Brief Announcement: Optimal Construction of Unique Identifiers from Bounded Registers

Michael Anoprenko
Institut Polytechnique de Paris
Paris, France
manoprenko@gmail.com

Petr Kuznetsov*
Telecom Paris, Institut Polytechnique Paris
Paris, France
petr.kuznetsov@telecom-paris.fr

Vitaly Aksenov
City, University of London
London, UK
aksenov.vitaly@gmail.com

Abstract

In this paper, we describe an algorithm implementing the *unique-id* abstraction from bounded-storage registers maintaining *read*, *write*, and *FAI* operations. Given k registers, storing w bits each, our implementation generates up to $(k-1)\cdot 2^w$ unique identifiers, assuming that $k \leq 2^w + 1$. We show that this is asymptotically optimal: no unique-id implementation can produce more than $k \cdot 2^w + k + 1$ identifiers.

CCS Concepts

• **Computing methodologies** → **Concurrent algorithms**; • **Theory of computation** → **Shared memory algorithms**.

Keywords

concurrency, bounded memory, unique identifiers

ACM Reference Format:
Michael Anoprenko, Petr Kuznetsov, and Vitaly Aksenov. 2025. Brief Announcement: Optimal Construction of Unique Identifiers from Bounded Registers. In *ACM Symposium on Principles of Distributed Computing (PODC '25), June 16–20, 2025, Huatulco, Mexico.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3732772.3733539

1 Introduction

Building efficient and convenient abstractions from components of bounded capacity is the bread and butter of computer science, and distributed computing in particular. A notable example is the collection of algorithms by Lamport [12], implementing read-write registers of stronger consistency, larger capacity, or richer interface from weaker ones. These results have been followed by a long and prolific research line on implementing large-capacity abstractions from low-capacity components, touching upon various abstractions, consistency criteria, and capacity metrics [7–9, 15] (to name a few). Recently, this question was addressed in the context of combinatorial properties of generic distributed task protocols [3, 14] and also in the context of concurrent data structures with minimal memory footprint [1].

*This work was supported by the CHIST-ERA grant CHIST-ERA-22-SPiDDS-05 (REDONDA project) and Agence Nationale de la Recherche (ANR-23-CHR4-0009).

PODC '25, Huatulco, Mexico

ACM ISBN 979-8-4007-1885-4/25/06
https://doi.org/10.1145/3732772.3733539

In this paper, we focus on the problem of generating *unique identifiers* from shared registers of bounded storage capacity. The *unique-id* abstraction can be accessed with an operation *get*() that returns an integer *identifier* and ensures that in every execution, no two operations return the same id. This abstraction is widely used in databases to generate primary keys [6, 16, 17], distributed tracing [13], cloud orchestration [5], etc. The unique-id problem is reminiscent of the classical task of *renaming* [2]. Unlike renaming, however, our goal is not to provide unique names in a given (ideally, small) name range *once*, but to enable a sufficiently long supply of unique identifiers in the *long-lived* manner. Similarly to renaming algorithms, to filter out trivial (and, arguably, useless) solutions, we, however, do assume *anonymity*—the processes are not supposed to use their initial identifiers in deriving new ones.

We consider concurrent read-write *unique-id* implementations, assuming that each shared variable (a *register*) can store up to w bits and exports *read*, *write*, and *fetch-and-increment* (FAI) operations. Trivially, just applying FAI operations to one register, we can generate up to 2^w unique identifiers. But what happens once we exceed the capacity of a single register and more *get*() operations are applied? Assuming that we have k w-bit registers, how close can we get to the theoretical maximum of 2^{kw} identifiers?

As we show in this paper, in a system with sufficiently many concurrent processes, no unique-id implementation can generate more than $k \cdot 2^w + k + 1$ identifiers. We match this upper bound asymptotically with an algorithm that is able to generate up to $(k-1)\cdot 2^w$ ids for any concurrency level, as long as $k \leq 2^w + 1$.

We see this result as a first step in the direction of deriving practical abstractions with large capacity implemented using bounded memory. In contrast with the *universal* approach [3, 7, 14] that aims at implementing an arbitrary sequential object or solving an arbitrary decision task, we focus on the practical problem of generating identifiers in the pragmatic read/write/FAI model. Of course, many questions beg for answers. Can we refine our upper bound by introducing the concurrency level in it? What about models in which shared variables are equipped with more general operations, such as *fetch-and-add* (FAA) or *compare-and-swap* (CAS)? What about other problems, e.g., the stronger variations of *unique ids* that would generate monotonically increasing ids (or even *gap-free*, which is equivalent to a large-range counter)? What about the use of randomization, improving the recent *communication-free* solution [4] with shared-memory primitives? We expect these questions to be tackled in the future work.

2 Problem Statement

Consider a concurrent system, where a set of *processes* (or *threads*) communicate via k shared-memory registers $R_0, \ldots, R_{k-1}$. Each register stores a w-bit unsigned integer and can be accessed by atomic *read*, *write*, and *fetch-and-increment* (*FAI*) primitives. *FAI* works in a cyclic manner: applied to a register with value $2^w - 1$, *FAI* sets the register to 0.

Every thread is assigned an *algorithm*, informally—an automaton that, given the local process state, the outcome of the last shared-memory primitive and/or the application input, generates the next primitive to be executed on a shared-memory location and/or the application output. We assume that the automaton is *anonymous* (does not use the process identifiers) and *deterministic*.

A *UniqueID* object exports one operation *get*() that takes no arguments and returns an integer (an *identifier*). In a sequential execution, the object should ensure that all outputs of *get* operations are distinct. [1]

We aim at *linearizable* [11] implementations of this specification above, informally, ones that create an illusion of an object exporting an *atomic get* operation. An implementation can be *wait-free* (every thread completes its operation in finitely many of its own steps) or *lock-free* (i.e., at least one thread completes all of its operations) [9]. Our impossibility result assumes the even weaker liveness condition of *obstruction-freedom* [10] which only guarantees progress to a thread that runs *in isolation* for a sufficient amount of time.

In this paper, we determine upper and lower bounds on the number of distinct identifiers a *UniqueID* object can produce. Intuitively, we want to determine B (as a function of k and w) such that: (1) there is a *UniqueID* that is correct as long as at most B operations are invoked, and (2) no correct implementation exists when more than B operations are invoked.

In the following, we first show that there is a wait-free algorithm that is able to produce up to $(k-1) \cdot 2^w$ identifiers, and then show that no obstruction-free algorithm can produce more than $k \cdot 2^w + k + 1$ identifiers.

3 Wait-free ID Generation

We describe a simple algorithm that generates up to $B = (k-1) \cdot 2^w$ identifiers, assuming the number of registers $k \leq 2^w + 1$, we then discuss possible extensions for larger values of k. (In the trivial case of $k = 1$, we can get up to $B = 2^w$ simply applying FAI to the single register.)

First we present an algorithm that assumes that $k = 2^x + 1$ for some integer x. This assumption allows us to build a simple correspondance between values of the first register and indices of other registers. Later we discuss algorithms for arbitrary k with slight relaxations in progress guarantees or id ranges.

Algorithm 1 contains pseudocode for id generation in case $k = 2^x + 1$. The first register is used to distribute operations among $k-1$ remaining registers. Each of those $k-1$ registers gets its own designated value range from which it produces new IDs in the sequential order.

[1] One might also consider stronger sequential specifications: *monotonic* (the identifiers returned by the *get* operations monotonically grow) or even *continuously monotonic* (identifiers are produced in the incremental fashion).

Algorithm 1 Wait-free UniqueID for $k = 2^x + 1 \leq 2^w + 1$

Initially:
$R_i \leftarrow 0$ for all i

```
1: procedure GET()
2:     i ← R_0.FAI() mod (k − 1)
3:     j ← R_{i+1}.FAI()
4:     return i · 2^w + j
5: end procedure
```

Theorem 3.1. *Algorithm 1 is a wait-free* UniqueID *implementation using k w-bit registers, assuming that the get operation is called at most $(k-1) \cdot 2^w$ times.*

Proof. The algorithm is trivially wait-free. We show now that if the *get* operation is called no more than $B = (k-1) \cdot 2^w$ times, then all returned values are distinct.

Consider any execution with B invocations of *get*. We denote these operations $op_0, op_1, \ldots, op_{B-1}$, in the order of the execution of $R_0.FAI()$ on line 2. For every operation op_x denote as i_x the value of i assigned in the line 2 and as j_x the value of j assigned in the line 3.

From semantics and atomicity of the *FAI* operation on R_0 we can notice that $i_x = (x \bmod 2^w) \bmod (k-1) = x \bmod (k-1)$. Thus, for some fixed value i' operations with $i_x = i'$ have identifiers of the form $op_{i'+c\cdot(k-1)}$ for some non-negative integer c. Solving the inequality $i' + c \cdot (k-1) < B$ we obtain $c < \frac{B-i'}{k-1} \leq \frac{B}{k-1} = 2^w$. Thus, operation $R_{i'+1}.FAI()$ is called no more than 2^w times for any i'. As $R_{i'+1}$ stores 2^w values, no two operations $R_{i'+1}.FAI()$ can return two equal values meaning that all values of j will be distinct for such operations.

Thus, for any two operations op_f ans op_s we have $i_f \neq i_s$ or $j_f \neq j_s$. Since both i and j are values of register and don't exceed 2^w, we can infer that $i_f \cdot 2^w + j_f \neq i_s \cdot 2^w + j_s$ which in turn means that all the values returned by `get` operations are distinct. □

We propose two simple variations of Algorithm 1 for the case when $k-1$ is not a power of two.

As the first one, we run Algorithm 1 on the maximal $k' \leq k$ registers such that $k' - 1 = 2^x$ for some x. This approach reduces the possible number of generated values, but since $k' \geq k/2$, we get the number of operations $B \geq k/2 \cdot 2^w$.

Second, we can find the *minimal* $k' \geq k$ such that $k' - 1 = 2^x$ and generate values of i, taking the result of $R_0.FAI()$ modulo $(k'-1)$ instead of $(k-1)$ (see Algorithm 2). This will sometimes lead to situations when there's no register with index $i+1$ because $i+1 > k$. In this case, we can invoke $R_0.FAI()$ again. We repeat this operation until a valid index is obtained. This strategy will ensure that all operations are still evenly distributed between all $k-1$ registers even when occasional overflows of R_0 occur. This algorithm does not have a constant bound on the number of steps required to complete each operation. Although a process might starve due to the progress of other processes, the algorithm remains wait-free since the object produces a finite number of distinct identifiers.

Remark. The reason Algorithm 2 does not work for $k > 2^w + 1$ is that by performing *FAI()* operation on a single register R_0, we can

Algorithm 2 UniqueID for $k \leq 2^w + 1$

Initially:
$R_i \leftarrow 0$ for all i
$k' = 2^{\lceil \log_2 (k-1) \rceil} + 1$

1: **procedure** GET()
2: **repeat**
3: $i \leftarrow R_0.FAI() \bmod (k' - 1)$
4: **until** $i < k - 1$
5: $j \leftarrow R_{i+1}.FAI()$
6: **return** $i \cdot 2^w + j$
7: **end procedure**

only generate 2^w distinct values. To make the algorithm operate on $k > 2^w + 1$ registers, we can use a *UniqueID* object (instead of a register) and invoke *get()* operation on it. If our implementation of UniqueID additionally guarantees that it returns values in range $[0, B-1]$ and can be reused after the whole range is exhausted, then by replacing the call to $R_0.FAI()$ in Algorithm 2 with $UniqueID.get()$ we can increase the number of generated ids. Algorithm 2 satisfies these properties, meaning that it can be recursively used within itself to generate ids in larger range in cases when $k > 2^w + 1$. For example, we can use three registers to implement a UniqueID object that would generate values from 0 to $2^{w+1} - 1$ and then use 2^{w+1} registers to generate actual ids. This gives us an algorithm that is capable of generating $(k-3) \cdot 2^w$ ids for $k \leq 2^{w+1} + 3$. We delegate the proof and the analysis of trade-offs of this solution to future work.

4 Impossibility Result

In this section, we show that the optimal upper bound B does not grow exponentially with k and is asymptotically smaller than the theoretical maximum of 2^{kw} for k registers of w bits each. This impossibility result applies to UniqueID implementations that are **even** obstruction-free (only guarantee progress to the processes that eventually run in isolation).

Theorem 4.1. *In a system with sufficiently many concurrent processes, no obstruction-free implementation of UniqueID using k w-bit registers allows the get() function to be called more than $k \cdot 2^w + k + 1$ times.*

Proof. By contradiction, suppose that there is a UniqueID algorithm that produces more than $B = k \cdot 2^w + k + 1$ identifiers. To establish a contradiction, we launch a series of concurrent *get*() operations. Once we launched an operation, we track all *updates* (*write* and *FAI* primitives) this operation performs. In certain cases, we will suspend the process that invoked the operation just before it performs an update primitive on some register R_i, and we say that the process is *poised* to perform the primitive on R_i. A process p_j is poised on R_i if:

(1) p_j is about to perform a *FAI()* primitive on R_i, and there are less than 2^w processes poised on R_i or
(2) p_j is about to perform a *write*(x) primitive on R_i, and there are no other processes poised on R_i with a *write*(x) primitive.

Claim 1. *Suppose that 2^w processes are poised on a register R_i. Then R_i can be set to any desired value simply by allowing some poised processes to perform their primitives.*

Proof. Recall that all values of *write* operations are distinct. Let f be the number of poised processes that are about to perform *FAI* primitives, then the remaining $2^w - f$ poised processes are about to perform *writes* with distinct values $\{w_1, w_2, \ldots, w_{2^w - f}\} = W$. Consider a value $x \in \{0, \ldots, 2^w - 1\}$. But, as $|W| = 2^w - f$, at least one value in $\{x, x-1, \ldots, x-f\}$ should be in W, therefore R_i can be set to x by resuming poised *write*$(x - f')$ $(0 \leq f' \leq f)$ and then scheduling f' poised FAI primitives. □

One after another, we launch new *get*() operations in a separate process and run in isolation. As the algorithm is obstruction-free, a process running an operation either eventually returns from it or is poised on a register.

Suppose that a process returns from the operation. If, within the execution, the process has performed *the first* update on some register (i.e., that register has never been modified before), we call it *special*. (There cannot be more than k special processes.)

Consider the first *non-special* process which has finished its *get*() operation without getting poised. We can show that once this process p_j finished its operation, it is possible to restore the state of the memory that occurred before this process was started.

Indeed, consider any register R_i that p_j has modified, and suppose p_j's *first* update on R_i changed the register's state from x to y. As p_j is not special, R_i has been modified before this. Hence, the last update primitive performed on R_i before the modification of R_i by p_j set the value of the register to x.

Let p_k be the process that performed this update on R_i. As p_k was not poised just before performing this update, then either (1) there were already at least 2^w operations poised on R_i, or (2) p_k performed *write*(x) and there was already another *write*(x) primitive poised on R_i.

In case (2), by scheduling the poised *write*(x), we bring R_i back to x, the state of R_i before p_j started its *get*() operation. In case (1), by Claim 1, by scheduling the steps of some of the poised on R_i processes, we can bring the state of R_i back to x. Thus, every trace of a non-special process can be wiped out from the memory by the poised processes.

As the algorithm is deterministic, obstruction-free, and anonymous, a *get*() operation by another process run in isolation from the restored state will bring the same output as p_j.

Let us calculate the number of *get*() operations invoked in the incorrect execution we constructed. There are the operations of at most 2^w processes poised on every register (up to $k \cdot 2^w$ altogether), up to k operations of the special processes and, finally, the two operations that returned the same value—$2^w \cdot k + k + 2$ operations in total. Thus, the total number of distinct identifiers the algorithm can produce cannot exceed $B \leq k \cdot 2^w + k + 1$. □

References

[1] Vitaly Aksenov, Nikita Koval, Petr Kuznetsov, and Anton Paramonov. 2024. Memory Bounds for Concurrent Bounded Queues. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 188–199. https://doi.org/10.1145/3627535.3638497

[2] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. 1990. Renaming in an Asynchronous Environment. *J. ACM* 37, 3 (1990), 524–548.
[3] Carole Delporte-Gallet, Hugues Fauconnier, Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers. 2024. The Computational Power of Distributed Shared-Memory Models with Bounded-Size Registers. In *PODC*, Ran Gelles, Dennis Olivetti, and Petr Kuznetsov (Eds.). ACM, 310–320.
[4] Peter C. Dillinger, Martín Farach-Colton, Guido Tagliavini, and Stefan Walzer. 2023. Optimal Uncoordinated Unique IDs. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Seattle, WA, USA) *(PODS '23)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/3584372.3588674
[5] Kubernetes Documentation. 2024. Object Names and IDs. https://kubernetes.io/docs/concepts/overview/working-with-objects/names/. Online; accessed 2025-02-07.
[6] PostgreSQL Documentation. 2024. UUID type. https://www.postgresql.org/docs/current/datatype-uuid.html. Last accessed: 2025-02-07.
[7] Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleni Kanellou. 2019. An Efficient Universal Construction for Large Objects. In *OPODIS (LIPIcs, Vol. 153)*, Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:15.
[8] S. Haldar and K. Vidyasankar. 1995. Constructing 1-writer Multireader Multivalued Atomic Variables from Regular Variables. *J. ACM* 42, 1 (Jan. 1995), 186–203.
[9] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 13, 1 (1991), 123–149.
[10] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2003. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *ICDCS*. 522–529.
[11] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.
[12] Leslie Lamport. 1978. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks* 2 (1978), 95–114.
[13] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. http://research.google.com/archive/papers/dapper-2010-1.pdf
[14] Guillermo Toyos-Marfurt and Petr Kuznetsov. 2024. On the Bit Complexity of Iterated Memory. In *SIROCCO (Lecture Notes in Computer Science, Vol. 14662)*, Yuval Emek (Ed.). Springer, 456–477.
[15] John Tromp. 1989. How to Construct an Atomic Variable (Extended Abstract). In *WDAG*, Jean-Claude Bermond and Michel Raynal (Eds.), Vol. 392. Springer, 292–302.
[16] Transact-SQL Documentation; https://learn.microsoft.com/en-us/sql/t-sql/functions/newid-transact-sql?view=sql-server-ver16. 2024. NEWID function. Last accessed: 2025-02-07.
[17] MongoDB Manual; https://www.mongodb.com/docs/manual/reference/method/ObjectId/. 2024. ObjectID. Last accessed: 2025-02-07.