The LevelArray: A Fast, Practical Long-Lived Renaming Algorithm

Dan Alistarh
Microsoft Research Cambridge

Justin Kopinsky
MIT

Alexander Matveev
MIT

Nir Shavit
MIT and Tel-Aviv University

***Abstract*—The long-lived renaming problem appears in shared-memory systems where a set of threads need to register and deregister frequently from the computation, while concurrent operations scan the set of currently registered threads. Instances of this problem show up in concurrent implementations of transactional memory, flat combining, thread barriers, and memory reclamation schemes for lock-free data structures.**

In this paper, we analyze a randomized solution for long-lived renaming. The algorithmic technique we consider, called the *LevelArray*, has previously been used for hashing and one-shot (single-use) renaming. Our main contribution is to prove that, in *long-lived* executions, where processes may register and deregister polynomially many times, the technique guarantees *constant* steps on average and $O(\log \log n)$ steps with high probability for registering, *unit* cost for deregistering, and $O(n)$ steps for collect queries, where n is an upper bound on the number of processes that may be active at any point in time. We also show that the algorithm has the surprising property that it is *self-healing*: under reasonable assumptions on the schedule, operations running while the data structure is in a degraded state implicitly help the data structure re-balance itself. This subtle mechanism obviates the need for expensive periodic rebuilding procedures.

Our benchmarks validate this approach, showing that, for typical use parameters, the average number of steps a process takes to register is less than *two* and the worst-case number of steps is bounded by *six*, even in executions with billions of operations. We contrast this with other randomized implementations, whose worst-case behavior we show to be unreliable, and with deterministic implementations, whose cost is linear in n.

I. Introduction

Several shared-memory coordination problems can be reduced to the following task: a set of threads dynamically register and deregister from the computation, while other threads periodically query the set of registered threads. A standard example is memory management for lock-free data structures, e.g. [17]: threads accessing the data structure need to register their operations, to ensure that a memory location which they are accessing does not get freed while still being addressed. Worker threads must register and deregister efficiently, while the "garbage collector" thread queries the set of registered processes periodically to see which memory locations can be freed. Similar mechanisms are employed in software transactional memory (STM), e.g. [3], [16], to detect conflicts between reader and writer threads, in flat combining [20] to determine which threads have work to be performed, and in shared-memory barrier algorithms [21]. In most applications, the time to complete registration directly affects the performance of the method calls that use it.

Variants of the problem are known under different names: in a theoretical setting, it has been formalized as *long-lived renaming*, e.g. [11], [14], [24]; in a practical setting, it is known as *dynamic collect* [17]. Regardless of the name, requirements are similar: for good performance, processes should register and deregister quickly, since these operations are very frequent. Furthermore, the data structure should be space-efficient, and its performance should depend on the contention level.

Many known solutions for this problem, e.g. [17], [24], are based on an approach we call the *activity array*. Processes share a set of memory locations, whose size is in the order of the number of threads n. A thread *registers* by acquiring a location through a test-and-set or compare-and-swap operation, and *deregisters* by resetting the location to its initial state. A *collect* query simply scans the array to determine which processes are currently registered.[1] The activity array has the advantage of relative simplicity and good performance in practice [17], due to the array's good cache behavior during collects.

One key difference between its various implementations is the way the register operation is implemented. A simple strategy is to scan the array from left to right, until the first free location is found [17], incurring *linear* complexity on average. A more complex procedure is to probe locations chosen at random or based on a hash function [2], [8], or to proceed by probing linearly from a randomly chosen location. The expected step complexity of this second approach should be constant on average, and at least logarithmic in the worst case. One disadvantage of known randomized approaches, which we also illustrate in our experiments, is that their worst-case performance is not stable over long

[1]The trivial solution where a thread simply uses its identifier as the index of a unique array location is inefficient, since the complexity of the collect would depend on the size of the id space, instead of the maximal contention n.

executions: while most operations will be fast, there always exist operations which take a long time. Also, in the case of linear probing, the performance of the data structure is known to degrade over time, a phenomenon known as *primary clustering* [22].

It is therefore natural to ask if there exist solutions which combine the *good average performance* of randomized techniques with the of *stable worst-case bounds* of deterministic algorithms.

In this paper, we show that such efficient solutions exist, by proposing a long-lived activity array with sub-logaritmic worst-case time for registering, and stable worst-case behavior in practice. The algorithm, called *LevelArray*, guarantees *constant* average complexity and $O(\log \log n)$ step complexity with high probability for registering, *unit* step complexity for deregistering, and *linear* step complexity for collect queries. Crucially, our analysis shows that these properties are guaranteed over *long-lived* executions, where processes may register, deregister, and collect polynomially many times against an oblivious adversarial scheduler.

The above properties should be sufficient for good performance in long-lived executions. Indeed, even if the performance of the data structure were to degrade over time, as is the case with hashing techniques, e.g. [22], we could rebuild the data structure periodically, preserving the bounds in an amortized sense. However, our analysis shows that this explicit rebuilding mechanism is not necessary since the data structure is "self-healing." Under reasonable assumptions on the schedule, even if the data structure ends up in extremely unbalanced state (possible during an infinite execution), the deregister and register operations running from this state automatically re-balance the data structure with high probability. The self-healing property removes the need for explicit rebuilding.

The basic idea behind the algorithm is simple, and has been used previously for efficient hashing [13] and *one-shot*[2] randomized renaming [6]. We consider an array of size $2n$, where n is an upper bound on contention. We split the locations into $O(\log n)$ *levels*: the first (indexed by 0) contains the first $3n/2$ locations, the second contains the next $n/4$ and so on, with the ith level containing $n/2^i$ locations, for $i \geq 1$. To register, each process performs a *constant* number of test-and-set probes at each level, stopping the first time when it acquires a location. Deregistering is performed by simply resetting the location, while collecting is done by scanning the $2n$ locations.

[2]One-shot renaming [10] is the variant of the problem where processes only register once, and deregistration is not possible.

This algorithm clearly solves the problem, the only question is its complexity. The intuitive reason why this procedure runs in $O(\log \log n)$ time in a one-shot execution is that, as processes proceed towards higher levels, the number of processes competing in a level i is $O(n/2^{2^i})$, while the space available is $\Theta(n/2^i)$. By level $\Theta(\log \log n)$, there are virtually no more processes competing. This intuition was formally captured and proven in [13]. However, it was not clear if anything close to this efficient behavior holds true in the long-lived case where threads continuously register and deregister.

The main technical contribution of our paper is showing that this procedure does indeed work in long-lived polynomial-length executions, and, perhaps more surprisingly, requires no re-building over infinite executions, given an oblivious adversarial scheduler. The main challenge is in bounding the correlations between the processes' operations, and in analyzing the properties of the resulting probability distribution over the data structure's state. More precisely, we identify a "balanced" family of probability distributions over the level occupancy under which most operations are fast. We then analyze sequences of operations of increasing length, and prove that they are likely to keep the data structure balanced, despite the fact that the scheduling and the process input may be correlated in arbitrary ways (see Proposition 3). One further difficulty comes from the fact that we allow the adversary to insert arbitrary sequences of operations between a thread's register and the corresponding deregister (see Lemma 2), as is the case in a real execution.

The previous argument does not preclude the data structure from entering an unbalanced state over an infinite execution. (Since it has non-zero probability, such an event will eventually occur.) This motivates us to analyze such executions as well. We show that, assuming the system schedules polynomially many steps between the time a process starts a register operation and the time it deregisters,[3] the data structure will rebalance itself from an arbitrary initial state, with high probability.

Specifically, in a bad state, the array may be arbitrarily shifted away from this good distribution. We prove that, as more and more operations release slots and occupy new ones, the data structure gradually shifts back to a good distribution, which is reached with high

[3]This assumption prevents unrealistic schedules in which the adversary brings the data structure in an unbalanced state, and then schedules a small set of threads to register and unregister infinitely many times, keeping the data structure in roughly the same state while inducing high expected cost on the threads.

probability after polynomially many system steps are taken. Since this shift must occur from *any* unbalanced state, it follows that, in fact, every state is well balanced with high probability. Finally, this implies that every operation verifies the $O(\log \log n)$ complexity upper bound with high probability.

From a theoretical perspective, the LevelArray algorithm solves non-adaptive long-lived renaming in $O(\log \log n)$ steps with high probability, against an oblivious adversary in polynomial-length executions. The same guarantees are provided in infinite executions under scheduler assumptions. We note that our analysis can also be extended to provide worst-case bounds on the long-lived performance of the Broder-Karlin hashing algorithm [13]. (Their analysis is one-shot, which is standard for hashing.)

The algorithm is *wait-free*. The logarithmic lower bound of Alistarh et al. [7] on the complexity of one-shot randomized adaptive renaming is circumvented since the algorithm is not namespace-adaptive. The algorithm is time-optimal for one-shot renaming when linear space and test-and-set operations are used [6].

We validate this approach through several benchmarks. Broadly, the tests show that, for common use parameters, the data structure guarantees fast registration—less than two probes on average for an array of size $2n$—and that the performance is surprisingly stable when dealing with contention and long executions. To illustrate, in a benchmark with approximately one billion register and unregister operations with 80 concurrent threads, the maximum number of probes performed by *any* operation was *six*, while the average number of probes for registering was around 1.75.

The data structure compares favorably to other randomized and deterministic techniques. In particular, the worst-case number of steps performed is at least an order of magnitude lower than that of any other implementation. We also tested the "healing" property by initializing the data structure in a bad state and running a typical schedule from that state. The data structure does indeed converge to a balanced distribution (see Figure 3); interestingly, the convergence speed towards the good state is higher than predicted by the analysis.

Roadmap. Section II presents the system model and problem statement. Section III gives an overview of related work. We present the algorithm in Section IV. Section V-A gives the analysis of polynomial-length executions, while Section V-B considers infinite executions. We present the implementation results in Section VI, and conclude in Section VII. Due to space constraints, some proofs which would have appeared in Section V have been deferred to full version of this paper.

II. System Model and Problem Statement

We assume the standard asynchronous shared memory model with N processes (or threads) $p_1, \ldots, p_N$, out of which at most $n \leq N$ participate in any execution. (Therefore, n can be seen as an upper bound on the contention in an execution.) To simplify the exposition, in the analysis, we will denote the n participants by $p_1, p_2, \ldots, p_n$, although the identifier i is unknown to process p_i in the actual execution.

Processes communicate through registers, on which they perform atomic read, write, test-and-set or compare-and-swap. Our algorithm only employs test-and-set operations. (Test-and-set operations can be simulated either using reads and writes with randomization [1], or atomic compare-and-swap. Alternatively, we can use the adaptive test-and-set construction of Giakkoupis and Woelfel [18] to implement our algorithm using only reads and writes with an extra multiplicative $O(\log^* n)$ factor in the running time.) We say that a process *wins* a test-and-set operation if it manages to change the value of the location from 0 to 1. Otherwise, it *loses* the operation. The winner may later *reset* the location by setting it back to 0. We assume that each process has a local random number generator, accessible through the call $\mathsf{random}(1, v)$, which returns a uniformly random integer between 1 and v.

The processes' input and their scheduling are controlled by an *oblivious adversary*. The adversary knows the algorithm and the distributions from which processes draw coins, but does not see the results of the local coin flips or of other operations performed during the execution. Equivalently, the adversary must decide on the complete schedule and input *before* the algorithm's execution.

An *activity array* data structure exports three operations. The $\mathsf{Get}()$ operation returns a unique index to the process; $\mathsf{Free}()$ releases the index returned by the most recent $\mathsf{Get}()$, while $\mathsf{Collect}()$ returns a set of indices, such that any index held by a process throughout the $\mathsf{Collect}()$ call must be returned. The adversary may also require processes to take steps running arbitrary algorithms between activity array operations. We model this by allowing the adversary to introduce a $\mathsf{Call}()$ operation, which completes in exactly one step and does not read or write to the activity array. The adversary can simulate longer algorithms by inputting consecutive $\mathsf{Call}()$ operations.

The input for each process is *well-formed*, in that Get and Free operations alternate, starting with a Get. Collect and Call operations may be interspersed arbitrarily. Both Get and Free are required to be linearizable. We say a process *holds* an index i between the linearization points of the Get operation that returned i and that of the corresponding Free operation. The key correctness property of the implementation is that no two processes hold the same index at the same point in time. Collect must return a set of indices with the following *validity* property: any index returned by Collect must have been held by some process during the execution of the operation. (This operation is not an atomic snapshot of the array.)

From a theoretical perspective, an activity array implements the *long-lived renaming* problem [24], where the Get and Free operations correspond to GetName and ReleaseName, respectively. However, the Collect operation additionally imposes the requirement that the names should be enumerable efficiently. The namespace upper bound usually required for renaming can be translated as an upper bound on the step complexity of Collect and on the space complexity of the overall implementation.

We focus on the *step complexity* metric, i.e. the number of steps that a process performs while executing an operation. We say that an event occurs *with high probability* (w.h.p.) if its probability is at least $1-1/n^{\gamma}$, for $\gamma \geq 1$ constant.

III. Related Work

The *long-lived renaming* problem was introduced by Moir and Anderson [24]. (A similar variant of renaming [10] had been previously considered by Burns and Peterson [15].) Moir and Anderson presented several deterministic algorithms, assuming various shared-memory primitives. In particular, they introduced an array-based algorithm where each process probes n locations linearly using test-and-set. A similar algorithm was considered in the context of k-exclusion [9]. A considerable amount of subsequent research, e.g. [4], [12], [14], [23] studied faster deterministic solutions for long-lived renaming. To the best of our knowledge, all these algorithms either have linear or super-linear step complexity [4], [12], [14], or employ strong primitives such as set-first-zero [24], which are not available in general. Linear time complexity is known to be inherent for deterministic renaming algorithms which employ read, write, test-and-set and compare-and-swap operations [7]. For a complete overview of known approaches for deterministic long-lived renaming we direct the reader to reference [14]. Despite progress on the use of randomization for fast *one-shot* renaming, e.g. [5], no randomized algorithms for long-lived renaming were known prior to our work.

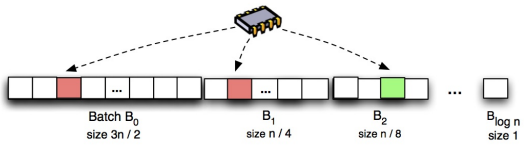

Figure 1: An illustration of the algorithm's execution. A process probes locations in batches of increasing index, until successful.

The idea of splitting the space into levels to minimize the number of collisions was first used by Broder and Karlin [13] in the context of hashing. Recently, [6] used a similar idea to obtain a *one-shot* randomized loose renaming algorithm against a strong adversary. Both references [6], [13] obtain expected worst-case complexity $O(\log \log n)$ in *one-shot* executions, consisting of exactly one Get per thread and do not consider long-lived executions. In particular, our work can be seen as an extension of [6] for the long-lived case, against an oblivious adversary. Our analysis will imply the upper bounds of [6], [13] in one-shot executions, as it is not significantly affected by the strong adversary in the one-shot case. However, our focus in this paper is analyzing *long-lived* polynomial and infinite executions, and showing that the technique is viable in practice.

A more applied line of research [17] employed data structures similar to activity arrays in the context of memory reclamation for lock-free data structures. One important difference from our approach is that the solutions considered are deterministic, and have $\Omega(n)$ complexity for registering since processes perform probes linearly. The *dynamic collect* problem defined in [17] has similar semantics to those of the activity array, and adds operations that are specific to memory management.

IV. The Algorithm

The algorithm is based on a shared array of size linear in n, where the array locations are associated to consecutive indices. A process *registers* at a location by performing a successful test-and-set operation on that location, and *releases* the location by re-setting the location to its initial value. A process performing a Collect simply reads the whole array in sequence. The challenge is to choose the locations at which the Get operation attempts to register so as to minimize contention and find a free location quickly.

Specifically, consider an array of size $2n$.[4] The locations in the arrays are all initially set to 0. The array is split into $\log n$ *batches* $B_0, B_1, \ldots, B_{\log n - 1}$ such that B_0 consists of the first $\lfloor 3n/2 \rfloor$ memory locations, and each subsequent batch B_i with $i \geq 1$ consists of the first $\lfloor n/2^{i+1} \rfloor$ entries after the end of batch $i-1$. Clearly, this array is of size at most $2n$. (For simplicity, we omit the floor notation in the following, assuming that n is a power of two.)

Get is implemented as follows: the calling process accesses each batch B_i in increasing order by index. In each batch B_i, the process sequentially attempts c_i test-and-set operations on locations chosen uniformly at random from among all locations in B_i, where c_i is a constant. For the analysis, it would suffice to consider $c_i = \kappa$ for all i, where the constant κ is a uniform lower bound of the c_i. We refrain from doing so in order to demonstrate which batches theoretically require higher values of c_i. In particular, larger values of c_i will be required to obtain high concentration bounds in later batches. In the implementation, we simply take $c_i = 1$ for all i.

A process stops once it *wins* a test-and-set operation, and stores the index of the corresponding memory location locally. When calling Free, the process resets this location back to 0. If, hypothetically, a process reaches the last batch in the main array without stopping, losing all test-and-set attempts, it will proceed to probe sequentially all locations in a second backup array, of size exactly n. In this case, the process would return $2n$ plus the index obtained from the second array as its value. Our analysis will show that the backup is essentially never called.

V. Analysis

Preliminaries. We fix an arbitrary execution, and define the linearization order for the Get and Free operations in the execution as follows. The linearization point for a Get operation is given by the time at which the successful test-and-set operation occurs, while the linearization point for the Free procedure is given by the time at which the reset operation occurs. (Notice that, since we assume a hardware test-and-set implementation, the issues concerning linearizability in the context of randomization brought up in [19] are circumvented.)

Inputs and Executions. We assume that the *input* to each process is composed of four types of operations: Get, Free, Collect and Call, each as defined in Section II. The *schedule* is given by a string of process IDs, where the process ID appearing at the i^{th} location in the schedule indicates which process takes a step at the i^{th} time step of the execution. An *execution* is characterized by the schedule together with the inputs to each process.

Running Time. The correctness of the algorithm is straightforward. Therefore, for the rest of this section, we focus on the running time analysis. We are interested in two parameters: the *worst-case* running time i.e. the maximum number of probes that a process performs in order to register, and the *average* running time, given by the expected number of probes performed by an operation. We will look at these parameters first in polynomial-length executions, and then in infinite-length executions.

Notice that the algorithm's execution is entirely specified by the schedule σ (a series of process identifiers, given by the adversary), and by the processes' coin flips, unknown to the adversary when deciding the schedule. The schedule is composed of low-level steps (shared-memory operations), which can be grouped into method calls. We say that an event occurs at time t in the execution if it occurs between steps t and $t+1$ in the schedule σ. Let σ_t be the t-th process identifier in the schedule. Further, the processes' random choices define a probability space, in which the algorithm's complexity is a random variable.

Our analysis will focus on the first $\log \log n$ batches, as processes access later batches extremely rarely. Fix the constant $c = \max_k c_k$ to be the maximum number of trials in a batch. We say that a Get operation *reaches* batch B_j if it probes at least one location in the batch. For each batch index $j \in \{0, \ldots, \log \log n - 1\}$, we define π_j to be 1 for $j = 0$ and $1/2^{2^j+5}$ for $j \geq 1$, and n_j to be n if $j = 0$ and $n/2^{2^j+5}$ for $j \geq 1$, i.e., $n_j = \pi_j n$. We now define properties of the probability distribution over batches, and of the array density.

Definition 1 (Regular Operations)**.** *We say that a* Get *operation is* regular *up to batch* $0 \leq j \leq \log \log n - 1$*, if, for any batch index* $0 \leq k \leq j$*, the probability that the operation reaches batch* k *is at most* π_k*. An operation is* fully regular *if it is regular up to batch* $\log \log n - 1$.

Definition 2 (Overcrowded Batches and Balanced Arrays)**.** *We say that a batch* j *is* overcrowded *at some time* t *if at least* $16n_j = n/2^{2^j+1}$ *distinct slots are occupied in batch* j *at time* t*. We say that the array is* balanced *up to batch* j *at time* t *if none of the batches* $0, \ldots, j$ *are overcrowded at time* t*. We say that the array is* fully balanced *at time* t *if it is balanced up to batch* $\log \log n - 1$.

In the following, we will consider both polynomial-length executions and infinite executions. We will prove

[4]The algorithm works with small modifications for an array of size $(1+\epsilon)n$, for $\epsilon > 0$ constant. This yields a more complicated exposition without adding insight, therefore we exclusively consider the case $\epsilon = 1$.

that in the first case, the array is fully balanced throughout the execution with high probability, which will imply the complexity upper bounds. In the second case, we show that the data structure quickly returns to a fully balanced state even after becoming arbitrarily degraded, which implies low complexity for most operations.

A. Analysis of Polynomial-Length Executions

We consider the complexity of the algorithm in executions consisting of $O(n^\alpha)$ Get and Free operations, where $\alpha \geq 1$ is a constant. A thread may have arbitrarily many Call steps throughout its input. Our main claim is the following.

Theorem 1. *For $\alpha > 1$, given an arbitrary execution containing $O(n^\alpha)$ Get and Free operations, the expected complexity of a Get operation is constant, while its worst-case complexity is $O(\log \log n)$, with probability at least $1 - 1/n^\gamma$, with $\gamma > 0$ constant.*

We now state a generalized version of the Chernoff bound that we will be using in the rest of this section.

Lemma 1 (Generalized Chernoff Bound [25])**.** *For $m \geq 1$, let $X_1, \ldots X_m$ be boolean random variables (not necessarily independent) with $\Pr[X_i = 1] \leq p$, for all i. If, for any subset S of $\{1, 2, \ldots, m\}$, we have that $\Pr(\wedge_{i \in S} X_i) \leq p^{|S|}$, then we have that, for any $\delta > 0$,*

$$\Pr\left(\sum_{i=1}^{n} X_i \geq (1+\delta)np\right) \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{np}.$$

Returning to the proof, notice that the running time of a Get operation is influenced by the probability of success of each of its test-and-set operations. In turn, this probability is influenced by the density of the current batch, which is related to the number of *previous* successful Get operations that stopped in the batch. Our strategy is to show that the probability that a Get operation *op* reaches a batch B_j decreases doubly exponentially with the batch index j. We prove this by induction on the linearization time of the operation. Without loss of generality, assume that the execution contains exactly n^α Get operations, and let $t_1, t_2, \ldots, t_{n^\alpha}$ be the times in the execution when these Get operations are linearized. We first prove that Get operations are fast while the array is in balanced state.

Proposition 1. *Consider a Get operation op and a batch index $0 \leq j \leq \log \log n - 2$. If at every time t when op performs a random choice the Activity Array is balanced up to batch j, then, for any $1 \leq k \leq j+1$, the probability that op reaches batch k is at most π_k. This implies that the operation is regular up to $j+1$.*

Proof: For $k = 1$, we upper bound the probability that the operation does *not* stop in batch B_0, i.e. fails all its trials in batch B_0. Consider the points in the execution when the process performs its trials in the batch B_0. At every such point, at most $n-1$ locations in batch B_0 are occupied by other processes, and at least $n/2$ locations are always free. Therefore, the process always has probability at least $1/3$ of choosing an unoccupied slot in B_0 in each trial (recall that, since the adversary is oblivious, the scheduling is independent of the random choices). Conversely, the probability that the process fails all its c_0 trials in this batch is less than $(2/3)^{c_0} \leq 1/2^7$, for $c_0 \geq 16$, which implies the claim for $k = 1$.

For batches $k \geq 2$, we consider the probability that the process fails all its trials in batch $k-1$. Since, by assumption, batch B_{k-1} is not overcrowded, there are at most $n/2^{2^{k-1}+1}$ slots occupied in B_{k-1} while the process is performing random choices in this batch. On the other hand, B_{k-1} has $n/2^k$ slots, by construction. Therefore, the probability that all of p's trials fail given that the batch is not overcrowded is at most

$$\left(\frac{n/2^{2^{k-1}+1}}{n/2^k}\right)^{c_k} = \left(\frac{1}{2}\right)^{c_k(2^{k-1}-k+1)}.$$

The claim follows since $(1/2)^{c_k(2^{k-1}-k+1)} \leq (1/2)^{2^k+4} \leq \pi_k$ for $c_k \geq 16$ and $k \geq 2$. ∎

We can use the fact that the adversary is oblivious to argue that the adversary cannot significantly increase the probability that a process holds a slot in a given batch. Due to space limitations, the full proof of the following lemma has been deferred to the full version of this paper.

Lemma 2. *Suppose the array is fully balanced at all times $t < T$. Let $B(q,t)$ be a random variable whose value is equal to the batch in which process q holds a slot at time t. Define $B(q,t) = -1$ if q holds no slot at time t. Then for all $q, j, t < T$, $\Pr[B(q,t) = j] \leq c_j \pi_j$.*

The proof of Lemma 2 is available in the full version of the paper. We can now use the fact that operations performed on balanced arrays are regular to show that balanced arrays are unlikely to become unbalanced. In brief, we use Lemmas 1 and 2 to obtain concentration bounds for the number of processes that may occupy slots in a batch j. This will show that any batch is unlikely to be overcrowded.

Proposition 2. *Let Q be the set of all processes. If, for all $q \in Q$ and some time T, the array was fully balanced at all times $t < T$, then for each $0 \leq j \leq \log \log n - 2$, batch j is overcrowded at time T with probability at most $(1/2)^{\beta\sqrt{n}}$, where $\beta < 1$ is a constant.*

Next, we bound the probability that the array ever becomes unbalanced during the polynomial-length execution.

Proposition 3. *Let t_i be the time step at which the i^{th}* Get *operation is linearized. For any $x \in \mathbb{N}$, the array is fully balanced at every time t in the interval $[0, t_x]$ with probability at least $1-O(x \log\log n/2^{\beta\sqrt{n}})$, where $\beta < 1$ is a constant.*

The proof of Proposition 3 is available in the full version of the paper. Proposition 3 has the following corollary for polynomial executions.

Corollary 1. *The array is balanced for the entirety of any execution of length n^α with probability at least $1 - O(n^\alpha \log\log n/2^{\beta\sqrt{n}})$.*

The Stopping Argument. The previous claim shows that, during a polynomial-length execution, we can practically assume that no batch is overcrowded. On the other hand, Proposition 1 gives an upper bound on the distribution over batches during such an execution, given that no batch is overcrowded.

To finish the proof of Theorem 1, we combine the previous claims to lower bound the probability that every operation in an execution of length n^α takes $O(\log\log n)$ steps by $1 - 1/n^\gamma$, with $\gamma \geq 1$ constant. The details of this are available in the full version of the paper.

Notes on the Argument. Notice that we can re-state the proof of Theorem 1 in terms of the step complexity of a single Get operation performing trials at times at which the array is fully balanced.

Corollary 2. *Consider a* Get *operation with the property that, for any $0 \leq j \leq \log\log n - 1$, the array is balanced up to j at all times when the operation performs trials in batch j. Then the step complexity of the operation is $O(\log\log n)$ with probability at least $1-O(1/n^\gamma)$ with $\gamma \geq 1$ constant, and its expected step complexity is constant.*

This raises an interesting question: at what point in the execution might Get operations start taking $\omega(\log\log n)$ steps with probability $\omega(1/n)$? Although we will provide a more satisfying answer to this question in the following section, the arguments up to this point grant some preliminary insight. Examining the proof, notice that a necessary condition for operations to exceed $O(\log\log n)$ worst case complexity is that the array becomes unbalanced. By Proposition 3 the probability that the array becomes unbalanced at or before time T is bounded by $O(T/2^{\sqrt{n}})$ (up to logarithmic terms). Therefore operations cannot have $\omega(\log\log n)$ with non-negligible probability until $T = \Omega(2^{\sqrt{n}})$.

B. Infinite Executions

In the previous section, we have shown that the data structure ensures low step complexity in polynomial-length executions. However, this argument does not prevent the data structure from reaching a bad state over infinite-length executions. In fact, the adversary could in theory run the data structure until batches $B_1, B_2, \ldots$ are overcrowded, and then ask a single process to Free and Get from this state infinitely many times. The *expected* step complexity of operations from this state is still constant, however the expected *worst-case* complexity would be logarithmic. This line of reasoning motivates us to analyze the complexity of the data structure in infinite executions. Our analysis makes the assumption that a thread releases a slot within polynomially many steps from the time when it acquired it.

Definition 3. *Given an infinite asynchronous schedule σ, we say that σ is* compact *if there exists a constant $B \geq 0$ such that, for every time t in σ at which some process initiates a* Get *method call, that process executes a* Free *method call at some time $t' < t + n^B$.*

Our main claim is the following.

Theorem 2. *Given a compact schedule, every* Get *operation on the* LevelArray *will complete in $O(\log\log n)$ steps with high probability.*

Proof Strategy. We first prove that, from an arbitrary starting state, in particular from an unbalanced one, the LevelArray enters and remains in a *fully balanced* state after at most polynomially many steps, with high probability (see Lemma 3). This implies that the array is fully balanced at *any* given time with high probability. The claim then follows by Corollary 2.

Lemma 3. *Given a compact schedule with bound B and a* LevelArray *in arbitrary initial state, the* LevelArray *will be fully balanced after $n^B \log\log n$ total system steps with probability at least $1 - O(n^B(\log\log n)^2/2^{\beta\sqrt{n}})$, where $\beta < 1$ is a constant.*

The proof of Lemma 3 is available in the full version of this paper. Lemma 3 naturally leads to the following corollary, whose proof is also available in the full version.

Corollary 3. *For any time $t \geq 0$ in the schedule, the probability that the array is not fully balanced at time t is at most μ.*

The completion of the proof of Theorem 2 is straightforward. The details are available in the full version of the paper.

VI. Implementation Results

Methodology. The machine we use for testing is a Fujitsu PRIMERGY RX600 S6 server with four Intel Xeon E7-4870 (Westmere EX) processors. Each processor has 10 2.40 GHz cores, each of which multiplexes two hardware threads, so in total our system supports 80 hardware threads. Each core has private write-back L1 and L2 caches; an inclusive L3 cache is shared by all cores.

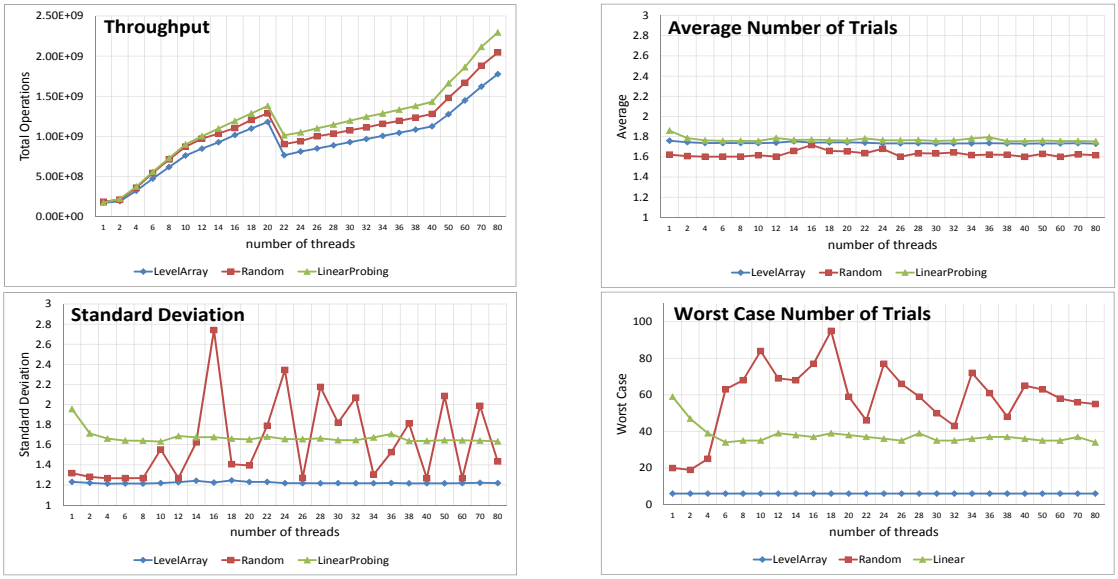

Figure 2: Comparing the performance of LevelArray with Random and LinearProbing. Throughput and average complexity are similar, while the LevelArray is significantly more stable in terms of standard deviation and worst-case complexity.

We examine specific behaviors by adjusting the following benchmark parameters. The parameter n is the number of hardware threads spawned, while N is the maximum number of array locations that may be registered at the same time. For $N > n$, we emulate concurrency by requiring each thread to register N/n times before deregistering. The parameter L is the number of slots in the array. In our tests, we consider values of L between $2N$ and $4N$.

The benchmark first allocates a global array of length L, split as described in Section IV, and then spawns n threads which repeatedly register and deregister from the array. In the implementation, threads perform exactly *one* trial in each batch, i.e. $c_\ell = 1$, for all batches $\ell = 1, \ldots, \log N$. (We tested the algorithm with values $c_\ell > 1$ and found the general behavior to be similar; its performance is slightly lower given the extra calls in each batch. The relatively high values of c_ℓ in the analysis are justified since our objective was to obtain high concentration bounds.) Threads use compare-and-swap to acquire a location. We used the Marsaglia and Park-Miller (Lehmer) random number generators, alternatively, and found no difference between the results.

The *pre-fill percentage* defines the percentage of array slots that are occupied during the execution we examine. For example, 90% pre-fill percentage causes every thread to perform 90% of its registers *before* executing the main loop, without deregistering. Then, every thread's main-loop performs the remaining 10% of the register and deregister operations, which execute on an array that is 90% loaded at every point.

In general, we considered regular-use parameter values; we considered somewhat exaggerated contention levels (e.g. 90% pre-fill percentage) since we are interested in the worst-case behavior of the algorithm.

Algorithms. We compared the performance of LevelArray to three other common algorithms used for fast registration. The first alternative, called Random, performs trials at random in an array of the same size as our algorithm, until successful. The second, called LinearProbing, picks a random location in an array and probes locations linearly to the right from that location, until successful. We also tested the *deterministic* implementation that starts at the first index in the array and probes linearly to the right. Its average performance is at least two orders of magnitude worse than all other implementations for all measures considered, therefore it is not shown on the graphs.

Performance. Our first set of tests is designed to determine the throughput of the algorithm, i.e. the total number of Get and Free operations that can be performed during a fixed time interval. We analyzed the throughput for values of n between 1 and 80, requiring the threads to register a total number of N emulated threads on an array of size L between $2N$ and $4N$. We also considered the way in which the throughput is affected by the different pre-fill percentages.

Figure 2 presents the results for n between 1 and 80, $N = 1000n$ simulated operations, $L = 2N$, and a pre-fill percentage of 50%. We ran the experiment for 10 seconds, during which time the algorithm performed between 200 million and 2 billion operations. The first graph gives the total number of successful operations as a function of the number of threads. As expected, this number grows linearly with the number of threads. (The variation at 20 is because this is the point where a new processor is used—we start to pay for the expensive inter-processor communication. Also, notice that the X axis is not linear.) The fact that the throughput of LevelArray is lower than that of Random and LinearProbing is to be expected, since the average number of trials for a thread probing randomly is lower than for our algorithm (since Random and LinearProbing use more space for the first trial, they are more likely to succeed in one operation; LinearProbing also takes advantage of better cache performance.) This fact is illustrated in the second graph, which plots the *average* number of trials per operation. For all algorithms, the average number of trials per Get operation is between 1.5 and 1.9.

The lower two graphs illustrate the main weakness of the simple randomized approaches, and the key property of LevelArray. In Random and LinearProbing, even though processes perform very few trials on average, there are always some processes that have to perform a large number of probes before getting a location. Consequently, the standard deviation is high, as is the worst-case number of steps that an operation may have to take. (To decrease the impact of outlier executions, the worst-case shown is averaged over all processes, and over several repetitions.) On the other hand, the LevelArray algorithm has predictable low cost even in extremely long executions. In this case, the maximum number of steps an operation must take before registering is at most 6, taken over 200 million to 2 billion operations. The results are similar for pre-fill percentages between 0% and 90%, and for different array sizes. These bounds hold in executions with more than 10 billion operations.

The Healing Property. The stable worst-case performance of LevelArray is given by the properties of the distribution of probes over batches. However, over long executions, this distribution might get skewed, affecting the performance of the data structure. The analysis in Section V-B suggests that the batch distribution returns to normal after polynomially many operations. We test this argument in the next experiment, whose results are given in Figure 3.

The figure depicts the distribution of threads in batches at different points in the execution. Initially, the first batch is a quarter full, while the second batch is half full, therefore overcrowded. As we schedule operations, we see that the distribution returns to normal. After about 32000 arbitrarily chosen operations are scheduled, the distribution is in a stable state. (Snapshots are taken every 4000 operations.) The speed of convergence is higher than predicted by the analysis. We obtained the same results for variations of the parameters.

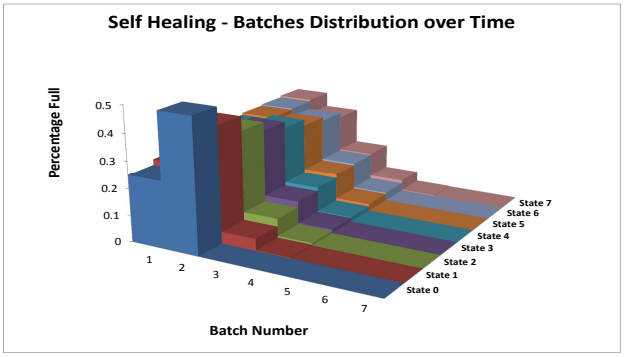

Figure 3: The healing property of the algorithm. The array starts in unbalanced state (batch two is overcrowded), and smoothly transitions towards a balanced state as more operations execute. Snapshots are taken every 4000 operations.

VII. Conclusions and Future Work

In general, an obstacle to the adoption of randomized algorithms in a concurrent setting is that, while their performance may be good on average, it can have high variance for individual threads in long-lived executions. Thus, randomized algorithms are seen as unpredictable. In this paper, we exhibit a randomized algorithm which combines the best of both worlds, guaranteeing good performance on average and in the worst case, over long finite or even infinite executions (under reasonable schedule assumptions). One direction of future work would be investigating randomized solutions with the same strong guarantees for other practical concurrent problems, such as elimination [26] or rendez-vous [2].

VIII. Acknowledgments

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.

References

[1] Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set (extended abstract). In *Proceedings of the 6th International Workshop on Distributed Algorithms*, WDAG '92, pages 85–94, London, UK, UK, 1992. Springer-Verlag.

[2] Yehuda Afek, Michael Hakimi, and Adam Morrison. Fast and scalable rendezvousing. In *Proceedings of the 25th international conference on Distributed computing*, DISC'11, pages 16–31, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Yehuda Afek, Alexander Matveev, and Nir Shavit. Pessimistic software lock-elision. In *Proceedings of the 26th international conference on Distributed Computing*, DISC'12, pages 297–311, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[5] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Morteza Zadimoghaddam. Optimal-time adaptive strong renaming, with applications to counting. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '11, pages 239–248, New York, NY, USA, 2011. ACM.

[6] Dan Alistarh, James Aspnes, George Giakkoupis, and Philipp Woelfel. Randomized loose renaming in O(log log n) time. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 200–209, New York, NY, USA, 2013. ACM.

[7] Dan Alistarh, James Aspnes, Seth Gilbert, and Rachid Guerraoui. The complexity of renaming. In *Proceedings of the 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25*, pages 718–727, 2011.

[8] Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proc. 24th International Conference on Distributed Computing (DISC)*, pages 94–108. Springer-Verlag, 2010.

[9] James H. Anderson and Mark Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11(1):1–20, 1997.

[10] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Ruediger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.

[11] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.

[12] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *J. ACM*, 50(4):444–468, 2003.

[13] Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pages 43–53, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[14] Alex Brodsky, Faith Ellen, and Philipp Woelfel. Fully-adaptive algorithms for long-lived renaming. *Distributed Computing*, 24(2):119–134, 2011.

[15] James E. Burns and Gary L. Peterson. The ambiguity of choosing. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 145–157, New York, NY, USA, 1989. ACM.

[16] David Dice, Alexander Matveev, and Nir Shavit. Implicit privatization using private transactions. In *Electronic Proceedings of the Transact 2010 Workshop*, April 2010.

[17] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '11, pages 99–108, New York, NY, USA, 2011. ACM.

[18] George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 19–28, New York, NY, USA, 2012. ACM.

[19] Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd annual ACM symposium on Theory of computing*, STOC '11, pages 373–382, New York, NY, USA, 2011. ACM.

[20] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[21] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[22] W. McAllister. *Data Structures and Algorithms Using Java*. Jones & Bartlett Learning, 2008.

[23] Mark Moir. Fast, long-lived renaming improved and simplified. *Sci. Comput. Program.*, 30(3):287–308, 1998.

[24] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, October 1995.

[25] Alessandro Panconesi and Aravind Srinivasan. Randomized distributed edge coloring via an extension of the chernoff–hoeffding bounds. *SIAM J. Comput.*, 26(2):350–368, April 1997.

[26] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks. *Theory Comput. Syst.*, 30(6):645–670, 1997.