Randomized Renaming in Shared Memory Systems

Petra Berenbrink[a], André Brinkmann[b], Robert Elsässer[c], Tom Friedetzky[d], Lars Nagel[e]

[a]*Fachbereich Informatik, Universität Hamburg, 22527 Hamburg, Germany*
[b]*Zentrum für Datenverarbeitung, Johannes Gutenberg-Universität Mainz, 55099 Mainz, Germany*
[c]*Department of Computer Sciences, University of Salzburg, 5020 Salzburg, Austria*
[d]*School of Engineering and Computing Sciences, Durham University, Durham, DH1 3LE, United Kingdom*
[e]*Department of Computer Science, Loughborough University, Loughborough, LE11 3TU, United Kingdom*

Abstract

Renaming is a task in distributed computing where $\boldsymbol{n}$ processes are assigned new names from a name space of size $\boldsymbol{m}$. The problem is called *tight* if $\boldsymbol{m = n}$, and *loose* if $\boldsymbol{m > n}$. In recent years renaming came to the fore again and new algorithms were developed. For tight renaming in asynchronous shared memory systems, Alistarh et al. describe a construction based on the AKS network that assigns all names within $\boldsymbol{\mathcal{O}(\log n)}$ steps per process. They also show that, depending on the size of the name space, loose renaming can be done considerably faster. For $\boldsymbol{m = (1+\epsilon)\cdot n}$ and constant $\boldsymbol{\epsilon}$, they achieve a step complexity of $\boldsymbol{\mathcal{O}(\log\log n)}$.

In this paper we consider tight as well as loose renaming and introduce randomized algorithms that achieve their tasks with high probability. The model assumed is the asynchronous shared-memory model against an adaptive adversary. Our algorithm for loose renaming maps $\boldsymbol{n}$ processes to a name space of size $\boldsymbol{m = (1+2/(\log n)^{\ell})\cdot n = (1+o(1))\cdot n}$ performing $\boldsymbol{\mathcal{O}(\ell\cdot(\log\log n)^2)}$ test-and-set operations. In the case of tight renaming, we present a protocol that assigns $\boldsymbol{n}$ processes to $\boldsymbol{n}$ names with step complexity $\boldsymbol{\mathcal{O}(\log n)}$, but without the overhead and impracticality of the AKS network.

Email addresses: berenbrink@informatik.uni-hamburg.de (Petra Berenbrink), brinkman@uni-mainz.de (André Brinkmann), elsa@cosy.sbg.ac.at (Robert Elsässer), tom.friedetzky@durham.ac.uk (Tom Friedetzky), l.nagel@lboro.ac.uk (Lars Nagel)

This algorithm utilizes modern hardware features in form of a counting device which is also described in the paper. This device may have the potential to speed up other distributed algorithms as well.

Keywords: tight renaming, loose renaming, distributed algorithm, shared memory model, randomized algorithm

---

1. Introduction

Renaming is a task in distributed computing in which processes are assigned distinct names from a new and usually small name space. The number of processes is denoted by n, the size of the name space by m. The problem is called *tight* if $m = n$ and *loose* if $m > n$. Dependent on the model, the processes communicate synchronously via messages or have asynchronous access to shared memory. In the former case, the objective is to restrict the number of communication rounds and possibly the size of the messages; in the latter case, the objective is to minimize the (total) step complexity. The *step complexity* is the maximum number of accesses to the shared memory by any process; the *total step complexity* sums up the accesses by all processes. We assume that an arbitrary number of processes can fail which are chosen by an adaptive adversary.

In recent years renaming gained new popularity and several papers appeared that investigated renaming in the synchronous message-passing model [1, 2, 3] and in the asynchronous shared-memory model [4, 5, 6, 7, 8, 9, 10, 11]. Assuming the shared-memory model with the names stored in test-and-set registers, the authors of [8] describe a construction based on the AKS network that assigns all names to a tight name space within $\mathcal{O}(\log n)$ steps per process. For loose renaming in the same model, it is shown in [10] that $\mathcal{O}(\log \log n)$ steps are sufficient to provide n processes with distinct names from a name space of size $(1 + \epsilon) \cdot n$ where ϵ is a constant.

In this paper we consider tight as well as loose renaming in the asynchronous shared-memory model[1]. The presented algorithms use random bits and achieve their tasks with high probability[2].

---

[1] This paper is an extended version of work published in [12]. It provides a more detailed description and analysis including the total step complexity and the space complexity of the algorithms.

[2] An event $\mathcal{A}$ occurs *with high probability* (w.h.p.) if $\mathbf{Pr}[\mathcal{A}] \geq 1 - 1/n^c$ for some constant

For tight renaming, our algorithm has a step complexity of $\mathcal{O}(\log n)$, asymptotically equal to the algorithm of Alistarh et al. [8] while avoiding the overhead of an AKS network. The total step complexity is lower with $\mathcal{O}(n)$ (compared to $\mathcal{O}(n \log n)$ [8]). In order to achieve this result, the names must be stored in a special type of hardware register with an integrated counting device.

Our two algorithms for loose renaming map n processes to a name space of size $m = (1 + \epsilon) \cdot n$ w.h.p. where $\epsilon = o(1)$. The first algorithm requires a name space of size $m = (1 + 2/(\log \log n)^{\ell}) \cdot n$ and has a step complexity of $\mathcal{O}((\log \log n)^{\ell})$. For the second algorithm, the size of the name space is only $m = (1 + 2/(\log n)^{\ell}) \cdot n$ and the step complexity $\mathcal{O}((\log \log n)^2)$. To the best of our knowledge, these are the first algorithms that achieve almost tight renaming (i.e., with only a sublinear addition of names) in a poly-double-logarithmic number of steps.

The remainder of the paper is structured as follows: After a discussion of the related work in Section 2, the model is described in Section 3. The special hardware register which used by the algorithm for tight renaming is explained in Section 4. This algorithm is stated and analyzed in Section 5. The algorithms for loose renaming are discussed in Section 6. The paper is summarized and concluded in Section 7.

2. Related Work

There is a substantial amount of work on algorithms for renaming in different models. In this section we only consider results in the asynchronous shared-memory model using test-and-set (`TAS`) registers, similar to our model. The names are stored in `TAS` registers. Unless stated otherwise, testing such a register is assumed to be an atomic hardware operation and takes one step. The first process that tests a `TAS` register *wins* it and gets the name in it. If several processes test a `TAS` register at the same time, an arbitrary one of them wins it.

Additionally, we focus on randomized algorithms; for an overview of deterministic algorithms we refer the reader to [5]. We distinguish between *loose* and *tight* renaming. In loose renaming the name space is larger than the number of processes, whereas in tight renaming the size of the name

$c > 0$.

	Tight renaming	**Loose renaming**	
	Complexity	Name space	Complexity
Step complexity	$\mathcal{O}(\log^2 n)$ [8]	$(1+\epsilon)n$	$\log\log n + \mathcal{O}(1)$ [10]
	$\mathcal{O}(\log k)$ [8] (AKS)	$\mathcal{O}(k)$	$\mathcal{O}((\log\log k)^2)$ [10]
	$\Omega(\log n)$ [8]	$\mathcal{O}(n)$	$\Omega(\log\log n)$ [10]
Total step complexity	$\mathcal{O}(n\log^2 n)$ [4, 8]	$(1+\epsilon)n$	$\mathcal{O}(n\log^2 n)$ [13]
	$\mathcal{O}(k\log k)$ [8] (AKS)	$(1+\epsilon)k$	$\mathcal{O}\left(\frac{k\log^2 k}{\log^2(1+\epsilon)}\right)$ [4]
		$(1+\epsilon)n$	$\mathcal{O}(n)$ [10]
		$\mathcal{O}(k)$	$\mathcal{O}(k\log\log k)$ [10]

Table 1: Known lower and upper bounds for the model used in this paper. All upper bounds hold with high probability. The lower bound for loose renaming assumes a linear number of `TAS` objects. $\epsilon > 0$ is a constant. Using k instead of n is to indicate that the respective algorithm is adaptive. AKS marks the algorithm using an AKS network.

space equals the number of processes. In the case of *adaptive* renaming, the number of processes is not known in advance. Table 1 summarizes the step complexity results from the literature.

Loose Renaming. Panconesi et al. [13] were the first to use randomization for loose renaming. They use a probabilistic `TAS` register that selects a winner among parallel accesses with probability $1-o(1)$. They present an algorithm that assumes a name space of size $(1+\epsilon)n$ for a constant positive ϵ. The total step complexity of their algorithm is $\mathcal{O}(n\log^2 n)$ with probability $1-o(1)$.

In [4] Alistarh et al. present the first adaptive randomized renaming algorithm where the contention k, i.e. the actual number of participating processes, is not known in advance. They propose an adaptive `TAS` register implementation called `RatRace` with access costs $\mathcal{O}(\log^2 k)$. Based on that implementation, the algorithm performs loose renaming on a name space of size $(1+\epsilon)\cdot k$ and has a total step complexity of $\mathcal{O}(k\log^4 k/\log^2(1+\epsilon))$ (where a $\log^2 k$ factor would be saved in the case of constant-cost `TAS` access). The space complexity of the `RatRace` algorithm is reduced from $\mathcal{O}(k^3)$ to $\mathcal{O}(k)$ by Giakkoupis and Woelfel in [9]. They also show that test-and-set can be implemented with a step complexity of $\mathcal{O}(\log^* k)$ assuming an

oblivious adversary, where $\log^*$ denotes the iterated logarithm[3].

Alistarh et al. [10] present one non-adaptive algorithm, `ReBatching`, and two adaptive ones, `AdaptiveReBatching` and `FastAdaptiveReBatching`, assuming `TAS` registers given in hardware and an adaptive adversary which is allowed to see the state of all processes (including the results of coin flips) when making its scheduling choices. We use their results in Section 6, which is why we phrase them in the following theorem and restate the `ReBatching` algorithm (Algorithm 1).

Theorem 1 (Theorem 4.1, 5.1 and 5.2 in [10])**.** *For any fixed $\epsilon > 0$ and a name space of size $(1+\epsilon)n$, `ReBatching` uses $\mathcal{O}(n)$ `TAS` objects and achieves w.h.p. step complexity at most $\log\log n + \mathcal{O}(1)$ and total step complexity $\mathcal{O}(n)$ against an adaptive adversary.*

The adaptive algorithms require a name space of size $\mathcal{O}(k)$. `AdaptiveReBatching` has step complexity $\mathcal{O}((\log\log k)^2)$ and `FastAdaptiveReBatching` a total step complexity of $\mathcal{O}(k \log\log k)$.

The `ReBatching` algorithm divides the name space of size $(1+\epsilon)n$ into disjoint sets B_i, $0 \leq i \leq \lceil \log\log n \rceil$, where $|B_0| = n$ and, for $i > 0$, $|B_i| = \lceil \epsilon n/2^i \rceil$. Every process tests t_i `TAS` registers from each set B_i, until it wins one. The t_i are defined in the pseudocode of Algorithm 1.

The authors of [10] also show a lower bound of $\Omega(\log\log n)$ on the step complexity of randomized renaming given an oblivious adversary and that the name space and the number of `TAS` objects are linear in n. For deterministic algorithms, in comparison, the lower bound is known to be $\Omega(n)$ and, thus, exponentially worse [11].

Tight renaming. In [14] Eberly et al. consider a renaming problem in which the name space size ℓ is smaller than the number n of processes. They use a different model that allows processes to release names and assume that no more than $k \leq \ell$ processes hold or acquire names at the same time. Their randomized tight renaming algorithm has an amortized step complexity of $\mathcal{O}(k \log k)$ where the step complexity is here defined as the maximum number of steps that any process performs to find a name.

Alistarh et al. [4] present a very simple tight renaming algorithm called `ReShuffle` which lets every process choose one random object out of all `TAS`

[3]The iterated logarithm of k is the number of times the logarithm must be applied before the result is equal to or is less than 1.

Algorithm 1 `ReBatching`(n, ϵ) [10]

1: /* $m = \lceil (1+\epsilon)n \rceil$ */
2: shared: array $B[0 \dots m-1]$ of `TAS` objects
3: /* $\kappa = \lceil \log \log n \rceil$
4: **for each** $i \in \{0, \dots, \kappa\}$
5: $$t_i = \begin{cases} \lceil 17 \ln(8e/\epsilon)/\epsilon \rceil & \text{if } i = 0 \\ 1 & \text{if } 1 \le i \le \kappa - 1 \\ 3 & \text{if } i = \kappa \end{cases}$$
6: $$b_i = \begin{cases} n & \text{if } i = 0 \\ \lceil \epsilon n / 2^i \rceil & \text{if } 1 \le i \le \kappa \end{cases}$$
7: $s_i = \sum_{0 \le j < i} b_j$
8: $B_i = B[s_i \dots s_i + b_i - 1]$ */
9: **for all** processes in parallel **do**
10: **for** $i \in \{0, \dots, \kappa\}$ **do**
11: **for** t_i times **do**
12: choose random `TAS` object x in B_i i.u.r.
13: **if** x is won **then**
14: **break**

objects in each round, until it wins a name. It is shown that `ReShuffle` has a total step complexity of $\mathcal{O}(n \log^2 n)$ if a `TAS` access has constant costs. In the paper the costs of an access are $\mathcal{O}(\log^2 n)$, and the total step complexity is given as $\mathcal{O}(n \log^4 n)$. In Section 6, we use a similar algorithm for loose renaming.

In [8] Alistarh et al. give two new randomized algorithms for tight renaming which work in the presence of an adaptive adversary. The first algorithm called `BitBatching` divides the `TAS` registers storing the names into $\ell = \mathcal{O}(\log n)$ clusters of decreasing size. The first $\ell - 1$ clusters have size $n \cdot 2^{-i}$ for $i = 1, ..., \ell - 1$; the last cluster contains the remaining $\mathcal{O}(\log n)$ registers. Every process goes through the clusters sequentially and tests $\mathcal{O}(\log n)$ registers in each of them, until it wins one and thus gets a name. The step complexity of the algorithm is therefore $\mathcal{O}(\log^2 n)$ if the `TAS` registers are implemented in hardware. We use the basic idea of their approach in two algorithms for tight and loose renaming.

The second tight renaming algorithm in [8] transforms any sorting network into an adaptive renaming protocol with an expected step complexity

cost equal to the depth of the sorting network. Using an AKS sorting network, this gives a strong adaptive renaming algorithm with step complexity $\mathcal{O}(\log k)$ and total step complexity $\mathcal{O}(k \log k)$. This approach has the disadvantage that while the depth of the AKS network is logarithmic, the constant is a rather unwieldy, not to mention the complicated structure of an AKS network. It also needs a large amount of `TAS` registers because the width of the network equals the initial name space of the processes. The authors show that the step complexity is asymptotically optimal. Deterministic algorithms for tight renaming, on the other hand, have a step complexity of $\Theta(n)$ [11].

Our tight renaming algorithm in Section 5 has the same step complexity as the AKS network algorithm, but a better total step complexity. Besides, it is simpler and more space-efficient, yet requires a special type of hardware register.

3. Preliminaries

3.1. Model

The considered machine model is the asynchronous shared-memory model with concurrent reads and concurrent writes (CRCW). The processes follow an algorithm composed of steps. Any number of processes may fail by crashing, and a failed process does not perform further steps in the execution. The order in which processes perform steps and their crashes are controlled by an adversary. We assume an adaptive adversary that is allowed to see the state of all processes (including the results of coin flips) when making its scheduling choices.

The asynchronous shared memory contains the name space with m names and can be accessed by all n processes. Besides the name space, additional memory can be used as temporary memory. Like in [11], each name is stored in a test-and-set (`TAS`) register that can be concurrently tested by several processes, but only *won* by one process.

For our tight renaming algorithm, we use a special hardware register, called $\tau(s)$-register. It includes a counting device with `TAS` bits, i.e. `TAS` registers consisting of only one bit. In each step each process is allowed to test at most one `TAS` register or `TAS` bit. When a process wins a `TAS` bit, the bit, initially set to 0, is set to 1, and the process knows that it has been successful. When a process wins a `TAS` register, it is assigned the name in it. Like in other papers, e.g. [11], we assume that concurrent accesses to the same `TAS` register or `TAS` bit can be executed in one step and that every name

and address can be read or written in one step. Some of these names and numbers have $\log n$ bits (or more). Likewise it is assumed that a processor's registers and instructions can handle numbers of this size and run processor instructions like `xor` and `popcnt` on these numbers in $\mathcal{O}(1)$.

The $\tau(s)$-register and the counting device are described in more detail in Section 4.

3.2. Technical tools

In the technical parts of this paper we use the following versions of the well-known Chernoff concentration inequality.

Lemma 2. *Let $X_1, \ldots, X_n$ be independent random variables such that $X_i \in \{0,1\}$. Let $p_i = P(X_i = 1) = \mathbb{E}[X_i]$, $X = \sum_{i=1}^{n} X_i$ and $\mu = \mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} p_i$. Then,*

1. *For any $\delta \in [0,1]$, $P[X \geq (1+\delta) \cdot \mu] \leq e^{-\mu\delta^2/3}$.*
2. *For any $\delta \geq 1$, $P[X \geq (1+\delta) \cdot \mu] \leq e^{-\mu\delta/3}$.*
3. *For any $\delta > 0$, $P[X \leq (1-\delta) \cdot \mu] \leq e^{-\mu\delta^2/3}$.*
4. *For any $\delta > 0$, $P[X \geq (1+\delta) \cdot \mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}$.*

4. $\tau(s)$-register

In order to efficiently calculate tight renaming, our algorithm depends on special hardware registers, called $\tau(s)$-*registers*. Each of these registers has two parts: (i) a set of s `TAS` registers that contain the names, (ii) a counting device managing $2 \log n$ `TAS` bits. The counting device counts the number of `TAS` bits set and allows at most s of them to be permanently set; the others are unset again. Any process that wants to get a name has to permanently win one of the $2 \log n$ `TAS` bits first. After checking if the `TAS` bit is permanently set, the process systematically goes through the `TAS` registers, until it wins one of them, and retrieves the name. It must win one of the `TAS` register because there are exactly s of them and at most s processes are allowed to search.

As the `TAS` registers and the search are straightforward, we only have a closer look at the counting device.

4.1. Counting Device

The counting device is composed of $2 \log n$ individual `TAS` bits and can restrict the number of permanently set 1-bits to any positive threshold $s \leq 2 \log n$. We assume that all individual bits of a $\tau(s)$-register have the same clock as input and that it is possible to read all $2 \log n$ individual bits within one operation. The register operates in clock cycles that are divided in phases. The synchronization of the bits permits that supernumerary `TAS` bits can be unset before the counting device is accessed again by new processes.

However, we do not make any assumptions about the arrival or the order of the requests. Processes can use different clocks and send their requests asynchronously at any time. Yet, since requests are only answered in a certain phase, the processing may start with a (constant) delay. The implementation of a $\tau(s)$-register is based on Algorithm 2 which represents one clock cycle.

Algorithm 2 Algorithm of $\tau(s)$-register

```
1:  allowed_bits ← s − popcnt(in_reg)
2:  for i ∈ {1, ..., 2 log n} in parallel do
3:      processes test-and-set bit b_i
4:  if s < popcnt(in_reg) then
5:      util_reg_0 ← out_reg xor in_reg
6:      for i ∈ {1, ..., 2 log n} in parallel do
7:          util_reg_i ← util_reg_0 << (i − 1)
8:          if popcnt(util_reg_i) = allowed_bits then
9:              if bt(util_reg_i, 1) then
10:                 util_reg_i ← util_reg_i >> (i − 1)
11:                 out_reg ← out_reg or util_reg_i
12:     in_reg ← out_reg
13: else
14:     out_reg ← in_reg
```

The counting device has two main registers: `in_reg` and `out_reg`. The register `in_reg` contains the `TAS` bits the processes access. The bits of register `out_reg` can be read by the processes to check whether they have really won their respective `TAS` bit. After each execution, both registers are updated such that exactly those bits are set in `in_reg` that have been won by processes and that `out_reg` is an exact copy of `in_reg`. Aside from these two registers, $2 \log n + 1$ auxiliary registers $\texttt{util\_reg}_i$, $i = 0, ..., 2 \log n$, are needed.

A clock cycle is divided in two phases, the first one covers lines 1–3, the second lines 4–14: In the first line, the algorithm determines the number of bits the register `in_reg` is short of the threshold s. Then, in lines 2–3, the `TAS` bits of `in_reg` parallelly handle requests of processes. Every request to a `TAS` bit b_i fails if b_i is already set to 1. If bit b_i is unset and if there is at least one request to b_i, b_i will be (preliminarily) set by exactly one of the processes. All other requests to b_i fail.

If the threshold s is exceeded in line 4, ($\texttt{popcnt}(\texttt{in\_reg}) - s$) many of the new bits have to be removed. For this purpose, $\texttt{util\_reg}_0$ is prepared in line 5 as a copy of `in_reg` without the old bits, i.e. the bits set prior to this cycle. The algorithm then shifts $\texttt{util\_reg}_0$ by every possible number of bits (line 7) and selects the only resulting bit array $\texttt{util\_reg}_i$ which has both, the correct number of new bits (line 8) and a 1-bit in the first position (line 9). (The first bit is tested using the instruction $\texttt{bt}(\texttt{util\_reg}_i, 1)$.) $\texttt{util\_reg}_i$ is shifted back (line 10) and combined with the old bits in `out_reg` (line 11). The resulting bit array having exactly s bits, $s - \texttt{allowed\_bits}$ old and `allowed_bits` new bits, is stored in `out_reg` (line 11) and `in_reg` (line 12).

If the threshold s is not exceeded in line 4, `in_reg` can simply be copied to `out_reg` (line 14).

A process that won a `TAS` bit (in line 3) has to check whether this `TAS` bit was later unset (in line 12). It can be certain that the `TAS` bit is unset as soon as it is unset in `in_reg`, and it can be certain to have won it, once the according bit has also been set in `out_reg`. In the latter case, it can immediately start searching the `TAS` registers for a free name.

Each step of the algorithm can be performed in a constant number of time steps, usually in one time step, so that the $\tau(s)$-register only induces a constant slowdown compared to a standard `TAS` register. Nevertheless, there is a significant hardware overhead of $\mathcal{O}(\log n)$ additional registers and arithmetic logic units. It is therefore unlikely that such a register will be actually built, but it could be constructed based on this description.

4.2. Example

Figure 1 describes one cycle of a $\tau(\log n)$-register where $\log(n) = 10$. Before the cycle 7 of the 10 names are assigned (indicated by filled quadrangles), and thus 7 of the $2 \cdot \log(n) = 20$ `TAS` bits were won. The `in_reg` contains the old won bits as well as 6 bits newly tested in the current cycle. The `out_reg` only contains the old bits. Since there are only 3 names left, 3 of the 6 newly set bits must be unset for which the utility registers

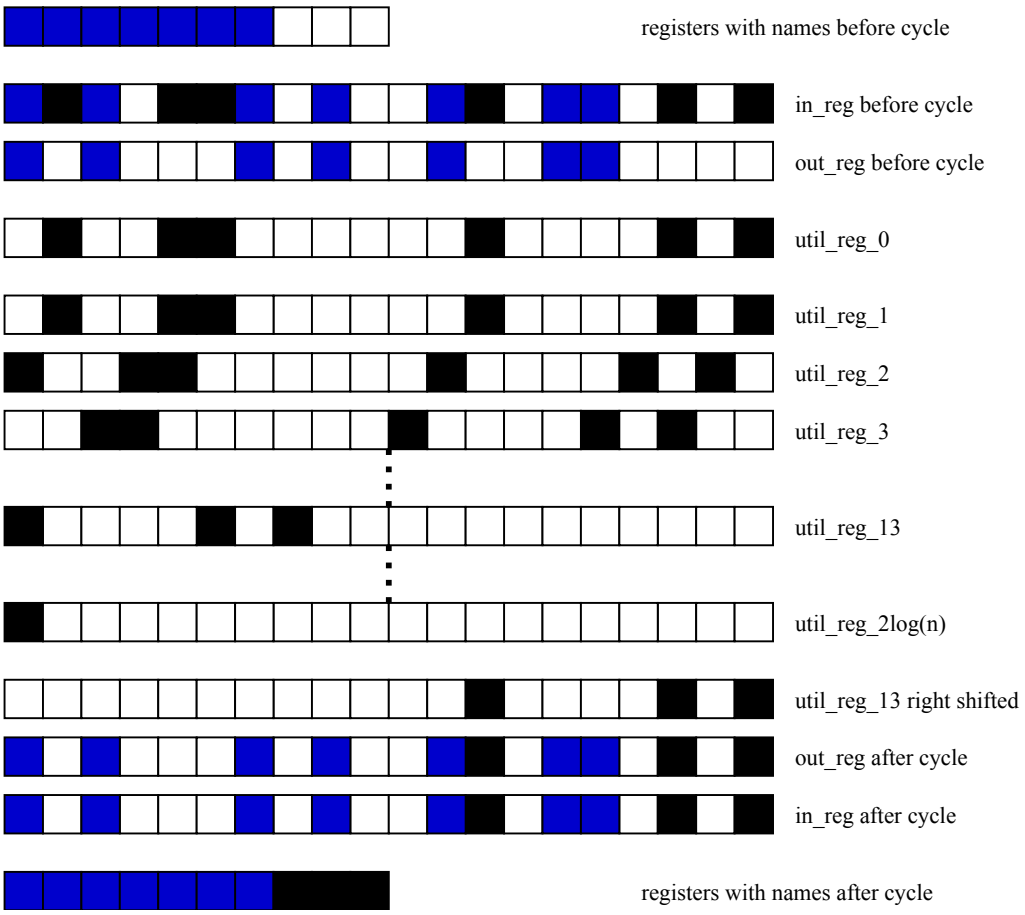

Figure 1: Example cycle of a $\tau(\log n)$-register.

$\texttt{util\_reg}_i$ are used. (They would not be used if the threshold of 10 was not exceeded.) $\texttt{util\_reg}_0$ only contains the new bits which are obtained by XORing `in_reg` and `out_reg`. $\texttt{util\_reg}_i$, $1 \leq i \leq 2 \cdot \log(n)$, is obtained by left-shifting $\texttt{util\_reg}_0$ by $i - 1$ positions. From these utility registers, the hardware algorithm chooses the one with $10 - 7 = 3$ bits in which the first bit is set. This is $\texttt{util\_reg}_{13}$. $\texttt{util\_reg}_{13}$ is right-shifted and ORed with `out_reg` to create the result which is then stored in `in_reg` and `out_reg`. The 3 processes that have won `TAS` bits pick the 3 remaining names from the 10 `TAS` registers.

5. Tight Renaming using $\tau(\log n)$-Registers

In this section we design and analyse an algorithm (given as Algorithm 3) for solving the tight renaming problem using $\tau(\log n)$-registers in time $\mathcal{O}(\log n)$ and space $\mathcal{O}(n)$. We start with a high-level description of algorithm and analysis and introduce the notation.

We are using an auxiliary array T_{aux} of length $2n$ of `TAS` bits belonging to $n/\log n$ many $\tau(\log n)$-registers. Recall, each $\tau(\log n)$-register has $2\log n$ `TAS` bits (which we also refer to as *blocks*). We divide the array into $R = \frac{\log n - \log\log n}{\log 2} = \mathcal{O}(\log n)$ clusters $C_1, C_2, \cdots, C_R$ defined as follows and one cluster C_{R+1} with the remaining names. For $i \in \{1, ..., R\}$, C_i consists of

$$c_i = \frac{2 \cdot n}{2^i}$$

`TAS` bits and hence of

$$b_i = \frac{c_i}{2\log n} = \frac{n}{2^i \cdot \log n}$$

many $\tau(\log n)$-registers (or blocks). Each of the blocks is responsible for $\log n$ names.

Algorithm 3 Tight renaming procedure for every process

```
1:  for i ∈ {1, ..., R} do
2:      for j ∈ {1, ..., k} do
3:          test any TAS bit in cluster C_i at random
4:          if TAS bit has been won then
5:              check control bit
6:              if control bit has been set then
7:                  search associated TAS registers
8:                  take first free name found
9:                  return success
10: for all TAS registers r ∈ C_{R+1} do
11:     test r
12:     if r is won then
13:         take name in r
14:         return success
15: return failure
```

Algorithm 3 is executed by every process and proceeds in rounds. As long as a process is *active* (initially all n of them), it probes the clusters in

the order of their indices (first loop) and every cluster a constant number of times (second loop). Hence, denoting the number of clusters by R and the number of constant tries per cluster by k, the maximum number of rounds is $R \cdot k = \mathcal{O}(\log n)$.

In every such round all active processes randomly pick one `TAS` bit from the current cluster (line 3). Each `TAS` bit having received at least one request accepts an arbitrary one of those. Each $\tau(\log n)$-register keeps at most $\log n$ many successful requests (we refer to this as the *block discarding step*). A process that has won the `TAS` bit checks the control bit of the $\tau(\log n)$-register (see Section 4.1). If the control bit is set, it was successful. It picks a name from the associated `TAS` register and becomes *inactive* (lines 7-9).

We will show that every $\tau(\log n)$-register receives at least $k \log n$ many process requests in expectation, and infer that, in each $\tau(\log n)$-register, at least half of the `TAS` bits receive at least one request with high probability, so that after the block discarding step we have *precisely* $\log n$ accepted requests per block. In other words, the idea is to choose cluster sizes such that w.h.p. each block in a given cluster receives just sufficiently many requests.

If a process is not successful probing the first R clusters, it will search the remaining `TAS` registers for a name. We will show in Observation 3 that the number of these registers is $\mathcal{O}(\log n)$.

We should like to point out that whilst we talk about *rounds* as though we have a synchronized protocol, this is in fact not the case. We use the notion only for ease of presentation. In reality, each process first tries cluster C_1, then cluster C_2, and so forth, until successful. In this sense the processes do operate in phases as indicated, but independent of one another. The algorithm is therefore *wait-free*.

The main result of this section is Theorem 5. To prove it, we will use the following observation and lemma.

Observation 3. *The cluster C_R and C_{R+1} have size*

$$c_R = c_{R+1} = \mathcal{O}(\log n).$$

Proof. C_{R+1} and C_R have the same size because

$$c_{R+1} = 2n - \sum_{i=1}^{R} c_i = 2n - \sum_{i=1}^{R} \frac{2n}{2^i} = \frac{2n}{2^R} = c_R.$$

Since R is defined as

$$R = \frac{\log n - \log\log n}{\log 2} = \log_2 \frac{n}{\log n}$$

it follows that

$$c_R = c_{R+1} = \frac{2n}{2^R} = 2\log n = \mathcal{O}(\log n).$$

□

Lemma 4. *Let ℓ be an arbitrary, positive constant, and let $c \geq \ell+2$. Suppose $2c\log n$ balls are allocated into $2\log n$ bins independently and uniformly at random. With probability at least $1 - 1/n^\ell$, there are no more than $\log n$ empty bins.*

Proof. For $1 \leq i \leq 2\log n$, let X_i be a binary random variable with $X_i = 1$ if and only if the i-th bin remains empty. Let

$$X = \sum_{i=1}^{2\log n} X_i$$

denote the number of empty bins. The probability that a ball is allocated to the i-th bin is $\frac{1}{2\log n}$, and since $2c\log n$ balls are allocated in total, the expected value $\mathbb{E}[X_i]$, $1 \leq i \leq 2\log n$, is

$$\mathbb{E}[X_i] = \mathbf{Pr}\left[X_i = 1\right] = \left(1 - \frac{1}{2\log n}\right)^{2c\log n} < \frac{1}{e^c}.$$

This implies

$$\mathbb{E}[X] = \sum_{i=1}^{2\log n} \mathbb{E}[X_i] < \frac{2\log(n)}{e^c}.$$

We wish to apply a Chernoff-type bound to X, but clearly the X_i are not independent. It is, however, well-known (see e.g. Theorem 46 on page 21 of [15]) that they are *negatively associated*, which immediately implies that we may use any Chernoff bound (normally requiring independent random variables) of our choosing. Intuitively, negative association of a collection of random variables means that if we know some subset of the variables to have "large" values, then this decreases the probability of another, disjoint subset to take "large" values as well – in our case, if a subset of bins remains empty

(with their $X_i = 1$), then another subset is less likely to remain empty as well (with *their* $X_i = 1$).

The remainder of the proof is now a mere formality. We apply the generic version of the Chernoff bound (given in Lemma 2) and choose $\delta = e^c/2 - 1$ so that $(1+\delta) \cdot \frac{2 \log n}{e^c} = \log n$. Notice that $c > \ln(2)$ implies $\delta = e^c/2 - 1 > 0$.

$$\begin{aligned}
\mathbf{Pr}\left[X \geq \log n\right] &= \mathbf{Pr}\left[X \geq (1+\delta) 2\log(n)/e^c\right] \\
&\leq \left(\frac{e^{(e^c/2-1)}}{(e^c/2)^{(e^c/2)}}\right)^{2\log(n)/e^c} = \left(\frac{e^{(e^c/2-1)2/e^c}}{(e^c/2)^{(e^c/2)2/e^c}}\right)^{\log n} \\
&= \left(\frac{e^{(1-2/e^c)}}{e^c/2}\right)^{\log n} = \frac{2^{\log n}}{n^{c-1+2/e^c}} \\
&\leq \frac{n^{1/\log_2 e}}{n^{c-1}} < n^{1.7-c} \\
&< n^{-\ell}
\end{aligned}$$

The last inequality follows from our constraint on c in the statement of this lemma. □

We are now ready to state and prove the main result of this section.

Theorem 5. *With high probability, Algorithm 3 assigns n processes to a name space of size n within $\mathcal{O}(\log n)$ steps per process, using $\mathcal{O}(n)$ extra space. The total step complexity is $\mathcal{O}(n)$.*

Proof. The space can be bounded by $\mathcal{O}(n)$ because storing the names in $\tau(\log n)$-registers increases the required space only by a constant factor. The step complexity of $\mathcal{O}(\log n)$ follows from the fact that every process following Algorithm 3 probes $R = \mathcal{O}(\log n)$ clusters and in every cluster only a constant number of `TAS` bits. When it wins a name in the first R clusters or in the last cluster C_{R+1}, it will retrieve the name in $\mathcal{O}(\log n)$ steps (see Section 4.1 and Observation 3). So, for the first claim it remains to show that every name is assigned with high probability. For this we will bound the probability of not assigning all names for every block in every cluster (except for C_{R+1}).

Every cluster C_i, $1 \leq i \leq R$, has $c_i = 2n/2^i$ `TAS` bits and $b_i = n/(2^i \cdot \log n)$ blocks. While it has $c_i/2$ names, it receives requests from at least c_i many processes which can be shown by induction: *Basis*: Cluster C_1 gets requests from $c_1 = n$ processes and has $c_1/2 = n/2$ names. *Step*: If cluster C_i gets

requests from at least c_i processes for $c_i/2$ names, then at least $c_{i+1} = c_i/2$ processes will proceed to the next cluster C_{i+1} which has $c_{i+1}/2$ names.

We let every process send up to $4c$ requests per cluster C_i, where c is a constant. If it gets a name before, it will stop, but at least half of the processes will send all requests, and therefore each of the b_i many blocks receives at least $\frac{4c}{2} \cdot \frac{2n}{2^i} \cdot \frac{2^i \cdot \log n}{n} = 4c \cdot \log n$ requests in expectation. Let X denote the random variable counting the requests one block receives. Applying a Chernoff bound of Lemma 2, we obtain:

$$\begin{aligned}\mathbf{Pr}\left[X \leq 2c \cdot \log n\right] &= \mathbf{Pr}\left[X \leq E[X]/2\right] \\ &\leq e^{-\frac{4c \cdot \log n}{3} \cdot \left(\frac{1}{2}\right)^2} \\ &= n^{-c/3}\end{aligned}$$

If each block receives $2c \log n$ requests, then, according to Lemma 4, half of the $2 \log n$ `TAS` bits in each block will receive at least one request with probability $1 - n^{-c+2}$. Consequently, after the last block discarding step, precisely $\log n$ of the $2 \log n$ `TAS` bits in each block will have accepted a request with probability at least

$$1 - n^{-c/3} - n^{-c+2}.$$

A union bound over all blocks proves the first claim for c large enough.

The total step complexity follows from the fact that only c_i many processes perform the (at most) k steps on cluster C_i:

$$\sum_{i=1}^{R} c_i \cdot k + c_{R+1}^2 = \sum_{i=1}^{R} \frac{2n}{2^i} \cdot k + (2 \log n)^2 \leq 2nk + 4 \log^2 n.$$

□

6. Loose Renaming in the Standard Model

In this section we consider the problem of loose renaming where the name space is larger than the number of processes n. In [10] the authors propose a loose renaming algorithm of step complexity $\mathcal{O}(\log \log n)$ that uses a name space of size $(1 + \epsilon) \cdot n$ where $\epsilon > 0$ is an arbitrary constant. Assuming the same model (which is described in Section 3.1), we introduce two renaming algorithms using smaller name spaces.

In both cases, we first present an algorithm that renames most but not all of n processes using a name space of size n. We call such a renaming algorithm *k-almost tight* if it assigns a name to all but k processes, with $k = o(n)$. In Lemma 6 and 8, respectively, we show that Algorithm 4 and 5 meet this condition.

To supply names to all processes, we then apply the method from [10] and assign to the remaining processes names from an additional name space starting at $n + 1$. The respective results are stated in Theorem 7 and 9.

Note that one can also apply the framework of [10] to transform our algorithms into adaptive algorithms when the number k of active processes that are looking for a name is not known in advance. Unfortunately, the name space would become $\mathcal{O}((1 + \epsilon) \cdot k)$ where ϵ is an arbitrary constant. Hence, using our protocols would not result in an improvement compared to [10].

Algorithm 4 Almost tight renaming procedure for each process (used in proof of Lemma 6)

```
1: for 2 · (log log n)^ℓ times do
2:     test a TAS register r randomly chosen from all n TAS registers
3:     if r has been won then
4:         take name in r
5:         return success
6: return failure
```

Lemma 6. *Assume we have n test-and-set registers and n processes. Then $\frac{n}{(\log\log n)^\ell}$-almost tight renaming can be done w.h.p. in the adaptive adversary model with a step complexity of*

$$\mathcal{O}((\log\log n)^\ell).$$

Proof. We use Algorithm 4 which is very simple as every process randomly selects from all n `TAS` registers in every step. For the analysis we divide the steps into $\ell \log_2 \log\log n$ many phases. Phase i has 2^i many steps. In every step of each phase, every register receiving requests is set by an arbitrary one of the accessing processes (and remains set for the rest of the algorithm). This process takes the name and becomes *inactive*. As given in the algorithm,

the total number of steps per process, i.e. the step complexity, is at most

$$\sum_{i=1}^{\ell \log_2 \log\log n} 2^i \leq 2 \cdot (\log\log n)^\ell.$$

We call phase i *successful* if, at the end of phase i, there are at most $n/2^i$ processes that are not assigned to a register. If all $\ell \log_2 \log\log n$ phases are successful, there will be

$$\frac{n}{2^{\ell \log_2 \log\log n}} = \frac{n}{(\log\log n)^\ell}$$

processes left which are not assigned to a name, and thus the renaming will be $\frac{n}{(\log\log n)^\ell}$-almost tight.

In the following we prove by contradiction that every phase is w.h.p. successful. Fix a phase i, $1 \leq i \leq \ell \log_2 \log\log n$, and assume that phase i is the first phase which is not successful. We can assume that during every step of phase i, we have at least $n/2^i$ active processes (otherwise phase i is successful) and unset registers. Hence, the total number of random choices in phase i is at least

$$\frac{n}{2^i} \cdot 2^i = n.$$

The probability that an arbitrary unset register r_j does not receive any of the requests is at most

$$\left(1 - \frac{1}{n}\right)^n \leq \frac{1}{e}.$$

For each r_j let X_{ij} be the binary random variable that is 1 if r_j remains unset and 0 otherwise. Let $X_i = \sum_j X_{ij}$ be the random variable that counts the number of unset registers at the end of phase i. Then

$$\mathbb{E}[X_i] \leq \frac{n}{2^{i-1}} \cdot \left(\frac{1}{e}\right).$$

Since the X_{ij} are negatively associated and since $\mathbb{E}[X_i] \geq n/((\log\log n)^\ell)$, we can apply Chernoff bounds (Lemma 2) and obtain

$$\begin{aligned}\mathbf{Pr}\left[X_i \geq \frac{n}{2^i}\right] &\leq \mathbf{Pr}\left[X_i \geq \mathbb{E}[X_i] \cdot \left(1 + \frac{e-2}{2}\right)\right] \\ &\leq e^{-\frac{n}{(\log\log n)^\ell} \cdot \frac{(e-2)^2}{4 \cdot 3}}.\end{aligned}$$

So, w.h.p. the number of unset registers at the end of the phase is at most $\frac{n}{2^i}$, meaning that the phase is successful. Now we can use the union bound over all phases i to show the lemma. □

Theorem 7. *Assume n processes and a name space of size*

$$n \cdot \left(1 + \frac{2}{(\log\log n)^\ell}\right).$$

Then, w.h.p., loose renaming can be done in the adaptive adversary model in linear space with a step complexity of

$$\mathcal{O}((\log\log n)^\ell).$$

The total step complexity is

$$\mathcal{O}(n \log\log\log n).$$

Proof. First Algorithm 4 is used to assign a name to all but $n/(\log\log n)^\ell$ many of the processes in $\mathcal{O}((\log\log n)^\ell)$ steps (Lemma 6). In a second step, the `ReBatching` algorithm (Algorithm 1) of [10] assigns names to the remaining unnamed processes from the name space $n+1$ to $n+2n/(\log\log n)^\ell$. The `ReBatching` algorithm has a step complexity of $\mathcal{O}(\log\log n)$ (Theorem 1) so that the step complexity of both algorithms combined is still $\mathcal{O}((\log\log n)^\ell)$.

For the total step complexity, consider the $\ell \log_2 \log\log n$ phases defined in the proof of Lemma 5. Assuming that every phase is successful, the maximum number of steps in Algorithm 4 is

$$\sum_{i=1}^{\ell \log_2 \log\log n} \frac{n}{2^{i-1}} \cdot 2^i = 2n\ell \log_2 \log\log n,$$

and the `ReBatching` algorithm on the reduced name space has a total step complexity of $o(n)$.

The number of `TAS` registers and the space required are linear because algorithm 4 does not require any space other than the `TAS` registers storing the names, and the space requirement of the `ReBatching` algorithm is also linear in the size of the name space. □

Algorithm 5 Almost tight renaming procedure for each process (used in proof of Lemma 8 which also defines the clusters C_i)

```
1: for i = 1, ..., 2ℓ log log n do
2:     for j = 1, ..., 2ℓ log log n do
3:         test a TAS register r randomly chosen from all TAS registers in C_i
4:         if r has been won then
5:             take name in r
6:             return success
7: return failure
```

Lemma 8. *Assume that a renaming instance is given with n `TAS` registers and n processes. Then, w.h.p., $n/(\log n)^\ell$-almost tight renaming can be done in the adaptive adversary model with a step complexity of*

$$(2\ell \log\log n)^2.$$

Proof. We use Algorithm 5. This algorithm works in $2\ell \log\log n$ phases for which the registers are divided into a sequence of clusters. For $1 \leq i \leq 2\ell \log\log n$, the ith cluster contains $n/2^i$ many registers. In phase i the processes randomly choose registers from the ith cluster only. Every phase consists of $2\ell \log\log n$ many steps. In every step of every phase i, all unnamed processes send a request to a randomly chosen `TAS` register from cluster i. A register receiving requests is set by one of the accessing processes, and the corresponding process becomes inactive.

At the beginning of phase $i \geq 2$ there are at least

$$n - \sum_{j=1}^{i-1} \frac{n}{2^j} = \frac{n}{2^{i-1}}$$

many active processes. At the end of phase i at least $n/2^i$ active processes are left which implies that at least $n/2^i \cdot 2\ell \log\log n$ requests are sent in phase i. The probability for a register in cluster i to receive no request is therefore at most

$$\left(1 - \frac{2^i}{n}\right)^{\frac{n}{2^i} \cdot 2\ell \log\log n} \leq e^{-2\ell \log\log n} = \frac{1}{(\log n)^{2\ell}}.$$

Let X count the number of unvisited registers in all clusters. Then we

have $\mathbb{E}[X] \leq n/(\log n)^{2\ell}$ and, applying Chernoff bounds (Lemma 2),

$$\begin{aligned}
\mathbf{Pr}\left[X \geq \frac{n}{(\log n)^{3\ell/2}}\right] &= \mathbf{Pr}\left[X \geq (1+\delta)\cdot E[X]\right] \\
&\leq e^{-\delta E[X]/3} \\
&\leq e^{-\left(\frac{n}{(\log n)^{3\ell/2}} - E[X]\right)\cdot\frac{1}{3}} \\
&\leq e^{-\left(\frac{n}{(\log n)^{3\ell/2}} - \frac{n}{(\log n)^{2\ell}}\right)\cdot\frac{1}{3}} \\
&\leq e^{-n\cdot\left(\frac{(\log n)^{\ell/2}-1}{3\cdot(\log n)^{2\ell}}\right)}.
\end{aligned}$$

Since the number of registers outside of the clusters is $n/2^{2\ell\log\log n} < 2n/(\log n)^{2\ell}$, the number of unvisited registers and thus of unnamed processes is w.h.p. at most

$$\frac{n}{(\log n)^{3\ell/2}} + \frac{2n}{(\log n)^{2\ell}} < \frac{n}{(\log n)^{\ell}}$$

for n large enough. □

Theorem 9. *Assume, we have $n + 2\cdot\frac{n}{(\log n)^{\ell}}$ test-and-set registers and n processes. Then, w.h.p., loose renaming can be done in the adaptive adversary model in linear space with a step complexity of*

$$\mathcal{O}((\log\log n)^2).$$

The total step complexity is

$$\mathcal{O}(n\log\log n).$$

Proof. The proof is very similar to the one of Theorem 7 and applies two algorithms. The first is Algorithm 5 which assigns a name to all but $n/(\log n)^{\ell}$ many of the processes (Lemma 8). The second one is the `ReBatching` algorithm (Algorithm 1) of [10] which assigns names to the remaining unnamed processes from the name space $n+1$ to $n + 2n/(\log n)^{\ell}$. The `ReBatching` algorithm has a step complexity of $\mathcal{O}(\log\log n)$ on the reduced name space (Theorem 1) so that the step complexity of both algorithms combined is $\mathcal{O}((\log\log n)^2)$.

For the total step complexity, we sum up the maximum number of steps per phase over all phases and obtain

$$\sum_{i=1}^{2\ell\log\log n} \frac{n}{2^{i-1}}\cdot 2\ell\log\log n \leq n\cdot 4\ell\log\log n.$$

The `ReBatching` algorithm on the reduced name space has a total step complexity of $o(n)$.

The number of `TAS` registers and the space required are linear because algorithm 5 does not require any space other than the `TAS` registers storing the names, and the space requirement of the `ReBatching` algorithm is also linear in the size of the name space. □

7. Conclusion

In this paper we have considered the renaming problem in the asynchronous shared-memory model. By utilizing new hardware features and extending the concept of the test-and-set register, we have shown that even a fairly straightforward randomized algorithm can perform tight renaming in $\mathcal{O}(\log n)$ steps with high probability. The hardware added is a set of register clusters, each containing $\log n$ names, which increase the success probability for the random accesses of the processes by seemingly enlarging the name space.

Our solutions to the loose renaming problem work in the standard model in which the names are stored in "plain" test-and-set registers. The algorithms are the first to achieve almost tight renaming in poly-double-logarithmic step complexity mapping n names to a name space of size only $(1 + o(1)) \cdot n$.

While there is a known matching lower bound for loose renaming, it remains open to show that the lower bound for tight renaming can be extended to the τ-register. An interesting future task will be the exploration of modern hardware capabilities and how new features can improve solutions to fundamental problems in distributed computing.

References

[1] S. Chaudhuri, M. Herlihy, M. R. Tuttle, Wait-free implementations in message-passing systems., Theor. Comput. Sci. 220 (1) (1999) 211–245.

[2] M. Okun, Strong order-preserving renaming in the synchronous message passing model., Theor. Comput. Sci. 411 (40-42) (2010) 3787–3794.

[3] D. Alistarh, O. Denysyuk, L. Rodrigues, N. Shavit, Balls-into-leaves: Sub-logarithmic renaming in synchronous message-passing systems, in: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC '14, ACM, New York, NY, USA, 2014, pp. 232–241.

[4] D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu, R. Guerraoui, Fast randomized test-and-set and renaming., in: N. A. Lynch, A. A. Shvartsman (Eds.), DISC, Vol. 6343 of Lecture Notes in Computer Science, Springer, 2010, pp. 94–108.

[5] A. Brdosky, F. Ellen, P. Woelfel, Fully-adaptive algorithms for long-lived renaming, Distributed Computing 24 (2) (2011) 119–134.

[6] A. Castañeda, S. Rajsbaum, M. Raynal, The renaming problem in shared memory systems: An introduction, Comput. Sci. Rev. 5 (3) (2011) 229–251.

[7] D. Alistarh, J. Aspnes, S. Gilbert, R. Guerraoui, The complexity of renaming, in: Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 718–727.

[8] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, M. Zadimoghaddam, Optimal-time adaptive strong renaming, with applications to counting., in: C. Gavoille, P. Fraigniaud (Eds.), PODC, ACM, 2011, pp. 239–248.

[9] G. Giakkoupis, P. Woelfel, On the time and space complexity of randomized test-and-set, in: PODC, 2012, pp. 19–28.

[10] D. Alistarh, J. Aspnes, G. Giakkoupis, P. Woelfel, Randomized loose renaming in O(log log n) time, in: Proceedings of the 2013 ACM Symposium on Principles of distributed computing, PODC '13, ACM, New York, NY, USA, 2013, pp. 200–209.

[11] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, R. Guerraoui, Tight bounds for asynchronous renaming, J. ACM 61 (3) (2014) 18:1–18:51.

[12] P. Berenbrink, A. Brinkmann, R. Elsässer, T. Friedetzky, L. Nagel, Randomized renaming in shared memory systems, in: Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS '15, IEEE Computer Society, USA, 2015, p. 542–549.

[13] A. Panconesi, M. Papatriantafilou, P. Tsigas, P. M. B. Vitanyi, Randomized naming using wait-free shared variables, Distributed Computing 11 (3) (1998).

[14] W. Eberly, L. Higham, J. Warpechowska-Gruca, Long-lived, fast, wait-free renaming with optimal name space and high throughput, in: DISC, 1998, pp. 149–160.

[15] D. Dubhashi, D. Ranjan, Balls and Bins: A Study in Negative Dependence, BRICS, 1996.
URL http://www.brics.dk/RS/96/25/BRICS-RS-96-25.pdf