Received: 27 October 2017 | Revised: 9 June 2018 | Accepted: 5 November 2018
DOI: 10.1002/cpe.5090

WILEY

RESEARCH ARTICLE

Accurate counting algorithm for high-speed parallel applications

Junchang Wang[1,2] | **Tao Li**[1,2] | **Xiong Fu**[1,2]

[1]School of Computer Science, Nanjing University of Posts and Telecommunications, Jiangsu, China
[2]Jiangsu Key Laboratory of Big Data Security and Intelligent Processing, Nanjing University of Posts and Telecommunications, Jiangsu, China

Correspondence
Junchang Wang, Box 844, School of Computer Science, Wenyuan Road 9, Nanjing University of Posts and Telecommunications, Nanjing, Jiangsu 210023, China.
Email: wangjc@njupt.edu.cn

Funding information
National Natural Science Foundation of China, Grant/Award Number: 61602264; China Postdoctoral Science Foundation, Grant/Award Number: 2017M611882

Summary

Statistical counter offers the appeal of an efficient and scalable counting mechanism on multi-core architectures where parallelism has been increasing sharply. Statistical counter has been widely used in practice (eg, in high-end network devices to count the number of packets received) despite the truth that it can only provide weak consistency guarantee on the counting results it returns, that is, statistical counter could miscount and the returned results may be inaccurate. As hardware and its parallelism advances, the miscount issue has raised concerns in both industry and academy. This paper is motivated by this real-world miscount issue that we were facing when building a high-speed intrusion detection system on a commercial multi-core server with 40Gbps NICs. To tackle the problem, we first systematically analyze the miscount issue and quantify the miscounts in counting results. Then, we present a novel counting algorithm that (1) is competitive to statistical counter in performance on multi-core architectures and (2) provides strong consistency guarantee on counting results returned. Experiments show that it takes the new counting algorithm 10ns and 1,500ns to perform an update and a read operation, respectively. Moreover, the counting results returned are accurate.

KEYWORDS

counting algorithm, lock-free data structure, multi-core, strong consistency

1 | INTRODUCTION

Counter is one of the fundamental data structures that almost every program relies on. Counting algorithm can be formulated as a reader-writer problem, which can be solved by properly protecting the shared resource, *counter*, to make the read and write operations atomic to avoid intermediate states. Over the past decades, a variety of counting algorithms have been invented[1-9] to meet requirements in algorithm efficiency, memory usage, and consistency of counting results. In the past decade, driven by Moore's Law, computer hardware and its parallelism evolved sharply. The increasing parallelism in computer systems challenges existing counting algorithms in properties such as algorithm efficiency and consistency of counting results. For example, once widely used *lock-based counter* protects the shared counter by utilizing a lock to serialize concurrent update requests. *Lock-based counter*, even though provides strong data consistency in the sense of returning counting results without miscounts, scales poorly on multi-core architectures due to the serialized update operations.

To overcome the performance bottleneck of *lock-based counter*, another mechanism, named *statistical counter*,[5,10] was proposed. It is worth noting that *statistical counter* too is a traditional algorithm and has been widely used in both applications and operation systems for scenarios where the counter is updated extremely frequently. For example, to count packets received/transmitted by a 10Gbps Network Interface Card (NIC henceforth), a counter will be incremented millions of times per second, by tens of updater threads simultaneously. *Statistical counter* achieves scalability by providing each updater thread a private counter, such that each updater thread increments its own counter, avoiding contentions between updater threads. When a read request arrives, the reader thread reads the values of all of the updater threads' counters out, sums them up, and returns the aggregated value, avoiding synchronization between the reader thread and updater threads.

Statistical counter, however, fundamentally trades off the consistency of counting results for algorithm efficiency because it has miscount issue in counting results returned. The reason of the miscounts is that while a reader is summing up updater threads' counters in order, these

Our algorithm, ECCount, is open source under GPL v2. The code is available at https://github.com/junchangwang/eccount

counters can be incremented by updater threads at the same time. Worst case happens when the reader thread unfortunately is scheduled out and suspend for a long time. The data miscount issue of *statistical counter* has been mentioned in literatures[5,10] and researchers often claims that the *statistical counter* provides a *weak data consistency* model, named *eventual consistency*[11,12] or *quiescent consistency*.[10] Users of this kind of counters are typically warned in advance about the miscounts in counting results they get.

Theoretically, the consistency definition for *statistical counter* is that given a read operation *A*, any read operation *B* such that the *end* of *A* happens before the *beginning* of *B* (with "happens before" being defined by the relevant memory model), then the value returned by *B* will be strictly greater than that returned by *A*, assuming that the two reads are spaced sufficiently closely to avoid counter wrap. This consistency definition, however, is different to what a programmer would expect. From a programmer's point of view, the consistency definition for a counter would be that the counting result of read operation *B* is larger than or equal to read operation *A*, only if the *beginning* of *A* happens before the *beginning* of *B* (read operation *A* is issued before B). That is, the consistency definition programmers expect is that given a read operation A, any read operation B such that the *beginning* of A happens before the *beginning* of B (with "beginning" being defined by starting reading counters, and "happens before" being defined by the relevant memory model), then the value returned by B will be strictly greater than that returned by A, assuming that the two reads are spaced sufficiently closely to avoid counter wrap. The time period between the *beginning* and the *end* of read operation A is relatively small when a *statistical counter* has a few counters in early days. However, with the rapid increase of hardware parallelism (and hence the number of counters in statistical counter), the time period between the *beginning* and the *end* of a read operation has been increasingly large, which is the root of miscounts in traditional counting algorithms. In subsequent sections, we refer to the consistency definition supported by *statistical counter* as *weak data consistency*, and consistency definition supported by ECCount as *strong data consistency*.

In recent years, hardware and its parallelism advance sharply, and the data inconsistency issue in *statistical counter* has raised concerns in both industry and academy. For example, in contrast to 1Gbps NIC, which provides a dozen of hardware queues, latest 40Gbps NICs from Intel[13] support in maximum 1,536 hardware queues and as a result, 1,536 concurrent updater threads in maximum. Theoretical analysis (Section 2) shows that as the number of updater threads increases, so does the miscount of *statistical counter*. Experiments (Section 5) show that on a server with simulated 40Gbps NIC and 16 hardware queues (updater threads), when the time period between two successive read requests become as small as 3 microseconds, up to 35% counting results are inaccurate and the average number of miscounts could be as high as hundreds of packets, which is unacceptable for applications that rely on consistent counting results. For example, load balancing.

Overall, existing counting algorithms are inadequate for one type of raising high-performance applications on multi-core architectures, in which (1) hundreds or even thousands of updater threads increment the counters simultaneously. For example, high-end NICs nowadays distributes hardware interrupts to as many CPU cores as possible[14] to avoiding overwhelming a single core. Each CPU core runs an updater thread, and as a result, tens of threads may increment the counters simultaneously. (2) The update operation must run extremely fast because it is in the fast path of these applications. For example, on a 1.7GHz CPU, to meet 40Gbps line-rate requirements, a packet's processing time must be less than 23 nanoseconds, forcing the update operation to complete in a few nanoseconds. (3) Strong consistency of counting result is required. For example, to do load balancing according the number of packets received/transmitted, miscount of hundreds of packets is unacceptable.

This paper is motivated by a real-world problem that we were facing when building a high-speed intrusion detection system on a commercial multi-core server with 40Gbps NICs. The driver of the 40Gbps NIC utilizes *statistical counter* to count the number of packets received by the NIC, and in some cases the application needs to read this statistical data out frequently. In the experiments, we found that miscount could happen when the application issues two subsequent read requests *A* and *B* in a short period; Counting result of read request *B* could be less than that of read request *A*, which confuses the application.

To tackle the problem, we invented ECCount*, an efficient and accurate counting algorithm, which not only scales as the number of updater threads increases, but also provides strong data consistency in counting results. The basic idea behind ECCount's strong consistency property is that updater threads' counters are organized as an array, and when a read request arrives, ECCount takes a "snapshot" of counting results by swapping the counter array which updater threads were updating with a new array which is just allocated. Then the reader thread accumulates counters in this "snapshot" and at the same time updater threads increment counters in the new allocated array, without incurring miscounts. This approach works well for use cases where there is few concurrent read requests (eg, tools such as *ifconfig* and *ethtool* to check the statistic information of a NIC). To support the corner case where concurrent read requests arrive at the system simultaneously, however, is challenging, given the strong data consistency requirement. Rather than giving up the simplicity and effectiveness of the strategy of taking "snapshot", ECCount adopts techniques such as global timestamps and customized lock-free linked list to allow reader requests to take snapshots, calculate counting results, and return results in a lock-free manner.

Experiments show that with ECCount, each update request takes less than 10 nanoseconds and each read request takes less than 1,480 nanoseconds to complete, in a stress testbed with 16 updater threads. In contrast, for traditional *statistical counter*, a update and a read operation takes 12 and 1,500 nanoseconds, respectively. Experiments show that ECCount is as efficient as *statistical counter*. Besides, ECCount does not suffer the miscount issue in *statistical counter*.

The major contributions of this paper are as follows:

- As the parallelism of hardware and applications keeps increasing, the data inconsistency issue in existing best counting algorithm, *statistical counter*, is becoming severe. This paper is the first research from literature to handle this problem.

* Source code and documents are available at https://github.com/junchangwang/eccount

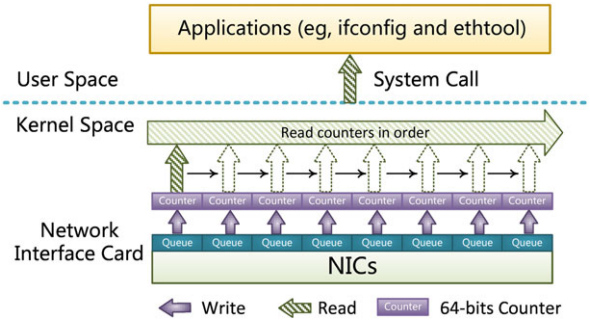

FIGURE 1 *Statistical counter* in packet receiving example. While a reader is summing up updater threads' counters in order, these counters can be incremented by updater threads at the same time, which is the root of miscounts in *statistical counter*

```
___#|_______|________|________|________$___ -->
```

FIGURE 2 Time sequence of a read request. Assume the system has 4 counters, and each counter takes the reader 8 time slots (_) to get the value. Miscounts happen in between # and $

- We quantify the miscounts in *statistical counter* in Section 2. This could be very useful for other researchers to understand and evaluate their counting algorithms.
- A novel counting algorithm, ECCount, is presented for high-speed applications running on multi-core architectures. ECCount can not only scales as the number of updater threads increases, but also provides strong consistency guarantee in counting results returned. ECCount is open source under GPL v2 and can serve as an example for researchers who want to build algorithms that are both efficient and accurate.

The rest of the paper is organized as follows. Section 2 details the data inconsistency issue of *statistical counter* and quantify the miscounts. Section 3 and 4 present the details of ECCount algorithm and a brief correctness proof, respectively. We present evaluations in Section 5, discuss related work in Section 6, and conclude in Section 7.

2 | CHALLENGES OF COUNTING ON MULTI-CORE ARCHITECTURES

This section details challenges of counting on multi-core architectures, and highlights drawbacks existing counters are facing. It is generally accepted that a *lock-based counter* is inappropriate for multi-core architectures. So this section focuses on *statistical counter* and its challenges.

In subsequent sections, we take the use case of counting the number of packets arriving at a NIC as example. The architecture of *statistical counter* is shown in Figure 1. When a packet arrives at the NIC, it is distributed to one of hardware queues (Queues in Figure 1). Each queue has a dedicated 64-bits counter. Upon receiving the packet, the hardware queue increments its own counter. If a user-space application, such as *ifconfig* or *ethtool*, wants to check the number of packets received, it invokes the system call and a reader thread is created. The reader thread reads each updater thread's counter in order, accumulates them, and then returns result.

In an ideal world, a read request should get accurate counting result; when the read request arrives, the sum of the value of all counters should be returned. Unfortunately, in *statistical counter*, it takes a reader a few microseconds (and longer if the reader thread is scheduled out) to sum up per-writer counters, and during this time period, updater threads are still incrementing the counters, making the returned value larger than the expected one, which is the root of miscounts in *statistical counter*. In the following subsection, we quantify the miscounts in results returned by *statistical counter*.

2.1 | Miscounts in Statistical Counter

We use Figure 2 to help demonstrate miscounts in *statistical counter*. In the figure, time clock increases from left to right. Symbol "#" indicates the *beginning* of a read request, symbol "|" indicates the moment the reader starts reading a updater thread's counter, and subsequent underscores "_" indicate time slots the reader has to wait to get the value of this counter, due to for example cache miss. The last symbol "$" means the reader gets the value of the last counter and will return accumulated result immediately. That is, "$" represents the *end* of the read request. In most implementations, the very first instruction of read operation is to start reading the value of the first counter, such that the time slot between "#" and the first read "|" can be ignored. So when we talk about absolute error, we are talking about the miscounts happened in between reading the first counter (indicated by symbol "#") and completing reading the last counter (indicated by symbol "$").

Assume that a *statistical counter* has N counters, and in overall the counter is incremented for each t nanoseconds. For simplicity, let's assume that the overall counters are evenly distributed to the N counters. So for each counter, it is incremented for each tN nanoseconds. A read request takes Δ nanoseconds to finish summing up all of the updater threads' counters. In such case, it takes the reader $\frac{\Delta}{N}$ nanoseconds to finish reading

a single counter. For simplicity, we assume the read operation to a single counter completes immediately, but is then delayed $\frac{\Delta}{N}$ seconds before returning, due to, for example, cache miss. Then the miscounts of the second counter are $\frac{\Delta}{tN^2}$, due to the delay of reading the first counter, miscounts of the third counter are $\frac{2\Delta}{tN^2}$, and so on, which is demonstrated in Figure 1. The overall miscounts is given by summing up miscounts of all of the counters, which is:

$$Overall\ miscounts = \frac{\Delta}{tN^2}\sum_{i=1}^{N} i \tag{1}$$

Expression the summing in closed form yields:

$$Overall\ miscounts = \frac{\Delta}{tN^2}\frac{N(N+1)}{2} \tag{2}$$

Canceling yields the intuitively expect results:

$$Overall\ miscounts = \frac{\Delta}{tN}\frac{N+1}{2} \tag{3}$$

Equation (3) shows that as the volume of data to be processed increases heavily, so does the overall miscounts in counting results returned. For example, for 40Gbps network links which is becoming ubiquitous nowadays, packets arrive at the NIC for each 23 nanoseconds, making the t in Equation (3) to be 23 nanoseconds. Let's assume a machine with 16 cores is used to handle incoming packets. Experimentally we found that a reader takes about 2,000 nanoseconds to finish summing up all of the 16 counters and return, due to cache misses. In this case, we get the expect miscounts:

$$\frac{2690}{23 * 16}\frac{(16+1)}{2} = 62 \tag{4}$$

It is worth noting that Δ increases nearly linearly to the number of hardware queues (updater threads). The major reason is that the value of Δ is largely due to cache misses when the reader accesses updater threads' counters, which are in most of the time written solely by updater threads. As a result, as hardware advances (eg, latest 40Gbps NIC typically supports thousands of hardware queues) and the number of updater threads increases in the system, programmers will be confronted with much more severe miscount issue.

3 | THE ECCOUNT ALGORITHM

ECCount is built upon *statistical counter* which is scalable on multi-core architectures and has been widely adopted as the counting algorithm on multi-core platforms. ECCount is competitive to *statistical counter* in performance. And at the same time, ECCount addresses the miscount issue in *statistical counter*.

We choose to present out ideas of ECCount in steps, starting from a simple yet useful version of ECCount for scenarios where read requests have been serialized, to make understanding easier. This version is referred to as *ECCounts-SR*, short for *ECCounts-single-reader version*. This version works well for scenarios where there is few concurrent read requests, which is the case for most network interface management tools such as *ifconfig* and *ethtool*. After that, we present another ECCount version which supports concurrent read requests (henceforth, *ECCounts-MR*), which can be viewed as the ‘‘composition’’ of ECCounts-SR.

3.1 | ECCounts-SR

Root of miscounts in *statistical counter* The root of miscounts in *statistical counter* is that, while a reader thread is summing up the array of counters, which typically takes thousands of CPU cycles, updater threads can increment the counters at the same time, making the results returned by the read request inaccurate.

Basic idea of ECCounts-SR To tackle this inconsistency issue, ECCount introduces the concept of ‘‘taking snapshot’’ in counting. The basic idea behind ECCounts's strong consistency property is that updater threads' counters are organized as an array, and when a read request arrives, ECCount takes a ‘‘snapshot’’ of counting result by swapping the counter array which updater threads were updating with a new array which is just allocated. Swapping the two arrays, in practice, could be achieved by inverting a global shared variable with a single atomic instruction, which prevents miscounts from happening.

Basic architecture of ECCounts-SR Figure 3 illustrates the basic architecture of ECCounts-SR, which consists of a pair of counter arrays, *countersA* and *countersB*, in addition to a global index named *ABflag*, which indicates one counter array for updater thread to increment, and another array for reader threads to read the value of counters out. The design philosophy of ECCount-SR is to trade off space for parallelism; By doubling the counter array, ECCounts-SR separates the resource on which reader threads and updater threads operate, allowing reader and updater threads proceed in parallel without miscounts.

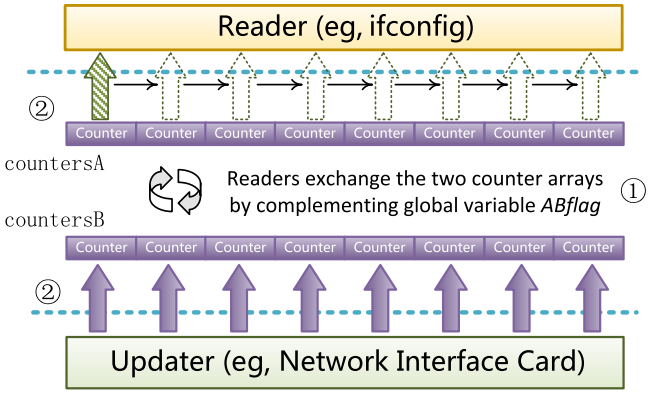

FIGURE 3 Overall architecture of ECCount-SR

Algorithm overview To increment the counter, an updater first references to the counter array by checking global variable *ABflag*, and then increments its corresponding counter in this array. Each time a read request arrives, the reader thread first complements the value of *ABflag*, which prevents updater threads from incrementing the array they were updating any more. We name this procedure "*taking a snapshot*". For example, suppose that the value of *ABflag* was 0 and the updater threads were incrementing counter array *countersA*, then when a read request arrives, it sets the value of *ABflag* as 1, which takes a snapshot of the current counting result in the sense that for any subsequent increments, updater threads check variable *ABflag*, get value 1, and as a result, update counters in *countersB*. That means the value of counters in *countersA* will no longer be incremented. It worth noting that a reader "takes a snapshot" by inverting the variable *ABflag*, which commonly takes a few CPU cycles.

Algorithm 1 shows the update primitive of ECCounts-SR. The API *update()* takes the thread number *t* as parameter. Line 1 reads the value of global variable *ABflag*. If the value of *ABflag* equals to 0, updater increments corresponding counter in counter array *countersA*. Otherwise, updater increments *countersB*.

Algorithm 1 update(t) of ECCounts-SR

```
Parameters: t (thread number)
Variables   : ABflag (global integer)
              countersA (global long integer array)
              countersB (global long integer array)
1 if(ABflag == 0){
2     countersA[t] ++;
3 }
4 else{
5     countersB[t] ++;
6 }
```

Algorithm 1 is simple and efficient; It checks a global variable, *ABflag*, and then increments corresponding counter. In most cases both the *ABflag* and corresponding counter reside in updater's CPU cache, given read operations rarely happen. As a result, function update() of ECCount runs extremely fast. Experiments (Section 5) show that the update() of ECCount takes about 10 nanosecond to complete, no matter the number of concurrent readers and updater threads in the system.

It is worth noting that if a long-term preemption occurs between reading the value of *ABflag* and incrementing corresponding counters, read operations during this period cannot count this increment; This particular increment will be lost until the following reads. This inaccuracy issue can be solved by asking programmers to perform a pair of reads, one after the other, to check if there are any pending increments. This issue in update, however, does not compromise the strong data consistency property of ECCounts.

Algorithm 2 shows the reader-side primitive of ECCounts-SR. Lines 1 and 2 initialize two local variables, *sum* and *tmp*, respectively. Line 3 – 8 complement the global variable *ABflag*, which takes the snapshot of current counting results by indicating subsequent updates to another counter array. Line 9 is a memory barrier which guarantees that preceding updates to variable ABflag to be performed before subsequent operations. Line 10 – 19 accumulate the counter array that was incremented by updater threads. Global variable *older_snapshot* is used to record the counting results of current snapshot for later use.

3.2 | ECCounts-MR

ECCounts-SR works well for most scenarios where there is few concurrent read requests. For example, configuration tools such as *ifconfig*. To meet remaining scenarios where there are concurrent read requests, this section presents ECCounts-MR, a counting algorithm built upon ECCounts-SR and can serve concurrent read requests.

Algorithm 2 read() of ECCount-SR

```
Variables : ABflag (global integer)
            countersA (global long integer array)
            countersB (global long integer array)
            older_snapshot (global long integer)
            sum, tmp (local long integer)
1  sum = 0;
2  tmp = 0;
   /* Complement ABflag.                                                    */
3  if( ABflag == 1 ){
4      ABflag = 0;
5  }
6  else{
7      ABflag = 1;
8  }
9  smp_mb();
   /* Sum up counters in snapshot.                                          */
10 if(ABflag == 1){
11     for(each counter t in countersA){
12         sum += countersA[t];
13     }
14 }
15 else{
16     for(each counter t in countersB){
17         sum += countersB[t];
18     }
19 }
   /* Save result on older_snapshot for later use.                          */
20 tmp = sum;
21 sum += older_snapshot;
22 older_snapshot = tmp;
23 return sum;
```

We recall that the design goal of ECCounts-SR is to provide efficient and consistent counting results. The design of ECCounts-MR is much more challenging because the algorithm not only needs to provide efficient and accurate counting results, as ECCounts-SR does, but also needs to provide accurate counting results while serving concurrent read requests. For example, the consistency requirement prevents the use of lock to serialize concurrent readers, because locking incurs unexpected delays, which in turn introduces miscounts.

In summary, ECCounts-MR must meet the following design criteria: (1) to provide consistent counting results, (2) update operation is efficient, and (3) read operation is non-blocking and the results returned reflect the order read requests arrive at the system.

3.3 | ECCounts-MR overview

Rather than giving up the simplicity and effectiveness of ECCounts-SR, which meets the first two design criteria, we invented ECCounts-MR, a high-performance counting mechanism that addresses the concurrency issues by parallelizing ECCounts-SR and then ''orchestrating'' multiple ECCounts-SR instances, each of which serves one read request.

Parallelize ECCounts-SR. To run multiple instances of ECCounts-SR simultaneously in the system, we first identify the sequential components and operations in ECCounts-SR, and replace them with parallelized counterparts to make ECCounts-SRs capable to run in parallel. Specifically, we performed the following updates:

- In ECCounts-SR, two counter arrays are adequate for taking snapshot—one for reader and another for updaters. However, in ECCounts-MR, since multiple readers can take snapshots simultaneously, for each reader, a counter array is required. And as a result, we use a simple yet efficient non-blocking memory allocator to dynamically allocate and free counter arrays.
- In ECCounts-SR, swapping two counter arrays can be achieved by inverting a shared variable, *ABflag*. In ECCounts-MR, however, to indicate the counter array that updaters are incrementing, the global variable *ABflag* is replaced by a global pointer to the counter array that updaters are incrementing.

Orchestrate ECCounts-SR instances. In ECCounts-MR, when a read request arrives, an instance of ECCounts-SR is allocated. To meet the consistency requirement, one needs to make sure read operations take effect (read results) in the order they arrive at the system. To achieve that, in ECCounts-MR, a field, *timestamp*, is added to distinguish ECCounts-SR instances in the system. The value of *timestamp* is monotonically increased by utilizing atomic primitives such as *Fetch_And_Add(FAA)*. When a new read request arrives, ECCounts-MR gets a timestamp value and assign it to the related ECCounts-SR instance.

Retrieve counting results. Finally, since in ECCounts-MR, multiple readers can take snapshots simultaneously, to get counting result, a reader needs to sum up not only counters in its own snapshot, but also counters in snapshots hold by other readers. As a result, even if a read request completes, the snapshot it hold cannot be freed. Instead, the snapshot must be saved. In ECCounts-MR, a lock-free linked list is used to save

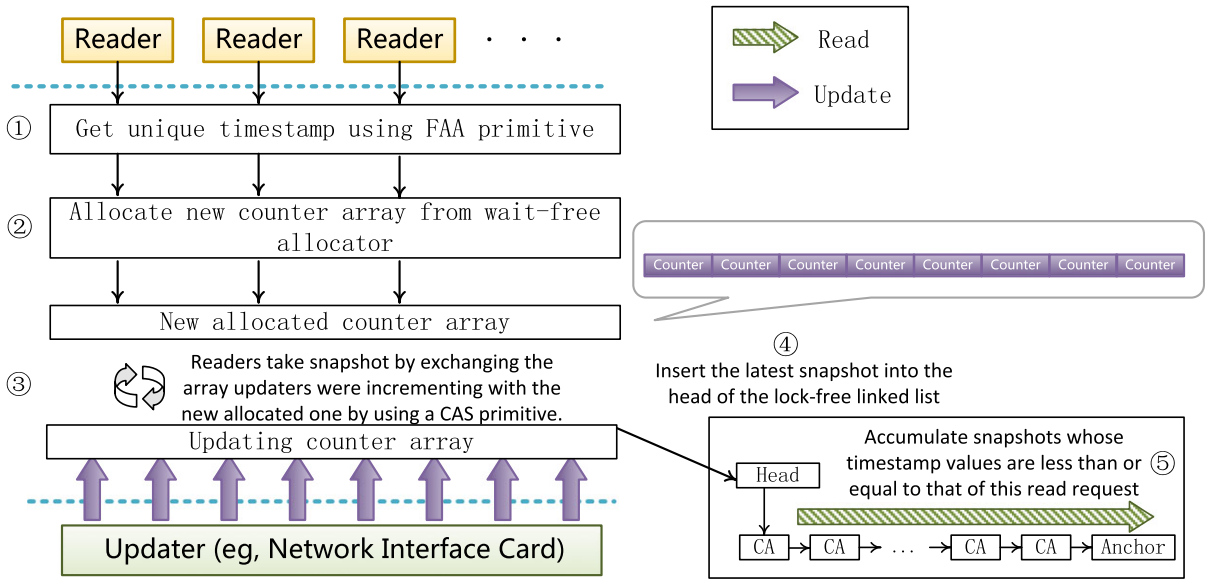

FIGURE 4 Overall architecture of ECCount-MR

snapshots. Instead of utilizing a general-purpose lock-free design, ECCounts-MR invents a customized linked list that is efficient and scalable (discussed in detail in Section 3.5).

In summary, Figure 4 illustrates the overall architecture of ECCounts-MR. When the system starts, a counter array is allocated and assigned to updaters for incrementing, denoted as *Updating counter array* in Figure 4. Each time a read request arrives at the system (eg, when a administrator types command *ifconfig*), a reader thread is created, which first retrieves a unique timestamp value by invoking atomic primitive *Fetch_And_Add* (step 1 in the Figure). And then the reader tries to allocate a new snapshot, which contains a new counter array, from a wait-free slab allocator (step 2 in the Figure). After that, the reader tries to take snapshot by swapping the counter array, which updaters were incrementing, with the new allocated one by using *Compare-And-Swap* instruction (step 3 in the Figure). If success, the reader now holds a snapshot which contains a valid counter array which were incrementing by updaters and will not be touched by updaters any longer. Since this snapshot could be used by subsequent readers (discussed in details in subsequent subsection), the reader inserts the snapshot into the head of a global lock-free linked list (step 4 in the Figure). Finally, the reader calculates the counting result by walking through the list and accumulating snapshots whose timestamp values are less than that of this read request (step 5 in the figure). After that, the reader can return the counting results.

3.4 | ECCounts-MR algorithm

The pseudocode of the basic ECCounts-MR algorithm appears in Algorithms 3, 4, 5, and 6.

Algorithm 3 Data structures and global variables of ECCount-MR

```
1  struct counter{
2      v : unsigned long integer;
3      // padded to cache line size;
4  };
5  struct snapshot{
6      reader_leaved : integer;
7      timestamp : unsigned long integer;
8      counterArray : array of counters;
9      next : pointer to snapshot, initally NULL;
10 };
11 global_updating_snapshot : global pointer indicating the counter array which updaters are incrementing;
12 global_old_snapshots : global pointer indicating old snapshots list;
13 global_timestamp : global shared timestamp;
```

Algorithm 4 update(t) of ECCount-MR

```
Parameters: t (thread number)
1  global_updating_snapshot→counterArray[t].v ++;
```

Algorithm 5 read() of ECCount-MR

```
1 new_snapshot = snapshot_malloc();
2 new_snapshot.timestamp = FAA(&global_timestamp, 1);
3 while(TRUE){
4     if(SWAP(global_updating_snapshot, new_snapshot)){
5         break;
6     }
7 }
  /* Now, new_snapshot points to the latest snapshot.                    */
  /* We then insert this snapshot into global linked                     */
  /* list, global_old_snapshots.                                         */
8 insert (global_old_snapshots, new_snapshot);
  /* Sum up snapshots whose timestamp values are                         */
  /* less than or equal to that of new_snapshot,                         */
  /* in linked list global_old_snapshots.                                */
9 sum = sumup (new_snapshot.timestamp);
  /* Set the reader_leaved field of new_snapshot as TRUE                 */
  /* before leaving, to allow the system to free the memory              */
  /* pointed by new_snapshot.                                            */
10 new_snapshot.reader_leaved = TRUE;
11 return sum;
```

Algorithm 6 sumup(timestamp)

```
   Parameters: timestamp (timestamp threshold)
1  for(each snapshot in global_old_snapshots){
2      if(snapshot.timestamp > timestamp){
3          continue;
4      }
5      else{
6          for(each possible thread t in snapshot.counter){
7              sum += snapshot.counter[t].v;
8          }
9      }
10 }
11 return sum;
```

Data structure and global variables. Algorithm 3 presents the key data structure in ECCounts-MR. For each read request, an instance of *snapshot* is allocated. A *snapshot* can be in one of two states: if the value of *reader_leaved* is 0, the reader request who creates this instance is not complete yet; otherwise, this instance is safe to free. Variable *timestamp* records the timestamp value when this read request arrives. *counterArray* points to an array of counters. The size of *counterArray* equals to the number of active updaters. Global pointer *global_updating_snapshot* always points to a snapshot instance whose counters are incrementing by updaters. It is worth noting that readers do not touch this instance. Each time a read request arrives, it takes a snapshot by swapping a new snapshot with the one pointed by *global_updating_snapshot*. After that, the snapshot that was pointed by *global_updating_snapshot* is added to a global linked list *global_older_snapshots* and is used by subsequent read requests to count.

Update. Operation update(t) (shown in Algorithm 4) is similar to its counterpart in ECCounts-SR. It first locates the snapshot instance that is currently for updating by checking *global_updating_snapshot*. Then the counter that is for this updater is found and incremented.

Read. Operation read() (shown in Algorithm 5) is similar to its counterpart in ECCounts-SR, except that it has to orchestrate multiple snapshot instances. When a read request arrives, it first allocates a new snapshot, and then retrieves a unique *timestamp* value. In an extreme case (eg, tens of thousands of read requests issued simultaneously), the memory allocation in ECCount-MR may fail. In this case, ECCount (1) gives up allocating new counter array and generating new snapshot, (2) sums up counters in global old_snapshot list in addition to counters in the updating_counter array, and then (3) returns the counting result immediately. For this path, ECCount deteriorates to statistical counter and can provides a weak data consistency in counting results. Lines 3-7 try to swap the new allocated snapshot with the one updaters were updating. This is similar to complementing the *ABflag* in ECCounts-SR, except that an atomic CAS primitive is required because multiple read requests can run this code snippet simultaneously. If success, *new_snapshot* now points to the snapshot which updater threads were updating. We then add this snapshot into global linked list, *global_old_snapshots*, in line 8. This linked list is to record old snapshots for upcoming read requests to accumulate, and thus the list could be any lock-free linked list. We present a fast lock-free linked list implementation in Section 3.5. Line 9 calls the sumup function, which is presented in Algorithm 6. The goal of this algorithm is to walk through the old snapshots list *global_old_snapshots* and sum up counters of snapshots whose timestamp values are less than or equal to the timestamp value of the read request. After that, a read request's job is mainly done; it will not touch global variables anymore and its snapshot instance linked in *global_old_snapshots* become ready to be freed if appropriate (detailed in Section 3.5). So, it sets field *reader_leaved* in its snapshot in line 10, and then returns the counting result.

3.5 | Optimizations for read() of ECCounts-MR

ECCounts-MR algorithm presented in Section 3.4 has already been able to (1) provide consistent counting results, (2) perform increment operations efficiently, and (3) provide strong consistent counting results in the sense that read operation is non-blocking and that the results returned reflect the order read requests arrive at the system. However, careful readers may notice that function read() of ECCounts-MR (shown in Algorithm 5) is inefficient due to the following reasons:

- For each read request, ECCounts-MR dynamically allocates a snapshot instance (line 1 of Algorithm 5), which is time consuming if a naive memory allocator is used (eg, system call *malloc()*).
- ECCounts-MR adopts a lock-free linked list, *global_old_snapshots*, to maintain old snapshots in the system. General purpose lock-free linked list,[15,16] however, is inefficient because it must handle corner cases such as concurrent insertion operation and deletion operation, and concurrent deletion operations (line 8 of Algorithm 5).
- To get the final counting result, ECCounts-MR needs to walk through the linked list which is time consuming because the list could be very long and cache misses arise in traveling the list (line 9 of Algorithm 5).

3.5.1 | Snapshot slab allocator

Since system call *malloc()* contains locks, it is not appropriate for ECCounts. ECCounts-MR adopts a customized slab allocator to manage snapshots. When the system starts, ECCounts-MR pre-allocates an array of snapshots, and then the subsequent snapshot allocation is done locally. When an snapshot is no longer used, it is returned to the slab allocator. As a result, the time complexity of snapshot allocation and free is *O(1)*.

3.5.2 | Customized non-blocking linked list

General lock-free linked list solutions[15,16] are complex and costly because they have to solve corner cases such as concurrent deletions, and interleaving insertion and deletion. In ECCounts-MR, we invented a customized non-blocking linked list, which is lock-free and efficient, by utilizing the following domain knowledge of ECCounts-MR to avoid handling corner cases.

- **No concurrent deletions.** Experiments show that a single dedicated *cleanup thread* is adequate to maintain a reasonably short list. As a result, in ECCounts-MR, only one thread, *cleanup thread*, performs deletion operation. So it is not necessary for the non-blocking linked list in ECCounts-MR to support concurrent deletions, one of the most complex corner cases to handle in designing lock-free linked list. This customized optimization can dramatically simplify the lock-free linked list design and boost the performance of function read().
- **No interleaving insertion and deletion.** In ECCounts-MR, deletions always happen at the tail of the list, and additions mostly happen at the head of the list. If incidentally a reader request with timestamp value t delays for a long period, the snapshot with timestamp value of t is missing in the tail of the in-order linked list, which prevents the cleanup threads from merging snapshots with timestamp values $t - 1$, t and $t + 1$. Instead, the *cleanup thread* returns and gives missing snapshots chances to be inserted, avoiding the deletion and insertion operations happen at the same snapshot (that is, snapshots with timestamp values $t - 1$ and $t + 1$.) As a result, it is not necessary for ECCounts-MR's non-blocking linked list to support interleaving insertion and deletion, another complex corner case to handle in designing lock-free linked list.

Algorithm 7 presents the insertion of the customized non-blocking linked list, which inserts a new snapshot *snapshot* into the global lock-free linked list *global_old_snapshots*. It repeats the following. The algorithm first compares the timestamp values of the new snapshot and snapshots in *global_old_snapshots*. If the timestamp value of the head snapshot of *global_old_snapshots* is less than that of *snapshot*, which is the common case, the new snapshot will be inserted into the head of *global_old_snapshots* by using a CAS primitive (lines 8–13). Otherwise, other snapshots with larger timestamp values have been inserted into *global_old_snapshots*, and as a result, we first search the right position (lines 17–20) to insert *snapshot* to maintain a in-order linked list, and then insert *snapshot* into the right place which is indicated by *rep* by using a CAS primitive (line 22.)

3.5.3 | Shrink linked list size

In ECCounts-MR, each time a read request arrives, a snapshot is inserted into the lock-free linked list. As a result, each time a read request wants to get the counting result, it has to walk through all of the old snapshots, which is time consuming. To solve this performance issue, ECCounts-MR cleanups old snapshots, if possible and appropriate. Specifically, ECCounts-MR creates a dedicated *cleanup thread*, which iteratively (1) searches for a series of snapshots which can be freed, and (2) invokes helper function *free_snapshots_RCU* to free the snapshots.

Algorithm 8 presents the pseudocode of the *cleanup thread*. It is worth noting that the cleanup thread can free a series of snapshots in each iteration. However, being an example, Algorithm 8 only cleans up the last and the second to the last snapshots. Specifically, Algorithm 8 repeats the following. The algorithm first walks through *global_old_snapshots* and locates the last snapshot, *repp*, and the second to the last snapshot, *rep* (lines 8 – 12). It is worth noting that this walk-through procedure can be optimized by keeping a reference to the tail of *global_old_snapshots* or by using a doubly linked list. For simplicity, we choose classic singly linked list in this paper. After that, the algorithm checks the length

of *global_old_snapshots* (line 13 – 15), and returns if its length is too small. This is a shortcut to predicting interleaving insertion and deletion operations, given deletions always happen at the tail of the list, and insertions almost at the head. This optimization indicates the cleanup thread to give up if it probably will conflict with other insertion operations, which can boost overall system performance. The micro LENGTH is configurable and is set to 16 by default. Then, the algorithm checks if there are missing snapshots which should be inserted in between *rep* and *repp* by checking the timestamp values (lines 16 – 18). Missing timestamp values in between the timestamp values of *rep* and *repp* means some snapshots, which should be inserted in between *rep* and *repp*, have not been inserted into the list yet, due to unexpected delays. The algorithm then returns and gives missing snapshots chances to be inserted.

Algorithm 7 insert(snapshot)

```
   Parameters: snapshot : pointer to the snapshot to be inserted into lock-free linked list global_old_snapshots
 1 // global variables
 2 global_old_snapshots : pointer to global lock-free linked list
 3 // local variables
 4 old_snapshot_t : pointer to struct snapshot
 5 rep, repp : pointers to struct snapshot
 6 while(TRUE){
 7     old_snapshot_t = global_old_snapshots;
 8     if(old_snapshot_t.timestamp < snapshot.timestamp){
           /* This is common case; snapshot is inserted at head.                    */
 9         snapshot.next = old_snapshot_t;
10         if(CAS(global_old_snapshot, old_snapshot_t, snapshot)){
11             Break;
12         }
13     }
14     else if(old_snapshot_t.timestamp > snapshot.timestamp){
           /* This is corner case. We first search                                  */
           /* the right position to insert this snapshot.                           */
15         rep = old_snapshot_t;
16         repp = rep.next;
17         while(repp.timestamp > snapshot.timestamp){
18             rep = repp;
19             repp = repp.next;
20         }
21         snapshot.next = repp;
22         if(CAS(rep.next, repp, snapshot)){
23             Break;
24         }
25     }
26 }
```

Algorithm 8 cleanup() : helper thread which cleans up two snapshots in each iteration.

```
 1 // global variables
 2 global_old_snapshots : pointer to global lock-free linked list
 3 // local variables
 4 rep, repp : snapshot pointers
 5 list_length : integer
 6 while(TRUE){
       /* Locate the last and the second to the last snapshots.                   */
 7     rep = repp = global_old_snapshots;
 8     while( repp.next != NULL){
 9         rep = repp;
10         repp = repp.next;
11         list_length ++;
12     }
       /* Not necessary to perform cleanup                                        */
13     if(list_length < LENGTH){
14         Break;
15     }
16     if( rep.timestamp != repp.timestamp + 1){
           /* There are missing snapshots which should be inserted                */
           /* in between rep and repp. Return and give the missing snapshots      */
           /* chances to be inserted.                                             */
17         Break;
18     }
19     free_snapshots_RCU(repp, rep);
20 }
```

Finally, the cleanup thread starts the procedure of deleting snapshots *rep* and *repp* by invoking helper function *free_snapshots_RCU()*. The deletion procedure, however, must be carefully designed to prevent other readers and updaters from accessing these snapshots that have been freed. The deletion operation is presented in details in next subsection.

3.5.4 | Free old snapshots

One of the biggest challenges in lock-free algorithm design is to free resources, where it is typically hard for a thread to check if other threads are holding references to the resources, given the algorithm must run in a lock-free and efficient manner. Specifically, in ECCounts-MR, other reader threads and updater threads may are holding references to snapshots which the cleanup thread is going to reclaim. In such a case, the *cleanup thread* must wait before all of these threads have left before freeing snapshots. To tackle this problem, inspired by the classic read-copy-update (RCU) synchronization mechanism,[17] ECCount adopts a novel RCU-like strategy in safely freeing snapshots *rep* and *repp*. Specifically, function *free_snapshots_RCU()* merges a series of snapshots by performing the following steps: (1) sum up the counting values of these snapshots and record the results on a new-allocated snapshot; (2) replace these old snapshots at the tail of the linked list with the new allocated one; (3) free the old snapshots, if no other reader/updater threads are accessing them.

Algorithm 9 free_snapshots_RCU(snapshot *repp, snapshot *rep)

```
   Parameters: repp, rep : pointers to the snapshots to be freed
   /* Allocate a new snapshot new_anchor.                                   */
   /* Accumulate counting results of rep and repp,                          */
   /* and record the results on new_anchor.                                 */
 1 new_anchor = snapshot_malloc ();
 2 if(! new_anchor){
 3     Return;
 4 }
 5 new_anchor.reader_leaved = 1;
 6 new_anchor.timestamp = rep.timestamp;
 7 foreach(counter t in counterArray){
 8     new_anchor.counterArray[t].v =
 9      rep.counterArray[t].v + repp.counterArray[t].v;
10 }
   /* Swap new_anchor with rep and repp                                     */
11 if(! CAS(&rep, rep, new_anchor)){
12     snapshot_free (new_anchor);
13     Return;
14 }
   /* Wait for exsiting readers, which may are accessing                    */
   /* rep and repp, complete and return.                                    */
15 timestamp_t = global_timestamp;
16 foreach( snapshot spt in global_old_snapshots){
17     while( (spt.timestamp < timestamp_t)
18      && spt.reader_leaved != 1){
19         poll(1);
20     }
21 }
   /* Wait for updaters, which may hold a retired reference to              */
   /* rep and repp, complete.                                               */
22 new_anchor = snapshot_malloc ();
23 if(! new_anchor){
24     Return;
25 }
26 foreach( snapshot spt in global_old_snapshots
27  && spt.timestamp > timestamp_t){
28     foreach(counter t in counterArray){
29         new_anchor.counterArray[t].v += spt.counterArray[t].v;
30     }
31 }
32 foreach(counter t in counterArray){
33     if( new_anchor.counterArray[t].v = 0){
34         Return;
35     }
36 }
   /* Now, it is safe to free rep and repp.                                 */
37 snapshot_free (repp);
38 snapshot_free (rep);
```

The pseudocode is presented in Algorithm 9. Specifically, to free old snapshots, *cleanup threads* do the following.

- The *cleanup thread* first allocates a new snapshot, *new_anchor* in line 1, and accumulates counting results of *rep* and *repp*, which are going to be reclaimed, and records the results on *new_anchor* (lines 7 – 10.)
- The *cleanup thread* then tries to replace the two snapshots *rep* and *repp* with the new allocated one, *new_anchor* (lines 11 – 14). If success, subsequent read requests will access *new_anchor*, instead of *rep* and *repp*. Otherwise, the *cleanup thread* returns and will try again later. Since subsequent reader/updater threads will not touch snapshots *rep* and *repp* anymore, it is safe for ECCount to reclaim these snapshots only if preceding reader threads and updater threads have completed and left.

- To check preceding readers, which may are holding references to snapshots *rep* or *repp*, have completed and left, *cleanup thread* waits until these read requests mark their *reader_leaved* field in their snapshot (lines 15 – 21). This field is marked just before a read request is going to return.
- Similarly, in rare cases, updater threads may delay unexpectedly, and as a result, it may hold a retired reference to *rep* or *repp* even after these two snapshots have been replaced by *new_anchor*. To solve this issue, the *cleanup thread* walks though *global_old_snapshots* and checks that since the swap operation happens, each updater has successfully update at least once and does not hold the retired references any more (lines 22 – 36). It is worth noting that if some updater threads refuse to do further updates, then snapshots *rep* or *repp* cannot be freed. They will be freed when the updater threads become online again and sucessfully update counters at least once.

It is worth nothing that algorithm cleanup can be optimized. For example, the algorithm walks through list *global_old_snapshots* multiple times. In practice, they can be merged and the algorithm only needs to walk through the list once. Another example is that the algorithm deletes two snapshots for each iteration, but in practice, it can delete a serial of snapshots only if there are no missing snapshots in between them.

4 | CORRECTNESS

To prove the correctness of ECCounts, we recall that ECCount is a counting algorithm which is efficient and can provide strong consistency guarantee on counting results returned. By correctness of ECCounts, we mean that the counting results returned are monotonically increased, and the algorithms (both function read() and update()) run in a lock-free manner. Due to space limitation, we regard the execution of ECCount algorithm is consistent with its program order. Besides, it is also reasonable to assume that aligned word-sized accesses are atomic (eg, accesses to long integers in X86 architectures). These assumptions are justified on modern multi-core processors by using helper functions such as READ_ONCE and WRITE_ONCE.[18]

4.1 | Non-blocking

Function update() of ECCounts-SR (Algorithm 1) is *wait-free* because it is guaranteed to complete within finite steps. What function update() does is to (1) locate the updating counter array, and (2) increment corresponding counter. Function read() of ECCounts-SR (Algorithm 2) is *wait-free* because there is no critical sections and lock operations, and the algorithm is guaranteed to complete within finite steps. These properties guarantee that ECCounts-SR takes finite steps to finish, making ECCounts-SR a good candidate for high-speed applications.

Similarly, function update() of ECCounts-MR (Algorithm 4) is *wait-free*. Function read() of ECCounts-MR is *lock-free*. It is worth noting that the slab allocator works on a pre-allocated array of snapshots (line 1 of Algorithm 5), such that the time to allocate a snapshot is fixed and the time complexity is *O(1)*. The *fetch-and-add* primitive guarantees the procedure of retrieving timestamp value is aways succeed without blocking (line 2 of Algorithm 5). The while-loop statement wrapping around *SWAP* primitive guarantees if concurrent read requests arrive at the system simultaneously, one of them will succeed and exchange its snapshot with the global one. In overall, these statements (lines 3 – 7 of Algorithm 5) are lock-free. The insert() operation (shown in Algorithm 7) is lock-free because a snapshot is inserted into a linked list at either the head or the body. For the lock-free linked list in ECCounts, in most cases, snapshot is inserted at head, which is performed by the *compare-and-swap* primitive (line 10 of Algorithm 7). For the corner case where the snapshot should be inserted into other positions, the function first searches the right position, and then inserts the snapshot into the list by using another *compare-and-swap* primitive in line 22. Overall, the insert() operation takes finite steps to complete. The sumup() (line 9 of Algorithm 4) operation is wait-free because there is no critical sections within this function call. In overall, function read() of ECCounts-MR is *lock-free*.

4.2 | Strong consistency guarantee

For strong consistency property of counting results, we mean that the counting result of a read request which first arrives at the system is less than or equal to that of any subsequent read requests. In other words, the counting results of read requests must monotonically increase. Specifically, in ECCounts, for each read request, the reader thread gets a monotonically increasing timestamp value t, reads values out of counters, and then sums them up and returns counting result, denoted as $result_t$. Strong consistency property guarantees that $result_{t1} <= result_{t2}$, if $t1 < t2$.

Obviously, this property holds in ECCounts-SR because read requests arrive at the system in series. For ECCounts-MR (Algorithm 5), all of the snapshots are inserted into the global lock-free linked list *global_old_snapshots* (line 8 in Algorithm 5), and then a read request gets the counting result by walking through the whole linked list and summing up snapshots whose timestamp values are less than or equal to the timestamp value of this read request (line 9 in Algorithm 5), which guarantees that the counting results are monotonically increase, given read requests have a monotonically increasing timestamp value. For example, read request $t1$ will sum up snapshots whose timestamp values are less than or equal to that of $t1$, and a subsequent read request $t2$ will sum up snapshots whose timestamp values are less than or equal to that of $t2$, which includes snapshot $t1$, given read request $t1$ arrives at the system before read request $t2$ and the timestamp value of read request $t1$ is less than that of read request $t2$. In overall, the counting results of ECCount is monotonically increasing, with strong consistency guarantee.

4.3 | Verification by Spin

To verify the correctness of ECCounts, especially when the algorithm is used in scenarios where preemption happens, we utilize Spin[19] to formally verify the correctness of the algorithm. Specifically, we first model ECCounts-MR shown in Section 3.2 as a concurrent system, by using modeling language Promela[19] and then verify the correctness (including safety, liveness, and data consistency) of the system by utilizing Spin. The concurrent system in Promela is equivalent to algorithm ECCounts-MR in control flow, except the following.

- ECCounts-MR can support a huge amount of concurrent updaters (eg, 1024, only if the maximum number is pre-defined), and unlimited concurrent readers. The concurrent system in Promela, in contrast, has two concurrent updaters and two concurrent readers. We choice the configuration of two concurrent updaters and two concurrent readers because it is sufficient to unveil any concurrency issues, given Spin performs a full state space search.[19] Adding more concurrent updaters or readers merely results in a state space explosion.
- ECCounts-MR utilizes a lock-free linked list to manage old snapshots. The concurrent system in Promela, in contrast, uses an sequential array-based list to simplify the system. We believe a sequential array-based list is sufficient in verifying the correctness of ECCount because ECCount is independent to the specific list used, and a list algorithm can be verified independently.

The Promela code has been open-sourced, and can be found on the website of ECCounts. The formal verification shows that ECCount is correct not only in safety and liveness, but also in the strong consistency guarantee. It is worth noting that without any optimizations, a full state space verification can run out of memory, even on a server with 264GB of physical memory. To perform the verification, optimizations such as Bit State Space Analyses should be used.

5 | EVALUATION

This section measures ECCount and compares it against two widely used counters, *lock-based counter* and *statistical counter*. In this section, ECCount refers to ECCounts-MR. Section 5.1 details experiment setup. Section 5.2 demonstrates the miscount issue in *statistical counter*. After that, we evaluate the performance of ECCount in Section 5.3.

5.1 | Experiment setup

We conduct our evaluation on a Dell R730 server. The server is equipped with two Intel Xeon E5-2609 processors, each of which consists of 8 CPU cores running at maximum speed of 1.7GHz. Each CPU core has one 32KB L1 data cache, one 32KB L1 instruction cache and a 256KB L2 cache. All of the 8 cores on the same die share one L3 cache of 20MB. The two processors are connected through two QPI links of 6.4GT/s. The server uses 128GB DDR4 memory.

The server runs an Ubuntu 16.04 system with 64-bit Linux kernel version 4.4.0. ECCount is compiled by GCC 5.4.0 with -O3 option. In each run, the updater threads increment the counter as fast as possible, and reader threads read the value out of the counter. Each thread is bound to a dedicated CPU core. Each run lasts 60 seconds. Linux Perf is used to collect hardware and Linux OS performance counters. Unless otherwise stated, each data point is obtained from the average result over 10 trials. The source code of test programs used in Section 5.2 and 5.3 can be found at the homepage of ECCount project.

5.2 | Miscount in statistical counter

To explore the miscount issue in *statistical counter*, we build a testbed which simulates the network packets receiving procedure of Intel NICs (shown in Figure 1). Specifically, each test creates an array of counters, each of which is updated exclusively by an updater thread. For each read request, the reader thread (1) gets a monotonically increasing timestamp value t, which represents the beginning of read operation, (2) reads values out of counters, (3) sums them up, and then (4) returns counting result, which is denoted as $result_t$. Miscount happens if $result_{t1} > result_{t2}$, given $t1 < t2$.

We explore the miscount issue by varying both the number of reader threads and updater threads. In Table 1, a test is denoted as *Test-X-Y* if the number of concurrent reader and updater threads in the system are X and Y, respectively. Column *# Read* shows the number of read requests complete, column *# Miscount* shows the number of miscounts, and column *Mis. Rate* lists the ratio of *# Miscount* to *# Read*. Subsequent columns list the maximum and mean value of miscounts. The minimum values of miscounts of all of the tests are 1's. So, we omit this column. Each test runs 3 seconds, and on average the time period between two successive read requests is 3 microseconds.

Table 1 shows that, for the use case where there are 16 concurrent updater threads, the miscount rate can be as high as 6.51%, even if there are only two concurrent readers (first row). As the number of readers increases from 2 to 16, the miscount rate increases to 35.73%, meaning one third of returned results are miscounted. Similarly, for the use cases where there are 2 concurrent reader threads, as the number of updater threads increases from 1 to 16, the miscount rate heads to 10.56%.

TABLE 1 Miscount in *statistical counter*

	# Read	# Miscount	Mis. Rate	Max	Mean
Test-2-16	905,110	58,890	**6.51%**	14,841,858	5,008
Test-4-16	1,531,290	248,855	**16.25%**	16,961,241	3,842
Test-8-16	2,167,900	579,915	**26.75%**	12,900,396	1,666
Test-16-16	2,367,050	845,863	**35.73%**	7,630,882	1,330
Test-2-1	3,633,680	78,628	**2.16%**	966	10
Test-2-2	2,938,250	71,106	**2.42%**	1,751,302	494
Test-2-4	1,669,120	72,069	**4.31%**	1,318,216	35
Test-2-8	843,780	71,842	**8.51%**	6,634,030	266
Test-2-16	259,890	27,445	**10.56%**	3,940,867	517

ECCounts, however, does not suffer the miscount issue of *statistical counter*, meaning that for two read requests with timestamp values $t1$ and $t2$, ECCount guarantees that $result_{t1} <= result_{t2}$, if $t1 < t2$. Besides, ECCount is competitive to the best of existing mechanism, *statistical counter*, in performance.

5.3 | Performance of ECCount

To measure the performance of all of the three counting algorithms, we run the test on scenario where there are 1 reader and 16 concurrent updaters. Figure 5 shows the performance of updaters, as the number of concurrent updaters increases. It is worth noting that the line of *statistical counter* looks similar to that of ECCount because the updater threads of these two mechanisms perform similarly in this test.

Figure 5 shows that as the number of updaters increases, the cost of update operations of *statistical counter* and ECCount are relatively stable, ranging from 9 to 12 nanoseconds. In contrast, the updater performance of lock-based mechanism decreases sharply, and the cost of an update operation go as high as about 500ns.

Similarly, Figure 6 shows that the cost of a read operation of *lock-based counter* is about 5ns, and the algorithm scales as the number of updater threads increases. This is because the lock is used only to serialize updater threads, and the reader thread can access the counter without requiring the lock. The cost of read operations of *statistical counter* increases to about 1,500ns when the number of updater threads increases to 15. As discussed in Section 2, the major reason of the performance degradation is cache miss; Each counter is very likely to incur one cache miss, which explains why in Figure 5 the cost of reader of *statistical counter* increases nearly linearly to the number of updater threads. The cost of read operation of ECCount also increases to about 1,500ns when the number of updater threads increases to 15. However, the major reason is that ECCount has to maintain the snapshot linked list (Section 3), and accessing a linked list where insertion and deletion happen frequently typically implies cache misses. Figure 6 shows that the reader performance of ECCount is competitive to that of *statistical counter*, the existing best counting algorithm for high-speed network devices.

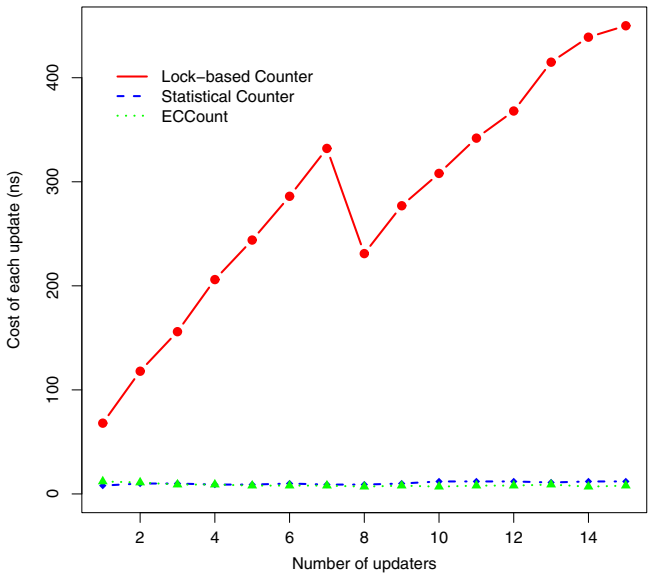

FIGURE 5 Performance of function update() as the number of updaters increases

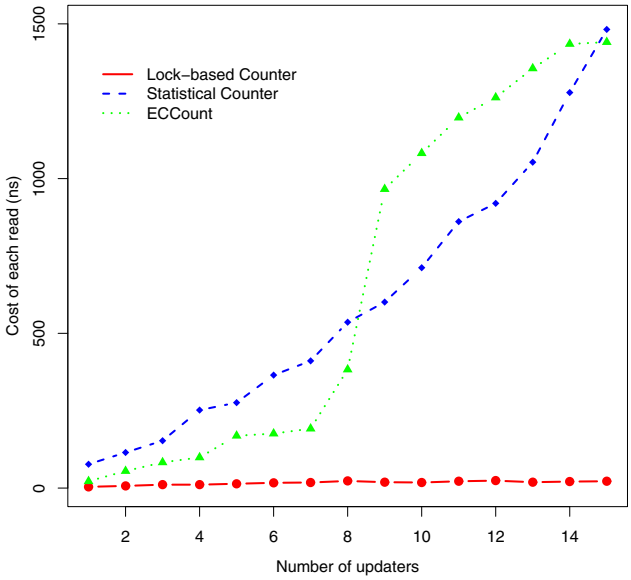

FIGURE 6 Performance of function read() as the number of updaters increases

TABLE 2 Performance of ECCount update operation as the number of readers increases

# readers	1	2	4	8	12	15
Updater cost(ns)	4	4	5	6	6	8

Table 2 shows the performance of ECCount update operation as the number of updater threads increases. Column *N* lists the result of test where *N* reader threads run simultaneously. Row *Updater cost* presents the cost of each update operation in nanosecond. Table 2 shows that the cost of ECCount updater is immutable to the number of readers, which demonstrate the usability of ECCount in real systems.

6 | RELATED WORK

A variety of counting techniques have been invented in the past decades to utilize the increasing parallelism from hardware. The most straightforward approach is by adding locks to protect the critical section: counters and related update operations. Variations of locking techniques,[1,2,20] even though work well on small-scale servers, degrade severely as the number of CPU cores increase[21] because locks not only force the access to counters to be sequential but also stresses underneath hardware bus due to the "hot-spot" in cachelines containing the lock variable.

To remove the contention on locks and scale on massively parallel systems, message-based counter technique[3,4] allocates an extra thread running on a unique processor as an agent. The thread receives update requests from worker threads, and access counters on behalf of them. Similarly, queue-based counter techniques[22] utilize a queue data structure to organize requests from worker threads. The thread at the head of the queue owns the lock, and when its job is done, the queue is passed to its successor. These techniques are relatively immune to contention, but nevertheless scale poorly because either the agent thread or the queue becomes a new bottleneck.

To meet the performance requirements of use cases such as network applications, *statistical counter* utilizing weaken consistency modules (eventual consistency and quiescent consistency) have been proposed.[5,10] Weak consistency module provides more parallelism in counting. The drawback of weak consistency, however, is that miscount happens if updaters make any progress while the reader is reading the counter. ECCount is built on top of *statistical counter*. However, ECCount provides accurate counting results, which is quite useful in real applications.

7 | CONCLUSION

This paper explores an efficient and consistent counting algorithm which not only scales on multi-core architectures but also provide accurate counting results without miscounts. We present novel techniques to enable concurrent reader threads in this algorithm. Experiments show the cost of reader could be a bottleneck. We will continue to optimize the algorithm.

ACKNOWLEDGMENTS

We thank the reviewers for their insight comments and suggestions which help improve this paper. This work was supported in part by National Natural Science Foundation of China under Grant No. 61602264, China Postdoctoral Science Foundation under Grant No. 2017M611882.

ORCID


Junchang Wang https://orcid.org/0000-0002-3465-1982

REFERENCES

1. Agarwal A, Cherian M. Adaptive backoff synchronization techniques. In: Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA'89); 1989; Jerusalem, Israel.
2. Graunke G, Thakkar S. Synchronization algorithms for shared-memory multiprocessors. *Computer*. 1990;23(6):60-69.
3. Kubiatowicz J, Agarwal A. Anatomy of a message in the Alewife multiprocessor. In: Proceedings of the 7th International Conference on Supercomputing (ICS'93); 1993; Tokyo, Japan.
4. Petrović D, Ropars T, Schiper A. Leveraging hardware message passing for efficient thread synchronization. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'14); 2014; Orlando, FL.
5. McKenney PE. Is parallel programming hard, and, if so, what can you do about it? Linux Technology Center, IBM Beaverton; 2011. arXiv:1701.00854.
6. Boyd-Wickizer S, Kaashoek MF, Morris R, Zeldovich N. Oplog: a library for scaling update-heavy data structures. 2014.
7. Morris R. Counting large numbers of events in small registers. *Commun ACM*. 1978;21:840-842.
8. Fuchs M, Lee CK, Prodinger H. Approximate counting via the Poisson-Laplace-Mellin method. Paper presented at: 23rd International Meeting on Probabilistic, Combinatorial, and Asymptotic Methods in the Analysis of Algorithms; 2011; Montreal, Canada.
9. Tsidon E, Hanniel I, Keslassy I. Estimators also need shared values to grow together. In: 2012 Proceedings IEEE INFOCOM; 2012; Orlando, FL.
10. Herlihy M, Shavit N. *The Art of Multiprocessor Programming*. Burlington, MA: Morgan Kaufmann; 2011.
11. Vogels W. Eventually consistent. *Commun ACM*. 2009;52(1):40-44.
12. Bailis P, Ghodsi A. Eventual consistency today: limitations, extensions, and beyond. *Commun ACM*. 2013;56(5):55-63.
13. Intel 40gbps network interface card. http://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-xl710-brief.html
14. Introduction to Message-Signaled Interrupts. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-message-signaled-interrupts
15. Harris TL. A pragmatic implementation of non-blocking linked-lists. Paper presented at: International Symposium on Distributed Computing (DISC); 2001; Lisboa, Portugal.
16. Zhang K, Zhao Y, Yang Y, Liu Y, Spear MF. Practical non-blocking unordered lists. Paper presented at: International Symposium on Distributed Computing (DISC); 2013; Jerusalem, Israel.
17. McKenney PE, Slingwine JD. Read-copy update: using execution history to solve concurrency problems. In: Parallel and Distributed Computing and Systems; 1998; Cambridge, MA.
18. Memory Barrier in Linux Programming. https://www.kernel.org/doc/Documentation/memory-barriers.txt
19. Promela/spin: Software Formal Verification. http://spinroot.com/spin/whatispin.html
20. Anderson TE. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Trans Parallel Distributed Syst*. 1990;1(1):6-16.
21. Herlihy M, Lim BH, Shavit N. Scalable concurrent counting. *ACM Trans Comput Syst*. 1995;13:343-364.
22. Mellor-Crummey JM, Scott ML. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans Comput Syst*. 1991;9(1):21-65.

How to cite this article: Wang J, Li T, Fu X. Accurate counting algorithm for high-speed parallel applications. *Concurrency Computat Pract Exper*. 2019;31:e5090. https://doi.org/10.1002/cpe.5090