NVRAM - response recover linearzability [*]

[Extended Abstract]

Hagit Attiya
Department of Computer-Science
Technion
Haifa, Israel
hagit@cs.technion.ac.il

Ohad Ben-Baruch
Department of Computer-Science
Ben-Gurion University of the Negev
Be'er Sheva, Israel
ohadben@post.bgu.ac.il

Danny Hendler
Department of Computer-Science
Ben-Gurion University of the Negev
Be'er Sheva, Israel
hendlerd@cs.bgu.ac.il

ABSTRACT

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.8 [**Analysis of Algorithms and Problem Complexity**]: Tradeoffs between Complexity Measures; F.1.3 [**Complexity Measures and Classes**]: Relations among complexity measures

General Terms

Theory, Performance

Keywords

Mutual exclusion, shared-memory, lower bounds, total store ordering, time complexity, remote memory reference (RMR)

1. INTRODUCTION

2. MODEL AND DEFINITIONS

3. RECOVERABLE BASE OBJECTS

The correctness properties variants discussed in section **??** 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 aborts the read upon recovering. Since read does not affect other processes, aborting the operation does not damage the linearizability of the program.

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.

Algorithm 1 Write
```
1: procedure WRITE OPERATION
2:     $res \leftarrow R$
3:     $R_p \leftarrow res$
4:     $R \leftarrow x$
5: procedure RECOVERY CODE
6:     if $R_p == R$ then return abort
```

For simplicity, we write the recovery code as a single instruction, although it needs to be written as several instructions, as it accesses two different locations in the shared memory. Since R_p is accessed by p only, the point where p reads R determines the recovery code outcome. Moreover, in case of a failure along the recovery code, the process can simply restart it, as it contains only reads.

[*]Partially supported by the Israel Science Foundation (grants 1227/10, 1749/14) and by the Lynne and William Frankel Center for Computing Science at Ben-Gurion University.

The intuition for correctness 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. Hence, the real write ”overwrite” p’s write, 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 same 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. A Compare-and-Swap (CAS) object supports a single operation which atomically compares the value of the shared variable with its first parameter, and if they are equal, sets the value of the variable to its second parameter. At a high level, a process p first reads the CAS variable. If it observes a value different then *old*, then it return false. In such case, the operation can be linearized at the time of the read. Otherwise, p announce the process which is value was stored in the CAS object, that it reads its value, by writing to a designated memory, and only then it can try and apply the CAS operation to the object. This way, if p fails after a successful CAS operation, a different process that wants to change the value of the CAS, first needs to announce p his CAS was successful. Therefore, upon recovery, p can identify if its CAS took affect by reading C, and looking for an info by different process that have seen p’s value.

Algorithm 2 Compare-and-Swap

```
1:  procedure CAS OPERATION
2:      $< id, val > \leftarrow C$
3:      if $val \neq old$ then
4:          return false
5:      $R[id][i] \leftarrow val$
6:      $res \leftarrow C.cas(< id, val >, < i, new >)$
7:  procedure RECOVERY CODE
8:      Read C, R[i][*]
9:      if $< id, val >$ appears in C, or $val$ appears in R[i][*] then
10:         return true
11:     else
12:         return false
```

4. DISCUSSION

5. REFERENCES