Are Lock-Free Concurrent Algorithms Practically Wait-Free?

Dan Alistarh*
MSR Cambridge
dan.alistarh@microsoft.com

Keren Censor-Hillel†
Technion
ckeren@cs.technion.ac.il

Nir Shavit‡
MIT & Tel-Aviv University
shanir@csail.mit.edu.

Abstract

Lock-free concurrent algorithms guarantee that *some* concurrent operation will always make progress in a finite number of steps. Yet programmers prefer to treat concurrent code as if it were *wait-free*, guaranteeing that *all* operations always make progress. Unfortunately, designing wait-free algorithms is generally a very complex task, and the resulting algorithms are not always efficient. While obtaining efficient wait-free algorithms has been a long-time goal for the theory community, most non-blocking commercial code is only lock-free.

This paper suggests a simple solution to this problem. We show that, for a large class of lock-free algorithms, under scheduling conditions which approximate those found in commercial hardware architectures, lock-free algorithms behave as if they are wait-free. In other words, programmers can keep on designing simple lock-free algorithms instead of complex wait-free ones, and in practice, they will get wait-free progress.

Our main contribution is a new way of analyzing a general class of lock-free algorithms under a *stochastic scheduler*. Our analysis relates the individual performance of processes with the global performance of the system using *Markov chain lifting* between a complex per-process chain and a simpler system progress chain. We show that lock-free algorithms are not only wait-free with probability 1, but that in fact a general subset of lock-free algorithms can be closely bounded in terms of the average number of steps required until an operation completes.

To the best of our knowledge, this is the first attempt to analyze progress conditions, typically stated in relation to a worst case adversary, in a stochastic model capturing their expected asymptotic behavior.

1 Introduction

The introduction of multicore architectures as today's main computing platform has brought about a renewed interest in concurrent data structures and algorithms, and a considerable amount of research has focused on their modeling, design and analysis.

The behavior of concurrent algorithms is captured by *safety properties*, which guarantee their correctness, and *progress properties*, which guarantee their termination. In this paper, we focus on progress guarantees, and in particular on the question of whether a concurrent algorithm is *lock-free* or *wait-free*. Intuitively, lock-free means that *some* process is always guaranteed to make progress by completing its operations within a finite number of system steps, while wait-free means that *each* process completes its operations within a finite number of its own steps.

An increasingly large fraction of concurrent commercial code is *lock-free*, i.e. ensures that the system always makes progress, without using locks [7,11]. Over the years, the research community has devised ingenious, technically sophisticated algorithms that are *wait-free*, i.e. ensure that each operation makes progress, without using locks [10]. Unexpectedly, wait-free algorithms are not being adopted by practitioners, despite the fact that the completion of all method calls in a program is a natural assumption that programmers implicitly make.

Recently, Herlihy and Shavit [11] suggested that perhaps the answer lies in a surprising property of lock-free algorithms: in practice, they often behave as if they were wait-free. Specifically, most operations complete in a timely

*Part of this work was performed while the author was a Postdoctoral Associate at MIT CSAIL, where he was supported by SNF Postdoctoral Fellows Program, NSF grant CCF-1217921, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

†Shalon Fellow.

‡This work was supported in part by NSF grants CCF-1217921 and CCF-1301926, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

manner, and the impact of long worst-case executions on performance is negligible. In other words, in real systems, the scheduler that governs the threads' behavior in long executions does not single out any particular thread in order to cause the theoretically possible bad behaviors. This raises the following question: could the choice of *wait-free* versus *lock-free* be based simply on what assumption a programmer is willing to make about the underlying scheduler, and, with the right kind of scheduler, one will not need wait-free algorithms except in very rare cases?

This question is important because the difference between a wait-free and a lock-free algorithm for any given problem typically involves the introduction of specialized "helping" mechanisms [10], which significantly increase the complexity (both the design complexity and time complexity) of the solution. If one could simply rely on the scheduler, adding a helping mechanism to guarantee wait-freedom (or starvation-freedom) would be unnecessary.

Unfortunately, there is currently no analytical framework which would allow answering the above question, since it would require predicting the behavior of a concurrent algorithm over long executions, under a scheduler that is not adversarial.

Contribution. In this paper, we take a first step towards such a framework. Following empirical observations, we introduce a *stochastic scheduler* model, and use this model to predict the long-term behavior of a general class of concurrent algorithms. The stochastic scheduler is similar to an adversary: at each time step, it picks some process to schedule. The main distinction is that, in our model, the scheduler's choices *contain some randomness*. In particular, a stochastic scheduler has a probability threshold $\theta > 0$ such that every (non-faulty) process is scheduled with probability at least θ in each step.

We start from the following simple observation: under *any stochastic scheduler*, every *bounded lock-free* algorithm is actually *wait-free with probability* 1. (A *bounded* lock-free algorithm guarantees that *some* process always makes progress within a finite progress bound.) In other words, for any such algorithm, the schedules which prevent a process from ever making progress must have probability mass 0. The intuition is that, with probability 1, each specific process eventually takes enough consecutive steps, implying that it completes its operation. This observation applies in fact to more general minimal/maximal progress conditions [11], and in particular to algorithms that employ locks. However, this intuition is insufficient for explaining why lock-free data structures are *efficient* in practice: the result applies to arbitrary algorithms, but the upper bound it yields on the number of steps until an operation completes is unacceptably high.

Our main contribution is analyzing a general class of lock-free algorithms under a specific stochastic scheduler, and showing that not only are they wait-free with probability 1, but that in fact they provide a pragmatic bound on the number of steps until each operation completes.

We address a refined *uniform* stochastic scheduler, which schedules each non-faulty process with uniform probability in every step. Empirical data suggests that, in the long run, the uniform stochastic scheduler is a reasonable approximation for a real-world scheduler. We emphasize that we do not claim real schedulers are uniform stochastic, but only that such a scheduler gives an approximation of what happens in practice for our complexity measures, over long executions.

We call the algorithmic class we analyze *single compare-and-swap universal* (SCU). An algorithm in this class is divided into a *preamble*, and a *scan-and-validate* phase. The preamble executes auxiliary code, such as local updates and memory allocation. In the second phase, the process first determines the data structure state by scanning the memory. It then locally computes the updated state after its method call would be performed, and attempts to commit this state to memory by performing an atomic *compare-and-swap* (CAS) operation. If the CAS operation succeeds, then the state has been updated, and the method call completes. Otherwise, if some other process changes the state in between the scan and the attempted update, then the CAS operation fails, and the process must restart its operation.

This algorithmic class is widely used to design lock-free data structures. It is known that every sequential object has a lock-free implementation in this class using a lock-free version of Herlihy's universal construction [10]. Instances of this class are used to obtain efficient data structures such as stacks [15], queues [13], or hash tables [7]. The read-copy-update (RCU) [8] synchronization mechanism employed by the Linux kernel is also an instance of this pattern.

We examine the class SCU under a uniform stochastic scheduler, and first observe that, in this setting, every such algorithm behaves as a Markov chain. The computational cost of interest is *system steps*, i.e. shared memory accesses by the processes. The complexity metrics we analyze are *individual latency*, which is the expected number of steps of the system until a specific process completes a method call, and *system latency*, which is the expected number of steps

of the system to complete *some* method call. We bound these parameters by studying the stationary distribution of the Markov chain induced by the algorithm.

We prove two main results. The first is that, in this setting, all algorithms in this class have the property that the individual latency of any process is n times the system latency. In other words, the expected number of steps for any two processes to complete an operation is *the same*. Moreover, the expected number of steps for the system to complete any operation is the expected number of steps for a specific process to complete an operation, divided by n. The second result is an upper bound of $O(q + s\sqrt{n})$ on the system latency, where q is the number of steps in the preamble, s is the number of steps in the scan-and-validate phase, and n is the number of processes. This bound is asymptotically tight.

The key mathematical tool we use is *Markov chain lifting* [4,9]. More precisely, for such algorithms, we prove that there exists a function which *lifts* the complex Markov chain induced by the algorithm to a simplified *system* chain. The asymptotics of the system latency are determined from the minimal progress chain.

More precisely, we bound system latency by characterizing the average behavior of a new type of *iterated* balls-into-bins game, consisting of iterations which end when a certain condition on the bins first occurs, after which some of the bins change their state and a new iteration begins. Once this bound is in place, we use the lifting to prove that the individual latency is always n times the system latency, which implies that the individual latency is $O(n(q + s\sqrt{n}))$.

This analysis suggests that, under this scheduler, the amortized cost of an operation depends on $\Theta(\sqrt{n})$. This factor can be seen as the *price of contention*, i.e., the extra cost of having processors interrupt each other's progress because of contention in the scan-and-validate phase. Notice that the worst-case analysis of the same algorithm yields an $\Omega(n)$ amortized step cost per operation.

In summary, our analysis shows that, under an approximation of the real-world scheduler, a large class of lock-free algorithms provide virtually the same progress guarantees as wait-free ones, and that, roughly, the system completes requests at a rate that is n times that of individual processes. It provides an analytical framework for predicting the behavior of a class of concurrent algorithms, over long executions, under a scheduler that is not adversarial.

Related work. To the best of our knowledge, the only prior work which addresses a probabilistic scheduler for a shared memory environment is that of Aspnes [3], who gave a fast consensus algorithm under a probabilistic scheduler model different from the one considered in this paper. The observation that many lock-free algorithms behave as wait-free in practice was made by Herlihy and Shavit in the context of formalizing minimal and maximal progress conditions [11], and is well-known among practitioners. For example, reference [1, Figure 6] gives empirical results for the latency distribution of individual operations of a lock-free stack. A parallel line of work considered designing contention managers which convert lock-free or obstruction-free algorithms into wait-free ones, e.g. [6]. In contrast, we explore how models of existing processor schedulers impact the behavior of lock-free algorithms.

Roadmap. We describe the model, progress guarantees, and complexity metrics in Section 2. In particular, Section 2.3 defines the stochastic scheduler. We show that lock-free becomes wait-free with probability 1 in Section 3. Section 4 defines the algorithmic class $SCU(q, s)$, while Section 5.1 analyzes individual and global latency. The full version of this paper [2] contains empirical justification for the model, and the complete analysis.

2 System Model

2.1 Preliminaries

Processes and Objects. We consider a shared-memory model, in which n processes $p_1, \ldots, p_n$, communicate through registers, on which they perform atomic read, write, and compare-and-swap (CAS) operations. A CAS operation takes three arguments (R, $expVal$, $newVal$), where R is the register on which it is applied, $expVal$ is the expected value of the register, and $newVal$ is the new value to be written to the register. If $expVal$ matches the value of R, then we say that the CAS is successful, and the value of R is updated to $newVal$. Otherwise, the CAS fails. The operation returns *true* if it successful, and *false* otherwise.

We assume that each process has a unique identifier. Processes follow an algorithm, composed of shared-memory steps and local computation. The order of process steps is controlled by the *scheduler*. A set of at most $n-1$ processes may fail by crashing. A crashed process stops taking steps for the rest of the execution. A process that is not crashed

at a certain step is *correct*, and if it never crashes then it takes an infinite number of steps in the execution.

The algorithms we consider are implementations of shared objects. A shared object O is an abstraction providing a set of *methods* M, each given by its sequential specification. In particular, an implementation of a method m for object O is a set of n algorithms, one for each executing process. When process p_i invokes method m of object O, it follows the corresponding algorithm until it receives a response from the algorithm. Upon receiving the response, the process is immediately assigned another method invocation. In the following, we do not distinguish between a method m and its implementation. A method invocation is *pending* at some point in the execution if has not received a response. A pending method invocation is *active* if it is made by a *correct* process (note that the process may still crash in the future).

Executions, Schedules, and Histories. An execution is a sequence of operations performed by the processes. To represent executions, we assume discrete time, where at every time unit only one process is scheduled. In a time unit, a process can perform any number of local computations or coin flips, after which it issues a *step*, which consists of a single shared memory operation. Whenever a process becomes active, as decided by the scheduler, it performs its local computation and then executes a step. The *schedule* is a (possibly infinite) sequence of process identifiers. If process p_i is in position $\tau \geq 1$ in the sequence, then p_i is active at time step τ.

Raising the level of abstraction, we define a *history* as a finite sequence of method invocation and response events. Notice that each schedule has a corresponding history, in which individual process steps are mapped to method calls. On the other hand, a history can be the image of several schedules.

2.2 Progress Guarantees

We now define progress guarantees, following the unified presentation from [11]. Instead of specifying progress guarantees for each method of an object, for ease of presentation, we adopt the simpler definition which specifies progress provided by an implementation. Consider an infinite execution e, with the corresponding history H_e. An implementation of an object O provides *minimal progress* in the execution e if, in every suffix of H_e, some pending active invocation of some method has a matching response. Equivalently, there is no point in the corresponding execution from which all the processes take an infinite number of steps without returning from their invocation.

An implementation provides *maximal* progress in an execution e if, in every suffix of the corresponding history H_e, *every* pending active invocation of a method has a response. Equivalently, there is no point in the execution from which a process takes infinitely many steps without returning.

Scheduler Assumptions. We say that an execution is *crash-free* if each process is always correct, i.e. if each process takes an infinite number of steps.

Progress. An implementation is *deadlock-free* if it guarantees minimal progress in every crash-free execution, and maximal progress in some crash-free execution.[1] An implementation is *starvation-free* if it guarantees maximal progress in every crash-free execution. An implementation is *lock-free* if it guarantees minimal progress in every execution, and maximal progress in some execution. An implementation is *wait-free* if it guarantees maximal progress in every execution.

Bounded Progress. While the above definitions provide reasonable measures of progress, often in practice more explicit progress guarantees may be desired, which provide an upper bound on the number of steps until some method makes progress. To model this, we say that an implementation guarantees *bounded minimal progress* if there exists a bound $B > 0$ such that, for any time step t in the execution e at which there is an active invocation of some method, some invocation of a method returns within the next B steps by all processes. An implementation guarantees *bounded maximal progress* if there exists a bound $B > 0$ such that *every* active invocation of a method returns within B steps by all processes. We can specialize the definitions of bounded progress guarantees to the scheduler assumptions considered above to obtain definitions for *bounded* deadlock-freedom, *bounded* starvation-freedom, and so on.

[1] According to [11], the algorithm is required to guarantee maximal progress in some execution to rule out pathological cases where a thread locks the object and never releases the lock.

2.3 Stochastic Schedulers

We define a stochastic scheduler as follows.

Definition 1 (Stochastic Scheduler)**.** *For any $n \geq 0$, a scheduler for n processes is defined by a triple $(\Pi_\tau, A_\tau, \theta)$. The parameter $\theta \in [0, 1]$ is the* threshold*. For each time step $\tau \geq 1$, Π_τ is a probability distribution for scheduling the n processes at τ, and A_τ is the subset of* possibly active *processes at time step τ. At time step $\tau \geq 1$, the distribution Π_τ gives, for every $i \in \{1, \ldots, n\}$ a probability γ_τ^i, with which process p_i is scheduled. The distribution Π_τ may depend on arbitrary outside factors, such as the current state of the algorithm being scheduled. A scheduler $(\Pi_\tau, A_\tau, \theta)$ is* stochastic *if $\theta > 0$. For every $\tau \geq 1$, the parameters must ensure the following:*

1. *(Well-formedness) $\sum_{i=1}^{n} \gamma_\tau^i = 1$;*
2. *(Weak Fairness) For every process $p_i \in A_\tau$, $\gamma_\tau^i \geq \theta$;*
3. *(Crashes) For every process $p_i \notin A_\tau$, $\gamma_\tau^i = 0$;*
4. *(Crash Containment) $A_{\tau+1} \subseteq A_\tau$.*

The well-formedness condition ensures that some process is always scheduled. Weak fairness ensures that, for a stochastic scheduler, possibly active processes do get scheduled with some non-zero probability. The crash condition ensures that failed processes do not get scheduled. The set $\{p_1, p_2, \ldots, p_n\} \setminus A_\tau$ can be seen as the set of crashed processes at time step τ, since the probability of scheduling these processes at every subsequent time step is 0.

An Adversarial Scheduler. Any classic asynchronous shared memory adversary can be modeled by "encoding" its adversarial strategy in the probability distribution Π_τ for each step. Specifically, given an algorithm A and a worst-case adversary $\mathcal{A}_A$ for A, let p_i^τ be the process that is scheduled by $\mathcal{A}_A$ at time step τ. Then we give probability 1 in Π_τ to process p_i^τ, and 0 to all other processes. Things are more interesting when the threshold θ is strictly more than 0, i.e., there is some randomness in the scheduler's choices.

The Uniform Stochastic Scheduler. A natural scheduler is the *uniform* stochastic scheduler, for which, assuming no process crashes, we have that Π_τ has $\gamma_i^\tau = 1/n$, for all i and $\tau \geq 1$, and $A_\tau = \{1, \ldots, n\}$ for all time steps $\tau \geq 1$. With crashes, we have that $\gamma_i^\tau = 1/|A_\tau|$ if $i \in A_\tau$, and $\gamma_i^\tau = 0$ otherwise.

2.4 Complexity Measures

Given a concurrent algorithm, standard analysis focuses on two measures: *step complexity*, the worst-case number of steps performed by a single process in order to return from a method invocation, and *total step complexity*, or *work*, which is the worst-case number of system steps required to complete invocations of all correct processes when performing a task together. In this paper, we focus on the analogue of these complexity measures for long executions. Given a stochastic scheduler, we define *(average) individual latency* as the maximum over all inputs of the expected number of steps taken by the system between the returns times of two consecutive invocations of the same process. Similarly, we define the *(average) system latency* as the maximum over all inputs of the expected number of system steps between consecutive returns times of any two invocations.

2.5 Background on Markov Chains

We now give a brief overview of Markov chains. Our presentation follows standard texts, e.g. [12, 14]. The definition and properties of Markov chain lifting are lifted from [9].

Given a set S, a sequence of random variables $(X_t)_{t \in \mathbb{N}}$, where $X_t \in S$, is a (discrete-time) *stochastic process* with states in S. A *discrete-time Markov chain* over the state set S is a discrete-time stochastic process with states in S that satisfies the *Markov condition* $\Pr[X_t = i_t | X_{t-1} = i_{t-1}, \ldots, X_0 = i_0] = \Pr[X_t = i_t | X_{t-1} = i_{t-1}]$.
The above condition is also called the *memoryless property*. A Markov chain is *time-invariant* if the equality $\Pr[X_t = j | X_{t-1} = i] = \Pr[X_{t'} = j | X_{t'-1} = i]$ holds for all times $t, t' \in \mathbb{N}$ and all $i, j \in S$. This allows us to define the *transition matrix* P of a Markov chain as the matrix with entries

$$p_{ij} = \Pr[X_t = j | X_{t-1} = i].$$

The *initial distribution* of a Markov chain is given by the probabilities $\Pr[X_0 = i]$, for all $i \in S$. We denote the time-invariant Markov chain X with initial distribution λ and transition matrix P by $M(P, \lambda)$.

The random variable $T_{ij} = \min\{n \geq 1 | X_n = j, \text{ if } X_0 = i\}$ counts the number of steps needed by the Markov chain to get from i to j, and is called the *hitting time* from i to j. We set $T_{i,j} = \infty$ if state j is unreachable from i. Further, we define $h_{ij} = E[T_{ij}]$, and call $h_{ii} = E[T_{ii}]$ the (expected) return time for state $i \in S$.

Given P, the transition matrix of $M(P, \lambda)$, a *stationary distribution* of the Markov chain is a state vector π with $\pi = \pi P$. (We consider *row* vectors throughout the paper.) The intuition is that if the state vector of the Markov chain is π at time t, then it will remain π for all $t' > t$. Let $P^{(k)}$ be the transition matrix P multiplied by itself k times, and $p_{ij}^{(k)}$ be element (i, j) of $P^{(k)}$. A Markov chain is *irreducible* if for all pairs of states $i, j \in S$ there exists $m \geq 0$ such that $p_{ij}^{(m)} > 0$. (In other words, the underlying graph is strongly connected.) This implies that $T_{ij} < \infty$, and all expectations h_{ij} exist, for all $i, j \in S$. Furthermore, the following is known.

Theorem 1. *An irreducible finite Markov chain has a unique stationary distribution π, namely*

$$\pi_j = \frac{1}{h_{jj}}, \forall j \in S.$$

The periodicity of a state j is the maximum positive integer α such that $\{n \in \mathbb{N} | p_{jj}^{(n)} > 0\} \subseteq \{i\alpha | i \in \mathbb{N}\}$. A state with periodicity $\alpha = 1$ is called *aperiodic*. A Markov chain is *aperiodic* if all states are aperiodic. If a Markov chain has at least one self-loop, then it is aperiodic. A Markov chain that is irreducible and aperiodic is *ergodic*. Ergodic Markov chains converge to their stationary distribution as $t \to \infty$ independently of their initial distributions.

Theorem 2. *For every ergodic finite Markov chain $(X_t)_{t \in \mathbb{N}}$ we have independently of the initial distribution that $\lim_{t \to \infty} q_t = \pi$, where π denotes the chain's unique stationary distribution, and q_t is the distribution on states at time $t \in \mathbb{N}$.*

Ergodic Flow. It is often convenient to describe an ergodic Markov chain in terms of its *ergodic flow*: for each (directed) edge ij, we associate a flow $Q_{ij} = \pi_i p_{ij}$. These values satisfy $\sum_i Q_{ij} = \sum_i Q_{ji}$ and $\sum_{i,j} Q_{ij} = 1$. It also holds that $\pi_j = \sum_i Q_{ij}$.

Lifting Markov Chains. Let M and M' be ergodic Markov chains on finite state spaces S, S', respectively. Let P, π be the transition matrix and stationary distribution for M, and P', π' denote the corresponding objects for M'. We say that M' is a *lifting* of M [9] if there is a function $f : S' \to S$ such that

$$Q_{ij} = \sum_{x \in f^{-1}(i), y \in f^{-1}(j)} Q'_{xy}, \forall i, j \in S.$$

Informally, M' is collapsed onto M by clustering several of its states into a single state, as specified by the function f. The above relation specifies a homomorphism on the ergodic flows. An immediate consequence of this relation is the following connection between the stationary distributions of the two chains.

Lemma 1. *For all $v \in S$, we have that*

$$\pi(v) = \sum_{x \in f^{-1}(v)} \pi'(x).$$

3 From Minimal Progress to Maximal Progress

We now formalize the intuition that, under a stochastic scheduler, all algorithms ensuring bounded minimal progress guarantee in fact maximal progress with probability 1. We also show the *bounded* minimal progress assumption is necessary: if minimal progress is not bounded, then maximal progress may not be achieved. The proof of this result is relatively straightforward, and is therefore left to the full version of this paper [2].

Theorem 3 (Min to Max Progress)**.** *Let $\mathcal{S}$ be a stochastic scheduler with probability threshold $1 \geq \theta > 0$. Let A be an algorithm ensuring bounded minimal progress with a bound T. Then A ensures maximal progress with probability 1. Moreover, the expected maximal progress bound of A is at most $(1/\theta)^T$.*

```
1  Shared: registers R, R_1, R_2, ..., R_{s-1}
2    procedure method-call()
3    Take preamble steps O_1, O_2, O_q /* Preamble region */
4    while true do
         /* Scan region:  */
5        v ← R.read()
6        v_1 ← R_1.read(); v_2 ← R_2.read(); ...; v_{s-1} ← R_{s-1}.read()
7        v' ← new proposed state based on v, v_1, v_2, ..., v_{s-1}
         /* Validation step:  */
8        flag ← CAS(R, v, v')
9        if flag = true then
10           output success
```

Algorithm 1: The structure of algorithms in $SCU_{q,s}$.

The proof is based on the fact that, for every correct process p_i, eventually, the scheduler will produce a solo sequence of length T. On the other hand, since the algorithm ensures minimal progress with bound T, p_i must complete its operation during this interval.

This result matches the intuition for why many practical lock-free algorithms behave as being wait-free in practice: even if we only allow operations to complete when running solo, there is always a chance that a process will get enough consecutive steps to complete. Notice that the same proof applies if instead of considering the minimal progress bound T, we consider the individual progress bound T', which is a bound on the number of consecutive steps a process has to take in order to complete. This reduces the obtained bound for maximal progress from $(1/\theta)^T$ to $(1/\theta)^{T'}$, which may be much smaller.

We then prove that the finite bound for minimal progress is necessary. For this, we devise an *unbounded* lock-free algorithm which is not wait-free with probability > 0. The main idea is to have processes that fail to change the value of a CAS repeatedly increase the number of steps they need to take to complete an operation. The argument is given in the full version of this paper.

Lemma 2. *There exists an unbounded lock-free algorithm that is* not wait-free *with high probability.*

4 The Class of Algorithms *SCU*

In this section, we define the class of algorithms $SCU(q, s)$. An algorithm in this class is structured as follows. (See Algorithm 1 for the pseudocode, and Figure 2 for an illustration.) The first part is the *preamble*, where the process performs a series of q steps. The algorithm then enters a *loop*, divided into a *scan* region, which reads the values of s registers, and a *validation* step, where the process performs a CAS operation, which attempts to change the value of a register. The goal of the scan region is to obtain a view of the data structure state. In the validation step, the process checks that this state is still valid, and attempts to change it. If the CAS is successful, then the operation completes. Otherwise, the process restarts the loop. We say that an algorithm with the above structure with parameters q and s is in $SCU(q, s)$.

We assume that steps in the preamble may perform memory updates, including to registers $R_1, \ldots, R_{s-1}$, but do not change the value of the decision register R. Also, for simplicity, two processes never propose the same value for the register R. (This can be easily enforced by adding a timestamp to each request.) The order of steps in the scan region can be changed without affecting our analysis. Such algorithms are used in several CAS-based concurrent implementations. In particular, the class can be used to implement a concurrent version of every sequential object [10]. It has also been used to obtain implementations of several concurrent objects, such as counters [5], stacks [15], and queues [13].

5 Analysis of the Class *SCU*

We analyze the performance of algorithms in $SCU(q, s)$ under the uniform stochastic scheduler. We assume that all threads execute the same method call with preamble of length q, and scan region of length s. Each thread executes an infinite number of such operations. To simplify the presentation, we assume all n threads are correct in the analysis. The claim is similar in the crash-failure case, and will be considered separately.

We examine two parameters: system latency, i.e., how often (in terms of system steps) does a new operation complete, and individual latency, i.e., how often does *a certain thread* complete a new operation. Notice that the worst-case latency for the whole system is $\Theta(q + sn)$ steps, while the worst-case latency for an individual thread is ∞, as the algorithm is not wait-free. We will prove the following result:

Theorem 4. *Let A be an algorithm in $SCU(q, s)$. Then, under the uniform stochastic scheduler, the system latency of A is $O(q + s\sqrt{n})$, and the individual latency is $O(n(q + s\sqrt{n}))$.*

We prove the upper bound by splitting the class $SCU(q, s)$ into two separate components, and analyzing each under the uniform scheduler. The first part is the loop code, which we call the *scan-validate* component. The second part is the *parallel code*, which we use to characterize the performance of the preamble code. In other words, we first consider $SCU(0, s)$ and then $SCU(q, 0)$.

5.1 The Scan-Validate Component

Without loss of generality, we can simplify the pseudocode to contain a single read step before the CAS. We obtain the performance bounds for this simplified algorithm, and then multiply them by s, the number scan steps. That is, we start by analyzing $SCU(0, 1)$ and then generalize to $SCU(0, s)$.

Proof Strategy. We start from the Markov chain representation of the algorithm, which we call the *individual chain*. We then focus on a simplified representation, which only tracks *system-wide progress*, irrespective of which process is exactly in which state. We call this the *system chain*. We first prove the individual chain can be related to the system chain via a lifting function, which allows us to relate the individual latency to the system latency (Lemma 4). We then focus on bounding system latency. We describe the behavior of the system chain via an iterated balls-and-bins game, whose stationary behavior we analyze in Lemmas 7 and 8. Finally, we put together these claims to obtain an $O(\sqrt{n})$ upper bound on the system latency of $SCU(0, 1)$.

Due to space limitations, some proofs are omitted. They can be found in the full version of the paper [2].

5.1.1 Markov Chain Representations

We define the *extended state* of a process in terms of the state of the system, and of the type of step it is about to take. Thus, a process can be in one of three states: either it performs a read, or it CAS-es with the current value of R, or it CAS-es with an invalid value of R. The state of the system after each step is completely described by the n extended states of processes. We emphasize that this is different than what is typically referred to as the "local" state of a process, in that the extended state is described from the viewpoint of the entire system. That is, a process that has a pending CAS operation can be in either of two different extended states, depending on whether its CAS will succeed or not. This is determined by the state of the entire system. A key observation is that, although the "local" state of a process can only change when it takes a step, its extended state can change also when another process takes a step.

The individual chain. Since the scheduler is uniform, the system can be described as a Markov chain, where each state specifies the extended state of each process. Specifically, a process is in state $OldCAS$ if it is about to CAS with an old (invalid) value of R, it is in state $Read$ if it is about to read, and is in state $CCAS$ if it about to CAS with the current value of R. (After CAS-ing, the process returns to state $Read$.)

A state S of the individual chain is given by a combination of n states $S = (P_1, P_2, \ldots, P_n)$, describing the extended state of each process, where, for each $i \in \{1, \ldots, n\}$, $P_i \in \{OldCAS, Read, CCAS\}$ is the extended state of process p_i. There are $3^n - 1$ possible states, since the state where each process CAS-es with an old value cannot occur. In each transition, each process takes a step, and the state changes correspondingly. Recall that every process p_i takes a step with probability $1/n$. Transitions are as follows. If the process p_i taking a step is in state

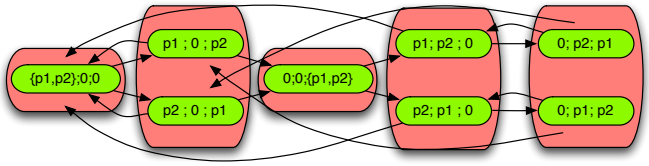

Figure 1: The individual chain and the global chain for two processes. Each transition has probability $1/2$. The red clusters are the states in the system chain. The notation $X; Y; Z$ means that processes in X are in state $Read$, processes in Y are in state $OldCAS$, and processes in Z are in state $CCAS$.

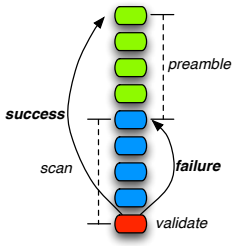

Figure 2: Algorithm in $SCU(q, s)$.

$Read$ or $OldCAS$, then all other processes remain in the same extended state, and p_i moves to state $CCAS$ or $Read$, respectively. If the process p_i taking a step is in state $CCAS$, then all processes in state $CCAS$ move to state $OldCAS$, and p_i moves to state $Read$.

The system chain. To reduce the complexity of the individual Markov chain, we introduce a simplified representation, which tracks system-wide progress. More precisely, each state of the system chain tracks the number of processes in each state, irrespective of their identifiers: for any $a, b \in \{0, \ldots, n\}$, a state x is defined by the tuple (a, b), where a is the number of processes that are in state $Read$, and b is the number of processes that are in state $OldCAS$. Notice that the remaining $n - a - b$ processes must be in state $CCAS$. The initial state is $(n, 0)$, i.e. all processes are about to read. The state $(0, n)$ does not exist. The transitions in the system chain are as follows. $\Pr[(a+1, b-1)|(a, b)] = b/n$, where $0 \leq a \leq n$ and $b > 0$. $\Pr[(a+1, n-a-1)|(a, b)] = 1 - (a+b)/n$, where $0 \leq a < n$. $\Pr[(a-1, b)|(a, b)] = a/n$, where $0 < a \leq n$. See Figure 1 for a simple example of the two chains, and their lifting, for $n = 2$.

5.1.2 Lifting the Individual Chain

We start from the observation that both the individual chain and the system chain are ergodic. Let π be the stationary distribution of the system chain, and let π' be the stationary distribution for the individual chain. For any state $k = (a, b)$ in the system chain, let π_k be its probability in the stationary distribution. Similarly, for state x in the individual chain, let π'_x be its probability in the stationary distribution.

We now prove that there exists a *lifting* from the individual chain to the system chain. Intuitively, the lifting from the individual chain to the system chain collapses all states in which a processes are about to read and b processes are about to CAS with an old value (the identifiers of these processes are different for distinct states), into to state (a, b) from the system chain.

Definition 2. *Let $\mathcal{S}$ be the set of states of the individual chain, and $\mathcal{M}$ be the set of states of the system chain. We define the function $f : \mathcal{S} \to \mathcal{M}$ such that each state $S = (P_1, \ldots, P_n)$, where a processes are in state $Read$ and b processes are in state $OldCAS$, is taken into state (a, b) of the system chain.*

We then obtain the following relation between the stationary distributions of the two chains.

Lemma 3. *For every state k in the system chain, we have $\pi_k = \sum_{x \in f^{-1}(k)} \pi'_x$.*

Proof. We obtain this relation algebraically, starting from the formula for the stationary distribution of the individual chain. We have that $\pi' A = \pi'$, where π' is a row vector, and A is the transition matrix of the individual chain. We partition the states of the individual chain into sets, where $G_{a,b}$ is the set of system states S such that $f(S) = (a, b)$. Fix an arbitrary ordering $(G_k)_{k \geq 1}$ of the sets, and assume without loss of generality that the system states are ordered according to their set in the vector π and in the matrix A, so that states mapping to the same set are consecutive.

Let now A' be the transition matrix across the sets $(G_k)_{k \geq 1}$. In particular, a'_{kj} is the probability of moving from a state in the set G_k to some state in the set G_j. Note that this transition matrix is the same as that of the system chain.

Pick an arbitrary state x in the individual chain, and let $f(x) = (a, b)$. In other words, state x maps to set G_k, where $k = (a, b)$. We claim that for every set G_j, $\sum_{y \in G_j} \Pr[y|x] = \Pr[G_j|G_i]$.

To see this, fix $x = (P_0, P_1, \ldots, P_n)$. Since $f(x) = (a, b)$, there are exactly b distinct states y reachable from x such that $f(y) = (a+1, b-1)$: the states where a process in extended local state $OldCAS$ takes a step. Therefore, the probability of moving to such a state y is b/n. Similarly, the probability of moving to a state y with $f(y) = (a+1, b-1)$ is $1 - (a + b)/n$, and the probability of moving to a state y with $f(y) = (a - 1, b)$ is a/n. All other transition probabilities are 0.

To complete the proof, notice that we can collapse the stationary distribution π' onto the row vector $\bar{\pi}$, where the kth element of $\bar{\pi}$ is $\sum_{x \in G_k} \pi'_x$. Using the above claim and the fact that $\pi' A = \pi'$, we obtain by calculation that $\bar{\pi} A' = \bar{\pi}$. Therefore, $\bar{\pi}$ is a stationary distribution for the system chain. Since the stationary distribution is unique, $\bar{\pi} = \pi$, which concludes the proof. □

In fact, we can prove that the function $f : \mathcal{S} \to \mathcal{M}$ defined above induces a lifting from the individual chain to the system chain.

Lemma 4. *The system Markov chain is a lifting of the individual Markov chain.*

Proof. Consider a state k in $\mathcal{M}$. Let j be a neighboring state of k in the system chain. The ergodic flow from k to j is $p_{kj}\pi_k$. In particular, if k is given by the tuple (a, b), j can be either $(a+1, b-1)$ or $(a+1, b)$, or $(a-1, b)$. Consider now a state $x \in \mathcal{M}$, $x = (P_0, \ldots, P_n)$, such that $f(x) = k$. By the definition of f, x has a processes in state $Read$, and b processes in state $OldCAS$.

If j is the state $(a + 1, b - 1)$, then the flow from k to j, Q_{kj}, is $b\pi_k/n$. The state x from the individual chain has exactly b neighboring states y which map to the state $(a + 1, b - 1)$, one for each of the b processes in state $OldCAS$ which might take a step. Fix y to be such a state. The probability of moving from x to y is $1/n$. Therefore, using Lemma 3, we obtain that

$$\sum_{x \in f^{-1}(k), y \in f^{-1}(j)} Q'_{xy} = \sum_{x \in f^{-1}(k)} \sum_{y \in f^{-1}(j)} \frac{1}{n} \pi'_x = \frac{b}{n} \sum_{x \in f^{-1}(k)} \pi'_x = \frac{b}{n} \pi_k = Q_{kj}.$$

The other cases for state j follow similarly. Therefore, the lifting condition holds. □

Next, we notice that, since states from the individual chain which map to the same system chain state are symmetric, their probabilities in the stationary distribution must be the same. The proof is straightforward.

Lemma 5. *Let x and x' be two states in $\mathcal{S}$ such that $f(x) = f(y)$. Then $\pi'_x = \pi'_y$.*

We now put together the previous claims to obtain an upper bound on the expected time between two successes for a specific process.

Lemma 6. *Let W be the expected system steps between two successes in the stationary distribution of the system chain. Let W_i be the expected system steps between two successes of process p_i in the stationary distribution of the individual chain. For every process p_i, $W = nW_i$.*

Proof. Let μ be the probability that a step is a success by *some* process. Expressed in the system chain, we have that $\mu = \sum_{j=(a,b)}(1 - (a + b)/n)\pi_j$. Let X_i be the set of states in the individual chain in which $P_i = CCAS$. Consider the event that a system step is a step in which p_i succeeds. This must be a step by p_i from a state in X_i. The probability of this event in the stationary distribution of the individual chain is $\eta_i = \sum_{x \in X_i} \pi'_x/n$.

Recall that the lifting function f maps all states x with a processes in state $Read$ and b processes in state $OldCAS$ to state $j = (a, b)$. Therefore, $\eta_i = (1/n) \sum_{j=(a,b)} \sum_{x \in f^{-1}(j) \cap X_i} \pi'_x$. By symmetry, we have that $\pi'_x = \pi'_y$, for every states $x, y \in f^{-1}(j)$. The fraction of states in $f^{-1}(j)$ that have p_i in state $CCAS$ (and are therefore also in X_i) is $(1 - (a + b)/n)$. Therefore, $\sum_{x \in f^{-1}(j) \cap X_i} \pi'_x = (1 - (a + b)/n)\pi_j$.

We finally get that, for every process p_i, $\eta_i = (1/n) \sum_{j=(a,b)}(1 - (a + b)/n)\pi_j = (1/n)\mu$. On the other hand, since we consider the stationary distribution, from a straightforward extension of Theorem 1, we have that $W_i = 1/\eta_i$, and $W = 1/\mu$. Therefore, $W_i = nW$, as claimed. □

5.1.3 System Latency Bound

In this section we provide an upper bound on the system latency. We prove the following.

Theorem 5. *The expected number of steps between two successes in the system chain is* $\Theta(\sqrt{n})$.

An iterated balls-into-bins game. To bound W, we model the evolution of the system as a balls-into-bins game. We will associate each process with a bin. At the beginning of the execution, each bin already contains one ball. At each time step, we throw a new ball into a uniformly chosen random bin. Essentially, whenever the process takes a step, its bin receives an additional ball. We continue to distribute balls until the first time a bin acquires *three* balls. We call this event a *reset*. When a reset occurs, we set the number of balls in the bin containing three balls to one, and all the bins containing two balls become empty. The game then continues until the next reset.

This game models the fact that initially, each process is about to read the shared state. To change its value, it must take two steps without the state changing in between. A process which changes the shared state by CAS-ing successfully causes all other processes which were about to CAS with the correct value to fail their operations. These processes now need to take *three* steps to change the shared state. We therefore reset the number of balls in the corresponding bins to 0. We define the game in terms of *phases*. A phase is the interval between two resets. For phase i, we denote by a_i the number of bins with one ball at the beginning of the phase, and by b_i the number of bins with 0 balls at the beginning of the phase. Since there are no bins with two or more balls at the start of a phase, we have that $a_i + b_i = n$.

Notice that this iterated game evolves in the same way as the system Markov chain. In particular, W is the expected length of a phase. To prove Theorem 5, we first obtain a bound on the length of a phase in terms of the parameters a_i and b_i.

Lemma 7. *Let* $\alpha \geq 4$ *be a constant. The expected length of phase* i *is at most* $\min(2\alpha n/\sqrt{a_i}, 3\alpha n/b_i^{1/3})$. *The phase length is* $2\alpha \min(n\sqrt{\log n}/\sqrt{a_i}, n(\log n)^{1/3}/b_i^{1/3})$, *with probability at least* $1 - 1/n^{\alpha}$. *The probability that the length of a phase is less than* $\min(n/\sqrt{a_i}, n/(b_i)^{1/3})/\alpha$ *is at most* $1/(4\alpha^2)$.

Proof. Let A_i be the set of bins with one ball, and let B_i be the set of bins with zero balls, at the beginning of the phase. We have $a_i = |A_i|$ and $b_i = |B_i|$. Practically, the phase ends either when a bin in A_i or a bin in B_i first contains three balls.

For the first event to occur, some bin in A_i must receive two additional balls. Let $c \geq 1$ be a large constant, and assume for now that $a_i \geq \log n$ and $b_i \geq \log n$ (the other cases will be treated separately). The number of bins in A_i which need to receive a ball before some bin receives two new balls is concentrated around $\sqrt{a_i}$, by the birthday paradox. More precisely, the following holds.

Claim 1. *Let* X_i *be random variable counting the number of bins in* A_i *chosen to get a ball before some bin in* A_i *contains three balls, and fix* $\alpha \geq 4$ *to be a constant. Then the expectation of* X_i *is less than* $2\alpha\sqrt{a_i}$. *The value of* X_i *is at most* $\alpha\sqrt{a_i \log n}$, *with probability at least* $1 - 1/n^{\alpha^2}$.

Proof. We employ the Poisson approximation for balls-into-bins processes. In essence, we want to bound the number of balls to be thrown uniformly into a_i bins until two balls collide in the same bin, in expectation and with high probability. Assume we throw m balls into the $a_i \geq \log n$ bins. It is well-known that the number of balls a bin receives during this process can be approximated as a Poisson random variable with mean m/a_i (see, e.g., [14]). In particular, the probability that no bin receives two extra balls during this process is at most

$$2\left(1 - \frac{e^{-m/a_i}(\frac{m}{a_i})^2}{2}\right)^{a_i} \leq 2\left(\frac{1}{e}\right)^{\frac{m^2}{2a_i}e^{-m/a_i}}.$$

If we take $m = \alpha\sqrt{a_i}$ for $\alpha \geq 4$ constant, we obtain that this probability is at most

$$2\left(\frac{1}{e}\right)^{\alpha^2 e^{-\alpha/\sqrt{a_i}}/2} \leq \left(\frac{1}{e}\right)^{\alpha^2/4},$$

where we have used the fact that $a_i \geq \log n \geq \alpha^2$. Therefore, the expected number of throws until some bin receives two balls is at most $2\alpha\sqrt{a_i}$. Taking $m = \alpha\sqrt{a_i \log n}$, we obtain that some bin receives two new balls within $\alpha\sqrt{a_i \log n}$ throws with probability at least $1 - 1/n^{\alpha^2}$. □

We now prove a similar upper bound for the number of bins in B_i which need to receive a ball before some such bin receives three new balls, as required to end the phase.

Claim 2. *Let Y_i be random variable counting the number of bins in B_i chosen to get a ball before some bin in B_i contains three balls, and fix $\alpha \geq 4$ to be a constant. Then the expectation of Y_i is at most $3\alpha b_i^{2/3}$, and Y_i is at most $\alpha(\log n)^{1/3} b_i^{2/3}$, with probability at least $1 - (1/n)^{\alpha^3/54}$.*

Proof. We need to bound the number of balls to be thrown uniformly into b_i bins (each of which is initially empty), until some bin gets three balls. Again, we use a Poisson approximation. We throw m balls into the $b_i \geq \log n$ bins. The probability that no bin receives three or more balls during this process is at most

$$2\left(1 - \frac{e^{-m/a_i}(m/b_i)^3}{6}\right)^{b_i} = 2\left(\frac{1}{e}\right)^{\frac{m^3}{6b_i^2}e^{-m/b_i}}.$$

Taking $m = \alpha b_i^{2/3}$ for $\alpha \geq 4$, we obtain that this probability is at most

$$2\left(\frac{1}{e}\right)^{\frac{\alpha^3}{6}e^{-\alpha/b_i^{1/3}}} \leq \left(\frac{1}{e}\right)^{\alpha^3/54}.$$

Therefore, the expected number of ball thrown into bins from B_i until some such bin contains three balls is at most $3\alpha b_i^{2/3}$. Taking $m = \alpha(\log n)^{1/3} b_i^{2/3}$, we obtain that the probability that no bin receives three balls within the first m ball throws in B_i is at most $(1/n)^{\alpha^3/54}$. □

The above claims bound the number of steps inside the sets A_i and B_i necessary to finish the phase. On the other hand, notice that a step throws a new ball into a bin from A_i with probability a_i/n, and throws it into a bin in B_i with probability b_i/n. It therefore follows that the expected number of steps for a bin in A_i to reach three balls (starting from one ball in each bin) is at most $2\alpha\sqrt{a_i}n/a_i = 2\alpha n/\sqrt{a_i}$. The expected number of steps for a bin in B_i to reach three balls is at most $3\alpha b_i^{2/3} n/b_i = 3\alpha n/b_i^{1/3}$. The next claim provides concentration bounds for these inequalities, and completes the proof of the Lemma.

Claim 3. *The probability that the system takes more than $2\alpha \frac{n}{\sqrt{a_i}}\sqrt{\log n}$ steps in a phase is at most $1/n^\alpha$. The probability that the system takes more than $2\alpha \frac{n}{b_i^{1/3}}(\log n)^{1/3}$ steps in a phase is at most $1/n^\alpha$.*

Proof. Fix a parameter $\beta > 0$. By a Chernoff bound, the probability that the system takes more than $2\beta n/a_i$ steps without throwing at least β balls into the bins in A_i is at most $(1/e)^\beta$. At the same time, by Claim 1, the probability that $\alpha\sqrt{a_i \log n}$ balls thrown into bins in A_i do not generate a collision (finishing the phase) is at most $1/n^{\alpha^2}$.

Therefore, throwing $2\alpha \frac{n}{\sqrt{a_i}}\sqrt{\log n}$ balls fail to finish the phase with probability at most $1/n^{\alpha^2} + 1/e^{\alpha\sqrt{a_i \log n}}$. Since $a_i \geq \log n$ by the case assumption, the claim follows.

Similarly, using Claim 2, the probability that the system takes more than $2\alpha(\log n)^{1/3} b_i^{2/3} n/b_i = 2\alpha(\log n)^{1/3} n/b_i^{1/3}$ steps without a bin in B_i reaching three balls (in the absence of a reset) is at most $(1/e)^{1+(\log n)^{1/3} b_i^{2/3}} + (1/n)^{\alpha^3/54} \leq (1/n)^\alpha$, since $b_i \geq \log n$. □

The previous results imply that, if $a_i \geq \log n$ and $b_i \geq \log n$, then the expected length of a phase is $\min(2\alpha n/\sqrt{a_i}, 3\alpha n/b_i^{1/3})$. The phase length is $2\alpha \min(\frac{n}{\sqrt{a_i}}\sqrt{\log n}, \frac{n}{b_i^{1/3}}(\log n)^{1/3})$, with high probability.

It remains to consider the case where either a_i or b_i are less than $\log n$. Assume $a_i \geq \log n$. Then $b_i \geq n - \log n$. We can therefore apply the above argument for b_i, and we obtain that with high probability the phase finishes in $2\alpha n(\log n/b_i)^{1/3}$ steps. This is less than $2\alpha \frac{n}{\sqrt{a_i}}\sqrt{\log n}$, since $a_i \leq \log n$, which concludes the claim. The converse case is similar. □

Next, we analyze the dynamics of the phases $i \geq 1$ based on the value of a_i at the beginning of the phase. Fix c a large constant. Phase i is in *the first range* if $a_i \in [n/3, n]$, in the *second range* if $n/c \leq a_i < n/3$, and is in the *third range* if $0 \leq a_i < n/c$. We analyze the probability of moving between ranges by carefully bounding the change in ball counts for the bins during a phase as a function of a_i's initial value.

Lemma 8. *For $i \geq 1$, if phase i is in the first two ranges, then the probability that phase $i+1$ is in the third range is at most $1/n^\alpha$. Let $\beta > 2c^2$ be a constant. The probability that $\beta\sqrt{n}$* consecutive *phases are in the third range is at most $1/n^\alpha$.*

Proof. We first bound the probability that a phase moves to the third range from one of the first two ranges.

Claim 4. *For $i \geq 1$, if phase i is in the first two ranges, then the probability that phase $i+1$ is in the third range is at most $1/n^\alpha$.*

Proof. We first consider the case where phase i is in range two, i.e. $n/c \leq a_i < n/3$, and bound the probability that $a_{i+1} < n/c$. By Lemma 7, the total number of system steps taken in phase i is at most $2\alpha \min(n/\sqrt{a_i}\sqrt{\log n}, n/b_i^{1/3}(\log n)^{1/3})$, with probability at least $1 - 1/n^\alpha$. Given the bounds on a_i, it follows by calculation that the first factor is always the minimum in this range.

Let ℓ_i be the number of steps in phase i. Since $a_i \in [n/c, n/3)$, the expected number of balls thrown into bins from A_i is at most $\ell_i/3$, whereas the expected number of balls thrown into bins from B_i is at least $2\ell_i/3$. The parameter a_{i+1} is a_i plus the bins from B_i which acquire a single ball, minus the balls from A_i which acquire an extra ball. On the other hand, the number of bins from B_i which acquire a single ball during ℓ_i steps is tightly concentrated around $2\ell_i/3$, whereas the number of bins in A_i which acquire a single ball during ℓ_i steps is tightly concentrated around $\ell_i/3$. More precisely, using Chernoff bounds, given $a_i \in [n/c, n/3)$, we obtain that $a_i \geq a_{i+1}$, with probability at least $1 - 1/e^{\alpha\sqrt{n}}$.

For the case where phase i is in range one, notice that, in order to move to range three, the value of a_i would have to decrease by at least $n(1/3 - 1/c)$ in this phase. On the other hand, by Lemma 7, the length of the phase is at most $2\alpha\sqrt{3n\log n}$, w.h.p. Therefore the claim follows. A similar argument provides a lower bound on the length of a phase. □

The second claim suggests that, if the system is in the third range (a low probability event), it gradually returns to one of the first two ranges.

Claim 5. *Let $\beta > 2c^2$ be a constant. The probability that $\beta\sqrt{n}$ phases are in the third range is at most $1/n^\alpha$.*

Proof. Assume the system is in the third range, i.e. $a_i \in [0, n/c)$. Fix a phase i, and let ℓ_i be its length. Let S_b^i be the set of bins in B_i which get a single ball during phase i. Let T_b^i be the set of bins in B_i which get two balls during phase i (and are reset). Let S_a^i be the set of bins in A_i which get a single ball during phase i (and are also reset). Then $b_i - b_{i+1} \geq |S_b^i| - |T_b^i| - |S_a^i|$.

We bound each term on the right-hand side of the inequality. Of all the balls thrown during phase i, in expectation at least $(1 - 1/c)$ are thrown in bins from B_i. By a Chernoff bound, the number of balls thrown in B_i is at least $(1 - 1/c)(1 - \delta)\ell_i$ with probability at least $1 - \exp(-\delta^2\ell_i(1 - 1/c)/4)$, for $\delta \in (0, 1)$. On the other hand, the majority of these balls do not cause collisions in bins from B_i. In particular, from the Poisson approximation, we obtain that $|S_b^i| \geq 2|T_b^i|$ with probability at least $1 - (1/n)^{\alpha+1}$, where we have used $b_i \geq n(1 - 1/c)$.

Considering S_a^i, notice that, w.h.p., at most $(1 + \delta)\ell_i/c$ balls are thrown in bins from A_i. Summing up, given that $\ell_i \geq \sqrt{n}/c$, we obtain that $b_i - b_{i+1} \geq (1 - 1/c)(1 - \delta)\ell_i/2 - (1 + \delta)\ell_i/c$, with probability at least $1 - \max((1/n)^\alpha, \exp(-\delta^2\ell_i(1 - 1/c)/4)$. For small $\delta \in (0, 1)$ and $c \geq 10$, the difference is at least ℓ_i/c^2. Notice also that the probability depends on the length of the phase.

We say that a phase is *regular* if its length is at least $\min(\, n/\sqrt{a_i}, n/(b_i)^{1/3})/c$. From Lemma 7, the probability that a phase is regular is at least $1 - 1/(4c^2)$. Also, in this case, $\ell_i \geq \sqrt{n}/c$, by calculation. If the phase is regular, then the size of b_i decreases by $\Omega(\sqrt{n})$, w.h.p.

If the phase is not regular, we simply show that, with high probability, a_i does not decrease. Assume $a_i < a_{i+1}$. Then, either $\ell_i < \log n$, which occurs with probability at most $1/n^{\Omega(\log n)}$ by Lemma 7, or the inequality $b_i - b_{i+i} \geq \ell_i/c^2$ fails, which also occurs with probability at most $1/n^{\Omega(\log n)}$.

To complete the proof, consider a series of $\beta\sqrt{n}$ consecutive phases, and assume that a_i is in the third range for all of them. The probability that such a phase is regular is at least $1 - 1/(4c^2)$, therefore, by Chernoff, a constant fraction of phases are regular, w.h.p. Also w.h.p., in each such phase the size of b_i goes down by $\Omega(\sqrt{n})$ units. On the other hand, by the previous argument, if the phases are not regular, then it is still extemely unlikely that b_i increases for the next phase. Summing up, it follows that the probability that the system stays in the third range for $\beta\sqrt{n}$ consecutive phases is at most $1/n^\alpha$, where $\beta \geq 2c^2$, and $\alpha \geq 4$ was fixed initially. □

□

Final argument. To complete the proof of Theorem 5, we group the states of the game according to their range: state $S_{1,2}$ contains all states (a_i, b_i) in the first two ranges, i.e. with $a_i \geq n/c$. State S_3 contains all states (a_i, b_i) with $a_i < n/c$. Using Lemmas 7 and 8, we obtain that the collective probability of states in $S_{1,2}$ is at least $1 - \beta\sqrt{n}/n^\alpha$, while the probability of states in S_3 is at most $\beta\sqrt{n}/n^\alpha$. Therefore, the expected length of a phase is at most $2\alpha\sqrt{n}(1-\beta\sqrt{n}/n^\alpha) + \beta n^{2/3}\sqrt{n}/n^\alpha = O(\sqrt{n})$, as claimed. This completes the proof of Theorem 5. Note that, per Lemma 7, this bound is asymptotically tight.

5.2 General Bounds for $SCU(q, s)$

We now put together the results of the previous sections to obtain a bound on individual and system latency. First, we notice that Theorem 5 can be easily extended to the case where the loop contains s scan steps, as the extended state of a process p can be changed by a step of another process $q \neq p$ only if p is about to perform a CAS operation. We obtain that both the system and the individual latency bounds are multiplied by s. We then use the lifting method to analyze the parallel code, i.e. $SCU(q, 0)$. The key observation is that, in this case, the stationary distribution for the individual chain is uniform.

Lemma 9. *For any $1 \leq i \leq n$ and $q \geq 0$, given an algorithm in $SCU(q, 0)$, its individual latency is $W_i = nq$, and its system latency is $W = q$.*

Proof. We examine the stationary distributions of the two Markov chains. Contrary to the previous examples, it turns out that in this case it is easier to determine the stationary distribution of the individual Markov chain M_I. Notice that, in this chain, all states have in- and out-degree n, and the transition probabilities are uniform (probability $1/n$). It therefore must hold that the stationary distribution of M_I is *uniform*. Further, notice that a $1/nq$ fraction of the edges corresponds to the counter of a specific process p_i being reset. Therefore, for any i, the probability that a step in M_I is a completed operation by p_i is $1/nq$. Hence, the individual latency for the algorithm is nc. To obtain the system latency, we notice that, from the lifting, the probability that a step in M_S is a completed operation by *some* process is $1/q$. Therefore, the individual latency for the algorithm is q. □

Clearly, an algorithm in $SCU(q, s)$ is a sequential composition of parallel code followed by s loop steps. Fix a process p_i. By Lemma 9, using linearity of expectation, we obtain that the expected individual latency for process p_i to complete an operation is $O(n(q + s\sqrt{n}))$. We define the individual chain and the system chain for the general algorithm, and show that a lifting exists. This again implies that the system latency is $O(q + s\sqrt{n})$, which completes the proof of Theorem 4.

Notes on the argument. We could have tailored the analysis to consider both components at the same time (via a more complicated iterated balls-into-bins game). However, we believe the modularity of the framework is a potentially useful property, and therefore chose to highlight it.

We also note that the above argument also gives an upper bound on the expected number of (individual) steps a process p_i needs to complete an operation (similar to the standard measure of individual *step complexity*). Since the scheduler is uniform, this is also $O(q + s\sqrt{n})$. Finally, we note that, if only $k \leq n$ processes are correct in the execution, we obtain the same latency bounds in terms of k: since we consider the stationary behavior of the algorithm, the latencies are only influenced by correct processes.

Corollary 1. *Given an algorithm in $SCU(q, s)$ on k correct processes under a uniform stochastic scheduler, the system latency is $O(q + s\sqrt{k})$, and the individual latency is $O(k(q + s\sqrt{k}))$. The expected step complexity of an operation is $O(q + s\sqrt{k})$.*

6 Discussion

This paper is motivated by the fundamental question of relating the theory of concurrent programming to real-world algorithm behavior. We give a framework for analyzing concurrent algorithms which partially explains the wait-free behavior of lock-free algorithms, and their good performance in practice. Our work is a first step in this direction, and opens the door to many additional questions.

In particular, we are intrigued by the goal of obtaining a realistic model for the unpredictable behavior of system schedulers. Even though it has some foundation in empirical results, our uniform stochastic model is a rough approximation, and can probably be improved. We believe that some of the elements of our framework (such as the existence of liftings) could still be applied to non-uniform stochastic scheduler models, while others may need to be further developed. A second direction for future work is studying other types of algorithms, and in particular implementations which export several distict methods. The class of algorithms we consider is *universal*, i.e., covers any sequential object, however there may exist specific implementations which do not fall in this class. Finally, it would be interesting to explore whether there exist concurrent algorithms which avoid the $\Theta(\sqrt{n})$ contention factor in the latency, and whether such algorithms are efficient in practice.

Acknowledgments. The authors would like to thank Mohsen Ghaffari, William Hasenplaugh, Maurice Herlihy, Jon Kelner, and Ronitt Rubinfeld for useful discussions, and Faith Ellen for very helpful comments on an earlier version of this paper. We also thank the anonymous reviewers for their useful comments.

References

[1] Samy Al-Bahra. Nonblocking algorithms and scalable multicore programming. *Commun. ACM*, 56(7):50–61, 2013.

[2] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? *CoRR*, abs/1311.3200, 2013.

[3] James Aspnes. Fast deterministic consensus in a noisy environment. *J. Algorithms*, 45(1):16–39, 2002.

[4] Fang Chen, László Lovász, and Igor Pak. Lifting markov chains to speed up mixing. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, STOC '99, pages 275–281, New York, NY, USA, 1999. ACM.

[5] Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, Montreal, QC, Canada , 2013*, pages 43–52, 2013.

[6] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In *Proceedings of the International Symposium on Distributed Computing*, pages 493–494, 2005.

[7] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[8] D. Guniguntala, P.E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux. *IBM Systems Journal*, 47(2):221–236, 2008.

[9] Thomas P. Hayes and Alistair Sinclair. Liftings of tree-structured markov chains. In *Proceedings of the 13th international conference on Approximation, and 14 the International conference on Randomization, and combinatorial optimization: algorithms and techniques*, APPROX/RANDOM'10, pages 602–616, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[11] Maurice Herlihy and Nir Shavit. On the nature of progress. In *15th International Conference on Principles of Distributed Systems (OPODIS), Toulouse, France, December 13-16, 2011. Proceedings*, pages 313–328, 2011.

[12] David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Society, 2008.

[13] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[14] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[15] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.