Public Data Structures: Counters as a Special Case*

Hagit Brit[†] Shlomo Moran[†‡] Gadi Taubenfeld[§]

June 5, 2001

Abstract

A public data structure is required to work correctly in a concurrent environment where many processes may try to access it, possibly at the same time. In implementing such a structure nothing can be assumed in advance about the number or the identities of the processes that might access it.

While most of the known concurrent data structures are not public, there are few which are public. Interestingly, these public data structures all deal with various variants of counters, which are data structures that support two operations: increment and read.

In this paper we define the notion of a public data structure, and investigate several types of public counters. Then we give an optimal construction of public counters which satisfies a weak correctness condition, and show that there is no public counter which satisfies a stronger condition. It is hoped that this work will provide insights into the design of other, more complicated, public data structures.

*A preliminary version of this work appeared in the *Proc. 3rd Israel Symposium on the Theory of Computing and Systems*, Tel Aviv, January 1995.

[†]Computer Science Department, Technion, Haifa 32000, Israel.

[‡]This reseraach was supported in part by the fund for promotion of the research in the Technion.

[§]The Open Univ., 16 Klausner st., P.O.B. 39328, Tel-Aviv 61392, Israel. Part of the work was done while the author was working for AT&T Bell Laboratories.

1 Introduction

Public data structures

The subject of concurrent data structures has been the focus of several recent works, which are motivated by the development of new parallel computers. A traditional implementation of a (sequential) data structure consists of the code for all the operations the data structure supports, which behaves correctly when all the operations are executed one after the other in a sequential fashion. An implementation of a concurrent data structure gives a code which must behave correctly even when executed by many processes concurrently.

Implementing a concurrent data structure is much trickier than a sequential one. It is usually required to be *wait-free*, that is, it should guarantee that any operation by a process will always be completed in a finite number of steps regardless of the behavior of other processes (such as abnormal termination). In implementing concurrent structures, one usually assumes that the total number of processes in the system, as well as the identities of these processes, are known. However, this assumption is not always valid: for instance, in common server-clients applications, the identities of the clients, and in some cases also their number, are not known a priori. Hence we define the notion of a *public data structure*. A public data structure is a concurrent data structure that is required to work correctly for any finite number of concurrent processes – nothing is assumed in advance about the number or the identities of the processes that might access it. Among the data structures studied in the literature, counting networks and concurrent counters [AHS94, MTY96] appear to satisfy the requirements of public data structures.

One way to implement a concurrent data structure, which is used in many practical applications, is first to implement it under the assumption that only one process may access it, and then to enforce sequentiality in accessing it using a mutual exclusion algorithm. That is, in order to access the structure, a process participates in a mutual exclusion algorithm that protects the structure, and accesses the structure only in its critical section. However, mutual exclusion algorithms are not wait-free, and a failure of a process in its critical section blocks any further access to the structure by other processes. Moreover, such a solution is not time-efficient since it does not allow concurrent access to the structure.

In this paper we focus on the construction of a simple public data structure: a public counter. A counter is a data structure that supports two operations: increment by 1, and read. While the meaning of these operations is obvious in a sequential context, it needs to be further clarified in a concurrent context. Previous works on concurrent counters considered two natural types of increment operations: a *weak* increment, which increments the counter but does not return a value, and a *strong* increment, which also returns the current value of the counter. Similarly, two types of read operations were studies: a *weak* read, which returns the correct value of the counter only if no increment operation is concurrent with it, and a *strong* read, which returns a meaningful value of the counter even when it is concurrent with increment or read operations.

Our main goal is to investigate the possibility of constructing public counters which count modulo some large number, from counters which count modulo smaller numbers. In particular, we show that this possibility depends on the correctness requirements from the constructed counters.

The computational model

Our model of computation consists of a collection of fully asynchronous identical deterministic processes that communicate via *atomic concurrent objects.* We model atomic concurrent objects by Mealy machines [HU79], where the input alphabet is the set of operations applicable to the object, and the output alphabet is the set of output values returned by the object. The objects are atomic in the sense that in every execution all the accesses to a given object are totally ordered in time. This assumption can be replaced by the more involved assumption that the objects are *linearizable*, in the sense of [HW90], without affecting our results. The atomic objects used in this paper will always be atomic counters (modeled by Mealy machines in a natural way), which support strong increment and strong read operations.

Depending on the context, we will either assume that the atomic counters are initially set to some default initial value (Section 2), or that initially the value of each atomic counter is arbitrary (Section 3).

Access to the atomic counters are via increment and read operations. We point out that, for example, read-modify-write registers of b values, where in a single indivisible step, it is possible to read the value in the register and then write a new value that can depend on the value just read, are stronger than counters which counts modulo b which support only the two simpler operations: increment and read.

A counter which counts modulo m, in short a *counter (modulo m)*, is a data structure which enables two basic operations: increment by 1 modulo m, and read. Each of these operations can be either weak or strong, in the sense defined in the introduction. We will distinguish between the term *increment step* and the term *increment operation.* An increment operation is the operation of incrementing the counter by one, and is performed on a counter which is possibly implemented using smaller atomic counters. An increment step is incrementing one of the atomic counters by one (i.e., an indivisible step in which a process increments the value of the atomic counter by one and get back its value). A similar distinction is made between *read step* and *read operation.* Thus, in order to complete one increment operation it is necessary to take one or more increment or read steps. A process that started executing an increment or read operation but has not completed it yet is *involved* in this operation. A process which is not involved in any operation is *idle.* A *run* of a public counter is a sequence of increment and read steps, performed by one or more identical processes. For runs x, y, $x \leq y$ means that x is a prefix of y. A *complete run* is a run in which all processes are idle. We consider three types of counters:

- A *static counter*: supports weak increment and weak read. An increment operation increments the value of the counter by 1 (and is not required to return a value). A read operation returns the correct value when it is not concurrent with any increment operation. In the case that a read operation overlaps an increment operation, it may return an arbitrary value. By "correct value", we mean the number of increment operations completed plus the initial value of the counter.

- A *dynamic counter*: supports weak increment and strong read. An increment operation increments the value of the counter by 1. A read operation returns the correct value even if the read is concurrent with other increments or reads. To define this formally, we use the notion of *cyclic interval* $[a, b]$ (mod m), defined for integers a, b

where $a \leq b$. This interval is the set $\{a \pmod m, a+1 \pmod m, \cdots, b \pmod m\}$. (In particular, if $b - a \geq m - 1$ then $[a, b] \pmod m = \{0, 1, \cdots, m-1\}$.) A value is correct for a run x of a dynamic counter (modulo m) if it is in the cyclic interval $[end(x) + c, begin(x) + c] \pmod m$ where $begin(x)$ is the number of processes that started an increment operation in x, $end(x)$ is the number of processes that completed their increment operation in x, and c is the value of the counter at the beginning of x.

- A *linearizable counter*: supports strong increment and strong read; and in every run the executions of the increment and read operations are linearizable. That is, it behaves as if each of these operations is atomic ([HW90]).

We are interested in *wait-free* implementations of counters as defined earlier. Sometime, in order to make the results more general, we will relax the wait-freedom assumption and only require that an implementation be *non-blocking*. While in a wait-free implementation an operation initiated by a correct process must terminate regardless of the speed of other processes, in a non-blocking implementation, whenever a correct process is trying to increment the counter, the counter is guaranteed to be eventually incremented, possibly by another process. Throughout the paper, unless we say otherwise, the word "counter" stands for "wait-free counter".

Summary of results

We ask the following question: let B be a set of integers, and suppose that for each $b \in B$ there is an unlimited supply of atomic counters (modulo b). For what values m, can we use these counters to construct public counters (modulo m)? The answer to this question, of course, would depend on the type of the counters assumed (i.e., whether the operations are strong or weak).

We prove two results. The first fully characterizes the static counters which can be implemented from a set of given atomic counters. The second result shows that it is impossible to construct large dynamic counters from smaller atomic counters, when the initial values of the smaller atomic counters are not known in advance.

More formally, let B be a set of integers. A counter *over* B is a counter which is constructed from a bounded number of atomic counters where each such atomic counter counts modulo some number b in B. (For each such b there may be many atomic counters (modulo b).) Our two results are:

1. It is possible to implement a static counter (modulo m) over B if and only if each prime number which divides m also divides some $b \in B$.

2. It is possible to implement a dynamic counter (modulo m) over B, assuming the counters are not initialized, only if $m \leq b$ for some $b \in B$.

The correctness of the second result depends on the assumption that the dynamic counter is required to work regardless of the initial values of the atomic counters it is constructed from. In the case where the initial values are known, it is possible to implement a dynamic counter (modulo 2^n) using atomic counters (modulo 2) [BIS95].

Related work

The area of concurrent and distributed data structure is relatively new, but already drawn the attention of many researchers. While the term concurrent data structure, refers to a data structure that is stored in shared memory, the term distributed data structure refers to a collection of local data structures stored at different processors in a message passing system. We will not try to review all the relevant work here, but rather give just few pointers to the literature.

Few works have introduced general methods for transforming a given sequential implementation (one that works for just one process) into a wait-free concurrent one [Her91b, Plo89]. These results are mainly of theoretical interest since the constructions involved are too inefficient to be practical. Other transformations are introduced in [Her90] for a large class of structures using the compare-and-swap synchronization primitive; in [Her91a] using the load_link and store_conditional primitives; and in [AT93] using timing assumptions.

More efficient constructions for specific data structures have been proposed. Many constructions of concurrent B-trees, have been implemented mainly for use in databases, see for example [BS77, LY81, Sag85]. AVL trees, 2-3 trees, and a distributed extendible hash file have been implemented in [Ell80a, Ell80b, Ell85]. A distributed dictionary structure is studied in [Pel90]. A wait-free implementation of a queue where one enqueuing operation can be executed concurrently with one dequeuing operation is given in [Lam83]. An implementation of a queue that allows an arbitrary number of concurrent queuing and dequeuing operations is given in [HW87], the implementation is deadlock-free but allows starvation of individual processes. A wait-free implementation of union-find structures is described in [AW91]. These data structures are not public data structures, as they all assume a fixed and known set of processes which may access the data structures.

The problem of implementing a counter in a concurrent environment has been the subject of intensive investigation recently. Aspnes, Herlihy and Shavit [AHS94] have implemented counters that support strong increment and weak read operations, which count modulo some given power of two, from basic elements called 2-balancers, which are essentially atomic counters (modulo 2). They named the implementations they have found *counting networks*. Counting networks achieve a high level of throughput by decomposing interactions among processes into pieces that can be performed in parallel, effectively reducing memory contention. Counting networks have been further investigated in [AA95, AHS94, AVY94, BHM94, BM98, FLL93, HKM93, HBS92, HSW91, HSW96, KP92, LSST96, MMT99, MPT97].

One result about counting networks that is more relevant to our work is proved in [AA95]. It is shown that a counting network with fan-out m (i.e., that counts modulo m) can be constructed from balancers of fan-out $b_1, \cdots, b_k$, only if for each prime factor p of m, p divides b_i, for some $1 \leq i \leq k$. Since a balancer of fan-out b is, in fact, a b-valued atomic counter, this condition immediately follows from our first (general) result stated earlier. Moreover, we show that this condition is in fact also sufficient for the construction of static counters from atomic ones. Independently of the work reported in this paper, several results about balancing and counting networks have been recently proven in [BM94b], including a proof that the necessary condition from [AA95], mention above, is also sufficient for the construction of counting networks.

Independently of the work on counting networks, counters that support weak increment and weak read (static counters) and counters which support weak increment and strong read (dynamic counters) were introduced and studied in [MTY96, MT97]. The results in these two papers concern the constructions of counters from read-modify-write bits. Notice that this model is different from our model which assumes objects which are atomic counters of arbitrary size.

One simple result from [MTY96] that we generalize in this paper is a space optimal static counter which can count modulo a given power of two. The main result in [MT97] is that in a model which supports only read-modify-write of single bits, a static counter (modulo m) exists only if $m = 2^k$, where k is bounded from above by the number of bits a process may change during a single increment operations. This result has the flavor of our first impossibility result, but it does not imply neither implied by it.

Finally, the relation between wait-free and bounded wait-free public data structures are studied in [BM94].

2 Static Counters

In this section we fully characterize the kind of static counters that can be constructed from smaller atomic counters. We assume that each counter has a single pre-defined initial value, though our results hold also when the initial contents of the counter is arbitrary.

Theorem 2.1 *There exists a static counter (modulo m) over B if and only if every prime number which divides m also divides some integer in B.*

The *only if* part of Theorem 2.1 was proved in [AA95] for counting networks, which as mentioned before, are special case of static counters. Recently, independent of our work, it was proved in [BM94b], that the *if* part of Theorem 2.1 holds for counting networks. We start by proving the *only if* part of the theorem, and then give a constructive proof for the *if* part.

2.1 Preliminaries

We start with two lemmas that are used later. The first is an elementary fact in number theory, while the second relates the wait freedom and bounded wait freedom properties.

For a set of integers B, let $lcm(B)$ be the *least common multiple* of all the integers in B.

Lemma 2.1 *Any prime that divides $lcm(B)$, divides some integer in B.*

Proof: The proposition follows from the Unique Representation Theorem which says that: Any integer α can be written in exactly one way as a product of the form $\alpha = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$ where $p_1 < p_2 \cdots < p_r$ are primes, and $e_1, e_2 \cdots, e_r$ are positive integers (see e.g. [MB79]). Thus, the least common multiple of the set $\{b_1, \cdots, b_t\}$ where each b_i is uniquely represented by $b_i = \prod_p p^{e_p^i}$, is represented by $\prod_p p^{e_p}$ where $e_p = max_{1 \leq i \leq t}\{e_p^i\}$. The result follows. $\square$

Let $L(Pr)$ be the supremum on the number of increment steps a process may need to take during an increment or a read operation, in the counter Pr. The wait freedom property guarantees that in every execution, every operation is terminated within a finite number of steps. However, there are examples of wait free data structures in which this number may be arbitrarily large, and hence, a priori, $L(Pr)$ may be infinite. However, the main result in [BM94] implies the following:

Lemma 2.2 *[BM94] Let Pr be a wait-free static counter. Then $L(Pr)$ is finite.*

2.2 Necessary Condition

We now prove several lemmas, the last of which proves the "only if" part of Theorem 2.1. The notation $(z - x)$ is used for the suffix of the run z obtained by removing x from z. We start with a technical lemma, which is also used later in the proof of Lemma 3.4.

Lemma 2.3 *Let Pr be a static counter over B, and let x be a run of Pr in which a set H of $lcm(B)^{L(Pr)}$ processes are in the same state. Then, there is an extension z of x in which only processes from H are active, such that the value of the counter is the same in x and z, and each process in H has completed one increment or read operation in $(z - x)$ and is idle in z.*

Proof: Assume without loss of generality that the processes in H are all involved in an increment operation. Let $L = L(Pr)$. The run z is constructed through a sequence of runs $x \leq y_0 \leq y_1, \cdots \leq y_L = z$. The construction is carried by induction, in rounds. In each round $1 \leq i \leq L$ we extend y_{i-1} constructed in the previous round to a run y_i, such that in y_i each process from H has taken at least i atomic increment steps or it has completed its increment operation. Since, by definition, at most L atomic increment steps are taken during a single increment operation, at y_L all the processes in H had completed one increment operation and are idle. By Lemma 2.2, L is finite and hence the number of rounds is going to be finite.

In the following we say that a process is *r-loaded* in a finite run x if its first next step in any extension of x is incrementing r; it is *loaded* if it is r-loaded for some atomic counter r.

The construction is such that, at the end of y_i the processes in H are partitioned to ℓ_i groups, denoted $H^i_1, \cdots, H^i_{\ell_i}$, where all the processes in H^i_j are in the same state (have the same history), and they are all either r-loaded on the same register r, or in the idle state. Also, it will always be the case that $lcm(B)^{L-i}$ divides $|H^i_j|$ for $1 \leq j \leq \ell_i$, and that the contents of the counter is the same in x and y_i.

Round 0: Run y_0 is constructed as an extension of the run x. Let $H^0_1 = H$. The run y_0 is constructed by activating each process in H^0_1 in turn, until it either becomes r-loaded for some r or completes its increment operation. The wait-freedom (or even the weaker non-blocking) property guarantees that one of the two must happen. It is clear that, since no change has been made, the value of the counter is the same in x and y_0; $lcm(B)^L$ divides $|H^0_1|$, and that all the processes in $H^0_1 = H$ are in the same state, which means that they are all either idle or r-loaded for some r.

Round $i+1$: Assuming we have constructed run y_i as required. We next show how to construct the run y_{i+1}. In the construction we partition each of the groups H^i_j, $1 \leq j \leq \ell_i$, to one or more groups which form the $i+1$-th partition. We start by explaining how this is done for H^i_1.

If all the processes in H_1^i have completed their increment operation then $H_1^i = H_1^{i+1}$ is a group in the $(i+1)$-th partition. Otherwise, we know that all the processes in H_1^i are r-loaded for some r. Let r be an atomic counter (modulo b). We split the set H_1^i to b sets, $H_1^{i+1}, \cdots, H_b^{i+1}$, such that $|H_j^{i+1}| = \frac{|H_1^i|}{b}$, for all $1 \leq j \leq b$. Then we activate the processes in $H_1^{i+1}, \cdots, H_b^{i+1}$ alternately as follows: first we let a process in H_1^{i+1} takes the atomic increment step which changes r, and then we let it continue until it completes its increment operation or becomes loaded for some other atomic counter. Then, we let a process in H_2^{i+1} do the same, and so on and so forth, until all the processes in H_1^i are activated.

When this procedure is completed, all the processes in H_j^{i+1} for each $1 \leq j \leq b$, are in the same state, which means that they are either idle or loaded. Since b divides $lcm(B)$ and $lcm(B)^{L-i}$ divides $|H_1^i|$, it follows that $lcm(B)^{L-i-1}$ divides $|H_j^{i+1}|$. Finally, r is incremented $0 \pmod{b}$ times and hence r has its original value (i.e., its value in x).

We repeat the procedure above sequentially with H_2^i, H_3^i and so on. After repeating this procedure ℓ_i times we get run y_{i+1}. The number of groups resulted from the construction of y_{i+1} is ℓ_{i+1}. And the value of the counter in y_i is the same in y_{i-1} (and thus also as in x).

Round L: After performing round L we get run $y_L = z$. Since each process in the set H has taken L increment steps in y_L or completed its increment operation, we have that all the processes in H have completed their increment operation in y_L and are idle, and the value of the counter is the same in x and z. □

Lemma 2.4 *Let Pr be a static counter (modulo m) over B. Then, m divides $lcm(B)^{L(Pr)}$.*

Proof: By applying Lemma 2.3 with x as the empty run and z as a run in which each process in H performs one increment operation, it follows that: there is a complete run z such that the value of the counter in z is the initial value, and exactly $lcm(B)^{L(Pr)}$ increment operations have been completed in z. By the definition of static counter, we must have that the value of the counter in z is equal to the initial value of the counter plus $lcm(B)^{L(Pr)}$ modulo m. On the other hand as explained the value of the counter in z is the same as the initial value. This two conditions can be satisfied only if m divides $lcm(B)^{L(Pr)}$. □

Lemma 2.5 *Let Pr be a static counter (modulo m) over B. Then, each prime factor of m divides some integer in B.*

Proof: By Lemma 2.4, m divides $lcm(B)^{L(Pr)}$, and hence any prime factor of m also divides $lcm(B)$. Thus, by Lemma 2.1, each prime factor of m divides some integer in B. □

The only place where we must use the wait-freedom property is in the proof of Lemma 2.2. If instead of using Lemma 2.2 we assume that $L(Pr)$ is finite, then the proof of Lemma 2.5 (i.e., the *only if* part) holds also for non-blocking static counters without requiring the counters to be wait-free. (Notice that finiteness of $L(Pr)$ does not imply that Pr is wait-free, since Pr may perform an infinite number of read operations in a run.)

2.3 The Mixed Radix Counter

Next we give a simple constructive proof of the *if* part of Theorem 2.1. That is, we prove the following Lemma:

Lemma 2.6 *If every prime number which divides m also divides some integer in B, then there exists a static counter (modulo m) over B.*

To prove the lemma we design a static counter called the *mixed radix counter*. This counter is an extension of the Positional Counter presented in [MTY96].

Let us write m's prime factorization as $m = \prod_{i=0}^{k-1} p_i$, where for all i, p_i is prime (the p_i's are not necessarily distinct). The lemma assumes that for all $0 \leq i \leq k-1$, p_i divides some number in B. Let r_i be an atomic counter which counts modulo some number in B which is divisible by p_i. In Figure 1 the code for a counter (modulo m), using the atomic counters $r_0, ..., r_{k-1}$, is given.

An increment operation by a process is performed by the following straightforward (sequential) algorithm: scan the registers from right to left (starting with r_0); when scanning register r_i, do the following: (1) increment r_i, and (2) if before the increment the value of r_i modulo p_i was $p_i - 1$ and $i < k-1$, then repeat this operation on register r_{i+1}, else terminate the increment operation.

The read operation is performed by simply reading the content of the registers, and returning the value of the counter which is associated with the content of the k atomic counters $r_{k-1}, \cdots, r_0$. This value is defined by the mixed radix systems (see [Knu69]) as,

$$\sum_{i=0}^{k-1} (r_i \pmod{p_i} \times \prod_{j=0}^{i-1} p_j).$$

```
function read;
begin
    /* read the values of the registers */
    for i := 0 to k − 1 do a[i] := r_i;
    return Σ_{i=0}^{k−1} (a[i] (mod p_i) × Π_{j=0}^{i−1} p_j);
end;

procedure increment;
begin
    i := 0;
    repeat
        /* execute an atomic increment step on r_i */
        flag := increment(r_i);
        i := i + 1;
    until flag (mod p_{i−1}) ≠ 0 or i = k;
end;
```

Figure 1: The mixed radix counter

2.4 Correctness Proof

We denote by *initial(x)* the values of the atomic counters at the beginning of the run x, and by *final(x)* the values of the atomic counters at the end of x. Recall that $begin(x)$ is the number of processes that started an increment operation in x. A run is a *serial run* if in any prefix of it, at most one process is involved in an `increment` operation. Two complete runs x and y are *similar* if $initial(x) = initial(y)$ and $begin(x) = begin(y)$. Two complete similar runs x and y are *equivalent* if they also satisfy $final(x) = final(y)$.

First we show that any two similar complete runs of the counter are equivalent. For this we show that the number of times an atomic counter is incremented during a run depends only on the initial values of the atomic counters and the number of `increment` operations performed. Then, we observe that the counter is correct when we consider only serial runs. Since any complete run is also equivalent to some complete serial run, the counter is correct for all completes runs.

Lemma 2.7 *Let x be a complete run of Pr, let v_i be the initial value of atomic counter r_i $(1 \leq i \leq k)$, and let $C(r_i, x)$ be the number of times r_i was incremented in x. Then $C(r_0, x) = begin(x)$ and for $i > 0$, $C(r_i, x) = \lfloor \frac{C(r_{i-1},x)+(v_{i-1} \pmod{p_{i-1}})}{p_{i-1}} \rfloor$.*

Proof: The proof is by induction on r_i. By observing the increment procedure it is immediate that each execution of `increment` operation changes r_0 exactly once. Thus r_0 is incremented $begin(x)$ times.

Suppose the lemma holds for atomic counter r_{i-1}, we show that is also holds for atomic counter r_i. The number of times that the value of r_{i-1} is changed from $p_{i-1} - 1 \pmod{p_i}$ to $0 \pmod{p_i}$ is $\lfloor \frac{C(r_{i-1},x)+(v_{i-1} \pmod{p_{i-1}})}{p_{i-1}} \rfloor$. Every complete execution of an `increment` operation that changes the value of r_{i-1} from $p_{i-1} - 1 \pmod{p_i}$ to $0 \pmod{p_i}$ changes also r_i, and every other execution that change the value of r_{i-1} from j to $j+1$,$(j \neq p_{i-1} - 1 \pmod{p_{i-1}})$ halts. Hence the number of times r_i was changed is $\lfloor \frac{C(r_{i-1},x)+(v_{i-1} \pmod{p_{i-1}})}{p_{i-1}} \rfloor$. □

Lemma 2.8 *Let x be a complete run of the counter Pr. Then the value of the counter at x equals to the initial value of the counter plus the number of processes that have started an increment operation in x, modulo m.*

Proof: It follow immediately from the properties of the increment procedure for addition that the lemma holds for any complete serial. It is shown in Lemma 2.7 that the number of times an atomic counter is changed during any complete run depend only on the initial values of the atomic counters and on the number of `increment` operations performed. This implies that any two similar complete runs of the counter Pr are equivalent. Since any complete run is similar to some complete serial run, it follows that any run is also equivalent to some complete serial run. Thus, since the lemma holds for all complete serial runs, it holds for all complete runs. □

3 Dynamic Counters

In this section we investigate properties of dynamic counters, assuming that the memory is bounded and that the initial contents of the counter is arbitrary. Our conjecture is that except for trivial cases, it is impossible to construct such counters:

Conjecture 1 *There is a dynamic counter (modulo m) over B if and only if m divides b for some $b \in B$.*

Notice that the *if* part of Conjecture 1 is trivial. We prove the following result, which is slightly weaker than the *only if* part of the conjecture.

Theorem 3.1 *There is a dynamic counter (modulo m) over B only if $m \leq b$ for some $b \in B$.*

We notice that Theorems 3.1 and 2.1 imply Conjecture 1 for the case where all the elements of B are powers of the same prime.

Theorem 3.1 does not hold when the dynamic counter is not required to be public. That is, if there is a known bound on the number of process that may access it (even if their identities are not known in advance). In such a case, one can construct a dynamic counter which uses only binary atomic counters, and counts modulo 2^k for arbitrary large k [MTY96]. Also, the theorem does not hold in the case where only one initial value is assumed, as it is possible to implement a dynamic counter (modulo 2^k) using binary atomic counters which are initialized to zero [BIS95].

To prove Theorem 3.1, we assume that there exists a dynamic counter (modulo m) over B, for some $m > \max(B)$. Then, we derive a contradiction by showing that for any integer k, if there is such a counter which uses k atomic counters, then there must be such a counter that uses $k-1$ atomic counters.

3.1 Preliminaries

We assume in this section that the read operation of a dynamic counter is performed by an atomic snapshot operation, which reads the values of all the atomic counters in one indivisible step (in [Bri94] it is shown that a read operation can always be implemented by performing only read steps, hence this assumption makes our impossibility result stronger). Hence, the read operation defines a function *val* which associates with each contents of the counter a value in $\{0, \cdots, m-1\}$.

Thus, a dynamic counter (modulo m) over B is given by a triple $Pr = (\mathsf{increment}, val, V_{init})$ where increment is a procedure for incrementing the counter, *val* is a function that associates an integer value in the range $\{0, ..., m-1\}$ to any possible contents of the counter, and V_{init} is the set of initial vectors of Pr.

A process performs the increment operation on the counter by executing an increment procedure. (Many increments can take place concurrently.) We assume that all the processes are identical. The correctness requirements for a dynamic counter has been defined in the Introduction.

Recall that we denote by the vector *initial(x)* the values of the atomic counters at the beginning of the run x, and by *final(x)* the values of the atomic counters at the end of x. A run x is *legal* if $initial(x) \in V_{init}$. A vector $\vec{v}$ is *reachable* from vector $\vec{u}$ (w.r.t. the given counter Pr) if there is a run x of Pr with $initial(x) = \vec{u}$ and $final(x) = \vec{v}$. Our assumption that the initial contents of the counter is arbitrary means that every vector which is reachable from an initial vector is also an initial vector. More formally, let $V_{reach} = \{\vec{u}|$ there is a vector $\vec{v} \in V_{init}$ such that $\vec{u}$ is reachable from $\vec{v}\}$. Then we assume that $V_{init} = V_{reach}$.

3.2 Two Graphs Associated with Counters

Let $Pr = (\mathsf{increment}, val, V_{init})$ be a dynamic counter (modulo m) over B which uses atomic counters $r_1, \cdots, r_k$, where r_i counts modulo b_i ($b_i \in B$). In our discussions we associate with Pr two directed graphs.

The first graph is the *run graph* of Pr which is defined as $G_r = (V_{init}, E_r)$, where $E_r = \{(\vec{u}, \vec{v}) : \vec{v}$ is reachable from $\vec{u}\}$.

Clearly, G_r is transitively closed (i.e., for each $\vec{u}$, $\vec{v}$, there is an edge from $\vec{u}$ to $\vec{v}$ iff there is a directed path from $\vec{u}$ to $\vec{v}$). A strongly connected component C of a directed graph G is *maximal* if there is no edge from a vertex in C to a vertex not in C. A vertex in G is maximal if it belongs to a maximal strongly connected component. Note that, since $V_{init} = V_{reach}$ is finite, G_r must contain a maximal strongly connected component. The following lemma gives a useful property of maximal strongly connected components of G_r. Recall that a process is *r-loaded* in a finite run x if its first next step in any extension of x is incrementing r; it is *loaded* if it is r-loaded for some atomic counter r.

Lemma 3.1 *Let C be a maximal strongly connected component of G_r. Then $C = C_1 \times \cdots \times C_k$, where for all $1 \le i \le k$, C_i is either a singleton or $C_i = \{0, \cdots, b_i - 1\}$.*

Proof: First we prove that for each i, if the value of r_i is not fixed in C, then for each vector $\vec{v} \in C$ there is a run which starts in $\vec{v}$ and in which some process is r_i-loaded. Let such an i and $\vec{v} = (v_1, \cdots, v_k)$ be given. By the assumption, there is a vector $\vec{u} = (u_1, \cdots, u_k) \in C$ such that $v_i \neq u_i$. Since C is strongly connected, there is a run x with $initial(x) = \vec{u}$ and $final(x) = \vec{v}$. Since the value of r_i is changed in x, x must have a prefix in which some process is r_i-loaded, as claimed.

The run $x_{\vec{v}}$ above can be extended to a run $z = z_{\vec{v}}$ with $initial(z) = \vec{v}$ with the following property: for each i such that the value of r_i is not fixed in C, there are b_i processes which are r_i-loaded in z. Thus, at the end of the run z, if the value of r_i is not fixed in C, we can change it to any value in Z_{b_i} ($Z_{b_i} = \{0, ..., b_i - 1\}$), by activating some of the r_i-loaded processes in z. The lemma follows. □

The second graph associated with Pr is the *operation tree* of Pr, denoted as $T_{op} = (V_{op}, E_{op})$, which is a (possibly infinite) tree defined as follows: V_{op} is the set of all states (local histories) of the $\mathsf{increment}$ procedure, and E_{op} is the set of all state transitions of this procedure. More specifically, T_{op} is defined inductively as follows: the root of T_{op} is the initial state of the $\mathsf{increment}$ procedure. For every state s in T_{op}, if s is an halting state then s is a leaf in T_{op}. Otherwise, let r be the next atomic counter accessed by the procedure $\mathsf{increment}$ in state s, and assume that r is a counter (modulo b). Then s has b children, one for each possible value of r.

Given a maximal strongly connected component C of G_r, we now construct a tree $T_{op}(C)$, the *operation tree of Pr induced by C*, as follows: Start with T_{op}, the operation tree of Pr, scan all its non-leaf vertices in a BFS order, starting from the root, and for each such vertex s do the following: Let r_i be the atomic counter accessed by a process in state

s. If $C_i = Z_{b_i}$ then do nothing; otherwise, by Lemma 3.1 $C_i = \{v_i\}$ for some $v_i \in Z_{b_i}$. For each $v \in Z_{b_i} \setminus \{v_i\}$, delete from $T_{op}(C)$ the vertex corresponding to v and all its descendants (i.e., the only remaining child of s in $T_{op}(C)$ is the one corresponding to the move in which the value returned by the operation on r_i is v_i). $T_{op}(C)$ is the tree defined by the limit of this (possibly infinite) procedure.

Observe that every directed path which starts from the root in $T_{op}(C)$ defines a sequence of state transitions of a process executing *op*. The next lemma shows that for each such path there is a run in which some process actually performs this sequence of transitions.

Lemma 3.2 *Let C be a maximal strongly connected component of G_r. Then for every path π of $T_{op}(C)$, starting from the root, there is a run $x = x(\pi)$, in which some process executes the sequence of state transitions defined by π.*

Proof: Let π be given. The run x is defined as the limit of a sequence of runs $(x_0, x_1, \cdots)$ where the runs x_i are defined inductively as follows: x_0 is the empty run where process p is in state s_0. At (the end of) run x_i, $i \geq 0$, process p is in state s_i, and there is an edge (s_i, s_{i+1}) corresponding to the next step of p. Assume that in this step p accesses the atomic counter r and the value of r is $\ell \in Z_b$. There is a vector $\vec{v} \in C$ such that the value of r in $\vec{v}$ is ℓ. x_{i+1} is defined as $x_{i+1} = x_i \cdot y_i \cdot z_i$, where y_i is a run satisfying $initial(y_i) = final(x_i)$ and $final(y_i) = \vec{v}$ (y exists since $\vec{v} \in C$ and C is strongly connected), and z_i consists of the single step by p in which it moves from s_i to s_{i+1}. □

In [BM94] it is shown that the conclusion of Lemma 3.2 holds also without the assumption that C is a maximal strongly connected component. Lemma 3.2 above implies the following:

Corollary 3.1 *For every maximal strongly connected component C of G, the induced operation tree $T = T_{op}(C)$ is of bounded height (and hence in every run x of Pr where $initial(x) \in C$, every increment operation is completed within at most L atomic steps, for some constant L).*

Proof: Otherwise, for each i there is a path in T whose length is i. Since the out degree of every vertex in T is finite, by König's Infinity Lemma [Kön36] this tree contains an infinite path π, and by Lemma 3.2 there is a run in which some process actually executes the state transitions defined by π, and thus never completes executing its increment operation - contradicting the wait-freedom assumption. □

For each maximal strongly connected component C, $T_{op}(C)$ defines a procedure, consisting of part of the states and transition rules of the increment procedure, in a natural way. This procedure is denoted *the increment procedure induced by C*.

Lemma 3.3 *Let Pr be a dynamic counter (modulo m) over B, which uses a minimum number of atomic counters. Then, the run graph of Pr is strongly connected.*

Proof: Recall that we assume that every vector which is reachable from an initial vector is also a possible initial vector. Let G_r be the run graph of $Pr = (\mathsf{increment}, val, V_{init})$, and assume to the contrary that G_r is not strongly connected. Let C be some maximal strongly component of G_r. Define a counter $Pr' = (\mathsf{increment}', val, V'_{init})$ where $\mathsf{increment}'$ is the increment procedure induced by C, and $V'_{init} = V_{init} \cap C$ (notice that V'_{init} is nonempty).

From the observation that the set of legal runs of Pr' is equal to the set of legal runs of Pr which start from a vector in V'_{init}, it follows that Pr' is also a dynamic counter (modulo m) over B. Since the number of vectors in C is strictly smaller than $|V_{init}|$, Lemma 3.1 implies that $C = C_1 \times \cdots \times C_k$, where for some i, C_i is a singleton. Hence, the dynamic counter Pr' can be implemented by a protocol that never uses r_i, and hence by fewer atomic counters than Pr, a contradiction. □

Let C be a maximal strongly connected component of G_r such that $V'_{init} = C \cap V_{init} \neq \emptyset$. For later use, we call the counter $Pr' = (\mathsf{increment}', val, V'_{init})$ defined in the proof of Lemma 3.3, the *restriction of Pr induced by C*. Notice that every legal run of Pr' is also a legal run of Pr.

From now on we assume that $Pr = (\mathsf{increment}, val, V_{init})$ is a dynamic counter (modulo m) which uses a minimum number of atomic counters. We assume that Pr uses the k atomic counters, $r_1, \cdots, r_k$, where r_i is an atomic counter (modulo b_i). By Lemma 3.3, we may assume that G_r, the run graph of Pr, is strongly connected, and in particular that every vector in $V_{init} = V_{reach}$ is maximal in G_r.

3.3 Extremal Vectors

Next, we define a subgraph of G_r, called the *complete run graph* of Pr, denoted by G_{cr}, which has the same vertex set as G_r but a smaller edge set: $G_{cr} = (V_{init}, E_{cr})$, where $E_{cr} = \{(\vec{u}, \vec{v}) : \vec{v}$ is reachable from $\vec{u}$ by a complete run (of Pr)$\}$.

Definition: A vector $\vec{v}$ is *extremal* w.r.t. counter Pr if it is maximal in both G_r and G_{cr} We next show that every vector in G_r extremal.

Lemma 3.4 *Every vector $\vec{v} \in G_r$ (and hence in G_{cr}) is an extremal.*

Proof: By Lemma 3.3, the run graph G_r of Pr is strongly connected, and hence every vector $\vec{v}$ is maximal in G_r. We have to show that every such vector $\vec{v}$ is also a maximal vector in G_{cr}. For this, it is sufficient to show that for every vector $\vec{u}$, if there is a complete run from $\vec{v}$ to $\vec{u}$ then there is also a complete run from $\vec{u}$ to $\vec{v}$. Let such a vector $\vec{u}$ be given. Since G_r is strongly connected, there is a run x_1 from $\vec{u}$ to $\vec{v}$. If x_1 is a complete run then we are done, so assume that x_1 is not complete. We prove the lemma by constructing a complete run from $\vec{u}$ to $\vec{v}$.

Let $Q = \{q_1, q_2, \cdots, q_{|Q|}\}$ be the set of processes which are not idle at (the end of) the run x_1. Let L be a bound on the number of atomic increment operations that are needed to complete an `increment` operation (L is finite by Lemma 3.2). Finally, let n be the least common multiple of $\{b_1, \cdots, b_k\}$.

We now construct a complete run, y, such that $initial(y) = \vec{u}$ and $final(y) = \vec{v}$. The run y is a concatenation $y = y_0 \cdot y_1 \cdots y_{|Q|}$, where y_0 is a (possibly incomplete) run from $\vec{u}$ to $\vec{v}$, while for $1 \leq i \leq |Q|$, y_i is a partial run satisfying $initial(y_i) = final(y_i) = \vec{v}$. This implies that $initial(y) = \vec{u}$ and $final(y) = \vec{v}$. We will also show that y is complete, which implies the lemma. Next we show how to construct each of the partial runs y_i, $0 \leq i \leq |Q|$.

We first explain how y_0 is constructed. By assumption, there is a complete run, x_2, from $\vec{v}$ to $\vec{u}$. The run y_0 starts with a run identical to x_1, followed by $n^L - 1$ runs, each of which is identical to the run $x_2 \cdot x_1$ (i.e., x_2 followed by x_1), such that the sets of processes activated in different occurrences of x_1 and x_2 are distinct. That is, $y_0 = x_1 \cdot (x_2 \cdot x_1)^{n^L - 1}$.

From the construction, it is clear that $initial(y_0) = \vec{u}$ and $final(y_0) = \vec{v}$. Also, the set of processes which are not idle in y_0 can be divided to $|Q|$ subsets $F_1, \cdots, F_{|Q|}$, such that F_i contains n^L processes, and the state of each process in F_i is identical to the state of q_i at the end of x_1.

For $1 \leq i \leq |Q|$, the (partial) run y_i is constructed by activating only processes from F_i such that: (1) at (the end of) y_i all the processes in F_i are idle, and $initial(y_i) = final(y_i) = \vec{v}$. The fact that we can construct y_i as above follows from Lemma 2.3, which is applied with $F_i = H$, $x = y_0 \cdots y_{i-1}$ and $z = y_0 \cdots y_{i-1} \cdot y_i$.

All this implies that y is a complete run and $final(y) = \vec{v}$, as needed. □

We point out that Lemma 3.4 does not imply that G_{cr} is strongly connected. It only implies that every maximal component of it is strongly connected.

3.4 Terminal Incrementors and Critical Atomic Counters

We now introduce the new notion of a process which is *terminal incrementor* in a run. Recall that a process is r-loaded in a run x if its next step is an atomic increment on the atomic counter r.

Definition: A process p is *r-terminal incrementor* in a run x if it is r-loaded in x, and the next step of p is its last step during the current `increment` operation, regardless of the value of r while this step is taken. Process p is a *terminal incrementor* if it is an r-terminal incrementor for some r.

Our proof is based on constructions of runs in which processes are forced to become terminal incrementors. For this, we describe terminal incrementors by considering T_{op}, the operation tree of the `increment` procedure, defined in Subsection 3.2.

By Corollary 3.1, the operation tree T_{op} is finite. Hence it includes an internal vertex all whose children are leaves (e.g., an internal vertex of maximum possible depth). Call such a vertex *terminal vertex*, and the corresponding state *terminal state*. Whenever a process executing the `increment` operation is in a terminal state, it is going to complete its `increment` operation in its next atomic step, regardless of the value of the atomic counter it accesses in this step. Hence, we may assume that the next atomic step of a process in terminal state is an atomic increment step, as otherwise we may modify the counter by omitting this step, without affecting the `increment` procedure. Therefore, we may assume that a process in a terminal state is a terminal incrementor. For reasons which will become obvious soon, we call a atomic counter accessed in a terminal state a *critical* atomic counter.

Lemma 3.5 *Let r be a critical atomic counter which counts modulo b. Then for each vector $\vec{v}$ and for each integer $\ell \geq 0$, there is a run z_ℓ with $initial(z_\ell) = \vec{v}$, satisfying:*

1. *ℓ processes are r-terminal incrementors in z_ℓ, and all other processes are idle in z_ℓ.*
2. *If $\ell = 0 \pmod b$, then $final(z_\ell) = \vec{v}$.*

Proof: The proof is by induction on ℓ. The run z_0 is the empty run starting and ending at $\vec{v}$. Assume that the lemma holds for $\ell \geq 0$, and let $Q_\ell = \{q_1, \cdots, q_\ell\}$ be the ℓ processes which are r-terminal incrementors in z_ℓ. The run $z_{\ell+1}$ is an extension of z_ℓ, constructed as follows.

First assume that $\ell \neq 0 \pmod{b}$. Let $q_{\ell+1}$ be some process which is idle in z_ℓ. By Lemma 3.2 there is an extension y_ℓ of z_ℓ by processes which are idle in z_ℓ, such that $q_{\ell+1}$ is an r-terminal incrementor in y_ℓ. Observe that since no process from Q_ℓ is activated in $y_\ell - z_\ell$, all the processes in Q_ℓ are also r-terminal incrementors in y_ℓ. Thus $Q_{\ell+1} = Q_\ell \cup \{q_{\ell+1}\}$ satisfies the Lemma at y_ℓ. The run $z_{\ell+1}$ is obtained by letting all the non idle processes in y_ℓ which are not in $Q_{\ell+1}$ complete their `increment` operations.

Now assume that $\ell = 0 \pmod{b}$. We modify the run z_ℓ above to a run z'_ℓ which satisfies also requirement 2.
If we let all the ℓ terminal incrementors in z_ℓ complete their `increment` operations, we get a complete run y with $final(y) = final(z_\ell)$. This implies that there is a complete run from $\vec{v}$ to $final(z_\ell)$. By Lemma 3.4, $\vec{v}$ is extremal, and hence there is also a complete run y' with $initial(y') = final(z_\ell)$ and $final(y') = \vec{v}$. The modified run z'_ℓ which satisfies both 1 and 2, is defined by $z'_\ell = z_\ell \cdot y'$. □

3.5 Independence of val on Values of Critical Atomic Counters

In this section we show that if $\vec{u}$ and $\vec{v}$ are two vectors that differ only by the value of a single critical atomic counter then $val(\vec{u}) = val(\vec{v})$. Later on we show that this fact implies that critical atomic counters are redundant, in some precise sense, and use this to prove Theorem 3.1. Let $\vec{u}$ be a vector which represents the values of all the atomic counters used in a counter Pr at some point in time. We use the notation $\vec{u}(r)$ to denote the value of the atomic register r in $\vec{u}$. Recall that the dynamic correctness requires that for every run x with $initial(x) = \vec{v}$ and $final(x) = \vec{u}$, it holds that $val(\vec{u}) \in [end(x) + val(\vec{v}), begin(x) + val(\vec{v})] \pmod{m}$, where $begin(x)$ [$end(x)$] denote the number of processes which started [completed resp.] an increment operation in x.

Definition: Let r be a critical atomic counter (modulo b), and let $\vec{u}$ and $\vec{v}$ be two vectors in the run graph of P_r. Vector $\vec{u}$ is the (r,i)-*companion* of $\vec{v}$ if,

1. For every atomic counter $r' \neq r$, $\vec{u}(r') = \vec{v}(r)$ and
2. $\vec{u}(r) - \vec{v}(r) = i \pmod{b}$.

Vector $\vec{u}$ is an *r-companion* of vector $\vec{v}$ if it is the (r,i)-companion of $\vec{v}$ for some i.

Notice that for a critical atomic counter r which counts modulo b, every vector is an (r,b)-companion of itself, and if $\vec{u}$ is the (r,i)-companion of $\vec{v}$ then $\vec{v}$ is the $(r, b-i)$-companion of $\vec{u}$.

Lemma 3.6 *Let r be an critical atomic counter. If $\vec{v}_1$ is the $(r,1)$-companion of $\vec{v}_0$, then $val(\vec{v}_1) - val(\vec{v}_0) \in [0,1] \pmod{m}$.*

Proof: For simplicity, assume that $val(\vec{v}_1) = 1$. Thus, we have to show that $val(\vec{v}_0) \in \{0,1\}$. By Lemma 3.4 the vector $\vec{v}_1$ is an extremal vector. Since r is critical, by Lemma 3.5 there is a run z_b from $\vec{v}_1$ to itself, at the end of which all the processes are idle, except for b processes which are r-terminal incrementors. If we let $b-1$ of these terminal incrementors perform an atomic increment operation on r and complete their `increment` operation, we get a run y with $initial(y) = \vec{v}_1$ and $final(y) = \vec{v}_0$. In y all the processes are idle except for one r-terminal incrementor, and if we let this terminal incrementor complete its `increment` operation, it will increment the value of r from 0 to 1 and become idle. Thus, the resulted run, y', is a complete run satisfying $initial(y') = final(y') = \vec{v}_1$, hence $begin(y') = end(y') = 0 \pmod{m}$, meaning that $begin(y) = 0 \pmod{m}$ and $end(y) = -1 \pmod{m}$.

By the correctness requirement for dynamic counters, we get that for run y,

$$val(\vec{v}_0) \in [val(\vec{v}_1) + end(y), val(\vec{v}_1) + begin(y)] \pmod{m} = [0, 1],$$

which completes the proof. $\square$

Corollary 3.2 *Let r be a critical atomic counter (modulo b), and let $1 \leq i \leq b$. If $\vec{v}_i$ is the (r,i)-companion of $\vec{v}_0$ then,*

- *$val(\vec{v}_i) - val(\vec{v}_0) \in [0, i] \pmod{m}$, and*
- *if $val(\vec{v}_i) - val(\vec{v}_0) = 0 \pmod{m}$, then $val(\vec{v}_j) = val(\vec{v}_0) \pmod{m}$, $0 \leq j \leq i$.*

Proof: Let $h_j = val(\vec{v}_j) \pmod{m}$, and let $\delta_j = h_{j+1} - h_j \pmod{m}$. Let further $\Delta_i = \sum_{j=0}^{i-1} \delta_j$. Then $val(\vec{v}_i) = val(\vec{v}_0) + \Delta_i \pmod{m}$. By Lemma 3.6, $\delta_j \in \{0, 1\}$, and hence $0 \leq \Delta_i \leq i$, which, since $i < m$, implies the first part of the corollary.

The second part follows from the fact that $0 \leq \Delta_i \leq i \leq b < m$, hence if $\Delta_i = 0 \pmod{m}$ then $\Delta_i = 0$, and hence $\delta_j = 0$ for $0 \leq j \leq i-1$. $\square$

Lemma 3.7 *Let r be a critical atomic counter. Then for every vector $\vec{u}$ and for every vector $\vec{v}$ which is an r-companion of $\vec{u}$, $val(\vec{u}) = val(\vec{v})$.*

Proof: Let r be a critical atomic counter (modulo b). Let $\vec{u} = \vec{u}_0$ be given, and for $1 \leq i \leq b-1$, let $\vec{u}_i$ be the (r,i)-companion of $\vec{u}_0$. We have to show that for each such i, $val(\vec{u}_i) = val(\vec{u}_0)$. By Corollary 3.2 with $i = b$, we have that for some integer Δ_b, $0 \leq \Delta_b \leq b \pmod{m}$,

$$val(\vec{u}_b) - val(\vec{u}_0) = \Delta_b \pmod{m}, \tag{1}$$

Since $\vec{u}_b = \vec{u}_0$, we also have that

$$val(\vec{u}_b) - val(\vec{u}_0) = 0 \pmod{m}. \tag{2}$$

Since $0 \leq \Delta_b \leq b < m$, equalities 1 and 2 imply that $\Delta_b = 0$. The lemma follows by Corollary 3.2. $\square$

3.6 Reducing the Number of Atomic Counters

We complete our impossibility proof by the following argument: Let k be the minimum possible number of atomic counters needed to implement a dynamic counter (modulo m) over B, and let Pr be such a counter which uses k atomic counters. Then there exists a dynamic counter (modulo m) over B, which uses only $k-1$ atomic counters - a contradiction to the minimality of Pr.

Let $r_1, \cdots, r_k$ be the atomic counters used by Pr. As already explained in the beginning of Subsection 3.4, since the operation tree is finite, at least one of the atomic counters is critical. So, w.o.l.g. assume that r_1 is a critical atomic counter. We derive a contradiction by constructing a dynamic counter (modulo m), called Pr', which uses only atomic counters $r_2, \cdots, r_k$. For a k-vector $\vec{v} = (v_1, \cdots, v_k)$, $trunc(\vec{v})$ denotes the $k-1$-vector $\vec{v}' = (v_2, \cdots, v_k)$. We now define the counter $Pr' = (\mathsf{increment}', val', V'_{init})$:

- $\mathsf{increment}'$ is identical to $\mathsf{increment}$, with the following modification: Any atomic increment step taken by $\mathsf{increment}$ on r_1 is replaced in $\mathsf{increment}'$ by a *virtual* atomic increment step, which assumes that the value of r_1 is 0 (and changes it to 1); similarly, each read step of $\mathsf{increment}'$ assumes that the value of r_1 is 0. Whenever such a virtual step is taken by $\mathsf{increment}'$, the state of the process is changed as it would have been changed in executing the original $\mathsf{increment}$ procedure, but no actual atomic increment or read step is taken.
- For every $k-1$-vector $(v_2, \cdots, v_k)$,
$$val'(v_2, \cdots, v_k) = val(0, v_2, \cdots, v_k) \; [= val(i, v_2, \cdots, v_k), 0 \le i < b].$$
- $V'_{init} = \{trunc(\vec{v}) | \vec{v} \in V_{init}\}$. That is, V'_{init} consists of vectors in V_{init} without their first entry. (Note that $V_{init} = Z_{b_1} \times \cdots \times Z_{b_k}$, hence $V'_{init} = Z_{b_2} \times \cdots \times Z_{b_k}$.)

The next lemma show that critical atomic counters can be ignored by dynamic counters. In this lemma we use the notation $domain(x)$ to denote the set of values which are correct for run x, that is:

$$domain(x) = [end(x) + val(initial(x)), begin(x) + val(initial(x))] \pmod m.$$

Lemma 3.8 *Let $\vec{v} = (0, v_2, \cdots, v_k)$, and let $\vec{v}' = trunc(\vec{v})$. Let x' be a run of Pr' with $initial(x') = \vec{v}'$ and $final(x') = \vec{u}'$ for some $\vec{u}'$. Then there is a run x of Pr satisfying: (i) $initial(x) = \vec{v}$, (ii) $final(x) = \vec{u}$, where $trunc(\vec{u}) = \vec{u}'$, and (iii) $domain(x) = domain(x')$.*

Proof: Let x' be a run from $\vec{v}'$ to $\vec{u}'$. We show how to construct the corresponding run x. Suppose that in x' there are t virtual atomic increment operations that (virtually) change the value of r_1 from 0 to 1. Let $\ell = (b-1)\lceil t/b \rceil$. The run $x = x_1 \cdot x_2$ is defined as follows:

1. Start with a run x_1 which is identical to the run $z_{b\ell}$ defined in Lemma 3.5. In particular, $initial(x_1) = final(x_1) = \vec{v}$, and in x_1 all the processes are idle except $b\ell$ r_1-terminal incrementors.
2. Now extend the run x_1 by x_2, which is identical to x', with the following exception: Whenever in x' a process p executes a virtual atomic increment step on r_1, x_2 is modified as follows: first, p actually performs this atomic increment step. Then, $b-1$ of the suspended r_1-terminal-incrementors are activated and complete their `increment` operation. As a result, the value of r_1 is reset to 0 in x_2.
3. If at the end of this simulation there are still suspended terminal incrementors, let them complete their `increment` operation.

We prove that x satisfies (i)-(iii) above. (i) holds trivially. To prove (iii), consider the run x_1. In this run $b\ell$ processes are r_1 terminal incrementors, and all other processes are idle. If we let these $b\ell$ terminal incrementors complete their `increment` operation, we get a complete run y, where $initial(y) = final(y) = \vec{v}$ and hence $end(y) = begin(y) = 0 \pmod{m}$. Since $begin(y) = begin(x_1)$ we conclude that $begin(x_1) = Km$ for some integer K. Since $begin(x) = begin(x_1) + begin(x_2)$ and $begin(x_2) = begin(x')$, we conclude that $begin(x) = begin(x') + Km$. Also, in x all the processes that started an `increment` operation during x_1 completed this operation. Hence we have by a similar argument that $end(x) = end(x') + Km$. Thus, $domain(x) = domain(x')$. This proves (iii).

Finally, by the construction of x_2 and the fact that the r_1 terminal incrementors changed only the value of r_1, we have that $final(x') = trunc(final(x))$. This proves (ii). □

Lemma 3.9 *If Pr is a dynamic counter (modulo m), then also Pr' is a dynamic counter (modulo m).*

Proof: We assume the contrary, and derive a contradiction. Assume that Pr' is not a dynamic counter (modulo m). This means that there is a legal run x' of Pr', such that $initial(x') = \vec{v}'$, $final(x') = \vec{u}'$, and $val(\vec{u}') \notin domain(x')$.

Consider the run x constructed for x' as in Lemma 3.8. Then, by lemma 3.8, $domain(x) = domain(x')$, and by Lemma 3.7 and the definition of val', $val(\vec{u}) = val'(\vec{u}')$. Therefore if $val(\vec{u}') \notin domain(x')$ then $val(\vec{u}) \notin domain(x)$, which contradicts the assumption that Pr is a dynamic counter (modulo m). □

Proof of Theorem 3.1: Assume that there exists a dynamic counter (modulo m), for $m > b$. Then, there exists such a counter, Pr, which use a minimal possible number of shared atomic counters, say k. By Lemma 3.9 there is another dynamic counter (modulo m), which uses only $k-1$ atomic counters, a contradiction. □

4 Discussion

Experience is showing that message-passing systems are more difficult to program than shared memory systems: "Parallel computers come with or without shared memory. One is hard to build, the other hard to program" [TKB92]. Shared memory is widely considered a useful programming abstraction for concurrent systems. Many experimental and commercial processors provide direct support for this abstraction, and increasing attention is being paid to implementing shared memory systems either in hardware or in software [Bel92, CG89, LH89, TKB92]. Gordon Bell predicts that: "Multicomputers from the score of companies combining computers will evolve to multiprocessors just to reduce overhead in simulating a single-memory address space, memory access, and supporting efficient multiprogramming" [Bel92].

As concurrent (shared memory) systems become popular, the task of implementing efficient public data structures for such systems is becoming important. As traditional data structures play an important role in the design of sequential algorithm, it is reasonable to expect that, the design of public data structures will be an important aspect in programming shared memory machines. We have investigated a deceptively simple public data structure: a counter which supports only two operations, increment by 1 and read. We have asked the following question: given small counters of type X, can we construct a big counter of type Y? We have looked only at the cases where X is of type atomic counter (or linearizable counter) and Y is either of type static counter or of type dynamic counter.

Many questions about public counters still remain open. First, what kind of dynamic counters can we construct from atomic counters when the atomic counters are initialized? A more general question is to consider other substitutions for X and Y in the question above, such as counters that support a wider variety of operations. For example, extending the counter definition to allow a decrement operation, which decreases the value of the counter by one, or a reset operation which set the counter to some default value.

A counter is a simple data structure whose implementation raises non-trivial problems. We believe that investigating such simple structures first, would be helpful in the development of more complicated public data structures.

References

[AA95] E. Aharonson and H. Attiya. Counting networks with arbitrary fan-out. *Distributed Computing,* Vol. 8, pp. 163–169, 1995. (Also in: *Proc. 3rd Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 104–113, January 1992.)

[AHS94] J. Aspnes, M. Herlihy and N. Shavit. Counting networks. *Journal of the ACM,* Vol. 41, No. 5, pp. 1020–1048, September 1994. (Also in: *Proc. 23rd ACM Symp. on Theory of Computing*, pages 348–358, May 1991.)

[AT93] R. Alur and G. Taubenfeld. How to share an object: A fast timing-based solution. In *Proc. 5th IEEE Symp. on Parallel and Distributed Processing*, December 1993.

[AVY94] W. Aiello, R. Venkatesan and M. Yung. Coins, weights and contention in balancing networks. *Proc. 13th Annual ACM Symp. on Principles of Distributed Computing,* pp. 193–205, August 1994.

[AW91] R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 370–380, May 1991.

[Bel92] G. Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):27–47, August 1992.

[BHM94] C. Busch, N. Hardavellas and M. Mavronicolas. Contention in counting networks. *Proc. 13th Annual ACM Symp. on Principles of Distributed Computing,* p. 404, August 1994.

[BIS95] A. Ben-Dor, A. Israeli and A. Shirazy. Dynamic Counting. In *Proc. 3rd Israel Symp. on the theory of computing and systems*, pages 111–120, January 1995.

[BM94] H. Brit and S. Moran. Wait-freedom vs. bounded wait-freedom in public data structures. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, August 1994.

[BM94b] C. Busch and M. Mavronicolas. A combinatorial treatment of balancing networks. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, August 1994.

[BM98] C. Busch and M. Mavronicolas. An Efficient Counting Network. *Proc. 1st Merged International Parallel processing Symp. and IEEE Symp. on Parallel and Distributed Processing,* pp. 380–385, May 1998.

[BS77] R. Bayer and M. Schkolnick. Concurrency operations on B-trees. *Acta Informatica*, 1(1):1–21, 1977.

[Bri94] H. Brit. Public data structure and public counters as a special case. Master's thesis, the Technion, Haifa, Israel, 1994.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[Ell80a] C. S. Ellis. Concurrency search and insert in 2-3 trees. *Acta Informatica*, 14:63–86, 1980.

[Ell80b] C. S. Ellis. Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers*, c-29:811–817, 1980.

[Ell85] C. S. Ellis. Distributed data structures: A case study. *IEEE Transactions on Computers*, c-34(12):1178–1185, 1985.

[FLL93] E. W. Felten, A. LaMarca and R. Ladner. Building counting networks from larger balancers. Technical Report TR-93-04-09, Department of Computer Science and Engineering, University of Washington, April 1993.

[HBS92] M. Herlihy, Lim. B., and N. Shavit. Low contention load balancing on large-scale multiprocessors. In *Proc. 3rd Annual ACM Symp. on Parallel Algorithms and Architectures*, July 1992.

[Her90] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proc. 2nd ACM Symp. on Principles and Practice of Parallel Programming*, pages 197–206, 1990.

[Her91a] M. Herlihy. A methodology for implementing highly concurrent objects. Technical Report CRL 91/10, Digital Equipment Corporation, October 1991.

[Her91b] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, January 1991.

[HKM93] N. Hardavellas, D. Karakos and M. Mavronicolas. Notes on sorting and counting networks. *Proc. 7th International Workshop on Distributed Algorithms (WDAG-93),* LNCS Vol. 725, Springer-Verlag, pp. 234–248, September 1993.

[HSW91] M. Herlihy, N. Shavit, and O. Waarts. Low contention linearizable counting. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 526–535, October 1991.

[HSW96] M. Herlihy, N. Shavit and O. Waarts. Linearizable counting networks. *Distributed Computing,* Vol. 9, pp. 193–203, 1996.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computations* Addison-Wesley, 1979.

[HW87] M. Herlihy and J. Wing. Axioms for concurrent objects. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 13–26, 1987.

[HW90] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[Kön36] D. König. Theorie der endlichen und unendlichen graphen. Liepzig, 1936. reprinted by Chelsea, 1950.

[Knu69] D. E. Knuth. *The Art of Computer Programming, vol. 2: Seminumerical Algorithms.* Addison-Wesley, 1969.

[KP92] M. Klugerman and C. Plaxton. Small-depth counting networks. In *Proc. 24rd ACM Symp. on Theory of Computing*, pages 417–428, 1992.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systems*, 5(2):190–222, 1983.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Programming Languages and Systems*, 7(4):321–359, 1989.

[LSST96] N. Lynch, N. Shavit, A. Shvartsman and D. Touitou. Counting networks are practically linearizable. *Proc. 15th Annual ACM Symp. on Principles of Distributed Computing,* pp. 280–289, May 1996.

[LY81] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[MB79] S. Mac-Lane and G. Birkhoff. *Algebra.* Macmillian, New York, 1979 (second edition).

[MMT99] M. Mavronicolas, M. Merritt and G. Taubenfeld. Sequentially consistent versus linearizable counting networks. *Proc. 18th annual symposium on principles of distributed computing,* May 1999, 133-142.

[MPT97] M. Mavronicolas, M. Papatriantafilou and Ph. Tsigas. The impact of timing on linearizability in counting networks. *Proc. 11th International Parallel Processing Symp.,* pp. 684–688, April 1997.

[MT97] S. Moran and G. Taubenfeld. A lower bound on wait-free counting. *Journal of Algorithms,* Vol. 24, pp. 1–19, 1997. (Also in: *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 251–260, August 1993.)

[MTY96] S. Moran, G. Taubenfeld and I. Yadin. Concurrent counting. *Journal of Computer and System Sciences,* Vol. 53, No. 1, pp. 61–78, August 1996. (Also in: *Proc. 11th ACM Symp. on Principles of Distributed Computing*, pages 59–70, August 1992.)

[Pel90] D. Peleg. Distributed data structures: A complexity-oriented structure. In *4rd International Workshop on Distributed Algorithms*, 1990. Lecture Notes in Computer Science, vol. 486, Springer-Verlag 1990, pages 71–89.

[Plo89] S. A. Plotikin. Sticky bits and universality of consensus. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 159–175, August 1989.

[Sag85] Y. Sagiv. Concurrent operations on B-trees with overtaking. In *ACM Principles of Dadabase Systems*, pages 28–374, January 1985.

[TKB92] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Balvrije. Parallel programming using shared objects. *IEEE Computer*, pages 10–19, August 1992.