LONG LIVED AND ADAPTIVE SHARED MEMORY IMPLEMENTATIONS

By

Gideon Stupp

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
TEL-AVIV UNIVERSITY
TEL-AVIV, ISRAEL
JANUARY 2001

TEL-AVIV UNIVERSITY
DEPARTMENT OF
MATHEMATICS

The undersigned hereby certify that they have read and recommend to the Faculty of Exact Sciences for acceptance a thesis entitled "**Long Lived and Adaptive Shared Memory Implementations**" by **Gideon Stupp** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Dated: January 2001

External Examiner: _____

Research Supervisor: _____
Yehuda Afek

Examing Committee: _____

_____

TEL-AVIV UNIVERSITY

Date: **January 2001**

Author: **Gideon Stupp**

Title: **Long Lived and Adaptive Shared Memory Implementations**

Department: **Mathematics**

Degree: **Ph.D.** Convocation: **May** Year: **2001**

Signature of Author

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

As distributed systems come of age they change from simple and small systems to big clusters of thousands of processes. Typically, however, only a small number of the processes participate in a task at the same time. This motivates the need for algorithms whose execution time does not depend on the size of the cluster. The main goal of this work is to present such algorithms.

A distributed system is a set of processes that coordinate via some communication media. In this work we assume that the communication media is shared memory. Furthermore, we assume only atomic read and write operations on individual memory registers. The processes collaborate in the implementation of a concurrent object. All the objects we investigate are long lived. I.e., a process can repeatedly perform some operation.

A classical example for a long lived concurrent object is the mutual exclusion problem [Dij65]. In it processes repeatedly try to enter a critical section, perform some operation in exclusion and leave the critical section. The algorithm must ensure that at any point in time at most one process is in the critical section. Furthermore, if any process wants to enter the critical section then some process must succeed.

Many solutions to the mutual exclusion problem were proposed in the past ([Dij65, Knu66, Lam74, Pet81, Lam86c, Lam86a, Lam86b]). In all of these solutions a process takes at least $F(N)$ steps (typically linear) when trying to enter the critical section (N is the number of processes in the system) even if it executes by itself. In 1987 Lamport [Lam87] suggested a mutual exclusion algorithm in which a process attempting to enter the critical section all by itself does so in 7 steps. This naturally raised the question as to the existence of an adaptive algorithm where the number of steps taken by a process depends on the *contention*, the number of processes that try to enter the critical section at the same time (the notion of adaptiveness is defined in Section 1.1). Many papers on adaptive implementations of mutual exclusion and other shared objects were published since then [Lam87, Sty92, CS93, BP89, MT93, ADT95, MA95, BGHM95, MG96, Moi98, AF98, AM98].

While an adaptive mutual exclusion algorithm is presented in this thesis, most of the thesis deals with adaptive implementations of *wait-free* long lived objects. In the wait-free model, up to $N-1$ processes out of N may fail; Still the remaining working processes must finish the task. In other words, processes work in their own pace and cannot wait for other processes. We present adaptive solutions for various known problems in this model, in particular for renaming, collect, snapshot and immediate snapshot. The adaptive algorithm for the snapshot object is of particular interest because it can be used with the full information algorithm. Any wait-free algorithm in the atomic read write shared memory model that uses only single writer multi reader registers can be represented as a sequence of snapshots. If the number of snapshots necessary for a specific algorithm is bounded by the size of the contention then it is possible to exchange the snapshots with the adaptive implementation and thus transform the full information algorithm into an adaptive algorithm.

In the following paragraphs we informally define the concurrent objects that are implemented in this thesis and describe the way they relate to each other (see also Figure 1.3).

In an adaptive long lived *M-renaming*, every process starts with a unique id from the set $\{1 \dots N\}$. During the execution processes may repeatedly acquire a new name in the range $\{1 \dots M \ll N\}$, hold it for an arbitrary amount of time and then release it. The algorithm must ensure that the new names are unique among the set of currently acquired names and that the step complexity of an operation depends on the contention (as defined below). Furthermore, M must be a function of the contention too (see Definition 2.5). In [MA95] Moir and Anderson introduced a novel technique for constructing renaming algorithms consisting of two elements: a *splitter* building block and a grid of splitters. A splitter is a variant of mutual-exclusion. If one process accesses the splitter alone it captures that splitter and the name (resource) associated with it. However, unlike standard mutual exclusion, when several processes access the splitter concurrently they may all fail, leaving the splitter un-captured. Instead, the splitter wait-freely partitions the processes that fail into two groups, `right` and `down`. Moreover, the splitter ensures that not all the contending processes go to the right and not all the contending processes go down. Therefore, when a set of processes contend on a set of splitters placed on a grid all starting their competition at the top left-most splitter they split and divide into smaller groups as they progress through the grid. After traversing enough splitters a process is guaranteed to access a splitter alone and to capture the associated resource, i.e., to acquire a name.

Here we define and implement a variant of the Moir-Anderson splitter that does not have all the properties that their splitter had but on the other hand, has an adaptive and long-lived implementation. Furthermore, we use this splitter as a building block not only in a grid as [MA95], but also in other constructions (for example a row of splitters or a tree of splitters). In this way we implement diverse applications such as mutual exclusion

and optimal name-space renaming. However, the renaming algorithms do not achieve the strongest possible adaptiveness (they are only adaptive to interval contention and not to point contention, see Section 1.1). Therefore, we separately provide a point contention long lived renaming based on the sieve object of Attiya and Fouren [AAF$^+$99b]. This algorithm is then used to implement the collect and snapshot objects.

Intuitively, the operational semantics of a *collect* object are non-sequentially specified as follows: There are N single-writer-multi-reader registers $C_1, \ldots, C_N$, one for each process. To write in the collect object process P_i writes its value in C_i. To perform a collect a concurrent process simply reads the N registers one at a time and returns the vector of values read. We actually begin by defining and implementing a weaker form of collect, called *gather*. The gather operation has the same specification as a collect but $C_1 \ldots C_N$ are assumed to be only regular registers [Lam86d] and not atomic. The main difference between a gather operation and a collect operation is that even if a gather operation g returns value v for process p, it is still possible that a later gather operation g' returns a value v' for p that was written before v. This can happen if the write operation of v was concurrent to g and g'. We use the gather to implement an *active set* object with which processes can adaptively retrieve the set of currently active processes. The operational semantics of this object are as follows: There are N single-writer-multi-reader regular boolean registers, $F_1, \ldots, F_N$, one for each process. Three concurrent operations are supported by the active set: joinSet(), leaveSet() and getSet(). In joinSet()$_i$ process p_i writes **true** to F_i, in leaveSet()$_i$ it writes **false** to F_i. In a getSet() operation a process reads all the N variables and returns the set of process ids whose flags were read **true**. This turns out to be a fundamental tool for constructing adaptive algorithms.

Unlike the gather and collect in a *snapshot* object the snapshot-scan operation returns an instantaneous view of the values stored, i.e., the snapshot scans and the updates can be linearized. Finally, we use the above constructions to implement an adaptive immediate snapshot object. The *immediate snapshot* supports only one operation, im-upscan(), with which a process both writes its current value and reads the values written by the other processes. The immediate snapshot object can be described as a synchronous execution of rounds, where in each round a process can either participate or not. A participating process writes its value and then snapshots the values of all the other processes (participating or not). The operational semantics require that a process participating in round i returns the last value written by every process that participated in rounds $1, \ldots, i$.

Since our implementations are adaptive to the (point) contention, they use multi-writer-multi-reader registers. Moreover, because each operation returns a vector of size N there can be no adaptive snapshot or immediate snapshot that uses registers of size smaller than N (consider the case that one process runs alone and has to return a vector of N values in $O(1)$ primitive operations). Therefore, our implementations use registers that can hold N values (as is also the case with the atomic snapshot algorithm of [AAD$^+$93]).

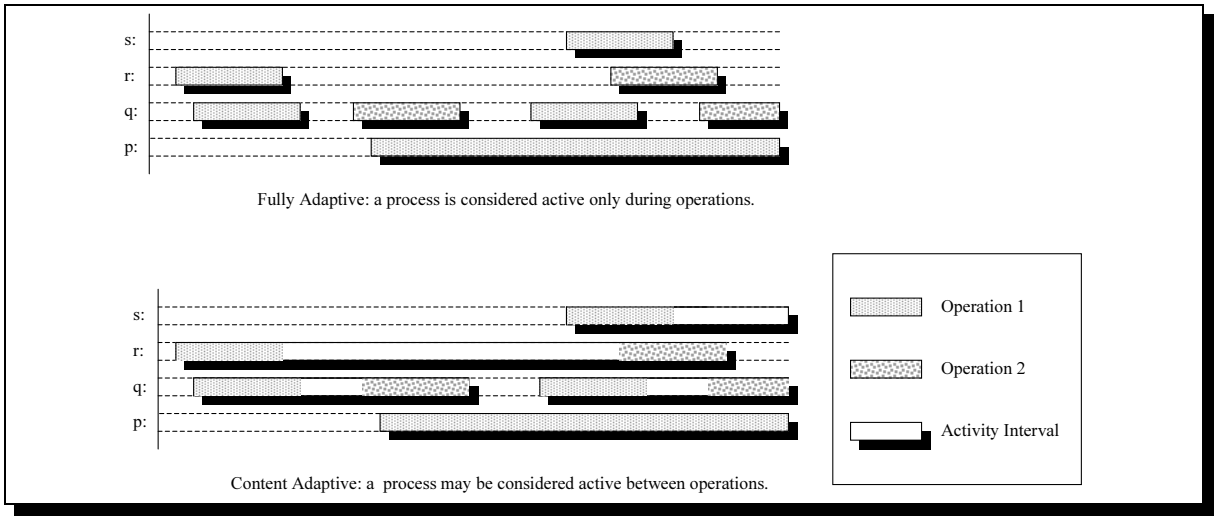


Figure 1.1: Content Adaptive vs. Fully Adaptive

1.1 Adaptivity

As described before, in an adaptive algorithm the step complexity of an operation depends only on the number of processes that actually take steps concurrently with that operation. In other words, the complexity of an operation is a function of the actual contention it encounters and not of the total number of processes or any bound on the number of active processes[1].

The strongest form of adaptiveness in the read/write shared memory model has been defined and achieved in the long-lived renaming algorithms presented in [AF99a, AAF+99b, AAF+99a]. In these specially tailored algorithms the complexity of an operation is a function of the *point contention* of the operation, defined as the maximum number of processes executing concurrently at some point during the operation's interval. Less strict definitions, such as *interval contention* and *total contention* were also used [AF98, AF99b, AAF+99b].

In this work we recognize two parameters for defining the adaptive properties of an implementation. First, an implementation can be *content adaptive* or *fully adaptive*. Then, it can either be adaptive to *total contention*, *interval contention* or *point contention*.

Content Adaptive vs. Fully Adaptive The first parameter describes when is a process considered active for the evaluation of contention. In a fully adaptive implementation, processes are only considered active during the execution of operations.

[1] What Lamport and [ADT95] call fast others call "*adaptive*" [AF98]. Moir and Anderson [MA95, Moi98, MG96, BGHM95] have used the term "fast" to denote algorithms whose complexity is a function of an a-priori known upper-bound on the number of processes that may concurrently access the object.

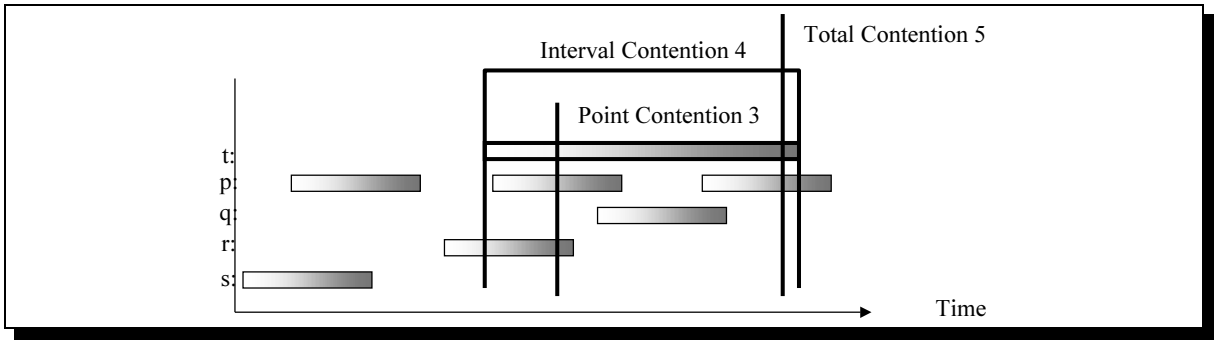


Figure 1.2: Total, Interval and Point Contention during the execution of process t.

However, in a content adaptive implementation a process may also be considered active in some state of the system, even if it is not in the middle of some operation (see Figure 1.1). For example, in the long lived renaming implementations, a process is considered active from the time it has acquired a name until it releases it, even if it is idle between the two operations. Since an algorithm that is fully adaptive is also content adaptive, fully adaptive algorithms are preferable.

Total, Interval and Point Contention Given the set of active processes, it is possible to define the contention during some operation. We recognize three different definitions. The total contention of an execution is simply the total number of processes that participated in the execution. Several algorithms adapt to this parameter (e.g., [CS93]), in particular all the adaptive single shot algorithms ([AM98, AF98, AF99b]). The interval contention during an operation can be informally defined as the number of different processes that were active during the operation while the point contention during an operation can be informally defined to be the maximum number of processes executing concurrently at some point during the operation (see Figure 1.2). Both interval and point contention have been used in previous works.

1.2 Our Results

We present some of the first long lived and adaptive implementations (some of the results are a joint work with Hagit Attiya and Arie Fouren, see [AAF$^+$99b, AAF$^+$99a, AST99]). Many of the adaptive algorithms presented can be used as building blocks for implementing other adaptive algorithms. This property is demonstrated by the implementations of the algorithms themselves since they use each other (see Figure 1.3).

- We start by implementing an adaptive and long lived splitter. The implementation is adaptive to interval contention with step complexity $O(k)$. It is interesting to

Paper	System Response Time	Worst case num of ops
Choy & Singh [CS93]	$O(k)$	$O(N)$
Anderson & Kim [AK99]	$O(\log(N))$	$\begin{cases} O(1), & \text{if } k = 1; \\ O(\log(N)), & \text{otherwise.} \end{cases}$
This work	$O(k^2)$	$O(\min(k^2, k \log(N)))$

Table 1.1: Mutual Exclusion Complexity Comparison

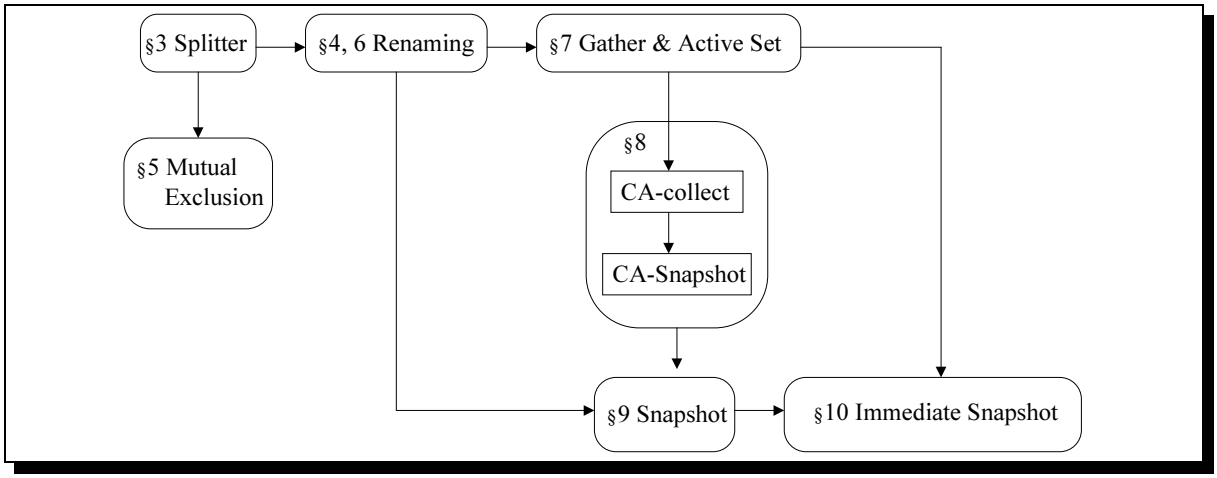


Figure 1.3: Dependencies among the various implementations

note that the worst case execution is extremely rare and that typically the step complexity of the implementation is constant.

- We use the splitter to provide a non-blocking, long-lived, and dynamic optimal name-space $2k-1$-renaming algorithm that renames processes whose initial unique names are taken from a set of size N, with new names taken from a set of size $2k-1$, where k is the number of processes that actually take steps or hold a name while a new name is being acquired.

- We then implement a wait-free long-lived and dynamic k^2-renaming that renames processes to a set of size $4k^2$. The algorithm is content adaptive, where the step complexity of acquiring a new name is $O(k^2)$, and the step complexity of releasing a name is 1. This stands in contrast to all previously known long-lived renaming algorithms whose step complexity is a function of either n or of N [BP89, MA95, BGHM95, MG96, Moi98]. The space complexity of the algorithm is $O(Nn^2)$, where n is an upper bound on the number of processes that may be active (acquiring or holding new names) at the same time, which could be N in the worst case.

- A long-lived adaptive n-process mutual exclusion algorithm is then presented. Both the system response time and the worst case number of operations per process of our algorithm are $O(k^2)$. Previously known algorithms are adaptive only in one of these measures, such as in [CS93] where the system response time is $O(k)$ but the worst case number of operations is $O(n)$ (see Table 1.1).

- To achieve point contention adaptive renaming, a new mechanism (based on [AAF+99b]) is developed and a long lived adaptive to the point contention $O(k^2)$-renaming with step complexity $O(k^2 \log k)$ is constructed.

- The renaming algorithms are used to implement long-lived and content-adaptive constructions of gather and active set. The step complexity of both implementations is $O(k^3)$ adaptive to point contention.

- The active set object is used to implement a long-lived and content-adaptive (CA) collect which is then employed to transform the snapshot algorithm presented in [AAD+93] into a content-adaptive snapshot algorithm. Both implementations are adaptive to point contention. The step complexity of the CA-collect is $O(k^3)$ while the step complexity of the CA-snapshot is $O(k^4)$.

- The content adaptive atomic snapshot is used to construct an adaptive to the point contention snapshot object with step complexity $O(k^4)$.

- Finally, the previous constructions are utilized in the construction of an adaptive to the point contention immediate snapshot object. The algorithm of [AF99b] is

transformed into an adaptive algorithm using the active set and adaptive snapshot objects. The new step complexity of the algorithm is $O(k^4)$ and it is adaptive to point contention.

1.3 Related Work

Since [Lam87], many works tried to extend and generalize Lamport's algorithm to get fast algorithms for different problems and in different models of communication. In [AT92, LS92] Alur and Taubenfeld, and Lynch and Shavit used Lamport's technique to generate a fast mutual exclusion algorithm in a real-time environment. Motivated by [TUX90], Meritt and Taubenfeld [MT93] set to employ Lamport's techniques to generate fast mutual exclusion algorithms for real systems. A lot of work on adaptive mutual exclusion was done this year. In particular, Attiya and Bortnikov presented an adaptive to point contention algorithm with logarithmic step complexity [AB00].

In [AM94] Anderson and Moir set out to employ techniques developed in the PODC community to generate more realistic, resilient and scalable shared objects. They considered the case in which the population of potential processes is huge (size N) but in reality no more than K, $K << N$, processes access an object at the same time (this K is therefore different than the k we defined and use throughout the paper). Towards this end, Anderson and Moir suggested to first protect the object by a K-exclusion algorithm, ensuring that no more than K processes access it at the same time. Processes that have entered through the K-exclusion would next go through a renaming algorithm to reduce their name space to K, thus enabling the usage of an object that was designed only for K or less processes and which is hence more efficient. However, in [AM94] Anderson and Moir considered environments in which strong primitives such as test-and-set are available, which we consider not available in this paper. Motivated by their first work Moir and Anderson developed renaming algorithms, in the read/write model, when such a bound on the maximum number of processes is known in advance. This led to a sequence of works on the renaming problem in this model [MA95, MG96, BGHM95] that lead to a long-lived $(2K-1)$-renaming algorithm with $O(K^2)$ step complexity and $O(K^4)$ space complexity [Moi98]. These works employed various variants of the splitter building block which is a descendant of Lamport's adaptive mutual exclusion algorithm, however the last one [Moi98] depends on an additional work which is the first long-lived renaming algorithm by Burns and Peterson [BP89].

The renaming problem was introduced in the message passing model by Attiya Bar-Noy Dolev Koller Peleg and Reischuk in [ABND$^+$87, ABND$^+$90] where they presented an exponential complexity one-shot $(n+t)$-renaming algorithm where t is a bound on the number of faults (i.e., $2n-1$ in the wait-free model). Bar-Noy and Dolev [BND89] took the problem to the shared-memory model where they presented an exponential step complexity one-shot $(2n-1)$-renaming. Burns and Peterson [BP89] gave an l-assignment

algorithm which boils down to a long-lived $(2n-1)$-renaming algorithm with $O(EXP)$ step complexity. In [Gaf92] Gafni has presented an optimal name space one-shot $(2n-1)$-renaming algorithm with step complexity $O(n^3)$. A one-shot $(2n-1)$-renaming algorithm with step complexity N^2n was later presented by Borowsky and Gafni in [BG93]. Last year, in a beautiful work Attiya and Fouren have introduced a few novel fast one-shot shared-memory objects. In particular they have presented fast one-shot collect and snapshot objects and a $(6k-1)$-renaming object whose step complexity is $O(k \log k)$ [AF98]. Based on their work Afek and Merritt [AM98] present a fast one-shot optimal name space $(2k-1)$-renaming object whose step complexity is $O(k^2)$. Also last year, Moir was able to construct a long-lived and almost fast optimal name space, $(2k-1)$-renaming algorithm, whose step complexity is $O(\log N \cdot k)$. A long lived and fast k^2-renaming was later presented my Attiya and Fouren in [AF99a, AAF$^+$99b, AAF$^+$99a].

Global state determination algorithms have long been used in the message passing model for solving various problems such as deadlock prevention and termination detection [CM82a, MM, CM82b, Fra80]. Chandy and Lamport [CL85] were the first to recognize the importance of snapshots and presented several algorithms for implementing and using them. The first snapshot algorithm implemented in the shared memory model was presented in [AAD$^+$93]. Since then, many snapshot algorithms were suggested in different models and with different complexities [AW99, AR93, RST95]. The first adaptive snapshot was presented in [AF98]. However, this algorithm is only adaptive to the total contention. Borowsky and Gafni extended the definition of a snapshot and defined the immediate snapshot object [BG93, Bor95].

An interesting lower bound was introduced this year by Afek, Boxer and Touitou [ABT00]. They prove that there is no implementation of a long-lived and adaptive renaming or collect object in the atomic read/write model that uses a constant number of MRMW registers.

Chapter 2

Model and Preliminaries

We assume a standard asynchronous shared-memory model of computation using I/O automata [Her91, Tut87, LT87]. For brevity we use pseudocode to describe our implementations.

A concurrent system consists of N processes, $p_1, \ldots, p_N$, communicating through *typed shared memory objects* $A_1, \ldots A_m$. Both processes and memory objects can be described as I/O automata and the concurrent system can be described as the *composite automaton* $\{p_1 \ldots p_N; A_1, \ldots, A_m\}$ (see [Her91]). Each process applies a sequence of *operations* to objects in the system. An operation on an object is defined by two incidents: an *invocation* from the process to the object and a *response* from the object to the process. These incidents are modeled as I/O automata events. An invocation from process p_i to object A_j is modeled as an output event from p_i and an input event to A_j while the response is modeled as an output event from A_j and an input event to p_i. The sequence of I/O automata events is the *history* of the execution. The history is *well formed* if each process starts with an invocation and alternates between matching invocations and responses. There are no fairness assumptions concerning the processes' execution and in particular, processes can not detect whether other processes have halted. If H is a history of a composite automaton and $A_1, \ldots, A_m$ are the components in that composite then we denote by $H|A_i$ the subhistory of H consisting of events of A_i and by $H|A_{i_1}, \ldots, A_{i_x}$ the subhistory of H consisting of events of $A_{i_1}, \ldots, A_{i_x}$.

A *specification* of an object is a description of the object's behavior when accessed by the processes. It defines the transitions of the object from one state to another upon receiving an invocation and the response the object returns. Many of the objects in this thesis cannot be sequentially specified (e.g., collect). Therefore, we usually describe the objects in free style.

An *implementation* $\mathcal{I}$ of an object $\mathcal{A}$ is a concurrent system $\{F_1, \ldots, F_n; R\}$, where R is the data structure that implements $\mathcal{A}$, and F_i is the procedure called by process p_i to execute an operation (w.l.o.g. we assume that $\mathcal{A}$ supports only one operation.

Every other object type can be simulated by passing the operation's type as a parameter). An implementation $\mathcal{I}_j$ of object $\mathcal{A}_j$ is *correct*, if for every history of every system $\{p_1, \ldots, p_N; A_1, \ldots, I_j, \ldots, A_m\}$, there exists a history H' of $\{p_1, \ldots, p_N; A_1, \ldots, A_j, \ldots, A_m\}$ such that $H|\{p_1 \ldots p_N\} = H'|\{p_1 \ldots p_N\}$.

The low level primitive objects from which all the objects in this thesis are composed are atomic read write registers. Every process can read and write to all the registers (MWMR registers). Given a composite automaton $\mathcal{A}$, composed of atomic shared memory registers $r_1, \ldots, r_m$, and a history H, we define the *execution* of $\mathcal{A}$ to be $H|\{r_1, \ldots, r_m\}$ (i.e., removing from H all the local operations and leaving in it only the shared memory invocations and responses). Since the all the operations on the registers are atomic, it is possible to represent every invoke–response pair with a single event, e_i, and to represent the execution as the sequence of read/write events $e_0, e_1, \ldots$.

Consider some execution $\alpha = e_0, e_1, \ldots$ of an implementation of a long-lived object $\mathcal{A}$ and let α' be some finite prefix of α. Then, at the end of α' every process p_i is either *participating* or idle. Unless stated otherwise process p_i is participating at the end of α' if and only if α' includes an invocation of some operation by process p_i without the matching response. The execution segment between two consecutive idle states of process p_i is a *busy period* of p_i.

We define the *total contention* during α to be the total number of processes that took steps in α. We denote it by k_t.

The *active processes* at the end of α', denoted $\mathrm{Cont}(\alpha')$, is the set of processes participating at the end of α'. Given a subsequence β of α, let $\alpha'\beta$ be the shortest prefix of α that contains β, we define the *interval contention* of β and the *point contention* of β, denoted $\mathrm{IntCont}(\beta)$ and $\mathrm{PntCont}(\beta)$ respectively, as follows:

$$\mathrm{IntCont}(\beta) = \Big| \bigcup_{\alpha'\beta' \text{ prefix of } \alpha'\beta} \mathrm{Cont}(\alpha'\beta') \Big|$$

$$\mathrm{PntCont}(\beta) = \max_{\alpha'\beta' \text{ prefix of } \alpha'\beta} |\mathrm{Cont}(\alpha'\beta')|$$

Intuitively, the interval contention of a subsequence β is the number of different processes that were active, (i.e., participating) during β while the point contention is the maximum number of process active at any point of time during β. Clearly, for any subsequence β, $\mathrm{PntCont}(\beta) \leq \mathrm{IntCont}(\beta)$.

For some operation (procedure) of $\mathcal{A}$, *op*, let the *execution interval* of *op*, denoted $\beta(op)$ be the subsequence of α starting at the invocation of *op* and ending at the completion of *op*. The *interval contention* (*point contention*) of an operation *op* is denoted $\mathrm{IntCont}(\beta(op))$ ($\mathrm{PntCont}(\beta(op))$) . We use k_i to denote the interval contention of operation *op* and k_p (or just k) to denote the point contention of operation *op*.

Definition 2.1 *An implementation of* $\mathcal{A}$ *is* adaptive to interval contention *(*adaptive to point contention*) if there is a function* F*, such that the number of primitive operations*

performed by any process p_i in any execution interval of an operation op of $\mathcal{A}$ is at most $\mathrm{F}(k_i)$ *(*$\mathrm{F}(k_p)$*).*

Processes may also be *content active*. Informally speaking a process is considered content active not only when it is participating but also when it is holding some resource of the system. The exact nature of the resource is given during the specification of the object that is implemented. For example, in the following definition of a splitter, a process is considered content active when it has captured the splitter, as well as when it is active. For a long lived collect operation, a process is considered content active when it is active and whenever it has a $\mathrm{non} - \mathrm{value}$ associated with it.

Definition 2.2 *Given a definition for content activeness , we define the* content *point contention during some execution interval β, denoted* $\mathrm{ConPntCont}(\beta)$ *to be the maximum number of processes* content *active at some point of time during β (similarly as above). Likewise, we define the* content *interval contention during some execution interval β, denoted* $\mathrm{ConIntCont}(\beta)$ *to be the number of different processes that were active, (i.e., participating) during β.*

We use $\hat{k}_i$ to denote the content interval contention of operation *op* and $\hat{k}_p$ (or just $\hat{k}$) to denote the content point contention of operation *op*.

Definition 2.3 *An implementation of $\mathcal{A}$ is* content adaptive to interval contention (content adaptive to point contention) *if there is a function* F*, such that the number of primitive operations performed by any process p_i in any execution interval of an operation op of $\mathcal{A}$ is at most* $\mathrm{F}(\hat{k}_i)$ *(*$\mathrm{F}(\hat{k}_p)$*).*

Notice that while every process that is active is also content active, the opposite is not necessarily true. Therefore a content adaptive implementation of an object is not necessarily fully adaptive.

2.1 Splitter

Informally, a splitter is a weak mutual-exclusion primitive. Processes access the splitter by invoking an *acquire* operation and subsequently getting one of three possible responses: $\{$`stop`, `right`, *and* `down`$\}$. A process that got a `stop` response is said to have *captured* the splitter. A process that has captured the splitter releases it by invoking a *release* operation which has only one possible response: *done*.

We consider only well-formed executions in which (a) no process has two or more pending invocations at the same time and (b) a process invokes a *release* operation if and only if its last event was a `stop` response.

An adaptive-splitter has the following properties:

1. At any point in an execution an adaptive-splitter is captured by at most one process (which is a standard mutual-exclusion property).
2. If the prefix of a busy period is an invocation of acquire and its corresponding response, then the response must be a `stop`. (i.e., a process that accesses an uncaptured adaptive-splitter by itself must successfully capture that splitter, also a standard mutual-exclusion property).
3. Not all possible responses to the acquire invocations during a busy period are `right`.
4. Not all possible responses to the acquire invocations during a busy period are `down`.
5. There is no busy period of the adaptive splitter that contains an infinite sequence of events where the events of this sequence are only acquire invocations and `down` responses (i.e. the splitter is used infinitely but no process captures it or goes `right`).

Notice that if two or more processes attempt to acquire an adaptive-splitter concurrently it is possible that none captures it.

The main difference between our adaptive splitter and the Moir-Anderson splitter is Property 5 (see Section 4.1 for a further discussion about the differences).

Definition 2.4 *Given an implementation of a splitter, a process is said to be* content active *from an acquire invocation of the splitter by p until the corresponding response if the splitter was not captured by p (the response was not* `stop`*) and from an acquire invocation of the splitter by p until the* `done` *response to the corresponding release invocation, given that the splitter was captured by p.*

If there was no acquire invocation by p during the execution then the process was idle during the entire execution.

Informally the key difference between the adaptive-splitter and the Moir Anderson splitter is that their splitter guarantees the following two properties (proof of invariant $I25$ in [MA95]):

1. At the point in time in the implementation in which a process is guaranteed to go to the `right` there is another process active in the splitter that might either `stop` or go `down`.
2. At the point in time in the implementation in which a process is guaranteed to go `down` there is at least one other process active in the splitter that either is guaranteed not to go `down` or is undecided (i.e., may still go either `down`, or `right`, or `stop`).

These two properties enable the implementation of a renaming object from a grid (actually half a grid) of size $n \times n$ of such splitters [MA95], while our adaptive-splitter requires a larger grid of size $2n \times 2n$.

2.2 Renaming

In the **long-lived M-renaming** problem, processes repeatedly acquire and release distinct names; the names must be in the range $\{1, \ldots, M\}$. In more detail, a solution to the problem must supply two procedures, acquire-name() and release-name(), for each process. Procedure acquire-name() returns a *new name*, y. A process alternates between invoking acquire-name() and release-name(), starting with acquire-name().

Definition 2.5 *An implementation of the long-lived M-renaming is* **adaptive in the name space** *if the name returned by the high level operation* acquire-name() *is in the range* $1, \ldots, f(k)$*, where op is the interval defined by the execution of the* acquire-name() *procedure.*

Definition 2.6 *Given an implementation of renaming, a process is said to be* content active *from the start of the execution of an* acquire-name() *to the end of its corresponding* release-name().

2.3 Information Retrieval Algorithms

We present several implementations of storage objects such as gather, collect, snapshot and immediate snapshot. These objects typically have the same interface. That is, they have an update operation that enables a process to store new values into the shared memory, and a retrieve operation for retrieving the values stored by the different processes. The exception is the immediate snapshot object which fuses these two operations into one. The objects are long lived, i.e., during an execution processes may repeatedly update their values and scan all the values updated in the shared memory. Once in a while, a process uses the retrieve operation to read the information stored by the processes into its local memory. The algorithms differ in their semantic behavior. Following are the increasingly more strict properties of the various objects.

Definition 2.7 *A* view, *V, is the set of values returned by a retrieve operation. We use the notation $V[p]$ to denote the value associated with process p in view V. If no such value exists then $V[p] = -$.*

Let W_p represent an update operation by process p and let $\mathfrak{W}_p$ be the update value. Let R_q represent a retrieve operation by process q and $\mathfrak{R}_q$ be the view returned by R_q. We use the notation $op_p > op_q$, where op is either a read or a write operation, if operation op_p started after operation op_q finished, and $op_p = op_q$ if they are the same operation.

Definition 2.8 (partial order) *Let $\mathfrak{R}_p, \mathfrak{R}_q$ be two views of the corresponding retrieve operations. We say that $\mathfrak{R}_p$* precedes or equals *$\mathfrak{R}_q$, denoted $\mathfrak{R}_p \preceq \mathfrak{R}_q$ if for every $\mathfrak{W}_r \in \mathfrak{R}_p$ there exists a $\mathfrak{W}'_r \in \mathfrak{R}_q$ s.t., $W_r \leq W'_r$.*

Property I (validity) *For every $v \in \mathfrak{R}_q$ there exists some $\mathfrak{W}_p = v$ s.t., $R_q \not< W_p$. This property asserts that a retrieve operation only returns values that were once written by an update operation.*

Property II (gather) *If $W_p < R_q$ then there is some $W'_p \geq W_p$ s.t., $\mathfrak{W}'_p \in \mathfrak{R}_q$. Intuitively this means that an update operation that finishes before the gather starts is included in the gather result, unless a later update by the same process overwrote.*

Property III (collect) *If $R_p < R_q$ then $\mathfrak{R}_p \preceq \mathfrak{R}_q$. I.e., later collects are at least as updated as previous ones.*

Property IV (snapshot) *For every R_p, R_q either $\mathfrak{R}_p \preceq \mathfrak{R}_q$ or $\mathfrak{R}_q \preceq \mathfrak{R}_p$. This means that all the reads and writes can be linearized.*

Property V (immediate snapshot [BG93]) *Let WR_p represent a combined operation of update and retrieve by process p. Let $\mathfrak{W}_p$ be the value written by p and $\mathfrak{R}_p$ be the view retrieved by p. Then, (1) $\mathfrak{W}_p \in \mathfrak{R}_p$ and (2) for every WR_p, WR_q, if $\mathfrak{W}_p \in \mathfrak{R}_q$ then $\mathfrak{R}_p \preceq \mathfrak{R}_q$. This means immediate snapshot operations can be linearized in sets ([Bor95]) because either $\mathfrak{R}_p \prec \mathfrak{R}_q$, or $\mathfrak{R}_p = \mathfrak{R}_q$ or $\mathfrak{R}_p \succ \mathfrak{R}_q$.*

Definition 2.9 *A valid implementation of a* gather object *satisfies Properties I, II. A valid implementation of a* collect object *satisfies Properties I – III. A valid implementation of a* snapshot object *satisfies Properties I – IV. A valid implementation of an* immediate snapshot object *satisfies Properties I – V.*

Definition 2.10 *Given an implementation of a storage object, a process is said to be* content active *during any of its retrieve operations and between any of its invocations of* update($val \neq -$) *until the next response to* update($-$).

A common data type, called *dataset*, used in our implementations is a set of at most N values containing at most one value for each process. The value associated with processes that do not have a value in the set is considered to be $-$. Examples are the set returned by a collect or snapshot operations. Our adaptive implementations require that the dataset implementation can be stored and retrieved from the shared memory in one operation (i.e., it is stored in one shared memory register). The dataset supports the following operations: read the value associated with a particular process in the dataset, update the value associated with a particular process, enumerate, i.e., iterate over all the non $-$ values in the dataset. To efficiently support these operations we implement this data type with a sparse accumulator data structure [GMS92]. That is, a linked list of the elements and an array of N pointers each pointing to the corresponding element in the linked list. Each element in the list contains the pair ⟨process-id,data⟩. In some of

our implementations the data consists of a value and a sequence number, so the elements are actually 3-tuples. We assume that N values can be atomically read and written to a single shared memory register in one step. Furthermore, we assume that once such an array is read into local memory it is possible to access the array's elements as if it is a normal array, in $O(1)$ steps.

2.4 Adaptive Mutual Exclusion

We follow standard mutual exclusion definitions, such as in [AW98]. We assume that variables used in the entry and exit sections are not accessed in the other sections and that processes do not stay in the critical section forever. Our complexity measurements follow [CS93].

Definition 2.11 *Every process that is participating in the* **mutual-exclusion** *algorithm performs the following code:*

```
while(true) {
  Remainder; // The rest of the code.
  Entry section; // Preparation to enter critical section.
  Critical section; // Protected code.
  Exit section; // Leaving the critical section.
}
```

The correctness of the algorithm requires that:

Mutual exclusion: *In every configuration (i.e., at any point in time) of every execution, at most one process is in the critical section.*

No deadlock: *In every execution, if some process is in the entry section in some configuration, then there is a later configuration in which some process is in the critical section.*

The algorithm is **fair** *if it has the following property:*

No lockout: *In every execution, if some process is in the entry section in some configuration then there is a later configuration in which that same process is in the critical section.*

Definition 2.12 *Given an implementation of mutual exclusion, a process is said to be* content active *when it is not in the Remainder section.*

Definition 2.13 *The* **worst case no. of operations** *of a mutual-exclusion algorithm is informally defined as the worst case number of steps a process may take from the time it enters the Entry section until the next time it enters the Critical section, where every local spinning is counted as one step.*

Definition 2.14 *An implementation of a mutual exclusion algorithm is* **adaptive in the worst case no. of operations** *if the worst case no. of operations for any execution r is only a function of k_t, the total contention in r.*

Following [CS93] we use a notion of time in the system for the next definition. We assume an upper bound on the time any operation may take. This time is constant for local operations and linear in the contention for shared memory operations.

Definition 2.15 *The* **system response time** *of a mutual exclusion algorithm is defined as the worst time it may take for the system to change from a state where there is some process in the entry section and there is no process in the critical section to a state where there is some process in the critical section.*

Definition 2.16 *An implementation of a mutual exclusion algorithm is* **adaptive in system response time** *if the system response time for any execution r is only a function of k_t, the total contention in r.*

Part I

Adapting to Interval Contention

Chapter 3

Splitter

Recall that a splitter is a mutation of mutual exclusion. If one process accesses the splitter alone it should capture that splitter (return `stop` in our code). However, if several processes access the splitter concurrently it is possible that *none* returns `stop`. Moir and Anderson based their one-shot implementation of the splitter on Lamport's fast mutual exclusion algorithm. This algorithm uses two registers, X and Y which are initially empty. To capture the splitter process p writes its name in X and then checks if Y is still empty in which case p writes its name in Y. If after writing in Y its name still appears in X, p is guaranteed that no other process could successfully perform the same sequence of operations. Hence, p may enter the critical section or capture the splitter. In any of the other two cases (Y was already written, or p did not find its name in X after writing to Y) p is assumed to fail. Moir and Anderson noticed that the set of failing processes may be splitted into two groups, *right* and *down*, according to the condition on which they fail. They also show that not all the processes that access the single shot splitter go to the right and not all go down.

The above implementation is inherently single shot. The difficulty in transforming it into a long-lived and adaptive splitter is in reseting the two variables concurrently with other accesses to the splitter[1]. Our implementation of the long-lived and adaptive-splitter is given in Algorithm 1 and Algorithm 2.

3.1 Informal algorithm description

Our approach in making the splitter long-lived is to maintain many copies of the single-shot splitter and to reset the long-lived adaptive-splitter by switching to a new clean single-shot copy. A process that attempts to capture the long-lived splitter decides to

[1]Moir and Anderson have presented long-lived splitters but their long-lived splitters are not adaptive according to the definition used in this paper.

Algorithm 1 Code for SPLITTER

Type:

pid = process id, $0, \ldots, N-1$.

Shared:

$X[0..3N-1]$, $3N$ atomic registers of type pid each initialized to 0.
$Y[0..3N-1]$, $3N$ atomic registers of type $(pid \times boolean)$ each initialized to (0,1).
$Z[0..3N-1]$, $3N$ atomic registers of type pid each initialized to 0.
$status[0..N-1]$, N atomic registers of type $\{start, active, idle\}$ initialized to $idle$.
$I[0..N-1]$, N atomic registers consisting of the pair (i, b),
where i is an integer $0 \leq i \leq 3N-1$, and b is the dirty bit, each initialized to (3N-1,1).
$Master$, an integer in the range $0, \ldots, N-1$ initialized to 0.

Function acquire() for process p returns `stop`, `down` or `right`

1: $status[p] := start$; // Set state to Book-keeping state.
2: $m := Master$;// $Master$ is suppose to be the last process to update the index.
3: $(i, dirtyB) := I[m]$; // i is considered to be the current copy of the splitter
4: $next := i + 1 \bmod 3N$;
5: if $next = 0$ then $nextDB = \neg dirtyB$; // Rap around. Flip the bit
else $nextDB = dirtyB$;
6: if $next \bmod N \neq p$ and $status[next \bmod N] \neq idle$ then $status[p] := idle$;
return `right`;
7: if $(status[(X[i])] = active)$ or // Are processes still alive in current copy
$(status[(Y[i].q)] = active)$ or
$(status[(Z[i])] = active)$ then $status[p] := idle$;
return `right`;
8: $status[p] := active$; // Set state to Single shot splitter state
9: $X[next] := p$; // Emulate single-shot splitter.
10: if $Y[next].b = nextDB$ then $status[p] := idle$;
return `right`;
11: $I[p] := (i, dirtyB)$; // Once p writes in Y some other process might read
// $I[p]$ so it must be up to date.
12: $Y[next] := (p, nextDB)$;
13: if $X[next] \neq p$ then updatel$(next, nextDB)$;
$status[p] := idle$;
return `down`;
$Z[next] := p$;
14: if $X[next] \neq p$ then updatel$(next, nextDB)$;
$status[p] := idle$;
return `down`;
15: updatel$(next, nextDB)$;
16: return `stop`;

Procedure release() for process p.

17: $status[p] = idle$; // Set status to idle
return;

Algorithm 2 Code for UPDATEI()

```
    // Utility procedure for acquire(). The index I is updated only by this procedure
    Procedure updatel(i,dirtyB) for process p..
18:     I[p] := (i, dirtyB);
19:     Master := p;
        // Check if next copy of X, Y, Z is empty. If so, exit. If not, jump to a new i and loop.
        do forever
20:         if Master ≠ p then return;
21:         i = i + 1 mod 3N;
22:         if i = 0 then dirtyB := ¬dirtyB;
23:         if Y[i].b ≠ dirtyB then return;
24:         (q, dirtyB) := Y[i];
25:         if I[q].i ≠ (i − 1) mod 3N then (i, dirtyB) := I[q];
26:                                         I[p]:=(i, dirtyB);
        od;
```

switch to a new single-shot copy when it observes the previous copy to be dirty, meaning some other processes wrote to it and these processes are now idle. However, as described, a copy may appear dirty and idle while in fact the process that has captured that splitter is still active. This happens because after a process has captured the splitter other processes may overwrite both X and Y, thus eliminating all traces of the capturing process. To alleviate this problem we have augmented each single-shot copy by adding to X and Y a Z register. After successfully writing to X and Y without being overwritten in X a process writes its name in Z (we know it is the only one to write in Z) and checks again that its name still appears in X. Only then it captures the splitter (Line 16 in Algorithm 1). A process that has thus captured the single-shot splitter is not only guaranteed to be in mutual-exclusion but also that it is seen by any other process that reads either X or Z. A process releases a splitter by simply writing idle to its status register after which a new acquire would start a new copy (Line 17 in Algorithm 1).

To keep track of which single-shot splitter should be used at any point of time we maintain an adaptive and relaxed pointer. The relaxed pointer has the property that if no process is accessing the adaptive-splitter the pointer points to the last copy of a single-shot splitter that has been used. Therefore, if a process reads the relaxed pointer and the pointer does not point to the last used copy there must be another process that is concurrently accessing the adaptive-splitter. A pointer does not point to the last copy used if the copy immediately after that is not empty, i.e., Y of that copy is not empty (Line 10 in Algorithm 1). This simplifies the algorithm since if process p observes that the copy following the one pointed by the pointer is/was already in use (i.e., the pointer

is incorrect) there must be another active process. In this case p is not accessing the adaptive-splitter alone and therefore may fail in the adaptive-splitter and return `right`.

The relaxed pointer consists of a special register called *Master* that contains a process id, and an array of index registers, one for each process. Our algorithm ensures that whenever the adaptive-splitter is in an idle state the *Master* register contains a process id whose index register points to the last single-shot copy used. To ensure that property, each process p after using a single shot copy, updates its index register to point to the copy it has just used, and then write itself into the *Master* register. However, there may be a situation in which the process is delayed before writing its id to the *Master* and during that delay many new single-shot copies are used. In this case p's index is out of date when it writes its id into *Master*. Therefore p has to traverse the single-shot copies from the last one it touched until it either finds the last copy to be used (detecting the last one as before, see above) or until it reads that some other process overwrote *Master*. In the later case, it is now the responsibility of the other process to make sure its index is up to date (see the code of Procedure updatel in Algorithm 2).

The algorithm thus described is neither adaptive nor bounded wait-free because the sequential traversal of the copies while bringing the index up-to-date is finite but unbounded (described in the previous paragraph). I.e., it is a function of the number of different operations on the adaptive-splitter each of which used a new copy, that took place just before a process wrote its id into *Master*[2]. The algorithm is made adaptive and wait-free by having each process p perform the scan as follows (see Algorithm 2): for each copy scanned, p reads the id of a process that has touched this copy and jumps forward to the copy pointed by the index variable of this process. In the proof we show that while traversing the array of copies in this way process p may not encounter another process more than three times (Lemma 3.12), thus bounding its step complexity to be a function of the number of different processes that took steps concurrently with its operation. Notice that aside from the procedure in which process p updates its index variable, the step complexity of each access is constant (writing to X, reading Y, writing Y, reading X, writing Z and again reading X).

The algorithm thus described is adaptive and long-lived but uses an unbounded number of registers (copies of single-shot splitter). But the unique property of the splitter, namely, that a process that observes another process active may simply fail to acquire the splitter and leave, enables us to bound the number of registers in a simple but innovative way. With each copy of the single-shot splitters we associate a unique process; With copy j we associate the process whose id equals $j \bmod N$. Before accessing copy j processes check that the status of the process associated with this copy is not active (see Line 6 in Algorithm 1). If the status of the associated process is active the processes simply fail the acquire operation and return `right` without touching the new copy. Thus we are able

[2]The step complexity is actually bounded by the number of non-overlapping operations on the adaptive-splitter that processes may do between two consecutive atomic steps of p.

to bound the number of new copies accessed while process p is active. Once the index of the foremost copy mod N equals $p-1$ no new copy is accessed. In the proof we show that this property is enough to ensure that the largest interval of single-shot splitters in use is at most $3N$. Thus, we keep $3N$ copies of the single-shot splitter, i.e., $3N$ copies of X, Y, and Z.

We tag all the Y variables of the single-shot copies with an alternating bit (called *dirty* in the code) and use a reference bit with the *Master* to enable the re-usability of the single-shot copies. When all $3N$ copies have been used the master bit is flipped. This, together with the fact that no more than $3N$ consecutive copies may be touched at any point of time enables the processes to distinguish between empty (not dirty) copies and dirty ones. The index kept by each process is augmented to hold the number of the last copy used and the value of the dirty bit for that copy at that time.

Notice that in our construction a process coming alone captures the adaptive-splitter in $O(1)$ steps. Furthermore, a captured adaptive-splitter is released in 1 step by the corresponding process changing its status to idle.

Theorem 3.1 *The code in Algorithm 1 implements a long lived splitter. Further more it is adaptive to interval contention with step complexity* $O(k_i)$.

3.2 Correctness of the Implementation

We now prove that the implementation satisfies Definition 2.1.

For this proof we informally define a *run* as a possibly infinite sequence of read and write events performed by the processes executing the adaptive-splitter algorithm, and a *busy run* as a run of a single busy period. We prove the correctness by first partitioning any run of the algorithm into sub-sequences, called *intervals*. During the lth interval all processes accessing a new single-shot copy access the same copy, copy $l+1 \bmod 3N$, and all access it with the same value in the dirty-bit. In other words, during interval l the single shot copy that processes try to capture is copy $l+1 \bmod 3N$.

Definition 3.1 *Let* $r = e_0, e_1, e_2, \ldots$ *be a run of the algorithm. The* transition subsequence *of* r, $te_{-1}, te_0, te_1, \ldots$ *is a possibly infinite subsequence of events from* r *inductively defined as follows:*

- $te_{-1} = e_0$.
- *Assuming* te_{l-1} *exists, then* te_l *is the first assignment event to any* $I[q]$ *to occur after* te_{l-1} *where* $< I[q] := (l \bmod 3N, \lfloor l/3N \rfloor \bmod 2) >$. *If there is no such event we say that* te_l *does not exist.*

Definition 3.2

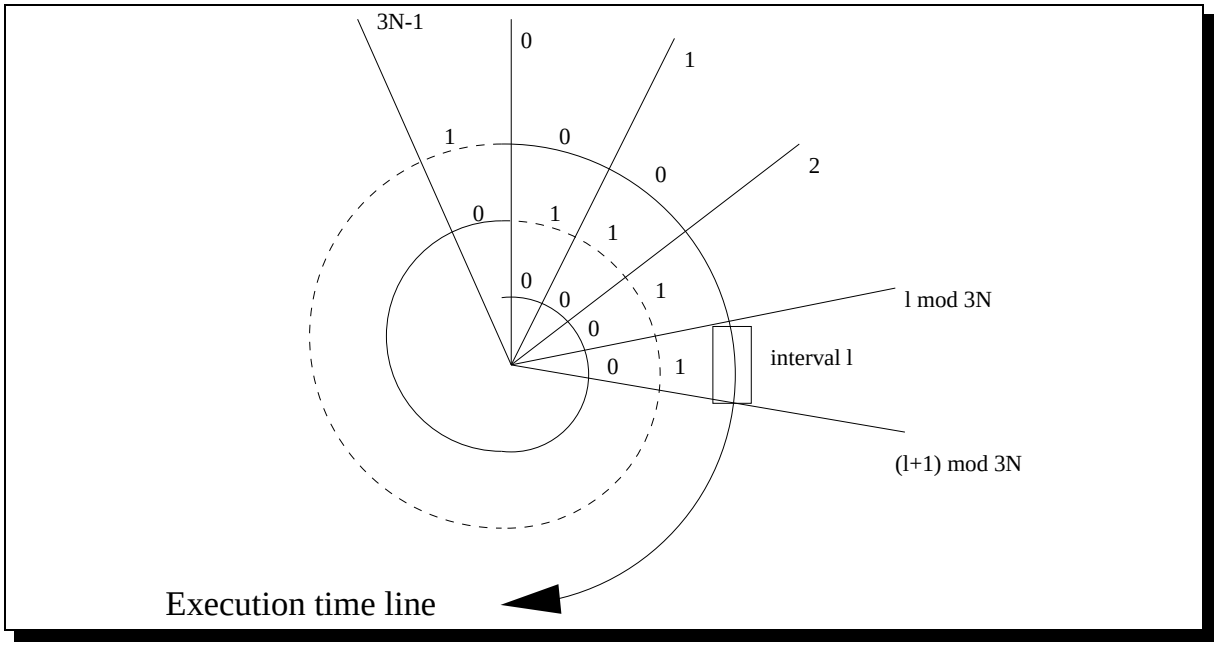


Figure 3.1: Calculating $index(l)$. The first part of $index(l)$ is represented by the sector ($l \bmod N$) and the second part is represented by the execution's time-line pattern ($\lfloor l/3N \rfloor \bmod 2$).

- *The* l'th interval *of run* r *is the subsequence of events in* r *starting at the first event following* te_l *and ending at* te_{l+1} *if exists. If* te_{l+1} *does not exist, then the* l*'th interval extends to the end of the run. Sometimes we denote the* l*'th interval just by* l*.*

- E_l *is the subsequence of events in interval* l*.*

It follows from Definition 3.2 that the intervals of any run r, are $-1, \ldots, l$ where l is the last interval in r.

Definition 3.3 SP_p^k *is the subsequence of events performed by process* p *in its* k*'th execution of the splitter algorithm (the* acquire() *and the corresponding* release() *procedure if the acquire operation returned* stop*). When it is not important we omit the subscript and/or superscript of* SP*.*

Definition 3.4

- $startI(SP_p^k)$*, the starting interval of* SP_p^k*, is the interval index in which the write of Line 1 of* SP_p^k *occur. I.e., let event* $e \in E_l$ *such that* $e \in SP_p^k$ *be the write event performed at Line 1. The* **start interval of** SP_p^k*, denoted* $startI(SP_p^k)$ *is then equal to* l*.*

- *$readI(SP_p^k)$, the read index interval of SP_p^k, is the interval index in which the read of Line 3 of SP_p^k occur. I.e., let event $e \in E_l$ such that $e \in SP_p^k$ be the read event at Line 3. The* **read index interval of** *SP_p^k, denoted $readI(SP_p^k)$ is then equal to l. The pair (i, b) read in e is the index value of SP, denoted $index(SP_p^k)$.*
- *Given an interval l, the index value of l, denoted $index(l)$ is the value $(l \bmod 3N, \lfloor l/3N \rfloor \bmod 2)$ (see Figure 3.1).*
- *We say that SP_p^k is a* `down` *or* `stop` *execution, denoted DoS-execution, if p has reached Line 11 (not including) in the code during the execution of SP_p^k*

The following lemma states that the arithmetic calculations for increasing the index actually work. That is, if the relevant registers contain $index(l)$ before the calculation, then after the calculation they contain $index(l+1)$.

Lemma 3.2 (helper) *During the execution of the algorithm presented in Algorithms 1 and 2 the following holds:*

1. *If immediately after Line 3 $(i, dirtyB) = index(l)$ then immediately after Line 5 $(next, nextDB) = index(l+1)$.*
2. *If immediately before Line 21 $(i, dirtyB) = index(l)$ then immediately after Line 22 $(i, dirtyB) = index(l+1)$.*

Proof: 1. From the code in Line 4 it follows that

$$next = (i+1) \bmod 3N = (l \bmod 3N + 1) \bmod 3N = (l+1) \bmod 3N$$

From Line 5 it follows that:

$$nextDB = \begin{cases} \lfloor l/3N \rfloor \bmod 2 & (l+1) \bmod 3N \neq 0 \\ \neg(\lfloor l/3N \rfloor \bmod 2) & (l+1) \bmod 3N = 0 \end{cases}$$

But

$$(l+1) \bmod 3N \neq 0 \;\Rightarrow\; \lfloor l/3N \rfloor \bmod 2 = \lfloor (l+1)/3N \rfloor \bmod 2$$
$$(l+1) \bmod 3N = 0 \;\Rightarrow\; \lfloor (l+1)/3N \rfloor \bmod 2 = (l+1)/3N \bmod 2 = (\lfloor l/3N \rfloor + 1) \bmod 2$$

which means that $nextDB = \lfloor (l+1)/3N \rfloor \bmod 2$ or that $(next, nextDB) = index(l+1)$.

2. From the code in Lines 21 and 22 and from exactly the same calculations as in Part 1 it follows that the second part of the lemma holds.

■

The following lemma captures the main characteristics of the algorithm and is the basis for the proof of the adaptive-splitter properties. Note that properties I1a and I1b are separated because of the special case where process p starts at some interval and reads the index in some later interval (Line 3) where $i+1 \bmod N = p$.

Lemma 3.3 (Invariants) *The following invariants hold for the splitter algorithm:*

I1a *If $e \in SP$, $e \in E_l$ and e is the read of Y in line 10 then $startI(SP) \geq l - N$. (That is, no more than N new intervals may start after a process starts the execution of* acquire() *and before it reads the content of register Y).*

I1b *If $e \in SP$, $e \in E_l$ then $readI(SP) \geq l - N$. (That is, no more than N new intervals may start after a process reads the index in Line 3 and before it finishes the execution).*

I2 *If SP is a DoS-execution then $index(SP) = index(readI(SP))$. (That is, if a process reached Line 11 then the index value it has read in Line 3 equals to the index of the interval that contains this read.)*

I3 *For every process r and event $e \in E_l$, if SP is the last DoS-execution of r that has reached Line 11 before e, then variable $I[r]$ contains at e $index(l')$ where $readI(SP) \leq l' \leq l$. Furthermore:*

1. *If the value of $I[r]$ was taken from some $I[q]$ during* updatel() *(Lines 24-26) then the adopted value was written by some SP_q such that $readI(SP_r) \leq readI(SP_q)$.*
2. *If after Line 11 during SP_r the value in $I[r]$ changes from $index(l')$ to $index(l'')$ then $l' < l''$.*

I4 *If $e = te_l$ then there is some SP such that $readI(SP) = l - 1$ and SP has written to $Y[l \bmod 3N]$ during interval $l - 1$.*

I5 *If $e \in E_l$ is a write to $Y[j]$ for some $0 \leq j \leq 3N-1$ such that e has changed the value of $Y[j].b$ then $j = l+1 \bmod 3N$ and the new value of $Y[j].b$ is $(\lfloor (l+1)/3N \rfloor) \bmod 2$.*

Proof: We prove the lemma by induction on the event e, in a run α of the algorithm. For the base of the induction, assume that $e = e_0 \in E_{-1}$ and without loss of generality, assume $e_0 \in SP_p$.

1. I1a,I3,I4 and I5 are immediately correct since e_0 cannot be the event referenced in the corresponding invariant.
2. I1b is correct because $readI(SP_p) \geq -1$ and it also holds that $readI(SP_p) \geq -1 - N$.

3. I2 holds since SP_p is not yet a DoS-execution.

Assume the lemma is correct for $e_0, \ldots, e_i \in \alpha$. We show correctness for event $e = e_{i+1}$.

I1a Assume by contradiction that $e \in SP_p, e \in E_l$, e is the read of Y in Line 10 during SP_p and $startI(SP_p) < l - N$. By the definition of the invariant and of the intervals it follows that intervals $-1, \ldots, l$ occurred and occurred in order. Therefore, there was some interval $startI(SP_p) < l' < l$ such that $l' + 1 \bmod N = p$. From the induction on I4 there was some process, r, such that $readI(SP_r) = l'$ and r updated $Y[l' + 1 \bmod 3N]$ during l'. Furthermore, Process p has not yet updated Y in SP_p and it follows that $p \neq r$.

Lets look at SP_r. From I4 and the helper lemma (Lemma 3.2) it follows that just before executing Line 6, $next = l' + 1 \bmod 3N = p \bmod 3N$. Since $r \neq p$ the first part of the ifin Line 6 statement holds. Since p set $status[p]$ in interval $startI(SP_p) < l'$ and the bit is still set during l it was set at least until $te_{l'+1}$. But SP_r wrote into $Y[l' + 1 \bmod 3N]$ during l' so it must have read $status[p]$ as set. It follows that the second part of the ifstatement in Line 6 also held and SP_r returned without updating Y. As we chose SP_r to be an execution that updates Y we get a contradiction.

I1b Following exactly the same arguments as in the beginning of I1a it follows that $p \neq r$.

As in I1a, lets look at SP_r. From the helper lemma it follows that just before executing Line 6, $next = l' + 1 \bmod 3N = p \bmod 3N$. Since $r \neq p$ the first part of the ifstatement holds. Since p set $status[p]$ in interval $startI(SP_p) < l'$ and the bit is still set during l it was set at least until $te_{l'+1}$. As SP_r wrote into $Y[l'+1 \bmod 3N]$ during l',it follows that the second part of the ifstatement in Line 6 also held and SP_r returned. But we chose SP_r to be an execution that updates Y and we get a contradiction.

I2 It is immediate that the induction on I2 continues to hold for all events e that are not the read of Y in Line 10 in which a new DoS-execution is created. Assume to the contradiction that $e \in SP_p, e \in E_l$, e is the read of Y in Line 10 in which it was decided that SP_p is a DoS-execution and $index(SP_p) \neq index(readI(SP_p))$.

Assume that after Line 2 in SP_p, $m = q$. This means that immediately after Line 3, $(i, dirtyB) = I[q]$. From I4 there is some l' such that $I[q] = index(l')$ (if there were no DoS-executions of q then $I[q] = index(-1) = (3N - 1, 1)$).

1. Assume that no value was ever written into $Master$ during this run. This means $m = q = 0$. If no new value was written to $I[0]$ too then $index(SP_p) =$

$index(-1)$ and the claim is correct. If p_0 wrote a new value into $I[0]$ it could have done it only once (since $Master$ has never been updated) and therefore, from the helper lemma, the value written must have been $index(0)$. It follows that $readI(SP_q) = -1$, $I[q] = index(l')$ where l' is either 0 or -1, and either $readI(SP_p) = 0$ or $readI(SP_p) = -1$. But since $l' \leq readI(SP_p)$ (the read in Line 3 is from I[0]) it follows that $readI(SP_q) \leq l' \leq readI(SP_p) \leq l$. From the assumption to the contradiction it follows that:

$$readI(SP_q) \leq l' < readI(SP_p) \leq l$$

2. If a value was written into $Master$ then q must have written itself into $Master$. Furthermore, the execution, SP_q, in which q wrote in $Master$ is its last DoS-execution that reached Line 11 before e (even if $p = q$). Since the read of $index(SP_p)$ happens before e we can use the induction on I3 to deduce that there is an interval l' such that $I[q] = index(l')$ and $readI(SP_q) \leq l' \leq readI(SP_p)$. By the assumption to the contradiction it follows that:

$$readI(SP_q) \leq l' < readI(SP_p) \leq l$$

It then follows that $readI(SP_p) > l'$. This means event $te_{l'+1}$ must have occurred before e, consequently, from I4, there was some process, r, that updated the $(l' + 1) \bmod 3N$ copy of Y ($Y[(l' + 1) \bmod 3N]$) during interval l'. Since SP_r is a DoS-execution and since r read the index in l' the value written was:

$$Y[l' + 1 \bmod 3N].b = (\lfloor (l' + 1) \bmod 3N \rfloor) \bmod 2$$

Invariant I5 guarantees that no other process writes a different value to $Y[l' + 1 \bmod 3N].b$ at least until $te_{l'+1}$ (including).

1. If the last DoS-execution of q is still active in $readI(SP_p)$ then from I1b, $readI(SP_p) - readI(SP_q) \leq N$, which means $readI(SP_p) - l' \leq N$, which surly means that $startI(SP_p) - l' \leq N$
2. If the last DoS-execution of q has finished and q is still written in $Master$ then SP_q must have finished at least in interval $startI(SP_p) - 1$. This is because from I4 during $startI(SP_p) - 1$ there was some process r that wrote into $Y[startI(SP_p) \bmod 3N]$. Lets look at the events of SP_r executing after Line 7. From the assumption interval $startI(SP_p)-2$ has also finished. Therefore some process wrote into $Y[startI(SP_p)-1 \bmod 3N]$ during interval $startI(SP_p)-2$. But r read $Y[startI(SP_p)-1 \bmod 3N]$ and did not see any active process. This means one of the processes that wrote into $Y[startI(SP_p)-1 \bmod 3N]$ during

interval $startI(SP_p) - 2$ has already finished but then it would have written into $Master$, overriding q. From this follows that $startI(SP_p) - readI(SP_q) \leq N + 1$.

From I1a $l - startI(SP_p) \leq N$ meaning that in both cases, $l - l' \leq 2N + 1$. This is why we need at least $2N + 2$ copies in our algorithm[3].

During interval l' the shared memory $Y[l' + 1 \bmod 3N].b$ is updated and from the induction on I5 it follows that $Y[l' + 1 \bmod 3N].b$ is not updated again until interval l.

From the helper lemma if follows that just before e in SP_p, $nextDB = \lfloor l' + 1/3N \rfloor \bmod 2$. Therefore, the ifstatement in Line 10 must succeed during SP_p which is in contradiction to the assumption that SP_p is a DoS-execution. This means our assumption to the contradiction was wrong and the induction holds.

I3 Assume that $e \in SP_p, e \in E_l$.

1. For every process $r \neq p$ the claim holds by induction.
2. For every $e \in SP_p$, e is not an update of $I[p]$ the claim holds by induction.
3. If e is the update of $I[p]$ in Line 11 then from I2 (not by induction) $index(SP_p) = index(readI(SP_q))$ and therefore after Line 11 $I[q] = index(readI(SP_p))$. Also, this must be the first time $I[p]$ is updated in this execution which means Parts 1,2 hold.
4. If e is the update of $I[p]$ in Line 18 then, from the helper lemma and I2, $I[p] = index(readI(SP_p) + 1)$. Also, this must be the second time $I[p]$ is updated in this execution which means Parts 1,2 continue to hold.
5. If e is the update of $I[p]$ in Line 26 then from the induction hypothesis the last value in $I[p]$ was some $index(l')$ such that $readI(SP_p) \leq l' \leq l$. We study the steps of SP_p since the last update of $I[p]$.
   (a) As we assume we reach Line 26 in the new iteration the ifstatement in Line 20 must have failed.
   (b) From the helper lemma it follows that after Line 22, $(i, dirtyB) = index(l' + 1)$.
   (c) As we reach Line 26 the ifstatement in Line 23 must have failed. Therefore $Y[l' + 1 \bmod 3N].b = \lfloor (l' + 1)/3N \rfloor \bmod 2$.
   From I5 it follows that the last meaningful update to $Y[l' + 1 \bmod 3N].b$ was in one of the intervals $\ldots, l' - 6N, l', l' + 6N, \ldots$. But, since we reached

[3] In fact, $l - l' < 2N$ which means 2N copies are enough. It can be shown that if SP_p and SP_q overlap then, as $p \neq q$, $l - readI(SP_p) \neq N$. If the executions do not overlap then SP_q has finished and therefore $I[q] = index(readI(SP_p))$.

interval l' it must be that $te_{l'-3N}$ occurred and from the induction on I4 during interval $l'-3N$ the value ($\neq \lfloor (l'+1)/3N \rfloor \bmod 2$) was written into $Y[l'+1 \bmod 3N].b$. Also, $readI(SP_p) \leq l' \leq l$ which means $l - readI(SP_p) \leq N$ or that $l - l' \leq N$. This means that the last value read in $Y[l'+1 \bmod 3N].b$ was indeed written in interval l' by some process q. From the helper lemma and Line 12 it follows that if any process updates $Y[l'+1 \bmod 3N]$ then $index(SP_q) = \ldots, index(l'-3N), index(l'), index(l'+3N), \ldots$. Note that SP_q is a DoS-execution and therefore, from I2, $index(readI(SP_q)) = index(SP_q)$. But as the update happens during l' it follows from I1b that $index(readI(SP_q)) = index(l')$ or that $readI(SP_p) \leq readI(SP_q) = l'$ and Part 1 holds.

(d) Because $l - l' \leq N$ the value in $Y[l'+1 \bmod 3N]$ did not change between Line 23 and Line 24. This means that after Line 24 the value in $dirtyB$ did not change and the value in variable q is the name of process q. If the ifstatement in Line 25 succeeds than by the induction on I3 for SP_q, some value,$index(l'')$, is assigned to $I[p]$ such that

$$readI(SP_p) \leq l' = readI(SP_q) \leq l'' \leq l$$

By the success of the ifstatement, $l'' \neq l'$ which means $l' < l''$. If the ifstatement fails then $index(l'+1)$ is assigned to $I[p]$, where $readI(SP_p) \leq l' \leq l$. Assume to the contradiction that $l'+1 = l+1$. Once $l'+1$ is assigned to $I[p]$, interval $l+1$ starts in contradiction to the assumption that $e \in E_l$. Therefore $l'+1 \leq l$ and the claim holds.

I4 Assume that $e \in SP_p, e \in E_{l-1}, e = te_l$. This means e is the assignment:

$$I[p] := (l \bmod 3N, \lfloor l/3N \rfloor \bmod 2)$$

1. If the assignment is made in Line 11 then $I[p] := index(SP_p)$. Since SP_p is a DoS-execution it follows from I2 that $I[p] := index(readI(SP_p))$. If $l = readI(SP_p)$ then e is not te_l. If not then $readI(SP_p) = l - 6N$ (or less) but this contradicts I1b and therefore it is impossible that te_l is the assignment in Line 11.
2. If the assignment is made in Line 18 then in Line 12 in the execution SP_p, p wrote into $Y[l \bmod 3N]$, hence $index(SP_p) = index(l-1)$. From I2 it follows that $index(readI(SP_p)) = index(SP_p) = index(l-1)$ and from I1b $readI(SP_p) = l-1$. Since interval $l-1$ started before SP_p performed Line 3 and ends in e (Line 18) it follows that the update in Line 12 happened during interval $l-1$ and the claim holds.

3. If the assignment happens in Line 26 then let $index(l')$ be the value that was written in $I[p]$ just before e. From I3 $readI(SP_p) \leq l' \leq l$. Furthermore, $e = te_l$, meaning that $l' < l$. We study the operations in SP_p since the last update of $I[p]$. From the helper lemma it follows that after Line 22 $(i, dirtyB) = index(l'+1)$. From the same reasons as in 5c the update of $I[q]$ by process q must have happened during interval l'. Process q wrote $index(l'-1)$ into $I[q]$ and, if in Line 25 $I[q] = index(l)$ it must have been updated after $readI(SP_p)$ which means at most N intervals before l which means in $l-1$. Thus e is not te_l in contradiction to the definition. Therefore, it must be that the ifstatement in Line 25 failed. But this means $l' = l-1$ which means q updated $Y[l \bmod 3N]$ in interval $l-1$. The updating execution SP_q is a DoS-execution and since q updated $Y[l \bmod 3N]$, $index(readI(SP_q)) = index(SP_q) = index(l-1)$ and as $I[q]$ was updated after $readI(SP_p)$, $readI(SP_q) = l-1$ and the claim holds.

I5 Assume to the contradiction that $e \in SP_p, e \in E_l$ a write to some $Y[j]$ that changes $Y[j].b$ but either $j \neq l+1 \bmod 3N$ or $j = l+1 \bmod 3N$ and the value written to $Y[j].b$ is not $\lfloor (l+1)/3N \rfloor \bmod 2$. Since SP_p is a DoS-execution if follows from I2 that $index(SP_p) = readI(SP_p)$. From the helper lemma it follows that if Y was written to during SP_p it must have been to entry $Y[(readI(SP_p)+1) \bmod 3N]$ and the written value was $\lfloor (readI(SP_p)+1)/3N \rfloor \bmod 2$. From the assumption:

1. If $(readI(SP_p)+1) \bmod 3N \neq (l+1) \bmod 3N$ then it must be that $readI(SP_p) \leq l$. This means interval $readI(SP_p)$ has already finished and therefore, from I4, there was some process, q, that wrote into $Y[(readI(SP_p)+1) \bmod 3N]$ during $readI(SP_p)$. Furthermore, from I4 $readI(SP_q) = readI(SP_p)$, therefore the value written to $Y[(readI(SP_p)+1) \bmod 3N].b$ was $\lfloor (readI(SP_p)+1)/3N \rfloor \bmod 2$. This means that between $readI(SP_p)$ and l there was another write to $Y[(readI(SP_p)+1) \bmod 3N]$ with the opposite value. But from the induction this happens only in intervals $readI(SP_p)+3N, readI(SP_p)+9N, \ldots$ and, as $l - readI(SP_p) \leq N$, it is impossible and the claim holds.
2. If $readI(SP_p)+1 \bmod 3N = l+1 \bmod 3N$ but $\lfloor (readI(SP_p)+1)/3N \rfloor \bmod \neq \lfloor (l+1)/3N \rfloor \bmod 2$ then $readI(SP_p) = l - 3N$. But $e \in E_l$ and from I1b $l - readI(SP_p) \leq N$. Therefore this case is impossible.

■

We use the invariants to prove the four properties of an adaptive-splitter.

Lemma 3.4 (Property 1) *At any point in an execution an adaptive-splitter is captured by at most one process.*

Proof: Assume otherwise. Let p, q be the processes that returned `stop` concurrently.

1. Assume that $l = readI(p) = readI(q)$. Since SP_p and SP_q are DoS-executions it follows from I2 that $index(SP_p) = index(readI(SP_p)) = index(readI(SP_q)) = index(SP_q)$ which means that p and q access the same copy and hold the same value for $nextDB$. This means p and q agree on which bit value means Y is free and which means Y is occupied.

   From I1b it follows that at most N intervals have passed from l until both runs finished. Therefore, it follows from I5 that $Y[l \bmod 3N].b$ has changed at most once in that period. This means it must be that both processes first wrote to X (Line 9) then read Y and found it empty (Line 10) and only then both processes wrote to Y (line 12). Without loss of generality assume that p was the first to write into X. Then, in Line 13, p reads the id of q in X and returns `down` in contradiction to the assumption.

2. Assume $readI(p) > readI(q) = l$. From I4 it follows that there is a process r such that $readI(SP_r) = readI(SP_q)+1$ and r has written to $Y[l+2 \bmod 3N]$ (note that it is possible that $r = q$). As both SP_p and SP_r are DoS-executions if follows from I2 that $index(r) = index(readI(SP_r)) = index(l+1)$ or:

$$\begin{aligned} index(q) &= (l \bmod 3N, \lfloor l/3N \rfloor \bmod 2) \\ index(r) &= (l+1 \bmod 3N, \lfloor (l+1)/3N \rfloor \bmod 2) \end{aligned}$$

   Process q returned `stop` it must have changed its status bit from idle, wrote its name in $X[l+2 \bmod 3N]$, wrote its name in $Z[l+2 \bmod 3N]$ and read its name in $X[l+2 \bmod 3N]$.

   Process r updated $Y[l+2 \bmod 3N]$ and therefore it must have passed line 7. This means r read $X[l+1 \bmod 3N]$ and did not see q written in it. Then r read $Z[l+1 \bmod 3N]$ and did not see q written there too (Note that, from the assumption, the status bit must stay on until after p returned `stop`). As r is using the $l+2 \bmod 3N$ copy, some other process, say s, updated $Y[l+1 \bmod 3N]$ (and also $X[l+1 \bmod 3N]$) and finished before Line 18 of SP_r was executed (I4). If r read $X[l+1 \bmod 3N]$ before q had wrote into it then q must have seen the write of s in $Y[l+1 \bmod 3N]$ and could not have returned `stop`, in contradiction to the assumption. If r read $X[l+1 \bmod 3N]$ after q wrote into it then this event must have occurred strictly after q wrote into $Z[l+1 \bmod 3N]$ (Line 13). From the previous case analysis we know that in the same interval only one process can write into $Z[l \bmod 3N]$. From I1b and I5 if follows that two runs with different $readI()$ values can not update the same copy so r must see the write of q in $Z[l+1 \bmod 3N]$ and return `right` in contradiction to the assumption that it wrote into $Y[l+2 \bmod 3N]$ (Line 7).

■

Lemma 3.5 *Let α be a prefix run of the adaptive-splitter and assume that after α the splitter is in an idle state. Denote by $i = I[Master].i$ and by $b = I[Master].b$ after α. Then in this state $Y[(i+1 \bmod 3N)].b \neq (b + (i+1)/3N) \bmod 2$.*

Proof: Let p_j be the last process to write into $Master$ before the idle state, and let SP be its last execution. Since p_j wrote into $Master$ it must have executed updateI(). Since p_j is the last one to write to $Master$ the condition in Line 20 always fail. Since we assume p_j stops (otherwise there is no idle state) it must have finished the loop in Line 23.

As $I[j]$ is updated at the beginning of updateI() (Line 18) and after every turn of the loop (Line 24) but the last, it holds in Line 23 that $i = I[j].i + 1 \bmod 3N$.

No other process p_k could have written to $Y[I[j].i]$ after the read operation of p_j in Line 23. This is because then p_k must be running in a DoS-execution so it must have updated $Master$ after it has updated Y, meaning after p_j updated $Master$ in contradiction to the assumption.

We show that when process p_j performed Line 23, $Y[i].b \neq I[j].b + \lfloor (I[j].i+1)/3N \rfloor \bmod 2$, by proving that at Line 23, $dirtyB = I[j].b + \lfloor (I[j].i + 1)/3N \rfloor \bmod 2$.

$I[j].b$ is updated at the entry to updateI() (Line 18) and in each iteration but the last (Line 24). Therefore, $dirtyB$ can be different from $I[j].b$ only if in the last iteration $I[j].i + 1 \bmod 3N$ is equal to 0 which is equivalent in all cases but the first to the case where $I[j].i + 1 = 3N$ and the claim holds. ■

From this lemma it follows that:

Corollary 3.6 (Property 2) *If the prefix of a busy period is an invocation of acquire and its corresponding response, then the response must be a* `stop`.

Lemma 3.7 (Property 3) *Not all responses to the acquire invocations during a busy period are* `right`.

Proof: Let α be a busy run of the splitter. Let α_0 be the largest prefix of α that does not include events of the write assignment in Line 8. That is, the next event after α_0 (if such an event exists) is the first write of any process p of the value *active* into $status[p]$.

Assume to the contradiction that $\alpha_0 = \alpha$. This means all the processes participating in α had to finish the algorithm in Line 6. Therefore, during α the value of $Master$ and the corresponding process index did not change. So all processes read the same value to $(i, dirtyB)$. But that includes the process $p = next \bmod N$ which (from the success of the ifstatement for the other processes) was active in α. But process p should have passed the ifstatement in Line 6 meaning we get a contradiction. Therefore $\alpha_0 \subset \alpha$.

Let $\alpha_1 \supset \alpha_0$ be the largest prefix of α that does not include updates to Y. Assume to the contradiction that $\alpha_1 = \alpha$. From the first part we know that there is at least one

execution, say SP_p, that did not end in α_0. This means SP_p must end in Line 10. From Lemma 3.5 it follows that after SP_p performed Line 5 (in α_0) the variable i points to a copy of Y where $Y[next].b \neq nextDB$. But, since Y is not updated in α_1, the ifstatement in Line 10 must fail in SP_p and p does not end, in contradiction to the assumption.

Because $\alpha_1 \neq \alpha$ some process executes Line 12 and, from the algorithm, does not return `right`. ■

Lemma 3.8 (Property 4) *Not all responses to the acquire invocations during a busy period are* `down`.

Proof: Let α be a busy run of the splitter. Assume to the contradiction that all the processes participating in α returned `down`. Let α_0 be the largest prefix of α that does not include events of update to Y. From the assumption to the contradiction if follows that $\alpha_0 \neq \alpha$. Furthermore, no process that participates in α could have finished in α_0 because then it would have had to return `right`. Let $e' \in SP_p$ be the next operation after α_0. That is, the update of Y.

1. No process in α_0 updates $Master$ because updatel() is called only after Y is updated.

2. If during the idle state before α, $Master = p$ then p did not change the value in $I[p]$ (at most, p rewrote the same value read from itself). This means all the participating processes in α_0 read the same value $(i, dirtyB) = index(l')$.

3. Since e' is the write to $Y[l' + 1 \bmod 3N]$ and as this is the first write since the idle state, this must also be a write that changes the value of $Y[l' + 1 \bmod 3N].b$ to $\lfloor l' + 1/3N \rfloor \bmod 2$. From I5 it follows that $e' \in E_{l'}$.

4. Assume that there is some process q that participates in α_0 and at the end of α_0 still does not read $Y[l' + 1 \bmod 3N].b$. Let $e \in E_l, e \in SP_q$ be that read in α. From I1a $l' = startI(SP_q) \geq l - n$ (q participates in α). But from I5 the change in $Y[l' + 1 \bmod 3N].b$ can only happen during interval $l' + 3N$. It follows that during SP_q the ifstatement in Line 11 succeeded and q returned `right` in contradiction to the assumption.

   Therefore, all the processes participating in α_0 had to pass the read statement of Y in Line 10 before the end of α_0. Particularly, they had to pass the update of $X[l' + 1 \bmod 3N]$.

5. Let r be the last process participating in α_0 to update $X[l' + 1 \bmod 3N]$, SP_r its execution.

   Let $e \in E_l, e \in SP_r$ be the read statement of X in Line 13 and assume that $X[l' + 1 \bmod 3N] \neq r$. Then some execution, SP_q, that did not participate in α_0 wrote into $X[l' + 1 \bmod 3N]$ between the write and read of X by SP_r. From the

assumption, all processes in α return `down` so they all execute DoS-executions. From I2 $index(SP_q) - index(readI(SP_q))$ which means $readI(SP_q) = l' - 3N, l', l' + 3N \ldots$. But SP_q started after α_0 so $l' \leq readI(SP_q)$. Also, from I1b $l' = readI(SP_r) \geq l - N$ and $readI(SP_q) \leq l$ which means $readI(SP_q) = l'$ from which follows that $index(SP_q) = index(l')$ (and not $index(l' + 3N)$ for example).

From I1b and I5 it follows that SP_q read from $Y[l' + 1 \bmod 3N]$ in Line 10 the value written by SP_r during interval l' so the ifstatement must succeed and q returns `right` in contradiction to the assumption. Therefore it is impossible that r read a name of another process in X.

From the same reasons as before it is impossible that r read a name of another process in $X[l'+1 \bmod 3N]$ in Line 14 and therefore r returns `stop` in contradiction to the assumption.

This means the assumption to the contradiction is false.

■

Lemma 3.9 (Property 5) *There is no busy period of the adaptive splitter containing an infinite segment of consecutive* `down` *responses.*

Proof: Assume to the contradiction that such a busy period exists. We denote its busy run by α, and the infinite sequence of consecutive `down` responses by α_0.

Without loss of generality, we assume α_0 starts at interval 0 (the value of the first interval of α_0 has no effect on the proof).

Since there is an infinite number of DoS-executions in α_0, but only a finite number of processes, if follows that some process, p, executed an infinite number of DoS-executions, $SP_p^k, k = 1, \ldots, \infty$, in α_0.

From I2 it follows that for each such SP_p^k, $index(SP_p^k) = index(readI(SP_p^k)$. From the helper lemma and since Line 18 must execute in a DoS-execution it follows that p writes $index(readI(SP_p^k) + 1)$ into $I[p]$. This means that during SP_p^k at least one interval was finished, or that there are an infinite number of intervals in α_0. ■

We prove the following property of α_0 by induction:

Claim 3.10 *During the l'th interval of α_0 there are at least $l + 1$ pending invocations that are never responded to in α_0.*

Proof: **l=0** From I4 there must be some execution,SP, that wrote into $Y[1 \bmod 3N]$ (and therefore also to $X[1 \bmod 3N]$) and $readI(SP) = 0$. Let SP_p be the last such execution to write into $X[1 \bmod 3N]$.

1. Since SP_p is a DoS-execution it must have reached Line 11.

2. All the executions in α_0 are DoS-executions. Therefore, if any execution SP_q wrote into $X[1 \bmod 3N]$ after SP_p then $readI(SP_q)$ is equal to $3N$ or $6N$ or $9N$ and so on. This means that, from I1b, the value SP_p wrote into $X[1 \bmod 3N]$ does not change until the end of its execution. But then the ifstatements in Lines 13 and 14 must fail and SP_p must return `stop`. Because α_0 does not contain any `stop` responses, SP_p does not end during α_0 and the claim holds.

l=k Assume by induction that the claim is correct for $l = k - 1$. We prove the claim for interval $l = k$. From the induction, there were at least k active processes during interval $k - 1$ that do not respond during α_0. Because interval k is still in α_0 those processes continue to be active. We show that there is at least one more such process.

From I4 there must be some execution,SP, that wrote into $Y[k + 1 \bmod 3N]$ (and therefore also to $X[k + 1 \bmod 3N]$) and $readI(SP) = k$. Let SP_p be the last such execution to write into $X[k + 1 \bmod 3N]$. Note that p must be different than all the k active processes of the induction because $readI(SP_p) = k$.

1. Since SP_p is a DoS-execution it must have reached Line 11.
2. All the executions in α_0 are DoS-executions. Therefore, if any execution SP_q wrote into $X[k + 1 \bmod 3N]$ after SP_p then $readI(SP_q) = k + 3N$ or $k + 6N$ or $k + 9N$ and so on. This means that ,From I1b, the value SP_p wrote into $X[k+1 \bmod 3N]$ does not change until the end of its execution. But then the ifstatements in Lines 13 and 14 must fail and SP_p must return `stop`. Because α_0 does not contain any `stop` responses, SP_p does not end during α_0 and the claim continues to hold.

Because there is an infinite number of intervals in α_0 it follows from the claim that an infinite number of processes are needed for α_0 and the assumption to the contradiction is wrong. ■

From lemmas 3.4,3.7,3.8,3.9 and corollary 3.6 follows the correctness of the algorithm.

Theorem 3.11 *The code in Algorithms 1 and 2 correctly implement an adaptive-splitter as specified in Section 2.1.*

3.3 Step Complexity

The proof of the algorithm complexity is based on the following lemma:

Lemma 3.12 *For every process p and every process r during one execution of procedure* updatel() *process p reads the id of process r into variable q in Line 24 at most three times.*

Proof: Let $e \in E_l, e \in SP_p$ the next step of p and assume that p reads the name of process q in Y three times. From I3 there are $l', l'', l''' \leq l$ such that $index(readI(SP_p)) \leq index(l') < index(l'') < index(l''') \leq index(l)$ and q wrote into $Y[l' \bmod 3N], Y[l'' \bmod 3N], Y[l''' \bmod 3N]$. Since, from I3, $readI(SP_p) \leq l', l'', l'''$ it follows that $readI(SP_p) \leq l' < l'' < l''' \leq l$.

Let SP_q be the execution in which q writes in $Y[l'' \bmod 3N]$.

The read event of $I[q]$ in SP_p that happened after the read of $Y[l' \bmod 3N]$ must have happened before the write in SP_q of $index(l'')$ in Line 12 for otherwise the value in $I[p]$ would have been either $index(l'')$ or $index(l''+1)$ and in the next iterations of updatel() $Y[l'' \bmod 3N]$ would not have been read.

The read event in SP_p of $Y[l''' \bmod 3N]$ in line 24 must have happened after the write to $Y[l''' \bmod 3N]$ by q. That is, after SP_q finished.

It follows that SP_q wrote into $Master$ after SP_p which means that the next time after the third read of q in SP_p that Line 20 is executed SP_p ends. ■

It is immediate that the step complexity of the release() operation is constant.

Theorem 3.13 *The step complexity of the* acquire() *operation is* $O(k_i)$ *where* k_i *is the number of processes that access the adaptive-splitter concurrently with the execution of function* acquire().

Proof: The number of operations made by any execution SP_p of any process p is constant outside the loop in updatel(). From I3 it follows that if during the loop in SP_p a name of another process, q, is read from Y then $readI(SP_p) \leq readI(SP_q)$ which means q executed concurrently with p. From lemma 3.12 SP_p can read the same name, q, at most three times and therefore the step execution of the algorithm is $O(k_i)$. ■

While the worst case step complexity of the acquire() operation is $O(k_i)$, most of the time the execution takes a constant number of steps. This is because typically, if process p writes into $Y[l]$ the other processes do not increment $Master$ until p finishes. The $Master$ register can be incremented if all the writes of p in copy l were overwritten by other processes. However, this is rare.

Chapter 4

Renaming

Given an implementation of a long lived and adaptive splitter it is possible to construct long lived and adaptive renaming (for a description of the Renaming problem see the Introduction and Section 2.2). By arranging the splitters once in a row and once in a grid we construct two different renaming algorithms, each with its own advantages.

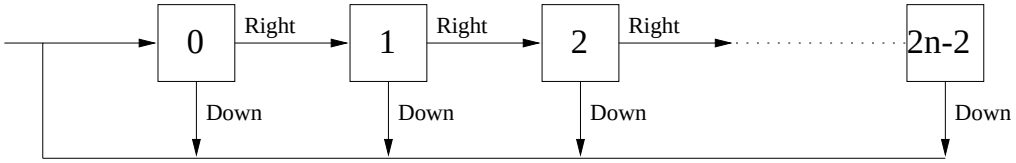
4.1 Non Blocking Optimal Name Space $(2\hat{k}_i - 1)$-Renaming

A non-blocking and name space adaptive $(2\hat{k}_i - 1)$-renaming algorithm is constructed by connecting together $2n - 1$ adaptive-splitters in one row (see Algorithm 3). The algorithm is very simple: any process that goes `down` from an adaptive-splitter starts again at the beginning of the row.

The algorithm is not wait-free. For example, a process may infinitely try to capture the first splitter in the row and continuously fail with a `down` response. This can happen because of new renaming requests that keep capturing and releasing the splitter. However, this algorithm achieves optimal name space [HS93].

Notice that placing the Moir-Anderson splitters instead of the adaptive-splitter could result in a deadlock (infinite loops). This is because in the Moir-Anderson splitter two processes can repeatedly get `down` responses by interfering with each other again and again (the Moir-Anderson splitter does not satisfy Property 5 of our adaptive splitter). Property 5 of the adaptive-splitter asserts that there is no infinite sequence of events where the events are only acquire events and `down` responses. However, it is possible that there is an infinite sequence of events where the events are only acquire events and `right` responses. This is why $2n - 1$ splitters are necessary in this algorithm and $4n$ splitters are necessary in the next algorithm.

Algorithm 3 Code for $2n-1$ RENAMING.



Shared:
$Splitters[0, \ldots, 2n-2]$, an array of adaptive-splitters;
Private and global:
i integer;

procedure acquire-name()
1: $move := -$; // not `stop`
2: $i := 0$;
while $move \neq$ `stop` do
3: $move := Splitters[i]$.acquire();
4: if $move =$ `down` then $i := 0$;
5: if $move =$ `right` then $i := i + 1$;
od;
return$(i+1)$; // Map to names $1, \ldots, 2n-1$.

Procedure release-name()
$Splitters[i].release()$;

4.1.1 Proof of the Algorithm

In this proof we consider a *run*, as a possibly infinite sequence of the following events: `acquire-invocation`^{p}, `release-invocation`^{p}, `right-response`^{p}, `down-response`^{p}, `stop-response`^{p} and `done-response`^{p} where event e^p is associated with process p for each p in the set of possible processes. Furthermore, we consider only legal runs satisfying the following:

1. No process may have two pending invocations. That is, if process p issues an invocation to splitter i it waits for a response from that splitter before issuing any more invocations.

2. Every splitter responds exactly once to each invocation - except perhaps the last one.

3. To an `acquire-invocation`^{p} a splitter responds with one of `right-response`^{p}, `down-response`^{p} or `stop-response`^{p}.

4. To a `release-invocation`^{p} a splitter responds only with a `done-response`^{p}.

Let $\alpha = e_0, \ldots, e_i, e_{i+1}, \ldots$ be a run. The prefix α_i of run α is $(e_0, \ldots, e_i)$, the sequence of the first $i+1$ events of α.

For the proof we assume that the array *Splitters* is unbounded. We then prove that no high level operation *op* of the renaming algorithm accesses splitters at index greater than $2\hat{k}_i$ (recall that $\hat{k}_i$ is the content interval contention during the operation's interval).

Definition 4.1 *A splitter is in a* fulfilled state *during a busy period from the first response either of type* `down-response` *or of type* `done-response`*, to the end of the busy period.*

Definition 4.2 *Process p is said to be at $Splitters[j]$ at the end of prefix run α_i in any of the following cases:*

1. *Process p has a pending invocation on $Splitters[j]$.*

2. *Process p has captured $Splitters[j]$ and has not yet released it.*

3. *$j > 0$ and the last event in α_i related to p is a* `right-response` *on $Splitters[j-1]$.*

4. *$j = 0$ and the last event in α_i related to p is a* `down-response`*.*

Lemma 4.1 *Assume that process p is at $Splitters[j]$, performing high level operation op and that at most $k-1$ processes are running concurrently with p's operation or hold names (i.e., $\hat{k}_i = k$). Then for every $j' \leq j$ the number of processes at splitters $j', \ldots, \infty$ + the number of fulfilled splitters at splitters $j', \ldots, \infty$ during op is at most $2k - j' - 1$.*

Proof: We prove the lemma by induction on the length of the prefix run α_i. Assume that the lemma is correct for prefix run α_i. We show correctness for $\alpha_{i+1} = \alpha_i e_{i+1}$. Without loss of generality, assume that event e_{i+1} was related to process q (q may be equal to p).

Obviously, only events that change the fulfillment state of splitters or the place of processes effect the correctness of the lemma. We consider all these cases:

(base) e_{i+1} is the first operation of p in α: This means p is at $Splitters[0]$. From the definition of $hatk_i$, there are at most $k-1$ other processes accessing splitters at α_i. By Definition 4.1 there is at least one process in any fulfilled splitter. Therefore, at α_i there are at most $k-1$ processes and $k-1$ fulfilled splitters in the system so, at α_{i+1}, at most $2k-1$ processes and fulfilled splitters.

Process q is at splitter 0 at α_{i+1}: If e_{i+1} is the first event of q in α then it follows that at α_i there are at most $k-1$ other processes accessing splitters (note that op, the operation of p, starts before e_{i+1} and ends after it). Following the same arguments as the previous case it follows that at α_{i+1} there are at most $2k-1$ processes and fulfilled splitters.

If e_{i+1} is not the first event of process q then by the induction at α_i there are at most $2k-1$ processes and fulfilled splitters. Event e_{i+1} must have been a `down` response related to process q at some splitter, $Splitters[j]$. For every $j' \leq i$ the number of processes at splitters $j', \ldots, \infty$ could at most be reduced. The only scenario to check is if $Splitters[j]$ was not fulfilled at α_i and is fulfilled at α_{i+1}. However, this means that some process other than q is also at $Splitters[j]$ at α_{i+1}. Thus, at α_i, there must have been at most $k-2$ fulfilled splitters and the claim continues to hold.

Splitter j becomes fulfilled: If $Splitters[j]$ was not fulfilled at α_i and is fulfilled at α_{i+1} then event e_{i+1} is a `down` response. But this means process q is now at $Splitters[0]$ and hence this case is covered by the previous cases.

Process q moves from splitter $l-1$ to splitter l: If $l > i$ the claim is not affected. Therefore assume $l \leq i$. By the induction there were at most $2k-l$ processes and fulfilled splitters with indices $l-1, \ldots, \infty$ at α_i. If at α_{i+1} there is some process still at $Splitters[l-1]$ then at α_i there were at most $2k-l-2$ processes and fulfilled splitters at $l, \ldots, \infty$ and the claim holds. If at α_{i+1} there is no process at $Splitters[l-1]$ then from Property 3 of an adaptive splitter, $Splitters[l-1]$ must have been in a fulfilled state at α_i which means at α_i there were at most $2k-l-2$ processes and fulfilled splitters at $l, \ldots, \infty$ and again the claim holds.

Process q releases splitter j: From the induction the claim was correct at α_i. At α_{i+1} there are less processes accessing splitters and no more fulfilled splitters. Therefore claim continues to hold.

■

From the Lemma it follows that even if all the processes concurrently try to get a name, none ever accesses splitters with index greater than or equal to $2n-1$. Correctness of the algorithm immediately follows because at most one process can acquire a splitter. Thus, assigned names are unique.

Corollary 4.2 *No process accesses an adaptive-splitter with index l, $l > 2n-2$.*

Corollary 4.3 *The name assigned to process p during its renaming operation, op, is at most $2\hat{k}_i - 1$.*

Claim 4.4 *The renaming algorithm in Algorithm 3 is non-blocking.*

Proof: Assume to the contradiction that there is an infinite sequence in some legal run α in which no process returns `stop`. From Corollary 4.2 it follows that there exists at least one splitter that is accessed infinitely many times. Let $splitters[j]$ be such a splitter with the largest index j.

1. If there is an infinite number of busy periods of $Splitters[j]$ in α then from Property 4 of an adaptive splitter and because no process returns `stop` it follows that an infinite number of processes returned `right` when accessing $Splitters[j]$. But this means an infinite number of processes accessed $Splitters[j+1]$ (Line 5) which contradicts our assumption about $Splitters[j]$.

2. If α contains an infinite busy period of $Splitters[j]$ then from Property 5 and because no process returns `stop` if follows that an infinite number of processes returned `right` when accessing $Splitters[j]$. Again this contradicts our assumption about $Splitters[j]$.

This means such an infinite run does not exist, hence the algorithm is non-blocking. ■

4.2 Wait Free $(4\hat{k}_i^2)$-Renaming Algorithm

We use the adaptive splitter to build a renaming algorithm that is based on a $2n \times 2n$ grid of adaptive-splitters. Following [MA95], each process in the renaming algorithm has to capture one splitter in the grid and rename itself to the unique name associated with the splitter it has captured. To capture a splitter a process starts acquiring splitters

from the top left most splitter in the grid, going down or right according to the response it gets, until it receives a `stop` response at which point it has captured a unique name (see Algorithm 4). We employ the Attiya and Fouren's adaptation [AF98] of Moir and Anderson's grid algorithm to ensure that processes rename into a name space that is a function of the contention, i.e, adaptive name space.

We associate names with splitters on the grid according to the following scheme:

```
1   2   5  10   17   26   .
4   3   6  11   18   27   .
9   8   7  12   19   28   .
16  15  14 13   20   .    .
25  24  23 22   21   .    .
.   .   .  .    .    .    .
```

4.2.1 Proof of the Algorithm

For the proof we assume that the grid is unbounded, i.e., having an infinite number of rows and columns. We then prove that no operation *op* accesses an adaptive-splitter at a row or column greater than $2\hat{k}$.

Definition 4.3 *A row (column) of splitters in the grid is* idle *if and only if all the adaptive-splitters in the row (column) are idle, and there are no processes in transition between two adaptive-splitters in the row (column).*

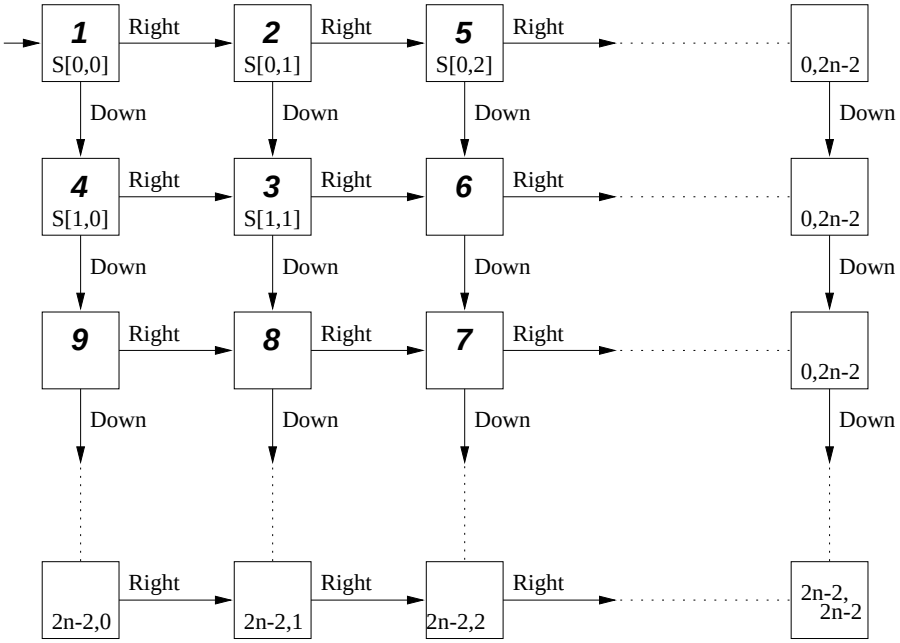
Definition 4.4 *A* busy period *of a row (column) of splitters in the grid is the subsequence of events between two consecutive idle states of the row (column).*

Lemma 4.5 *In each finite busy period of a row (column) there is at least one* `stop` *response in one of the row's (column's) adaptive-splitters.*

Proof: Assume to the contrary that there is a busy period of the row (column) that does not satisfy the lemma. Let s be the splitter most to the right (lowest) that was accessed during that busy period (since the busy period is finite such a splitter exists). By the assumption no process captured that splitter during the busy period. Therefore, by the definition of adaptive-splitter at least one of the processes should have gone to the `right` (`down`) - a contradiction to the fact that s is the most to the right (lowest) splitter. ■

Definition 4.5 *A row (column) is said to be in a* fulfilled *state during a busy period from the first* done *response (to a release invocation that must exists by Lemma 4.5) to the end of the busy period of that row (column).*

Algorithm 4 Code for RENAMING. (The structure of procedure **acquire-name** was borrowed from Attiya and Fouren.



Shared:

$Splitters[0, \ldots, 2n-2 : 0, \ldots, 2n-2]$, a two dimensional grid of adaptive-splitters.

Private and Global:

1: i, j integers;

procedure acquire-name()

2: $move := -$; // not `stop`

3: $i := j := 0$;

4: while $move \neq$ `stop` do

5: $move := Splitters[i, j].acquire()$;

6: if $move =$`down` then $i := i + 1$;

7: if $move =$`right` then $j := j + 1$;

8: od;

9: return $((max(i, j))^2 + i - j + max(i, j) + 1)$;// An encoding of the Splitters enumeration.

procedure release-name()

$Splitters[i, j].release()$;

Definition 4.6 *Process p is said to be* in *row (column) j in either one of the following three cases: (1) Process p has a pending invocation on some adaptive-splitter in row (column) j; (2) There is an adaptive-splitter in row (column) j that is captured by process p; and (3) The last event related to p is a* `down` (`right`) *response on a splitter in row (column) $j-1$.*

Lemma 4.6 *Assume that process p is at row (column) j performing operation op and that at most $k-1$ processes are running concurrently with p's operation or hold names (i.e., $\hat{k}_i = k$). Then for every $j' \leq j$ the number of processes and fulfilled rows (columns) in rows (columns) $j', \ldots, \infty$ is at most $2k - j' - 1$.*

The proof repeats the proof of Lemma 4.1. Therefore, we only provide a sketch.

Sketch of proof: By induction on the length of the execution. We need to consider the following cases:

(base) Process p enters the grid ($j = 0$): Just before process p enters the grid there are at most $k-1$ other processes in the grid. By definition 4.5 there is at least one process in any fulfilled row (column). Therefore, there are at most $k-1$ fulfilled rows (columns) and $k-1$ processes in the grid and together with process p there are $2k-1$ processes and fulfilled rows (columns).

A new process other than p enters the grid while p is at row (column) j ($j \geq 0$). For $j' = 0$ it follows by the same arguments as in the previous case. For $j' > 0$ it does not effect the inductive claim.

A row (column) becomes fulfilled. According to the definition this may happen only when a process that captured a splitter leaves the grid. In that case there is one more fulfilled row (column) but the number of processes has just decreased by one.

A process q moves from row (column) $l-1$ to row (column) l. If $l > j$ the claim is not affected. Therefore assume $l \leq j$. By the inductive hypothesis before q (which might be p) left row (column) $l-1$, there were at most $2k-l$ processes and fulfilled rows (columns) in rows (columns) $l-1, \ldots, \infty$. If process q is not the last one to leave $l-1$, then before q left $l-1$ there were at most $2k-l-2$ processes and fulfilled rows (columns) in rows (columns) $l, \ldots, \infty$ and the claim holds. If q is the last process to leave then before q left row (column) $l-1$, according to Lemma 4.5 row (column) $l-1$ was in a fulfilled state. In that case before q left there were at most $2k-l-2$ processes and fulfilled rows (columns) in rows (columns) $l, \ldots, \infty$, thus claim holds.

■

From the Lemma it follows that even if all the processes concurrently try to get a name, none ever accesses rows and columns with index greater or equal to $2n-1$.

Corollary 4.7 *No process accesses an adaptive-splitter in row (column) l, $l > 2n-2$.*

Since processes access adaptive-splitters in strictly monotonic increasing rows and columns (Lines 6,7) it follows that:

Corollary 4.8 *A process may access at most $4\hat{k}_i - 2$ adaptive-splitters.*

From Corollary 4.8 it follows that the name that may be assigned by an operation *op* is at most $4\hat{k}_i^2$.

Chapter 5

n-Process Mutual Exclusion

We use the adaptive splitter to transform the a mutual exclusion tournament tree implementation into an adaptive algorithm. For a description of the mutual exclusion problem see the Introduction and Section 2.4.

5.1 Description of the Algorithm

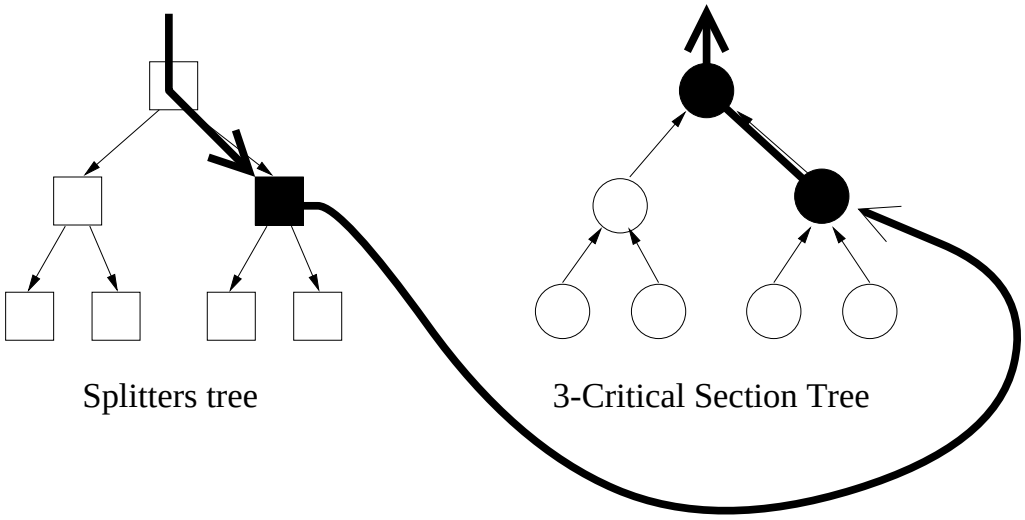
To implement the adaptive mutual exclusion algorithm among n processes we use two shared binary trees of depth $2n-2$ (the depth of a tree is the maximum number of edges between any leaf and the root). The first, tournament tree T, contains in each node a 3-process fair mutual-exclusion object (bakery for example). The second tree, splitters tree $Splitters$, contains in each node a long lived adaptive splitter.

Every process starts the algorithm by going down the splitters tree, trying to acquire splitters one after the other until it succeeds. If the acquire() operation fails in node $Splitters[i]$ and returns `down` the process continues to the left child of $Splitters[i]$. If the acquire() operation returns `right` the process continues to the right child of $Splitters[i]$ (see Algorithm 5).

Once a process succeeds in acquiring a node (say $Splitters[i]$) it starts climbing up the tournament tree from node $T[i]$. In each node including $T[i]$ the process participates in the 3-process mutual exclusion until it wins. Once the process enters the 3-process Critical Section in node $T[i]$ it moves up to $T[i]$'s parent, and so on until the root. When the process enters the 3-process Critical Section in $T[1]$ it also enters the n-process Critical Section.

Going out of the n-process Critical Section is simply going out of all the 3-process Critical Sections in the path from the root to $T[i]$ and then releasing the splitter in $Splitters[i]$.

Algorithm 5 Code for CRITICAL SECTION.



Shared:
$Splitters[1, \ldots, 2^{2n-1} - 1]$, an array of adaptive-splitters. // if n is not known then use N
$T[1 \ldots, 2^{2n-1} - 1]$, an array of 3-process fair mutual-exclusion objects.

Entry:
1: $move := -$;
2: $i := 1$;
3: while $move \neq$ `stop` do // Traverse down the splitters tree
4: $move := Splitters[i]$.acquire();
5: if $move =$ `down` then $i := i * 2$; // Move to left child.
6: if $move =$ `right` then $i := i * 2 + 1$; // Move to right child.
od;
// Since the mutual-exclusion object in $T[i]$ supports at most three processes,
// every process needs to choose a number from $0, 1, 2$ to act as its id. The correctness proof
// shows that no two processes ever access the same node with the same id at the same time.
7: $T[i]$.enter(2);
8: $last := i$;
9: for $(j := \lfloor i/2 \rfloor; j > 0; j := \lfloor j/2 \rfloor)$ do // Traverse up the tournament tree
10: $T[j]$.enter($last$ mod 2); // If we come from left child execute enter(0).
// If from right execute enter(1).
11: $last := j$;
od;
Critical Section:

Exit:
for all $T[j]$ on the path from $T[1]$ to $T[i]$ do
12: $T[j]$.exit();
od;
13: $Splitters[i]$.release();

5.2 Proof of Correctness

First we prove some properties of the splitters tree *Splitters* and then we use these properties to prove correctness and adaptivity of the algorithm.

Definition 5.1 *A* path *in the splitters tree Splitters is an ordered set of splitters that contain the root, exactly one child of the root and exactly one child of every splitter in the path. Splitter Splitters[i] is said to be the j'th splitter of path pt if it is at depth j in the tree.*

In the following proofs we consider the behavior of the system on a single path *pt*. We later use this behavior to prove properties of the entire splitters tree.

For the proof we first assume that *pt* is unbounded (this means that the array *Splitters* is unbounded). We then prove that no high level operation *op* of the mutual-exclusion algorithm accesses splitters that are more than $2k - 2$ deep.

Definition 5.2 *Process p is said to be at Splitters[i] in either one of the following cases:*

1. *Process p has a pending invocation on Splitters[i].*
2. *Process p captured Splitters[i].*
3. *The last event related to p is a* `right` *response on the parent of Splitters[i] and Splitters[i] is the right child.*
4. *The last event related to p is a* `down` *response on the parent of Splitters[i] and Splitters[i] is the left child.*

Definition 5.3 *A splitter Splitters[i] in path pt is in a* fulfilled state *with respect to pt during Splitters[i]'s busy period either:*

1. *From the first* `done` *response to a* `release()` *invocation by Splitters[i] in the busy period, until the end of the busy period.*
2. *From the first* `right` *or* `down` *response by Splitters[i] that moves a process to a splitter outside path pt until the end of the busy period.*

Lemma 5.1 *For path pt, assume that process p is at Splitters[i], the j'th splitter in the path, performing high level operation op. Then for every $j' \leq j$ the number of processes at splitters in depth $j', \ldots, \infty$ plus fulfilled splitters at splitters $j', \ldots, \infty$ in pt during op is at most $2k - j' - 1$.*

Again, the proof follows the proof of Lemma 4.1 so we only provide a sketch of the proof.

Sketch of proof: We consider all the cases where events change the fulfillment state of splitters or the place of processes in the tree.

(base) Process p enters the tree: This means p is at $Splitters[0]$. From the definition of k, there are at most $k-1$ other processes accessing splitters. By Definition 5.3 there is at least one process in any fulfilled splitter. Therefore, there are at most $k-1$ processes and $k-1$ fulfilled splitters in the system before p joined in so after p joined the system there are at most $2k-1$ processes and fulfilled splitters.

A new process other than p enters the tree while p is at the j ($j > 0$) splitter: For $j' = 0$ it follows by the same arguments as in the previous case. For $j' > 0$ it does not effect the inductive claim.

Splitter i becomes fulfilled: According to the definition when this happens either the process leaves the path or the entire tree. In any case, there is one more fulfilled splitter but the number of processes on path pt has just dropped by one.

Process q moves from the $l-1$'th splitter in the path to the l'th splitter: If $l > j$ the claim is not affected. Therefore assume $l \leq j$. By the inductive hypothesis before q (which might be p) left splitter $l-1$ in the path, there were at most $2k-l$ processes and fulfilled splitters at $l-1, \ldots, \infty$ in the path. If process q is not the last one to leave $l-1$, then before q left splitter $l-1$ there were at most $2k-l-2$ processes at $l, \ldots, \infty$ in the path and the claim holds. If q is the last process to leave then before q left the l'th splitter, according to Definition 5.3 and Property 3 of an adaptive splitter the $l-1$'th splitter was in a fulfilled state. In that case before q left there were at most $2k-l-2$ fulfilled splitters and processes at $l, \ldots, \infty$, and the claim follows.

Process q releases splitter i: From the induction, the claim was correct before q releases the splitter. Since after the splitter is released there are less processes accessing splitters and no more fulfilled splitters the claim continues to hold.

■

From the lemma it follows that even if all the processes try to enter the mutual-exclusion concurrently, no process ever accesses a splitter at depth greater or equal $2n-2$ on any path. This means no process ever accesses splitters that are more than $2n-2$ deep in the splitters tree and, since the number of elements in a binary tree of depth d is $2^{d+1}-1$, the size of the $Splitters$ array is $2^{2n-1}-1$.

Corollary 5.2 *At most one process captures any splitter in a subtree rooted at a splitter that is $2n-2$ deep in the splitters tree.*

We now use the properties of the splitters tree to prove correctness of the algorithm.

Lemma 5.3 *There can be at most three pending* enter() *invocations that have not been responded to at any node, $T[i]$, in the tournament tree. Furthermore, at most one of them is* enter(0)*, one is* enter(1) *and one is* enter(2)*.*

Proof: We proof the lemma by induction on k,the depth of node $T[i]$, from $k = 2n - 1$ to $k = 1$.

The case in which $k > 2n - 2$: From the code it is easy to see that only processes that have captured a splitter in the splitters sub-tree rooted at $Splitters[j]$ ever accesses $T[j]$, for all j. But from Lemma 5.1 it follows that no process ever accesses nodes of such depth in the splitters tree from which follows that no process accesses such nodes in the tournament tree.

(base) $k = 2n - 2$: From corollary 5.2 there is at most one process that accessed $Splitters[i]$ meaning there is at most one process that may have invoked enter(2) on $T[i]$. Since $Splitters[i]$ has no children, neither has $T[i]$ and therefore no other process may have access $T[i]$ at the same time.

$k = k - 1$: Assume that the claim is correct for all nodes at depth larger or equal to k, we show correctness for node $T[i]$ at depth $k - 1$.

1. From the algorithm it follows that only a process that captured $Splitters[i]$ may invoke $T[i]$. enter(2). There can be at most one such process at any point in time.
2. By the inductive hypothesis, all the processes accessing the left child of $T[i]$ do so in a way consistent with the specification of the 3-process mutual-exclusion object installed there. Therefore, by the properties of the 3-process critical section at the left child, at most one process at a time moves up to $T[i]$ and invokes its mutual-exclusion. Furthermore, by Line 10, it invokes $T[i]$. enter(0).
3. By the inductive hypothesis, all the processes accessing the right child of $T[i]$ do so in a way consistent with the specification of the 3-process mutual-exclusion object installed there. Therefore, by the properties of the 3-process critical section at the right child, at most one process at a time moves up to $T[i]$ and invokes its mutual-exclusion. Furthermore, by Line 10, it invokes $T[i]$. enter(1).

Therefore the lemma continues to hold.

■

From the lemma it follows that at most one process at a time can enter the mutual-exclusion at $T[1]$ hence the algorithm is correct.

Theorem 5.4 *The code in Algorithm 5 implements an n-process mutual exclusion algorithm.*

5.3 Complexity Analysis

As in [CS93] we assume that at most k_t processes participate in the algorithm (k_t is then the total contention during the execution). We show that in our implementations the worst case no. of operations a process may take until it enters the critical section is $O(k_t^2)$. Likewise, we show that the system response time is $O(k_t^2)$ as well.

Lemma 5.5 *In a 3-process bakery algorithm every process takes a constant number of step + up to four different busy wait loops before entering the critical section.*

Lemma 5.6 *The system response time and the worst case number of operations of the 3-process bakery algorithm are constant.*

Lemma 5.7 *If process p accesses the 3-process mutual exclusion object $T[i]$ at depth l from the root then after at most $O(l)$ steps and busy-waits process p enters the critical section.*

Proof: Since $T[i]$ is at depth l, process p must win in l nodes of T, one after the other, before entering the critical section. From Lemma 5.6 it follows that the number of steps and busy-waits process p may take in each node is constant which means that the number of steps p may take until reaching the root is $O(l)$. ■

Theorem 5.8 *The worst case no. of operations a process may take until it enters the critical section is $O(k_t^2)$.*

Proof: From Lemma 5.1 it follows that the process that goes down the splitters tree to the most depth reaches depth of at most $2k_t - 1$. This is also the process that takes the most steps in the splitters tree and in the tournament tree. Because every splitter is adaptive, the process may take at most $O(k_t^2)$ steps in each splitter (see Section 3.3). It therefore follows that the worst case no. of operations that a process may take is $O(k_t^2)$ steps in the splitters tree and $O(k_t)$ steps and busy-waits in the tournament tree, meaning the algorithm is adaptive. ■

Lemma 5.9 *Given that some process accesses node $T[i]$ at depth l in the tournament tree, the worst time it may take until some process enters the critical section is $O(l)$.*

Proof: We prove the lemma by induction on l.

l=0: If there is some process that accesses $T[1]$ then, by Lemma 5.6 , after a constant time some process wins the 3-process mutual exclusion and, since this is the root, enters the critical section.

l=l+1: Assume that the lemma is correct for l, we prove it is correct for $l+1$. If a process accesses node $T[i]$ at depth $l+1$ then by Lemma 5.6 some process p wins $T[i]$ after a constant time and then, by the algorithm, process p accesses the parent of $T[i]$ at depth l. By the induction, once p accesses the parent of $T[i]$ it takes at most $O(l)$ time until some process enters the critical section. Therefore, it takes at most $O(l+1)$ time from the time a process accessed $T[i]$ until some process entered the critical section and the claim holds.

■

From the lemma it follows that the time processes may spend in the tournament tree until some process enters the critical section is at most linear in k_t. We now show that the time processes may spend in the splitters tree is at most quadratic in k_t.

Lemma 5.10 *If k' processes concurrently access the same splitter then the maximum time it may take from a time some process tries to capture the splitter until all processes return a response (either* `stop`*,* `down` *or* `right`*) is ck'^2.*

Proof: From the adaptiveness of a splitter it follows that every process can take at most k' steps in a splitter before returning a response. Even if all processes access the same registers concurrently then the slowest process may take ck'^2 time, for some constant c.
■

Lemma 5.11 *If k' processes access a subtree, sub, of the splitters tree then the longest time it may take until a splitter is captured is $f(k')$ where*

$$f(k') = \begin{cases} k' = 1 & c \\ k' > 1 & ck'^2 + f(\lceil \frac{k'}{2} \rceil) \end{cases}$$

Proof: We assume there is no subtree of *sub* that contains all the splitters that processes access. If such a subtree exists simply exchange *sub* with its subtree.

We prove the lemma by induction on k'.

$k' = 1$: If at most one process accesses *sub* then, by the definition of an adaptive splitter, it must stop at the root of *sub*, and by Lemma 5.10, it must stop after at most c operations.

$k' = k' + 1$: Assume that the lemma holds for k'. We show it holds for $k' + 1$. If there is some subtree in *sub* that contains splitters that only part of the processes access, then by the induction after at most $f(k') < f(k' + 1)$ time some process in the subtree, and therefore in *sub*, captures a splitter. Therefore, we assume all the $k' + 1$ processes access the root of *sub* together.

1. From Lemma 5.10 it follows that after at most $c(k' + 1)^2$ time all processes responded with one of `stop`, `right` or `down`.
   (a) If some process decides to `stop` then, as $c(k' + 1)^2 < f(k' + 1)$, the claim holds.
   (b) If no process stopped then at least one returned `right` and one `down`. The time it would take for a splitter to be captured is now the minimum between the time it takes at the left subtree of *sub* and the right subtree of *sub*. In the worst case, they are the same, when half the processes go `down` and half go `right`. From this is follows that the time it takes for a splitter in *sub* to be captured is bounded by $c(k'+1)^2 + f(\lceil \frac{k'+1}{2} \rceil) = f(k'+1)$ and the claim holds.

■

Given the fact that $\lceil \frac{k'}{2} \rceil \leq \frac{k'}{1.5}$ we can bound $f(k')$ by a geometric series.

Corollary 5.12 *$f(k')$ is bounded by $O(k'^2)$.*

Theorem 5.13 *The system response time of the n process mutual exclusion algorithm is $O(k_t^2)$.*

Proof: From Lemma 5.11 if follows that at most $O(k_t^2)$ time can pass from the time any process enters the Entry section of the n-process mutual exclusion until some process captures a splitter. From Lemma 5.10 if follows that at most $O(k_t)$ time can pass from the time a splitter is captured until some process enters the Critical section. Therefore, the system response time of the algorithm is $O(k_t^2)$. ■

5.4 Complexity Improvements

It is possible to improve the worst case number of operations per process with a simple transformation of the original algorithm.

First, we map the N nodes of T at depth $\log(N)$ to the processes' names (w.l.o.g. we assume that $\log(N)$ is an integer). During the Entry Section every process accesses at most $\log(N) - 1$ splitters. If after $\log(N) - 1$ tries the process still did not capture a splitter it *abandons* the splitters tree without capturing a splitter and starts participating

in the tournament tree from its corresponding node at depth $\log(N)$. As before, it enters all the mutual exclusion objects from its starting node to the root before it enters the n-process Critical Section. The Exit Section is exactly the same as in the original algorithm, except of course that there is no splitter to release.

Since no process goes down the splitters tree to depth that is more than $\log(N) - 1$ and because there is a one to one mapping from the processes names to the nodes in T, at most one process at a time accesses a node of the tournament tree at depth $\log(N)$ so Corollary 5.2 can be exchanged with the following:

Corollary 5.14 *At most one process captures any splitter in a subtree rooted at a splitter that is* $\log(N)$ *deep in the splitters tree.*

From this follows that the correctness lemma (Lemma 5.3) continues to hold.

The system response time is still $O(k_t^2)$. However, following the proof of Theorem 5.8 it is easy to see that, as the deepest node that can now be reached is $\log(N)$ deep, the worst case number of operations per process is $O(\min(k_t^2, k\log(n)))$.

Part II

Adapting to Point Contention

Chapter 6

Renaming

In this chapter we present a bounded space, adaptive to point contention renaming. The algorithm uses bounded memory but unbounded sequence numbers. It is content adaptive to point contention.

6.1 Bounded Space Renaming with Unbounded Sequence numbers

The algorithm of [AF99a, AAF+99b] uses a sequence of *sieves*, numbered $1, 2, \ldots 2N$. The name obtained from a sieve is a pair containing the sieve number and the rank of the process among the processes possibly getting a name from the sieve. A process tries to get a name in the sieves, one after the other, until it succeeds in some sieve; the implementation of a sieve guarantees that each sieve "catches" at least one of the processes that access it. In the implementation of [AAF+99b] each sieve has an infinite number of copies, resulting in an unbounded number of shared variables. In this section, we show how to bound the number of copies used in each sieve; the idea is inspired by the splitter algorithm in Chapter 3.

In the original algorithm, names are obtained only from the last copy of a sieve. Thus, the natural idea would be to "recycle" previous copies. The main obstacle is that "slow" processes can lag behind, working in previous copies of the sieve. These processes can corrupt the previous copies and confuse processes that reuse the copy. We solve this problem with two mechanisms. First, we recycle copies of a sieve only if we are sure that all processes have left these copies. Instead of using an unbounded number of copies, we only take 2N copies, numbered $0, \ldots, 2N-1$ and use them cyclically. With each copy of a sieve we associate a unique process. I.e., copy c is associated with process $p_{c \bmod N}$. Any process other than $p_{c \bmod N}$ accesses copy c only if the process associated with this copy is not active in this sieve. Otherwise, the process skips this sieve. This procedure

guarantees that if the counter of the copies of a particular sieve has reached value t, then no process accesses copy t with counter $t - 2N$.

Now, since copies of the same sieve are only reused once all the processes have left, it is only necessary to reset the copy to its initial value after each iteration. However, clearing all the registers in the copy may take too many steps. Instead, we tag every value written to the copy of the sieve with an unbounded iteration counter. In new iterations we ignore values in the current copy of the sieve that are tagged with old iteration numbers. If such a value is read from a register, we just assume the value is the initial value of the register.

The main pseudocode for the long-lived renaming appears in Algorithms 6 and 7. In procedure getName, a process tries to obtain a name from sieves $1, \ldots 2N$, sequentially, until successful in some sieve. In each sieve, a process first declares itself active and then checks what is the currently active copy of the sieve (which is the iteration number modulo $2N$). Then the process checks whether the owner of this copy is still active in some old copy of the sieve. If this is the case, the process declares it is not active in this sieve and continues to the next sieve; otherwise, the process tries to enter the sieve, by calling sc_sieve with the current iteration number as a parameter. Procedure sc_sieve returns a set of possible winners for the current copy of the sieve (which may be empty). If the process is in this set, then it is a winner in this sieve and it returns a new name according to its rank in the set of possible winners. Otherwise, the process declares it is not active in this sieve and continues to the next sieve.

The pseudocode for accessing a single copy of a sieve appears in Algorithm 7. This code is similar to the original algorithm, except that each value written is tagged, and values tagged with outdated iteration counters are ignored. In the code we denote by $(X)^{=tag}$ a read of shared variable X, which returns v if $X =< v, tag >$, and the initial default value of X otherwise.

A process accessing sieve s tries enter a sieve by checking if it is among the first processes to access the current copy of this sieve. The process succeeds only if this copy is still free (no other process is already inside it), and all processes which could get a name from the previous copy of the sieve have already left it. If a process fails to enter the sieve then there is some other process that is concurrently accessing this sieve. Furthermore, that process is either a winner in this sieve or it was concurrent with some other process that was a winner in the sieve. It can be shown that the number of sieves a process accesses is bounded by a linear function of the number of simultaneously participating processes, i.e., the point contention during its operation interval. This allows us to bound the space of new names and the step complexity. Once inside the sieve, the process computes a set of candidates which are possible winners of this sieve and returns this set.

Algorithm 6 Renaming with unbounded values: Code for renaming by p_i

```
    procedure getName()                                          // get a new name
1:      s := 0;
2:      repeat
3:          s++;
4:          sieve[s].status[i]:=active;
5:          c := sieve[s].count;                                 // locate current copy
6:          nextC:=c+1 mod 2N;
7:          if ( nextC ≠ i and sieve[s].status[nextC mod N] ≠ idle ) then
8:              rank:= 0;
9:              sieve[s].status[i]:=idle;
            else
10:             W:=sc_sieve(sieve[s],nextC,c);
11:             if ( p_i ∈ W ) then                              // p_i is a winner in copy c
12:                 sieve[s].count := c+1;
13:                 rank := rank of p_i in W;
14:             else rank := 0;                  // otherwise, not a winner in this sieve
15:     until (rank ≠ 0)
16:     return( ⟨s, rank⟩);

    procedure releaseName(⟨s, rank⟩)                             // release a name
17:     leave(sieve[s]);
```

Algorithm 7 Accessing a single copy of a sieve: Code for process p_i.

```
    procedure sc_sieve(sieve,c,t)
18:     if ((sieve.allDone[c − 1])^{=t−1} = true) and ( not (sieve.inside[c])^{=t}) then
                        // candidates of previous copy are done and current copy is free
19:         sieve.inside[c] := < true, t >;                          // close the doorway.
20:         V := sieve.latticeAgreement[c]^{=t}(p_i); // single-shot scan of participants [AF98].
                                  // All writes in the lattice agreement are tagged with t
      // If a variable is not tagged with t, then its read returns its initial default value.
21:         sieve.R[c][i] := < V, t >;
22:         W := candidates(sieve,c,t);
23:         if ( p_i ∈ W ) then return W;                            // p_i is a winner in copy c
24:         sieve.done[c][i] := < true, t >;      // otherwise, not a winner in this sieve
25:         W := candidates(sieve,c,t);
26:         leave(sieve[s],c,t);
27:     return ∅;

    procedure candidates(sieve,c,t) // returns the possible winners in a copy of the sieve
28:     V := sieve.R[c][i];                                                    // own view
29:     W := min{(sieve.R[c][j])^{=t}  |  p_j ∈ V and  (sieve.R[c][j])^{=t} ≠ ∅};
                                                                  // minimal by containment
30:     if for every p_j ∈ W, (sieve.R[c][j])^{=t} ⊇ W then return W
31:     else return ∅;

    procedure leave(sieve,c,t)      // leave a sieve, c and W remain from the last enter
32:     sieve.done[c][i] := < true, t >;
33:     if W ≠ ∅ and for every p_j ∈ W, (sieve.done[c][j])^{=t} then
34:         sieve.allDone[c] := < true, t >;     // all candidates of this copy are done
35:     sieve.status[i]:=idle;
```

Correctness Proof

The crucial point in the proof is showing that the **active/idle** flags and the tags guarantee that copies of a sieve are re-cycled correctly. The rest of the proof, showing that the name space and the step complexity of the algorithm adapt to point contention can be proved along the lines of [AF99a, AAF+99b]. However, we present a different proof of correctness for the algorithm; this proof extends our abilities to argue about point contention and it is of interest by itself.

We start with some definitions. Let $R = e_0, e_1, e_2, \ldots$ be an execution of the algorithm, and let $r = e_{i_0}, e_{i_1}, e_{i_2}, \ldots e_{i_l}$ be a finite subsequence of R. Given some event $e_k \in R$, we say that the interval r is *active* at the event e_k if $i_0 \leq k \leq i_l$, i.e., e_k overlaps r.

Let $R^s = e_0, e_1, e_2, \ldots$ be the subsequence of R containing all the events performed by processes accessing sieve s. The *transition sub-sequence* of s, $t_0^s, t_1^s, \ldots$ is a (possibly infinite) subsequence of events from R^s inductively defined as follows:

- $t_0^s = e_0$.
- Assuming t_{l-1}^s exists, then t_l^s is the first event that assigns l to c (in Line 5). If there is no such event, then t_l^s does not exist.

The *l'th interval* of R^s is the subsequence of events in R^s starting at the first event following t_l^s and ending at t_{l+1}^s, if t_{l+1}^s exists; if t_{l+1}^s does not exist, then the l'th interval extends to the end of R^s. We sometimes denote the l'th interval just by l.

Any execution interval of getName by some process p_i, β, can be partitioned into disjoint intervals during which p_i accesses different sieves. If p_i accesses sieve s, then A_β^s is the subsequence of β which starts when p_i sets the variable s to s, and ends when p_i sets the variable s to $s+1$ (or the end of β, if p_i wins sieve s or does not take any further steps). Thus, β can be written as $A_\beta^{s_1}, A_\beta^{s_2}, \ldots$. We call such subsequences *sieve accesses* and sometimes, omit the s superscript and/or the β subscript.

The following terminology is used in the proof.

- A sieve access A *uses* a copy x of the sieve when the process executing A performs Line 19 with $c = x$.
- A sieve access A that uses a copy x of the sieve is *done* at an event $e \in R^s$, if the process executing A performed Line 32 before e.
- A sieve access A executed by process p_i is *winning* if it contains a call to sc_sieve that returns a set that contains p_i (see Line 11).
- Given a sieve access A, let $e \in A$ be the event reading *count* at Line 5. The *interval* of A is the interval in R^s that contains e, denoted $int(A^s)$; the value read by e is denoted $rv(A^s)$.

Lemma 6.1 *For every sieve s the following invariants hold during R^s.*

I1 If some sieve access A^s is active at some event $e \in R^s$, and e belongs to interval l then $int(A) \geq l - N$.

I2 For every sieve access A^s, $rv(A^s) = int(A^s)$.

I3 For every l, if t^s_l exists then at t^s_l, no active sieve access uses copy $(l+1) \bmod 2N$.

I4 For every l, at t_l all the tags kept in the registers are smaller or equal to l.

I5 Assume that A^s_1 and A^s_2 are two sieve accesses that completed candidates *and returned non-empty views, W_1 and W_2, respectively. If $int(A^s_1) = int(A^s_2)$ then $W_1 = W_2$.*

I6 If a sieve access A^s executed by process p is winning, then if A^s_1 is a sieve access that satisfies $int(A^s_1) = int(A^s)$, and A^s_1 get a non-empty view from the invocation of candidates *then p appears in A^s_1's view.*

I7 If a sieve access A^s uses some sieve copy, then any winning sieve access A^s_1 that satisfies $int(A^s_1) = int(A^s) - 1$ is done.

Proof: We show by induction that the invariants hold for every event in R^s.

Proof of **I1**: Assume by way of contradiction that A^s is active at $e \in l$ but $int(A^s) < l - N$ and let p_i be the process executing A^s. In such case there is an interval l' such that $int(A^s) < l' < l$ and $l' \bmod N = i$. Consider A^s_1 the sieve execution that contains $t^s_{l'+1}$. Since A^s is active during all the events in l, A^s_1 cannot be executed by i. Since by the algorithm $rv(A^s_1) = l'$ and since by the induction hypothesis, $int(A^s_1) = rv(A^s_1)$, the process executing A^s_1 must have seen process i active and should have left sieve s without updating c, which is a contradiction.

Proof of **I2**: Assume, by way of contradiction that for some sieve access A^s, $rv(A^s) \neq int(A^s)$ and assume w.l.o.g that $int(A^s) = l$. That means that there is some sieve access A^s_1 that altered the content of c after t^s_l. By the induction hypothesis, $int(A^s_1) = rv(A^s_1)$. Now, if $int(A^s_1) = l$, A^s_1 wrote $l + 1$ in c, and in that case $int(A) > l$, which is a contradiction. Therefore, $int(A^s_1) = l' < l$. Let A^s_2 be the sieve access that executed the event $t^s_{l'+1}$, by the induction hypothesis, when A^s_2 entered the sieve copy all winning sieve accesses of $t_{l'}$, including A^s_1, were done, which is a contradiction

Proof of **I3**: Assume by way of contradiction that at e there is some sieve access A^s_1 that uses copy $(l+1) \bmod 2N$. By the algorithm, $rv(A^s_1) \bmod 2N = l \bmod 2N$. By the induction hypothesis, $int(A^s_1) = rv(A^s_1)$ and therefore, $int(A^s_1) \leq l - 2N$ which contradicts the induction hypothesis on Invariant **I1**.

Proof of **I4**: By the algorithm any sieve access A^s, writes $rv(A^s)$ in the register. The claim holds since $rv(A^s) = int(A^s) < l$, by the induction hypothesis.

Proof of **I5**: Let A_1^s and A_2^s be two sieve accesses that returned from candidates with non-empty views W_1 and W_2, respectively, such that $int(A_1^s) = int(A_2^s)$. The invariant is first violated at e only if it is the latest sieve access; assume by way of contradiction that this is the case and that A_2^s returns W_2 and that $W_1 \neq W_2$. By the induction hypothesis, (1) $rv(A_1^s) = rv(A_2^s) = l$ for some l, (2) at t_l^s there were no sieve access using copy $(l+1) \bmod 2N$ (3) at t_l^s all counter values contained in the registers are less then l and (4) event e is included in at most interval $l + N$. From these properties it follows that for any sieve access A^s using copy $(l+1) \bmod 2N$ between t_l^s and e, $int(A^s) = l$. The rest of the proof is similar to the proof of Lemma 3 in [AAF+99b].

Proof of **I6**: Similar to the proof of Lemma 4 in [AAF+99b], using the induction hypothesis.

Proof of **I7**: Similar to the proof of Lemma 5 in [AAF+99b], using the induction hypothesis. ■

Corollary 6.2 *No two processes hold the same name at the end of a prefix of the execution*

Corollary 6.3 *A process accessing a sieve alone, will get a name.*

Given an execution of the renaming algorithm a sieve s is said to be *idle* if there is no process accessing it. The execution segment between two consecutive idle periods of a sieve s is a *busy period* of s. A crucial point in the proof is that in each busy period there is at least one sieve access which is winning. Only winning sieve accesses update c, and thus, if no sieve access is winning, c remains unchanged during the busy period. Note that some processes accessing s during the busy period evaluates the expression at Line 7 to true and calls sc_sieve; for example, at least process $p_{c \bmod N}$ evaluates the condition to false. Using Lemma 6.1, the next lemma is proved following Lemma 7 of [AAF+99b]:

Lemma 6.4 *During a busy period of a sieve s there is at least one winning sieve access to s.*

Definition 6.1 *A sieve is* utilized *during a busy period, from the first time a process completes the invocation of* releaseName *(which exists, by Lemma 6.4) to the end of the busy period. Intuitively, a utilized sieve is "wasted", since no process holds a name from this sieve, yet no process can enter it.*

Lemma 6.5 *Assume that process p is at sieve j and that the point contention during p's operation is $k-1$, then for every $j' \leq j$, the sum of the number of accessing processes and the number of sieves $j', \ldots, \infty$ which are utilized is at most $2k - j' - 1$.*

Proof: The proof is by induction on the length of the execution. We need to consider the following cases:

Process p starts getName **(base):** Just before process p starts getName there are at most $k-1$ other processes on all the sieves. By Definition 6.1, there is at least one process in any utilized sieve. Therefore, there are at most $k-1$ utilized sieves and $k-1$ accessing processes and together with process p there are $2k-1$ accessing processes and utilized sieves.

A new process other than p starts getName **while p is at sieve $j \geq 0$:** If $j' = 0$, it follows by the same arguments as in the previous case. If $j' > 0$, then the inductive claim is not affected.

A sieve becomes utilized: By Definition 6.1, this happens only when a process releases a name. In this case, there is one more utilized sieve but the number of accessing processes has decreased by one.

A process q starts accessing sieve $l > 1$: If $l > j$, then the inductive claim is not affected. If $l \leq j$, then by the inductive hypothesis, before q (which might be p) left sieve $l-1$, there were at most $2k-l$ accessing processes and utilized sieves among sieves $l-1, \ldots, \infty$. If process q is not the last one to leave sieve $l-1$, then before q left $l-1$ there were at most $2k-l-2$ in sieves $l, \ldots, \infty$ and the claim holds. If q is the last process to leave then before q left sieve l, sieve $l-1$ is utilized, by Lemma 6.4. In this case, before q left sieve $l-1$, there were at most $2k-l-2$ utilized sieves and accessing processes on sieve l, and the claim follows.

■

This implies the next corollary:

Corollary 6.6 *Assume that the point contention during p's operation is k, then p wins in sieve at most* $2k-1$.

As in [AF99a, AAF+99b], this implies that a process obtains a name in the range $1, \ldots, k(2k-1)$; the step and space complexity calculations are simple.

Theorem 6.7 *Algorithm 6 solves adaptive long-lived* $O(k^2)$*-renaming with step complexity* $O(k^2 \log k)$*, using* $O(n^3 N)$ *registers.*

Chapter 7

Content Adaptive Gather and Active Set

The fundamental concurrent storage object that we implement is the gather object. Its behavior is regular because its only required property is that a gather operation must return all the updates that finished before the gather started. Constructions in later chapters extensively use a simplified version of the gather object – the active set object. For a description of the gather and active set objects see the Introduction and Section 2.3.

7.1 Description

Definition 7.1 *The content adaptive* gather *object supports the following operations:*

1. CA-put(v): *with which a process updates its current value.*
2. CA-gather(): *which returns a dataset containing an element,* $\langle p, v \rangle$ *for each process with non* $-$ *value.*

The object must satisfy Properties I, II (validity, gather).

Definition 7.2 *The content adaptive* active set *object supports the following operations:*

1. joinSet(): *with which a process joins the active set.*
2. leaveSet():*with which a process leaves the active set.*
3. getSet():*which returns the process id of any process* p' *s.t.,* p' *has finished a* joinSet() *operation, op', before op started and did not start a* leaveSet() *operation after op' and before op finishes. Operation op does* not *return the process id of any process*

Algorithm 8 Code for content adaptive gather for process p.

```
   Type:
       pid = process id, 1..N;
   Shared:
       A[1..2N^2], atomic MRMW registers of type ⟨pid, val⟩ initialized to −;
       last[1..2N^2], atomic MRMW registers of type pid initialized to 1;
       C[1..N][1..2N^2], atomic SWMR registers of type dataset, each initialized to ∅;
   Local registers global to the program:
       index = 0;

   procedure CA-put(v)
1:     if v ≠ − then
2:         if index = 0 then index:=2k^2-rename(p);
3:         bubble_up(⟨p, v⟩);
4:     else                                                    // v = −
5:         if index > 0 then
6:             bubble_up(−);
7:             2k^2-release-name(p);
8:             index := 0;

   procedure bubble_up(v)
9:     A[index]:=v;
10:    for t = index down to 1 do
11:        C[p][t]:=−;                                         // different than ∅
12:        last[t]:=p;
13:        C[p][t]:=CA-gather(t);
       od;

   function CA-gather() returns dataset.
14:    return CA-gather(1);

   function CA-gather(t) returns dataset.
15:    q:=last[t];
16:    tmp:=C[q][t];
17:    if tmp = − then
18:        tmp:=CA-gather(t+1);
19:        if ⟨A[t].pid, ∗⟩ ∉ tmp then tmp := tmp ∪ {A[t]};
20:    return tmp;
```

Algorithm 9 Code for active set for process p.

```
   Shared:
       G, a CA-gather object;

   procedure joinSet()
1:     G.CA-put(p);

   procedure leaveSet()
2:     G.CA-put(−);

   procedure getSet()
3:     return(G.CA-gather());
```

p' s.t., (1) p' has never performed a joinSet() *or (2) p' has finished a* leaveSet() *operation, op', before op started and did not start a* joinSet() *operation after op' and before op finishes.*

Given an implementation of a gather object, it is easy to implement an active set object (Algorithm 9). Replacing the N flag bits described in the specification (Section 1) with one content adaptive gather object creates a content adaptive active set with the same step complexity as the gather.

To implement a gather object we use a $2N^2$ entries array in which processes store their values. To update its value a process captures an entry in the array, in exclusion, as close to the beginning of the array as possible. It then records its id and value in the multi-writer-multi-reader register associated with this entry of the array. At this point the process has successfully written its value as later updates can not overwrite it until the process releases the entry (writes $-$). However, subsequent gather operations may have to scan the array to find the values in all the slots. If this gather happens with low contention (e.g., in solo) such a scan would make it not adaptive. Therefore, before finishing the put operation the process has to bubble up its current value to the beginning of the array. Future gathers then start at the beginning of the array, and in the absence of contention find the needed values at the beginning of the array. In bubbling up, a process iterates on the entries of the array from the entry it has captured up to the top (beginning) of the array. In each such entry it recursively performs a sub-gather of the values recorded in the part of the array which is below this entry, and records this gather in a dedicated register that is associated with this entry and process.

To perform a gather a process starts at the top (beginning) of the array gathering values as it goes down. A key point is that the number of entries the gathering process scans depends on the point-contention it encounters. In the absence of contention it

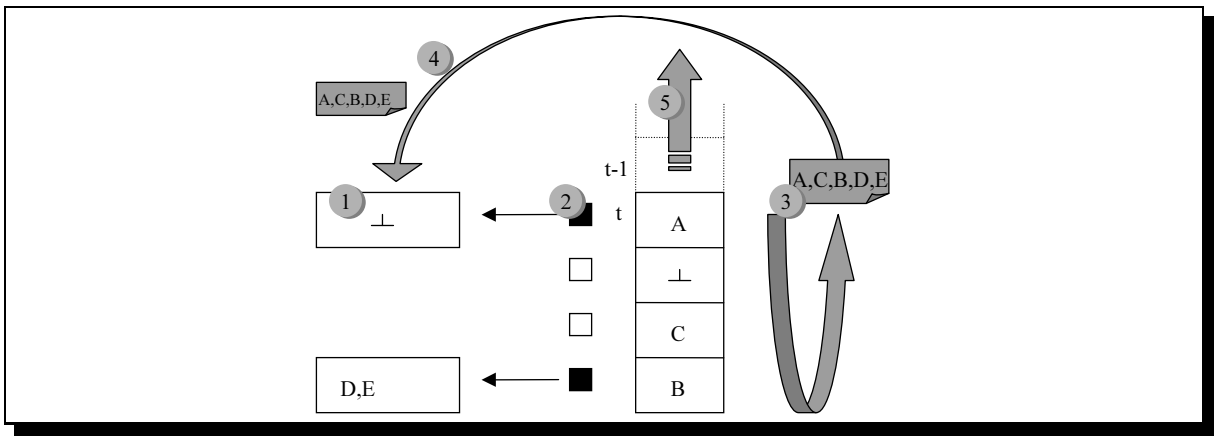


Figure 7.1: The Bubble up operation.

finds the necessary values at the first entry of the array, gathered by the last bubble up operation.

To capture an entry in the array in exclusion we use any of the long-lived and adaptive $2k^2$-renaming algorithms presented before (Section 4.2 or Chapter 6. After acquiring a name *index* (procedure CA-put(v)), and assuming that $v \neq -$, the process calls the bubble_up() procedure, passing its id and value as parameters. The procedure in turn writes the pair in a register associated with entry *index* in the array (Line 9) and starts bubbling up. If $v = -$, the CA-put($-$) operation passes $-$ to the bubble_up() procedure (Line 6), which has the effect of erasing its id and value from the captured entry and bubbling the change up. Only then the process releases the acquired name (Line 7).

The bubbling up process (Lines 10 to 13) is schematically presented in Figure 7.1. The process goes through the entries of the array from the index it had captured to the top. In each entry it performs a sub-gather of the bottom part of the array (Function CA-gather(t)) and records it with this entry. To do that we associate with each entry a pointer, called *last*, that points to the single-writer-multi-reader register of the last process that has recorded a sub-gather with this entry. The process performs the bubbling up through an entry in a particular order: First it writes $-$ in its single-writer-multi-reader register associated with this entry, called $C[p][i]$ for process p in entry i (①), secondly it writes its name into *last* (②), thirdly it performs the sub-gather (③) and finally it records its result in $C[p][i]$ (④) and iterates (⑤).

This particular order of recording gathered information in each entry guarantees the following property: If a process q reads $last[i] = p$ and subsequently it reads $C[p][i] = vector$ then any put operation that has updated a value below entry i in the array and has terminated before q read $last[i]$ is included in $vector$. Furthermore, if q observes $C[p][i] = -$ then it knows process p is concurrent with its operation and by the adaptiveness it may now perform a few more operations, in particular recursively gathering the information

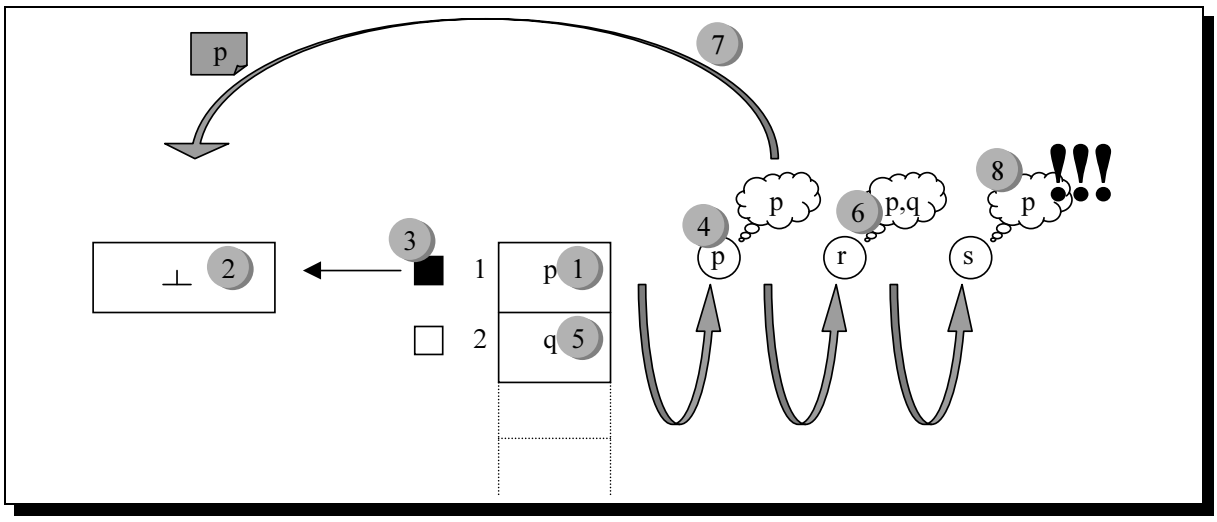


Figure 7.2: The gather algorithm does not satisfy the collect propery.

at entry $i+1$ (Line 18 in the code).

The algorithm satisfies Properties I, II (validity, gather) but not Property III (collect). For example, look at Figure 7.2. Let process p be some process that writes its value in the first slot (①), updates $C[p][1]$ to $-$ (②), updates $last[1]$ to p (③), performs a CA-gather(1)operation which returns only its own value and is just about to write the result into $C[p][1]$ (④). Let process q be some process that only then joins in, chooses slot 2 as its entry slot and writes its id and value in $A[2]$ (⑤). Now, if some process r performs a CA-gather(1) operation, it returns q's value (⑥). However, after r finishes, process p may perform its write in $C[p][1]$ (⑦) and then some other process, s, may perform a CA-gather(1) (⑧). Process s does not return q's new value because it finds a non $-$ value in $C[p][1]$. The gather property (Property II) is not violated because q has not finished its update operation but the later gather returns older values which means that the collect property is not satisfied.

7.2 Correctness

Any execution α of procedure bubble_up() by process p has a value, s^p_α, that is passed to it as parameter (s^p_α can either be a pair $\langle p, v\rangle$, or $-$). Given a set R of procedure bubble_up() executions, we say that a value s^p_α is in R, denoted $s^p_\alpha \in R$, if $\alpha \in R$. We say that s^p_x is *bigger* than s^p_y (denoted $s^p_x > s^p_y$) if the bubble up operation x ended before the bubble up operation y started. Given a set R of bubble_up() executions and a process id p, let $Latest^p_R$ be the value passed as parameter to the latest bubble up execution of p in R. If there is no such execution then $Latest^p_R$ is undefined.

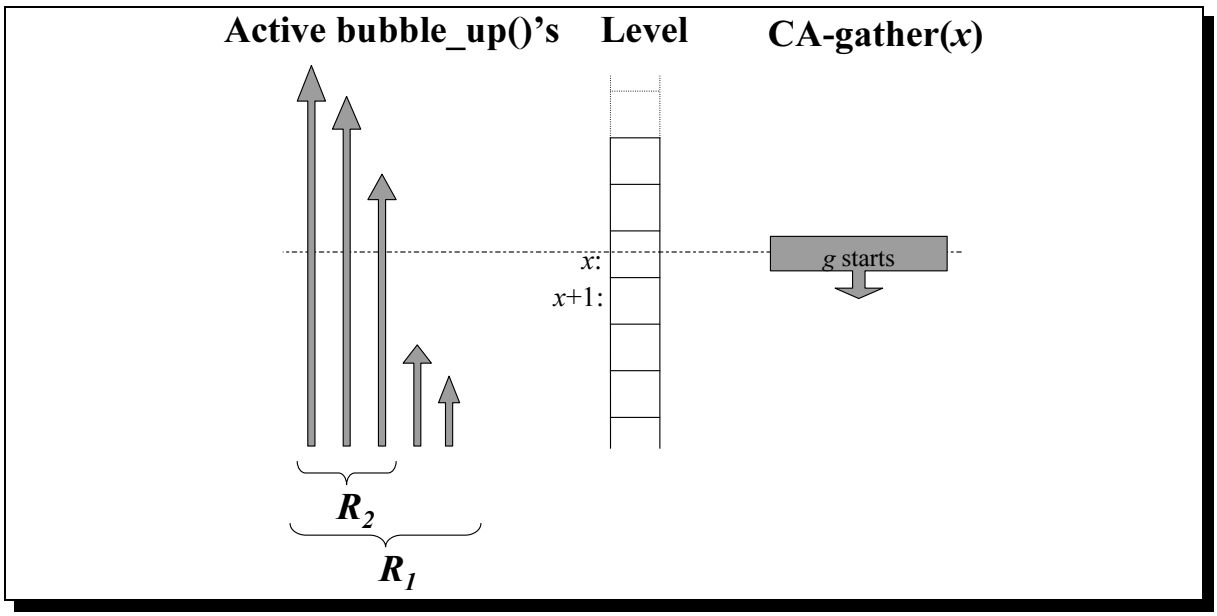


Figure 7.3: CA-gather(x) returns at least the values written by $R2$ and at most the values written by $R1$.

Given an execution α of bubble_up(), we define the *entry slot* of α to be the value of *index* after Line 2, if $v \neq -$ (Line 1) or the value of *index* after Line 4 otherwise. Given an execution α of bubble_up() with entry slot x, we say that α crossed slot $x' \leq x$ if the process executing α has already changed $last[x']$ to its id (Line 12).

Given an execution, g, of CA-gather() we denote by s_g the set of pairs returned by g. Given a slot x in A, we denote by s_{ax} the set of pairs which is written in $A[x]$ (s_{ax} is either an empty set or a singleton). For every set of pairs, s, we denote by $s[p]$ the pair which corresponds to p (the id of p is its first element) or $-$ if there is no such pair.

Lemma 7.1 *Let α be an execution of the gather object, α' a prefix of α. Then for every process p and any prefix α' of α, there is at most one slot, x, in A s.t., $s_{ax}[p] \neq -$.*

Proof: Process p writes in A only in the bubble_up(v) procedure (Line 9), using *index* as the slot number and v as the value to write. Procedure CA-put() sets *index* and then calls bubble_up(v) (Lines 1-3). However, a new value is chosen for *index* only if it is found to be equal to 0 . But *index* is set to 0 (Line 8) only after a bubble_up($-$) operation (Line 6), which means that before a new entry in A is chosen, the value in the last entry is erased with $-$ and the claim holds. ■

The following claim states that a CA-gather(x) operation returns the values of the last update operation of every process that entered (wrote its value) in floors x and below and that bubbled up to floor x before the CA-gather(x) started (see Figure 7.3).

Claim 7.2 *Let α be an execution of the gather object, and let g be an execution of* CA-gather(x) *contained in α. Assume that R_1 is the set of all bubble up operations with entry point bigger or equal to x that started before the end of g and R_2 is the set of all bubble up operations that crossed x before the beginning of g. Then, for every process p, if $Latest^p_{R_2}$ is defined then there exists some bubble_up() execution of p, b, s.t, $s^p_b \in R_1$, $s^p_b \geq Latest^p_{R_2}$ and $s^p_b \in s_g$.*

Proof: We prove the claim by induction on the length of the prefixes of α.

The claim trivially holds for $\alpha_0 = e_0$ because no CA-gather() operation can be contained in it. Assume that the claim is correct for prefix execution α_i. We show correctness for $\alpha_{i+1} = \alpha_i e_{i+1}$.

The only event, e_{i+1} that may affect the the correctness of the claim is the return operation in Line 20 of some CA-gather(x) operation, denoted g, by some process q. Let p be some process s.t., $Latest^p_{R_2}$ is defined and let r be its last bubble_up() execution in R_2.

1. Assume that the returned set of g was the result of a CA-gather($x+1$) (Line 18), denoted g', perhaps merged with $A[x]$ (Line 19). Denote by s'_g the set of pairs returned from g' and by s_{ax} the set of pairs read from $A[x]$. As $r \in R_2$ it follows that p crossed x so its entry point is bigger or equal to x.

   (a) Assume that r's entry point is bigger than x. Denote by R'_1 the set of all bubble up operations with entry point bigger or equal to x that started before the end of g' and by R'_2 the set of all bubble up operations that crossed x before the beginning of g'. Since $r \in R_2$ and since g' is contained in g, it follows that $r \in R'_2$. $Latest^p_{R'_2}$ must be defined (at most it is equal to $Latest^p_{R_2}$) so by the induction hypothesis, $s_{g'}[p] \geq Latest^p_{R'_2} \geq s^p_r$. If $s_{g'}[p] \neq -$ then $s_g[p] = s_{g'}[p]$ (Line 19) and the claim holds. If $s_{g'}[p] = s_{ax}[p] = -$ then $s_g[p] = -$ and again the claim holds. Assume that $s_{g'}[p] = -$ but $s_{ax}[p] \neq -$. The entry point of r is assumed to be bigger than x. (1) If $s^p_r \neq -$ then, as r writes s^p_r in its entry point ((Line 9), if follows from Lemma 7.1 that during r, $s_{ax}[p] = -$. Since during g $s_{ax}[p] \neq -$it follows that the value read by g from $A[x]$ was written in $A[x]$ by an update operation of p that was performed after r and therefore $s_{ax}[p] > s^p_r$. (2) If $s^p_r = -$ then by the definition of entry point, the previous update operation of p must have been some bubble_up($v \neq -$) operation to the same entry point so by the same arguments as in (1), the claim holds.

   (b) If the entry point of r was x then every value read from $A[x]$ for p after r is bigger than s^p_r ($s_{ax}[p] \geq s^p_r$). If $s_{g'}[p] = -$ then $s_{ax}[p]$ is returned for p and the claim holds. Assume otherwise and let r' be the update operation by p s.t., $s_{g'}[p] = s^p_{r'} \neq -$. Assume that $s^p_r \neq -$. As r crossed x before g started it also entered at x (so wrote s^p_r in $A[x]$) before g' was executed. By Lemma 7.1,

during r the only non $-$ entry for p was in $A[x]$. By the induction hypothesis, the entry point of r' was bigger than x so it must be that $s^p_{r'} \geq s^p_r$ and the claim holds. If $s^p_r = -$ then by the definition of entry point, the last update operation by p before r must have been bubble_up($v \neq -$) so, by the same arguments as in the previous case, the claim holds.

2. Otherwise $tmp \neq -$ (Line 17). Then q first read the name of some process q' from $last[x]$ and then read some (possibly empty) set of values from $C[q'][x]$. Denote by g' the CA-gather(x) operation performed by q' the result of which was read by q. From the assumption p already crossed x before g started so p wrote $-$ in $C[p][x]$ and then p in $last[x]$. Therefore, q' must either be equal to p, or be some process that crossed x after p. Since q' first erases its old gather result in $C[x][q']$ (Line 11) and only then crosses x (Line 12) it follows, as $C[x][q'] \neq -$, that g' was performed after q' crossed x so after p crossed x. In other words, $r \in R'_2$ and therefore, by the induction hypothesis, the result of g' includes some $s^p_{r'} \geq s^p_r$ and the claim holds.

■

Because every pair of every process in s_g is from R_1, Property I (validity) holds. Because the value for process p is bigger than $Latest^p_{R_2}$, Property II (gather) holds.

Theorem 7.3 *Algorithm 8 is a valid implementation of a gather object (as defined in Definition 2.9).*

7.3 Complexity

Algorithm 8 may take $O(\hat{k}^4)$ steps. For example, consider the following scenario. Process p enters the array at entry point $\hat{k}^2$ and then starts bubbling up, where in each CA-gather(t) operation it descends all the way down to its entry point. Process p keeps reading $-$ in all the entries of C because some process repeatedly updates its value, always interfering with p (writing $-$ in C just before p reads it). In fact, it may be that in each slot a different process interferes with p. The point contention stays $\hat{k}$ because these processes finish immediately afterwards.

A full complexity analysis is presented later for the gather with early stopping mechanism. Intuitively, A CA-gather operation only descends through slots that have active processes bubbling up in. Therefore, a single CA-gather operation can descend down to at most slot $2\hat{k}^2$ so its step complexity is $O(\hat{k}^2)$. Since the name process p gets from the renaming algorithm is at most $2k^2$ the total step complexity of the bubble_up() operation and also of the CA-put() operation is at most $O(\hat{k}^4)$.

The CA-gather object uses the result of the $2k^2$ renaming as an index into array A. Since we assume that process id's are in the range $1 \ldots N$, process i may use the

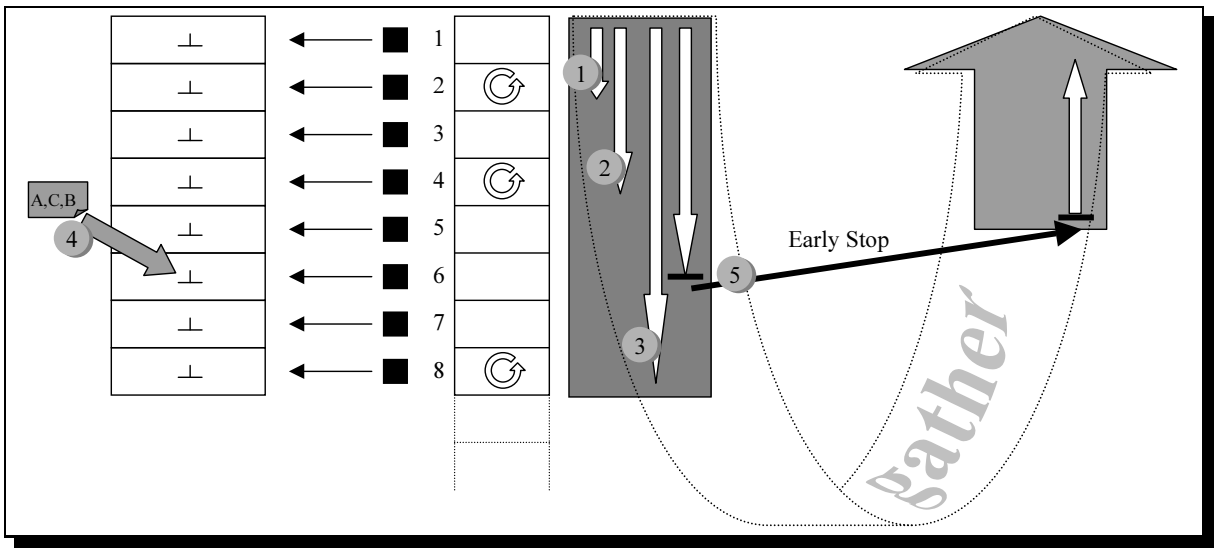


Figure 7.4: Gather with early stopping.

renaming result as an index into A only if the result is smaller or equal to N, and use $N + i$ otherwise. The complexity of the CA-gather operation is then $O(\min(\hat{k}^3, N\hat{k}))$.

7.4 Early Stopping

The early stopping mechanism is used to improve the efficiency of the CA-gather() operation. Assume that process p reads $-$ in $C[q][x]$ during its CA-gather(x) operation and therefore has to recurse to CA-gather($x+1$). The idea behind the early stopping mechanism is that if now process q writes a gather result in $C[q][x]$ then p can stop the recursion and return this value as the result of CA-gather(x), as if it has never had to recurse. The crux of the mechanism is to keep re-checking entries that were previously skipped over for new data (non $-$ values), and to rollback the recursion if a new value is detected.

More specifically, assume that process p calls CA-gather(x) and that during the recursion, entry $x' \geq x$ was skipped. That is, during CA-gather(x') process p found $last[x'] = q$ and $C[q][x'] = -$, for some process id q. If during the recursion calls of p to CA-gather($x'+1$),CA-gather($x'+2$),... it detects that a value was written to $C[q][x']$ then the recursion can stop, rollback to CA-gather(x') and return $C[q][x']$ as its result. Because q was the last process to cross x' before the CA-gather(x') operation of p started, the result of the CA-gather(x') operation performed by q and written into $C[q][x']$ is a valid return value for p.

Two obstacles come to mind. First, it is possible that $C[q][x']$ is continuously being

cleared (set to $-$) by q so p never has a chance of reading it. We solve this problem by adding a backup array to $C[][]$, called $Cprev[][]$. Processes copy their entries in C to $Cprev$ before clearing them. This way, there is always some CA-gather(x') result of q available for p. We also attach a time stamp (increasing counter) to every update of C so that it is possible to accurately detect changes in C and $Cprev$. In particular, if $C[q][x']$ is read twice to be $-$ we need to know if $Cprev[q][x']$ was updated in between. If so, $Cprev[q][x']$ can be returned as a valid CA-gather(x') result.

The second obstacle is the complexity of the operation. If every recursive call checks all the previously skipped over entries then the complexity of the CA-gather(x) operation is $\Omega(\hat{k}^2)$ and, again, the overall complexity is $O(\hat{k}^4)$. This is why a process rechecks entries only every $2^i, i = 1, 2, \ldots$ slots. In the proof we show that after at most $2\hat{k}$ skips, the process either stops or finds a new value in one of the skipped entries and the recursion can rollback. The sum of rechecks performed up to that point converges to $O(\hat{k})$.

Figure 7.4 shows a possible execution of the gather with early stopping mechanism. At the beginning of the execution there is an active process, p_i, in each floor, i, where p_i wrote $-$ in $C[i][p_i]$ and then wrote its id in $last[i]$. The gathering process starts its execution at floor 1 and, since there are active processes in all the floors, keeps skipping floors. The early stopping mechanism comes into effect at floors 2, 4, and 8 with the result that the gathering process rechecks all the floors it skipped over (①,②,③). Just after ③, the active process in floor 6 writes its data in C (④). The early stopping mechanism detects this (⑤) and the recursive gather calls are rolled back as if the recursion terminated at floor 6.

The code of the gather with early stopping for process p is presented in Algorithm 10. The only changes to the bubble_up() procedure are in Line 3 (copying C to $Cprev$) and in Line 6 (attaching a time stamp to the value written in C). Procedure CA-gather(x) first initializes *crossed* to $\emptyset$ (Line 8). The variable will contain the information about the slots and processes that were skipped over while descending. Then, CA-gather(x) initializes *earlyStop* to a value bigger than all the possible slot values (Line 9). A valid slot number in *earlyStop* will signal that an early stopping has occurred and the procedure call can rollback to the slot number in *earlyStop*. Last, CA-gather(x) calls a helper function, CA-gather_rec() which recursively calls itself again and again.

Procedure CA-gather_rec() starts with an if block (Lines 10-16) that is only executed when the number of skipped over slots is an exponent of 2. Every slot t' and process r that were registered in *crossed* are checked. If $C[r][t'] \neq -$ then r finished a CA-gather(t') operation since p skipped it and it is possible to rollback to CA-gather_rec(t'). To do this, we set $earlyStop = t'$ and return the value of $C[r][t']$ (Lines 12, 13). This has the effect of rolling back the calculation up to CA-gather_rec(t') without changing the return value (Line 22). The same is done if the time stamp in $Cprev[r][t']$ is bigger than the time stamp registered in *crossed* (Lines 15, 16).

After the if block, the procedure performs the same steps as the original gather()

Algorithm 10 Code for content adaptive gather with early stopping for process p. Changes from the original code are marked with "!".

```
   Local registers global to the program:
        crossed, a set of ⟨pid, entry, timestamp⟩ initialized to ∅;
        earlyStop, an integer;
        TS, an integer initialized to 0;

   procedure bubble_up(v)
1:      A[index]:=v;
2:      for t = index down to 1 do
3: !        Cprev[p][t] := C[p][t];                          // keep copy available
4:          C[p][t]:=−;
5:          last[t]:=p;
6: !        C[p][t]:=⟨CA-gather(t), TS⟩;                     // attach time stamp
7: !        TS := TS + 1;
        od;

   function CA-gather(t) returns dataset.
8:      crossed := ∅;
9:      earlyStop := ∞;                                      // greater than 2N^2
        return(CA-gather_rec(t));

   function CA-gather_rec(t) returns dataset.
10:!    if log_2 |crossed| ∈ ℕ then                          // early stopping
   !        for every ⟨r, t', ts'⟩ ∈ crossed do
11:!            tmp := C[r][t'];
12:!            if tmp ≠ − then earlyStop := t';
13:!                                  return tmp.set;
14:!            tmp := Cprev[r][t'];
15:!            if tmp.ts > ts' then earlyStop := t';
16:!                                  return tmp.set;
   !        od;
17:     q:=last[t];
18:     tmp:=C[q][t];
19:     if tmp = − then
20:!        crossed := crossed ∪ ⟨q, t, Cprev[q][t].ts⟩;
21:         tmp:=⟨CA-gather_rec(t+1), ·⟩;      // just to have uniform tmp datatype
22:!        if earlyStop > t and ⟨A[t].pid, ∗⟩ ∉ tmp.set then tmp.set := tmp.set ∪ {A[t]};
23:     return tmp.set;
```

procedure except for two changes. In Line 20 the procedure saves in *crossed* the information about the skipped over slot and process. In Line 22 the value in the current slot, t, is added to the values gathered by the recursion call to CA-gather_rec$(t+1)$ only if $earlyStop > t$.

Notice that if the early stopping mechanism in Lines 11-16 does not come into effect during the execution (the if statements in Lines 15 and 12 always fail) then $earlyStop > t$ for *all* CA-gather_rec(t) operations and, as can be verified by the code, the algorithm's execution is equivalent to the execution of Algorithm 8.

7.5 Correctness of Early Stopping

The following technical lemma asserts the correctness of the rollback mechanism.

Lemma 7.4 *If at the end of a* CA-gather_rec(x) *procedure call, $x > earlyStop$ then there is a pending* CA-gather_rec$(earlyStop)$ *call that recursively called* CA-gather_rec(x). *Furthermore, the return value of* CA-gather_rec$(earlyStop)$ *will be the return value of* CA-gather_rec(x).

Proof: (Directly from the flow of the algorithm.) If $x > earlyStop$ then some procedure CA-gather_rec() must have set $earlyStop$ (Lines 12, 15). Therefore, the value of $earlyStop$ must be the value of some slot registered in a 3-tuple saved in *crossed*. But slot $earlyStop$ is only added to *crossed* just before the single recursive call to CA-gather_rec$(earlyStop+1)$ (Lines 20, 21) and since $x \neq earlyStop$ the CA-gather_rec$(earlyStop)$ call is still unfinished. Let CA-gather_rec(x') $x > x' \geq earlyStop$, be an unfinished procedure call that recursively called CA-gather_rec(x). Since $x' \geq earlyStop$ its return value is precisely the value returned from CA-gather_rec$(x'+1)$ (Line 22). But this is correct for all x', s.t., $x > x' \geq earlyStop$ so the second part of the lemma is also correct. ■

The following Claim is equivalent to Claim 7.2. We reconstruct the proof taking into account the early stopping mechanism.

Claim 7.5 *Let α be an execution of the gather object, and let g be an execution of* CA-gather_rec(x) *contained in α s.t., at the end of the operation $earlyStop \geq x$. Assume that R_1 is the set of all bubble up operations entry point bigger or equal to x that started before the end of* CA-gather_rec(x) *and R_2 is the set of all bubble up operations that crossed x before the beginning of* CA-gather_rec(x). *Then, for every process p, if $Latest^p_{R_2}$ is defined then there exists some bubble_up() execution of p, b, s.t, $s^p_b \in R_1$, $s^p_b \geq Latest^p_{R_2}$ and $s^p_b \in s_g$.*

Proof: As in the proof of Claim 7.2, we prove the claim by induction. Assume that the claim is correct for α_i. We prove correctness for $\alpha_{i+1} = \alpha_i e_{i+1}$.

At the end of the CA-gather_rec() operation $earlyStop \geq x$. Even if the operation performed the early stopping mechanism (Lines 10-16), *crossed* contains only entries with slot number smaller than x and one such slot is copied to *earlyStop* upon terminating (Lines 12 and 15). Therefore, the operation must have terminated at Line 23 and to effect the correctness e_{i+1} must be the return operation in Line 23 of some CA-gather_rec(x) operation, denoted g, by some process q.

If $earlyStop > x$ or $earlyStop = x$ and the if statement in Line 19 fails ($tmp \neq -$) then it is possible to exactly follow the arguments of the proof of Claim 7.2.

Assume that $earlyStop = x$ and the condition in Line 19 evaluates to **true** ($tmp = -$). From Lemma 7.4 it follows that the returned set of g is in fact the returned set of some CA-gather_rec($x + i$) call recursively called by g (Line 21). Since $earlyStop \neq 2N^2 + 1$ it follows that the early stopping mechanism was activated in CA-gather_rec($x + i$) and the call terminated at Line 13 (case 1) or at Line 16 (case 2).

Let $\langle r, x, ts \rangle$ be the 3-tuple added by g to *crossed* (Line 20) and found by CA-gather_rec($x + i$) to be such that either $C[r][x] \neq -$ (Line 12) or $Cprev[r][x].ts > ts$ (Line 15). Process r was the last process to cross x before g started so any CA-gather(x) result calculated by r after crossing x is also a valid result for g.

(case 1) In the proof of Claim 7.2 Part 2 we show that if g reads the id of r in $last[x]$ and then reads a non $-$ value in $C[r][x]$ then this value is a valid return result for g. Since the recursive calls both in the implementation with early stopping and the one without have no side effects, it is possible to use the same arguments as in the proof of Claim 7.2, Part 2 to show that this property continues to hold from the time that g reads the id of r in $last[x]$ until the end of the execution of g. In other words, every value written in $C[r][x]$ at any point of the execution after g reads the id of r in Line 17 and before g terminates is a valid return result for g. The correctness of case 1 immediately follows.

(case 2) Every value written in $Cprev[r][x]$ was also written at some point in $C[r][x]$ (Line 3). Operation g first reads $last[x]$ (Line 17) and then registers the time stamp read from $Cprev[r][x]$ (denoted ts) in *crossed* (Line 20). By the monotonicity of the time stamps of r it follows that every value read from $Cprev[r][x]$ with time stamp greater than ts was written in $C[r][x]$ after g executed Line 17. Following the same arguments as in case 1, such a value is a valid return result for g.

■

7.6 Complexity Analysis

We analyze the complexity of the gather object with early stopping algorithm. We consider process p to be participating (active) from any invocation of CA-put($v \neq -$) by p until the next response to a CA-put($-$) by process p, and denote the induced content adaptive point contention as $\hat{k}$. First, we prove that the step complexity of the

CA-gather_rec(x) operation is $O(\hat{k})$ and then we show that the bubble_up(), and therefore also the CA-put() operation take $O(\hat{k}^3)$ steps.

Assume that p is a process executing a CA-gather_rec(x) operation, g. We say that p *skips* over slot x if it reads $last[x] = p'$, $C[p'][x] = -$ for some process p' (Lines 17-19), and then has to read $Cprev[p'][x].ts$ and perform a recursive call to CA-gather_rec($x+1$). In such a case we also say that p *skips over* operation g' of p', where g' is the bubble up operation that wrote the time stamp read by g from $Cprev[p'][x]$ (Line 20). Bubble up operation g' is said to be *at* slot x while variable t of the loop (Line 2) is equal to x. The operation is said to have *finished* at slot x when $C[p'][x]$ is updated by p' to its CA-gather_rec(x) result (Line 6).

Lemma 7.6 *If g skips over g' in slot x then g' was at slot x concurrently with the skipping operation (Lines 17–20) of g.*

Proof: Process p' writes a value different than $-$ into $C[p'][x]$ before continuing up to slot $x-1$ (Line 6) so there is at most one entry $C[p'][*]$ at any point of time which is equal to $-$. Therefore, if p skips over p' at slot x it means that p' is currently active at slot x, executing bubble up operation op. Operation op first writes in $Cprev[p'][x]$ and only then writes $-$ in $C[p'][x]$. Operation g first reads $C[p'][x]$ and only then reads $Cprev[p'][x]$. It follows that either $g' = op$ or g' is some later operation of p' that wrote in $Cprev[p'][x]$ before g read it. In both cases the lemma holds. ■

Lemma 7.7 *If process p executing gather operation g has already skipped over $\hat{k}+i$ slots then the operations in at least i of the slots are finished.*

Proof: By Lemma 7.6, a process skips a slot when it encounters another process p' that traverses this slot in its bubbling up procedure (lines 4 – 6). Because gather operations access slots in increasing order while bubble up operations access slots in decreasing order, p may skip over p' at most once during the same gather call of p and the same bubble up call of p'. Since there can be at most $\hat{k}$ concurrent bubble up operations, if p skipped over $\hat{k}+i$ slots then at least i out of the bubble ups must have finished. ■

Claim 7.8 *The largest slot index that can be reached during an execution of procedure* CA-gather(x) *by process p is $x+2\hat{k}$.*

Proof: The size of *crossed* is equal to the number of slots that were skipped by p since it is increased with every recursive call (Line 20). By Lemma 7.7, after at most $\hat{k}+1$ recursions of p there must be some process p' and entry x s.t., $\langle p', x, ts\rangle \in crossed$ and the skipped over operation of p' at x finished. After at most $2\hat{k}$ recursion calls the early stopping mechanism (Lines 10–16) must operate. If $C[p'][x'] \neq -$ then operation CA-gather_rec($x+2\hat{k}$) terminates in Line 13. Assume otherwise, then process p' must

have started yet another bubble_up() procedure(s) after g'. Process p' copies the value of $C[p'][x']$ to $Cprev[p'][x']$ before clearing it (Line 3). This value was either written by g' when it finished at slot x', or by a later operation of p'. Because the time stamps increase monotonically it follows that now $Cprev[p'][x'].ts > ts$ and therefore procedure CA-gather_rec($x + 2\hat{k}$) must terminate in Line 16. ■

A CA-gather(x) procedure may make up to $2\hat{k}$ recursion calls, All but $\lfloor \log_2 2\hat{k} \rfloor$ out of which make a constant number of steps. In $\log_2 2\hat{k}$ calls, the procedure traverses the entire *crossed* array collected up to that point in time. The number of steps made by the CA-gather(x) procedure due to the early stopping mechanism is $C(2\hat{k} + \hat{k} + \frac{\hat{k}}{2} + \ldots + 1)$ which converges to $O(\hat{k})$.

Since the entry slot of any bubble_up() procedure is at most $2\hat{k}^2$ away from the top, a bubble_up() procedure may call CA-gather() at most $\hat{k}^2$ times and therefore its step complexity is $O(\hat{k}^3)$.

Theorem 7.9 *Algorithm 10 is a valid implementation of a gather object (as defined in Definition 2.9). Furthermore, it is content adaptive to point contention with step complexity* $O(\hat{k}^3)$.

The next theorem immediately follows.

Theorem 7.10 *There is a content adaptive implementation of an active set object with step complexity* $O(\hat{k}^3)$.

Chapter 8

CA-collect and CA-snapshot

As is, the gather object is of limited use, due to its regular property. In this chapter we use the gather object to implement a collect object and then use the collect to implement a linearizable snapshot. However, all these implementations are only content adaptive. For a description of collect and snapshot see the Introduction and Section 2.3. For the definition of content adaptiveness see Definition 2.3.

8.1 CA-collect

The content adaptive collect object supports two operations, the CA-write$_p$($data$) operation and the CA-collect() operation. The CA-collect() operation returns a *dataset* containing a pair $\langle p, data \rangle$ for every process p whose associated value is different than $-$. If the last operation executed by any process was CA-write($-$) (or the the process had never written any value at all) then its value is implicitly returned by the fact that it is not in the set. Algorithm 11 satisfies Properties I, II and III (validity, gather and collect). Furthermore, it is content adaptive.

Essentially the algorithm guarantees that whenever the value stored for a process is different than $-$, then that process is registered in the active set object, $\mathcal{S}$ (i.e., appears as being active). This is achieved by processes registering in $\mathcal{S}$ before changing their stored value from $-$ to $\neq -$ (Line 2), and checking-out from $\mathcal{S}$ after updating their stored value to $-$ (Line 5). To collect the values stored (CA-collect()) a process performs getSet() and then reads the stored values for each process in the returned set (Lines 6-10).

In order to show that Algorithm 11 satisfies properties I – III we first prove that all the write operations performed by a single process p can be ordered with respect to all the collect operations performed during the algorithm's execution. We call this property *p-linearizability*.

Algorithm 11 Code for Content Adaptive Collect for process p.

```
    Shared:
        A[1 .. N], N atomic SWMR registers initialized to −;
        S, an active set object;

    procedure CA-write_p(data)
1:      if data ≠ − then
2:          S.joinSet();                                   // check-in
3:          A[p] := data;
        else                                               // data = −
4:          A[p] := −;
5:          S.leaveSet();                                  // check-out
        return;

    procedure CA-collect() returns dataset
6:      S' := S.getSet();
7:      view := ∅;
8:      for every p' ∈ S' do
9:          data' := A[p'];
10:         if data' ≠ − then view := view ∪ {⟨p', data'⟩};
        return(view);
```

Lemma 8.1 (p-linearizabilty) *Let α be an execution of Algorithm 11 and let G_p be the set of all* CA-write$_p$ *operations of p and collect operations (of all processes) executed in α. Then there exists a total order, $<_{G_p}$, between the operations in G_p s.t,*

1. *If operation op_0 completes before operation op_1 starts then $op_0 <_{G_p} op_1$.*
2. *Every collect operation returns the value written by the last* CA-write$_p$ *operation that is ordered before it.*

Proof: For every operation $op \in G_p$ we define the *representative event* of op as follows:

1. If op is a CA-write$_p$ operation then its representative event is the write event to $A[p]$ performed by op either at Line 3 or at Line 4.
2. if op is a collect operation then:
   (a) If op performs a read event from $A[p]$ (Line 9) then that event is its representative event.
   (b) If op does not perform a read event from $A[p]$ and during op's execution interval there is at least one write event (by p) of $-$ into $A[p]$, then op's representative event is the event immediately after the first of these write events.
   (c) Otherwise, we define the representative event of op to be the first event performed by op.

We define $<_{G_p}$ to be total order that exists between the operations' representatives and show that $<_{G_p}$ satisfies the stated properties.

Because every operation is mapped to an event in its execution interval the first part of the claim immediately holds. Because register $A[p]$ is atomic, every collect operation that reads $A[p]$ reads the last write of p in $A[p]$ which is also the update value of the last CA-write() operation, op, ordered before the collect operation.

Assume that some collect operation, op', by process q does not read $A[p]$. This means that p was not returned by the getSet() operation (Line 6). If p never wrote a non $-$ value then the representative event of op' is its first event and the claim holds (the initial value of all the processes is $-$). Otherwise, process p must have started a leaveSet() before the getSet() performed by q was finished and did not finish a joinSet() after that leaveSet() operation and before the getSet() started. Therefore, p must have wrote $A[p] := -$ before the getSet() was finished (Line 4) and did not afterwards write $A[p] := (v \neq -)$ (Line 3) at least until the getSet() operation started. If the write $-$ event occurs after op' started then the representative event of op' is the event immediately after this write event and therefore op' correctly returns the last update value of p (which is $-$). If this write event occurs before op' started then because p did not write a non $-$ value until the getSet() operation started the representative event of op' can be its first event. ■

Corollary 8.2 *Algorithm 11 satisfies properties I–III*

Proof: The gather and collect properties immediately follow from the lemma (proof to the contradiction). The value returned for process p is read from $A[p]$ so the validity property holds. ■

Claim 8.3 *The step complexity of the* CA-collect() *and* CA-write() *operations is* $O(\hat{k}^3)$, *where* $\hat{k}$ *is the content point contention as defined in Definition 2.3.*

Proof: The step complexity of the CA-collect object is dominated by the step complexity of the active set object, which is $O(k'^3)$ where k' is at most the number of processes that executed joinSet() and did not finish leaveSet(). But surely such processes started executing a CA-write($data \neq -$) operation and did not yet finish an CA-write($-$) operation so the claim holds. ■

Theorem 8.4 *Algorithm 11 is a valid implementation of a collect object (as defined in Definition 2.9). Furthermore, it is content adaptive to point contention with step complexity* $O(\hat{k}^3)$.

8.2 CA-snapshot

The CA-snapshot object supports two operations, CA-update(), and CA-scan() which returns a *dataset* containing 3-tuples of the form $\langle p, data, seq \rangle$. The object satisfies Properties I – IV. Furthermore it is content adaptive to point contention. I.e., to the maximum number of processes whose stored value is $\neq -$ at a point of time during the operation execution.

The basic idea of the implementation is to use the content-adaptive collect algorithm from the previous section in the atomic snapshot algorithm of [AAD+93]. In [AAD+93] processes take an atomic snapshot before every update and attach its result to the value written. To take a snapshot a process repeatedly double collects the values in the shared memory until either two collects are identical or some process was observed to change its value twice (to make this check robust every value is tagged with a sequentially increasing label, called a sequence number). In the latter case the snapshot associated with the second value of a process that has moved twice is returned.

There are several issues in the adaptation of this atomic snapshot algorithm. First, the content adaptive collect returns only the non $-$ values. Therefore, it is possible that although two consecutive collects return the same set of values (with associated sequence numbers) still some process p wrote twice between these two collects (p's value in the collects is $-$. It first writes a non $-$ value and then it writes $-$ again). We overcome this problem by using one MWMR register, called V, and requiring every process to write its

Algorithm 12 Code for Content Adaptive Snapshot for process p.

```
    Shared:
        V, atomic MWMR register of type ⟨pid, sequence #⟩;
        B[1 .. N], N atomic SWMR registers of type ⟨dataset, view⟩ initialized to ⟨∅, ∅⟩;
        C,S, content adaptive collect objects;
    Local persistent:
        seq, seq_1, integer initialized to 0;

    function CA-scan() returns dataset
 1:     seq_1 := seq_1 + 1;
 2:     S.CA-write_p(seq_1);                                   // from Algorithm 11
 3:     while(true) do
 4:         v_1 := V;
 5:         c_1:=C.CA-collect();
 6:         c_2:=C.CA-collect();
 7:         v_2 := V;
 8:         if (c_1 = c_2 and v_1 = v_2) then
 9:             S.CA-write_p(−);                                // check-out
10:             return c_2;
11:         if ∃p' s.t., ⟨p', ·⟩ ∈ c_1 ∪ c_2 ∪ {v_2} and ⟨p, seq_1⟩ ∈ B[p'].s then
                                        // p' moved & p' started after this CA-scan
12:             S.CA-write_p(−);                                // check-out
13:             return B[p'].sc;
        od;

    procedure CA-update(data)
14:     seq := seq + 1;
15:     s := S.CA-collect();
16:     sc:=CA-scan();
17:     B[p] := ⟨s, sc⟩;
18:     V := ⟨p, seq⟩;
19:     C.CA-write_p(⟨data, seq⟩);
```

name and sequence number in V before storing its value in the content adaptive collect object. Before and after performing the double collect a scanning process checks V. If the value in V has changed, it is considered that some process has moved, even if the two collects are identical.

Second issue, simple substitution of the collect operation with a content adaptive collect creates an adaptive with respect to interval contention algorithm. This is because, during the snapshot, k processes may update their values one after the other with no overlap in their executions. In this scenario the point contention is 2 but the algorithm makes $O(k)$ collect operations.

We solve this problem by having every scanning process, p, register in a content adaptive collect object $\mathcal{S}$. The registration includes a sequence number (seq_1 in Algorithm 12) that is increased with each scan operation (Lines 1, 2). At the end of the scan operation the process checks out by writing $-$ (Lines 9 and 12). Updating processes collect the set of processes and sequence numbers of the currently active scans. Then the updating processes perform a scan and atomically save the two results together in a SWMR array, called B (Lines 15-17). The key idea in making the algorithm adaptive to point-contention is that during a scan it is not necessary to wait until some process q moves twice. If the sequence number of the current scan of p is contained in the set written by q in $B[q]$ then the scan result written in $B[q]$ is a legal result for p because it was calculated after p started its scan operation. If not, then the update operation of q must have started before the scan of p and is concurrent with any other process that was observed to change its variable.

The implementation is shown in Algorithm 12. During updates process p first increases its sequence number, then collects the data of the currently active scans (Line 15), performs a CA-scan() (Line 16) and then writes the results to $B[p]$ (Line 17). Process p then updates V with $\langle p, seq \rangle$ (Line 18) to mark that it has moved and only then it writes its current data, along with the sequence number, in $\mathcal{C}$ (Line 19).

The CA-scan() operation begins by incrementing the sequence number and then registering it in $\mathcal{S}$ (Lines 1, 2). Then, the algorithm starts a loop of CA-collect() operations and reads of V. In each iteration process p first reads V (Line 4), then performs a double CA-collect() (Lines 5, 6) and then reads V again (Line 7). There are two terminating conditions:

1. If the two collects are equal and the two reads of V are equal then the process terminates, returning one of the CA-collect results (Lines 8-10).

2. If some process p' has moved during the current CA-scan() operation and the set of registered scan operations saved in $B[p']$ includes the sequence value (seq_1) of the current scan of process p, then the scan result in $B[p']$ is returned (Lines 11-13).

In both cases, the scan operation first checks out from $\mathcal{S}$ by writing $-$ (Lines 9 and 12). If the two terminating conditions fail the process iterates.

To prove linearizability we first define a representative event for every content adaptive update or scan operation. We use the total order among the events to define a total order among the updates and scans.

Definition 8.1 *Let α be an execution of Algorithm 12 and let S be the set of all content adaptive update and scan operations executed in α. For every operation $op \in S$ we define the* representative event *of op as follows:*

1. *If op is a* CA-update(*data*) *operation then its representative event is equal to the representative event of the* CA-write$_p$() *operation contained in it (Line 19), as defined in the proof of Lemma 8.1.*

2. *If op is a* CA-scan() *operation, s.t., the* `if` *statement in Line 8 succeeds then its representative event is the last event performed by the* CA-collect() *in Line 5 during op. We name such scan operations* base scans.

3. *As can be verified from the code, every non-base* CA-scan() *operation, op, returns the result of some other* CA-scan() *operation, op′ (Line 13). The representative event of op is equal to the representative event of op′.*

Lemma 8.5 (linearizability) *Let α be an execution of Algorithm 12 and let S be the set of all update and scan operations executed in α. Then there exists a total order, $<_S$, between the operations in S s.t,*

1. *If operation op_0 completes before operation op_1 starts then $op_0 <_S op_1$.*

2. *If u is the last update operation of some process p ordered before a scan operation op_1 then op_1 returns the update value of u for process p.*

Proof: We define $<_S$ to be total order that exists between the operations' representatives and show that $<_S$ satisfies the stated properties.

We prove the claim by induction. Let $\alpha = e_0, e_1, \ldots, e_i, e_{i+1}, \ldots$ and let the prefix α_i of α be $(e_0, \ldots, e_i)$, the sequence of the first $i+1$ events of α. Let S_i be the set of all finished update and scan operations in α_i, $<_{S_i}$ the total order among the operations in S_i.

At the beginning of the algorithm ($\alpha_0 = e_0$) there are no finished update and scan operations so the claim trivially holds.

Assume that the claim is correct for prefix run α_i. We show correctness for $\alpha_{i+1} = \alpha_i e_{i+1}$. The event e_{i+1} can effect the correctness of the claim only if it is the end of some operation. We assume this operation was performed by process p and denote this operation as op_1 and its representative event as rep_1.

Part 1 To prove Part 1 of the claim we show that rep_1 is contained in the execution interval of op_1. If op is an update operation or a base scan then the claim immediately holds. Otherwise rep_1 is equal to the representative of some other CA-scan() operation, op' (Line 13), performed by process p' during its update operation u (Line 16). Denote by w the $\mathcal{S}$.CA-write$_p$($seq1$) operation contained in op_1 (Line 2) and by r the S.CA-collect() operation contained in u (Line 15). Operation op_1 verifies that $\langle p, seq1 \rangle \in B[p'].s$ before returning $B[p'].sc$ (Line 11). Operation u writes in $B[p'].s$ the result of r. It follows from the properties of the CA-collect object that r did not finish before w which means that op' did not start before op_1. Since op_1 reads the result of op' in $B[p'].s$ it follows that op' also finishes before op_1. By the induction hypothesis rep_1 is contained in the execution interval of op' so it is also contained in the execution interval of op_1.

Part 2 If op_1 is a non base scan then the claim is correct by the induction hypothesis. Assume otherwise and let u be an update operation by process p_u s.t., u is the last update of p_u ordered before op_1. let v_0 be the value returned for p_u by the first collect (Line 5) and v_x be the value returned by the second collect (Line 6). Because op_1 is a base scan, the results of the two collects must be identical. If u is ordered by $<_{G_{p_u}}$ (Lemma 8.1) before the first CA-collect() operation in op_1 (Line 5) then the collect and therefore also op_1 return its update value. Assume otherwise. Then it must be that $v_0 = v_x = -$ because non $-$ values have sequence numbers attached to them. Let u_0 be the update operation by p_u that wrote v_0 and u_x be the update operation that wrote v_x. If there is no other update operation by p_u between u_0 and u_x it follows that $u = u_x$ and u just changed the value of p_u from $-$ to $-$, which means that the claim holds. Assume otherwise, then operation u_x must have performed the assignment to V (Line 18) after op_1 started the collect operation in Line 5 and before op_1 finished the collect operation in Line 6. In such a case op_1 would fail in the test in Line 8 which makes it a non base scan in contradiction to the assumption. Notice that sequence numbers are attached to all updates of V (even to $-$ updates) so overwrites in V can never hide the fact that some process moved.

■

The gather, collect and snapshot properties immediately follow from Lemma 8.5.

We use the following lemma to analyze the step complexity of the algorithm. The lemma states that if a CA-scan() operation performed by process p makes i iterations then there must be at least $i - 1$ processes that perform CA-update() concurrently with p and each other.

Lemma 8.6 *Let g be an execution of a* CA-scan() *operation by process p and assume that*

g terminates in its i'th iteration of the while loop (Line 3). Then $\text{ConPntCont}(\beta(g)) \geq i$ $(\hat{k} > i)$.

Proof: We prove the lemma by induction on i, the number of iterations. For $i = 1$ the lemma is correct because p is active. Assume that the lemma is correct for iteration i we prove correctness for iteration $i + 1$. Let e_{pc} be the last event executed by the CA-write(*seq1*) operation (Line 2) performed during g.

Because iteration $i + 1$ started, it follows that during iteration i, the if statement at Line 8 did not succeed. Therefore, there was at least one process, say p', s.t. p' updated either V or $\mathcal{C}$ during the i'th iteration of g, in particular between the first reads of V and $\mathcal{C}$ by p (Lines 4, 5) and the second reads (Lines 6, 7). We denote this update operation u_i. Since iteration i did not terminate it follows that the result of the collect operation,c', performed by u_i and written to $B[p'].s$ did not include the sequence number of g (Line 11). Operation g did not yet terminate so by Property II of the collect object it follows that c' started before e_{pc}. But operation c' is called from u_i and therefore it follows that u_i started before e_{pc}. Because u_i updates V or $\mathcal{C}$ between the first and the second reads of V and $\mathcal{C}$ in iteration i it follows that u_i finishes after e_{pc} and furthermore, u_i is different than $u_1, \ldots, u_{i-1}$. Therefore the claim holds. ■

Since any CA-scan() operation makes at most k iterations and in each such iteration at most $O(\hat{k}^3)$ steps, the step complexity of the CA-scan() and therefore also of the CA-update() is at most $O(\hat{k}^4)$.

Claim 8.7 *The step complexity of the* CA-snapshot *object is* $O(\hat{k}^4)$.

Theorem 8.8 *Algorithm 12 is a valid implementation of a snapshot object (as defined in Definition 2.9). Furthermore, it is content adaptive to point contention with step complexity* $O(\hat{k}^4)$.

Chapter 9

Snapshot

We can implement a fully adaptive snapshot object using the content-adaptive snapshot presented in Section 8.2 and a pile object (which will be described shortly). The main difference between the content-adaptive snapshot and the adaptive snapshot is that in the latter, even if there are many non $-$ values stored, still a solo run (or a run with very low contention) must complete within a constant number of primitive operations. Moreover, here processes never write $-$ values, but still the complexity is adaptive to the point contention.

Our snapshot object supports two operations: an update operation update($data$) and a retrieve operation scan(). The scan operation returns a *dataset* consisting of tuples of the form $\langle p, data, seq\rangle$, representing the value $data$ which was written by process p in its $seq/2$ update (the sequence numbers are incremented twice with every update operation). The scan also returns Σ, the sum of the seq part of all the tuples. The algorithm satisfies properties I – IV.

9.1 Description of the Algorithm

The algorithm (the pseudo code is in Algorithm 13) uses two objects: a content-adaptive snapshot and a pile of snapshots. A pile object is an adaptive object that supports two operations: throw_in($\langle num, data\rangle$) and pick(). The pick() operation returns the pair $\langle num, data\rangle$ with the biggest num part that was thrown in the pile before or perhaps during the operation. The pile of snapshots is used to store and return snapshots of update operations that have finished. The unique properties of the pile guarantee that in a point adaptive manner it returns the most recent snapshot that have been stored in it (before the pick operation has started). The content-adaptive snapshot is used by processes to capture updates that are concurrent with their own operation (either update or scan).

To take an atomic snapshot a process first scans the content-adaptive snapshot, thus

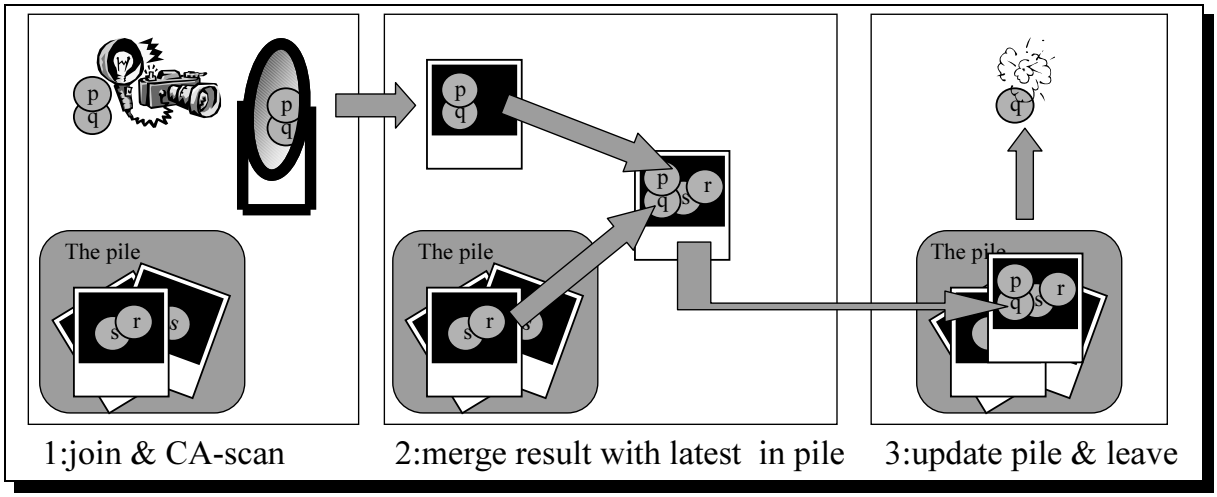


Figure 9.1: Adaptive Snapshot using a Pile.

obtaining a snapshot of the values written concurrently with it (Line 1). Secondly, it queries the pile to get the most recent snapshot that preceded the scan operation (Line 2). Finally, it unifies the two snapshots to get the desired atomic snapshot (Line 10). The scan operation also recalculates Σ (Lines 7-9) to reflect the sum of sequence numbers in the resulting union.

The entire update procedure is illustrated in Figure 9.1. To update a new value, a process first updates the value in the content-adaptive snapshot (Line 11). Secondly, it performs an atomic snapshot as described above (Line 12), and writes it into the pile (Lines 13). Finally the process updates its value in the content adaptive snapshot to $-$ (Line 14). Thus, the complexity of later updates and scans is not effected by it. Because snapshots are comparable and later snapshots have a bigger Σ part attached to them, only the latest snapshot will be returned by subsequent picks from the pile.

The pile object is implemented as follows: To throw a new value into the pile, process p first acquires a unique place in a shared array (called the *core array*) using adaptive renaming (Line 15). We use the long-lived and adaptive $2k^2$-renaming algorithm in Section 4.2 whose step complexity is $O(k^3)$ adaptive to point-contention. The process writes its scan result in its now dedicated slot in the array only if its scan result is newer than the snapshot already written in the slot (Line 16). This way data is never overwritten since the renaming guarantees that there are no concurrent writes to an entry in the array.

The adaptive renaming procedure guarantees that the slot chosen for a process is at most $O(k^2)$ slots away from the top (slot 1) of the array. However, process p may not yet terminate because if all the processes finish their update and a single process starts to query the information from the top of the pile in isolation, it may take $O(k^2)$ steps and

Algorithm 13 Code for adaptive snapshot for process p.

```
   Shared:
       S, a content adaptive CA-snapshot object;
       P, a pile;

   procedure scan() returns ⟨sum, dataset⟩
1:     D':=S.CA-scan();                          // |D'| is bounded by the contention.
2:     ⟨Σ, D⟩:=P.pick();
3:     for every ⟨p', v', seq'⟩ ∈ D' do
           // Recalculate D and Σ. For every p' take latest entry either from D or D'.
4:         if ∃⟨p'', v'', seq''⟩ ∈ D s.t., p'' = p' then
5:             if seq' > seq'' then
6:                 D := D ∪ {⟨p', v', seq'⟩} \ {⟨p', v'', seq''⟩};
7:                 Σ := Σ + seq' − seq'';
           else                                          // no tuple in D for p'
8:             D := D ∪ {⟨p', v', seq'⟩};
9:             Σ := Σ + seq';
       od;
10:    return(⟨Σ, D⟩);

   procedure update(data)                                // data ≠ −
11:    S.CA-update(data);                                // register
12:    ⟨Σ, D⟩:=scan();
13:    P.throw_in(⟨Σ, D⟩);
14:    S.CA-update(−);                                   // check-out
```

not a constant number of steps as it should. Therefore, after p wrote its snapshot result in its associated place in the core array, it starts a bubble up procedure (Lines 17-20) from its place in the array to the top. In the bubble up process the most recent snapshot in the array is percolated up to the top.

In the bubbling up process the updating process goes through the entries of the array from the index it had captured to the top. In each entry it recursively selects the most recent snapshot it observes in the bottom part of the array and records it with its current entry (Line 20). To do that we associate with each entry a pointer, called *last*, that points to the single-writer-multi-reader register of the last process that has recorded a snapshot with this entry. The process performs the bubbling up through an entry in a particular order: First it writes $-$ in its single-writer-multi-reader register associated with this entry (called $C[p][i]$ for process p in entry i), secondly it writes its name into $last[i]$, and finally

Algorithm 14 Implementation of a pile object for process p.

```
    Shared:
        A[1 .. 2N^2], 2N^2 atomic MRMW registers,
                each initialized to ⟨0, ∅⟩;
        last[1 .. 2N^2], 2N^2 atomic MRMW registers of type pid;
        C[1 .. N][1 .. 2N^2], 2N^3 atomic SWMR registers,
                each initialized to ⟨0, ∅⟩;

    procedure throw_in(⟨Σ, D⟩)
15:     index:=rename(p);
16:     if A[index].Σ < Σ then A[index]:=⟨Σ, D⟩;
                                        // don't overwrite newer snapshots.
17:     for i := index to 1 do                          // Bubble up.
18:         C[p][i]:=−;
19:         last[i] := p;
20:         C[p][i]:=choose(i);
21:     release_name(p);

    procedure pick() returns ⟨sum, dataset⟩
22:     return(choose(1));

    procedure choose(i) returns ⟨sum, dataset⟩          // Latest in pile.
23:     q := last[i];
24:     tmp:=C[q][i];
25:     if tmp ≠ − then return tmp;
26:     ⟨Σ′, D′⟩:=A[i];
27:     ⟨Σ, D⟩:=choose(i + 1);
28:     if Σ ≥ Σ′ then return ⟨Σ, D⟩;
29:     else return ⟨Σ′, D′⟩;
```

recurses to select a snapshot from the bottom part and records its result in $C[p][i]$.

This particular order of recording information in each entry guarantees the following property: If a process q reads $last[i] = p$ and subsequently it reads $C[p][i] = snapshot$ then any update that has recorded information below entry i in the array and has terminated before q reads $last[i]$ is included in the *snapshot*. Furthermore, if q observes $C[p][i] = -$ then it knows process p is concurrent with its operation and by the adaptiveness it may now perform a few more operations, in particular recursively choosing from snapshots at entry $i + 1$ (Line 27 in the code).

To select a snapshot from the pile process p starts at the beginning of the array (choose($i = 1$)), comparing snapshots as it goes down. It is imperative that the number of slots the choosing process descends to depends on the point contention it encounters. In the absence of contention it finds the latest snapshot at the first entry of the array, in one of the SWMR registers associated with this entry (Line 25). If not, the process recursively performs choose(2) (Line 27) and then chooses between the result and the snapshot written in slot 1 of the core array (Lines 28, 29).

The step complexity of the content adaptive operations is $O(k^4)$. Since the name process p gets from the renaming algorithm is at most $2k^2$, and the choose() procedure takes at most $O(k^2)$ the total step complexity of the update() and the scan() is at most $O(k^4)$.

9.2 Correctness

9.2.1 Pile

A pair, $s_\alpha = \langle \Sigma, D \rangle$, is passed as parameter to any execution α of procedure throw_in(), where Σ is an integer. Given a set R of procedure throw_in() executions, we say that a pair s_α is in R, denoted $s_\alpha \in R$, if $\alpha \in R$. We denote by $Latest_R$ a pair in R with the highest Σ part. That is, $Latest_R = \{s | s \in R, \forall s_\alpha \in R \; s.\Sigma \geq s_\alpha.\Sigma\}$. For any two pairs, s_1, s_2 we say that s_1 is *larger or equal* to s_2 (denoted $s_1 \geq s_2$) iff $s_1.\Sigma \geq s_2.\Sigma$. Given an execution, g, of choose() we denote by s_g the pair returned by g.

Given an execution α of throw_in(), we define the *entry slot* of α to be the index returned by the $2k^2$-rename algorithm in Line 15. Given an execution prefix α' of throw_in() with entry slot x, we say that α' crossed slot $x' \leq x$ if the process executing α' has already changed $last[x']$ to its id (Line 19), in α'.

Lemma 9.1 *Let α be an execution of the snapshot object, and let g be an execution of* choose(x) *contained in α. Assume that R_1 is the set of all* throw_in() *operations that started before the end of* choose(x) *and R_2 is the set of all* throw_in() *operations that crossed x before the beginning of* choose(x). *Then, $s_g \in R_1$ and $s_g \geq Latest_{R_2}$.*

Proof: We prove the claim by induction on the length of the prefixes of α.

The claim trivially holds for $\alpha_0 = e_0$ because no choose() operation can be contained in it. Assume that the claim is correct for prefix execution α_i. We show correctness for $\alpha_{i+1} = \alpha_i e_{i+1}$.

To effect the correctness of the claim event e_{i+1} must be a return operation (Lines 25, 28 or 29) of some choose(x) operation, denoted g, by some process p. Let $r \in R_2$ be the update by some process p' whose parameter was $Latest_{R_2}$.

1. Assume that $tmp \neq -$ (Line 25). Then p first read the name of some process, p'', from $last[x]$ and then read a pair from $C[p''][x]$, denoted s''. Denote by g'' the choose(x) operation performed by p'' that returned s'' (Line 20). From the assumption p' already crossed x before g started, that is, p'' wrote $-$ in $C[p][x]$ and then p' in $last[x]$. Therefore, p'' must either be equal to p', or be some process that crossed x after p'. Since p'' first erases its old value in $C[p''][x]$ (Line 18) and only then crosses x (Line 19) it follows, as $C[p''][x] \neq -$, that g'' was performed after p'' crossed x so after p' crossed x. By the induction hypothesis, $s'' \in R_1$ and $s'' \geq Recent_{R_2}$ so the claim holds.

2. Otherwise, g returns the bigger pair out of (1) the pair read in $A[x]$ or (2) the result of a choose($x+1$), denoted g'' (Lines 26-29). If the entry point of u was x then, as u crossed x before g started, it must have executed the if statement at Line 16 before g read $A[x]$. If the condition failed then $A[x]$ already contains a pair bigger than $Latest_{R_2}$. If the condition succeeds then, as the if statement asserts that pairs written into $A[x]$ monotonically increase, the pair read by g in $A[x]$ must be bigger than $Latest_{R_2}$. Except the initial pair, pairs read from $A[x]$ must be written by executions in R_1. But $Latest_{R_2} \geq \langle 0, \emptyset \rangle$ so the initial pair must have been overwirtten.

   If the entry point of u was greater than x then u must have crossed $x+1$ before g'' started so by induction the pair returned from g'' must be in R_2 and bigger than $Latest_{R_2}$.

   Since the returned result of g is greater or equal then both $A[x]$ and the result of g'', and it is one of them, the claim holds.

■

9.2.2 Snapshot

Lemma 9.2 *The Σ part of every pair $\langle \Sigma, D \rangle$ returned by the scan operation equals the sum of the sequence numbers attached to all the values in D.*

Proof: We prove the lemma by induction on the events. Assume that the claim is correct for prefix execution α_i. We show correctness for $\alpha_{i+1} = \alpha_i e_{i+1}$, where e_{i+1} is the end of some scan() operation, *op*, by process p. By Lemma 9.1, the result of any choose() operation is a pair passed as parameter to some throw_in() operation. Since the only call to throw_in() is in Line 13, it follows that choose() operations return the result of some previous scan. By the induction the Σ part returned from pick() (Line 2) is the sum of the sequence numbers attached to values in D. The code in the loop (Lines 3-9) adds to Σ the sequence number of every element in D' that is added to D. It also subtracts the sequence numbers attached to elements in D that are removed. Therefore, after the loop Σ still contains the sum of the sequence numbers attached to elements in D and the claim holds (Line 10). ■

To prove linearizability we first define a representative operation for every update or scan operation, where representative operations are either CA-update() operations or CA-scan() operations. We use the total order among content adaptive snapshot operations (Lemma 8.5) to define a total order among the updates and scans.

Definition 9.1 *Let α be an execution of Algorithm 13 and let S be the set of all update and scan operations executed in α. For every operation $op \in S$ we define the* representative operation *of op as follows:*

1. *If op is an* update(*data*) *operation then its representative operation is the* CA-update(*data*) *operation contained in it (Line 11).*
2. *Assume that op is a scan operation and let ca_1 be the* CA-scan() *operation contained in it, D' its result (Line 1). Let D be the result of the* pick() *operation (Line 2). Notice that D is either equal to $\langle 0, \emptyset \rangle$ or it is the result of some previous scan operation, op'.*
   (a) *If $D = \langle 0, \emptyset \rangle$ then ca_1 is the representative operation of op.*
   (b) *Otherwise, let rep' be the representative operation op'. Then the representative operation of op is the later operation between ca_1 and rep'.*

Lemma 9.3 *Let $op \in S$, and assume that rep is its representative operation. Then rep ends before op.*

Proof: For update operations the lemma immediately holds since the representative operations of an update is contained in the updates. Assume that *op* is a scan operation. We proof the lemma by induction on the termination of the scan operations in α. If *op* is the first scan operation to terminate in α then the pick() operation (Line 2) during *op* must have returned $\langle 0, \emptyset \rangle$, because update operations must first finish a scan before throwing a value in the pile (Lines 12,13). It follows that *rep* must be the CA-scan()

operation contained in op and the lemma holds. Assume that the lemma holds for all terminated scan operations $op_1, op_2, \ldots, op_i$ in some prefix of α and let op be the next scan operation to terminate. If rep is the CA-scan() operation contained in op the the lemma immediately holds. Otherwise, rep is the representative of some scan operation that wrote the view read from the pile by op. That scan operation must have already finished so by the induction hypothesis, rep must have already finished and the lemma holds. ■

Lemma 9.4 (linearizability) *Let α be an execution of Algorithm 13 and let S be the set of all update and scan operations executed in α. Then there exists a total order, $<_S$, between the operations in S s.t,*

1. *If operation op_0 completes before operation op_1 starts then $op_0 <_S op_1$.*
2. *If u is the last update operation of some process p ordered before a scan operation op_1 then op_1 returns the update value of u for process p.*

Proof: We define $<_S$ to be total order that exists between the operations' representatives, $<_G$ (Lemma 8.5), and show that $<_S$ satisfies the stated properties.

We prove the claim by induction. Let $\alpha = e_0, e_1, \ldots, e_i, e_{i+1}, \ldots$ and let the prefix α_i of α be $(e_0, \ldots, e_i)$, the sequence of the first $i+1$ events of α. Let S_i be the set of all finished update and scan operations in α_i, $<_{S_i}$ the total order among the representatives of the operations in S_i.

At the beginning of the algorithm ($\alpha_0 = e_0$) there are no finished update and scan operations so the claim trivially holds.

Assume that the claim is correct for prefix run α_i. We show correctness for $\alpha_{i+1} = \alpha_i e_{i+1}$. The event e_{i+1} can effect the correctness of the claim only if it is the end of some operation. We assume this operation was performed by process p and denote this operation as op_1 and its representative as rep_1.

To prove Part 1 of the claim we show that the representative, rep_0 of every operation op_0 that finishes before op_1 starts is ordered before rep_1.

By Lemma 9.3, rep_0 finishes before op_0 finishes. If op_1 is an update operation then rep_1 is the CA-update() operation contained in op_1 and since rep_0 finishes before op_1 it follows that $rep_0 <_G rep_1$. If op_1 is a scan operation then let ca_1 be the CA-scan() operation contained in it. By the definition of rep_1, $ca_1 \leq rep_1$. But since op_1 starts after op_0 finishes, $rep_0 < ca_1$ and Part 1 of the lemma holds.

Let u be the last update operation of some process p_u ordered before op_1, with update value v_u. Denote by rep_u the CA-update($data$) operation in Line 11 performed during u, by $scan_u$ the scan() operation in Line 12, by rep_{su} the representative of $scan_u$ and by $u\_$ the CA-update($-$) operation in Line 14. Denote by ca_1 the CA-scan() operation in Line 1 performed during op, by op' the scan operation whose view D was returned by the pick() operation in Line 2 (if there is such an operation) and by rep' the representative of op'.

By definition, $rep' \leq rep_1$. If $u_* < rep'$ then, as it is the last update of p_u before rep_1 it is also the last update by p_u before rep'. By the induction hypothesis, D includes v_u. D' can either contain v_u or have no value for p_u. In both cases, the claim holds.

Assume that $rep' < u_*$ and $ca_1 < u_-$. By definition, $ca_1 \leq rep_1$. Since u is the last update ordered by p_u before op_1, u_* is the last content adaptive update ordered before rep_1 so also before ca_1. This means that D' must contain v_u. D may contain an older value for p_u but only the latest value from D and D' is chosen for every process (Line 5) so the claim holds.

Assume that $rep' < u_*$ and $ca_1 > u_-$. From the properties of the content adaptive snapshot, if u_- is ordered before ca_1 then u_- must have started before ca_1 finished, which means that u finished the throw_in() operation in Line 13 before op_1 started the pick() operation in Line 2. It follows from the properties of the pile object that the Σ part attached to D is at least as large as the Σ part attached to the result of $scan_u$, which by the inductive hypothesis means that $rep_{su} \leq rep'$. By the previous cases, $scan_u$ must contain v_u which means that $u_* < rep_{su}$ in contradiction to the assumption.

■

Properties I-IV follow from the linearization.

9.3 Complexity Analysis

Lemma 9.5 *The step complexity of any* choose() *operation is* $O(k^2)$.

Proof: Let g be some choose(x) operation by p with point contention k and denote by $g_1, g_2, \ldots$ the recursive calls by p to choose($x+1$),choose($x+2$),.... Assume that g_i is not the last recursive call (it calls g_{i+1}) and that $x + i > 2k^2$. Because g_i is not the last call process p must have read a process id, p', in $last[x+i]$ and then found that $C[p'][x+i] = -$ (Lines 23-25). This means that process p' was concurrently executing an update_pool() operation, u, with op. Process p' wrote $-$ in $C[p'][x+i]$ and did not yet write another value before the read operation of g_i (Lines 18-20). It follows that the entry slot of p', denoted x', was such that $x' \geq x+i > 2k^2$. Therefore, u must have higher point contention than g so must have started before g. But there can only be k such concurrent update_pool() operations with g so some g_i operation s.t., $2k^2 < x+i < 2k^2+k+2$ must find a non $-$ value in C and terminate the recursion. The number of steps in every part of the recursion is constant so, as g may make up to $O(k^2)$ recursive calls, the step complexity of g is $O(k^2)$. ■

The renaming algorithm in the update_pool() procedure (Lines 15, 21) takes $O(k^3)$ steps. Since the entry point of the algorithm is at most $2k^2$ deep, it calls choose() at most $2k^2$ times in the loop (Lines 17-20), which makes its step complexity $O(k^4)$.

Lemma 9.6 *The step complexity of any* update_pool() *operation is* $O(k^4)$.

Although the update and scan procedures use a content adaptive snapshot object, every update operation ends with a CA-update(−) operation. Therefore, the step complexity of using the CA-snapshot object depends only on the number of concurrent scan and update operations, which in this context makes it truly adaptive. Accessing the object may take $O(k^4)$ steps, just as accessing the pool. Other than using the CA-snapshot object and the snapshot pool, the scan operation iterates over the entries in the CA-snapshot, which may take at most k steps. Therefore, the step complexity of the scan is $O(k^4)$. The update operation just calls the scan operation (as well as accessing the snapshot pool and the CA-snapshot object) so, again, its step complexity is $O(k^4)$.

Theorem 9.7 *Algorithm 13 is a valid implementation of a snapshot object (as defined in Definition 2.9). It is adaptive to point contention with step complexity* $O(k^4)$.

Chapter 10

Immediate Snapshot

Adaptive immediate snapshot ([BG93]) algorithms were presented in [GK98] and in [AF99b] (based on [AW99] which appears also in [Bor95]). However, these algorithms are only adaptive to total contention, which means that every process that had ever took steps during the algorithm is counted in the contention.

We transform the algorithm of [AF99b] into an adaptive to point contention algorithm with the following changes. First, we change all the collects and snapshots used in [AF99b] to their point contention adaptive counterparts. Then, we change the proximate snapshot algorithm by replacing the $Suggest_i$ collect object per process of [AF99b] with an unbounded vector of collect objects per process. This is necessary so process i would not read values suggested to it before it even started executing, which would have made the algorithm not adaptive. Last, we change the estimation of *start_level*, the parameter that is passed to 1s-immss(), to be a function of the point contention. This ensures that the executions of 1s-immss() are adaptive to point contention.

The algorithm in [AW99, AF99b] is implemented by an infinite number of floors, where each floor represents a possible snapshot of the values written by the different executions of the immediate-snapshot (each floor corresponds to one atomic write operation). It is assumed that the values written by every process in its different immediate-snapshot executions are increasing monotonically, i.e., sequence numbers (see [AW99, AF99b]). In principal, to choose a floor process p writes its current sequence number, scans all the sequence numbers and sums the result. The resulting value is the floor chosen by p (which we call the *entry floor* for process p). By the snapshot properties, all the processes that enter at the same floor agree on the scan result. After p chooses a floor it writes the scan result in that floor and then it starts descending the floors one by one. In each such floor (which we call a *participating floor* for process p) it participates in a distinct one-shot immediate snapshot algorithm. This procedure terminates when process p reaches a floor that contains a snapshot which does not contain the current sequence number of p. Process p terminates at this floor (called the *terminating floor* for p) and returns the

union of the snapshot associated with this floor and the result of its one-shot immediate snapshot executed at this floor. If there are two possible values for p, one from each source, then the value from the one-shot immediate snapshot result is taken.

Several changes are needed to make this algorithm adaptive to point contention. First, we need to limit the number of participating floors for process p (floors that process p may descend through). A proximate-snapshot object was presented in [AW99, AF99b] for this purpose. This object supports a single operation, px-upscan(seq), which writes seq and then returns a snapshot of the current values (sequence numbers) in the object. The object satisfies the following property: Assume that k processes concurrently perform the px-upscan(seq) and let p be one of them. Then the return snapshot of p may contain at most k values whose write is serialized after p's write. That is, the snapshot returned by px-upscan() is serialized no more than k writes after p's write. Processes use the proximate-snapshot object instead of performing an update() and a subsequent scan(). From the property of the proximate-snapshot it follows that the maximal number of floors that a process may descend is $O(k)$. The proximate-snapshot algorithms presented in [AW99, AF99b] are not adaptive to point contention. Later in the section we show how to implement an adaptive to point contention proximate-snapshot object (Algorithm 15).

Another change needed is the summation of the sequence numbers. Since a proximate-snapshot may return N values it is impossible to sum them in an adaptive way. Our implementation of the proximate-snapshot returns Σ, the sum of the counters in the snapshot, with the snapshot result. In fact, no counter values are passed to our proximate-snapshot implementation. Instead, we use the internal sequence numbers managed by our snapshot implementation (which in turn uses the sequence numbers of the CA-snapshot, Section 8.2). Due to the way the adaptive snapshot is implemented, the sequence number for process p is incremented by two with every execution of the proximate-snapshot operation.

It follows from Corollary 10.7 that the total number of processes that participate in the same floor with process p during its immediate snapshot operation op is at most k, the point contention during op. In each floor process p participates, it accesses one single-shot collect object and one single-shot immediate snapshot algorithm, both dedicated to the floor. Since we use adaptive collect objects, the collects and updates of the collect object at each floor are bounded by a function of k. The one shot immediate snapshot algorithm (Algorithm 16) is taken as is from [AF99b] (based on [BG93]). In this one-shot algorithm, process p descends through the levels one by one, as in [BG93], until it reaches a level t s.t., there are t or more processes at levels $1 \ldots t$. At this point the process can terminate and return the values of all the processes at levels $1 \ldots t$ as the result of its immediate snapshot. The advantage of the algorithm in [AF99b] is that the processes do not have to start at level N (the total number of processes in the system) but can start at any level, according to a parameter ($start\_level$). Therefore, the step complexity of the algorithm depends on the $start\_level$ parameter passed in each call. It

Algorithm 15 Code for point contention adaptive proximate snapshot for process p. Let V be a view. Then, abusing notation, we denote by $V[p]$ the sequence number in V associated with p.

```
    Shared variables:
        Suggest[1 .. N][∞], An unbounded array of
            fully adaptive collect objects for each process;
        F[1 .. N][∞], An unbounded array of bits for each process, all initialized to false;
        A, an active set object;
        S, an adaptive snapshot object;

    procedure px-upscan() returns ⟨sum, dataset⟩
1:      A.joinSet();                                          // register
2:      S.update(dummy);                          // Only the seq is important.
3:      ⟨Σ, V⟩:=S.scan();
        denote: c_p' ≡ Suggest[p'][V[p']];                  // current object for p'
                f_p' ≡ F[p'][V[p']];                          // current flag for p'
4:      T:=A.getSet();                                        // Phase 1
5:      for every p' ∈ T do
6:          if f_p' = false then
7:              f_p' := true;
8:              c_p'.write(⟨Σ, V⟩);
9:      T := c_p.collect();                                   // Phase 2
10:     ⟨Σ, V⟩:= element in T with minimal Σ part;
11:     A.leaveSet();                                         // check-out
        return(⟨Σ, V⟩);
```

Algorithm 16 Code for restricted one shot immediate snapshot for process p

```
    procedure 1s-immss(c, start_level:integer)
        level := start_level;
        store(⟨id_p, c, level⟩);
        while (true) do
            level := level − 1;
            store(⟨id_p, c, level⟩);
            V:=collect();
            W:={⟨id_j, c_j, level_j⟩ ∈ V | level_j ≤ level};
            if ( |W| ≥ level) then return (W);
```

is only required that *start_level* is bigger than the total number of processes that may actually access it (for a proof see [AF99b]), which in this case is equal to k, the point contention. To get an adaptive immediate-snapshot object we need to correctly estimate this number for each floor.

To estimate *start_level*, the number of processes that may participate in the same floor, f, with process p in its execution *op*, we use one long lived active set object, $\mathcal{A}$, and an unbounded number of single shot collect objects, one per floor. Immediate snapshot operations that are concurrent with *op* are detected using $\mathcal{A}$, as processes register in $\mathcal{A}$ at the beginning of their immediate snapshot operation and check-out at the end (Lines 12 and 24). It follows from Lemma 10.6 that operations that start after p collects the active set (Line 16) do not participate in the same floors as *op*. There may have been operations that were concurrent with *op* and accessed f, but have finished before p performed its getSet() operation. Because every process registers itself in the single shot collect object in each floor it participates in (Line 19), such operations can be detected for each floor. The sum of the sizes of the active set and the collect associated with floor f is an upper bound on the number of processes that may participate in the one-shot immediate snapshot in this floor. Processes may calculate different estimations for the same floor, f. Still, all those estimations are bigger than the total number of processes that accesses the floor.

The implementation of the immediate-snapshot() operation is presented in Algorithm 17. Process p starts the immediate snapshot operation by joining the set of currently active processes using an active set object (Line 12). Then, it performs a proximate-snapshot operation to get a view, V. The proximate-snapshot operation returns both the view and the sum of its counters, which is used by p as its entry floor (Line 13). Process p then extracts its own sequence number (Line 14), writes the view in its floor (Line 15) and calculates an initial estimate on the number of processes that may access any floor that p accesses. This estimation is the number of processes currently registered in $\mathcal{A}$ (Line 16). Then, process p starts descending the floors one by one (Lines 18-25). In each floor the process registers itself in the floor's collect object (Line 19), evaluates the number of registered processes in the floor (Line 20) and then participates in the floor's one shot immediate snapshot (Line 22). However, the process also marks in a dedicated flag for p in f if it has seen a view in the floor (Line 21). Now process p must decide if it should terminate or iterate. The terminating condition consists of two parts (Line 23). The process checks that (1) the sequence value associated with it in the view written in floor f is smaller than its current sequence number and (2) at least one of the processes that are in W, the result of the one shot immediate snapshot, has set its dedicated flag to **true**. In such a case, process p checks out from $\mathcal{A}$ (Line 24) and returns the merged results of W and the view in f (Line 25).

It remains to be seen how to implement the proximate snapshot object in a point

Algorithm 17 Code for adaptive immediate snapshot for process p. The object satisfies Properties I-V.

```
    Shared:
        A, an active set object;
    Local variables:
        V, U, W: view;
        f, start_level:integer;
    procedure im-upscan()
12:     A.joinSet();
13:     ⟨f, V⟩:=px-upscan();
14:     count := V[p].seq;                          // Find out own sequence number.
15:     view[f]:=V;
16:     start_level:=|A.getSet()|;
17:     while (true) do
18:         f := f − 1;
19:         update_f(p);
20:         curr_level := |collect_f()|;
21:         flag[f][p] := (view[f] ≠ −);
22:         W:=1s-immss_f(count, start_level + curr_level);
23:         if (count > view[f][id_p] and
                        for some j ∈ W, flag[f][j] =true) then
24:             A.leaveSet();
25:             return(join(W, view[f]));

    procedure join(V_1, V_2:view)                   // V_2 is updated with the entries of V_1
        for every ⟨p', v'⟩ ∈ V_1 do
            if ∃⟨p', v''⟩ ∈ V_2 then V_2 := V_2 − ⟨p', v''⟩;   // if old entry exists, remove it
            V_2 := V_2 + ⟨p', v'⟩;
        return(V_2);
```

contention adaptive way. The proximate-snapshot implementation is presented in Algorithm 15. It uses an unbounded vector of collect objects for each process p ($Suggest[p][\infty]$), where eventually entry $Suggest[p][2l]$, denoted c_p^l, contains the set of possible results for the l'th px-upscan() of p. And upon terminating, p chooses the earliest entry in c_p^l.

Process p starts by registering in $\mathcal{A}$ (Line 1) and then makes a dummy update to the snapshot object (Line 2). The result of this update is that the sequence counter of p is incremented by two, because of the way the snapshot object is implemented. Then, the process executes a scan() operation (Line 3), which returns a view, V, and the sum of its counters, Σ.

Process p now progresses in two phases. In the first phase the process suggests its scan result to the other (active) processes. For this, the process gets the set of processes in $\mathcal{A}$ (Line 4) and then, for every process p' in this set, it accesses the flag bit, $f_{p'}^l$, in $F[p'][V[p']]$ and the collect object, $c_{p'}^l$, in $Suggest[p'][V[p']]$, where $V[p']$ is the value associated with p' in V (Lines 6-8). Process p uses the flag to decide weather it should add its snapshot result to $c_{p'}^l$. Process p sets $f_{p'}^l$ to **true** and adds its snapshot result to $c_{p'}^l$ only if no previous views were written to $c_{p'}^l$ ($f_{p'}^l$=**false**). This way process p does not update $c_{p'}^l$ twice and furthermore, the number of entries in $c_{p'}^l$ is bounded by k, the point contention.

In the second phase, process p collects all the suggestions made to it in its current execution (Line 9), which must have been written in c_p^l. It then chooses the earliest view between the suggestions (Line 10). Once such a view is chosen (one must be chosen because V contains the current sequence number) the process checks-out from $\mathcal{A}$ (Line 11) and returns.

Most of the lemmas and proofs in the following subsections are taken from [AF99b].

10.1 The Proximate Snapshot Algorithm

The pairs returned by the algorithm are the result of scans of $\mathcal{S}$. Therefore, the views satisfy the properties of atomic snapshots (Properties I-IV). Furthermore, the Σ part of every pair is the sum of all the sequence number in the view it is attached to.

Let $\langle \Sigma_p^l, V_p^l \rangle$ be the pair returned by the lth px-upscan() operation, op, of process p. Let S_p^l be the view of $\mathcal{S}$ immediately after p completes the update operation, u, in Line 2, of op. Denote by Σ_S the sum of sequence numbers in S_p^l.

Lemma 10.1 *If op and op' are two* px-upscan() *operations by processes p and p' respectively, and op finishes before op' starts phase 1 (Line 4) then there is no entry in $Suggest[][]$, c_x, that both operations update.*

Proof: Assume to contradiction that op and op' update the same collect object, c_x, in the *Suggest* array. Then both access it in Line 8. If op updated c_x, then it also set the

entry in F[][] associated with c_x, denoted f_x, to **true** (Line 7). The flags are only set once to **true** and never change back. Therefore, the test in Line 6 during op' must have failed which means that op' did not update c_x in contradiction to the assumption. ■

Corollary 10.2 *Two non concurrent operation do not update the same entries in Suggest*[][].

Lemma 10.3 *For every process p and every $l \geq 0, \Sigma_S \leq \Sigma_p^l \leq \Sigma_S + 2k$.*

Proof: Denote by c_p^l the collect object of p in $Suggest[p][]$, from which V_p^l was chosen (Line 10), f_p^l its associated flag in $F[p][]$. Object c_p^l is only updated by processes that have seen the update by u (Line 8). The scan of op must have returned the update of u, so there is at least one such process. Furthermore, either p wrote in c_p^l, or f_p^l was set to **true**, which means that some other process already wrote in c_p^l. In any case, there is a minimal entry in the set and, as all the scan results that return the update of u must have been linearized after u, $\Sigma_S \leq \Sigma_p^l$.

Assume by way of contradiction that $\Sigma_p^l > \Sigma_S + 2k$. Since there are at most $k-1$ active processes (besides p), there is some process q which performed more than one update after u and before the scan operation of op. Let $u_1, u_2, \ldots, u_x$ be such updates of q, $op_1, op_2, \ldots, op_x$ their corresponding px-upscan operations, and assume that V_p^l returned the update value of u_x ($V_p^l[q] = u_x$). Surly update u_x have finished before op collected c_p^l (Line 10). But then op_1 must have finished too. Update u_1 is linearized after u, so the scan in op_1 (Line 3) must include the update value of u, and the getSet() operation (Line 4) must return p. Update u_x was performed after op_1 finished, so every operation op' that read the update value of u_x in its scan (Line 3) did not start Phase 1 until op_1 was finished. If op_1 did not update c_p^l then it must have read **true** in f_p^l which means that op' also must read **true** in f_p^l and not update c_p^l. If op_1 updated c_p^l (Line 8) then by Lemma 10.1 op' did not update c_p^l. Both cases contradict the assumption that $V_p^l[q] = u_x$. ■

The step complexity of the px-upscan() is dependent on the step complexity of accessing the CA-collect object $\mathcal{C}$. But since every process writes $-$ to $\mathcal{C}$ before terminating, the step complexity of the px-upscan() is adaptive to point contention. Let op be a px-upscan() operation performed by process p. The snapshot calls during op (Lines 2, 3) may take $O(k^4)$ steps. Phase 1 (Lines 4-8) performs k iterations. In each iteration the condition is checked and at most one collect operation is performed. From Corollary 10.2 this operation takes at most $O(k^3)$ steps. It follows that the step complexity of Phase 1 is $O(k^4)$.

During Phase 2 (Lines 10), process p collects data from its dedicated collect object, c_p^l, and then iterates on the entries to find a minimum. By Lemma 10.1, all the processes that update c_p^l were concurrent, so iterating over the entries takes $O(k)$ steps. Therefore, Phase 2 takes at most $O(k^3)$ steps.

Theorem 10.4 *Algorithm 15 solves the proximate snapshot problem, with $O(k^4)$ step complexity.*

10.2 A Restricted Algorithm for One-shot Immediate Snapshot

The code in Algorithm 16 is taken as is from [AF99b] so we skip the proof of the theorem. Note that if we use our implementation of a collect object instead of the single shot implementation of [AF99b] the algorithm becomes adaptive to point contention.

Theorem 10.5 *If of every process p_i, $start\_level_i > k$, then Algorithm 16 solves the one-shot immediate snapshot problem; p_i performs $O(k^3 \cdot start\_level_i)$ steps.*

10.3 The Immediate Snapshot Algorithm

Lemma 10.6 *Let op be an* im-upscan() *by process p with entry floor sf_p. Assume that op' is an* im-upscan() *operation by process p' that started after op updated its scan result in floor sf_p (Line 15), $sf_{p'}$ its corresponding entry floor. Then $sf_{p'} > sf_p$ and op' does not descend to levels below sf_p.*

Proof: The px-upscan() call in operation *op* finished before the px-upscan() in op' started and therefore cannot include the new sequence number of op'. By the snapshot property of the px-upscan() and by Lemma 9.2 it follows that $sf_p < sf_{p'}$. Assume that op' descended to level sf_p. Because *op* has already performed Line 15, $view[sf_p] \neq -$, so $flag[sf_p][p'] =$**true**. By the self inclusion property of the os-im-upscan() operation, $p' \in W$. Furthermore, $count > view[sf_p][p']$ so, op' must stop at level sf_p. ■

Corollary 10.7 *Two non-concurrent* im-upscan() *operations do not descend to the same floors. That is, they do not access the same floors in the loop (Lines 17-25)*

Let P_f be the set of processes that access floor $f \geq 1$. The following lemma states that the estimation passed to the os-im-upscan() in every floor is correct.

Lemma 10.8 *If process p starts an* im-upscan() *operation, op, at floor sf_p, and descends down to floor f then after p executes Line 20, $|P_f| < start\_level + curr\_level$.*

Proof: Assume by way of contradiction that there exists some process p' s.t., p' participates in the os-im-upscan$_f$ operation in floor f and is not counted for by *op* in *start_level* or *curr_level*. Let op' be the im-upscan() operation by p', $sf_{p'}$ its entry floor.

If p' performed the $\mathsf{update}_f(p')$ (Line 19) before p collected the set (Line 20) then surly p' is counted for in $curr\_level$. Assume otherwise, then if p' registered in $\mathcal{C}$ (Line 12) before p collected the set from $\mathcal{C}$ (Line 16) then again, it must be counted for. Therefore, process p must have performed the px-upscan() and wrote the result in sf_p (Lines 13-15) before process p' performed its px-upscan() operation. It follows that $sf_{p'} > sf_p$. Operation op' had to pass through level sf_p before it accessed level f. But, by Lemma 10.6, this is impossible which means that the lemma is correct. ■

The proofs of Lemmas 10.9-10.13 follow the proofs of the corresponding Lemmas in [AF99b], which in turn, follows [AW99] and [Bor95].

Lemma 10.9 *If process p returns a view V from floor f during operation op, and process p' performing its lth immediate snapshot, op' passes through floor f, then $2l \leq V[p']$.*

Proof: Process p stopped at floor f so the condition in Line 23 must have evaluated to **true** in op. Process p' did not stop at floor f and so the condition evaluated to **false** in op'.

(case 1) Denote by W_p^f the result of the one shot immediate snapshot (Line 22) at floor f during op, and by $W_{p'}^f$ the result during op'. Assume that p' reads **false** from $flag[f][x]$ for every process $x \in W_{p'}^f$. There must be some process q s.t., $q \in W_p^f$ and p read the flag of q to be **true**. Process q sets it flag to **true** or **false** only once, before participating in the $\mathsf{os\text{-}im\text{-}upscan}_f$ operation. Therefore, it must be that $q \notin W_{p'}^f$, and thus, $W_{p'}^f \subset W_p^f$. By the self inclusion property (Property V, Part 1), $\langle p', 2l \rangle \in W_{p'}^f \subset W_p^f$, hence $2l \leq V[p']$.

(case 2) p' reads a non $-$ value from $view[f]$ and $2l \leq view[f][p']$. Since p' read a non $-$ value from $view[f]$, p reads the same value and sees $2l \leq view[f][p'] \leq V[p']$.

(case 3) If p' read $-$ in $view[f]$ at Line 23 then every process that has started the $\mathsf{os\text{-}im\text{-}upscan}_f$ operation up to that point must have set its flag to **false**. Denote this set W, and note that $p' \in W$. Because p stopped in this level, there must be some process q s.t., $q \in W_p^f$ and $flag[f][q]$ was found to be **true**. This means that $q \notin W$ so it follows that $\langle p', 2l \rangle \in W \subset W_p^f$. ■

Lemma 10.10 *If p returns V_p from floor f_p and q returns V_q from floor $f_p < f_q$ then $V_p \preceq V_q$.*

Proof: For every process p' s.t., $V_p[p'] = -$ the claim is immediate. Assume that $V_p[p'] = seq$, where seq is the sequence number of p' in its im-upscan() operation, op'. Let $V_{p'}$ be the view returned by the px-upscan() operation call of op', and $sf_{p'}$ its entry floor. View V_q is the union of the view written in $view[f_q]$ and W (Line 25). If $sf_{p'} \leq f_q$ then $\langle p', seq \rangle \in V_{p'} \preceq view[f_q]$, so $seq \leq V_q[p']$. If $sf_{p'} > f_q$ then by Lemma 10.9 $seq \leq V_q[p']$. It follows that $V_p \preceq V_q$. ■

Assume that process p returns V_p from floor f_p and process q returns V_q from floor f_q. If $f_q \neq f_q$ then by Lemma 10.10, V_p and V_q are comparable. If $f_p = f_q$ then p and q retrieve the same (non empty) view from $view[f_p]$. The view $W_p^{f_p}$ and $W_q^{f_p}$, returned from the os-im-upscan() by p and q correspondingly, are comparable. Therefore, V_p and V_q are comparable, and Property IV holds.

Lemma 10.11 (Immediacy) *The returned views satisfy Property V*

Proof: The proof is exactly as in [AF99b]. We present it for completeness.

If process p returns V_p from floor f in its lth im-upscan() operation, then $\langle p, 2l\rangle \in W_p^f \preceq V_p$, by the self inclusion property of the os-im-upscan(). Therefore , Part 1 of Property V holds.

Assume that process q returned view V_q which contains the lth value written by p in operation *op*. Let V_p be the view returned by *op*. Since $V_q[p] = 2l$, $\langle p, 2l\rangle$ appears either in the view read by q from $view[f_q]$ or in the set $W_q^{f_q}$ returned from the os-im-upscan() operation.

If $\langle p, 2l\rangle \notin W_q^{f_q}$ then $\langle p, 2l\rangle \in view[f_q]$ so p returns from a floor below f_q and by Lemma 10.10 $V_p \preceq V_q$. If $\langle p, 2l\rangle \in W_q^{f_q}$, then in floor f_q, process p gets a view from the os-im-upscan(), $W_q^{f_q}$, s.t., $W_p^{f_q} \preceq W_q^{f_q}$ (by the immediacy of os-im-upscan). If p returns from floor f_q, then $V_p = \mathsf{join}(view[f_q], W_p^{f_q}) \preceq \mathsf{join}(view[f_q], W_q^{f_q})$. Otherwise, p returns from a floor below f_q and $V_p \preceq V_q$ by Lemma 10.10. ■

The result of every immediate snapshot operation is a union of the result of a px-upscan() operation and a os-im-upscan() operation (Line 25). Both operations satisfy Property I (validity) so their union also satisfy it.

The proofs of the next two Lemmas can be taken word by word from the proofs of Lemmas 5.11,5.12 in [AF99b].

Lemma 10.12 *After p writes into $view[sf_p]$, there is no consecutive sequence of $k+1$ empty entries in $view[1], \ldots, view[sf_l]$.*

Lemma 10.13 *During operation* im-upscan(), *op, by process p, the process descends through at most $2k$ floors.*

The im-upscan() performs at most $O(k)$ calls to the os-im-upscan(), where the number of steps taken by the operation in each call is $O(start\_level + curr\_level)$. It is easy to see that $start\_level \leq k$, because it counts only active processes. From Corollary 10.7 it follows that two operations descend into the same floor only if they are concurrent. It follows that $curr\_level \leq k$ so the step complexity of any os-im-upscan() operation is $O(k)$.

Theorem 10.14 *Algorithm 17 is a valid implementation of an immediate snapshot object (as defined in Definition 2.9). Furthermore, it is adaptive to point contention with step complexity $O(k^4)$.*

Chapter 11

Discussion

In this thesis we present diverse long-lived and adaptive algorithms in the shared memory model. Several questions remain unanswered. For example, is it possible to implement a fully adaptive renaming algorithm? Also, an adaptive to point contention implementation of a splitter would be desirable.

An interesting research area is to define adaptivity for other models of computation such as for the message passing model or for random algorithms. Finally, ideas from this field have been used to improve the complexity of basic algorithms (e.g., [AB00]). It is interesting to see if these ideas can also improve the implementation of real computer systems.

Bibliography

[AAD+93] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[AAF+99a] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Adaptive long-lived renaming using bounded memory. Submitted to DISC99. ftp://ftp.math.tau.ac.il/pub/stupp/PAPERS/name99.ps.gz, 1999.

[AAF+99b] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proc. 18th Annual ACM Symp. on Principles of Distributed Computing*, pages 91–103, May 1999.

[AB00] Hagit Attiya and Vita Bortnikov. Adaptive and efficient mutual exclusion. In *PODC2000*, 2000.

[ABND+87] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asynchronous environement. In *Proc. of the 28th IEEE Ann. Symp. on Foundation of Computer Science*, pages 337–346, October 1987.

[ABND+90] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, July 1990.

[ABT00] Yehuda Afek, Pazi Boxer, and Dan Touitou. Bounds on the shared memory requirements for long-lived & adaptive objects. In *PODC2000*, 2000.

[ADT95] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proc. of the 27th Ann. ACM Symp. on Theory of Computing*, pages 538–547, May 1995.

[AF98] H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proc. 17th Annual ACM Symp. on Principles of*

Distributed Computing, pages 277–286, June 1998. Extended version available as Technion Computer Science Department Technical Report #0931, April 1998.

[AF99a] H. Attiya and A. Fouren. Adaptive long-lived renaming with read and write operations. Technical Report 0956, Faculty of Computer Science, Technion, Haifa, 1999. http://www.cs.technion.ac.il/~hagit/pubs/tr0956.ps.gz.

[AF99b] Hagit Attiya and Arie Fouren. An adaptive collect algorithm with applications. Unpublished manuscript, 1999.

[AK99] James H. Anderson and Yong-Jik Kim. Fast and scalable mutual exclusion. In *DISC99*, 1999.

[AM94] J. H. Anderson and M. Moir. Using k-exclusion to implement resilient, scalable shared objects. In *Proc. of the 13th ACM Symposium on Principles of Distributed Computing*, pages 141–150, August 1994.

[AM98] Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. Manuscript, June 1998.

[AR93] H. Attiya and O. Rachman. Atomic snapshot in $o(n \log n)$ operations. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 29–40, August 1993.

[AST99] Y. Afek, G. Stupp, and D. Touitou. Dynamic fast and long-lived renaming. Technical Report TR 335/99, Computer Science Department Tel-Aviv University, 1999. http://www.math.tau.ac.il/~stupp/ARCHIVE/tr-33599.ps.gz.

[AT92] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, December 1992.

[AW98] Hagit Attiya and Jennifer Welch. *Distributed Computing Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill, 1998.

[AW99] Yehuda Afek and Eytan Weisberger. The instancy of snapshots and commuting objects. *Journal of Algorithms*, 30(1):68–105, January 1999.

[BG93] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 41–51, August 1993.

[BGHM95] Harry Buhrman, Juan A. Garay, Jaap-Henk Hoepman, and Mark Moir. Long-lived renaming made fast. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 194–203. ACM, August 1995.

[BND89] A. Bar-Noy and D. Dolev. Shared memory vs. message-passing in an asynchronous distributed environment. In *Proc. of the 8th Ann. ACM Symp. on Principles of Distributed Computing (PODC)*, pages 307–318, August 1989.

[Bor95] Elizabeth Borowsky. *Capturing the Power of Resiliancy and Set Consensus in Distributed Systems.* PhD thesis, University of California, Los Angeles, 1995.

[BP89] J. E. Burns and Gary L. Peterson. The ambiguity of choosing. In *Proc. of the 8th ACM Symp. on Principles of Distributed Computing*, pages 145–158, Edmonton, Alberta, Canada, August 1989.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems*, 3(1):63–75, January 1985.

[CM82a] K. M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributred systems. In *Proc. of the ACM Symp. on Principles of Distributed Computing*, pages 157–164, August 1982.

[CM82b] K. M. Chandy and J. Misra. Termination detection of diffusing computations in csp. *ACM TOPLAS*, 4:37–42, 1982.

[CS93] M. Choy and A. K. Singh. Adaptive solutions to the mutual exclusion problem. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 183–194, August 1993.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications Of The ACM*, 8(9):569, September 1965.

[Fra80] N. Francez. Distributed termination. *TOPLAS*, 2,1:42–55, January 1980.

[Gaf92] E. Gafni. More about renaming: Fast algorithm and reduction to the k-set test-and-set problem. Unpublished manuscript, 1992.

[GK98] E. Gafni and E. Koutsoupias. On uniform protocols. Extended Abstract, November 1998.

[GMS92] J. R. Gilbert, C. B. Moler, and R. Schreiber. Sparse matrices in MATLAB : Design and implementation. *SIAM J. Matrix Anal. and Appl.*, 13(1):333–356, 1992.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HS93] M. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proc. 25th ACM Symp. on Theory of Computing*, May 1993.

[Knu66] Donald E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9:321–322, May 1966.

[Lam74] L. Lamport. A new solution of *dijkstra's* concurrent programming problem. *Communications Of The ACM*, 17:453–455, 1974.

[Lam86a] L. Lamport. The mutual exclusion problem, part I: A theory of interprocess communication. *J. of the ACM*, 33(2):313–326, April 1986.

[Lam86b] L. Lamport. The mutual exclusion problem, part II: Statement and solutions. *J. of the ACM*, 33(2):327–348, April 1986.

[Lam86c] L. Lamport. The mutual exclusion problem, parts I and II. *J. of the ACM*, 33(2):313–384, April 1986.

[Lam86d] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1:77–101, 1986.

[Lam87] L. Lamport. A fast mutual exclusion algorithms. *ACM Trans. on Computer Systems*, 5(1):1–11, February 1987.

[LS92] N. Lynch and N. Shavit. Timing based mutual exclusion. In *Proc. of the 13th IEEE Real-Time Systems Symp.*, pages 2–11, December 1992.

[LT87] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. of 6th ACM Symp. on Principles of Distributed Computation*, pages 137–151, August 1987. Expanded version available as Technical Report MIT/LCS/TR-387, April 1987.

[MA95] M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995. Also in Proc. 8th Int. Workshop on Distributed Algorithms, September 1994, 141-155.

[MG96] Mark Moir and Juan A. Garay. Fast, long-lived renaming improved and simplified. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, October 1996.

[MM] D. Menasce and R. Muntz. Locking and deadlock detection in distributed data bases. *IEEE Trans. on Software Engineering*, SE-5, 3:195–202, May 1979.

[Moi98] M. Moir. Fast, long-lived renaming improved and simplified. *Science of Computer Programming*, 30, 1998. Also in Proceedings of the 10th International Workshop on Distributed Algorithms, Bologna, Italy, October 1996.

[MT93] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.

[Pet81] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[RST95] Y. Riany, N. Shavit, and D. Touitou. Towards practical snapshots. In *Proc. of the 3rd IEEE Israeli Symp. on Theory of Computing and Systems*, pages 121–129, January 1995.

[Sty92] E. Styer. Improving fast mutual exclusion. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, pages 159–168, August 1992.

[Tut87] M. Tuttle. I/o automata. Master's thesis, M.I.T., 1987.

[TUX90] TUXEDO. system release 4.0 - product review. Unpublished, 1990.