, unnecessary restrictions to concurrency, intolerance to even crash failures, priority inversions) and pointed the way to the possibility of synchronising without using locks. There is, by now, an extensive literature on so-called ***lock-free*** data structures. In this context, lock free doesn't necessarily mean wait free. It is a term that encompasses wait freedom as well as the weaker ***non-blocking*** property, which requires that progress be made by ***some*** non-faulty process, not necessarily ***every*** non-faulty process.[3]

Impact 3: Weaker liveness properties

Linearisable wait-free implementations tend to be complex, and one culprit seems to be wait freedom. The most intricate aspect of Herlihy's universal construction is the so-called helping mechanism, which ensures that "no process is left behind". If one is willing to settle for the less demanding non-blocking property, the universal construction becomes much simpler.

[3]The terms "lock free" and "non-blocking" are not used consistently in the literature; in some papers their meaning is as given here, in others it is reversed.

The observation that wait freedom seems to complicate things and that it is perhaps too strong a liveness property has led researchers to investigate other liveness properties, weaker than wait freedom, easier to implement, but hopefully still useful in practice. The following are some examples of objects with relaxed liveness requirements:

Obstruction-free objects: Every operation invoked by a nonfaulty process that eventually runs solo (i.e., without interference from other processes) terminates [4, 11].

"Pausable" objects: Every operation invoked by a live process eventually returns control to the caller, either by completing normally, or by aborting without taking effect, or by "pausing" so that another operation can run solo and terminate. An operation can abort or pause only if it encounters interference. A nonfaulty process whose operation was paused is required to resume the paused operation and complete it (normally or by aborting) before it can do anything else [2].

Nondeterministic abortable objects: Every operation invoked by a nonfaulty process eventually returns to the caller either by completing normally or by aborting. An operation can abort only if it encounters interference. An aborted operation may or may not have taken effect, and the caller doesn't know which of these two possibilities is the case [1].

Abortable objects: Every operation invoked by a nonfaulty process eventually returns to the caller either by completing normally or by aborting. An operation can abort only if it encounters interference. An aborted operation is guaranteed not to have taken effect [7].

Impact 4: Structure of the "A implemented by B" relation

Though the consensus number of an object type A encapsulates much information about A's ability to implement other types, it does not tell the whole story. By Theorem 1.2, if A has consensus number n, it can support the implementation of any object shared by n processes; but what about the implementation of even "weak" objects, i.e., objects of types whose consensus number is no greater than n, ***shared by more than n processes***? In this setting, there are phenomena that run counter to the notion that the higher the consensus number of a type the greater its power to implement other types.

Consider the following question: Are all object types at the same level of the consensus hierarchy equivalent? That is, if A and B are two types at the same level n of the consensus hierarchy, can an object of type A, shared by ***any*** number m of processes, be implemented using objects of type B? Or, equivalently (in view of Theorem 1.2), can any object of a type with consensus number n, shared by ***any*** number of processes, be implemented using n-consensus? Herlihy himself proved that this is not the case for level 1: He demonstrated a type at level 1 that cannot be implemented from registers (which are also at level 1) [9]. Rachman proved that this is the case for every level [20]: For every positive integer n, he demonstrated a type T_n at level n of the consensus hierarchy such that an object of type T_n shared by $2n+1$ processes cannot be implemented using n-consensus objects.[4] (In fact, Rachman's result is more general: for any positive integers n, m such that $m \leq n$, there is a type T_m at level m of the consensus hierarchy such that an object of type T_m, shared by $2n+1$ processes, cannot be implemented using n-consensus objects.)

[4]My account in this paragraph differs from my oral presentation in Paris, as a result of things I learned in the meanwhile — but should have known then!

A related set of investigations concern the matter of "robustness" of the consensus hierarchy. Consider a system with n processes. By the definition of consensus number, objects of a type with consensus number less than n cannot implement an n-consensus object. Is it possible, however, to use objects of ***multiple*** "weak" types (with consensus number less than n) to implement n-consensus? If this is possible, we say that the consensus hierarchy is ***not robust***. Jayanti was the first to identify and study the issue of robustness; he proved that under a restricted definition of implementation of one type by others, the consensus hierarchy is not robust [14]. Later, Schenk proved that under a restricted definition of wait freedom, the consensus hierarchy is not robust [21]. Lo and Hadzilacos proved that under the usual definitions of implementation and wait freedom, the consensus hierarchy is not robust [18].

Impact 5: Elevating the status of the bivalency argument

George Pólya and Gabor Szegö made a famous quip about the distinction between a trick and a method:

> "An idea that can be used only once is a trick. If one can use it more than once, it becomes a method." (*Problems and Theorems in Analysis*, 1972.)

Fischer, Lynch, and Paterson gave us the bivalency argument as a brilliant trick in their proof of the impossibility of consensus in asynchronous message-passing systems [5]. With his masterful use of the same argument to prove that consensus among n processes cannot be solved using objects of type B (for several choices of n and B), Herlihy elevated bivalency to the more exalted status of a method!

Impact 6: Design of multiprocessors?

I put a question mark for this impact, because here I am speculating: I do not really know why, in the late 1980s and early 1990s, multiprocessor architects abandoned operations with low consensus number in favour of universal ones. But the timing is such that I wouldn't be surprised to learn that these architects were influenced, at least in part, by Herlihy's discovery that, from the perspective of wait-free synchronisation, much more is possible with operations such as compare-and-swap or load-linked/store-conditional than with operations such as test-and-set or fetch-and-add.

Great papers answer important questions, but also open new ways of thinking, and perhaps even influence practice. Herlihy's paper on wait-free synchronisation delivers on all these counts!

Acknowledgements

I am grateful to Naama Ben-David and David Chan for their comments on this paper.

References

[1] Marcos K. Aguilera, Sven Frolund, Vassos Hadzilacos, Stephanie Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC '07: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 23–32, 2007.

[2] Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. Computing with reads and writes in the absence of step contention. In *DISC '05: Proceedings of the 19th International Symposium on Distributed Computing*, pages 122–136, 2005.

[3] Edgar W. Dijkstra. Solution of a problem in concurrent programming control. *Commununications of the ACM*, 8(9):569, 1965.

[4] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, 1998.

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

[6] Rachid Guerraoui and Eric Ruppert. Linearizability is not always a safety property. In *Networked Systems - Second International Conference, NETYS 2014*, pages 57–69, 2014.

[7] Vassos Hadzilacos and Sam Toueg. On deterministic abortable objects. In *PODC '13: Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing*, pages 4–12, 2013.

[8] Maurice Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC '88: Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 276–290, 1988.

[9] Maurice Herlihy. Impossibility results for asynchronous PRAM. In *SPAA '91: Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 327–336, 1991.

[10] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[11] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Maurice Herlihy and Jeannette Wing. Axioms for concurrent objects. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 13–26, New York, NY, USA, 1987. ACM Press.

[13] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[14] Prasad Jayanti. On the robustness of Herlihy's hierarchy. In *PODC '93: Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 145–157, 1993.

[15] Leslie Lamport. On concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[16] Leslie Lamport. On interprocess communication. Part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[17] Leslie Lamport. On interprocess communication. Part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[18] Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of is: wait-free hierarchies are not robust. In *STOC '97: In Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 579–588, 1997.

[19] Gary Peterson. Concurrent reading while writing. *ACM Transactions of Programming Languages and Systems*, 5(1):46–55, 1983.

[20] Ophir Rachman. Anomalies in the wait-free hierarchy. In *WDAG '94: Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 156–163, 1994.

[21] Eric Schenk. The consensus hierarchy is not robust. In *PODC '97: In Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, page 279, 1997.

Hardware Trends: Challenges and Opportunities in Distributed Computing

Tim Harris
Oracle Labs
Cambridge, UK
`timothy.l.harris@oracle.com`

This article is about three trends in computer hardware, and some of the challenges and opportunities that I think they provide for the distributed computing community. A common theme in all of these trends is that hardware is moving away from assumptions that have often been made about the relative performance of different operations (e.g., computation versus network communication), the reliability of operations (e.g., that memory accesses are reliable, but network communication is not), and even some of the basic properties of the system (e.g., that the contents of main memory are lost on power failure).

Section 1 introduces "rack-scale" systems and the kinds of properties likely in their interconnect networks. Section 2 describes challenges in systems with shared physical memory but without hardware cache coherence. Section 3 discusses non-volatile byte-addressable memory. The article is based in part on my talk at the ACM PODC 2014 event in celebration of Maurice Herlihy's sixtieth birthday.

1 Rack-Scale Systems

Rack-scale computing is an emerging research area concerned with how we design and program the machines used in data centers. Typically, these data centers are built from racks of equipment, with each rack containing dozens of discrete machines. Over the last few years researchers have started to weaken the boundaries between these individual machines, leading to new "rack-scale" systems. These architectures are being driven by the need to increase density and connectivity between servers, while lowering cost and power consumption.

Different researchers mean somewhat different things by "rack-scale" systems. Some systems are built from existing components. These are packaged together for a particular workload, providing appropriate hardware, and pre-installed software. Other researchers mean systems with internal disaggregation of components: rather than having a rack of machines each with its own network interface and disk, there might be a pool of processor nodes, disk nodes, and networking nodes, all connected over an internal intra-machine interconnect. The interconnect can be configured to connect sets of these resources together in different ways.

Initial commercial systems provide high-density processor nodes connected through an in-machine interconnect to storage devices or to external network interfaces. Two examples are the HP MoonShot [12] and AMD SeaMicro [22] single-box cluster computers. Many further ideas are now being explored in research projects—for instance, the use of custom system-on-chip (SoC) processors in place of commodity chips.

These systems should not just be seen as a way to build a faster data center. Communicating over a modern interconnect is different from communicating over a traditional packet-switched network. Some differences are purely trends in performance—a round-trip latency for over InfiniBand is around 1μs, not much longer than the time it takes to access data stored in DRAM on a large shared-memory multiprocessor. The Scale-Out NUMA architecture provides one example of how latencies may be reduced even further: it exposes the interconnect via a specialized "remote memory controller" (RMC) on a multi-core SoC [18]. Threads in one SoC can instruct the RMC to transfer data to or from memory attached to other processors in the system. Threads communicate with their RMC over memory-mapped queues (held in the SoC's local caches). These operations have much lower latency than accessing a traditional network interface over PCI-express. If network latencies continue to fall, while memory access latencies remain constant, then this will change the optimization goals when designing a protocol.

Other differences are qualitative: as with the Scale-Out NUMA RMC, the main programming interface in many rack-scale systems is RDMA (remote direct memory access). To software, RDMA appears as a transfer from a region of a sender's address space into a region in the receiver's address space. Various forms of control message and notification can be used—e.g., for a receiver to know when data has arrived, or for a sender to know when transmission is complete. Flow control is handled in hardware to prevent packet loss.

Some network devices provide low-latency hardware distribution of data to multiple machines at once (for instance, the ExaLINK matrix switches advertise 5ns latency multicasting data from an input port to any number of output ports [1]). Researchers are exploring how to use this kind of hardware as part of an atomic broadcast mechanism [7].

Research questions: What are the correct communication primitives to let applications benefit from low-latency communication within the system? What are the likely failure modes and how do we achieve fault tolerance? What is the appropriate way to model the guarantees provided by the interconnect fabric in a rack-scale system? How should the interconnect fabric be organized, and how should CPUs, DRAM, and storage be placed in it?

2 Shared Memory Without Cache Coherence

The second trend I will highlight is toward systems with limited support for cache coherence in hardware: Some systems provide shared physical memory, but rely on threads to explicitly flush and invalidate their local caches if they want to communicate through them. Some researchers argue that cache coherence will be provided within a chip, but not between chips [15].

This kind of model is not entirely new. For instance, the Cray T3D system distributed its memory across a set of processor nodes, providing each node with fast access to its local memory, and slower access to uncacheable remote memory [6]. This kind of model makes it important to keep remote memory accesses rare because they will be slow even in the absence of contention (for instance, lock implementations with local spinning are well suited in this setting [16]).

One motivation for revisiting this kind of model is to accommodate specialized processors or accelerators. The accelerator can transfer data to and from memory (and sometimes to and from the caches of the traditional processors) but does not need to participate in a full coherence protocol. A recent commercial example of this kind of system is the Intel Xeon Phi co-processor accessed over PCI-express [13].

A separate motivation for distributing memory is to provide closer coupling between storage and computation. The IRAM project explored an extreme version of this with the processor on the same chip as its associated DRAM [19]. Close coupling between memory and storage can improve the latency and energy efficiency of memory accesses, and permit the aggregate bandwidth to memory to grow by scaling the number of memory-compute modules.

Some research systems eschew the direct use of shared memory and instead focus on programming models based on message passing. Shared memory buffers can be used to provide a high-performance implementation of message passing (for instance, by using a block of memory as a circular buffer to carry messages). This approach means that only the message passing infrastructure needs to be aware of the details of the memory system. Also, it means that software written for a genuinely distributed environment is able to run correctly (and hopefully more quickly) in an environment where messages stay within a machine.

Systems such as K2 [14] and Popcorn [4] provide abstractions to run existing shared-memory code in systems without hardware cache coherence, using ideas from distributed shared memory systems.

Conversely, the Barrelfish [5] and FOS [23] projects have been examining the use of distributed computing techniques within an OS. Barrelfish is an example of a *multikernel* in which each core runs a separate OS kernel, even when the cores operate in a single cache-coherent machine. All interactions between these kernels occur via message-passing. This design avoids the need for shared-memory data structures to be managed between cores, enabling a single system to operate across coherence boundaries. While it is elegant to rely solely on message passing, this approach seems better suited to some workloads than to others—particularly when multiple hardware threads share a cache, and could benefit from spatial and temporal locality in the data they are accessing.

Research questions: What programming models and algorithms are appropriate for systems which combine message passing with shared memory? To what extent should systems with shared physical memory (without cache coherence) be treated differently from systems without any shared memory at all?

3 Non-Volatile Byte-Addressable Memory

There are many emerging technologies that provide non-volatile byte-addressable memory (NV-RAM). Unlike ordinary DRAM, memory contents are preserved on power loss. Unlike traditional disks, locations can be read or written at a fine granularity—nominally individual bytes, although in practice hardware will transfer complete cache lines. Furthermore, unlike a disk, these reads and writes may be performed by ordinary memory access instructions (rather than using RDMA, or needing the OS to orchestrate block-sized transfers to or from a storage device).

This kind of hardware provides the possibility of an application keeping all of its data structures accessible in main memory. Researchers are starting to explore how to model NV-RAM [20]. Techniques from non-blocking data structures provide one starting point for building on NV-RAM. A power loss can be viewed as a failure of all of the threads accessing a persistent object. However, there are several challenges which complicate matters:

First, the memory state seen by the threads before the power loss is not necessarily the same as the state seen after recovery. This is because, although the NV-RAM is persistent, the remainder of the memory system may hold data in ordinary volatile buffers such as processor caches and memory controllers. When power is lost, some data will transiently be in these volatile buffers. Aggressively flushing every update to NV-RAM may harm performance. Some researchers have explored flushing updates upon power-loss, but that approach requires careful analysis to ensure that there is enough residual power to do so [17].

The second problem is that applications often need to access several structures—for instance, removing an item from one persistent collection object, processing it, and adding it to another persistent collection. If there is a power loss during the processing step, then we do not want to lose the item.

Transactions provide one approach for addressing these two problems. It may be possible to optimize the use of cache flush/invalidate operations to ensure that data is genuinely persistent before a transaction commits, while avoiding many individual flushes while the transaction executes. As with transactional memory systems, transactions against NV-RAM would provide a mechanism for composing operations across multiple data structures [10]. What is less clear is whether transactions are appropriate for long-running series of operations (such as the example of processing an object when moving it between persistent collections).

Having an application's data structures in NV-RAM could be a double-edged sword. It avoids the need to define translations between on-disk and in-memory formats, and it avoids the time taken to load data into DRAM for processing. This time saving is significant in "big data" applications, not least when restarting a machine after a crash. However, explicit loading and saving has benefits as well as costs: It allows in-memory formats to change without changing the external representation of data. It allows external data to be processed by tools in a generic way without understanding its internal formats (backup, copying, de-duplication, etc.). It provides some robustness against transient corruption of in-memory formats by restarting an application and re-loading data.

It is difficult to quantify how significant these concerns will be. Earlier experience with persistent programming languages explored many of these issues [3]. Recent work on dynamic software updates is also relevant (e.g., Arnold and Kaashoek in an OS kernel [2], and Pina *et al.* in applications written in Java [21]).

Research questions: How should software manage data held in NV-RAM, and what kinds of correctness properties are appropriate for a data structure that is persistent across power loss?

4 Discussion

This article has touched on three areas where developments in computer hardware are changing some of the traditional assumptions about the performance and behavior of the systems we build on.

Processor clock rates are not getting significantly faster (and, many argue, core counts are unlikely to increase much further [9]). Nevertheless, there are other ways in which system performance can improve such as by integrating specialized cores in place of general-purpose ones, or by providing more direct access to the interconnect, or by removing the need to go through traditional storage abstractions to access persistent memory.

I think many of these trends reflect a continued blurring of the boundaries between what constitutes a “single machine” versus what constitutes a “distributed system”. Reliable interconnects are providing hardware guarantees for message delivery, and in some cases this extends to guarantees about message ordering as well even in the presence of broadcast and multicast messages. Conversely, the move away from hardware cache coherence within systems means that distributed algorithms become used in systems which look like single machines—e.g., in the Hare filesystem for non-cache-coherent multicores [8].

Many of these hardware developments have been proceeding ahead of the advancement of formal models of the abstractions being built. Although the use of verification is widespread at low levels of the system – especially in hardware – I think there are important opportunities to develop new models of the abstractions exposed to programmers. There are also opportunities to influence the direction of future hardware evolution—perhaps as with how the identification of the consensus hierarchy pointed to the use of atomic compare and swap in today’s multiprocessor systems [11].

References

[1] EXALINK Fusion (web page). Apr. 2015. `https://exablaze.com/exalink-fusion`.

[2] J. Arnold and M. F. Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proc. 4th European Conference on Computer Systems (EuroSys)*, pages 187–198, 2009.

[3] M. Atkinson and M. Jordan. A review of the rationale and architectures of PJama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform. Technical report, University of Glasgow, Department of Computing Science, 2000.

[4] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. Popcorn: bridging the programmability gap in heterogeneous-ISA platforms. In *EuroSys ’15: Proc. 10th European Conference on Computer Systems (EuroSys)*, page 29, 2015.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP ’09: Proc. 22nd Symposium on Operating Systems Principles*, pages 29–44, 2009.

[6] Cray Research Inc. *CRAY T3D System Architecture Overview Manual.* 1993. `ftp://ftp.cray.com/product-info/mpp/T3D_Architecture_Over/T3D.overview.html`.

[7] M. P. Grosvenor, M. Fayed, and A. W. Moore. Exo: atomic broadcast for the rack-scale computer. 2015. `http://www.cl.cam.ac.uk/~mpg39/pubs/workshops/wrsc15-exo-abstract.pdf`.

[8] C. Gruenwald III, F. Sironi, M. F. Kaashoek, and N. Zeldovich. Hare: a file system for non-cache-coherent multicores. In *EuroSys '15: Proc. 10th European Conference on Computer Systems*, page 30, 2015.

[9] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.

[10] T. Harris, M. Herlihy, S. Marlow, and S. Peyton Jones. Composable memory transactions. In *PPoPP '05: Proc. 10th Symposium on Principles and Practice of Parallel Programming*, June 2005.

[11] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[12] HP Moonshot system: a new class of server. `http://www.hp.com/go/moonshot`, Accessed 9 July 2014.

[13] Intel Corporation. Intel Xeon Phi coprocessor system software developers guide. 2012. IBL Doc ID 488596.

[14] F. X. Lin, Z. Wang, and L. Zhong. K2: a mobile operating system for heterogeneous coherence domains. In *ASPLOS '14: Proc. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 285–300, 2014.

[15] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, 2012.

[16] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[17] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS '12: Proc. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410, 2012.

[18] S. Novaković, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-Out NUMA. In *ASPLOS '14: Proc. 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[19] D. A. Patterson, K. Asanovic, A. B. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. E. Kozyrakis, D. B. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. A. Yelick. Intelligent RAM (IRAM): the industrial setting, applications and architectures. In *Proceedings 1997 International Conference on Computer Design: VLSI in Computers & Processors, ICCD '97, Austin, Texas, USA, October 12-15, 1997*, pages 2–7, 1997.

[20] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[21] L. Pina, L. Veiga, and M. Hicks. Rubah: DSU for Java on a stock JVM. In *OOPSLA '14: Proc. Conference on Object-Oriented Programming Languages, Systems, and Applications*, Oct. 2014.

[22] A. Rao. SeaMicro SM10000 system overview, June 2010. `http://www.seamicro.com/sites/default/files/SM10000SystemOverview.pdf`.

[23] D. Wentzlaff and A. Agarwal. Factored operating systems (FOS): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, Apr. 2009.

Transactional Memory Today[1]

Michael Scott
Computer Science Department
University of Rochester, NY, USA
scott@cs.rochester.edu

It was an honor and a privilege to be asked to participate in the celebration, at PODC 2014, of Maurice Herlihy's many contributions to the field of distributed computing—and specifically, to address the topic of transactional memory, which has been a key component of my own research for the past decade or so.

When introducing transactional memory ("TM") to people outside the field, I describe it as a sort of magical merger of two essential ideas, at different levels of abstraction. First, at the language level, TM allows the programmer to specify that certain blocks of code should be atomic without saying how to *make* them atomic. Second, at the implementation level, TM uses speculation (much of the time, at least) to execute atomic blocks in parallel whenever possible. Each dynamic execution of an atomic block is known as a *transaction.* The implementation guesses that concurrent transactions will be mutually independent. It then monitors their execution, backing out and retrying if (and hopefully only if) they are discovered to conflict with one another.

The second of these ideas—the speculative implementation—was the focus of the original TM paper, co-authored by Maurice with Eliot Moss [22]. The first idea—the simplified model of language-level atomicity—is also due largely to Maurice, but was a somewhat later development.

1 Motivation

To understand the original motivation for transactional memory, consider the typical method of a nonblocking concurrent data structure. The code is likely to begin with a "planning phase" that peruses the current state of the structure, figuring out the operation it wants to perform, and initializing data—some thread-private, some visible to other threads—to describe that operation. At some point, a critical *linearizing instruction* transitions the operation from "desired" to "performed." In some cases, the identity of the linearizing instruction is obvious in the source code; in others it can be determined only by reasoning in hindsight over the history of the structure. Finally,

[1]Based on remarks delivered at the Maurice Herlihy 60th Birthday Celebration, Paris, France, July 2014

the method performs whatever "cleanup" is required to maintain long-term structural invariants. Nonblocking progress is guaranteed because the planning phase has no effect on the logical state of the structure, the linearizing instruction is atomic, and the cleanup phase can be performed by any thread—not just the one that called the original operation.

Two issues make methods of this sort very difficult to devise. The first is the need to effect the transition from "desired" to "performed" with a single atomic instruction. The second is the need to plan correctly in the face of concurrent changes by other threads. By contrast, an algorithm that uses a coarse-grained lock faces neither of these issues: writes by other threads will never occur in the middle of its reads; reads by other threads will never occur in the middle of its writes.

2 The Original Paper

While Maurice is largely celebrated for his theoretical contributions, the original TM paper was published at ISCA, the leading architecture conference, and was very much a hardware proposal. We can see this in the subtitle—"Architectural Support for Lock-Free Data Structures"—and the abstract: "[TM is] . . . intended to make lock-free synchronization as efficient (and easy to use) as conventional techniques based on mutual exclusion."

The core idea is simple: a transaction runs almost the same code as a coarse-grain critical section, but with special load and store instructions, and without the actual lock. The special instructions allow the hardware to track conflicts between concurrent transactions. A special end-of-transaction commit instruction will succeed (and make transactionally written values visible to other threads) only if no concurrent conflicting transaction has committed. Here "conflict" means that one transaction writes a cache line that another reads or writes. Within a transaction, a special validate instruction allows code to determine whether it still has a chance to commit successfully—and in particular, whether the loads it has performed to date remain mutually consistent. In response to a failed validate or commit, the typical transaction will loop back (in software) and start over.

Looking back with the perspective of more than 20 years, the original TM paper appears remarkably prescient. Elision of coarse-grain locks remains the principal use case for TM today, though the resulting algorithms are "lock-free" only in the informal sense of "no application-level locks," not in the sense of livelock-free. Like almost all contemporary TM hardware, Herlihy & Moss (H&M) TM was also a "best-effort-only" proposal: a transaction could fail due not only to conflict or to overflow of hardware buffers, but to a variety of other conditions—notably external interrupts or the end of a scheduling quantum. Software must be prepared to fall back to a coarse-grain lock (or some other hybrid method) in the event of repeated failures.

Speculative state (the record of special loads and stores) in the H&M proposal was kept in a special "transactional cache" alongside the "regular" cache (in 1993, processors generally did not have multiple cache layers). This scheme is still considered viable today, though commercial offerings vary: the Intel Haswell processor leverages the regular L1 data cache [40]; Sun's unreleased Rock machine used the processor store buffer [10]; IBM's zEC12 uses per-core private L2s [25].

In contrast with current commercial implementations, H&M proposed a "responder wins" coherence strategy: if transaction A requested a cache line that had already been speculatively read or written by concurrent transaction B, B would "win" and A would be forced to abort. Current machines generally do the opposite: "responder loses"—kill B and let A continue. Responder-loses has the advantage of compatibility with existing coherence protocols, but responder-wins turns out to be considerably less vulnerable to livelock. Nested transactions were not considered by H&M, but current commercial offerings address them only by counting, and subsuming the inner transactions in the outer: there is no way to abort and retry an inner transaction while keeping the outer one live.

Perhaps the most obvious difference between H&M and current TM is that the latter uses "modal" execution, rather than special loads and stores: in the wake of a special tm-start instruction, all ordinary memory accesses are considered speculative. In keeping with the technology of the day, H&M also assumed sequential consistency; modern machines must generally arrange for tm-start and commit instructions to incorporate memory barriers.

While designers of modern systems—both hardware and software—think of speculation as a fundamental design principle—comparable to caching in its degree of generality—this principle was nowhere near as widely recognized in 1993. In hindsight, the H&M paper (which doesn't even mention the term) can be seen not only as the seminal work on TM, but also as a seminal work in the history of speculation.

3 Subsequent Development

Within the architecture community, H&M TM was generally considered too ambitious for the hardware of the day, and was largely ignored for a decade. There was substantial uptake in the theory community, however, where TM-like semantics were incorporated into the notion of universal constructions [3, 5, 24, 28, 35]. In 1997, Shavit and Touitou coined the term "Software Transactional Memory," in a paper that shared with H&M the 2012 Dijkstra Prize [33].

And then came multicore. With the end of uniprocessor performance scaling, the difficulty of multithreaded programming became a sudden and pressing concern for researchers throughout academia and industry. And with advances in processor technology and transistor budgets, TM no longer looked so difficult to implement. Near-simultaneous breakthroughs in both software and hardware TM were announced by several groups in the early years of the 21st century.

Now, another decade on, perhaps a thousand TM papers have been published (including roughly a third of my own professional output). Plans are underway for the 10th annual ACM TRANSACT workshop. Hardware TM has been incorporated into multiple "real world" processors, including the Azul Vega 2 and 3 [7]; Sun Rock [10]; IBM Blue Gene/Q [36], zEnterprise EC12 [25], and Power8 [6]; and Intel Haswell [40]. Work on software TM has proven even more fruitful, at least from a publications perspective: there are many more viable implementation alternatives—and many more semantic subtleties—than anyone would have anticipated back in 2003. TM language extensions have become the synchronization mechanism of choice in the Haskell community [16], official extensions for C++ are currently in the works (a preliminary version [1] already ships in gcc), and research-quality extensions have been developed for a wide range of other languages.

4 Maurice's Contributions

Throughout the history of TM, Maurice has remained a major contributor. The paragraphs here touch on only a few of his many contributions. With colleagues at Sun, Maurice co-designed the DSTM system [18], one of the first software TMs with semantics rich enough—and overheads low enough—to be potentially acceptable in practice. Among its several contributions, DSTM introduced the notion of out-of-band *contention management*, a subject on which Maurice also collaborated with colleagues at EPFL [13, 14]. By separating safety and liveness, contention managers simplify both STM implementation and correctness proofs.

In 2005, Maurice collaborated with colleagues at Intel on mechanisms to virtualize hardware transactions, allowing them to survive both buffer overflows and context switches [30]. He also began a series of papers, with colleagues at Brown and Swarthmore, on transactions for energy efficiency [12]. With student Eric Koskinen, he introduced *transactional boosting* [20], which refines the notion of conflict to encompass the possibility that concurrent operations on abstract data types, performed within a transaction, may commute with one another at an abstract level—and thus be considered non-conflicting—even when they would appear to conflict at the level of loads and stores. With student Yossi Lev he explored support for debugging of transactional programs [21]. More recently, again with the team at Sun, he has explored the use of TM for memory management [11].

Perhaps most important, Maurice became a champion of the promise of transactions to simplify parallel programming—a promise he dubbed the “transactional manifesto” [19]. During a sabbatical at Microsoft Research in Cambridge, England, he collaborated with the Haskell team on their landmark exploration of *composability* [16]. Unlike locks, which require global reasoning to avoid or recover from deadlock, transactions can easily be combined to create larger atomic operations from smaller atomic pieces. While the benefits can certainly be oversold (and have been—though not by Maurice), composability represents a fundamental breakthrough in the creation of concurrent abstractions. Prudently employed, transactions can offer (most of) the performance of fine-grain locks with (most of) the convenience of coarse-grain locks.

5 Status and Challenges

Today hardware TM appears to have become a permanent addition to processor instruction sets. Run-time systems that use this hardware typically fall back to a global lock in the face of repeated conflict or overflow aborts. For the overflow case, hybrid systems that fall back to software TM may ultimately prove to be more appropriate. STM will also be required for TM programs on legacy hardware. The fastest STM implementations currently slow down critical sections (though not whole applications!) by factors of 3–5, and that number is unlikely to improve. With this present status as background, the future holds a host of open questions.

5.1 Usage Patterns

TM is not yet widely used. Most extant applications are actually written in Haskell, where the semantics are unusually rich but the implementation unusually slow. The most popular languages for research have been C and C++, but progress has been impeded, at least in part, by the lack of high quality benchmarks.

The biggest unknown remains the breadth of TM applicability. Transactions are clearly useful—from both a semantic and a performance perspective—for small operations on concurrent data structures. They are much less likely to be useful—at least from a performance perspective—for very large operations, which may overflow buffer limits in HTM, run slowly in STM, and experience high conflict rates in either case. No one is likely to write a web server that devotes a single large transaction to each incoming page request. Only experience will tell how large transactions can become and still run mostly in parallel.

When transactions *are* too big, and frequently conflict, programmers will need tools to help them identify the offending instructions and restructure their code for better performance. They will also need advances, in both theory and software engineering, to integrate transactions successfully into pre-existing lock-based applications.

5.2 Theory and Semantics

Beyond just atomicity, transactions need some form of condition synchronization, for operations that must wait for preconditions [16, 37]. There also appear to be cases in which a transaction needs some sort of "escape action" [29], to generate effects (or perhaps to observe outside state) in a way that is not fully isolated from action in other threads. In some cases, the application-level logic of a transaction may decide it needs to abort. If the transaction does not restart, but switches to some other code path, then information (the fact of the abort, at least) has "leaked" from code that "did not happen" [16]. Orthogonally, if large transactions prove useful in some applications, it may be desirable to parallelize them internally, and let the sub-threads share speculative state [4]. All these possibilities will require formalization.

A more fundamental question concerns the basic model of synchronization. While it is possible to define the behavior of transactions in terms of locks [27], with an explicit notion of abort and rollback, such an approach seems contrary to the claim that transactions are simpler than locks. An alternative is to make atomicity itself the fundamental concept [8], at which point the question arises: are aborts a part of the language-level semantics? It's appealing to leave them out, at least in the absence of a program-level abort operation, but it's not clear how such an approach would interact with operational semantics or with the definition of a data race.

For run-time–level semantics, it has been conventional to require that every transaction—even one that aborts—see a single, consistent memory state [15]. This requirement, unfortunately, is incompatible with implementations that "sandbox" transactions instead of continually checking for consistency, allowing doomed transactions to execute—at least for a little while—down logically impossible code paths. More flexible semantics might permit such "transactional zombies" while still ensuring forward progress [32].

5.3 Language and System Integration

For anyone building a TM language or system, the theory and semantic issues of the previous section are of course of central importance, but there are other issues as well. What should be the syntax of atomic blocks? Should there be atomic expressions? How should they interact with existing mechanisms like try blocks and exceptions? With locks?

What operations can be performed inside a transaction? Which of the standard library routines are on the list? If routines must be labeled as "transaction safe," does this become a "viral" annotation that propagates throughout a code base? How much of a large application must eschew transaction-unsafe operations?

In a similar vein, given the need to instrument loads and stores inside (but not outside) transactions, which subroutines must be "cloned"? How does the choice interact with separate compilation? How do we cope with the resulting "code bloat"?

Finally, what should be done about repeated aborts? Is fallback to a global lock acceptable, or do we need a hybrid HTM/STM system? Does the implementation need to adapt to observed abort patterns, avoiding fruitless speculation? What factors should influence adaptation? Should it be static or dynamic? Does it need to incorporate feedback from prior executions? How does it interact with scheduling?

5.4 Building and Using TM Hardware

With the spread of TM hardware, it will be increasingly important to use that hardware well. In addition to tuning and adapting, we may wish to restructure transactions that frequently overflow buffers. We might, for example—by hand or automatically—reduce a transaction's memory footprint by converting a read-only preamble into explicit (nontransactional) speculation [2, 39]. One of my students has recently suggested using advisory locks (acquired using nontransactional loads and stores) to serialize only the portions of transactions that actually conflict [38].

Much will depend on the evolution of hardware TM capabilities. Nontransactional (but immediate) loads and stores are currently available only on IBM POWER machines, and there at heavy cost. Lightweight implementations would enable not only partial serialization but also ordered transactions (i.e., speculative parallelization of ordered iteration) and more effective hardware/software hybrids [9, 26]. As noted above, there have been suggestions for "responder-wins" coherence, virtualization, nesting, and condition synchronization. With richer semantics, it may also be desirable to "deconstruct" the hardware interface, so that features are available individually, and can be used for additional purposes [23, 34].

6 Concluding Thoughts

While the discussion above spans much of the history of transactional memory, and mentions many open questions, the coverage has of necessity been spotty, and the choice of citations idiosyncratic. Many, many important topics and papers have been left out. For a much more comprehensive overview of the field, interested readers should consult the book-length treatise of Harris, Larus, and Rajwar [17]. A briefer overview can be found in chapter 9 of my synchronization monograph [31].

My sincere thanks to Hagit Attiya, Shlomi Dolev, Rachid Guerraoui, and Nir Shavit for organizing the celebration of Maurice's 60th birthday, and for giving me the opportunity to participate. My thanks, as well, to Panagiota Fatourou and Jennifer Welch for arranging the subsequent write-ups for BEATCS and SIGACT News. Most of all, my thanks and admiration to Maurice Herlihy for his seminal contributions, not only to transactional memory, but to nonblocking algorithms, topological analysis, and so many other aspects of parallel and distributed computing.

References

[1] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich, editors. Draft Specification of Transaction Language Constructs for C++. Version 1.1, IBM, Intel, and Sun Microsystems, Feb. 2012.

[2] Y. Afek, H. Avni, and N. Shavit. Towards Consistency Oblivious Programming. In *Proc. of the 15th Intl. Conf. on Principles of Distributed Systems*, pages 65-79. Toulouse, France, Dec. 2011.

[3] Y. Afek, D. Dauber, and D. Touitou. Wait-Free Made Fast. In *Proc. of the 27th ACM Symp. on Theory of Computing*, 1995.

[4] K. Agrawal, J. Fineman, and J. Sukha. Nested Parallelism in Transactional Memory. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[5] G. Barnes. A Method for Implementing Lock-Free Shared Data Structures. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, Velen, Germany, June–July 1993.

[6] H. W. Cain, B. Frey, D. Williams, M. M. Michael, C. May, and H. Le. Robust Architectural Support for Transactional Memory in the Power Architecture. In *Proc. of the 40th Intl. Symp. on Computer Architecture*, Tel Aviv, Israel, June 2013.

[7] C. Click Jr. And now some Hardware Transactional Memory comments. Author's Blog, Azul Systems, Feb. 2009. `blogs.azulsystems.com/cliff/2009/02/and-now-some-hardware-transactional-memory-comments.html`.

[8] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the Foundation of a Memory Consistency Model. In *Proc. of the 24th Intl. Symp. on Distributed Computing*, Cambridge, MA, Sept. 2010. Earlier but expanded version available as TR 959, Dept. of Computer Science, Univ. of Rochester, July 2010.

[9] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proc. of the 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, Mar. 2011.

[10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proc. of the 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, Mar. 2009.

[11] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. On The Power of Hardware Transactional Memory to Simplify Memory Management. In *Proc. of the 30th ACM Symp. on Principles of Distributed Computing*, San Jose, CA, June 2011.

[12] C. Ferri, A. Viescas, T. Moreshet, I. Bahar, and M. Herlihy. Energy Implications of Transactional Memory for Embedded Architectures. In *Wkshp. on Exploiting Parallelism with Transactional Memory and Other Hardware Assisted Methods (EPHAM)*, Boston, MA, Apr. 2008. In conjunction with *CGO*.

[13] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[14] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, Aug. 2005.

[15] R. Guerraoui and M. Kapałka. On the Correctness of Transactional Memory. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[16] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. of the 10th ACM Symp. on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[17] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory*, Synthesis Lectures on Computer Architecture. Morgan & Claypool, second edition, 2010.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, Boston, MA, July 2003.

[19] M. Herlihy. The Transactional Manifesto: Software Engineering and Non-blocking Synchronization. In *Invited keynote address, SIGPLAN 2005 Conf. on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[20] M. Herlihy and E. Koskinen. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[21] M. Herlihy and Y. Lev. tm_db: A Generic Debugging Library for Transactional Programs. In *Proc. of the 18th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Raleigh, NC, Sept. 2009.

[22] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, Dec. 1992.

[23] M. D. Hill, D. Hower, K. E. Moore, M. M. Swift, H. Volos, and D. A. Wood. A Case for Deconstructing Hardware Transactional Memory Systems. Technical Report 1594, Dept. of Computer Sciences, Univ. of Wisconsin–Madison, June 2007.

[24] A. Israeli and L. Rappoport. Disjoint-Access Parallel Implementations of Strong Shared Memory Primitives. In *Proc. of the 13th ACM Symp. on Principles of Distributed Computing*, Los Angeles, CA, Aug. 1994.

[25] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System z. In *Proc. of the 45th Intl. Symp. on Microarchitecture*, Vancouver, BC, Canada, Dec. 2012.

[26] A. Matveev and N. Shavit. Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory. In *Proc. of the 25th ACM Symp. on Parallelism in Algorithms and Architectures*, Montreal, PQ, Canada, July 2013.

[27] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[28] M. Moir. Transparent Support for Wait-Free Transactions. In *Proc. of the 11th Intl. Wkshp. on Distributed Algorithms*, 1997.

[29] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open Nesting in Software Transactional Memory. In *Proc. of the 12th ACM Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[30] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proc. of the 32nd Intl. Symp. on Computer Architecture*, Madison, WI, June 2005.

[31] M. L. Scott. *Shared-Memory Synchronization.* Morgan & Claypool, 2013.

[32] M. L. Scott. Transactional Semantics with Zombies. In *Invited keynote address, 6th Wkshp. on the Theory of Transactional Memory*, Paris, France, July 2014.

[33] N. Shavit and D. Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99-116, Feb. 1997. Originally presented at the *14th ACM Symp. on Principles of Distributed Computing*, Aug. 1995.

[34] A. Shriraman, S. Dwarkadas, and M. L. Scott. Implementation Tradeoffs in the Design of Flexible Transactional Memory Support. *Journal of Parallel and Distributed Computing*, 70(10):1068-1084, Oct. 2010.

[35] J. Turek, D. Shasha, and S. Prakash. Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proc. of the 11th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, Vancouver, BC, Canada, Aug. 1992.

[36] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.

[37] C. Wang, Y. Liu, and M. Spear. Transaction-Friendly Condition Variables. In *Proc. of the 26th ACM Symp. on Parallelism in Algorithms and Architectures*, Prague, Czech Republic, June 2014.

[38] L. Xiang and M. L. Scott. Conflict Reduction in Hardware Transactions Using Advisory Locks. In *Proc. of the 27th ACM Symp. on Parallelism in Algorithms and Architectures*, Portland, OR, June 2015.

[39] L. Xiang and M. L. Scott. Software Partitioning of Hardware Transactions. In *Proc. of the 20th PPoPP*, San Francisco, CA, Feb. 2015.

[40] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization. In x. f. H.-P. Computing, editor, *Proc., SC2013: High Performance Computing, Networking, Storage and Analysis*, pages 1-11. Denver, Colorado, Nov. 2013.