Analyzing Contention and Backoff in Asynchronous Shared Memory

Naama Ben-David
Carnegie Mellon University
nbendavi@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

ABSTRACT

Randomized backoff protocols have long been used to reduce contention on shared resources. They are heavily used in communication channels and radio networks, and have also been shown to greatly improve the performance of shared memory algorithms in real systems. However, while backoff protocols are well understood in many settings, their effect in shared memory has never been theoretically analyzed. This discrepency may be due to the difficulty of modeling asynchrony without eliminating the advantage gained by local delays.

In this paper, we introduce a new cost model for contention in shared memory. Our model allows for adversarial asynchrony, but also provides a clear notion of time, thus enabling easy calculation of contention costs and delays. We then consider a simple use case in which n processes try to update a single memory location. Using our model, we first show that a naïve protocol, without any backoff, requires $\Omega(n^3)$ work until all processes successfully update that location. We then analyze the commonly used exponential delay protocol, and show that it requires $\Theta(n^2 \log n)$ work with high probability. Finally, we show that the exponential delay protocol is suboptimal, by introducing a new backoff protocol based on adaptive probabilities and showing that, for the same use case, it requires only $O(n^2)$ work with high probability.

ACM Reference format:
Naama Ben-David and Guy E. Blelloch. 2017. Analyzing Contention and Backoff in Asynchronous Shared Memory. In *Proceedings of PODC’17, July 25-27, 2017, Washington, DC, USA, ,* 10 pages.
DOI: 10.1145/3087801.3087828

1 INTRODUCTION

Exponential backoff protocols are commonly used in many settings to avoid contention on a shared resource. They appear in transactional memory [27], communication channels [9, 10, 14, 15], radio and wireless networks [6, 8], local area networks [25], and even shared memory algorithms [3, 22]. Many backoff protocols have been developed and proved efficient for these communication tasks. The general setting is that many processes want to send a message over a channel (or a network), but the transmission fails if more than one message is sent at the same time. Usually, transmissions are assumed to happen in synchronous rounds, and time is measured by the number of rounds until all messages are successfully transmitted. A round is wasted if more than one process attempts to transmit, but the cost remains constant regardless of how many processes attempted to transmit in that round.

The situation is different in a shared memory setting. If many processes attempt to access the same memory location, they all suffer *contention*, which slows down their response time, and becomes worse as the number of conflicting processes increases. For example, if P processes all access the same memory location at the same time, this can require $\Theta(P)$ ‘wait time’ for each process, for a total of $\Theta(P^2)$ total work. Several theoretical models have been developed to try to capture these costs [12, 13, 17, 18, 21]. However, none of these models have been applied to analyze backoff protocols. The reason is, at least in part, that these models are not well suited for the task. In general, shared memory contention and performance is more difficult to model than channel communication because of the inherent asynchrony in shared memory systems.

Asynchrony in shared memory algorithms is usually modeled by assuming a powerful adversary that determines the schedule of instructions. Some models capturing contention, like those of Dwork, Herlihy, and Waarts [12] and Fich, Hendler and Shavit [13], give too much power to the adversary to be able to get any meaningful worst case bounds on backoff protocols. The model of Dwork *et al.* assumes every instruction has an invocation and a response, and each instruction suffers contention equivalent to the number of responses to other instructions that occurred between its own invocation and response. Fich *et al.* assign cost differently; in particular, they assume only modifying instructions that immediately precede a given instruction i can have an effect on i’s running time. While these models are reasonable for obtaining lower bounds, they are impractical for attaining meaningful upper bounds. Because of the great power the adversary is given, the advantage of backoff protocols is lost in these models.

Further work by Hendler and Kutten [21] restricts the power of the adversary in an attempt to capture more likely executions. They present a model similar to that of Dwork *et al.* [12], but add the notion of k-synchrony on top of it. This requirement states that an execution is k-synchronous if in any subexecution, no process can do $k + 1$ operations before all other processes do at least one. This property, while attractively simple, does not succeed in capturing all possible real executions. Asynchrony in real systems often arises from processes being swapped out. In such cases, one process can be delayed a great deal with respect to the others, and no reasonable value of k allows for such delays.

PODC’17, July 25-27, 2017, Washington, DC, USA
© 2017 ACM. 978-1-4503-4992-5/17/07…$15.00
DOI: 10.1145/3087801.3087828

All of these models have an additional problem: there is no clear notion of time. This further complicates the analysis of time-based protocols such as exponential delay, where processes delay themselves for increasingly longer amounts of time.

A different line of work aiming to create a more realistic model for shared memory has focused on relaxing the adversarial scheduler to a stochastic one, leading to tractable analysis of the step complexity of some lock-free algorithms [1, 2]. However, this line of work also relies on strong assumptions on the behavior of the scheduler. Furthermore, it has not directly accounted for contention, rendering the analysis of backoff under this model meaningless.

In spite of the difficulty of theoretically analyzing backoff, these protocols have been shown to be effective in combating shared memory contention. Many systems benefit greatly from the application of an exponential backoff protocol for highly contented memory [3, 19, 22, 24, 26]. For about 30 years, exponential delay has been successfully used in practice to significantly reduce running time in algorithms that suffer from contention.

In this paper, we take a first step towards closing this gap between theory and practice. We define a model for analyzing contention in shared memory protocols, and use it to analyze backoff. Our model allows for asynchrony, but maintains a clear notion of time. We do this by giving the adversary full power on one part of the processes' delay, but taking it away in another part. In particular, we allow each process to have up to one *ready* instruction, and the adversary determines when to invoke, or *activate*, each ready instruction. However, once active, instructions have a priority order that the adversary can no longer control. This models the difference between delays caused by the system between instructions of a process (e.g. caused by an interrupt), and delays incurred after the instruction has been invoked in the hardware (assuming an instruction cannot be interrupted midstream). We then define *time steps*, in which all active instruction attempt to execute. Furthermore, we define *conflicts* between instructions; two conflicting instructions cannot be executed in the same time step. An active instruction will be executed in the first time step in which it does not conflict with any higher-priority instruction. In this paper, we say that modifying instructions (such as compare-and-swap) conflict with all other instructions on the same location, but any two non-modifying instructions do not conflict. This reflects the fact that in modern systems, multiple reads to the same location can go through in parallel, but a modifying instruction requires exclusive access to the memory, sequentializing other instructions behind it. This is due to cache coherency protocols.

We then consider a situation in which n processes want to update the same memory location. Such a situation arises often in the implementations of various lock-free data structures such as counters and stacks. We use our model to analyze various protocols that achieve such an update. We first show that a naïve approach, which uses a simple read-modify-write loop with no backoff, can lead to a total of $\Omega(n^3)$ work, while a classic exponential delay protocol improves that to $\Theta(n^2 \log n)$. To the best of our knowledge, this is the first analysis of exponential delay for shared memory. Perhaps surprisingly, we show that this protocol is suboptimal. We propose a different protocol, based on adapting probabilities, that achieves $O(n^2)$ work.

The key to this improvement stems from using read operations to get a more accurate estimate of the contention that the process is facing. Recall that in our model, reads can go through in parallel, while modifying instructions conflict with others. In our protocol, instead of delaying for increasingly long periods of time when faced with recurring contention, a process becomes increasingly likely to choose to read rather than write. Intuitively, reading the value of a memory location allows a process to gauge the level of contention it's facing by comparing the new value to the previous value it read. Each process keeps a write-probability that it updates after every operation. It uses this probability to determine its next instruction; i.e. whether it will attempt a compare-and-swap or only read. If the value of the memory does not change between two consecutive reads, the process raises its write-probability. Otherwise, the process lowers it. This constant feedback, and the possibility of raising the probability back up when necessary, is what allows this protocol to do better than exponential delay.

In summary, the contributions of this paper are as follows: (1) We define a new time-based cost model to account for contention in shared memory; (2) We provide the first analysis of the commonly used exponential delay protocol; and (3) We propose a new backoff protocol that is asymptotically better than the classic one.

2 MODEL

Our model aims to separate delays caused by contention from other delays that may occur in shared memory systems. In most known shared memory models, all types of delays (or causes of asynchrony) are grouped into one, and are treated by assuming an adversary that controls the order of execution. However, we note that once an instruction is successfully invoked in the hardware, its performance, while it could greatly vary because of contention, is relatively predictable. Throughout this paper, we only consider instructions that are applied on shared memory.

Thus, we say that each process $p \in P$ may have up to one pending instruction $I_t(p)$ at any time t. Each pending instruction may either be *ready* or *active*. The ready instructions represent instructions that are ready to run, but have not yet been invoked in the hardware; they might be delayed because the calling process is swapped out, failed or otherwise delayed by the system. Active instructions are instructions that have already started executing in the hardware, and are now delayed by contention (e.g. executing the cache coherency protocol). The adversary controls when each instruction moves from ready to active, and then instructions are moved from active to 'done' as soon as they do not contend with any earlier instruction, not under the control of the adversary.

We thus divide the set of pending instructions at time t, denoted I_t, into two subsets; an unordered set of *ready* instructions R_t and an ordered set of *active* instructions A_t, such that $A_t \cup R_t = I_t$ and $A_t \cap R_t = \emptyset$. There is a priority order, denoted $<$ (less is higher priority), among instructions in A_t that corresponds to the order in which they were invoked in the hardware. Furthermore, to model contention, we define *conflicts* between instructions. For any pair of operations $i, j \in I_t$, the predicate $C(i, j)$ is *true* if i conflicts with j, and is *false* otherwise. For the purpose of this paper, we say that $C(i, j)$ is true if and only if i and j both access the same memory location and at least one of them is a modifying instruction. This function can also represent other types of conflicts, such as false sharing, depending on the behavior of the system we want to model.

In each time step t, each instruction may undergo at most one of the following transitions: it may be moved from R_t to A_{t+1}, it may be removed from A_t, or it may be inserted into R_{t+1}. More specifically, the adversary chooses a (possibly empty) subset $S_t \in R_t$ to activate. It gives a priority order among the instructions in S_t, and places them in priority order behind elements already in A_t. Furthermore, in time step t, we define the set of *executed* instructions to be $E_t = \{i \in A_t | \nexists j \in A_t.C(i,j) \wedge j < i\}$. That is, an instruction $i \in A_t$ is executed in time t if and only if there is no other instruction in A_t that conflicts with i and is ahead of i in the priority order. In addition, every process $p \in P$ may add a new instructionto R_{t+1} as long as there is only one instruction per process in I_{t+1}. The set of newly ready instructions is called N_t. The next time step, $t + 1$ has the following sets: $R_{t+1} = (R_t \setminus S_t) \cup N_t$, $A_{t+1} = (A_t \setminus E_t) \cup S_t$ and $I_{t+1} = (I_t \setminus E_t) \cup N_t = R_{t+1} \cup A_{t+1}$. Furthermore, for all instructions $i \in S_t$ and all instructions $j \in A_t$, $j < i$.

An *execution E* is a sequence of the instructions executed by processes in the system, which defines a total order on the instructions. This total order must be an extension of the partial order defined by time steps; that is, for any two instructions i and j, executed in time t and t' respectively, if $t < t'$ then i appears before j in the execution E. $E|p$ is the projection of E onto the process p; that is, it is the subsequence of E that contains only instructions by process p, and no other instruction.

We consider two measurements of efficiency for algorithms evaluated by this model. One measurement is immediate: we say the running time of an execution is the number of time steps taken until termination. Another measurement is called *work*. Intuitively, this measures the total amount of cycles that all processes spent on the execution. We say that a process spends one cycle for every time step t in which $I_t(p) \in A_t$. Thus, the amount of work taken by an execution is given by the sum over all time steps t of the size of A_t (i.e. $W = \sum_t |A_t|$). All of the bounds in this paper are given in terms of work. We feel that work better represents the demands of these protocols on the resources of the machine, as inactive processes could be asleep or making progress on some unrelated task.

Adversary. We assume an oblivious adversary [5]. That is, the adversary does not see processes' local values before making them active, and thus cannot make decisions based on that knowledge. Such values include the results of any random coin flips and the type of operation that the process will perform. The adversary can, however, know the algorithm that the processes are following and the history of the execution so far. We also assume that once an instruction is made active, the adversary may see its local values. However, at that point, the adversary is no longer allowed to make changes to the priorities, and thus cannot use that knowledge.

Protocols. In the rest of this paper, we analyze and compare different backoff protocols. To keep the examples clean and the focus on the backoff protocols themselves rather than where they are used, we consider a simple use case: a read-modify-write update as defined in Algorithm 1. A *read* instruction takes a memory location x, and, if it is executed at time t, will return the value stored in x at t. A *CAS* instruction takes three arguments; a memory location x, an expected value v_e and a new value v_n. As usual, a CAS executed at time t is *successful* if and only if the value stored in x at t is the expected value v_e. In that case, the CAS instruction changes the value in x to the new value v_n. Otherwise, it is a *failed* CAS, and does not change the value stored in x. The CAS returns a boolean indicating whether it was successful.

An *update attempt* by process p is one iteration of Algorithm 1. In every update attempt, p reads the current value, then uses the *Modify* function provided to it to calculate the new value it will use for its CAS instruction. The Modify function can represent any change a process might need to make in an algorithm that uses such updates. A *successful update attempt* is one in which the CAS succeeds and the loop exits. Otherwise, we say that the update attempt *failed.* For simplicity, we assume that every CAS instruction has a distinct v_n, and thus an update attempt fails if and only if there was a successful CAS by a different process between its read and its CAS. We assume that CAS instructions conflict with all other instructions on the same location. However, read instructions do not conflict with each other. Note that such an update protocol is commonly used in the implementation of shared objects such as counters and stacks.

Algorithm 1 Update Loop

```
procedure UPDATE($x$, Modify)
    while $True$ do                    ▷ One iteration is an update attempt
        $currnetVal \leftarrow Read(x)$
        $val \leftarrow$ Modify($currentVal$)
        if $CAS(x, currentVal, val)$ then return
```

3 EXPONENTIAL BACKOFF

Exponential backoff protocols are widely used in practice to improve the performance of shared-memory concurrent algorithms [3, 22, 24]. However, to the best of our knowledge, there has been no running time analysis of these algorithms that shows why they do so well. In this section, we provide the first analysis of the standard exponential delay algorithm for concurrent algorithms.

In general, the exponential backoff protocol works as follows. Each process starts with a maxDelay of 1 (or some other constant), and attempts an update. If it fails, it doubles its maxDelay, and then picks a number d uniformly at random in the range [1, maxDelay]. It then delays for d time units, and attempts another update. We refer to this approach as the *exponential delay* protocol.

Exponential delay is common in the literature and is applied in several settings to reduce running time; for example, Mellor-Crummey and Scott use it for barriers [24], Herlihy uses it between successive attempts of lock-free operations [22], and Anderson used it for spin locks [3]. The pseudo-code for exponential delay on an update protocol is given in Algorithm 2.

Using our model, as defined in section 2, we show that in an asynchronous but timed setting, the exponential delay protocol does indeed significantly improve the running time. In particular, we show that for n processes to each successfully update the same memory location once, the naïve (no-backoff) approach requires $\Theta(n^3)$ work, but using exponential delay, this is reduced to

Algorithm 2 Exponential Delay

```
procedure UPDATE($x$, Modify)
    maxDelay ← 1
    while $True$ do                  ▷ One iteration is an update attempt
        $currentVal \leftarrow Read(x)$
        $val \leftarrow$ Modify($currentVal$)
        if $CAS(x, currentVal, val)$ then return
        maxDelay ← 2 · maxDelay
        $d \leftarrow rand(1,$ maxDelay$)$
        Wait $d$ steps
```

$\Theta(n^2 \log n)$. For simplicity, we assume that each process only needs to successfully write once; after its first successful write, it will not rejoin the ready set. To show the upper bound for exponential delay, we rely on one additional assumption, which is not used in the proofs of the lower bounds.

ASSUMPTION 1. *In any time step t where the active set A_t is empty and the ready set R_t is not, the adversary* must *move a non-empty subset of R_t to A_t.*

Analyzing the naïve protocol. We consider n processes trying to update the same memory location using a naïve update loop with no backoff. The pseudocode of this protocol is given in Algorithm 1. To give a lower bound for this protocol, we imagine a very simple adversarial behavior; whenever any instruction is ready, the adversary immediately activates it. Note that this is not a contrived or improbable adversary. Nevertheless, this scheduling strategy is enough to show a $\Omega(n^3)$ work bound.

The intuition is as follows: the active set has around n instructions at all times. A successful update attempt can only happen about once every n time steps, because CAS attempts always conflict, and the active queue is long. The average process will thus have to make around $n/2$ update attempts before succeeding, and wait around $n/2$ time steps for each one, thus needing $\Omega(n^2)$ work. This gives us the bound of $\Omega(n^3)$ work for all processes together.

THEOREM 3.1. *The naïve update protocol requires $\Omega(n^3)$ work for n processes to each successfully update one location once.*

PROOF. Consider an adversary that simply activates every instruction as soon as it becomes ready, and assume that n instructions are ready at the start. Recall that an update attempt can only succeed if there was no successful CAS by any other process between that update attempt's read and CAS instructions. Furthermore, all CAS instructions conflict with each other, and for a CAS attempt to be successful, there have to be no other successful CAS instructions while it waits in the active set to be executed.

We partition the execution into *rounds*. Each round has exactly one successful CAS in it (the ith successful CAS is in round R_i), and ends immediately after this CAS. The execution ends after round R_n. Note that R_1 is short; all processes start with their read instructions, which takes one time step since reads do not conflict with each other, and in the next time step there is a successful CAS. However, after R_1, all processes that have not yet terminated will have a failed CAS instruction in each round. In particular, every round R_i for $i \geq 2$ has at least $n - i$ failed CAS attempts in it. Furthermore, note that in each round R_i, there are at least $n - i$ instructions in the active set at every time step during R_i. This is because each process immediately returns to the active set with a new instruction.

Recall that the work of an execution is equal to the sum over all time steps of the size of the active set. Thus, we sum over all rounds and all time steps in them, to get

$$W = \sum_{i=1}^{n} R_i = \sum_{i=1}^{n} (n-i)^2 < \sum_{i=1}^{n/2} \left(\frac{n}{2}\right)^2 = \Omega(n^3). \qquad \square$$

3.1 Analyzing Exponential Delay

To analyze the exponential delay protocol, we assume that when a process delays itself for d time, it waits d time steps between leaving the active set and rejoining the ready set.

The lower bound. To show a lower bound of $\Omega(n^2 \log n)$ on the running time of an update loop using the exponential delay protocol, we simply need to present a strategy for the adversary that forces the algorithm to take that long.

THEOREM 3.2. *The standard exponential delay algorithm takes $\Omega(n^2 \log n)$ work for n processes to each successfully update one location once.*

PROOF. Consider an adversary that simply places every operation in the active set as soon as it becomes ready, and assume that n operations are ready at the start. Recall that work is defined as the sum over all time steps of the length of the active set.

For every process p, let M_r be p's maxDelay and D_r be its actual delay after its rth update attempt. Note that the adversary's strategy is to activate p immediately after D_r steps. However, regardless of D_r, in this proof we only start charging work for p's presence in the active set M_r steps after its rth update attempt, or we charge zero if the process already finished after M_r steps. Thus, we actually charge for less work than is being done in reality.

Note that at time step t, there will be $n - d_t$ processes in the active set, where d_t is the number of processes that are done by time t or are delayed at time t. Then note that any single process p can only do at most one update attempt per n time steps. This is because all update attempts contain a CAS, and thus conflict with each other under our model. Furthermore, after p executes one update attempt, it must delay for maxDelay time steps, and then wait behind every other instruction in the active set. The active set starts with n instructions at the beginning of the execution. This also means that successful CAS instructions happen at most once every n time steps.

Every time a process executes an update attempt, assuming it does not terminate, it doubles its maxDelay. As noted above, for every process p, this happens about once every n time steps. As delays grow, there are proportionally many processes that are delayed at any given time t. We can model this as $d_t \leq 2^{\lceil t/n \rceil + 1}$. We now calculate the amount of work W this execution will take.

$$\begin{aligned} W &= \sum_t |A_t| = \sum_t \max\{n - d_t, 0\} \\ &\geq \sum_t \max\{n - 2^{\lceil t/n \rceil + 1}, 0\} \\ &= n \cdot (n-4) + n \cdot (n-8) + \ldots + n \cdot (n-n) \\ &= \Omega(n^2 \log n). \qquad \square \end{aligned}$$

The upper bound. Recall that for the purpose of this analysis, we use Assumption 1. Given this assumption, we show that once every process's max delay reaches $12n$, the algorithm terminates within $O(n \log n)$ work w.h.p.. This is because sufficiently many time slots are empty to allow quick progress. Thus, the adversary can only expand the work while not all processes' maxDelays have reached at least $12n$. We show that the adversary cannot prevent this from happening for more than $O(n^2 \log n)$ work w.h.p..

Lemma 3.3. *Consider n processes attempting to update the same memory location, using the exponential delay algorithm, and assume all of them have a max delay of at least S by some time t. Then in any segment of length S starting after time t in the execution, there will be at most 2n update attempts in expectation and w.h.p..*

Proof. For each process p, let $R(p,k)$ be the event that process p was ready with a new update attempt k times during the segment.

Then the probability that p will attempt k updates in the segment is bounded by:

$$P[R(p,k)] \leq \prod_{j=1}^{k} \frac{S}{2^{j-1} \cdot S} = \prod_{j=1}^{k} \frac{1}{2^{j-1}} = \frac{1}{2^{k(k-1)/2}}$$

The above calculation assumes that every time p attempts to execute and then restarts its delay, it gets to start over at the beginning of the segment. This assumption is clearly too strong, and means that in reality the probability of attempting k updates within one segment decays even faster as k grows. However, for our purposes, this bound suffices. Furthermore, note that the probability that any single process p is ready at least $2 \cdot (\sqrt{\log n} + 1)$ times is:

$$P[R(p, 2 \cdot (\sqrt{\log n} + 1))] \leq \frac{1}{2^{(\sqrt{\log n}+1)\sqrt{\log n}}} < \frac{1}{n}.$$

Thus, no process will be ready in any one segment more than $O(\sqrt{\log n})$ times w.h.p.. Let N be the number of update attempts by all the processes during the segment. We can calculate the expected number of update attempts in the segment as follows:

$$E[N] < \sum_{k=1}^{S} n \cdot P[R_i(p,k)] = n \sum_{k=1}^{S} \frac{1}{2^{k(k-1)/2}} < \frac{7}{4} n$$

Using Hoeffding's inequality, we can similarly bound the number of update attempts in the segment with high probability:

$$P(X - E[X] \geq \delta) \leq \exp\left(-\frac{2n^2\delta^2}{\sum_{i=1}^{n}(\max_i - \min_i)^2}\right)$$
$$P(X \geq \frac{15}{8}) \leq \exp\left(-\frac{n^2}{32n(\sqrt{\log n})^2}\right)$$
$$= \exp\left(-\frac{n}{32 \log n}\right)$$

Therefore, the number of update attempts ready by processes with max delay at least S during any segment of length S is bounded by $\frac{15}{8} n < 2n$ in expectation and with high probability. □

Lemma 3.4. *Consider n processes attempting to update the same memory location, using the exponential delay algorithm. Within $O(n \log n)$ time steps and $O(n^2 \log n)$ work, all processes will have a max delay of at least $12n$, with high probability.*

Proof. We partition the set of processes P into three subsets – small-delay, large-delay and done – according to the processes' state in the execution, and update these sets as the execution proceeds.

For a given time step t, let S_t be the set of *small-delay* processes, whose max delay is small, i.e. maxDelay $< 12n$. Similarly, let L_t be the set of *large-delay* processes whose, max delay is large, i.e. maxDelay $\geq 12n$, and let D_t be the set of processes that are *done* (i.e., have already CASed successfully and terminated by time t). Note that $S_0 = P$, $L_0 = D_0 = \emptyset$, that is, at the beginning of the execution, all processes have a small max delay. Furthermore, by definition, the execution ends on the first time step t such that $D_t = P$ and $S_t = L_t = \emptyset$.

We want to bound the amount of work in the execution until the first time step t such that $L_t \cup D_t = P$. To do so, we need to show that small-delay processes make progress regularly. That is, we need to show that the adversary cannot prevent small-delay processes from raising their delays by keeping them in the ready set for long. By Assumption 1, if only small-delay processes are in the ready set R_t at some time t, then the adversary must activate at least one of them. Thus, we show that in any execution, a constant fraction of the time steps have no large-delay processes ready.

We split up the execution into segments of length $12n$ time steps each. By Lemma 3.3, in each such segment S_t that starts at time t, there are at most $2|L_t|$ update attempts by processes in L_t with high probability. Thus, even if $|L_t| = \Theta(n)$, in every segment of size $12n$, there is a constant fraction of time steps (at least $10n$ with high probability) in which no process in L_t is ready. Furthermore, note that since their max delays are $< 12n$, every process in S_t must be ready at least once in every segment. Recall that the adversary must make at least one instruction active in every step in which there is at least one instruction that is ready. Thus, allowing for processes in S_t to 'miss' the empty time steps in one segment, we have the following progress guarantee: at least once every two segments of length $12n$, at least n instructions of processes with small delays will be executed.

Note that every process needs to execute $\log 12n$ instructions in order to have a large delay. Therefore, for all processes to arrive at a large delay, we need $n \log 12n$ executions of instructions that belong to processes with a small delay. By the above progress guarantee, we know that every $24n$ time steps, at least n such instructions are executed. Thus, in at most $24n \log 12n$ time steps, all processes will reach a delay of at least $12n$, with high probability. Note that in every time step t, there can be at most n instructions in the active set, A_t. Thus, the total work for this part of the execution is bounded by $W = \sum_t^{24n \log 12n} |A_t| = O(n^2 \log n)$ with high probability. □

Lemma 3.5. *Consider n processes attempting to update the same memory location, using the exponential delay algorithm. If the max delay of every process is at least $12n$, then the algorithm will terminate within $O(n \log n)$ additional work with high probability.*

Proof. We can apply the analysis of linear probing in hash tables [23] to this problem. Every process has to find its own time slot in which to execute its instruction, and if it collides with another process on some time slot, it will remain in the active set, and try again in the next one.

We think of the update attempts of the processes as elements being inserted into a hash table, and the time slots as the bins of the hash table. If a process's instruction collides with another's then it will 'probe' the next time slot, until it finds an empty slot in which to execute. Counting the number of probes that each instruction does gives us exactly the amount of work done in the execution.

For the purpose of this analysis, we assume that an update attempt succeeds if and only if neither its read nor its CAS collide with any other instruction. If either one of them does, we assume that this process will have to try again. Furthermore, since each update attempt involves two instructions, we put two time slots in a bin; that is, we assume the number of bins is half of the minimum max delay of the processes (in this case, we know all processes have max delay $\geq 12n$). Thus, we effectively double the load factor in our analysis to account for the two instructions of the update attempts.

Thus, if we have $k \leq n$ update attempts, and all of their processes have max delay at least m, then the maximum amount of work for one update attempt of any process is at most the number of probes an element would require to be inserted into a hash table with load factor $\alpha = 2k/m$.

We start with n processes participating, all of them having a max delay of at least $12n$. By Lemma 3.3, we know that there are at most $2n$ update attempts in $12n$ time steps, with high probability. Thus, our load factor is $\alpha = (2 \cdot 2n)/12n = 1/3$. Let X_i be the amount of work that process p_i does for one update attempt. Let $X = \frac{1}{n}\sum_i E[X_i]$. It is well known [23] that in this case, the expected number of probes an insertion does in linear probing is at most

$$E[X] = \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right) = \frac{1}{2}\left(1 + \frac{9}{4}\right) = \frac{13}{8}.$$

Thus, by Markov's inequality,

$$P[X \geq \frac{15}{8}] \leq \frac{8E[X]}{15} = \frac{13}{15}.$$

Thus, $X < \frac{15}{8}$ with probability at least $\frac{2}{15}$. Note that the amount of work per process is integral and at least one (when it experiences no collisions). Therefore, a simple averaging argument shows that at least $\frac{1}{8}$ of the processes cost only 1, and thus experience no collisions, with probability at least $\frac{2}{15}$. Any process that experiences no collisions terminates in that step. Therefore, with constant probability, a constant fraction of the processes terminate out of every n update attempts.

We can imagine repeatedly having rounds in which the remaining processes are inserted into a hash table of size. Note that the expected cost for n processes to do one update attempt is at most $\frac{16n}{13}$. Furthermore, note that this cost is in reality even better, since process delays increase and the number of inserted instructions decreases, both causing less collisions as the load factor α decreases. However, even if we do not take the decreasing load factor into account, a constant factor of the processes terminate in every such 'round'. Note that since we do not account for the improvements from the previous rounds when calculating the expected cost of round i, the rounds are independent. Thus, we can apply the Chernoff bound to show that within $O(\log n)$ rounds, all processes successfully update, costing a total of $O(n \log n)$ work with high probability. ◻

The lemmas above immediately lead to the following theorem.

Theorem 3.6. *The standard exponential delay algorithm takes $O(n^2 \log n)$ work with high probability for n processes to each successfully update the same location once.*

We can further strengthen this result by showing that the time that processes spend delaying does not asymptotically increase the work of the algorithm. That is, even if we account for process delays as part of the total work, exponential delay still achieves $O(n^2 \log n)$ work in expectation. Due to lack of space, we leave this to the full version of the paper.

4 A NEW APPROACH: ADAPTIVE PROBABILITY

We present a new protocol that beats the standard exponential delay protocol under our model. The pseudocode for our algorithm is given in Algorithm 3. We say that every iteration of the while loop executed by process p is an update attempt, or a *step* by p.

Each process maintains a probability *prob*, which it updates in every step. All processes start with $prob = 1$. Each process also keeps track of the value *val* that was returned by its most recent read. On each step, p tries to CAS with probability *prob* into location $x$and if successful it is done. Otherwise it re-reads the value at x and if changed (indicating contention) it halves its probability, and if not changed it doubles its probability. Intuitively, this protocol lets processes receive more frequent feedback on the state of the system than exponential delay (by reading the contended memory), and hence processes can adjust to different levels of contention faster. Furthermore, 'backing-on' allows processes to recover more quickly from previous high loads. Since reads do not conflict with each other in our model, reads between CASes are coalesced into a single time step, and do not significantly affect work, as our analysis will show. In particular, we show that for n processes to successfully update a single memory location following our protocol, it takes $O(n^2)$ work in expectation and with high probability.

Algorithm 3 Adaptive Probability

```
procedure UPDATE($x$, Modify)
    $currentVal \leftarrow Read(x)$
    $prob \leftarrow 1$
    while $True$ do                ▷ One iteration is an update attempt
        $heads \leftarrow flip(prob)$
        if $heads$ then
            $val \leftarrow$ Modify($currentVal$)
            if $CAS(x, currentVal, val)$ then return
        $newVal \leftarrow Read(x)$
        if $newVal = currentVal$ then
            $prob \leftarrow max(1, 2 \cdot prob)$
        else
            $prob \leftarrow prob/2$
            $currentVal = newVal$
```

4.1 Analysis

The most important factor for understanding the behaviour of our algorithm is the CAS-probabilities of the processes. These probabilities determine the expected number of reads and CAS attempts each

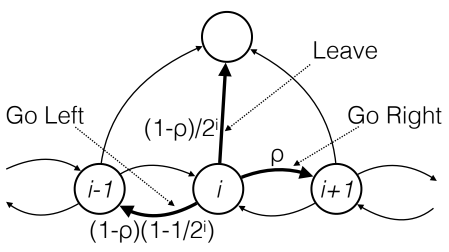

Figure 1: Simple state transition system for one process's execution. The transitions p can make are labeled with their name and their transition probability. The probability ρ is determined by the adversary.

process does during the execution, and thus they greatly influence the overall work.

We first analyze an execution for a single process in isolation, and then consider the progress of the execution as a whole, when all processes participate. Consider any process $p \in P$ as it executes Algorithm 3. Let E be any legal execution, and $E|p$ be the execution restricted to only p's instructions. For the rest of this discussion, and for the proof of Lemma 4.1, when we discuss numbered steps, we mean p's steps in $E|p$. Recall that a step is defined to be a single iteration of the while loop (or a single update attempt), and thus every step may contain more than one instruction. For every process $p \in P$, we define p's *state* at step t, $s(t)$, to be i if p's probability at step t is 2^{-i}. Note that p's state is never the same in two consecutive steps. Furthermore, as long as p doesn't terminate, p's state can only change by one in every step. That is, if $s(t) = i$, then $s(t+1) = i+1$ or $s(t+1) = i-1$. We can model a process p's states over the execution using a simple state transition system (Figure 1). In addition to these states, we need one state in the chain to represent p terminating.

The transition probabilities between p's probability states depend on the other processes in the system. Note that in our algorithm, p decreases its probability in step t, or transitions to the right, if and only if there has been a successful update by some other process between p's read and its CAS. Otherwise, p's state transition will depend on its own CAS-probability—if it CASes, its successfully updates and terminates, and otherwise it goes left. Thus, p's state transition depends on whether other processes successfully update. This is heavily controlled by the adversary.

Recall that we assume an oblivious adversary. That is, the adversary cannot base its decision of which processes to activate on their local values. In particular, that means that we do not allow the adversary to know the result of a process's coin flips, and thus whether it will read or CAS in the next step. However, we do allow the adversary to observe the instruction being done once that instruction is in the active set, and thus keep track of each process's history to know its state at every time step.

Therefore, the adversary can decide which instructions to activate given the probability states of their process. In particular, when considering a single process p's state transition, the adversary can decide how many processes it activates between two consecutive steps of p and how likely those processes are to successfully update. Thus, we can abstract the adversary's power by saying it can pick any probability ρ for how likely the state will go right, i.e., conflict with another process and decrease its probability. Note that in reality, if all participating processes are using the same protocol, the adversary will be more restricted than this, however despite the power we are giving the adversary this abstraction still allows us to prove the first important lemma of our analysis.

Lemma 4.1. *Every process executes at most 4 CAS attempts in expectation before terminating.*

For this Lemma, we will focus on any one process $p \in P$, and track its movement in the state transition system during a given execution E. In the rest of this section, unless otherwise indicated, all analysis refers to process p and execution E of Algorithm 3.

We denote by $R(i)$ the number of 'right' state transitions of p from state i to state $i+1$ in E, and by $L(i)$ the number of 'left' state transitions of p from i to $i-1$ in E. We first make an observation about these state transitions.

Observation 1. *For every state i, $L(i+1) \leq R(i) + 1$.*

That is, the number of transitions from $i+1$ to i is at most the number of transitions from i to $i+1$, plus 1. This can be easily seen by considering, for every time p transitions from i to $i+1$, how it got to state i. p starts at state 0. Thus, for every state $i > 0$, the first time p reaches state $i+1$ is by going right from state i. However, for every subsequent time p transitions right from i to $i+1$, it must have, at some point, gone left from $i+1$ to i. Thus, the number of times p transitions from i to $i+1$ is at most one more than the number of times it transitions from $i+1$ to i.

Proof of Lemma 4.1. Consider execution $E|p$; that is, the execution E restricted to just the instructions of p, and let C_E be the number of CAS attempts in this execution. We partition $E|p$ into two buckets, and analyze each separately; for every state i, we place the first time p reaches i into the first bucket, denoted *Init*, and place every remaining instruction in the execution into the second bucket, called *Main*. Thus, $C_E = C_I + C_M$, where C_I and C_M correspond to the cost (number of CAS attempts) of *Init* and *Main* respectively.

Note that *Init* corresponds to an execution in which p only goes down in probability. In every state i it reaches, it has probability $\frac{1}{2^i}$ of attempting a CAS, and thus expected cost of $\frac{1}{2^i}$ for that state. Thus, we can easily calculate its expected cost.

$$E[C_I] = \sum_{i=0}^{\infty} 2^{-i} \leq 2.$$

We now consider the *Main* bucket of the execution $E|p$. There are two random variables that affect the steps that p takes. The adversary can affect whether p goes right or not, by picking a value for ρ (see Figure 1). However, if p does not go right, then only p's coin flip determines whether it will go left or leave (this is independent of any choice of the adversary). We will charge the right transitions, that the adversary can affect, against the left-or-leave transition that the adversary cannot. We can do this since for every left transition from a state i in the main bucket, there is at most one corresponding right transition from i when it goes back to the right through state i (there might be none if it terminates at a position to the left).

We define D to be the sequence of times t in *Main* in which p does not go right, and use j to indicate indices into this sequence (to avoid confusion with i for state and t for position in $E|p$). Let $C_{M,j}$ to be the random variable for the remaining cost of *Main*, after the jth step in D, but not including the final successful CAS. If just before the jth step in D, the process is at state i, there is a $c_j = \frac{1}{2^i}$ probability that the process will CAS and terminate. Thus, there is a $1 - c_j$ probability that the execution will continue. We let R_j be the random variable indicating that the future right transition assign to j does a CAS. The expected cost (probability) of CASing when going right from state i is $\frac{1}{2^i}$, but since there might not be a future right transition, $E[R_j] \leq c_j$. We now have the following recurrence for the expectation of $C_{M,j}$:

$$\begin{aligned} E[C_{M,j}] &= E[R_j] + \left(1 - c_j\right) E[C_{M,j+1}] \\ &\leq c_j + \left(1 - c_j\right) E[C_{M,j+1}] \end{aligned}$$

The c_j's represent the results of independent coin flips, and therefore the expectations can be multiplied. We therefore have:

$$\begin{aligned} E[C_{M,1}] &\leq c_1 + (1 - c_1) \cdot E[C_{M,2}] \\ &= c_1 + (1 - c_1)(c_2 + (1 - c_2)(c_3 + (1 - c_3)(\ldots))) \\ &= 1 - (1 - c_1) + (1 - c_1) \cdot c_2 + (1 - c_1)(1 - c_2) \cdot c_3 + \ldots \\ &= 1 - (1 - c_1)(1 - c_2) + (1 - c_1)(1 - c_2) \cdot c_3 + \ldots \\ &= 1 - (1 - c_1)(1 - c_2)(1 - c_3) \ldots \\ &\leq 1 \end{aligned}$$

Recall that in addition to this cost, there is exactly one successful CAS that causes p to terminate, so $C_M = C_{M,1} + 1$. Adding this to the cost of the two buckets of the execution, we can bound the total expected cost of $E|p$ by $E[C_E] = E[C_I + C_M] \leq 2 + 1 + 1 = 4$. □

We now consider the total number of steps that p does in E. Recall that a step, or an update attempt, is defined as an iteration of the loop in Algorithm 3, even if the process doesn't execute a CAS. We first show that the maximum number of steps of p depends on the number of successful updates by other processes. For this purpose, we stop considering $E|p$ in isolation, and instead look at the execution E as a whole.

Lemma 4.2 (Throughput lemma). *For any process p and state i, let t and t' be two times in E at which p executes a read instruction that causes it to transition to state i. Then the number of steps p takes between t and t' is at most $2k$, where k is the number of successful updates in the execution between t and t'.*

Proof. Let $E' \subseteq E$ be a sub-execution where the first and last instructions in E' are by process p, and both bring p to state i. Let k be the number of successful updates that occur during E'.

We say that p *observes* a successful update w if p decreased its probability due to the value written by w. We note that for p to return to a state i, it has to raise its probability exactly as many times as it decreases it, when counting starting at the last time it was at state i. Note that p can only decrease its probability if it observes a new successful update. Since it must change its probability on every step, p raises its probability on every step where it does not observe a new update. Thus, one observed update's effect gets 'canceled out' by one update attempt of p in which it did not observe any successful updates. Clearly, p can observe at most k successful updates during E'. Thus, it can make at most k update attempts raising its probability, for a total of at most $2k$ steps during E'. □

Corollary 4.3. *Every process makes at most $2n$ update attempts during an execution of Algorithm 3, where n is the number of successful updates in the execution.*

Proof. Consider any process $p \in P$ and recall that every process starts the execution at state $i = 0$. At this state, p has probability 1 of attempting a CAS. If, at any time, p is in state 0, and no successful update happens before its next step, then p succeeds and terminates. Note that there are at most $n - 1$ successful updates by other processes during p's execution, since each process only updates once. By Lemma 4.2, if p takes $2n - 2$ steps without terminating, then p is back at state 0, and all other processes have executed successful updates. Thus, p's next step is a CAS that succeeds, since all other processes have terminated. Therefore, p can make at most $2n$ update attempts before it succeeds. □

We are now ready to analyze the entire execution.

Theorem 4.4. *Algorithm 3 takes $O(n^2)$ work in expectation for n processes to each successfully update the same location once.*

Proof. Note that by Corollary 4.3, every process executing Algorithm 3 does at most $O(n)$ instructions until it terminates. Thus, in the entire execution, there are $O(n^2)$ instructions. Furthermore, by Lemma 4.1, only $O(n)$ of these are CAS attempts in expectation. The rest must be read instructions.

Recall that $W = \sum_t |A_t|$, and that for all times t, $|A_t| \leq n$. Furthermore, recall that CAS instructions conflict with every other instruction, but read instructions do not conflict with each other. Thus, a CAS instruction w executed at time t can cause every instruction $i \in A_t$ such that $i <_A w$ to be delayed one time step. There are at most $O(n)$ such delayed instructions per CAS instruction. In addition, any read instruction will be active for exactly one time step if there is no CAS instruction in front of it. Thus, we have: $W = O(n) \cdot O(n) + O(n^2) = O(n^2)$. □

We can extend our results to be with high probability. For this, the key is to show that there is a linear number of CAS attempts with high probability in every execution, and then the rest of the results carry through.

Lemma 4.5. *There are $O(n)$ CAS attempts in any execution with high probability.*

Proof. We first show that, with high probability, no process attempts more than $O(\log n)$ CAS instructions. To do this, we define an indicator variable $X_i = 1$ if p executes a CAS instruction in its ith update attempt, and $X_i = 0$ otherwise. Note that X_i is independent from X_j for all $i \neq j$. Furthermore, let $X = \sum_i X_i$. In Lemma 4.1, we show that $E[X] \leq 4$. We can now use the Chernoff inequality to bound the probability of a single process attempting more than $c \log n$ CAS instructions for any constant c.

$$P[X > (1 + c \log n) \cdot 4] \leq e^{\frac{-4c \log n}{3}} = \frac{1}{n^{4c/3}}$$

Thus, every process attempts at most $O(\log n)$ CAS instructions w.h.p. We can now use this bound on the number of CAS attempts per process when bounding the probability of getting more than a linear number of CAS attempts in the system in any execution.

We can now number the processes, and define, for each process p_i, $Y_i = w_i$ where w_i is the number of CAS attempts p_i executes in E. We assume that for every i, $Y_i \in [0, c \log n]$ for some constant c. Let $\bar{Y} = 1/n \sum_{i=1}^{n} Y_i$. Then by Lemma 4.1, $E[\bar{Y}] = 4$. Plugging this into Hoeffding's inequality, we get

$$P[\bar{Y} \geq 5] \leq e^{\frac{-2n}{c \log^2 n}}$$

Therefore, with high probability, there are at most $5n$ CAS attempts in the execution E. □

We thus have the following theorem.

Theorem 4.6. *Algorithm 3 takes $O(n^2)$ work in expectation and with high probability for n processes to each successfully update the same location once.*

5 OTHER CONTENTION MODELS

Contention has been the subject of a lot of research in the past 25 years. Many researchers noted its effect on the performance of algorithms in shared memory and used backoff to mitigate its cost [3, 19, 22, 24]. Anderson [3] used exponential delay to improve the performance of spin locks. Herlihy [22] used a very similar exponential delay protocol in read-modify-write loops of lock-free and wait-free algorithms. However, all of these studies, while showing empirical evidence of the effectiveness of exponential delay, do not provide any theoretical analysis.

Studying the effect of contention in theory requires a good model to account for its cost. Gibbons, Matias and Ramachandran first introduced contention into PRAM models [17, 18]. They observed that the CRCW PRAM is not realistic, and instead defined the QRQW PRAM. This model allows for concurrent reads and writes, but queues them up at every memory location, such that the cost incurred is proportional to the number of such operations on location. Their model also allows processes to pipeline instructions. That is, processes are allowed to have multiple instructions enqueued on the memory at the same time. Instructions incur a delay at least as large as the length of the queue at the time they join.

Dwork, Herlihy and Waarts [12] defined memory *stalls* to account for contention. In their model, each atomic instruction on a base object has an *invocation* and a *response*. An instruction experiences a stall if, between its invocation and response, another instruction on the same location receives its response. Thus, if many operations access the same location concurrently, they could incur many stalls, capturing the effect of contention. In this model, the adversary is given more power than in ours. In particular, we can imagine our model's 'activation' of an instruction as its invocation. Our model then allows the adversary to control the order of invocations of the instructions, but not their responses. Furthermore, Dwork *et al.*'s model does not distinguish between different types of instructions—all are assumed to conflict if they are on the same location. Thus, our model is similar to a version of Dwork *et al.*'s model that restricts the allowed executions according to instructions' invocation order and only considers conflicts with modifying instructions. However, they still cannot deal with exponential delay since they have no direct measure of time.

Fich, Hendler and Shavit [13] defined a different version of memory stalls, based on Dwork et al.'s work. Their model distinguishes between instructions, or *events*, that may modify the memory and ones that may not. An instruction is only slowed down by *modifying* instructions done by different processes on the same memory location. They show a lower bound on the number of stalls incurred by a single instruction in any implementation of a large class of objects. In our model, we consider the same types of conlicts as Fich *et al.* do. However, Fich *et al.* use the same fully adversarial model as in Dwork *et al.*'s work. Thus, our model allows only a subset of the executions allowed by theirs. However, even given the same execution, the two models do not assign it the same cost. In particular, in Fich *et al.*'s model, the only instructions that stall a given instruction i are the modifying instructions *immediately* preceding it. Thus, if there is at least one read instruction between any constant number of modifying instructions, each will only suffer a constant number of stalls. The same is not true for our model.

Further work in this area was done by Hendler and Kutten [21]. Their work introduces a model that restricts the power of the adversary. They define the notion of *k-synchrony*, which, intuitively, states that no process is allowed to be more than a k factor faster than any other process. They consider a model which assigns a cost to a given linearized execution by considering the dependencies between the instructions. While in their analysis they assume that all instructions on the same location contend, they discuss the possibility of treating read instructions differently than writes.

Atalar *et al.* [4] also address conflicts in shared memory and their effect on algorithm performance. They consider a class of lock-free algorithms whose operations do some local work, and then repeat a 'Read-CAS loop' (similar to our definition of an update attempt) until they succeed. Their proposed model divides conflicts into two types; hardware and logical conflicts. This division is reminiscent of our model's ready vs. active instructions. However, the two classifications do not address the same sources of delays.

Contention has been studied in many settings other than shared memory as well. Bar-Yehuda, Goldreich and Itai [8] applied exponential backoff to the problem of broadcasting a message over multi-hop radio networks. In this setting, there are synchronous rounds in which nodes on a graph may transmit a message to all of their neighbors. A node gets a message in a given round if and only if exactly one of its neighbors transmitted in that round. They show that using exponential backoff, broadcast can be solved exponentially faster than any deterministic broadcast algorithm. Radio networks have since been extensively studied, and many new algorithms using backoff have been developed [7, 16, 20].

Similar backoff protocols have been considered for communication channels. Bender *et al.* [9] consider a simple channel with adversarial disruptions. They show that using a back-off/back-on approach, in which probability of transmission is allowed to rise in some cases, achieves better throughput. This is similar to the approach taken by our adaptive probability protocol. An in-depth study of backoff on multiple access channels is given by Bender *et al.* in [10]. They introduce a new backoff protocol which achieves good throughput, polylog transmission attempts, and robustness to a disruptive adversary. Further work has been done on different varieties of communication channels [6, 11, 14, 15].

The models considered in these papers are similar to each other (with some variations on the initial state of the system, how collisions are handled, and the adversary's power), but differ from shared memory in several important ways. Firstly, these communication networks generally assume synchronous rounds, whereas this is not reasonable for shared memory. Furthermore, unlike shared memory, running time in networks and communication channels is measured by the number of rounds for an algorithm to complete, regardless of how many transmissions are sent per round. However, in shared memory the cost grows with the number of attempted operations. Another difference is the behavior upon collision; in shared memory, one modifying operation still goes through, and slows down the rest, whereas in other communication models, no message can be transmitted in that round. Furthermore, a single 'message' in shared memory can take multiple steps (such as an update that is implemented with two instructions). These differences make it hard to apply results from other models directly to shared memory.

6 CONCLUDING REMARKS

We've presented a new cost model for contention in shared memory. Using insights from this model, we've proposed the adaptive probabilities algorithm and analyzed the work it requires when all processes update the same location. Furthermore, we've shown that in this situation, it beats the classic exponential backoff approach.

We believe that our work is a step in the direction of truly being able to analyze the performance of shared memory algorithms, and leaves many interesting open questions. Although we've only discussed the performance of these protocols on one specific use case, we believe that our results can be generalized to other scenarios. We note that update loops such as those we consider are common in many algorithms; they are used in implementations of many important objects such as locks, counters, and stacks. It would be interesting to see how these results transfer to implementations of lock-free data structures, and their performance with and without backoff mechanisms in place. One could also apply our analysis and adaptive probabilities approach to lock free algorithms that iterate on a more complicated procedure until they succeed. We also believe key ideas from our approach can be applied to early-exit protocols, such as test-and-set, where processes that fail may choose to leave without successfully updating.

7 ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under grant number CCF-1314590, and the Natural Sciences and Engineering Research Canada post graduate scholarship. We thank Mor Harchol-Balter for insightful discussions that led to the proof of Lemma 4.1.

REFERENCES

[1] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? *Journal of the ACM (JACM)*, 63(4):31, 2016.

[2] Dan Alistarh, Thomas Sauerwald, and Milan Vojnović. Lock-free algorithms under stochastic schedulers. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 251–260. ACM, 2015.

[3] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[4] Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. Analyzing the performance of lock-free data structures: A conflict-based model. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 341–355, 2015.

[5] Yonatan Aumann and Michael A Bender. Efficient asynchronous consensus with the value-oblivious adversary scheduler. In *International Colloquium on Automata, Languages, and Programming*, pages 622–633. Springer, 1996.

[6] Baruch Awerbuch, Andrea Richa, and Christian Scheideler. A jamming-resistant mac protocol for single-hop wireless networks. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 45–54. ACM, 2008.

[7] R. Bar-Yehuda, A. Israeli, and A. Itai. Multiple communication in multi-hop radio networks. *SIAM Journal on Computing*, 22(4):875–887, 1993.

[8] Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *Journal of Computer and System Sciences*, 45(1):104–126, 1992.

[9] Michael A Bender, Martin Farach-Colton, Simai He, Bradley C Kuszmaul, and Charles E Leiserson. Adversarial contention resolution for simple channels. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 325–332. ACM, 2005.

[10] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Maxwell Young. How to scale exponential backoff: Constant throughput, polylog access attempts, and robustness. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 636–654. Society for Industrial and Applied Mathematics, 2016.

[11] Michael A Bender, Tsvi Kopelowitz, Seth Pettie, and Maxwell Young. Contention resolution with log-logstar channel accesses. In *STOC*, pages 499–508, 2016.

[12] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM (JACM)*, 44(6):779–805, 1997.

[13] Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM Journal on Computing*, 41(3):519–536, 2012.

[14] Jeremy T Fineman, Seth Gilbert, Fabian Kuhn, and Calvin Newport. Contention resolution on a fading channel. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 155–164. ACM, 2016.

[15] Jeremy T Fineman, Calvin Newport, and Tonghe Wang. Contention resolution on multiple channels with collision detection. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 175–184. ACM, 2016.

[16] Mohsen Ghaffari, Bernhard Haeupler, and Majid Khabbazian. Randomized broadcast in radio networks with collision detection. *Distributed Computing*, 28(6):407–422, 2015.

[17] Phillip B Gibbons, Yossi Matias, and Vijaya Ramachandran. The queue-read queue-write asynchronous PRAM model. In *European Conference on Parallel Processing*, pages 277–292. Springer, 1996.

[18] Phillip B Gibbons, Yossi Matias, and Vijaya Ramachandran. The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, pages 638–648, 1997.

[19] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, 1990.

[20] Bernhard Haeupler and David Wajc. A faster distributed radio broadcast primitive. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 361–370. ACM, 2016.

[21] Danny Hendler and Shay Kutten. Constructing shared objects that are both robust and high-throughput. In *International Symposium on Distributed Computing*, pages 428–442. Springer, 2006.

[22] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *ACM SIGPLAN Notices*, volume 25, pages 197–206. ACM, 1990.

[23] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[24] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[25] Robert M Metcalfe and David R Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[26] Maged M Michael and Michael L Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *journal of parallel and distributed computing*, 51(1):1–26, 1998.

[27] William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005.