J. Parallel Distrib. Comput. 121 (2018) 1–14

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Nontrivial and universal helping for wait-free queues and stacks

Hagit Attiya [a], Armando Castañeda [b,*], Danny Hendler [c]

[a] Department of Computer Science, Technion., Israel
[b] Instituto de Matemáticas, UNAM., Mexico
[c] Department of Computer Science, Ben-Gurion University, Israel

HIGHLIGHTS

- Two formalizations of helping are presented: nontrivial and universal.
- Nontrivial helping separates queues and stacks from Test&Set primitives.
- A weaker version of nontrivial helping is introduced and shown necessary in queue implementations.
- Universal helping for queues and stacks is universal: it solves consensus.
- Strongly linearizable queue or stack for n processes requires primitives with consensus number at least n.

ARTICLE INFO

Article history:
Received 13 August 2016
Received in revised form 25 April 2018
Accepted 13 June 2018
Available online 4 July 2018

Keywords:
Wait-freedom
Non-blocking
Common2
Consensus number
Helping
Strong linearizability
Shared objects
Queues
Stacks

ABSTRACT

This paper studies two approaches to formalize helping in wait-free implementations of shared objects. The first approach is based on *operation valency*, and it allows us to make an important distinction between *trivial* and *nontrivial* helping. We show that any wait-free implementation of a queue from Test&Set requires nontrivial helping. We also define a weaker type of nontrivial helping and show that any wait-free queue implementation from a set of *arbitrary base objects* requires it. In contrast, there is a wait-free implementation of a stack from Test&Set with only trivial helping. These results shed light on the well-known open question of whether there exists a wait-free implementation of a queue in Common2, and indicate why it seems to be more difficult than implementing a stack.

The other approach formalizes the helping mechanism employed by Herlihy's universal wait-free construction and is based on having an operation by one process restrict the possible linearizations of operations by other processes. We show that queue and stack implementations possessing such *universal helping* can be used to solve consensus. This result can be used to show that a *strongly linearizable* (Golab et al., 2011) implementation of a queue or a stack for n processes must use objects that allow to solve consensus among n or more processes.

1. Introduction

A key component in the design of concurrent applications are *shared objects* providing high-level semantics for communication among processes. For example, a *shared queue* to which processes can concurrently enqueue and dequeue allows them to share tasks. Shared objects are implemented by using Read/Write atomic registers and *base objects* providing more powerful atomic operations, e.g., Test&Set, Fetch&Add or Compare&Swap. The implementation is required to be *linearizable* [17], roughly, ensuring that each (high-level) operation that completes appears to take effect atomically at some point between its invocation and response.

Generally speaking, an implementation of a shared object is *wait-free* if any operation on the shared object is guaranteed to terminate after a finite number of steps; the implementation is *nonblocking* if it only ensures that *some* operations complete. Clearly, a wait-free implementation is nonblocking but not necessarily vice versa.

The *consensus number* of a shared object is the maximum number of processes that can solve consensus using copies of the object, in addition to Read/Write registers [15]. Objects with an infinite consensus number, like Compare&Swap, are *universal*. This means that they can be used for obtaining a linearizable wait-free implementation of any shared object that has a sequential specification [15]. This is not the case for objects with finite consensus number, like Test&Set or stack, whose consensus number is 2, allowing to solve consensus exactly for two processes. Common2 is the class of objects that includes all objects that have a wait-free

* Correspondence to: Instituto de Matemáticas, Ciudad Universitaria, Ciudad de México, 04510, Mexico
E-mail address: armando.castaneda@im.unam.mx (A. Castañeda).

https://doi.org/10.1016/j.jpdc.2018.06.004
0743-7315/

n-process implementation, for any n, from Read/Write registers and 2-consensus objects [3]. Common2 contains some objects with consensus number 2, such as Test&Set, Fetch&Add, Swap [3] and stack [2]. An interesting case is a shared queue, whose consensus number is 2, but is not known to be in Common2. This is a well-known open question, first introduced in [3] and later studied in [2,5,9]. However, there is a simple nonblocking implementation of a shared queue in Common2, using Test&Set objects [19].

Many implementations of shared objects, especially the wait-free ones, include one process *helping* another process to make progress. The helping mechanism is often a piece of code that is added to a nonblocking implementation, in order for a process that is sure to complete its own operation to "help" other processes make progress so that they eventually terminate their operations too, making the resulting implementation wait-free.

This article investigates ways of formalizing the notion of helping, with the purpose of being able to separate objects for which there are wait-free implementations from those with only nonblocking implementations. We are interested in helping that uses the same base objects as the nonblocking implementation being considered, e.g., Fetch&Add or Test&Set, in addition to Read/Write registers, since nonblocking and wait-free implementations cannot be separated when helping uses universal objects, like Compare&Swap. An interesting object to study is a shared queue, which, as already mentioned, has a nonblocking implementation using Test&Set. A helping mechanism for a queue using Test&Set, if one exists, would imply the existence a wait-free queue implementation using only Test&Set, showing that queue is in Common2.

Nontrivial helping. We first introduce a notion of helping that is based on one process determining the return value of an operation by another process. This formalization relies on the notion of *operation valency* [14], i.e., the possible values an operation might return. Roughly speaking, there is helping in an implementation if in some situation, the step of a process makes an undecided operation of another process become decided on some value. In the context of specific objects, like queues and stacks, which have a distinguished "empty" value (denoted $\perp$), we say that helping is *nontrivial* if one process makes an operation of another process decided on a non-$\perp$ value. In nontrivial helping, the helping process somehow needs to acquire the value it gives to the helped process, so it cannot be returned by another process, in contrast to a $\perp$ value that can be returned by several processes.

Our first main result is that nontrivial helping is a distinguishing factor between queues and stacks implemented from Test&Set. We prove that any linearizable wait-free queue implementation from Test&Set must have nontrivial helping. In contrast, we show that the wait-free stack of Afek et al. [2] (which established that stacks are in Common2) does not have nontrivial helping. We stress that if queues are in Common2, then there is a wait-free implementation of a queue from 2-process Test&Set. This result is extended to show that any linearizable wait-free implementation of a queue must have a weaker version of nontrivial helping that we call *generalized nontrivial helping*, regardless of the base objects used in the implementation.

Universal helping. The article also studies an alternative way to formalize helping, which is based on restricting the possible linearizations of an operation by the progress of another process. This kind of helping, which we call *universal*, formalizes the helping mechanism employed in Herlihy's universal construction [15]. Intuitively, an implementation has universal helping if for every execution α, for every long enough extension of it, all pending operations in α (which might be still pending in the extension) are linearized, and that linearization order does not change due to any future steps.

We say that an implementation is *readable*, if every base object it uses provides a Read operation that returns its state. We show that universal helping in nonblocking implementations of readable queues and stacks is strong enough to solve consensus. Namely, a nonblocking n-process linearizable readable implementation of a queue or a stack with universal helping can be used to solve n-process consensus. We also prove that if the implementation is wait-free, then a *single instance* of the implementation is universal: any shared object with a sequential specification can be wait-free implemented using one instance of the implementation, and Read/Write registers.

Interestingly, our results concerning universal helping can be extended to strongly linearizable implementations of queues or stacks. Roughly speaking, an implementation is *strongly linearizable* [12] if once an operation is linearized, its linearization order cannot be changed in the future. We show that any readable strongly linearizable nonblocking implementation of a queue or stack can be used to solve consensus, and if the implementation is wait-free, even a single copy can be used to wait-free implement any shared object with a sequential specification. This result has two implications. The first one is that there is no separation between wait-freedom and nonblocking for strongly linearizable readable stack or queue implementations, since consensus is universal [15]. The second one is that any n-process strongly linearizable and nonblocking or wait-free implementation of a queue or a stack must be based on base objects whose *readable* versions have consensus number n or more. For example, there is no such an implementation from Test&Set objects, because readable Test&Set has consensus number 2.

Viewed collectively, our results provide insights on why a wait-free linearizable implementation for queues from objects with consensus number 2 has been elusive for a long time. Any such implementation must incorporate nontrivial helping, however, this mechanism cannot be too strong, otherwise the resulting implementation would be able to implement any shared object for $n \geq 3$ processes, which is impossible to do with objects of consensus number 2.

Related work. Helping has been used in several implementations and comes in different flavors. The helping mechanism in Herlihy's universal construction is a well-known example [15]. Another example is the wait-free linearizable snapshot implementation [1], in which processes take snapshots while executing update operations to help overlapping processes, which can use any of these snapshots in order to make progress. There are additional examples in the literature where helping plays an important role (e.g. [10,11,16,18,20]).

Recently, a formalization of helping based on the possible linearizations of a given execution was introduced [4]. Under this notion of helping, both queue and stack implementations must have helping, hence these shared objects cannot be distinguished from this perspective. Section 6.1 compares nontrivial helping and universal helping with the formalization of [4].

Common2 is the family of objects that are wait-free implementable from 2-consensus for any number of processes [3]. Initially, it was proven that Test&Set, Fetch&Add, Swap and other objects are in Common2 [3], and later it was shown that stacks are in Common2 as well [2]. The question of whether queues belong to Common2 has received a considerable amount of attention. Various restrictions of queues are in Common2 [5–7,9,19] but it is unknown if queues in general are in Common2. Our article studies this question, for the first time, from the perspective of helping.

It has been observed that linearizability presents some anomalies when used with randomized implementations [12]. One such anomaly is, roughly, that the expected running time or error probability of a linearizable randomized implementation cannot be analyzed under the typical assumption that any high-level