ISF Grant

September 26, 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 emphasizes disposable constructs optimized for parallelism, in contrast to the structures that hold recovery data, which cannot be discarded upon a failure and which beneft 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 behavior 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 and Progress properties

Correctness property specifies the requirments for an implementation to be considerd correct. Linearizability, presented by Wang and Herliry [*], is the most common correctness property in use for shared memory objects. Roughtly 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 op1 ends before op2 begins, then op1 should precede op2 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 recovered processes are correct remain unclear.

References