Resilience of Mutual Exclusion Algorithms to Transient Memory Faults

Thomas Moscibroda
Microsoft Research
Redmond, WA
moscitho@microsoft.com

Rotem Oshman
Computer Science and AI Laboratory, MIT
Cambridge, MA
rotem@mit.edu

ABSTRACT

We study the behavior of mutual exclusion algorithms in the presence of unreliable shared memory subject to transient memory faults. It is well-known that classical 2-process mutual exclusion algorithms, such as Dekker and Peterson's algorithms, are not fault-tolerant; in this paper we ask what degree of fault tolerance can be achieved using the same restricted resources as Dekker and Peterson's algorithms, namely, three binary read/write registers.

We show that if one memory fault can occur, it is not possible to guarantee both mutual exclusion and deadlock-freedom using three binary registers; this holds in general when fewer than $2f+1$ binary registers are used and f may be faulty. Hence we focus on algorithms that guarantee (a) mutual exclusion and starvation-freedom in fault-free executions, and (b) only mutual exclusion in faulty executions. We show that using only three binary registers it is possible to design an 2-process mutual exclusion algorithm which tolerates a single memory fault in this manner. Further, by replacing one read/write register with a test&set register, we can guarantee mutual exclusion in executions where one variable experiences unboundedly many faults.

In the more general setting where up to f registers may be faulty, we show that it is not possible to guarantee mutual exclusion using $2f+1$ binary read/write registers if each faulty register can exhibit unboundedly many faults. On the positive side, we show that an n-variable single-fault tolerant algorithm satisfying certain conditions can be transformed into an $((n-1)f+1)$-variable f-fault tolerant algorithm with the same progress guarantee as the original. In combination with our three-variable algorithm, this implies that there is a $(2f+1)$-variable mutual exclusion algorithm tolerating a single fault in up to f variables without violating mutual exclusion.

Categories and Subject Descriptors:
D.4.1 [Operating Systems]: Process Management–*mutual exclusion*
D.4.5 [Operating Systems]: Reliability–*fault tolerance*

General Terms: Algorithms, Theory

Keywords: mutual exclusion, fault tolerance, transient memory faults

1. INTRODUCTION

Mutual exclusion is among the most important and well-studied problems in distributed computing. It is used in concurrent programming to avoid the simultaneous use of shared data structures by pieces of computer code called critical sections. In a shared memory environment, synchronization among processes trying to access a critical section is achieved via a small set of shared variables that can be accessed by the processes. Existing mutual exclusion algorithms are based on the underlying assumption that these shared variables are reliable: if a process sets a shared variable to a certain value x, any subsequent read access to the variable will return x, until some other process overwrites the value.

In this paper we study the implications of relaxing this assumption, and consider mutual exclusion algorithms in the presence of unreliable shared memory. Our motivation for this relaxation is the observation that due to faster clock rates, increasing on-chip transistor density, decreasing voltages and smaller hardware feature sizes, the likelihood of encountering *transient memory faults* is non-negligible in today's computer systems, and is bound to rapidly increase in future systems. A transient memory fault, also known as a *soft error*, is a temporary hardware failure that alters a signal transfer, a register value, or some other processor component. Transient faults can occur due to many reasons; there are several recent examples where they have caused substantial reliability problems, leading to costly failures in industrial high-end systems.

In the context of mutual exclusion algorithms, the possibility of sudden changes to shared memory variables is particularly problematic, since it could result in a violation of the mutual exclusion property. Indeed, none of the well-known existing mutual exclusion algorithms (e.g., Dekker's algorithm, Peterson's algorithm, or Lamport's Bakery algorithm) is designed to be *resilient* to transient faults. Each of these algorithms may fail to maintain mutual exclusion if a shared variable used for communication among the processes suddenly changes. In fact, this holds even when processes always execute the entry and exit sections of the mutual exclusion algorithms all by themselves (that is, no other process can take steps when some process is in the entry or exit section).

Motivated by these observations, this paper investigates the extent to which 2-process mutual exclusion algorithms can withstand transient memory faults. The paper is divided into three parts. In the first part (Section 4) we give a basic characterization of fault-resilient 2-process mutual exclusion algorithms. One basic observation is that any f-fault-resilient 2-process mutual exclusion algorithm must satisfy the following structural property: when a process p_i executes the critical section by itself while the other process is in the remainder, p_i must change $f+1$ shared variables before it enters the critical section. We use this observation to show that any algorithm that uses $2f + 1$ binary read/write registers must exhibit either deadlock or mutual exclusion violation in f-fault executions.

In the second part of the paper (Sections 5–7), we show that a certain level of fault-resilience to transient faults comes "for free". We present a new starvation-free algorithm that, like Dekker's or Peterson's algorithms, uses three binary read/write shared variables; unlike Dekker and Peterson's algorithm, our algorithm guarantees mutual exclusion even in the presence of a single memory fault. The algorithm only guarantees progress in fault-free executions; it may deadlock in executions where memory faults occur. However, given the above impossibility result, this is in some sense the best one can do. Given the choice between guaranteeing mutual exclusion or guaranteeing deadlock-freedom in faulty executions, we choose the former in this paper. This seems to be the more natural choice in the context of mutual exclusion algorithms, and in many systems, deadlocks are arguably easier to detect and break, and their consequences less severe than mutual exclusion violations.

In fact, this is the best we can do in more than one sense. In Section 6 we prove a lower bound showing that $2f + 1$ binary variables are not sufficient to guarantee mutual exclusion when f of the variables can experience unboundedly many faults. Translated to the 3 variable case, this implies that no algorithm that uses 3 binary read/write registers can tolerate a single "Byzantine variable" which can flip unboundedly many times.

Given this gap, it is natural to ask if there is some relaxation of the model that would allow us to achieve unbounded fault-resilience. In Section 7, we give an answer to this question, by presenting a mutual exclusion algorithm which uses test&set register instead of one of the read/write registers, and is able to withstand unbounded faults to one variable. Both this and the above algorithm are non-trivial, and their structure is quite different from that of existing mutual exclusion algorithms.

One reason we are interested in understanding the possibilities and limitations of fault-resilience in the 3-variable, 1-fault setting is that these results have implications for the ratio of faulty variables that can be tolerated in general. It is reasonable to expect that the number of faults will increase with the amount of memory used, and hence this ratio is interesting to study. In the third part of the paper (Section 8), we show that our results for the 3-variable case imply more general results for m-variable algorithms tolerating f faults. We show that using $(n - 1)f + 1$ variables, of which f can be faulty, one can simulate a "well-behaved" n-variable algorithm that tolerates one fault. "Well-behaved" here means that the algorithm contains no data races, and that if one process attempts to read from a variable, the other process eventually stops writing to it. This property, which is satisfied by many existing mutual exclusion algorithms (including Dekker's algorithm and the algorithms we present in this paper), allows us to use a simple and lightweight simulation of n variables with one faulty variable from $(n-1)f+1$ variables of which f can be faulty. In conjunction with the 3-variable algorithm from Section 5, this implies the existence of a mutual exclusion algorithm using $2f + 1$ variables and tolerating f faulty variables, each of which can flip once. Moreover, the same simulation can be used to transform Dekker's algorithm into a $3(f + 1)$-variable algorithm tolerating f "Byzantine" faulty variables, which can each flip unboundedly many times.

Due to lack of space, the full proofs for some of the claims in the paper are omitted here. The algorithms presented in Sections 5 and 7 were model-checked using the NuSMV2 finite-state model checker (in addition to a manual proof of correctness), to verify both starvation-freedom in fault-free executions and mutual exclusion in faulty executions.

2. BACKGROUND & RELATED WORK

Transient faults. Transient faults (or "soft errors") can occur in different parts of the hardware stack in a computer system, and arise for various reasons, such as energetic particles that strike a transistor and cause it to change its state. In memory, for instance, alpha particles emitted by traces of radioactive elements present in the packaging materials of the device can penetrate the die and generate a high density of holes and electrons in its substrate, thereby creating an imbalance in the electrical potential distribution and causing stored data to be corrupted. A single alpha particle that possesses enough energy can cause a soft error all by itself. Transient faults are usually random and non-recurring, and their rate of occurrence depends on circuit sensitivity and the alpha flux emitted by the device. Such faults have led to costly failures in high-end systems in recent years. For example, they are known to have caused crashes at Sun's major customer sites including America Online and eBay [4], and HP's Los Alamos Labs supercomputers [21].

Unfortunately, while transient errors already cause substantial reliability problems, current trends in hardware design suggest that fault rates will further increase in the future. Faster clock rates, increasing transistor density, decreasing voltages and smaller feature sizes all contribute to increasing fault rates, e.g. [3, 22]. In fact, fault rates in modern processors have been increasing at a rate of approximately 8% per generation [5]. To counter soft errors, computer architects and compiler researchers have proposed various solutions, which usually involve adding redundancy to computations in one way or another. For instance, there are proposals involving hardware-only solutions such as error-correcting codes, watchdog co-processors or redundant hardware threads (e.g. [16, 17]) as well as software-only techniques that use both single and multiple cores (e.g. [20, 18]). These solutions are typically "heavy-weight" and quite costly in terms of memory and performance.

Resilient algorithms. In the area of algorithms, designing resilient algorithms for unreliable memories has also attracted interest. Problems such as fault-resilient selection, sorting, and matrix computations in various failure models have attracted a lot of interest in recent years (see [10] for a survey). Faulty memory has also been studied in multiprocessors. There is significant research in the parallel computing literature devoted to deliver general simulation mechanisms of fully operational parallel machines on their faulty counterparts, e.g. [7, 8].

Fault-tolerant simulations. In the shared memory distributed computing literature, the problem of implementing fault-tolerant registers (and other objects) from faulty objects under various fault models was studied, e.g. in [1, 2, 13]. With regard to our simulation in Section 8, the most relevant results are the ones given in [1] and [13] on implementing various read/write registers from faulty registers in the arbitrary, responsive failure mode.[1] For example, in combination with earlier work [19, 24], it is shown that one *safe* read/write register can be implemented from $2f+1$ safe faulty registers, and one *atomic* read/write register using $6f+3$ ($8f+4$) safe registers and $24f+12$ ($16f+8$) safe binary registers, respectively, if the f faulty registers can have infinitely many faults. A relevant result from [1] shows that one reliable atomic register can be implemented from $20f+8$ atomic registers if at most f are faulty. In our context, however, the simulation in Section 8 serves a different purpose; we do not seek to mask faults completely, as the high-level

[1]Much better results are known for more benign failure modes, e.g. [13, 11]

mutual exclusion algorithm that uses the objects can tolerate some degree of faulty behavior. Instead, we seek to reduce f faults to a single fault, which can then be handled by the algorithm. Together with the fact that we make assumptions about the behavior of the algorithm and do not require liveness in faulty executions, this allows us to get away with a very lightweight simulation, where $2f + 1$ low-level registers simulate three high-level registers of which at most one is faulty.

Fault-tolerant mutual exclusion. The issue of fault-tolerance in mutual exclusion algorithms was one of the principal themes of Lamport's paper on non-atomic algorithms [14]. Several failure models are considered. Among many other malfunctions, one failure type studied are transient faults, which allows arbitrary changes to the shared memory (and local) variables of the algorithm. A mutual exclusion algorithm tolerating all these types of failures was presented in [25], but it required 17 binary shared variables. This was subsequently improved to 8 binary variables for 2-process mutual exclusion in [23]. These algorithms require more shared variables than the algorithms we present here, but they do not deadlock in faulty executions.

3. MODEL & DEFINITIONS

Mutual exclusion algorithms. We represent a 2-process mutual exclusion algorithm as follows. Let PC_0, PC_1 be the control locations (code lines) for processes 0 and 1 respectively, and let Var be the set of shared variables (in the current paper we assume that the shared variables are binary). We assume that PC_0, PC_1 each include two distinguished locations N, C, representing the remainder and the critical section, respectively. [2]

A *configuration* of the algorithm is a triplet $(\ell_0, \ell_1, \bar{v})$, where $\ell_0 \in PC_0$ and $\ell_1 \in PC_1$ are the control locations of p_0 and p_1 respectively, and $\bar{v} \in 2^{Var}$ represents the state of the shared variables Var. A *step* of the algorithm is a transition from one global configuration to another, in which some process p_i executes either a $\mathsf{read}(x)$ or a $\mathsf{write}(x, v)$ operation on some shared variable x, and transitions to a new control location. If the control location of a process is N or C, it can also take null-transitions, in which its location and the values in shared memory do not change.

An *execution* of the algorithm is a sequence $\sigma_0\sigma_1\ldots$ of configurations, starting from the initial configuration σ_0, in which each configuration is obtained from the previous configuration by either a step of p_0 or p_1, or by a *memory fault*, in which the value of some shared variable $x \in Var$ changes from 0 to 1 or vice-versa. In an (f, c)*-fault execution*, at most f shared variables experience at most c memory faults each; in a *fault-free execution* there are no memory faults. We are concerned only with *admissible* executions, in which both processes take infinitely many steps. (This includes idle steps in which a process that is currently in the remainder stays in the remainder.)

The algorithms we present in this paper are *starvation-free*: for each process p_i, if p_i begins executing the entry section, then p_i eventually enters the critical section. For our lower bounds we typically assume *deadlock-freedom*, a weaker progress condition which asserts that if some process p_i is in the entry section, then eventually some process (either p_i or p_{1-i}) enters the critical section.

[2]For convenience we assume that the algorithm is memoryless, and each process has a single control location that it returns to whenever it goes into the remainder. However, this assumption is not necessary for our lower bounds.

Fault-tolerant mutual exclusion. In the current paper we are concerned with algorithms that guarantee mutual exclusion in the face of memory faults. We say that an algorithm is (f, c)*-resilient* if it guarantees mutual exclusion in (f, c)-fault executions, and deadlock-freedom (or starvation-freedom) in admissible fault-free executions. In the remainder of the paper, when we refer to deadlock- or starvation-freedom, these are restricted to fault-free executions (unless otherwise stated).

Notation and terminology. A *schedule* is a finite sequence $\alpha \in (\{p_0, p_1\} \cup \{\mathsf{flip}(x) \mid x \in Var\})^*$ of process identifiers, interspersed with memory faults $\mathsf{flip}(x)$ in which a variable x changes its value. A schedule is p_i*-only* if it does not contain any steps of p_{1-i}. We use $exec(\sigma, \alpha)$ to denote the execution fragment obtained by letting the system take the steps in α starting from configuration σ, and we use $config(\sigma, \alpha)$ to denote the final configuration reached in $exec(\sigma, \alpha)$.

A common lower bound technique is to maneuver the system into a configuration σ where the next step of some process p_i is to write to a register x, obliterating whatever value was stored there previously. In this case we say that p_i *covers* x in σ.

A configuration $\sigma = (\ell_0, \ell_1, \bar{v})$ is *indistinguishable to* p_i from $\sigma' = (\ell'_0, \ell'_1, \bar{v}')$, denoted $\sigma \sim_{p_i} \sigma'$, if $\ell_i = \ell'_i$ and $\bar{v} = \bar{v}'$. It can be shown by induction on the length of the schedule that if $\sigma \sim_{p_i} \sigma'$, then for any p_i-only schedule α we also have $config(\sigma, \alpha) \sim_{p_i} config(\sigma', \alpha)$.

4. BASIC IMPOSSIBILITY RESULTS

In this section, we derive a set of results that characterize the resilience of mutual exclusion algorithms to a single memory fault. These results have implications throughout the remainder of the paper. We begin by observing that any $(1, 1)$-resilient algorithm must have the following property.

DEFINITION 4.1 (HAMMING DISTANCE 2 PROPERTY, **HD2**). *Suppose that $\sigma = (\ell_0, \ell_1, \bar{v})$ and $\sigma' = (\ell'_0, \ell'_1, \bar{v}')$ are reachable configurations such that σ is an idle configuration ($\ell_0 = \ell_1 = N$) and for some $i \in \{0, 1\}$, $\ell'_i = C$ and $\ell'_{1-i} = N$. Then the Hamming distance between $\bar{v}$ and $\bar{v}'$ must be at least 2.*

Algorithms that do not have the **HD2** property can violate mutual exclusion when a single memory fault occurs: if σ and σ' are configurations as in the definition above, whose Hamming distance is smaller than 2, then we can flip a single bit in σ' to obtain a configuration τ that is indistinguishable to p_{1-i} from σ. Since σ is idle, when we let p_{1-i} run by itself from τ (which p_{1-i} cannot distinguish from σ) it must eventually enter the critical section, violating mutual exclusion.

If there are only two shared variables, then in order to satisfy the **HD2** property each process must modify both variables when it executes its entry section by itself; it can be shown that no algorithm can accomplish this.

THEOREM 4.1. *No deadlock-free mutual exclusion algorithm that uses two binary variables can satisfy the **HD2** property.*

This result is similar in spirit to the lower bound of [6], which shows that n shared variables are necessary for n-process mutual exclusion; in other words, each process must have a variable that it "owns" in some sense. Technically, however, the proof of Theorem 4.1 shares very little with the lower bound of [6], because in our case the number of shared variables does match the number of processes. The proof is quite similar to the proof of Theorem 6.1 in Section 6, and it is omitted here. In general, no f-binary variable mutex algorithm can satisfy the **HD-*f*** property (the proof is again similar to that of Theorem 6.1).

It follows from Theorem 4.1 that two binary variables cannot be used to guarantee $(1, 1)$-resilience, even if only deadlock-freedom is required, and even in executions where the steps of the two processes are never interleaved while one of the two is in the entry or exit section. In Section 5 we show that three binary variables suffice to guarantee $(1, 1)$-resilience and starvation-freedom in fault-free executions.

Impossibility of achieving both safety and liveness. Our definition of *resilience* focuses on algorithms that guarantee mutual exclusion, but sacrifice liveness in faulty executions; one might ask whether it is possible to guarantee mutual exclusion *and* deadlock-freedom or even starvation-freedom. Unfortunately, for the case of 3 variables and one fault, the answer is negative. The following theorem shows that in general, when fewer than $2f + 1$ registers are used, liveness in faulty executions comes at the cost of violating mutual exclusion. If starvation-freedom is desired in fault-free executions, then $2f + 1$ registers are also insufficient. This result motivates our definition of resilience. Unlike the other negative results in this paper, the following theorem is not restricted to binary registers, if one assumes that in the multi-valued case a faulty register's value can flip to any other value.

THEOREM 4.2. *Let $\mathcal{A}$ be an m-variable deadlock-free mutual exclusion algorithm. If $m \leq 2f$, or if $m \leq 2f + 1$ and $\mathcal{A}$ is also starvation-free, then $\mathcal{A}$ fails to satisfy either deadlock-freedom or mutual exclusion in some $(f, 1)$-fault execution where process steps are never interleaved while some process is in the entry or exit section.*

PROOF. Consider an execution fragment in which starting from the initial configuration σ_0, we let p_0 run solo until it enters the critical section. Let σ_C be the resulting configuration. If $m \leq 2f$, then for any two states $\bar{v}, \bar{v}' \in \{0, 1\}^m$ of the shared memory, there is a third state $\bar{u}$ whose Hamming distance from both $\bar{v}$ and $\bar{v}'$ is at most f. Thus, we can flip no more than f registers from σ_C, and obtain a configuration τ whose Hamming distance from both σ_0 and σ_C is no more than f (see Fig. 1).

Because the Hamming distance of the shared memory in τ from that in σ_0 is no more than f, p_1 cannot distinguish τ from a configuration τ' obtained from σ_0 by flipping no more than f variables. In τ' both processes are idle, so if the algorithm satisfies deadlock-freedom in $(f, 1)$-fault executions where processes are not interleaved in the entry and exit sections, when we let p_1 run by itself from τ' it will eventually enter the critical section. But $\tau \sim_{p_1} \tau'$, so the same is true for τ, and mutual exclusion is violated.

Next, suppose that the algorithm guarantees starvation-freedom in fault-free executions, and $m \leq 2f + 1$. Let σ be a reachable idle configuration such that when p_0 runs by itself from σ, eventually σ occurs again. Then there must exist some register x that p_0 does not write to in its solo run from σ: if there is no such register, then we can let p_1 begin the entry section as well, but each time p_1 covers some register y, we let p_0 run until it covers y as well. Then we let p_1 write to y, followed immediately by p_0. All evidence that p_1 is in the entry section is erased from the shared memory, so p_0 cannot distinguish this execution from the one in which it runs solo from σ. We can continue this way to construct an infinite admissible execution in which p_1 remains in the entry section forever. Thus there must be some register to which p_0 does not write.

Since $m \leq 2f + 1$, p_0 writes to at most $2f$ registers when it runs solo from σ. We can repeat the argument we used for $m \leq 2f$ to show again that either deadlock-freedom or mutual exclusion must

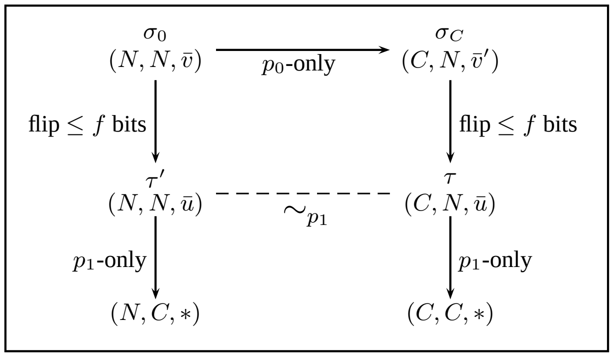

$$\begin{tikzcd}
\overset{\sigma_0}{(N,N,\bar{v})} \arrow[r, "p_0\text{-only}"'] \arrow[d, "\text{flip} \leq f \text{ bits}"'] & \overset{\sigma_C}{(C,N,\bar{v}')} \arrow[d, "\text{flip} \leq f \text{ bits}"] \\
\overset{\tau'}{(N,N,\bar{u})} \arrow[r, dashed, no head, "\sim_{p_1}"'] \arrow[d, "p_1\text{-only}"'] & \overset{\tau}{(C,N,\bar{u})} \arrow[d, "p_1\text{-only}"] \\
(N,C,*) & (C,C,*)
\end{tikzcd}$$

Figure 1: Illustration for the proof of Thm. 4.2

be violated in some $(f, 1)$-fault execution where process steps are not interleaved when a process is in the entry or exit section. $\square$

5. A (1,1)-RESILIENT THREE-VARIABLE ALGORITHM

In this section we give a starvation-free mutex algorithm, Algorithm HANDSHAKE, that uses three binary read/write registers and guarantees mutual exclusion in the face of one memory fault. The algorithm satisfies the Hamming Distance 2 property: when a process executes the entry section solo, it sets two of the shared variables to 1.

As in the Peterson and Dekker algorithms, two of the shared variables, c_0 and c_1, serve as flags indicating whether p_0 and p_1 are active. However, the function of the third variable is different. The Peterson and Dekker algorithms achieve starvation-freedom by using the third shared variable as a "turn variable", but the Hamming Distance 2 property precludes this strategy; the third variable must now be used more like a *lock*: processes set it to 1 when they enter the critical section by themselves and reset it to 0 when they leave. Thus, if process p_i executes its entry section by itself, before it enters the ciritical section it sets both its flag c_i and the third variable $lock$ to 1, protecting itself by two bits in case of a single memory fault. We use a different mechanism to guarantee fairness (see below).

One major difficulty a $(1, 1)$-resilient algorithm must face is the following. Suppose that the two processes begin executing their entry section in lockstep, until the first time they write to the shared memory. Assume they write to different variables (as eventually they must); now the state of the shared memory is 110. Both processes are in the entry section, but neither process can distinguish this configuration from the one in which the other process is in the critical section and the third variable ($lock$) has flipped to 0. Therefore neither process can enter the critical section until it has verified that the other process is not in the critical section, by interacting with the other process in a sequence of reads and writes that we call a *handshake*. The handshake is designed so that even if a memory fault occurs, a process can never reach the end of the handshake if the other process is in the critical section. This is achieved by having each process p_i go through a sequence of writes to shared memory, leaving a unique footprint in shared memory that is never encountered elsewhere in the algorithm; in particular, it cannot be "faked" by the adversary using a memory fault, or by having p_i go in and out of the critical section.

The handshake comprises lines 008–010 for p_0 and lines 104–106 for p_1. At the end of the handshake, p_0 enters the critical

Algorithm HANDSHAKE: code for process 0

```
001 $c_0 := 1$
002 wait until $lock = 0$
003 while $c_1 = 1$ do
004     if $c_0 = 0$ then goto 009
005 $lock := 1$
006 if $c_1 = 1$ then
007     $lock := 0$
008     wait until $c_0 = 0$
009     $c_0 := 1$
010     wait until $c_1 = 0$
        (enter critical section)
011     $c_0 := 0$
012     $c_1 := 1$
013     $lock := 1$
014 else
015     if $c_0 = 0$ then // A fault occurred
016         $lock := 0$
017         goto 002
        (enter critical section)
018     $c_0 := 0$
019     $lock := 0$
```

Algorithm HANDSHAKE: code for process 1

```
101 $c_1 := 1$
102 wait until $lock = 0$
103 if $c_0 = 1$ then
104     $c_0 := 0$
105     wait until $c_0 = 1$
106     $c_1 := 0$
107     wait until $c_1 = 1$
108     wait until $lock = 1$
109 else
110     $lock := 1$
111     if $c_0 = 1$ then
112         $lock := 0$
113         goto 104
114     if $c_1 = 0$ then // A fault occurred
115         $lock := 0$
116         goto 102
    (enter critical section)
117 $lock := 0$
118 $c_1 := 0$
```

section, and p_1 waits in line 107 for a signal from p_0. When p_0 exits the critical section, it hands the critical section over to p_1 by setting both c_1 and $lock$ to 1. Notice that (a) p_1 must observe both c_1 and $lock$ change to 1 in order to enter the critical section, so that a single memory fault cannot cause it to enter; and (b) we achieve starvation-freedom, because whenever p_0 and p_1 contend in the entry section, eventually both processes enter the critical section.

The overall strategy for both processes is as follows:

(1) Set the flag, c_i (lines 001 and 101).

(2) Check if the other process is present (lines 003 and 103), and if so, try to engage in a handshake with it.

(3) If the other process is not around, set $lock$ (lines 005, 110).

(4) Check again if the other process is present (lines 006, 111) if so, release $lock$ and engage in a handshake.

(5) If the other process is still not around, enter the critical section.

The reason we require (4) is that if p_{1-i} begins the entry section when p_i is executing (2) or (3), p_{1-i} may see either $lock = 0$ or $lock = 1$ when it executes its second line (line 002 or 102), depending on the specific interleaving of process steps. If $lock = 1$, then p_{1-i} becomes stuck in the second line; if $lock = 0$ then p_{1-i} falls through the second line and attempts to participate in a handshake. Significantly, p_i cannot tell what p_{1-i} saw when it checked $lock$. Thus, to make sure that p_{1-i} does not get stuck waiting for a handshake that is never reciprocated, p_i releases $lock$, allowing p_{1-i} to fall through the second line (if it has not done so already). Then both processes engage in a handshake.

There are a few subtleties beyond this basic pattern. First, note that p_0 does not necessarily engage in a handshake if it sees $c_1 = 1$ in line 003 (it can fall through to line 005), but p_1 always executes a handshake if it sees $c_0 = 1$ in line 103. This ties in to the different order of writes in the processes' exit sections, lines 018–019 and 117–118: when p_0 exits, it releases first its flag and then $lock$, whereas p_1 releases $lock$ first and then c_1.

Informally, we want p_0 to release $lock$ last to make sure that p_1 cannot get past line 102 until p_0 has finished the exit section, otherwise the sequence "observe $c_0 = 1$, set $c_0 := 0$, observe $c_0 = 1$", which gets p_1 through lines 103–105, can also be created by p_0 being in the exit section (having already released $lock$) and later beginning the entry section again and setting $c_0 := 1$. That creates mis-coordination from which the algorithm cannot recover. On the other hand, if p_1 were to set $lock$ to 0 after it sets c_1 to 0, then we would have a dangerous situation in which p_1 has already set c_1 to 0, erasing this evidence of its presence, and is covering $lock$, about to write 0. If c_0 experiences a memory fault and flips to 0, this situation can arise when p_0 is in the entry section, has already set $lock$, and believes that it is protected by both $c_0 = 1$ and $lock = 1$. But when we let p_1 take its next step, it writes 0 to $lock$, erasing all evidence of p_0's presence and freeing p_1 to enter the critical section even though p_0 is already critical. Consequently we must ensure that whenever p_1 is about to set $lock$ to 0, we allow p_0 to see that this may be the case by having $c_1 = 1$.

As a consequence of the different write order, when p_0 sees $c_1 = 1$ in line 003, there are two cases: either p_1 is in the entry section (but has either not set $lock$ yet, or has set $lock$ and later released it), or p_1 is in the exit section, about to execute line 118. Thus, p_0 waits to see what p_1 does: if p_1 sets c_0 to 0 then it is in the entry section and wants to execute a handshake, and if p_1 sets c_1 to 0 then it is in the exit section. In this last case p_0 continues as though it never saw $c_1 = 1$ when it executed line 003. As for p_1, because it cannot get past line 102 (where it waits to see $lock = 0$) until p_0 has finished the exit section and gone into the remainder, if p_1 sees $c_0 = 1$ in line 103 then there is only one possibility: p_0 is in the entry section and will engage in a handshake.

Finally, when the processes believe they are about to enter the critical section uncontended (line 015 for p_0 and line 114 for p_1), they perform one final test, which is to check that their own flag c_i has not flipped after they set it in the first line of the entry section. If the test succeeds, it guarantees that the process has managed to secure both c_i and $lock$ before the other process started its entry section, so that if one of the two variables were to flip, the other variable would remain non-faulty and block the other process from entering the critical section. If the test fails, then a memory fault has occurred; the process releases $lock$ and starts the entry section from the beginning.

THEOREM 5.1. *There is a two-process $(1,1)$-resilient mutual exclusion algorithm that uses three binary read/write registers, and guarantees starvation-freedom in fault-free executions.*

The correctness proof of algorithm HANDSHAKE is quite tedious, and we do not include it here. In addition to the manual proof, the NuSMV2 model-checker was used to verify that the algorithm is starvation-free and $(1,1)$-resilient.

6. IMPOSSIBILITY OF (f,∞)-RESILIENCE WITH $2f+1$ REGISTERS

We have shown that using three binary registers it is possible to achieve $(1,1)$-resilience; next we show that it is not possible to guarantee $(1,\infty)$-resilience using three variables. More generally, we show that no algorithm using $2f+1$ binary registers can be (f,∞)-resilient.

We begin by giving a characterization of f-resilience that is similar to the **HD2** property defined in Section 4; we show that in a $(f,1)$-resilient algorithm, each process must use $f+1$ variables to protect itself whenever it enters the critical section. These variables must be written when the process enters the critical section by itself, and restored to their initial value when the process exits the critical section by itself.

DEFINITION 6.1 (FLAG REGISTERS). *We say that register x is a* flag register for p_i in σ *if*

(a) σ is an idle configuration,

(b) When p_i runs by itself from σ in a fault-free execution it eventually writes both 0 and 1 to x, and

(c) When p_i runs by itself from σ in a fault-free execution, the system eventually returns to configuration σ.

As with the HD2 property we saw in Section 4, in any algorithm that tolerates f faulty variables (even restricted to a single fault in each variable), each process that enters the critical section by itself must protect itself by $f+1$ bits.

LEMMA 6.1. *In any $(f,1)$-resilient algorithm, for each process p_i there is an idle configuration σ that is reachable in a fault-free execution, such that p_i has at least $f+1$ flag registers in σ.*

PROOF. From any idle configuration σ, if we let p_i run by itself until it enters the critical section, it must change the values of at least $f+1$ shared registers; otherwise, once p_i enters the critical section, we could flip the values of all shared registers that p_i modifies back to their values in σ, and obtain a configuration σ'_C that is indistinguishable to p_{1-i} from σ. If we let p_{1-i} run by itself from σ'_C, it eventually enters the critical section as well, violating mutual exclusion.

Since there are only finitely many configurations, there exists an idle configuration σ, reachable by a fault-free execution fragment, such that if we let p_i run by itself in a fault-free execution from σ then eventually configuration σ occurs again. But p_i changes the values of at least $f+1$ registers from their values in σ on its way into the critical section, and when we return to σ these registers have returned to their values in σ. Therefore p_i must write both 0 and 1 to each of these $f+1$ registers at some point in its solo run from σ before σ occurs again. $\square$

THEOREM 6.1. *No 2-process mutex algorithm using $2f+1$ binary read/write registers is (f,∞)-resilient.*

PROOF. Suppose for the sake of contradiction that an (f,∞)-resilient algorithm that uses at most $2f+1$ registers does exist. Let σ be the idle configuration whose existence is guaranteed by Lemma 6.1, such that p_1 has at least $f+1$ flag registers in σ. Let Y denote the set of p_1's flag registers in σ.

If we let p_0 run solo from σ it must eventually enter the critical section. Let $\ell_0\ell_1\ldots\ell_m \in PC_0^*$, where $\ell_0 = N$ and $\ell_m = C$, be the sequence of control locations that p_0 passes through on its way into the critical section in a solo run from σ. We show by induction on k that for all $0 \le k \le m$, there is a configuration σ_k such that

(a) In σ_k we have $pc_0 = \ell_k$,

(b) σ_k is reachable from σ in an execution fragment where all the register in Y are non-faulty, and

(c) $\sigma_k \sim_{p_1} \sigma$.

In other words, we can "sneak p_0 into the critical section" step by step, without p_1 being able to distinguish any step from the idle configuration σ. The contradiction follows immediately.

The base of the induction is $k=0$, for which the claim holds trivially. For the step, suppose that we have already shown that there is a reachable configuration σ_k in which $pc_0 = \ell_k$ and such that $\sigma_k \sim_{p_1} \sigma$. Consider the step that p_0 takes to reach location ℓ_{k+1} from location ℓ_k. We will show that this step can be simulated by an execution fragment from σ_k in which we do not corrupt any register in Y (see Fig. 2 for an illustration).

There are two types of steps that p_0 can take. The first is a write; this operation has no return value, and control always passes to ℓ_{k+1} (i.e., the code does not branch at ℓ_k). In this case we must ensure that p_1 does not observe p_0's write. On the other hand, p_0 can execute a read operation, and then branch on the result. In this case we must ensure that the value p_0 reads is the "right" value, the one that will cause it to reach ℓ_{k+1}.

Let us first handle the cases where p_0 accesses a variable $x \notin Y$, which we are allowed to corrupt. If the step is a $\mathsf{read}(x)$ step which is expected to return v, then we simulate the step by flipping x to v (if its value is not already v), letting p_0 take its read step, and flipping x back to its previous value. Similarly, if the step is a $\mathsf{write}(x,v)$, then we let p_0 take the step, and then flip x back to its previous value before the write. Let σ_{k+1} be the resulting configuration. In both cases the values in shared memory are the same in σ_k and in σ_{k+1}, and p_1 does not take any steps between σ_k and σ_{k+1}, so $\sigma_{k+1} \sim_{p_1} \sigma_k \sim \sigma$.

Now suppose that some variable $y \in Y$ is accessed. If the step is a $\mathsf{read}(y)$ step expected to return v, then we simulate the step as follows. Because y is a flag register of p_1 in σ, we know that there is some p_1-only schedule α such that the last step in $exec(\sigma,\alpha)$ is $\mathsf{write}(y,v)$, and there is another p_1-only schedule β such that $config(\sigma,\alpha\beta) = \sigma$ (that is, β returns us to configuration σ). Because $\sigma_k \sim_{p_1} \sigma$, the last step in $exec(\sigma_k,\alpha)$ is also a $\mathsf{write}(y,v)$ by p_1. Now we let p_0 take its $\mathsf{read}(y)$ step, which returns v, and does not change the shared memory. Because $config(\sigma_k,\alpha) \sim_{p_1} config(\sigma_k,\alpha p_0)$, if we append β to the schedule αp_0, we obtain a configuration σ_{k+1} in which $pc_0 = \ell_{k+1}$, such that $\sigma_{k+1} \sim_{p_1} \sigma$.

Finally, if the step is a $\mathsf{write}(y,v)$ such that $y \in Y$, then we proceed as follows. Since y is a flag register of p_1 in σ, there is a p_1-only schedule α such that in $config(\sigma,\alpha)$, p_1 covers y. Since $\sigma_k \sim_{p_1} \sigma$, the same holds for $config(\sigma_k,\alpha)$. Thus, from σ_k, we let p_1 run until it covers y; then we let p_0 take its $\mathsf{write}(y,v)$ step, followed by p_1's step, which overwrites y. The resulting configuration σ'_k is indistinguishable to p_1 from $config(\sigma,\alpha p_1)$, so there is some p_1-only schedule β which returns p_1 to σ ($config(\sigma,\alpha p_1\beta) = \sigma$). We have $config(\sigma_k,\alpha p_0 p_1 \beta) \sim_{p_1} \sigma$, as required.

We have now shown that from σ, which is reachable in a fault-free configuration, there is an execution fragment in which no register in Y is corrupted, and mutual exclusion is violated. Since $|Y| \geq f+1$, the number of faulty variables is at most $2f+1-(f+1) = f$. Hence the algorithm is not (f, ∞)-resilient. $\square$

We remark that the proof of Theorem 6.1 does not extend to registers that can take more than two values. The majority of the proof relies only on the fact that the algorithm must have the **HD-$(f+1)$** property, i.e., any process that enters the critical section uncontended must write to at least $f+1$ registers; this holds for multi-valued registers as well as for binary ones. The one part of the proof that fails is the case in the induction step where p_0 reads a variable $y \in Y$, which we cannot corrupt. In the proof we handle this case by maneuvering p_1 into writing the value that p_0 expects to read from y. In the multi-valued case we cannot do this, as there is no necessity for a process to write all possible values into its flag variables (it still must write at least two different values, but now there can exist values it does not write at any point). In fact, p_0 can detect p_1's presence by writing some unique value which is never written by p_1 into a flag register $y \in Y$. On its way into the critical section (and possibly back out), p_1 must then write some different value into y. If p_0 later checks y again, it can detect that it is not alone, because its value has been overwritten. Thus it is entirely conceivable that an (f, ∞)-resilient algorithm using $2f+1$ multi-valued registers does exist.

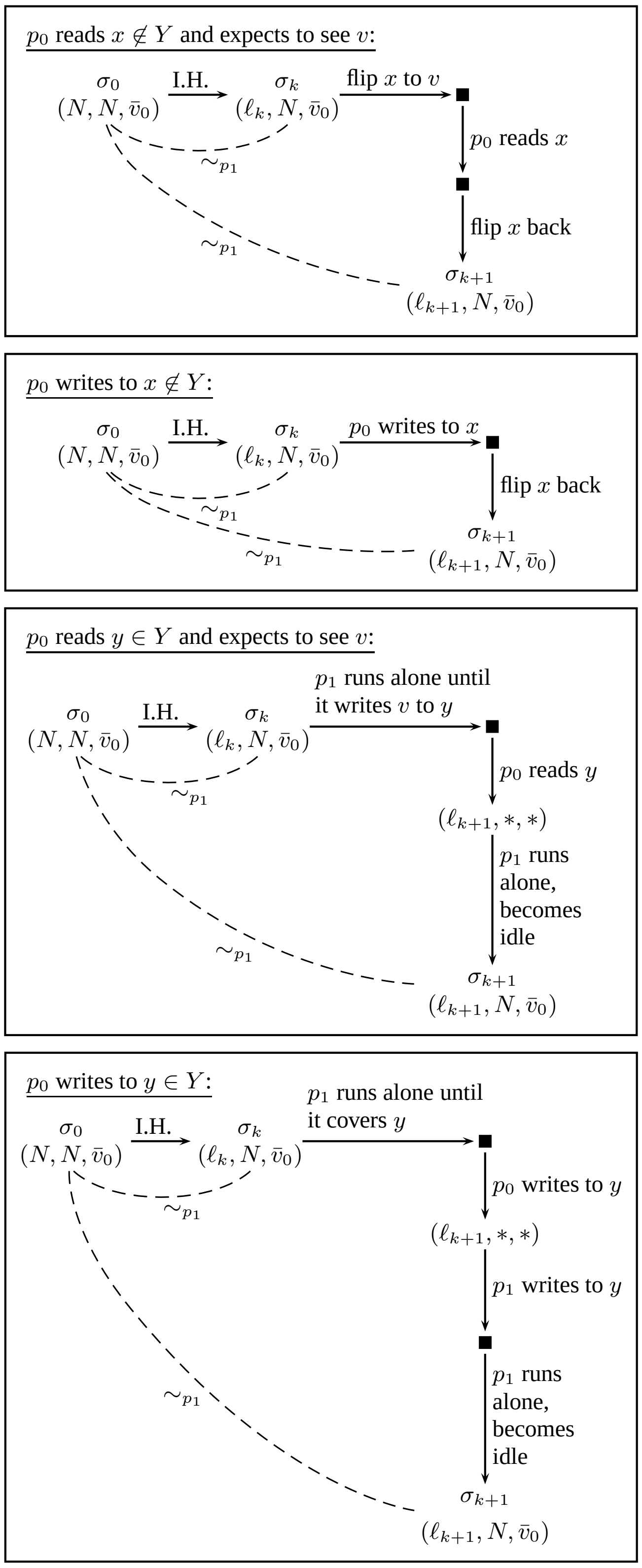
7. A $(1, \infty)$-RESILIENT THREE-VARIABLE ALGORITHM USING TEST&SET

As we saw above, there does not exist a $(1, \infty)$-resilient algorithm that uses three binary read/write registers. In particular, Algorithm HANDSHAKE also does not tolerate more than a single fault in any variable. To see why, consider an execution where

1. p_0 runs by itself until it enters the critical section.
2. p_1 begins the entry section, and becomes stuck in line 102.
3. The value of c_1 flips from 1 to 0.
4. p_0 exits the critical section.
5. p_1 progresses to line 105, where it waits for p_0 to begin its part of a handshake by setting c_0 to 1.
6. p_0 starts the entry section again, setting c_0 to 1 in line 001. This is mis-interpreted by p_1 as the start of a handshake.
7. p_0 continues to run until it enters the critical section (recall that c_1 has flipped to 0), and p_1 runs until it reaches the **wait** statement in line 107.
8. The value of c_1 flips from 0 to 1, freeing p_1 to enter the critical section and violate mutual exclusion.

We can resolve such error scenarios by (a) including as part of the handshake a change of *lock* from 1 to 0, rather than from 0 to 1 (a value of $lock = 1$ generally indicates that the other process is interested in the critical section, while $lock = 0$ indicates that it is not); and (b) replacing *lock* with a test&set register, as shown below. Instead of writing 1 to *lock*, the two processes always execute a test&set(*lock*); if the test&set fails, then a fault must have occurred, because in fault-free executions no process writes 1 to *lock* when *lock* is already 1. In this case the process returns to the beginning of its entry section and starts over. We call the new algorithm T&S-HANDSHAKE; the code is given below.

Originally in Algorithm HANDSHAKE, after a handshake, p_0 enters the critical section without setting *lock* to 1. Then, after the handshake, one of the signals that p_1 uses to decide it can enter the critical section is that it observes the value of *lock* change to 1. The reason we could get away with not setting *lock* is that when only one fault can occur, we are guaranteed that at the end of a hand-



p_0 reads $x \notin Y$ and expects to see v:
σ_0 $(N, N, \bar{v}_0)$
I.H.
σ_k $(\ell_k, N, \bar{v}_0)$
flip x to v
p_0 reads x
flip x back
$\sim_{p_1}$
$\sim_{p_1}$
σ_{k+1} $(\ell_{k+1}, N, \bar{v}_0)$
p_0 writes to $x \notin Y$:
σ_0 $(N, N, \bar{v}_0)$
I.H.
σ_k $(\ell_k, N, \bar{v}_0)$
p_0 writes to x
flip x back
$\sim_{p_1}$
$\sim_{p_1}$
σ_{k+1} $(\ell_{k+1}, N, \bar{v}_0)$
p_0 reads $y \in Y$ and expects to see v:
σ_0 $(N, N, \bar{v}_0)$
I.H.
σ_k $(\ell_k, N, \bar{v}_0)$
p_1 runs alone until it writes v to y
p_0 reads y
$(\ell_{k+1}, *, *)$
p_1 runs alone, becomes idle
$\sim_{p_1}$
$\sim_{p_1}$
σ_{k+1} $(\ell_{k+1}, N, \bar{v}_0)$
p_0 writes to $y \in Y$:
σ_0 $(N, N, \bar{v}_0)$
I.H.
σ_k $(\ell_k, N, \bar{v}_0)$
p_1 runs alone until it covers y
p_0 writes to y
$(\ell_{k+1}, *, *)$
p_1 writes to y
p_1 runs alone, becomes idle
$\sim_{p_1}$
$\sim_{p_1}$
σ_{k+1} $(\ell_{k+1}, N, \bar{v}_0)$

Figure 2: The induction step in the proof of Thm. 6.1. The figures illustrates the four possibilities for the step that p_0 takes to move from location ℓ_k in its code to location ℓ_{k+1}. Here, "I.H." stands for the execution fragment whose existence is guaranteed by the induction hypothesis.

shake p_1 is also in the entry section, at the end of its part of the handshake, waiting for a very specific signal from p_0.

With unbounded faults, this is no longer true: if c_0 is faulty, p_1 could be in the remainder when p_0 completes its handshake and enters the critical section. Consequently, we cannot allow p_0 to enter the critical section without first setting $lock$ (to protect itself by an additional bit in case c_0 is faulty). Achieving this requires a new and more involved handshake procedure. In the new handshake, when p_0 exits the critical section, it first sets $lock$ to 0 to release p_1 from its wait. Then p_0 waits for p_1 to set $lock$ to 1, to ensure that p_1 is protected by 2 bits when p_0 leaves the exit section and goes into the remainder. Hence, replacing one read/write register with a test&set register is not the only price we pay for the extra degree of resilience; the $(1, \infty)$-resilient algorithm also does not have a wait-free exit section.

Finally, the algorithm also incorporates a number of further "sanity checks" in which processes verify that a variable's value has not changed since they last wrote to it, as that may indicate that the other process is not where they expect it to be.

THEOREM 7.1. *There is a two-process $(1, \infty)$-resilient mutual exclusion algorithm that uses two binary read/write variables and one test&set variable, and guarantees starvation-freedom in fault-free executions.*

The code uses the test&set(x) atomic instruction. If $x = 0$, the instruction sets x to 1 and returns 1; otherwise x is left unchanged and a value of 0 is returned. We assume that in addition to the test&set operation, test&set registers can be read and written like a read/write register.

Algorithm T&S-HANDSHAKE: code for process 0

```
001 $c_0 := 1$
002 wait until $lock = 0$
003 while $c_1 = 1$ do
004     if $c_0 = 0$ then goto 010
005 if test&set($lock$) = 0 then
006     goto 001
007 if $c_1 = 1$ then
008     $lock := 0$
009     wait until $c_0 = 0$
010     $c_0 := 1$
011     wait until $c_1 = 0$
012     if test&set($lock$) = 0 then
013         goto 001
014     if $c_0 = 0$ then
015         $lock := 0$
016         goto 002
017     if $c_1 = 1$ then
018         $lock := 0$
019         goto 002
        (enter critical section)
020     $lock := 0$
021     $c_0 := 0$
022     wait until $lock = 1$
023     $c_1 := 1$
024 else
025     if $c_0 = 0$ then // A fault occurred
026         $lock := 0$
027         goto 002
        (enter critical section)
028     $c_0 := 0$
029     $lock := 0$
```

8. TRANSFORMING 1-RESILIENCE INTO f-RESILIENCE

So far we have focused on algorithms that tolerate a single faulty variable, using as few variables as possible. We now turn to consider the more general case of f faulty variables. We show that under a certain "well-behavedness" condition on the algorithm, a $(1, c)$-resilient algorithm using n variables (for $c \in \mathbb{N} \cup \{\infty\}$) implies an (f, c)-resilient algorithm using $(n-1)f+1$ registers. Further, even a non-resilient (but correct) mutual exclusion algorithm that is "well-behaved" can be transformed into an (f, ∞)-resilient using $(f + 1)n$ registers. The property we require of the original 1-resilient or non-resilient algorithm is the following. (The results in this section apply to general m-process mutual exclusion algorithms, but for simplicity we present the results for two processes.)

DEFINITION 8.1 (BOUNDED INTERFERENCE). *An algorithm is said to have the* bounded-interference property *if in all configurations reachable in a fault-free execution,*

(a) Both processes never cover the same register, and

(b) If one process p_i is about to read from a register x_i, then the other process p_{1-i} can only write to x_i a bounded number of times before p_i executes its read.

Many mutual exclusion algorithms (e.g., Dekker and Burns' algorithms) enjoy the bounded-interference property; however, not all do. For example, Peterson's algorithm and Lamport's fast mutual exclusion algorithm both contain a data race, violating requirement (a) (we are not aware of examples in which requirement (b) is violated). The algorithms we present in this paper do have bounded interference, and this allows us to use a simple simulation (instead of, e.g., a linearizable snapshot object, which would require much more memory and may itself be vulnerable to memory faults).

The idea of the simulation is to implement one high-level register x using f low-level registers $x_1, \ldots, x_f$, in such a way that x only exhibits a fault if $x_1, \ldots, x_f$ all experience a memory fault. In the sequel we use Read and Write to denote high-level operations on x (as opposed to low-level operations on $x_1, \ldots, x_f$).

To be useful, our implementation should be *linearizable* [12]: the operations invoked on x should appear to take place instantaneously, as though the algorithm were accessing an atomic faulty read/write register. However, unlike many fault-tolerant simulations (see Section 2), the simulation we give here exposes some subset of the low-level faults, instead of masking them completely. The standard notion of linearizability does not completely characterize the behavior we require. A linearizable implementation takes high-level operations as external input, and maps them into low-level operations. In contrast, with memory faults, we wish to take the *low-level* faults of $x_1, \ldots, x_f$ as (adversarially controlled) input and generate *high-level* faults of x as output. To complicate matters further, we wish to expose only a subset of faults. To capture this behavior we introduce the following definition.[3]

[3]It is also possible to take a more ad-hoc approach and use the standard definition of linearizability, treating low-level faults as high-level operations and high-level faults as low-level operations. A low-level fault that is not exposed on the high level can be viewed as an invoked operation that never returns, and is therefore not linearized.

Algorithm T&S-HANDSHAKE: code for process 1

```
101 $c_1 := 1$
102 wait until $lock = 0$
103 if $c_1 = 0$ then
104     goto 101
105 if $c_0 = 1$ then
106     $c_0 := 0$
107     wait until $c_0 = 1$
108     $c_1 := 0$
109     wait until $c_0 = 1$
110     if test&set($lock$) = 0 then
111         goto 101
112     wait until $c_1 = 1$
113 else
114     if test&set($lock$) = 0 then
115         goto 101
116     if $c_0 = 1$ then
117         $lock := 0$
118         goto 106
119     if $c_1 = 0$ then // A fault occurred
120         $lock := 0$
121         goto 102
    (enter critical section)
122 $lock := 0$
123 $c_1 := 0$
```

DEFINITION 8.2 (FAULT LINEARIZABILITY). *An implementation of a high-level object $\mathcal{O}$ from low-level objects $\mathcal{O}'$ is* fault linearizable *if in every execution, one can*

(a) Embed linearization points *for all the operations on $\mathcal{O}$ that complete and some subset of the operations that do not complete, and*

(b) Insert high-level Flip *events for $\mathcal{O}$ coinciding with some subset of low-level* flip *events of $\mathcal{O}'$,*

such that the following conditions are satisfied:

(a) Each high-level operation of $\mathcal{O}$ is linearized at some point between its invocation and its return (or after its invocation, for operations that do not complete);

(b) If c high-level Flip *events are inserted, then each low-level $\mathcal{O}'$ object experiences at least c faults in the execution; and*

(c) The sequential history obtained by the linearization points and Flip *events represents a valid history of a faulty $\mathcal{O}$ object.*

LEMMA 8.1. *There is a fault linearizable implementation of a faulty read/write register from f faulty read/write registers, such that when used in an fault-free execution of a bounded-interference algorithm, all operations of the high-level register complete.*

PROOF SKETCH. The implementation is very simple: to write a value v to the high-level register x, a process writes v to each of the low-level registers $x_1, \ldots, x_f$. To read from x, a process reads $x_1, \ldots, x_f$, and if all registers contain the same value, that value is returned as the result of the Read. Otherwise the process reads again, until it makes a full pass over $x_1, \ldots, x_f$ in which all registers are observed to have the same value. This value is then returned as the result of the Read.

To show that the implementation is fault linearizable, fix a low-level register x_i, and divide the execution into segments such that each segment ends when x_i flips for the first time inside the segment. We linearize each high-level operation at the point where it last accesses x_i (or at the only point where it accesses x_i, in case of a Write). Note that the value associated with each high-level operation is the value it reads from or writes to x_i for the last time; since x_i does not flip inside each segment, the linearization points of all the operations linearized in a particular segment form a valid sequential history fragment. To complete the picture we insert a high-level Flip at the end of each segment. It is not hard to verify that the sequential history thus obtained is valid. Note also that the number of Flips inserted is exactly the number of times that x_i flips. If we choose x_i to be a register that experiences the minimum number of faults in the execution, we obtain a linearization that satisfies condition (b).

A linearization can be viewed as a mapping from low-level executions to high-level executions, which annotates each low-level configuration with a configuration of the high-level algorithm (formally, the linearization induces a trace simulation between the low-level implementation to the high-level algorithm. The relationship between linearizability and refinement was already explored in [12], where the notion of linearizability was first introduced, and it has also been extensively studied in the formal methods community recently, e.g., [9, 15]). We saw above that we can choose any register and linearize all operations when they last access it. In particular, we can choose the last register accessed, x_f. When we embed linearization points using x_f, any low-level configuration where a Write(x, v) operation is in progress corresponds to a high-level configuration where the invoking process covers x.

The implementation does not, in general, guarantee liveness of any sort. However, when the implementation is used by a bounded-interference algorithm in a fault-free execution, we are guaranteed that no two Write operations overlap: if they did, then the configuration where the second Write is invoked corresponds to a configuration of the algorithm where both processes cover x. This guarantees that whenever a Write completes, all low-level registers $x_1, \ldots, x_f$ contain the same value. Furthermore, if a process invokes a Read(x), we are guaranteed that eventually the other process will cease writing to x. Therefore all operations complete. □

Using the implementation above we can transform a $(1, c)$-resilient algorithm using n registers into an (f, c)-resilient algorithm using $(n - 1)f + 1$ registers. In conjunction with Algorithm HANDSHAKE, we obtain the following general result.

COROLLARY 1. *For any $f \geq 1$, there is a starvation-free $(f, 1)$-resilient mutual exclusion algorithm using $2f + 1$ binary read/write registers.*

PROOF SKETCH. We use $2f$ low-level registers to implement two high-level registers using Lemma 8.1, and the last high-level register is simulated by the remaining single low-level register. Since f faulty low-level registers are required to corrupt either of the first two high-level registers, the adversary cannot cause more than one high-level register to exhibit faulty behavior. Further, if each low-level register flips at most once, then the high-level registers also do not flip more than once. In fault-free executions the simulation guarantees liveness, so the overall algorithm is starvation-free. □

Unfortunately we do not obtain a similar result for the $(1, \infty)$-resilient algorithm, because it uses a test&set register. However, if we use more than f low-level registers to simulate each high-level register, we can use the simulation from Lemma 8.1 to mask faults completely (though the implementation may still deadlock in executions that contain memory faults). In conjunction with, e.g., Dekker's algorithm, we obtain the following result.

COROLLARY 2. *For all $f \geq 1$, there is a starvation-free (f, ∞)-resilient mutex algorithm using $3(f+1)$ binary registers.*

PROOF. We use $f+1$ low-level registers to simulate each of the three high-level registers used by Dekker's algorithm. The adversary cannot corrupt any simulated register, because $f+1$ faulty registers are required to do so. As before, in fault-free executions the simulation is live and starvation-freedom is satisfied. $\square$

9. CONCLUSION

There has been a growing interest in various communities to understand the impact of increasingly unreliable hardware on software in general, and on algorithms in particular. Mutual exclusion is a particularly interesting problem, because the consequences of failure could be dramatic. Off-the-shelf solutions, such as error-correcting codes and specialized hardware, tend to be heavy-weight, and it is not clear that the extra cost is always necessary.

In this paper we have introduced a new variant of fault-tolerance, *safe* fault-tolerance, under which an algorithm must be safe even when memory faults occur, but not necessarily live. The consequences of violating liveness are often less sever than those of violating safety, and in many cases there are already systems in place to detect and resolve deadlock. Sacrificing liveness in faulty executions allowed us to design two-process mutual exclusion algorithms that tolerate a faulty variable, at the cost of no extra memory.

It is clear that our work is only a first step; many problems remain open. Our results in this paper focus mostly on the two-process case; in follow-up work we intend to extend the results to n-processes. Also, we focus in this paper primarily on binary shared variables, i.e., the type of variable used in Peterson's and Dekker's algorithms. Our results show that even in this restricted setting, a significant degree of fault-resilience can be achieved; nevertheless, it is interesting to consider whether the lower bound from Section 6 continues to hold for general multi-valued registers, or whether a $(2f+1)$-variable (f, ∞)-resilient algorithm exists.

Acknowledgement. We are indebted to John Douceur and Karthik Pattabiraman for fruitful early discussions on the problem.

10. REFERENCES

[1] Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld. Computing with Faulty Shared Memory. In *Proceedings of Symposium on Principles of Distributed Computing (PODC)*, 1992.

[2] Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld. Computing with Faulty Shared Objects. *Journal of the ACM*, 1995.

[3] R. C. Baumann. Soft Errors in Advanced Semiconductor Devices – Part I: The Three Radiation Sources. *IEEE Transactions on Device and Materials Reliability*, 2001.

[4] R. C. Baumann. Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, 2002.

[5] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 2005.

[6] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107:171–184, December 1993.

[7] B. S. Chlebus, A. Gambin, and P. Indyk. Shared-Memory Simulations on a Faulty-Memory DMM. In *Proceedings of 23rd Colloquium on Automata, Languages and Programming (ICALP)*, 1996.

[8] B. S. Chlebus, L. Gasieniec, and A. Pelc. Deterministic Computations on a PRAM with Static Processor and Memory Faults. *Fundamenta Informaticae*, 2003.

[9] J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In J. Davies and J. Gibbons, editors, *IFM*, volume 4591 of *Lecture Notes in Computer Science*, pages 195–214. Springer, 2007.

[10] I. Finocchi, F. Grandoni, and G. F. Italiano. Designing Reliable Algorithms in Unreliable Memories. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 1–8, 2005.

[11] R. Guerraoui and M. Raynal. From Unreliable Objects to Reliable Objects: The Case of Atomic Registers and Consensus. In *Proceedings of PaCT*, 2007.

[12] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[13] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 1998.

[14] L. Lamport. The Mutual Exclusion Problem: Part II – Statement and Solutions. *Journal of the ACM*, 1986.

[15] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 321–337, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] T. N. V. M. Gomaa, C. Scarbrough and I. Pomeranz. Transient-fault Recovery for Chip Multiprocessors. In *Proceedings of 30th Symposium on Computer Architecture (ISCA)*, pages 98–109, 2003.

[17] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of 29th Symposium on Computer Architecture (ISCA)*, pages 99–110, 2002.

[18] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Transactions on Reliability*, 2002.

[19] G. L. Peterson. Concurrent Reading while Writing. *Transactions on Programming Languages and Systems*, 1983.

[20] G. A. Reis, J. Chang, and D. I. August. Automatic Instruction-Level Software-Only Recovery Methods. *IEEE Micro Top Picks*, 2007.

[21] N. W. H. B. E. T. S. E. Michalak, K. W. Harris and S. A. Wender. Predicting the Number of Fatal Soft Errors in Los Alamos National Labratory's ASC Q Computer. *IEEE Transactions on Device and Materials Reliability*, 2005.

[22] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proceedings of the Conference on Dependable Systems and Networks*, pages 389–388, 2002.

[23] B. K. Szymanski. Mutual Exclusion Revisited. In *Proceedings of 5th Jerusalem Conference on Information Technology*, 1990.

[24] J. Tromp. How to Construct an Atomic Variable. In *Proceedings of 3rd Workshop on Distributed Algorithms*, 1989.

[25] K. Truuvert. A Self-Stabilizing First-Come-First-Serve Mutual Exclusion Algorithm with Small Shared Variables. *Technical Note, University of Toronto*, 1989.