ISF Grant

September 27, 2017

Abstract

a

1 Introduction

Traditional computer architectures containing both a volatile main memory alongside a non-volatile secondary storage, have led to a separation of program state into operational data stored using in-memory data structures, and recovery data stored using sequential on-disk structures such as transaction logs. A system-wide failure, such as a power outage, cause a full lost of the in-memory data structures, which must be reconstructed entirely from the recovery data, making the software system temporarily unavailable. As a result, the design of in-memory data structures emphasise disposable constructs optimized for parallelism, in contrast to the structures that hold recovery data, which cannot be discarded upon a failure and which benefit less from parallelism since their performance is limited by the secondary storage.

Current industry projections indicate that non-volatile main memory (NVRAM) will become commonplace over the next few years. While the availability of NVRAM suggests the possibility of keeping persistent data in main memory (not just in the file system), the fact that recent updates to registers and cache may be lost during a power failure means that the data in main memory, if not carefully managed, may not be consistent at recovery time. Traditional log-based recovery techniques, although can be applied correctly in such systems, fail to take full advantage of the parallelism enabled by allowing processing cores to access recovery data directly using memory operations rather than slow block transfers from secondary storage. As a result, harnessing the performance benefits of NVRAM-based platforms requires a careful rethinking of recovery mechanisms.

The study of shared data structures and concurrent programming dates back over a half century ago. 1965 Dijkstra's pioneering work presented the concept of concurrent programming [*], followed by a series of seminal papers on inter-process communication, wait-free synchronization, and linearizability [*]. Recent research has paid close attention to asynchronous models, in which there is no bound on the amount of time a process takes to transition to its next step,

or to complete a memory operation. This assumption captures the behaviour of most modern memory hierarchies, where different media (e.g., L1 cache vs. L2 cache vs. main memory) incur vastly different and often unpredictable access latencies, as well as the effect of the operating system (e.g., via preemption and interrupts) on the liveness of processes. In addition, asynchrony is also related to reliability in a precise way: algorithms that provide non-blocking progress properties (e.g., lock-freedom and wait-freedom) in an asynchronous environment with reliable processes continue to provide the same progress properties if crash failures are introduced. Informally speaking, this property holds because a process that crashes permanently at an arbitrary point in the execution of its algorithm is indistinguishable to the other processes from one that is merely very slow.

1.1 Correctness properties

Correctness property specifies the requirements for an implementation to be considered correct. Herlihy and Wings linearizability property [*] 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 Frlund 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 [*]. 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 [*]. 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 [*], 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. 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.

1.1.1 Recoverable response

All the correctness properties discussed above focused on keeping the object or 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. As the operation is atomic, it obviously satisfies strict linearizability - either the write took effect before the crash failure, or that 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 (part of the cache), which is lost due to the failure. Assuming the process is in a middle of executing a bigger 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 rest of 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) (good term????). 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 is complete before the crash, but the response of it is lost due to the cache's lost, we allow the process, upon recovering, extending the interval of that operation using the recovery code (even though the operation was complete), 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 is aborted, it can reissue it.

RR-linearizability does not specify the progress property required from the implementation. One can always use a recoverable mutual exclusion lock at the recovery code in order to sequentially let the processes complete their pending operations. Obviously, this solution is not desired when the operation imple-

mentation 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. One more point that requires our attention is the possibility of a failure along the recovery code. Would 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 always use nesting technique. A better approach will be such that in the case of failure inside of a recovery code, restarting the recovery code from the beginning 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, if given enough time with no failure, it will finish the recovery code (and therefore the pending operation) in a wait-free manner.

References