Functional Faults

Gali Sheffi
galish@cs.technion.ac.il
Department of Computer Science, Technion
Haifa, Israel

Erez Petrank
erez@cs.technion.ac.il
Department of Computer Science, Technion
Haifa, Israel

ABSTRACT

Hardware and software faults increasingly surface in today's computing environment and vast theoretical and practical research efforts are devoted to ameliorate the effects of malfunctionality in the computing process. Most research to date, however, has focused on how to discover and handle faulty data. In this paper we formalize and study faulty functionality in a modern multicore shared-memory environment. Functional faults have been previously studied in the architecture community. However, they have never been formally defined and lower/upper bounds were not previously proven. We present a model of functional faults, and study avenues that allow tolerating functional faults while maintaining the correctness of the entire computation. We exemplify this model by constructing a robust consensus protocol from functionally-faulty compare-and-swap objects. We then show a (tight) impossibility result for the same construction, when the number of faults exceeds a certain threshold. Interestingly, for some fault types, more *functional* faults can be tolerated than the analogue *data* faults, beating an impossibility result for data faults and demonstrating the difference between the two models.

CCS CONCEPTS

• **Mathematics of computing** → **Approximation**; • **Theory of computation** → **Concurrency**; **Parallel computing models**; **Hoare logic**; **Shared memory algorithms**; **Concurrent algorithms**; **Object oriented constructs**; **Pre- and post-conditions**.

KEYWORDS

concurrent algorithms, shared memory, lowerbounds, fault-tolerance, hardware faults, software faults

ACM Reference Format:
Gali Sheffi and Erez Petrank. 2020. Functional Faults. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3350755.3400261

1 INTRODUCTION

As computing environments strive to achieve higher concurrency, record performance, and low energy, hardware faults become more observable on high-end platforms, as well as on older worn-out machines. Making computation robust in the presence of hardware and software faults has attracted vast theoretical and applied research [6, 16, 17, 25, 33, 42, 45, 46]. Most existing studies focus on failing processes and memory faults. In the parallel and distributed computing domain, the literature offers numerous designs for robust parallel and distributed systems that can tolerate the failure of one or more executing processes. For example, in a shared memory environment, a crash of a process holding a lock can prevent all other processes from acquiring that lock and completing their tasks. One way to deal with this type of crash failure is the design of wait-free implementations [12, 26, 28, 39, 41, 44], guaranteeing that each process completes its execution, regardless of the behavior (and, in particular, failures) of other processes. Another solution example is to add a crash-recovery protocol [21–23], enabling processes to overcome their own crashes and complete their execution.

Memory data faults occur when data in the memory gets corrupted during computation, and further reading of its content is either impossible or yields incorrect results. This term sometimes relates to the occurrence of *silent data corruptions* [6, 17, 25], which are relatively small errors in the data saved in the memory. These faults are usually hard to detect and handle, because they do not lead to a system halt or crash. The most commonly used method for dealing with memory corruption is the application of error correcting codes [38]. Other approaches for dealing with data corruption for specific tasks were proposed in the seminal papers of Jayanti et al. [30] and Afek et al. [2]. They operate in the realm of multiprocessing and shared objects that synchronize the coordinated execution, and they propose clever algorithms that deliver algorithmic robustness to the faulty underlying objects. In these papers, object faults consist of corrupted *memory* that represents general data or specific corrupted operation results. These papers also provide impossibility results: when too many underlying objects are faulty, a correct overall computation becomes provably impossible.

Applied research highlights functional errors as different from memory corruptions. Computation errors may arise for various reasons. For example, the voltage of a processor is sometimes scaled-down in order to save resources [46]. Along with downgrading the voltage, computations are sometimes halted, which may cause them to output intermediate calculation results. In other energy-aware methods [16, 42], computations are initially designed to output imprecise results, while utilizing less resources. Another example is the occurrence of *soft errors* [45], which are temporary circuit faults that arise due to a variety of both internal and external noise sources. Interestingly, while memory corruption models have been formally treated, there is no formal representation of such computation errors, as a class of faults.

In the present work, we formalize *functional faults*. A functional fault happens when a function is computed and its returned output is incorrect. For example, consider a function execution in

SPAA '20, July 15–17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-6935-0/20/07…$15.00
https://doi.org/10.1145/3350755.3400261

which some arithmetic evaluation is slightly off (e.g., a carry evaluation is wrong for an addition operation), or a logical condition is erroneously computed. Previous work regarded such faults in a coarse-grained manner: either the function yields the correct result, or it is faulty, and then the result is arbitrarily corrupted or unavailable. We propose a finer-grained consideration of functions, allowing us to argue about how they fail. When a function result is incorrect, it is not necessarily arbitrary. Sometimes errors can manifest themselves in a more structured manner. More formally, if the preconditions of a function are satisfied on entry to its computation, the result of the function computation is incorrect if it does not satisfy its postconditions. A faulty computation of a function can be erroneous and still satisfy differently structured postconditions. If the functional fault is arbitrary, adhering to no postcondition, then we simply return to the *memory data faults* model, because arbitrary functional faults are equivalent to an arbitrary corruption of the data that the function returns. In contrast, when a faulty function is known to produce a structured error, then robustness may be obtained using more sophisticated techniques.

The way we overcome functional faults is by composing several objects, some of which may be faulty, in a well-designed algorithmic manner, in order to achieve a reliable compound object that can tolerate faults. This is a standard method, also used to handle other types of faults. Examples of such constructions in the memory data fault model are presented both in [2] and [30], along with some impossibility results. Given an upper bound f on the total number of faulty base objects, the question is how many base objects are needed overall in order to construct a reliable compound object?

A Case Study: Building Consensus from CAS Objects with Potential Overriding Faults

The consensus problem is important and widespread in both theory and practice. In a consensus protocol, each process starts with an input value, and, at the end of the execution, each (correct) process terminates with an agreed-upon output stipulating that: (1) all correct processes agree on the same output value, (2) the output value is the input value of one of the participating processes, and (3) all processes eventually terminate. Consensus has been widely used in practice, e.g., for blockchain and reliable distributed storage [8, 11, 15]. From a theoretical standpoint, consensus has been shown by Herlihy [26] to be universal, in the sense that it can be used to implement any wait-free object. Herlihy also presented the consensus hierarchy that associates consensus numbers with wait-free objects. The consensus number is the maximum number of processes that can participate in a consensus protocol, using this object.

Because of its universality, the consensus problem has often been the focus of study for the universal construction of reliable objects in the presence of process failures in the standard models of shared memory and message passing, (e.g., [10, 32]) and also in the presence of memory faults [2, 30]. Complementary studies investigated impossibilities in the presence of failures. Much effort has been devoted to showing that consensus, as well as other tasks, depending on the number of faulty processes in the execution, cannot be achieved in these scenarios [18, 19].

A compare-and-swap (CAS) operation receives a register R, an expected value E, and a new value N, and if E equals the content of register R, it atomically replaces this content with N. It does nothing otherwise. We use the CAS object, which is an object that allows executing CAS operations (only), and consider a specific functional fault, denoted the *overriding* fault, as a case study.

When no faults occur, it is known that consensus can be obtained from a single CAS object [26]. We present constructions of a reliable consensus object from multiple such (possibly functionally-faulty) CAS objects. The consensus number of a (non-faulty) CAS object is ∞. This is not, however, the consensus number of a CAS object with an overriding fault. We demonstrate the richness of the fault levels: for every $n > 1$, there is a faulty CAS with a consensus number n.

The following results are shown for building consensus using faulty overriding CAS objects. For a system with two processes, there is a simple reliable construction that uses a single (possibly faulty) CAS object. Next, we look at consensus with more than two processes. If the number of faults per object is *unbounded* and with at most f CAS objects being faulty during the computation, one can implement a reliable consensus object using $f + 1$ CAS objects. Moreover, in this case, we show that consensus is not achievable with f (or fewer) objects. Otherwise, with a *bounded* number of faults per faulty object, and at most f faulty objects, one can implement a consensus protocol using f CAS objects (all may be faulty) when the number of processes is at most $f + 1$. For more than $f + 1$ processes, however, at least $f + 1$ objects are needed (as in the unbounded case).

Contribution and Organization

This paper makes the following contributions:

- We present and formalize functional faults as a means to study structured errors in function computations.
- We present an example of an important function (the CAS operation), a natural fault (the overriding fault), and a construction of an important protocol (consensus) out of faulty CAS objects.
- We prove that our constructions are optimal by showing that using fewer objects would foil consensus.
- We show that functional faults are more expressive than data faults, by beating the data faults lower bound for the overriding CAS.
- We show that different levels of faults yield the entire Herlihy consensus hierarchy.

Organization. This paper is organized as follows. In Section 2, we describe the (standard) shared-memory model we use and a formal definition of the consensus problem. The functional fault model is formally presented in Section 3, along with a review of the prior data fault model, a formal definition of the CAS overriding fault and some additional fault types. Specific constructions of a reliable consensus object from faulty CAS objects are presented in Section 4, and impossibility results appear in Section 5. Additional related work is discussed in Section 6, and we conclude in Section 7.

2 PRELIMINARIES

We consider a standard shared memory setting with a fixed collection of processes, communicating via a finite set of shared objects [36]. Each object has a certain type, and can be accessed via operations. An object's *type* includes a (finite or infinite) set of possible states (with a distinguished initial state) and a sequential specification. The *sequential specification* defines the expected behavior of the object in a sequential run. It specifies, per operation, the preconditions and their respective postconditions. It also takes into account the operation's input parameters. A type is considered deterministic when, given an operation and its current state (assuming it obeys the operation's preconditions), there exists a single resulting state that satisfies the operation's postconditions. An object's type only defines its expected behavior, and thus the object's internal implementation is not restricted.

An execution *state* is an instantaneous snapshot of the system, describing the content of the shared objects and the local state of each process (its program counter, invocation stack and the content of its local variables). The system's *initial state* is defined, given the set of its processes, their local variables and their programs, and the set of shared objects. In the initial state, the state of each shared object and local variable is its initial state, and each process's program counter is set to the beginning of its program. The processes change the system's state by taking *steps*, which may include performing some local computations and, additionally, an operation on one of the shared objects. In particular, an invocation of an operation (together with the respective input parameters) and the return from an operation are each considered to be a single step. We assume each step is atomic. An *execution* consists of an alternating sequence of states and steps, starting with the initial state.

We say that an object is *wait-free* [26] if any of its operations, executed by a system process, is guaranteed to terminate after a finite number of steps by this process (regardless of the behavior of the rest of the running processes).

The CAS primitive. The CAS primitive receives three input parameters: a target register, an expected value and a new value. It then atomically compares the register's content to the received expected value, and if they are equal, it replaces the original register value with the new value. Otherwise, the content of the register does not change. A CAS execution is considered successful when the new value is written to the target register, and unsuccessful otherwise. Regardless of the operation's success, it returns the original content of the register.

Consensus. In a *consensus protocol*, there is a fixed set of processes, communicating via the system's shared objects. Each process begins with an input value and decides on an output value at the end. The protocol has to satisfy the following requirements: (1) Validity: the decided-upon value must be the input of some process, (2) Consistency: all processes must decide on the same value, and (3) Wait-freedom: each process finishes the protocol after a finite number of steps, regardless of the behavior of the rest of the processes.

A *consensus object* is a shared object that provides a single operation, receiving the input value and returning the decided-upon value, while satisfying the above postconditions. Such an object can be implemented using a single CAS object [26] in the following way: The object is initialized to $\perp$, which is different from the input values of all of the processes. Then, each process tries to write its input to the object, using the CAS operation, assuming its current value is $\perp$. Since exactly one process succeeds, choosing its input value satisfies consistency and validity. Finally, executing the CAS operation is wait-free.

The *consensus number* of an object O (first presented in [26]) is the maximal number of processes for which the object O can implement a consensus protocol. For example, the consensus number of CAS is ∞, so the above implementation is suitable for any number of executing processes.

3 FAULT MODEL

Previous theoretic work on dealing with hardware faults focused on data faults. An overview of the data fault model appears in Section 3.1. We present our model of functional faults in Section 3.2. In Section 3.3, we present the overriding CAS as a functional fault example, which we use to demonstrate overcoming functional faults. Finally, in Section 3.4, we present some other possible functional faults of the CAS operation, and review ways to deal with them.

3.1 Data Faults

A memory data fault is an unexpected modification of the content of a shared memory address, or an event where the content of a memory address becomes unreadable (i.e., a read from this address returns an error mark, or does not return at all). A data fault occurs regardless of the behavior of the executing processes, and independently of their writes to the shared memory. In this model, a shared object is considered faulty if during the execution, its content is replaced at least once by a data fault. A faulty object may suffer from a finite or infinite number of data faults.

Jayanti et al. [30] divided object faults into two classes: *responsive* and *nonresponsive*. An object with a responsive fault continues to respond to every operation, but its output does not necessarily adhere to the object's specifications. In contrast, an object with a nonresponsive fault is not guaranteed to respond at all. Both fault classes are further divided into three sub-classes, called *crash*, *omission* and *arbitrary*.

Afek et al. [2] considered a scenario where the memory gets corrupted occasionally, either a finite or infinite number of times, during execution. An object is faulty in this model if it suffers at least one memory fault during the execution. This definition implies that all faults are responsive. Afek et al. used the term *fault operation* for occasional faults. Nevertheless, they considered a data fault in memory and not a fault of an operation computation as we propose in this paper.

3.2 Functional Faults

We now present the notion of *functional faults* that deals with structured faults of function execution. Intuitively, functional faults are faults that occur during the execution of an object operation. The definition covers any structured deviation from the correct execution, but we are most interested in cases where functional faults occur due to minor problems such as a single gate miscalculation, or

a single wrong branch (perhaps one-sided as the overriding CAS), etc. While data faults occur at any time during the computation, a functional fault occurs only during the execution of an operation, and while data faults can produce arbitrary values (or no access to values), a functional fault deviates from the correct result in a limited, specified manner. Therefore, functional faults can sometimes be addressed by an algorithm that handles the specific structured deviation.

Following [27], the set of correctness conditions for an operation O can be expressed using a triplet $\Psi\{O\}\Phi$. Both Ψ and Φ are assertions (conjunctions of formulas), Ψ stands for O's preconditions and Φ for its postconditions. When the preconditions Ψ are satisfied on an invocation of O and O is correct, then the postconditions Φ must be satisfied on return from O. A functional fault occurs during the execution of O when Ψ is satisfied before executing O, but Φ is not satisfied at the end of the O's execution. In practice, not satisfying the postconditions could manifest itself by returning an incorrect output or by writing an incorrect value to the shared memory. We can characterize a functional fault by providing deviating postconditions Φ' that a faulty result satisfies. The postconditions Φ' are different from Φ and usually allow more behaviors. Definition 1 states the above formally.

Definition 1. *Let $\Psi\{O\}\Phi$ be an operation O, its preconditions Ψ and postconditions Φ. We say that an $\langle O, \Phi'\rangle$-fault occurred in Step i of an execution E if Step i represents the response from an invocation of O, s_0 is the state that precedes the invocation step, s_1 is the state that follows i, $s_0 \models \Psi$, $s_1 \not\models \Phi$, and $s_1 \models \Phi'$.*

Functional faults $\langle O, \Phi'\rangle$ are more constrained than data faults. First, data faults manifest arbitrary corruption of memory locations or computation results, whereas functional faults must adhere to the specific postconditions Φ'. Second, functional faults occur only as a result of an operation invocation (and not at arbitrary points during the execution). Restricting the possible errors raises several interesting questions. Can we achieve better fault tolerance (with fewer resources) using clever algorithms? Which deviations from the postconditions of a function can be overcome by compound constructions? Which deviations are impossible to handle?

In this work we show that resources can be saved, given a bound f on the number of faulty objects in the execution. We provide an example where the total number of objects required in order to compose a reliable compound object in the functional faults model is smaller than the number of objects required in the data fault model. Particularly, in some scenarios, one can implement a reliable computation in the functional faults model from faulty-only base objects.

From an object perspective, a shared object is faulty in an execution E, when one of its operations is faulty in E. This is formalized in Definition 2.

Definition 2. *Let $\Psi\{O\}\Phi$ be an operation O, its preconditions Ψ and postconditions Φ. Given an execution E, we say that an object O is faulty if there exists an operation O, executed on O during E, and a formula Φ', such that an $\langle O, \Phi'\rangle$-fault occurs in E.*

In this work we use the notion of *total correctness*, rather than *partial correctness*, in which the postconditions must be met only if the operation terminates. The use of total correctness captures only the scenario of responsive faults, by requiring that the operation terminates. Namely, we do not consider a nonresponsive fault as a structured fault with relaxed postconditions. However, Definition 1 and 2 can be easily modified to capture the scenario of nonresponsive faults as well (by using partial correctness conditions and updating them accordingly).

Following [2], we distinguish the case in which the number of functional faults per faulty object is bounded, from the case of infinitely many faults per faulty object. In both cases, there are no restrictions on the frequency of the faults or the identity of the executing processes that cause them.

As in previous work [2, 30], we overcome functional faults by composing objects so that a limited number of faults does not foil the correctness of the entire computation. Several parameters determine the nature of faults during computation. The first is the maximum number f of faulty objects in the execution. The second parameter is a limit t on the number of faults per faulty object. This limit can be bounded or unbounded (i.e., ∞). In some scenarios, it is even possible to implement a fault tolerant object from a set of objects that are all faulty, if the number of faults per object is bounded. The last parameter is the number of executing processes n. Similarly to the notion of consensus numbers [26], some solutions may be reliable for a limited number of processes only.

Definition 3. *Let $f \in \mathbb{N}^+$ and $t, n \in \mathbb{N}^+ \cup \{\infty\}$. We say that an implementation is (f, t, n)-tolerant for a task S, if in any execution that involves at most n processes, with at most f faulty objects, and with at most t functional faults per faulty object, S is computed correctly. We say that an implementation is (f, t)-tolerant when it is (f, t, ∞)-tolerant and f-tolerant when it is (f, ∞, ∞)-tolerant.*

In Definition 3, $t = \infty$ stands for an unbounded number of faults per faulty object and $n = \infty$ stands for an unbounded number of processes in the system. The construction and lower bound examples in this work deal with a single kind of fault and a single type of object. Nevertheless, in general, the definition allows us to present a discussion about a mix of object types and a mix of functional faults.

3.3 The Overriding CAS Functional Fault

We demonstrate functional faults using the *overriding fault* of the CAS operation. We consider CAS objects that allow a single operation – the CAS operation. In particular, such objects do not allow a read operation. When executing the CAS operation on the CAS object O, the new value val is written to the target register even if its original content old is not equal to the expected value exp received as input. More formally, let us denote by R' the value of the CAS object O on entry to the CAS execution (and by R the value of O at the end of the invocation). Accordingly, the standard postconditions Φ of the operation $old \leftarrow CAS(O, exp, val)$ comprise the formula:

$$R' = exp \ ? \ R = val \ \wedge \ old = R' \ : \ R = R' \ \wedge \ old = R'$$

while the overriding postconditions Φ' consist of the formula:

$$R = val \ \wedge \ old = R'$$

Note that this functional fault is responsive (a CAS execution always terminates, in a wait-free manner), and that even when a fault

occurs, the output is correct. i.e., it returns *old*. As stated in Section 2, we denote a (correct or faulty) CAS execution *successful* if the new value is written to the target register.

3.4 Other Functional Faults in the CAS Object

We chose the CAS primitive as our functional fault example because it is widely used in practice and because it is a strong synchronization primitive. Theoretically, its consensus number is ∞ (a single reliable CAS object solves consensus for any number of processes). We will show that the simple and natural overriding fault can cause the CAS object's consensus number to be reduced to any natural number (shown in Section 5).

Specifically, the occurrence of an overriding fault is possible due to the hardware implementation of the CAS primitive [37]. The implementation involves a comparison of the expected and current register value, which may yield an incorrect answer due to energy-aware computations [16, 42, 46] or soft-errors [45].

There are other possible CAS functional faults, but as discussed next, they are less interesting to study. Some of them are simple enough to be reducible to data faults, implying that they can be dealt with using techniques from previous work [2, 30]. Other faults can easily be shown as impossible to overcome. Let us review possible alternative faults.

A Silent Fault. The new value is not written to the target register, even when its original content is equal to the expected value. When the total number of faults is bounded, each process can execute the original protocol presented in [26], until one process succeeds and an output is chosen. When the total number of faults is unbounded, one can construct an execution in which no process ever updates the CAS object and the protocol never terminates. From a different perspective, such a fault can be modeled as a nonresponsive data fault, and it then relates to the next presented type.

A Nonresponsive Fault. As proven in [30], solving the consensus problem for any number of processes, with a nonresponsive CAS object (even with only one nonresponsive fault), would imply that it is possible to solve the consensus problem for two processes, of which one may crash, using only read/write registers. This would violate the impossibility result shown by Loui and Abu-Amara [35] and Dolev et al. [14].

An Invisible Fault. The invisible fault causes the output of the CAS operation to be incorrect, i.e., the returned *old* value is not equal to the original content of the CAS object. This fault can be considered as a memory data fault according to the model introduced by Afek et al. [2]. A CAS execution in which the *old* output parameter is incorrect can be replaced by a fault operation that replaces the register's content right before the CAS with the returned *old*, and another one that writes the correct value right after the CAS. This way, the *old* parameter is actually correct, but the execution is indistinguishable to the one with the functional fault.

An Arbitrary Fault. This fault occurs when an arbitrary new value is written to the CAS object (regardless of the operation input value). This case is similar to the responsive arbitrary data fault, presented in [30], in the following way: Every faulty CAS execution in the functional fault model can be replaced by the same CAS execution, followed by an arbitrary change of the object. The data fault allows additional similar errors at other execution times, i.e., not only when executing the CAS operation. We do not, however, see how restriction of the functional model can provide an advantage over the data fault model. Therefore, the solution suggested in [30] for responsive arbitrary faults using $O(f log f)$ base objects would work.

4 IMPLEMENTING A RELIABLE CONSENSUS OBJECT

In this section we demonstrate how to overcome functional faults by constructing a reliable consensus object from CAS objects that may manifest the overriding fault (see Section 3.3). Notice that our constructions beat the impossibility result for building consensus using faulty CAS objects [2] in the data faults model, showing that the functional faults model is more expressive than this model.

A construction for two processes is described in Section 4.1. For a system with $n > 2$ processes, a solution for an unbounded number of faults per faulty object appears in Section 4.2, and a construction for a bounded number of faults is presented in Section 4.3.

4.1 A Protocol for Two Processes

We start by pointing out an anomaly for the case when only two processes execute the consensus protocol. It turns out that in this case we can easily overcome the overriding CAS fault. The following theorem asserts that one can implement a consensus object for two processes, even using only one faulty CAS object, and even when the number of faults per faulty object is unbounded.

Theorem 4. *For any $f \in \mathbb{N}$, there exists an $(f, \infty, 2)$-tolerant consensus implementation that uses a single CAS object.*

Proof. The protocol for two processes appears in Figure 1. It uses a single CAS object O, which is initialized with $\perp$. Let p_0 and p_1 be the two processes executing the protocol, and let v_0 and v_1 be their input values, respectively.

```
1: decide(val)
2:     old ← CAS(O, ⊥, val)
3:     if (old ≠ ⊥) then return old
4:     else return val
```

Figure 1: An $(f, \infty, 2)$-tolerant implementation of consensus

Validity. A process p_i either returns v_i in line 4, or O's old content in line 3. Since O's old content in this case is not $\perp$, it must be the content written in line 2 by p_{1-i}, which is v_{1-i}. In both cases, the value returned is the input to some process.

Consistency. Let v_i be the first value written into O in line 2. Since it is the first value written into O, O's old content is $\perp$ and p_i returns v_i in line 4. When v_{1-i} executes line 4, O's old content is v_i. Therefore, p_{1-i} returns v_i in line 3. Both processes agree on v_i, which derives consistency.

Wait-freedom. each process finishes the protocol after at most three steps.

4.2 An Unbounded Number of Faults per Object

Suppose that there are at most f faulty CAS objects in the execution, each with an unbounded number of faults. For these objects, all CAS executions may incorrectly succeed, even if the target register's current value is different from the expected value.

Theorem 5. *For any $f \in \mathbb{N}^+$, there exists an f-tolerant consensus implementation that uses $f + 1$ CAS objects.*

Proof. Although we are dealing with a different type of faults, we can adopt a solution similar to the one in [30] for data faults. We use $f + 1$ CAS objects, out of which f may be faulty, in order to implement a reliable consensus object. The solution appears in Figure 2. Denote the $f + 1$ CAS objects by $O_0, O_1, \ldots, O_f$, and assume they are initialized with $\perp$. The method input parameter is val and the local variable that holds the current decided-upon value is $output$. In addition, each process has a local variable old for holding CAS results. For convenience, we denote the var local variable of a process p with var_p.

```
1: decide(val)
2:     output ← val
3:     for i = 0 to f do
4:         old ← CAS(O_i, ⊥, output)
5:         if (old ≠ ⊥) then output ← old
6:     return output
```

Figure 2: An f-tolerant implementation of consensus

Each process p starts by setting $output_p$ to hold its initial estimation of the decided-upon value (line 2). Next, each process tries to set each of the CAS objects to its current estimation. When a certain register already contains another input value, meaning $old_p \neq \perp$, p adopts old_p as its new estimation for the decided-upon value (line 5). The output of a process p is the final content of $output_p$.

Validity. Assume by contradiction that at some point during the execution, either a CAS object O_i ($0 \leq i \leq f$) or an $output_p$ variable of a process p contains a value different from $\perp$, and which is not an input to some process. Initially, all CAS objects contain $\perp$ and $output_p$ contains val_p and therefore, it must happen after a write to O_i in line 4 or a write to $output_p$ in line 5. Now, consider the first such write. If it is a write to O_i in line 4, then the value written is the content of some $output_q$ variable at an earlier point during the execution. If it is a write to $output_p$ in line 5, then the value written is the content of some O_t CAS object at an earlier point during the execution. From the choice of this write as the first one of a value which is neither $\perp$ nor an input one, we get a contradiction in both cases. Since $\perp$ cannot be written into $output_p$ in line 5, $output_p$ always holds a value which is an input to some process. Therefore, the value returned in line 6 is indeed the input of some process.

Consistency. Let O_j ($0 \leq j \leq f$) be a non-faulty CAS object (there must be at least one), and let x be the first value written into O_j during the execution. Since O_j is initialized to $\perp$, x is written during the first execution of line 4 in the j-th iteration, by some process p, having $output_p = x$. As shown in our validity argument above, $output_p$ cannot contain the value $\perp$, and thus $x \neq \perp$. Since O_j is non-faulty, all of the next update trials to update it in line 4, assuming it contains $\perp$, fail. I.e., when a process $q \neq p$ executes line 5 in the j-th iteration, it updates its $output_q$ variable to hold x. In addition, since p does not change the content of its $output_p$ variable in this iteration, it remains x. Therefore, for every $j < i \leq f$, even if O_i is faulty, each process q (either $q = p$ or $q \neq p$) tries to write x to O_i and thus, $output_q$ does not change anymore. In the end, all of the processes return the same value, x.

Wait-freedom. Each process finishes the protocol after $f + 1$ iterations of the loop in lines 3-5. Each iteration terminates after a finite number of steps, regardless of the behavior of the rest of the processes. □

4.3 A Bounded Number of Faults per Object

Here, as in Section 4.2, we consider more than two processes, i.e., $n > 2$ (otherwise, see Section 4.1). Interestingly, when the number of faults per faulty object is bounded, there exist reliable solutions that use only faulty objects. Theorem 6 asserts that we can implement a consensus object that is fault-tolerant for f faulty CAS objects, each manifesting at most t functional faults, with f CAS objects overall. This theorem, however, holds only when the number of threads n is at most the number of faulty objects f plus one. This is tight as we show in Section 5.2. For every $n > f + 1$, there does not exist an (f, t, n)-tolerant implementation of consensus that uses f CAS objects. Alternatively, if $f + 1$ CAS objects are allowed, i.e., one object is guaranteed to be correct, then one can deal with any number of processes as shown in Section 4.2.

Theorem 6. *For every $f, t \in \mathbb{N}^+$, there exists an $(f, t, f + 1)$-tolerant implementation of consensus that uses f CAS objects.*

Proof. The implementation appears in Figure 3. We use f CAS objects, all of which may be faulty, that are initialized with $\perp$, and denoted $O_0, O_1, \ldots, O_{f-1}$. In addition, we assume a finite set of processes, denoted $p_0, p_1, \ldots, p_f$, and denote the local variable var of a process p_j with var_j. The idea here extends the protocol of Section 4.2: the execution is divided into $maxStage + 1$ stages. In the first $maxStage$ stages, each process p_j tries to write its current decision, saved in its $output_j$ variable, together with its current stage, saved in its s_j variable, to all of the f CAS objects (lines 3–18), and in the last stage, to O_0 (the final stage, presented in lines 19-24). During the entire execution, the exp_j local variable is updated in order to hold the current content of the next CAS object to be updated. As we are going to show next, setting $maxStage$ to $t \cdot (4f + f^2)$ (as done in line 2) is sufficient for achieving consistency (validity and wait-freedom are achieved for any choice of $maxStage$). The final common decision is the $output_j$ value written to O_0 in line 20, together with the maximal stage. Notice that choosing an earlier maximal stage might work, but we chose to concentrate on correctness and space complexity rather than on performance issues.

During the first $maxStage$ stages (lines 3–18), p_j tries to write its current decided-upon value (initially its input val_j) to each of the CAS objects, together with a stage indicator s_j (line 6). After executing the CAS operation on O_i, the only indication of whether it was a successful one is received when the content of the exp_j variable is equal to the original content of O_i, assigned into the

old_j local variable in line 6. If they are equal, p_j can safely move to the next CAS object in line 16. When the contents of the exp_j and old_j variables are not the same, p_j cannot distinguish between the following two scenarios: (1) the CAS execution is unsuccessful, and (2) the write involves a fault that causes $\langle output_j, s_j \rangle$ to be written to O_i. Therefore, these two scenarios are handled in the same manner (lines 7–15). If p_j's current stage is earlier than or equal to the last stage written to O_i (the condition in line 8 holds), then another process has already updated O_i, during that same stage or a later one. In this case, p_j adopts the new decision value (possibly identical to its own current decision) and updates its current stage accordingly (lines 9–10). It then updates its exp_j variable to hold the expected content of the next CAS object (line 13), and moves on to the next CAS object (line 14). Notice that p_j does not try to update O_i again, even when a fault occurs. If p_j's current stage is later than the last stage written to O_i (the condition in line 8 does not hold), then there is no need to update p_j's current decision value. In this case, exp_j is updated to hold the expected current value of O_i (line 15), to ensure success in the next CAS trial.

```
1:  decide(val)
2:      output ← val ; exp ← ⊥ ; s ← 0 ; maxStage ← t · (4f + f²)
3:      while (s < maxStage) do
4:          for i = 0 to f − 1 do                          ▷ Handling O_0, ..., O_{f−1}
5:              while (true)
6:                  old ← CAS(O_i, exp, ⟨output, s⟩)
7:                  if (old ≠ exp)
8:                      if (old.stage ≥ s)                 ▷ Needs to update output
9:                          output ← old.val
10:                         s ← old.stage
11:                         if (s = maxStage)
12:                             return output              ▷ The decided value
13:                         exp ← ⟨old.val, old.stage − 1⟩
14:                         break                          ▷ No need to update O_i
15:                     else exp ← old                     ▷ Still needs to update O_i
16:                 else break                             ▷ A successful CAS execution
17:         exp.stage ← s
18:         s ← s + 1
19:     while (true)                                       ▷ The final stage
20:         old = CAS(O_0, exp, ⟨output, maxStage⟩)
21:         if (old ≠ exp ∧ old.stage < maxStage)
22:             exp ← old
23:         else break
24:     return output
```

Figure 3: An $(f, \mathbf{t}, f+1)$-tolerant implementation of consensus

To prove that the protocol presented in Figure 3 is an $(f, t, f+1)$-tolerant implementation of consensus, we need to show that, given that the number of faults per object is at most t, that $maxStage$ is set to $t \cdot (4f + f^2)$, and that there are at most $f + 1$ processes in the system, it is valid, consistent and wait-free.

We offer some claims regarding executions that include overriding faults. Our first claim asserts that the $output_j$ variable of a process p_j always holds the input value of some process p_k, which directly implies validity.

Claim 7. *Throughout the execution, it holds that (1) each $output_j$ variable contains an input value, and (2) for every $0 \leq i \leq f-1$, O_i contains either $\perp$ or $\langle x, n \rangle$ for some input value x and a stage number $0 \leq n \leq maxStage$.*

Proof. After initialization, the $output_j$ variable of each process p_j contains its input, and O_i contains $\perp$ for every $0 \leq i \leq f-1$. Now, assume by contradiction that at some point during the execution, the claim does not hold for the first time. There are two possible scenarios.

In the first scenario, the $output_j$ local variable of a certain process p_j contains a value which is not an input value of any process. Since p_j only updates $output_j$ in line 9, with a value read from some CAS object O_i, we get a contradiction. In the second scenario, a certain CAS object O_i contains a value which is neither $\perp$ nor $\langle x, n \rangle$ for some input value x and a stage $0 \leq n \leq maxStage$. Since O_i is only updated in lines 6 and 20, with a value saved in some $output_k$ variable and a stage $0 \leq n \leq maxStage$, we get a contradiction in this case as well. Since every possible scenario derives a contradiction, the claim holds throughout the execution. □

Our next two claims relate to the fact that p_j's stage (saved in its s_j local variable), can only increase over time, and that it cannot increase before the previous stage is written to all of the CAS objects.

Claim 8. *Let $0 < n_1 < n_2 \leq maxStage$ be the stage values assigned to the s_j variable of some process p_j during the execution. Then n_1 is assigned to s_j before n_2.*

Proof. Let n be the stage value assigned to the s_j variable of some process p_j at some point during the execution, and let n' be s_j's content right before this assignment. The assignment is done either in line 10, in which n is not smaller than n' (since the condition checked in line 8 must hold) or in line 18, in which $n = n' + 1$. I.e., the stage value assigned into a stage variable is never smaller than its old content, and the claim follows. □

Claim 9. *Let x be a value, $0 \leq n_1 \leq maxStage$ a stage number and $0 \leq i \leq f-1$. If $\langle x, n_1 \rangle$ is written to O_i, then (1) for every $n_0 < n_1$ and $k \leq f-1$, $\langle x, n_0 \rangle$ was written to O_k at some point before this write, and (2) for every $k < i$, $\langle x, n_1 \rangle$ was also written to O_k at some point before this write.*

Proof. We are going to prove the claim by induction on n_1 and i. For the base case, the claim vacuously holds when $n_1 = 0$ and $i = 0$. For the induction step, let $\langle x, n_1 \rangle$ be the value written to O_i. If $i \neq 0$, then the write is executed in line 6, in the i-th iteration of the loop. If the executing process p_j wrote the same value x and n_1 to O_{i-1}, during the previous loop iteration, then by the induction assumption, we are done. Otherwise, p_j must have updated its output and stage (in lines 9-10) during the previous loop iteration, to hold the content of O_{i-1}. Therefore, $\langle x, n_1 \rangle$ had already been written to O_{i-1} by some process, and by the induction assumption, the claim still holds.

The remaining case is when $i = 0$, $n_1 > 0$, and the current write is either an execution of line 6 or line 20. The latest update of the s_j variable of the executing process, p_j, is executed in line 18, during the previous iteration of the main *while* loop (lines 3-18). i.e, during the last iteration of the *for* loop, either $\langle x, n_1 - 1 \rangle$ was written to O_{f-1}, or that $output_j$ and s_j were updated to hold x and $n_1 - 1$ (respectively) in lines 9-10. In both cases, $\langle x, n_1 - 1 \rangle$ had

already been written to O_{f-1} by some process, and by the induction assumption, the claim still holds. □

Observation 10. *The protocol consists of $t \cdot (4f + f^2)$ stages, each including at least f successful writing trials (by Claim 9). Since at most $t \cdot f$ faults may occur (at most t faults per CAS object), during the execution there must be a sequence of $4f + f^2$ consecutive non-faulty writes to the CAS objects.*

Let us denote the latest stage, written to any CAS object **before** the first such sequence (guaranteed by Observation 10) during the execution, with m. By Claim 9, stage $m-1$ is already written to all of the CAS objects before that guaranteed sequence begins.

Definition 11. *Let $S \subseteq \{O_0, O_1, \ldots O_{f-1}\}$ be the set of CAS objects which are overridden with a stage earlier than $m-1$, after containing the stage $m-1$ for the first time during the execution.*

Claim 12. *After the sequence, guaranteed by Observation 10, begins, at most $f - |S|$ processes write stages which are smaller than $m-1$ to the CAS objects.*

Proof. When a process overrides a CAS object with a smaller stage, it immediately updates its local stage in line 10. In addition, according to Claim 8, the stage of a process can only increase. Therefore, a process would not override a CAS object $O_i \in S$ with a stage $< m-1$ more than once. All of the overrides must occur before the guaranteed sequence begins (since it does not contain overriding faults, by the choice of it) and thus, at least $|S|$ processes (the ones which executed these overriding faults) are at a stage $\geq m-1$ when the sequence begins. In addition, according to Claim 8, the process that wrote the stage $m-1$ to O_0 for the first time, can no longer override any other object with a smaller stage. Therefore, this process is not one of the $|S|$ processes mentioned above, and it would indeed not write a stage which is smaller than $m-1$ after the guaranteed execution begins. There are at most $f+1$ processes in the system, and in total we get at most $f + 1 - |S| - 1 = f - |S|$ processes, and the claim holds. □

Claim 13. *Let $0 \leq i \leq f-1$. If $\langle x_1, n_1 \rangle$ is the content of O_i, replaced by $\langle x_2, n_2 \rangle$ via a successful, non-faulty CAS execution, then $n_2 > n_1$.*

Proof. In a successful, non-faulty CAS execution by a process p_j, exp_j contains the old content of the CAS object. When $exp_j = \perp$, the claim holds vacuously. In addition, from Claim 8, s_j can only increase over time. Therefore, it is sufficient to show that throughout the execution, right after exp_j is updated, $s_j > exp_j.stage$.

The exp_j is updated in lines 13, 15, 17 and 22. After executing line 13, s_j is equal to $old_j.stage$ (assigned in line 10) and $exp_j.stage$ is equal to $old_j.stage - 1$. After executing line 15, $exp_j.stage$ is equal to $old_j.stage$, which is smaller than s_j (since the condition checked in line 8 does not hold in this case). Right after executing line 17, in which s_j is assigned into $exp_j.stage$, s_j is incremented. Finally, after executing line 22, $exp_j.stage$ is equal to $old_j.stage$, which is smaller than s_j (since the condition checked in line 21 must hold). □

Claim 14. *Let $0 \leq i \leq f-1$. During the sequence guaranteed by Observation 10, after at most f^2 writes, O_i contains a stage of at least $m-1$.*

Proof. Let $0 \leq i \leq f-1$. If $O_i \notin S$, then by Claim 13, O_i contains a stage of at least $m-1$ since the first write of this stage, and throughout the guaranteed sequence. Now, Let p_j be a process that writes a stage $< m-1$ after the sequence begins. By Claim 12, it holds that $f - |S| \geq 1$ and thus, $f - 1 \geq |S|$. i.e., there is at least one CAS object which has not been overridden with a stage smaller than $m-1$, since containing it for the first time. After writing to at most $|S| < f$ CAS objects, p_j would encounter an object with a stage of at least $m-1$, would not override it (by the choice of the sequence), and would update its stage to be at least $m-1$. Overall, there are at most $(f - |S|$ such processes, each executing at most $|S|$ writes before getting to the $(m-1)$-th stage. Therefore, by Claim 13, after at most $(f - |S|) \cdot |S| \leq f^2$ writes, each CAS object O_i contains a stage of at least $m-1$. □

By Claim 14, during the sequence guaranteed by Observation 10, after f^2 successful CASes, if $\langle x, n \rangle$ is written to O_i then $n \geq m-1$. By Claim 13, after $f + f^2$ out of the $4f + f^2$ guaranteed in the scope of the sequence, only stages of at least m are written to the CAS objects. By Claim 13, we get:

Observation 15. *Let $0 \leq i \leq f-1$. During the sequence guaranteed by Observation 10, after $4f + f^2$ successful CASes, if $\langle x, n \rangle$ is written to O_i then $n \geq m+2$.*

Claim 16. *Let $0 \leq i \leq f-1$ and let $\langle x, n_1 \rangle$ be the content of O_i when the sequence guaranteed by observation 10 ends. Then for every $0 \leq j \leq f-1$, O_j contains the value $\langle x, n_2 \rangle$ for some stage n_2.*

Proof. Assume by contradiction that there exists $0 \leq j \leq f-1$ for which O_j contains $\langle y, n_2 \rangle$ for some $y \neq x$. Notice that both n_1 and n_2 (may be equal) are bigger than $m+1$. By Claim 9, $\langle x, m+1 \rangle$ and $\langle y, m+1 \rangle$ have already been written to all of the CAS objects. By the choice of m, both type of writes have occurred during the sequence guaranteed by Observation 10, containing only non-faulty writes. By Claim 13, we get a contradiction. Therefore, for every $0 \leq j \leq f-1$, O_j contains the value $\langle x, n_2 \rangle$ for some stage n_2, and the claim holds. □

By Claim 16, after the sequence ends, all of the CAS objects contain the same decision, x. In addition, by Observation 15 they contain a stage of at least $m+2$. By Claim 9, for every $y \neq x$ and a stage $n_1 \geq m+1$, $\langle y, n_1 \rangle$ has not been written to any O_i at this point. Our next claim shows that there exists no value $y \neq x$ and a stage $n_1 \geq m+2$ for which $\langle y, n_1 \rangle$ is written to a CAS object during the execution.

Claim 17. *Let $0 \leq i \leq f-1$ and $y \neq x$. Then $\langle y, m+2 \rangle$ is never written to O_i during the execution.*

Proof. Assume by contradiction that $\langle y, m+2 \rangle$ is written to some O_i during the execution. By the choice of m, it must happen after the guaranteed sequence ends. By Claim 9, it also must hold that before this write, $\langle y, m+1 \rangle$ is written to $O_0, O_1, \ldots, O_{f-1}$, in that order. In particular, by Claim 13, all of the CAS objects must be overridden before this write.

For every $0 \leq k \leq f-1$, we denote with O_{i_k} the k-th CAS object which is overridden with $\langle y, n_1 \rangle$ (for some $y \neq x$ and $n_1 \leq m+1$) for the first time after the guaranteed sequence. We are going to prove, by induction on k, that after $\langle y, n_1 \rangle$ is written to O_{i_k}, there

are at most $f - k - 1$ processes in the system having $output_j \neq x$ and $s_j < m + 2$.

For the base case, Let p_j be the process executing the last successful write during the guaranteed sequence. Then $output_j = x$ and $s_j \geq m + 2$. In addition, let p_h be the process overriding O_{i_0} with $\langle y, n_1 \rangle$. By the choice of i_0, p_h overrides $\langle x, n_2 \rangle$ for some $n_2 \geq m + 2$, and immediately updates s_h to be n_2 and $output_h$ to hold x. Indeed, after the first such faulty write, there are at most $(f + 1) - 2 = f - 1 = f - 0 - 1$ processes in the system having $output_j \neq x$ and $s_j < m + 2$.

For the inductive step, let p_r be the process overriding $O_{i_{k+1}}$ for the first time with a value $y \neq x$ and a stage $n_1 \leq m + 1$. By the induction hypothesis, there are already $k + 2$ processes having $output_j = x$ and $s_j \geq m + 2$. By Claim 8, p_r is not one of them. By the choice of i_{k+1}, p_r overrides $\langle x, n_2 \rangle$ for some $n_2 \geq m + 2$, and immediately updates s_r to be n_2 and $output_r$ to hold x. At this point, there are at most $(f + 1) - (k + 2) - 1 = f - (k + 1) - 1$ processes in the system having $output_j \neq x$ and $s_j < m + 2$.

Notice that as long as no other output value is being written, together with a stage which is at least $m + 2$, a process p_j does not change its $output_j$ variable. Therefore, after $\langle y, m + 1 \rangle$ is written to $O_0, O_1, \ldots, O_{f-1}$, for some value $y \neq x$. there are at most $f - (f - 1) - 1 = 0$ processes in the system, having a value different from x, and we get a contradiction to the assumption that $\langle y, m + 2 \rangle$ is written to some O_i during the execution, for some $y \neq x$. □

We are now ready to prove that when the number of faults per faulty object is bounded by t, there are at most f faulty objects and at most $f + 1$ processes in the system, the protocol presented in Figure 3 is valid, consistent and wait-free.

Validity. By Claim 7, the $output_j$ variable of a process p_j always contains an input of some process p_h. The decision value of a process p_j is the content of its $output_j$ variable (returned either in line 12 or 24), and validity derives.

Consistency. First, notice that if there are no faulty writes during the execution, then besides the process p_j that executes the first successful write of $\langle val_j, 0 \rangle$ to O_0 in line 6, all processes adopt that same decision value upon their first write trial to O_0, in line 9. Therefore, the decision value of each process remains v_j (the content of the $output_h$ variable of a process p_h does not change anymore, even when executing line 9 again), and consistency holds.

The case in which faults do occur is more complicated. According to Claim 9 and 17, there exists a single value x for which $\langle x, maxStage \rangle$ is written to O_0. In addition, the condition checked in line 3 guarantees that it is only written in line 20. Therefore, each process p_j either returns x in line 12, after reading $\langle x, maxStage \rangle$ from O_0, or updates $output_j$ to hold x during the former stage at the latest, and returns it in line 24, and consistency derives.

Wait-Freedom. From Claim 13, after a sufficient number of writes to each CAS object (out of which at most t can be faulty), its content should remain stable (containing a stage that is equal to $maxStage$ for O_0 and $maxStage - 1$ for the rest of the CAS objects). The loop in lines 5–16 terminates upon reading a bigger stage (in line 14) or executing a successful CAS (in line 16). In addition, the loop in lines 3–18 eventually terminates, since by Claim 8 the stage s_j of a process p_j can only increase. For similar reasons, the loop in lines 19–23 eventually terminates as well, and wait-freedom follows. □

5 ACHIEVING CONSENSUS - IMPOSSIBILITY RESULTS

In this section we show that the upper bounds presented in Section 4 are tight. This easily holds for Section 4.1, since the solution for two processes uses a single CAS object, and no solution can use fewer objects (than 1). In Section 5.1, we show that an (f, ∞, n)-tolerant consensus implementation that uses fewer than than $f + 1$ base objects (when $n > 2$) does not exist. Finally, we show, in Section 5.2, that for every $f, t \in \mathbb{N}^+$ and $n > f + 1$, there is no (f, t, n)-tolerant consensus that uses f CAS objects exists.

5.1 An Unbounded Number of Faults per Object

In this section we show that the upper bound presented in Section 4.2 is tight. i.e., when the number of faults per faulty object is unbounded and there are more than two processes in the system, a reliable consensus implementation that uses only faulty objects does not exist.

Theorem 18. *For any $f \in \mathbb{N}$, $n > 2$, it is impossible to implement an (f, ∞, n)-tolerant consensus protocol using f CAS objects and an unbounded number of read/write registers.*

Proof. As defined in [26], we say that while executing a consensus protocol, the global state of the system is *multivalent* if there are at least two decision values that are still possible in some extension of the current execution. I.e., given that the current state is multivalent, the protocol may terminate with several output values, depending on the future schedule. When only one decision value remains possible (for every extension of the current execution), we say that the state is *univalent*, or *x-valent* (when that value is x). A *decision step* is a step, executed by one of the processes, which turns the system's state from multivalent to univalent. This step may either be a read, a write or another atomic operation (for example, a CAS execution).

When there are at least two different input values, the system's initial state must be multivalent (implied by the validity condition). Because the protocol is wait-free, letting a process run by itself, starting from any multivalent state, would eventually lead to a decision step. Therefore, the standard impossibility technique [26] is to construct an execution, carrying the protocol to a multivalent state where the next step of each running process is a decision step. Moreover, decision steps made by different processes would lead to states with different valencies. Then, it is shown that although each decision step, made by a certain group of two or more processes, should carry the protocol to univalent states with different valencies, there exists a process that cannot distinguish between these two states and thus, would output the same decision value after a solo run, starting after each of these two step possibilities. This way, a contradiction is derived (since a process cannot output x when the system is y-valent and $x \neq y$).

In a similar way to [21, 34], the standard impossibility technique cannot be used directly in our fault model, since it cannot be modified to fit non-deterministic executions. The specification of a step

as a decision step may depend on the fact that it is not the execution of a faulty operation. Although we face different kinds of non-determinism (for example, in [21], the shared objects are all considered to be reliable), we use a similar extension that adjusts the standard impossibility technique to a non-deterministic execution. We define a reduced model, in which the CAS executions of a certain process are always faulty (this is possible since the number of faults is unbounded), and faults are caused only by this process. More specifically, we denote the executing processes with $p_1, p_2, \ldots, p_n$, and assume that, given a faulty CAS object O, whenever p_1 executes CAS(O, *exp*, *new*), *new* is written to O even if the original content of O is not equal to *exp*. For every $2 \leq i \leq n$, all CAS executions, done by p_i, are non-faulty (even if the CAS object is a faulty one), and follow the operation's sequential specification.

Now assume by contradiction that there exists an (f, ∞, n)-tolerant consensus implementation, that uses f CAS objects and an unbounded number of read/write registers, in the reduced model. We assume, without loss of generality, that p_1's input value is different then p_2's input value. Consider the following execution, starting from the initial state (which is multivalent, from validity). First, p_1 executes a sequence of operations until it reaches a state in which the next step would be a decision step. Then, p_1 is halted and p_2 executes a sequence of operations, until its next step is a decision step. If p_1's next step is no longer a decision step, it is run again until it reaches a similar state, and so on, until both p_1 and p_2 are about to make a decision step. The system must reach such a state, since the protocol is wait-free (i.e., both processes must terminate after a finite number of steps) and consistent (the system must reach a univalent state eventually). We denote the current multivalent state by s_0, the x-valent state the system would reach if p_1 is the next process to take a step by s_1, and the y-valent state the system would reach if p_2 is the next process to take a step by s_2. By the definition of a decision step, and since s_0 is a multivalent state, $x \neq y$. As proven in [26], if at least one of their next steps is a read from a read/write register, or if p_1 and p_2's next steps are executed on different shared objects, then s_1 and s_2 are indistinguishable to p_3. Therefore, in a solo-run of p_3, starting from either s_1 or s_2, it would output the same decision value, in contradiction to the fact that s_1 is x-valent and s_2 is y-valent.

If p_1 and p_2's next steps are writes to the same read/write register, then let the state the system would reach if p_1 is the next process to take a step after s_2, be s_3. Notice that s_3 is also y-valent, and that s_1 and s_3 are indistinguishable to p_3. In a solo-run of p_3, starting from either s_1 or s_3, it would output the same decision value, in contradiction to the fact that s_1 is x-valent and s_3 is y-valent.

Therefore, the next operation of p_1 and p_2 must be a CAS execution, executed on the same object O. Let s_2' be the state of the system, reached from s_0 after p_2 executes its CAS on O and then p_1 executes its CAS on O. Obviously, s_2' is a y-valent state. Since O must be a faulty CAS objects (all of the f CAS objects are faulty), p_1's CAS execution overrides p_2's and thus, s_1 and s_2' are indistinguishable to p_3. Again, in a solo-run of p_3, starting from either s_1 or s_2', it would output the same decision value, in contradiction to the fact that s_1 is x-valent and s_2' is y-valent.

There does not exist a reliable consensus implementation that uses only faulty objects in the reduced model. Therefore, there does not exist such an implementation in our fault model. □

5.2 A Bounded Number of Faults per Object

Although one can implement an $(f, t, f+1)$-tolerant implementation of consensus that uses f CAS objects (as shown in Section 4.1 for $f = 1$ and in Section 4.3 for $f > 1$), for every $f, t \in \mathbb{N}^+$ and $n > f + 1$, there does not exist an (f, t, n)-tolerant implementation of consensus that uses less than $f + 1$ CAS objects.

Theorem 19. *For any $f, t \in \mathbb{N}^+$, it is impossible to implement an $(f, t, f+2)$-tolerant consensus protocol using f CAS objects.*

Proof. We show that for every $f \in \mathbb{N}^+$, one cannot implement an $(f, 1, f+2)$-tolerant consensus object that uses f CAS objects (since we are going to show a lower bound, it suffices to show it for $t = 1$), using covering arguments [7, 9, 18, 31, 36].Let $p_0, p_1, \ldots, p_{f+1}$ be the $f+2$ system's processes, $O_0, O_1, \ldots, O_{f-1}$ be the f CAS objects, and for every $0 \leq i \leq f + 1$, let v_i be p_i's input value. Without loss of generality, we assume that $O_0, O_1, \ldots, O_{f-1}$ are initialized to $\perp$, that for every $0 \leq i \leq f + 1$, $v_i \neq \perp$, and that for every $1 \leq i \leq f + 1$, $v_i \neq v_0$. For our impossibility result, we will build a certain execution.

We define our execution as follows: First, p_0 runs alone until it returns its own input value, v_0 (from wait-freedom and validity). Then, for every $1 \leq i \leq f$, p_i runs alone until it executes its first CAS to an object not yet written by $p_1, \ldots, p_{i-1}$, denoted by O_{j_i}. Its write to O_{j_i} is faulty and thus overrides O_{j_i}'s previous content, and then it is halted. Notice that for every $1 \leq i \leq f$, p_i is halted right after its first write to an object that is not yet written by $p_1, \ldots, p_{i-1}$ and, therefore, the number of faulty CAS executions per CAS object is 1. It remains to show that for every $1 \leq i \leq f$, p_i indeed writes at least to one such object before it terminates its solo run.

Claim 20. *For every $1 \leq i \leq f$, p_i writes to a CAS object that has not yet been written by $p_1, \ldots, p_{i-1}$.*

Proof. We prove the claim by induction on i. For the base case, since writing to a CAS object is the only way to read its content, if p_1 does not write to any object, it cannot distinguish between this run and another run in which it runs alone. Since in such a run, it must eventually decide (for wait-freedom) and return v_1 (for validity), we get a contradiction to the consistency of the protocol.

For the inductive step, we assume that $p_1, \ldots, p_i$ have already committed their faulty writes to the distinct objects $O_{j_1}, \ldots, O_{j_i}$, respectively, and have immediately been halted. Here as well, if p_{i+1} does not write to a new CAS object, it cannot distinguish between the current run and a run in which the only active processes are $p_1, \ldots, p_{i+1}$. From the above arguments, it would eventually return a value $v \in \{v_1, \ldots, v_{i+1}\}$, and we get a contradiction to the protocol's consistency. □

Given that f processes have committed an overriding fault to f distinct CAS objects, all of p_0's writes have been overridden. Notice that each process $p_1, \ldots, p_f$ cannot distinguish between the current execution and an execution in which p_0 is not active, until it commits its faulty write, and is halted. Therefore, when p_{f+1} starts a solo-run from the point of p_f's overriding fault, it cannot distinguish between this run and a run in which p_0 is not active at all. Eventually, from wait freedom and validity, p_{f+1} would return a value $v \in \{v_1, \ldots, v_{f+1}\}$, and we get a contradiction to the protocol's consistency. In conclusion, for every $f \geq 1$, one cannot

implement an $(f, 1, f + 2)$-tolerant consensus object that uses f CAS objects. □

Notice that by combining the results of Sections 4.3 and 5.2, we show that, when there is a bound on the total number of faults per faulty object, the number of available CAS objects determines the consensus number [26] of the entire construction. The consensus number of a set of f faulty CAS objects (each with a bounded number of overriding faults) is $f + 1$. This way, we place a faulty setting in each level of the Herlihy consensus hierarchy.

6 RELATED WORK

Much previous work was discussed in the introduction and throughout the paper. We discuss additional related work here.

Another approach for dealing with process failures is to employ a recovery mechanism enabling a process that has crashed to recover and complete its execution after going through a predefined recovery protocol (e.g., for critical sections [23] or consensus protocols [21, 22]). Similar techniques are used with architectures that employ NVRAM (e.g., [13, 20, 29]).

As mentioned, Afek et al. [2] and Jayanti et al. [30] presented two independent models of memory data faults. Both papers presented upper and lower bounds on the number of base objects, some of which may be faulty, needed for constructing a reliable shared one. Examples of such objects are read/write registers, *test&set* bits and consensus objects. Further studies of faults in these models include [1, 4, 24, 40]. Jayanti et al. also classified the severeness of various fault types and introduced the notion of *graceful degradation*. Graceful degradation is a property of fault-tolerant implementations, which ensures that even when too many base objects used in the construction are faulty, the construction manifests a fault whose severity remains in the fault class to which its base objects belong.

Byzantine faults[1, 10] signify a worst-case type of process faults (unlike fail-stop, which is benign). In this work, we typically assume worst-case faults, similarly to byzantine process failures.

Traditional correctness conditions [27] and the relaxed postconditions, as defined in our model, are required to hold on every execution. An alternative probabilistic model limits the probability that applications do not adhere to their specifications. Such probabilistic correctness conditions are modeled in [43].

Another related research area studies performance benefits that can be obtained by executing approximate calculations instead of accurate ones [47], and allowing data structures to deviate from their original specifications [3, 5]. For example, the *pop* operation, performed on a relaxed queue, may sometimes return a value that is not the first in line, while still adhering to some predefined relaxed specification. Such constructions are shown to obtain better scalability and performance. These models form a special case of the general functional faults model presented in this paper, and they provide additional intriguing examples. In contrast to the examples provided in this paper, the involved techniques and goals are very different. Our goal is not necessarily to improve performance, but to achieve fault tolerance in the presence of functional faults.

7 CONCLUSIONS AND FUTURE WORK

Faults in the literature can be categorized as process failures or memory faults, in which a a value in memory gets arbitrarily distorted, independently of the behavior of the executing processes. In this work, we initiate a study of functional faults, in which the outcome of the function does not satisfy its specification, but that does follow some structured specifications. We demonstrated this concept with a specific natural functional fault of the important CAS synchronization primitive: the CAS overriding fault. We have shown constructions and impossibilities in the presence of the CAS overriding faults.

It would be interesting to examine other widely used functions with natural faults and understand whether they can be overcome with clever constructions. A very interesting question is whether a way can be found to categorize, in general, which faults can be overcome and which are too harsh to handle. Many other questions remain. Can resources be saved by reusing these constructions? How do general faults affect the consensus hierarchy? Similarly to [30], it would also be interesting to define severity levels of faults in the functional fault model, and then study the possibility of their graceful degradation in the functional fault model.

REFERENCES

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. 2006. Byzantine disk paxos: Optimal resilience with byzantine shared memory. *Distributed Computing* 18, 5 (2006), 387–408.

[2] Yehuda Afek, David S Greenberg, Michael Merritt, and Gadi Taubenfeld. 1995. Computing with faulty shared objects. *Journal of the ACM (JACM)* 42, 6 (1995), 1231–1274.

[3] Yehuda Afek, Guy Korland, and Eitan Yanovsky. 2010. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*. Springer, 395–410.

[4] Yehuda Afek, Michael Merritt, and Gadi Taubenfeld. 1993. Benign failure models for shared memory. In *International Workshop on Distributed Algorithms*. Springer, 69–83.

[5] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A scalable relaxed priority queue. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 11–20.

[6] Ijeoma Anarado, Mohammad Ashraful Anam, Fabio Verdicchio, and Yiannis Andreopoulos. 2016. Mitigating Silent Data Corruptions In Integer Matrix Products: Toward Reliable Multimedia Computing On Unreliable Hardware. *IEEE transactions on circuits and systems for video technology* 27, 11 (2016), 2476–2489.

[7] Hagit Attiya, Alla Gorbach, and Shlomo Moran. 2002. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation* 173, 2 (2002), 162–183.

[8] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2017. Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936* (2017).

[9] James E Burns and Nancy A Lynch. 1993. Bounds on shared memory for mutual exclusion. *Information and Computation* 107, 2 (1993), 171–184.

[10] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[11] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. 2016. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*. Springer, 106–125.

[12] Matei David. 2004. A single-enqueuer wait-free queue implementation. In *International Symposium on Distributed Computing*. Springer, 132–143.

[13] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 373–386. https://www.usenix.org/conference/atc18/presentation/david

[14] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. 1987. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)* 34, 1 (1987), 77–97.

[15] Andrew D Eckhardt and Michael J Koster. 2014. Failure recovery using consensus replication in a distributed flash memory system. US Patent 8,856,593.

[16] Amirhossein Esmaili, Mahdi Nazemi, and Massoud Pedram. 2019. Energy-Aware Scheduling of Task Graphs with Imprecise Computations and End-to-End Deadlines.

[17] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 78.

[18] Faith Fich and Eric Ruppert. 2003. Hundreds of impossibility results for distributed computing. *Distributed computing* 16, 2-3 (2003), 121–163.

[19] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1982. *Impossibility of distributed consensus with one faulty process*. Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE.

[20] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-free Queue for Non-volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. 28–40.

[21] Wojciech Golab. 2018. Recoverable Consensus in Shared Memory. *arXiv preprint arXiv:1804.10597* (2018).

[22] Wojciech Golab. 2019. The Recoverable Consensus Hierarchy. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 212–214.

[23] Wojciech Golab and Aditya Ramaraju. 2016. Recoverable mutual exclusion. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. ACM, 65–74.

[24] Rachid Guerraoui and Marko Vukolić. 2006. How fast can a very robust read be?. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*. ACM, 248–257.

[25] Siva Kumar Sastry Hari, Sarita V Adve, and Helia Naeimi. 2012. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 1–12.

[26] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.

[27] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[28] Amos Israeli and Lihu Rappoport. 1993. Efficient wait-free implementation of a concurrent priority queue. In *International Workshop on Distributed Algorithms*. Springer, 1–17.

[29] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. 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*. 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[30] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. 1998. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)* 45, 3 (1998), 451–500.

[31] Prasad Jayanti, King Tan, and Sam Toueg. 2000. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.* 30, 2 (2000), 438–456.

[32] Leslie Lamport. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[33] Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A Jacobson, and Subhasish Mitra. 2010. ERSA: Error resilient system architecture for probabilistic applications. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 1560–1565.

[34] Wai-Kau Lo and Vassos Hadzilacos. 1997. All of us are smarter than any of us: wait-free hierarchies are not robust. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 579–588.

[35] Michael C Loui and Hosame H Abu-Amara. 1987. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing research* 4, 163-183 (1987), 31.

[36] Nancy A Lynch. 1996. *Distributed algorithms*. Elsevier.

[37] Douglas MacGregor, David S Mothersole, and John Zolnowsky. 1986. Method and apparatus for a compare and swap instruction. US Patent 4,584,640.

[38] F.J. MacWilliams and N.J.A. Sloane. 1978. *The Theory of Error-Correcting Codes* (2nd ed.). North-holland Publishing Company.

[39] Maged M Michael and Michael L Scott. 1995. *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*. Technical Report. ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE.

[40] Achour Mostéfaoui, Sergio Rajsbaum, Michel Raynal, and Matthieu Roy. 2004. Condition-based consensus solvability: a hierarchy of conditions and efficient protocols. *Distributed Computing* 17, 1 (2004), 1–20.

[41] Aravind Natarajan, Lee H Savoie, and Neeraj Mittal. 2013. Concurrent wait-free red black trees. In *Symposium on Self-Stabilizing Systems*. Springer, 45–60.

[42] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. *ACM SIGPLAN Notices* 46, 6 (2011), 164–174.

[43] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and verifying probabilistic assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 112–122.

[44] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2012. Wait-free linked-lists. In *International Conference On Principles Of Distributed Systems*. Springer, 330–344.

[45] Avi Timor, Avi Mendelson, Yitzhak Birk, and Neeraj Suri. 2008. Using underutilized CPU resources to enhance its reliability. *IEEE Transactions on Dependable and Secure Computing* 7, 1 (2008), 94–109.

[46] Sungseob Whang, Tymani Rachford, Dimitra Papagiannopoulou, Tali Moreshet, and R Iris Bahar. 2017. Evaluating critical bits in arithmetic operations due to timing violations. In *High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA USA, 1–7.

[47] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. 2015. Approximate computing: A survey. *IEEE Design & Test* 33, 1 (2015), 8–22.