ISF Grant

October 24, 2017

0.1 Correctness properties

Correctness property specifies the requirements for an implementation to be considered correct. Herlihy and Wings *linearizability* property [6] is widely adopted for shared objects in conventional shared memory models. Roughly speaking, linearizability requires that operations on objects appear to take effect instantaneously in some serial order, while preserving the happens before relation, i.e., if op_1 ends before op_2 begins, then op_1 should precede op_2 in the serial order. Linearizability has no notion of an aborted or failed operation, and requires that a process finish one operation before it invokes the next. The criteria according to which implementations that support crash-recovery failures are correct remains unclear.

Aguilera and Frølund have proposed *strict linearizability* as a safety condition for persistent concurrent objects, which treats the crash of a process as a response, either successful or unsuccessful, to the interrupted operation [1]. A successful response means that the operation takes effect at some point between its invocation and the crash failure, and an unsuccessful response means that the operation does not take effect at all. Strict linearizability preserves both locality and program order. However, they proved that it precludes the implementation of some wait-free objects for certain machine models - there is no wait-free implementations of multi-reader single-writer (MRSW) registers from single-reader single-writer (SRSW) registers under strict linearizability.

Guerraoui and Levy have proposed *persistent atomicity* [5]. It is similar to strict linearizability, but allows an operation interrupted by a failure to take effect before the subsequent invocation of the same process, possibly after the failure. They also proposed *transient atomicity*, which relaxes this criterion even further and allows an interrupted operation to take effect before the subsequent write response of the same process. Both conditions ensure that the state of an object will be consistent in the wake of a crash, but it does not provide locality: correct histories of separate objects, when merged, will not necessarily yield a correct composite history.

Berryhill et al. have proposed an alternative, *recoverable linearizability* [2], which achieves locality but may sacrifice program order after a crash. It is a relaxed version of persistent atomicity, which requires the operation to be linearizes or aborts before any subsequent linearization by the pending thread

on that same object.

Izraelevitz et al. considered a real-world failure model, in which processes are assumed to fail together, as part of a full-system crash [7]. Under this model, persistent atomicity and recoverable linearizability are indistinguishable (and thus local). The term *durable linearizability* was used to refer to this merged safety condition under the restricted failure model.

0.1.1 Recoverable response

All the correctness properties discussed above focused on keeping the object or the data structure consist in case of a failures. However, none of which address the problem of the returned value by the failed operation. Consider for example a write operation. Since write is atomic it obviously satisfies strict linearizability - either the write took effect before the crash failure, or it did not. However, once the process is recovered it has no knowledge which of the two is the case, since the acknowledgement returned by the write is saved to the local memory of the process (a part of the cache), which is lost due to the failure. Assuming the process is in a middle of executing an operation which it desired to complete once it is recovered, it must know whether the write took effect (in which case it can continue and execute the code), or that it did not (in which case it needs to rewrite). Skipping the write is not an option, as it might cause an undesired behaviour, which in turn might violate the requirements of the object.

For this case, we present a different approach called *recoverable response linearizability* (RR-linearizability). In case of a failure, once the process is recovered it is "aware" of the failure, in the sense that it runs a designated recovery code. In a sense, this recovery code "extends" the interval of the interrupted operation. We require that the pending operation linearizes between it's invocation and the end of the recovery code, or aborts. In addition, in case of linearization, the returned value of the operation is known to the process. Notice that this definition introduce a relaxation which is not intuitive neither trivial - in case that a process operation took affect before the crash, but the response of it is lost due to the cache's lost, we allow the process, upon recovering, to extend the interval of that operation using the recovery code, such that at the end of it the process knows the returned value or aborts. This gives the process the flexibility to move the linearization point to after the crash, if needed. In addition, although the definition allows a process to abort the interrupted operation, it is equivalent to a definition in which a process must complete the operation - once a process knows the operation was aborted, it can reissue it.

0.2 Model

RR-linearizability does not specify the progress property required from the implementation. One can use a recoverable mutual exclusion lock [3, 4] at the recovery code in order to sequentially let the processes complete their pending operations. Obviously, this solution is not desired when the operation implementation we are recovering is wait-free. In this case, we would like the recovery

code to be wait-free as well, meaning that a process finish it in a finite number of its own steps in the absence of failure. Another point that requires our attention is the possibility of a failure along the recovery code. Will it be realistic to assume that such a scenario will not happen? Of course, a recovery code that can suffer itself failures is better. However, this is not well defined. If we allow each failure to run new and different recovery code, we can use a nesting technique in a way which increase the running time with each failure. This may also requires infinite memory and code. A better approach will be such that in the case of failure inside of a recovery code, restarting the recovery code from some point is enough. In such case, if the recovery code is wait-free in the absence of a failure, we can guarantee that no matter how many failures the process experienced, once given enough time with no failure, it will finish the recovery code (and therefore the pending operation) in a wait-free manner.

The standard abstract model is consider in this work. The system consist of a set of asynchronous unreliable processes $P = \{p_1, \ldots, p_n\}$ that communicate by applying operations on shared *base objects* that support atomic operations such as reads, writes, and read-modify-write primitives. All base objects are non-volatile, i.e, in case of a crash the values stored in the base objects are preserved. In order to implement a more complex object, such as stack and counter, each process is supplied with access procedures that simulate each operation on the implemented object using operations on base objects.

The failure model considered in this paper is individual. Each process may crash, recover, and then continue executing, at any point of the execution. Once a process crash, we assume that the system control knows at which line of the code the failure occurs, or more precisely, at the course of which atomic operation the process failed. However, there is no induction whether the atomic operation took affect or not before the crash. Each process is also equipped with a recovery code for any possible failure, that is, for any point of the execution it specifies the recovery code in case of a crash at that point. Upon recovering, the system control moves the process to the beginning of the appropriate recovery code. In this sense, by running the recovery code, the process is aware of the crash. At the end of the recovery code, the process goes back to the main code, to the location specified by the recovery code. The above model can be implemented by storing the program counter at the non-volatile memory. In this case, after each operation on a base object, there should be an operation updating the value of the program counter. This can be done by the system control, and the programmer need not be aware of it. Despite doubling the number of accesses to the main memory, this approach reduce the complexity of recovering in case of a crash.

Real world system presents more complications to the abstract model described above. In most modern system, each process has its own local volatile cache, which is used to store variables accessed by the process. Moreover, in order to improve performance in such systems, writes do not go directly to the shared memory, but rather stored in a buffer at the process's cache, while the system commit the writes later on according to the protocol and model it satisfies. Real system thus introduce reordering of instructions, as well as the

possibility of losing large amount of data in case of a failure, which makes the recovery procedure harder and more complicated. However, the abstract model can be simulated in many of those architectures, by using special instructions as memory fences and by avoid using the cache. The cost of such simulation, nonetheless, might be very expensive in terms of running time. The results presented in this paper implies that even in the simple abstract model the recovery procedure is not trivial. For future work, more complicated and realistic models needs to be studied.

Recoverable primitives

The correctness properties variants discussed in section 0.1 does not need to consider primitives, as any atomic operation satisfies any of the properties. Unlike it, RR-linearizability is not satisfied by primitives since the respond of the primitive is lost in case of a failure. Therefore, there is a need to supply each primitive with an implementation as well as a recovery code. A system which supply recoverable primitives can be used to implement any object in a recoverable way - in case of a crash the process will simply recover the last atomic operation it executed before the crash. In case the operation is aborted, the process will reissue it. After that, it can continue and execute the remaining code safely. This observation focus our attention to implementation of primitives in a recoverable manner.

In the following section we present RR-linearizable implementations for well known primitives, as well as presenting impossibility result for others. We focus our attention on bounded wait-free implementations, that is, the number of steps a process takes when executing the recovery code in the absence of a failure is finite and bounded by a known constant (may be a function of n, the number of processes in the system), regardless of the other processes steps and the failures the process experienced so far. In addition, we would like the recovery code to use a finite number of variables. The fact that RR-linearizabilty allows us to swift the linearization point of an operation to after the crash is used to recover after a primitive failure.

Read The process simply abort the read upon its recovering. Since read does not affect the rest of the processes, this can be done safely.

Write Write instruction is "wrap" with code such that in case of a failure the extra data will be used for recovering. For an instruction writing value x to variable R by process p, we provide the following implementation. We use the convention of capital letters names for shared memory variables, and small letters for local ones. In the following code, R_p is a variable in the memory designated for process p.

For simplicity, we write the recovery code as a single instruction, although it needs to be written as several instructions, as it access two different locations in the memory. Since R_p is accessed by p only, the point where p reads R determine the outcome of the recovery code. Moreover, since the recovery code

Algorithm 1 Write

```
1: procedure WRITE OPERATION
2:     res ← R
3:     R_p ← res
4:     R ← x
5: procedure RECOVERY CODE
6:     if R_p == R then return abort
```

contains only reads, and we already know how to recover reads, the recovery code itself is recoverable.

The intuition for the correctness of the code is that if there was a write to R between the two reads of p (at line 2, and at the recovery code), then either this write is by p, and we can linearize it at the point where it took affect, or that there was a write by some other process, and we can linearize the write of p just before it, such that we overwrite it, and the rest of the processes can not distinguish between the two situations. Therefore, in case of a failure before line 4, p will simply abort upon recovering, and in case of a failure at line 4, p will execute the recovery code.

The above analysis ignores the ABA problem - it might be that p reads the value from R, even though there was a write to R in between the two different reads. To overcome this problem, we can augment any value written with the writing process's id, and a sequential number (each process will have its own sequential number). This way, reading the same value guarantee that no write to R took place between the two different reads.

Compare-and-Swap Compare-and-Swap (CAS)...

Algorithm 2 Compare-and-Swap

```
1: procedure CAS OPERATION
2:     < id, val > ← C
3:     if val ≠ old then
4:         return false
5:     S[i] ← < id, val >
6:     res ← C.cas(< id, val >, < i, new >)
7:     if res = true then
8:         R[id][i] ← val
9:     S[i] ← null
10: procedure RECOVERY CODE
11:     Read C, S[], R[i][]
12:     if < id, val > appears in C or S[], or val appears in R[i][] then
13:         return true
14:     else
15:         return false
```

Test-and-Set Test-and-Set (TAS) is a known object supported as a primitive in many modern architectures. A TAS object contains a binary value initialized to 0, and support a single atomic operation TAS which set the value of the object to 1 and returns the old value.

We now show that there is no bounded wait-free RR-linearizable implementation for TAS even for two processes, assuming the system support only read, write and TAS primitives. We give a high level description for the proof. The construction also implies that any wait-free implementation need to use an unbounded number of TAS objects.

Assume by contradiction that there is such an implementation. Denote the set of processes by $P = \{p, q\}$. First, notice that the code implementing the TAS operation must contain a TAS object O_1 which accessed by the two processes and determine their output, otherwise we can solve consensus for two processes using reads and writes only, contradicting to FLP result [*]. We let both processes run to the point where they both about to access O_1, and then perform a TAS one after the other, followed by a crash of p only.

Once p is recover, it needs to execute a recovery code, since the value returned from O_1 is lost. We look at a solo run of p starting from this point, and consider the possible return values. If p returns 1, then we look at the execution where q was the first to access O_1 before the crash. If p returns 0 or abort, we look at the execution where q is the second to access O_1. Notice that after the crash, p does not able to distinguish between the different executions, thus it performs the same run after any of these. TAS can be seen as a consensus for two processes, the one that returns 1 wins. In any case we get a bivalet configuration - both processes want to return the same value (we consider abort as 0), while only one can return 1, while the other must return 0. Thus, it must be that both processes access another TAS object O_2. We can continue and use the same argument, constructing an execution where more and more different TAS objects are accessed.

References

[1] Marcos Kawazoe Aguilera and Svend Frølund. Strict linearizability and the power of aborting. In *Tech. Rep. HPL-2003-241*, 2003.

[2] Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, pages 20:1–20:17, 2015.

[3] Wojciech M. Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 211–220, 2017.

[4] Wojciech M. Golab and Aditya Ramaraju. Recoverable mutual exclusion: [extended abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 65–74, 2016.

[5] Rachid Guerraoui and Ron R. Levy. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems (ICDCS 2004), 24-26 March 2004, Hachioji, Tokyo, Japan*, pages 400–407, 2004.

[6] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[7] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, pages 313–327, 2016.