Fast, Long-Lived Renaming*

(Extended Abstract)

Mark Moir and James H. Anderson

Department of Computer Science
The University of North Carolina at Chapel Hill
Chapel Hill, North Carolina 27599-3175, USA

April 1994

Abstract

We consider wait-free solutions to the renaming problem for shared-memory multiprocessing systems [3, 5]. In the renaming problem, processes are required to choose new names in order to reduce the size of their name space. Previous solutions to the renaming problem have time complexity that is dependent on the size of the original name space, and allow processes to acquire names only once. In this paper, we present several new renaming algorithms. Most of our algorithms have time complexity that is independent of the size of the original name space, and some of our algorithms solve a new, more general version of the renaming problem called long-lived renaming. In long-lived renaming algorithms, processes may repeatedly acquire and release names.

Keywords: fast renaming, long-lived renaming, shared-memory multiprocessors, synchronization primitives, wait-free synchronization

1 Introduction

In the *M-renaming* problem [2], each of k processes is required to choose a distinct value, called a *name*, that ranges over $\{0, ..., M-1\}$. Each process is assumed to have a unique process identifier ranging over $\{0..N-1\}$. It is further required that $k \leq M < N$. Thus, an M-renaming algorithm is invoked by k processes in order to reduce the size of their name space from N to M .

Renaming is useful when processes perform a computation whose time complexity is dependent on the size of the name space containing the processes. By first using an efficient renaming algorithm to reduce the size of the name space, the time complexity of that computation can be made independent of the size of the original name space.

The renaming problem has been studied previously for both message-passing [2] and shared-memory multiprocessing systems [3, 5]. In this paper, we consider wait-free implementations of renaming in asynchronous, shared-memory systems. A renaming algorithm is *wait-free* iff each process is guaranteed to acquire a name after a finite number of that process's steps, regardless of the execution speeds of other processes.

Previous research on the renaming problem has focused on *one-time* renaming: each process acquires a name only once. In this paper, we also consider *long-lived* renaming, a new, more general version of renaming in which processes may repeatedly acquire and release names.

*Work supported, in part, by NSF Contract CCR 9216421. Email: {anderson,moir}@cs.unc.edu. Phone: (919)962-1757. Fax: (919)962-1799

Reference	M	Time Complexity	Long-Lived?
[3]	$k(k+1)/2$	$O(Nk)$	No
[3]	$2k-1$	$O(N4^k)$	No
[5]	$2k-1$	$O(Nk^2)$	No
Thm. 1	$k(k+1)/2$	$O(k)$	No
Thm. 2	$2k-1$	$O(k^4)$	No
Thm. 3	$k(k+1)/2$	$O(Nk)$	Yes

Table 1: A comparison of M-renaming algorithms that employ only atomic reads and writes.

A solution to the long-lived renaming problem is useful in settings in which processes repeatedly access identical resources. The specific application that motivated us to study this problem is the implementation of shared objects. The complexity of a shared object implementation is often dependent on the size of the name space containing the processes that access that implementation. For such implementations, performance can be improved by restricting the number of processes that concurrently access the implementation, and by using long-lived renaming to acquire a name from a reduced name space. This is the essence of a methodology we are developing for the implementation of resilient, scalable shared objects. For details of this methodology, and the use of long-lived renaming, see [1]. In that paper, we presented a simple long-lived renaming algorithm. To our knowledge, this is the only previous work on long-lived renaming. In this paper, we present several new long-lived renaming algorithms, one of which is a generalization of the algorithm we presented in [1].

In the first part of the paper, we present three renaming algorithms that use only atomic read and write instructions. It has been shown that if $M < 2k-1$, then M-renaming cannot be implemented in a wait-free manner using only atomic reads and writes [6]. Previous papers have given wait-free algorithms for one-time renaming that employ only reads and writes [3, 5], including some that have an optimal name space of $M = 2k-1$. However, in all of these algorithms, the time complexity of choosing a name is dependent on N. Thus, these algorithms suffer from the same shortcoming that the renaming problem is intended to overcome, namely time complexity that is dependent on the size of the original name space.

Our read/write algorithms for one-time renaming yield various-sized name spaces, including $M = 2k-1$. In contrast to prior algorithms, our one-time algorithms have time complexity that depends only on k, the number of participating processes. These algorithms employ a novel technique that uses "building blocks" based on the "fast path" mechanism employed by Lamport's fast mutual exclusion algorithm [7].

We also present a read/write algorithm for long-lived renaming that yields a name space of size $k(k+1)/2$. This algorithm uses a modified version of the one-time building block that allows processes to "reset" the building block, so that it may be used repeatedly. Unfortunately, this modification results in time complexity that is dependent on N. Nevertheless, this result breaks new ground by showing that long-lived renaming can be implemented with only reads and writes.

Previous and new renaming algorithms that use only read and write operations are summarized in Table 1. We leave open the question of whether read and write operations can be used to implement long-lived renaming with a name space of size $2k-1$ and with time complexity that depends only on k.

In many applications of renaming, it is important to achieve a name space that is as small as possible. For example, in our methodology for implementing scalable, resilient shared objects [1], a process accesses a wait-free shared object implementation using a name acquired from a renaming algorithm as a process identifier. The time complexity of such wait-free shared object implementations often increases dramatically with the size of the name space containing the processes that access the implementation. We are therefore motivated to design M-renaming algorithms for the smallest possible value of M. By definition, M-renaming for $M < k$ is impossible, so with respect to the size of the name space, k-renaming is optimal.

In the second part of the paper, we consider long-lived k-renaming algorithms. As was previously men-

Reference	Time Complexity	Bits / Variable	Instructions Used
Thm. 4	$O(k)$	1	write and test_and_set
Thm. 4	$O(k/b)$	b	set_first_zero and clear_bit
Thm. 5	$O(\log k)$	$O(\log k)$	bounded_decrement and atomic_add
Thm. 6	$O(\log(k/b))$	$O(\log k)$	above, set_first_zero, and clear_bit

Table 2: A comparison of long-lived k-renaming algorithms.

tioned, it is impossible to implement k-renaming using only atomic read and write operations. Thus, all of our k-renaming algorithms employ stronger read-modify-write operations. Our long-lived k-renaming algorithms are summarized in Table 2.

The remainder of the paper is organized as follows. Section 2 contains definitions used in the rest of the paper. In Sections 3 and 4, we present one-time and long-lived renaming algorithms that employ only atomic reads and writes. In Section 5, we present long-lived renaming algorithms that employ stronger read-modify-write operations. Concluding remarks appear in Section 6.

2 Definitions

Our programming notation should be self-explanatory; as an example of this notation, see Figure 2. In this and subsequent figures, each line of a program is assumed to be executed atomically, unless otherwise specified. When reasoning about programs, we define safety properties using invariant assertions. A state assertion is an *invariant* iff it holds initially and is not falsified by any statement execution.

Notational Conventions: The predicate $p@i$ holds iff process p's program counter has the value i. We use $p@S$ as shorthand for $(\exists i : i \in S :: p@i)$. We use $p.i$ to denote statement i of process p, and $p.var$ to denote p's local variable var. It is assumed that $0 < k \leq M < N$, and that p and q range over $0..N-1$. $\Box$

In the *one-time M-renaming* problem, each of k processes with distinct process identifiers ranging over $\{0, ..., N-1\}$ chooses a distinct value ranging over $\{0, ..., M-1\}$. A solution to the M-renaming problem consists of a wait-free procedure *Getname* for each process p that, when called by process p, returns a value ranging over $\{0, ..., M-1\}$. We assume that each process calls its *Getname* procedure at most once. For $p \neq q$, the same value cannot be returned to both p and q.

In the *long-lived M-renaming* problem, each of N distinct processes repeatedly executes a remainder section, acquires a name by assigning $p.name := Getname()$, uses that name in a working section, and then releases the name by calling $Putname()$. The organization of these processes is shown in Figure 1. It is assumed that the remainder section guarantees that at most k processes are outside their remainder sections at any time. A solution to the long-lived M-renaming problem consists of wait-free *Getname* and *Putname* procedures, along with associated shared variables. If process p is in its working section, then it is required that $0 \leq p.name < M$. If distinct processes p and q are in their working sections, then it is further required that $p.name \neq q.name$.

Among other well-known operations, our algorithms use *set_first_zero*, *clear_bit*, and *bounded_decrement*. The *set_first_zero*(X, m) operation atomically accesses a variable of m-bits, which are indexed from 0 to $m-1$. If all bits of X are set, then *set_first_zero*(X, m) returns m and does not modify X. Otherwise, *set_first_zero*(X, m) sets the bit with the smallest index among clear bits in X and returns the index of that bit. The *clear_bit*(X, i) operation clears the ith bit of variable X. For $m = 1$, the *set_first_zero* and *clear_bit* operations can be trivially implemented using *test_and_set* and *write* operations. For $m > 1$, these operations can be implemented, for example, using the *atomff0andset* and *fetch_and_and* operations on the BBN TC2000 multiprocessor [4]. The *bounded_decrement* operation is similar to the commonly-available

```
process p                                                              /* 0 ≤ p < N */
private variable name : 0..M − 1                                       /* Name received */
    while true do
        Remainder Section;     /* Ensure at most k processes access renaming instance concurrently */
        name := Getname();                                   /* Get a name between 0 and M − 1 */
        Working Section;
        Putname(name)                                                  /* Release the name obtained */
    od
```

Figure 1: Organization of processes accessing a long-lived renaming algorithm.

fetch_and_decrement operation, except that it does not modify a variable whose value is zero.

We measure the time complexity of our algorithms in terms of the worst case number of shared variable accesses. A variable is considered to be *private* if at most one process can access that variable and *shared* otherwise. We measure the time complexity for one-time renaming as the worst case number of shared variable accesses executed by a process in acquiring a name. We measure the time complexity for long-lived renaming as the worst case number of shared variable accesses executed by a process in acquiring and releasing a name once.

3 One-Time Renaming using Atomic Reads and Writes

In this section, we present renaming algorithms that employ only atomic read and write operations. We start by presenting a one-time $(k(k+1)/2)$-renaming algorithm that has $O(k)$ time complexity. We then describe how this algorithm can be combined with previous results [5] to obtain a $(2k-1)$-renaming algorithm with $O(k^4)$ time complexity. It has been shown that renaming is impossible for fewer than $2k-1$ names when using only reads and writes, so with respect to the size of the resulting name space, this algorithm is optimal. In the next section, we present a long-lived $(k(k+1)/2)$-renaming algorithm that has time complexity $O(Nk)$. This is the first long-lived renaming algorithm that employs only read and write operations.

Our one-time $(k(k+1)/2)$-renaming algorithm is based on a "building block", which we describe next.

3.1 The One-Time Building Block

The building block, depicted in Figure 2, is in the form of a wait-free code segment that assigns to a private variable *dir* one of three values, *stop*, *right*, or *down*. If each of n processes executes this code segment at most once, then at most 1 process receives a value of *stop*, at most $n-1$ processes receive a value of *right*, and at most $n-1$ processes receive a value of *down*. We say that a process that receives a value of *down* "goes down", a process that receives a value of *right* "goes right", and a process that receives a value of *stop* "stops". Figure 2 shows n processes accessing a building block, and the maximum number of processes that receive each value.

The code fragment shown in Figure 2 shows how the building block described above can be implemented using atomic read and write operations. The technique employed is essentially that of the "fast path" used in Lamport's fast mutual exclusion algorithm [7]. A process that stops corresponds to a process successfully "taking the fast path" in Lamport's algorithm. The value assigned to *dir* by a process p that fails to "take the fast path" is determined by the branch p takes: if p detects that Y holds, then p goes right, and if p detects that $X \neq p$ holds, then p goes down.

To see that the code fragment shown in Figure 2 satisfies the requirements of our building block, note that it is impossible for all n processes to go right — a process can go right only if another process previously assigned $Y := true$. Second, the last process p to assign $X := p$ cannot go down, because if it tests X, then it detects that $X = p$, and therefore stops. Thus, it is impossible for all n processes to go down. Finally, because Lamport's algorithm prevents more than one processes from "taking the fast path", it is impossible

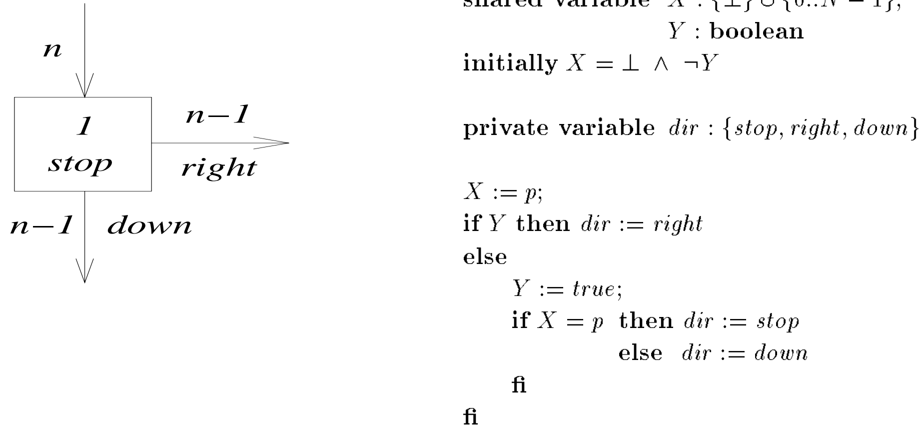

Figure 2: The one-time building block and the code fragment that implements it.

for more than one process to stop. Thus, the code fragment shown in Figure 2 satisfies the requirements of the building block.

In the next section, we show how these building blocks can be combined to solve the renaming problem. The basic approach is to use such building blocks to "split" processes into successively smaller groups. Because at most one process stops at any particular building block, a process that stops can be immediately given a unique name associated with that building block. Furthermore, when the size of a group has been decreased often enough that at most one process remains in the group, that process (if it exists) can be given a name immediately.

3.2 Using the One-Time Building Block to Solve Renaming

In this section, we use $k(k-1)/2$ one-time building blocks arranged in a "grid" to solve one-time renaming; this approach is depicted in Figure 3 for $k = 5$. In order to acquire a name, a process p accesses the building block at the top left corner of the grid. If p receives a return value of *stop*, then p acquires the name associated with that building block. Otherwise, p moves either right or down in the grid, according to the return value received. This is repeated until the p receives a return value of *stop* at some building block, or p has accessed $k-1$ building blocks. The name returned is calculated based on the p's final position in the grid. In Figure 3, each grid position is labeled with the name associated with that position. Because no process takes more than $k-1$ steps, only the upper left triangle of the grid is used, as shown in Figure 3.

The algorithm is presented more formally in Figure 4. A process acquires a name by calling the *Getname* procedure. The *Grid* procedure and associated variable declarations provide the grid of building blocks, each of which is implemented using the code segment shown in Figure 2. The grid position at the ith row and jth column, where $0 \leq i < k$ and $0 \leq j < k$, is referred to as (i, j); the top left corner of the grid is $(0, 0)$. We say that a grid position (i, j) is *downstream* from a grid position (i', j') iff $i \geq i'$ and $j \geq j'$. Grid position (i, j) is $i + j$ *steps away from* grid position $(0, 0)$. We now sketch the proof of a lemma that is central to the correctness proof for this algorithm.

Lemma 1: At most $k - i - j$ processes reach grid positions downstream from grid position (i, j).

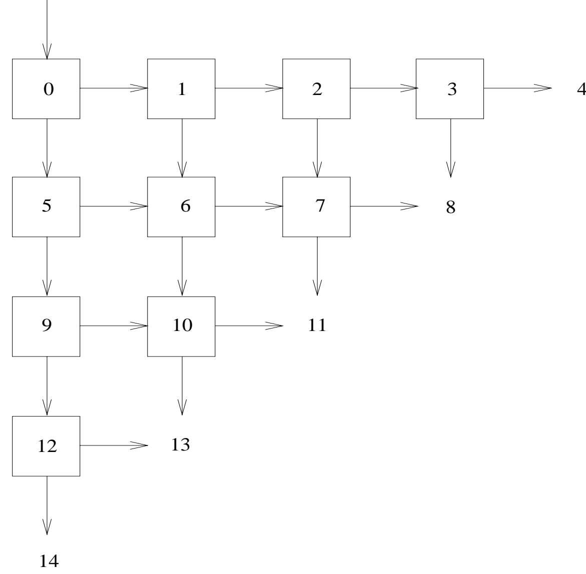

Figure 3: $k(k+1)/2$ building blocks in a grid, depicted for $k = 5$.

```
shared variable   X : array[0..k−2, 0..k−2] of {⊥} ∪ {0..N−1};
                  Y : array[0..k−2, 0..k−2] of boolean
initially (∀i, j : 0 ≤ i < k−1 ∧ 0 ≤ j < k−1 :: X[i, j] = ⊥ ∧ ¬Y[i, j])

procedure Grid(row, col : 0..k−2; p : 0..N−1) returns {stop, right, down}
private variable dir : {stop, right, down};
    X[row, col] := p;
    if Y[row, col] then dir := right
    else
        Y[row, col] := true;
        if X[row, col] = p then dir := stop
                           else dir := down
        fi
    fi;
    return dir

procedure Getname() returns 0..k(k+1)/2
private variable  move : {stop, right, down};
                  i, j : 0..k−1
    i, j, move := 0, 0, down;
    while i + j < k−1 ∧ move ≠ stop do       /* Move down or across grid until stopping or reaching edge */
        move := Grid(i, j, p);                /* Access building block at this position to get direction or stop */
        if move = down then                   /* Move according to direction acquired */
            i := i + 1
        elseif move = right then
            j := j + 1
        fi
    od;
    return i(k − (i−1)/2) + j                 /* Calculate name based on position in grid */
```

Figure 4: One-time renaming using a grid of building blocks.

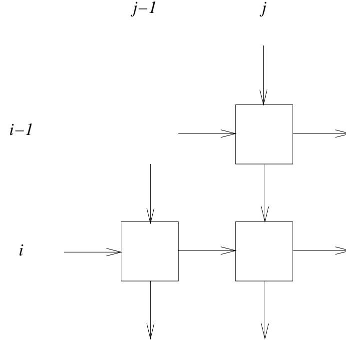

Figure 5: Inductive step for proof of Lemma 1.

Proof Sketch: By induction on $i+j$. The lemma trivially holds for $i+j=0$, because it is assumed that at most k processes access the renaming algorithm. For the inductive step, consider Figure 5.

We inductively assume that Lemma 1 holds for grid positions that are fewer than $i+j$ steps away from $(0,0)$. In particular, we assume that at most $k-i-j+1$ processes reach grid positions downstream from $(i,j-1)$ and similarly for $(i-1,j)$. The lemma can be falsified by a process moving right from a building block in column $j-1$ at or below row i, or by a process moving down from a building block in row $i-1$ at or to the right of column j. Below we argue that the former case is impossible; the proof for the latter case is similar.

Suppose, towards a contradiction, that $k-i-j$ processes have already reached positions downstream of (i,j), and that another process p moves right from a building block in column $j-1$ at or below row i. After p moves right, $k-i-j+1$ processes have reached positions downstream of (i,j), so the lemma does not hold. We show that this is impossible. By the inductive assumption, there are only $k-i-j+1$ processes downstream of $(i,j-1)$. In the scenario described above, all of these $k-i-j+1$ processes are downstream from (i,j) after p moves right. Thus, after process p moves right, all processes that have accessed building blocks in column $j-1$ at or below row i have moved right. This implies that there is some building block in column $j-1$ at or below row i such that all processes that accessed that building block moved right. Recall that if n processes access a building block, then at most $n-1$ of them move right. Thus, the scenario described above is impossible. □

A process acquires a name either by stopping at a building block in the grid, or by taking $k-1$ steps in the grid. At most one process stops at each building block, and by Lemma 1, at most one process reaches each of the grid positions that is $k-1$ steps from $(0,0)$. Thus, all processes acquire distinct names. Also, because each process takes at most $k-1$ steps in the grid, and because each building block access has time complexity $O(1)$, this algorithm has time complexity $O(k)$. A full assertional correctness proof is given for this algorithm in the full version of the paper. The algorithm shown in Figure 4 yields the following result.

Theorem 1: Using reads and writes, one-time $(k(k+1)/2)$-renaming can be implemented with time complexity $O(k)$. □

Using the algorithm described above, k processes can reduce the size of their name space from N to $k(k+2)/2$ with time complexity $O(k)$. Using the algorithm presented in [5], k processes can reduce the size of their name space from N to $2k-1$ with time complexity $O(Nk^2)$. Combining the two algorithms, k processes can reduce the size of their name space from N to $2k-1$ with time complexity $O(k)+O((k(k+1)/2)k^2) = O(k^4)$. Thus, we have the following result. By the results of [6], this algorithm is optimal with respect to the size of the name space.

Theorem 2: Using reads and writes, one-time $(2k-1)$-renaming can be implemented with time complexity $O(k^4)$. □

4 Long-Lived Renaming using Atomic Reads and Writes

In this section, we present a long-lived renaming algorithm that uses only atomic read and write operations. Processes using a long-lived renaming algorithm are organized as shown in Figure 1. Recall from Section 2 that a solution to the long-lived renaming problem consists of *Getname* and *Putname* procedures, along with associated shared variables. The *Getname* and *Putname* procedures allow processes to repeatedly acquire and release names.

The long-lived renaming algorithm presented in this section is based on the grid algorithm presented in the previous section. To enable processes to release names as well as acquire names, we modify the one-time building block. The modification allows a process to "reset" a building block that it has accessed. This algorithm yields a name space of size $k(k+1)/2$ and has time complexity $O(Nk)$.

Before presenting this long-lived renaming algorithm, we first describe the modified building block that is used in the algorithm.

4.1 The Long-Lived Building Block

The long-lived building block is similar to the one-time building block used in the previous section. However, after a process has accessed the long-lived building block, and has acquired a return value of *stop*, *down*, or *right*, the process can "reset" the building block. As seen in Figure 6, processes are required to access and reset the building block alternately. We have the same requirements of the values returned as before, but now only for processes outside their remainder sections. That is, if it is guaranteed that at most n processes are outside their remainder sections at any time, then at most one process is in its working section with $dir = stop$, at most $n-1$ processes are in their working sections with $dir = right$, and at most $n-1$ processes are in their working sections with $dir = down$. By resetting a building block, a process records that it is no longer in the working section.

In the long-lived building block, each process has its own Y-bit, instead of all processes writing the same Y-variable. Thus, instead of reading a single Y-variable, each process reads all N Y-bits. The time complexity for accessing the long-lived building block is therefore $O(N)$. In order to "reset" the building block, a process p resets its bit $Y[p]$.

We now prove that our long-lived building block has the properties described above. In order to simplify the proof, we assume that each numbered code fragment in Figure 6 is executed atomically. Note that each such fragment accesses at most one shared variable. The invariants used to prove that the required properties hold are listed below. In accordance with the definition of the building block, we first assume that at most n processes access the building block concurrently.

invariant $|\{p :: p@\{1..5\}\}| \leq n$ (I1)

The next four invariants are used to prove that if at most n processes access the building block concurrently, then at most one process is in its working section with $dir = stop$, at most $n-1$ processes are in their working sections with $dir = right$, and at most $n-1$ processes are in their working sections with $dir = down$. The proofs of these invariants are straightforward, and are therefore omitted. The proof of (I3) uses (I2), the proof of (I6) uses (I1), and the proof of (I7) uses (I4).

invariant $p@4 \Rightarrow p.dir \neq right$ (I2)
invariant $(p@\{4,5\} \ \wedge \ p.dir \neq right) = Y[p]$ (I3)
invariant $p@\{2,3\} \Rightarrow p.dir \neq stop$ (I4)
invariant $(p@5 \ \wedge \ p.dir = down) \Rightarrow X \neq p$ (I5)
invariant $(|\{p :: p@\{2..5\}\}| = n) \Rightarrow (\exists p : p@\{2..5\} :: X = p)$ (I6)
invariant $(p \neq q \ \wedge \ p@5 \ \wedge \ p.dir = stop) \Rightarrow (Y[p] \ \wedge \ (q@\{0..1\} \ \vee \ (q@2 \ \wedge \ q.i \leq p) \ \vee$
$(q@3 \ \wedge \ q.dir = right) \ \vee \ (q@\{2,3,4\} \ \wedge \ X \neq q) \ \vee \ (q@5 \ \wedge \ q.dir \neq stop)))$ (I7)

```
shared variable   X : {⊥} ∪ {0..N − 1};
                  Y : array[0..N − 1] of boolean
initially X = ⊥ ∧ (∀p : 0 ≤ p < N :: p@0 ∧ ¬Y[p])

private variable  dir : {stop, right, down};            /* Direction for process to leave box */
                  h : 0..N − 1

    while true do
0:      Remainder Section;
1:      X, h, dir := p, 0, down;         /* Access the long-lived building block (statements 1 through 4) */
        while h < N ∧ dir ≠ right do
2:          if Y[h] then dir := right
                    else h := h + 1
            fi
        od;
3:      if dir ≠ right then
            Y[p] := true;
4:          if X ≠ p then dir := down
                      else dir := stop
            fi
        fi;
        Working Section;
5:      Y[p] := false                                  /* Reset the long-lived building block */
    od
```

Figure 6: Implementation of the long-lived building block.

The following three invariants show that the building block is correct.

invariant $|\{p :: p@5 \ \wedge \ p.dir = stop\}| \leq 1$ (I8)

Proof: (I8) follows from (I7) □

invariant $|\{p :: p@5 \ \wedge \ p.dir = down\}| < n$ (I9)

Proof: Initially, the left-hand side is zero, so (I9) holds. By the assumption that (I9) holds before a statement execution, it follows that (I9) can only be falsified by establishing $q@5 \ \wedge \ q.dir = down$ when $|\{p :: p@5 \ \wedge \ p.dir = down\}| = n - 1$ holds. The only statement that can establish $q@5 \ \wedge \ q.dir = down$ is $q.4$. By (I5) and (I6), $X = q$ holds in this case. Thus, $q.4$ establishes $q@5 \ \wedge \ q.dir = stop$, and therefore does not falsify (I9). □

invariant $|\{p :: p@\{3..5\} \ \wedge \ p.dir = right\}| < n$ (I10)

Proof: Initially, the left-hand side is zero, so (I10) holds. By the assumption that (I10) holds before a statement execution, it follows that (I10) can only be falsified by establishing $q@\{3..5\} \wedge q.dir = right$ when $|\{p :: p@\{3..5\} \ \wedge \ p.dir = right\}| = n - 1$ holds. The only statement that can establish $q@\{3..5\} \ \wedge \ q.dir = right$ is $q.2$. By (I1) and (I3), $(\forall i : 0 \leq i < N :: \neg Y[q.i])$ holds in this case, so $q.2$ does not establish $q@\{3..5\} \ \wedge \ q.dir = right$, and therefore does not falsify (I10). □

Having described the modified building block and proved it correct, we now proceed to describe the long-lived renaming algorithm.

4.2 Using the Long-Lived Building Block to Solve Long-Lived Renaming

Our first long-lived renaming algorithm is similar to the one-time renaming algorithm presented in the previous section. We use a grid of long-lived building blocks, analogous to the grid shown in Figure 3 for the one-time renaming algorithm. This implementation of this grid, shown in Figure 7, provides procedures *LL_Grid* and *LL_Reset*.

In order to access the building block at grid position (i, j), process p calls *LL_Grid*(i, j, p); similarly for *LL_Reset*(i, j, p). Note that the *LL_Grid* procedure is implemented using statements 1 through 4 in Figure 6, and that the *LL_Reset* procedure is implemented using statement 5 in Figure 6.

As in the one-time algorithm presented in the previous section, a process acquires a name by starting at the top left corner of the grid, and moving through the grid according to the return value received from each building block. However, in this algorithm, after a process has accessed a building block and received a return value other than *stop*, the process "resets" that building block before proceeding to the next building block in the grid. This allows the grid to be "reused" by processes that release their names by calling *Putname*, and then subsequently call *Getname* again. The *Getname* and *Putname* procedures are shown in Figure 7.

To prove this algorithm correct, we prove a lemma similar to Lemma 1. Again referring to Figure 5, we inductively assume that at most $k - i - j + 1$ processes concurrently occupy positions downstream of $(i-1, j)$, and similarly for $(i, j-1)$. Again, we assume towards a contradiction that $k-i-j$ processes already occupy positions downstream of (i, j). Suppose that process p is about to test a Y-bit in a building block $(h, j-1)$ for some $h \geq i$. By the inductive assumption and the assumption that $k - i - j$ processes already occupy positions downstream of (i, j), it can be shown that no Y-bit in the building block at $(h, j-1)$ is set. Therefore, process p does not move right. The proof that a process does not falsify the lemma by moving down from a building block in row $i-1$ and at or to the right of column j is somewhat more difficult. The key observation is that the last process p to write $X[i-1, h]$ for any $h \geq j$ cannot go down from row $i-1$. It can be shown that process p, and all processes that do go down from row $i-1$ do not have Y-bits set in any building block in column $j-1$ at or below row i. Using this, it can be shown that the last process to occupy a position downstream from (i, j) does so by moving right from $(h, j-1)$ for some $h \geq i$. As is argued in the first case, this is impossible. A complete assertional proof is given in the full paper. This algorithm yields the following result.

Theorem 3: Using reads and writes, long-lived $(k(k+1)/2)$-renaming can be implemented with time complexity $O(Nk)$. □

5 Long-Lived Renaming Algorithms using Read-Modify-Writes

In this section, we present three long-lived renaming algorithms. By using read-modify-write operations, these algorithms achieve significantly better performance than the algorithms presented in the previous section. Furthermore, these algorithms all yield a name space of size k, which is clearly optimal (the lower bound results of [6] do not apply to algorithms that employ read-modify-write operations).

The first algorithm has time complexity $O(k/b)$. This is achieved using *set_first_zero* and *clear_bit*. As discussed in Section 2, these operations can be implemented, for example, using operations available on the BBN TC2000 [4]. The second algorithm in this section has time complexity $O(\log k)$, which is a significant improvement over the first algorithm. To achieve this improvement, this algorithm uses a primitive operation called *bounded_decrement*. We do not know of any systems that provide *bounded_decrement* as a primitive operation. However, as noted in Section 6, this operation can be efficiently implemented in a lock-free manner using the commonly-available *fetch_and_decrement* operation. Finally, we describe how the techniques from these two algorithms can be combined to obtain an algorithm whose time complexity is better than that of either algorithm.

```
shared variable   X : array[0..k−2, 0..k−2] of {⊥} ∪ {0..N−1};
                  Y : array[0..k−2, 0..k−2] of array[0..N−1] of boolean
initially (∀i, j, h :: 0 ≤ i < k−1 ∧ 0 ≤ j < k−1 ∧ 0 ≤ h < N :: X[i, j] = ⊥ ∧ ¬Y[i, j][h])


procedure LL_Grid(row, col : 0..k−2; p : 0..N−1)
                  returns {stop, right, down}
private variable  dir : {stop, right, down};
                  h : 0..N
   X[row, col], h, dir := p, 0, down;
   while h < N ∧ dir ≠ right do
      if Y[row, col][h] then dir := right
                        else h := h + 1
      fi
   od;
   if dir ≠ right then
      Y[row, col][p] := true;
      if X[row, col] ≠ p then dir := down
                          else dir := stop
      fi
   fi;
   return dir


procedure LL_Reset(row, col : 0..k−2; p : 0..N−1)
Y[row, col][p] := false;
return


procedure Getname() returns 0..k(k+1)/2
private variable  move : {stop, right, down};
                  i, j : 0..k−1
   i, j, move := 0, 0, down;                  /* Start at top left building block in grid */
   while i + j < k−1 ∧ move ≠ stop do          /* Move down or across grid until stopping or reaching edge */
      move := LL_Grid(i, j, p);               /* Access building block at this position to get direction or stop */
      if move ≠ stop then
         LL_Reset(i, j, p);                   /* Reset block if we didn't stop at it */
         if move = down then i := i + 1       /* Move according to move */
                        else j := j + 1
         fi
      fi
   od;
   return i(k − (i−1)/2) + j                  /* Calculate name based on position in grid */


procedure Putname() returns 0..k(k+1)/2
   if i + j < k−1 then                        /* If we stopped on a building block ... */
      LL_Reset(i, j, p)                       /* ... then reset that building block */
   fi;
   return
```

Figure 7: Long-lived renaming with $O(k^2)$ name space and $O(Nk)$ time complexity. Variables i and j are assumed to retain their values between calls to *Getname* and *Putname*.

```
shared variable X : array[0..⌊k/b⌋] of array[0..b−1] of boolean      /* b-bit "segments" of the name space */
initially (∀h, g : 0 ≤ h < ⌊k/b⌋ ∧ 0 ≤ g < b :: ¬X[h][g])

private variable  h : 0..⌊k/b⌋ + 1;
                  g : 0..b

procedure Getname() returns 0..k − 1
    h, g := 0, b;
    while g = b do
        g := set_first_zero(X[h], b);                        /* Set first zero bit of X[h] if one exists */
        h := h + 1
    od;
    return (b * (h − 1)) + g                                  /* Calculate name */

procedure Putname()
    clear_bit(X[h − 1], g);                 /* Return name by clearing the bit that was set */
    return
```

Figure 8: k-renaming using *set_first_zero* and *clear_bit*. Variables g and h are assumed to retain their values between calls to *Getname* and *Putname*.

5.1 Long-Lived Renaming using *set_first_zero* and *clear_bit*

In this section, we present a long-lived k-renaming algorithm that employs the *set_first_zero* and *clear_bit* operations described in Section 2. The algorithm is shown in Figure 8. In order to acquire a name, a process tests each name in order. Using the *set_first_zero* operation, up to b names can be tested in one atomic shared variable access. If $k \leq b$, this results in a long-lived renaming algorithm that acquires a name with just one shared variable access. If $k > b$, then "segments" of size b of the name space are tested in each access. To release a name, a process simply clears the bit associated with the name it acquired — that is, the bit that was set by that process when the name was acquired.

Because each process tests the available names in segments, and because processes may release and acquire names concurrently, it may seem possible for a process to reach the last segment when none of the names in that segment are available. The following lemma shows that this is not possible when $b = 1$. The proof for $b \geq 1$ is a straightforward generalization.

Lemma 2: At most n processes concurrently hold or test names in $\{k-n, ..., k-1\}$, for $1 \leq n \leq k$.

Proof Sketch: For $n = k$, the lemma holds because at most k processes concurrently access the renaming algorithm. We inductively assume that at most $n+1$ processes concurrently hold or test names in $\{k-n-1, ..., k-1\}$. Suppose, towards a contradiction, that n processes are holding or testing names in $\{k-n, ..., k-1\}$, and that another process p tests name $k-n-1$. If p were to fail to acquire name $k-n-1$, then p would proceed to test name $k-n$. Then there would be $n+1$ processes testing or holding names in $\{k-n, ..., k-1\}$, thereby falsifying the lemma. However, when process p tests name $k-n-1$, it follows by the inductive assumption that no process is holding name $k-n-1$. Thus, process p acquires name $k-n$. □

By Lemma 2, it follows that if process p tests name $k-1$, then that name is available. A complete assertional proof is given in the full paper. The algorithm yields the following result.

Theorem 4: Using *set_first_zero* and *clear_bit* on b-bit variables, long-lived k-renaming can be implemented with time complexity $O(k/b)$. □

```
shared variable X : 0..⌈k/2⌉                                  /* Counter of names available on right */
initially X = ⌈k/2⌉

private variable  side : {left, right}                         /* Which instance to use inductively */

procedure Getname() returns 0..k − 1
                    /* Ensure at most ⌈k/2⌉ access right instance and at most ⌊k/2⌋ access left */
   if bounded_decrement(X) > 0 then
      side := right;                                            /* Get name from right instance */
      return Getname_right()
   else
      side := left;                                             /* Get name from left instance */
      return ⌈k/2⌉ + Getname_left()
   fi

procedure Putname()
   if side = right then
      Putname_right()                                           /* Return name to right instance */
   else
      Putname_left()                                            /* Return name to left instance */
   fi;
   if side = right then atomic_add(X, 1) fi;       /* Increment counter again if it was decremented */
   return
```

Figure 9: k-renaming using *bounded_decrement*. *Getname_left* and *Putname_left* are inductively assumed to implement long-lived $\lfloor k/2 \rfloor$-renaming. Similarly, *Getname_right* and *Putname_right* are inductively assumed to implement long-lived $\lceil k/2 \rceil$-renaming.

5.2 Long-Lived Renaming using *bounded_decrement* and *atomic_add*

In this section, we present a long-lived k-renaming algorithm that employs the *bounded_decrement* and *atomic_add* operations described in Section 2. In this algorithm, shown in Figure 9, processes are separated into two groups *left* and *right* using the *bounded_decrement* operation. The *right* group contains at most $\lceil k/2 \rceil$ processes and the *left* group contains at most $\lfloor k/2 \rfloor$ processes. This is achieved by initializing a variable X to $\lceil k/2 \rceil$, and having each process perform a *bounded_decrement* operation on X. Processes that receive positive return values join the *right* group, and processes that receive zero join the *left* group. To leave the *right* group, a process increments X. To leave the left group, no shared variables are updated.

Because processes must be able to repeatedly join and leave the groups, the normal *fetch_and_decrement* operation is not suitable for this "splitting" mechanism. If X is decremented below zero, then it is possible for too many processes to be in the *left* group at once. To see this, suppose that all k processes decrement X. Thus, $\lceil k/2 \rceil$ processes receive positive return values, and therefore join the *right* group, and $\lfloor k/2 \rfloor$ processes receive non-positive return values, and therefore join the *left* group. Now, $X = -\lfloor k/2 \rfloor$. If a process leaves the *right* group by incrementing X, and then decrements X as the result of another call to *Getname*(), then that process receives a non-positive return value, and thus joins the *left* group. Repeating this for each process in the *right* group, it is possible for all processes to be in the *left* group simultaneously. The *bounded_decrement* operation prevents this from happening by ensuring that X does not become negative.

The algorithm employs two instances *left* and *right* of long-lived renaming, which are inductively assumed to be correct. For notational convenience, we assume that the *left* instance is accessed by calling the *Getname_left* and *Putname_left* procedures; similarly for the *right* instance. A process decides which of these two instances to access based on which of the two groups it joins. The algorithm that results from

"unfolding" this inductively-defined algorithm forms a tree. To acquire a name, a process goes down a path in this tree from the root to a leaf. As the processes progress down the tree, the number of processes that can simultaneously go down the same path is halved at each level. When this number becomes one, a name can be assigned, so no more inductive levels are necessary. Thus, the *Getname*() procedure has time complexity $\lceil \log_2 k \rceil$. To release a name, a process simply retraces the path it took through the tree in reverse order, incrementing X at any node at which it received a positive return value.

Note that with b-bit variables, if $b < \log_2 \lceil k/2 \rceil$, then X cannot be initialized to $\lceil k/2 \rceil$, so this algorithm cannot be implemented. However, in any practical setting, this will not be the case. A complete assertional correctness proof of the following result is given in the full paper.

Theorem 5: Using *bounded_decrement* and *atomic_add* on b-bit variables, long-lived k-renaming can be implemented with time complexity $O(\log k)$, if $b \geq \log_2 \lceil k/2 \rceil$. $\square$

Finally, we note that if the *set_first_zero* and *clear_bit* operations are available, then it is unnecessary to "unfold" the second algorithm described above so far that at most one process can reach a leaf of the tree. If the tree is deep enough that at most b processes can reach a leaf, then by Theorem 4, a name can be assigned with one more shared access. This amounts to "chopping off" the bottom $\lfloor \log_2 b \rfloor$ levels of the tree. The time complexity of this algorithm is $O(\log k - \log b) = O(\log(k/b))$. Thus, using all the operations that are employed by the first two algorithms, it is possible to achieve better time complexity than either of them. This approach yields the following result.

Theorem 6: Using *set_first_zero*, *clear_bit*, *bounded_decrement*, and *atomic_add* on b-bit variables, long-lived k-renaming can be implemented with time complexity $O(\log(k/b))$. $\square$

6 Concluding Remarks

We have presented two one-time renaming algorithms that employ only atomic read and write operations. One of these algorithms yields an optimal-size name space. These algorithms improve on previous read/write renaming algorithms in that their time complexity is independent of the size of the original name space.

In addition, we have defined a new version of the renaming problem called long-lived renaming, in which processes can release names as well as acquire names. We have provided several solutions to this problem, including one that employs only read and write operations. In comparing these algorithms, there is a trade-off between time complexity, the size of the name space achieved, and the availability of the primitives used.

All of our algorithms have the desirable property that time complexity is proportional to the level of contention. This is an important practical advantage because contention should be low in most well-designed applications [7].

There are numerous questions left open by our research. For example, it follows from our work that one-time $(2k-1)$-renaming can be solved using reads and writes with time complexity $O(k^4)$. We would like to improve on this time complexity while still providing an optimal-size name space. Our fastest read/write algorithm has time complexity $O(k)$ and yields a name space of size $k(k+1)/2$. In this algorithm, there are pairs of names that cannot be held concurrently, so in fact, these names need not be distinct. It can be shown that roughly $k^2/4$ names suffice for large k. However, this modification still does not achieve an optimal-size name space.

The long-lived renaming algorithm presented in Section 4 yields a name space of size $k(k+1)/2$ with time complexity $O(Nk)$. We would like to improve on this result by obtaining an optimal name space of size $2k-1$ using only read and write operations, and by making the time complexity independent of N.

Our most efficient long-lived renaming algorithm uses a *bounded_decrement* operation. Although this operation is similar to the standard *fetch_and_decrement* operation, we have been unable to design an efficient wait-free implementation of the former using the latter. We have, however, designed an efficient lock-free

implementation in which a process can only be delayed by a very unlikely sequence of events. We believe this implementation will perform well in practice.

Acknowledgement: We would like to thank Gadi Taubenfeld and Rajeev Alur for helpful discussions about this work.

References

[1] J. Anderson and M. Moir, "Using k-Exclusion to Implement Resilient, Scalable Shared Objects", to appear in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*.

[2] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk, "Achievable Cases in an Asynchronous Environment", *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, October 1987, pp. 337-346.

[3] A. Bar-Noy and D. Dolev, "Shared Memory versus Message-Passing in an Asynchronous Distributed Environment", *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, August 1989, pp. 307-318.

[4] BBN Advanced Computers, *Inside the TC2000 Computer*, February, 1990.

[5] E. Borowsky and E. Gafni, "Immediate Atomic Snapshots and Fast Renaming", *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, August 1993, pp. 41-50.

[6] M. Herlihy and N. Shavit, "The Asynchronous Computability Theorem for t-Resilient Tasks", *Proceedings of the 25th ACM Symposium on Theory of Computing*, 1993, pp. 111-120.

[7] L. Lamport, "A Fast Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems*, Vol. 5, No. 1, February, 1987, pp. 1-11.