Resettable Encoded Vector Clock for Causality Analysis With an Application to Dynamic Race Detection

Tommaso Pozzetti and Ajay D. Kshemkalyani, *Senior Member, IEEE*

Abstract—Causality tracking among events is a fundamental challenge in distributed environments. Much previous work on this subject has focused on designing an efficient and scalable protocol to represent logical time. Several implementations of logical clocks have been proposed, most recently the Encoded Vector Clock (EVC), a protocol to encode Vector Clocks (VC) in scalar numbers through the use of prime numbers, to improve performance and scalability. We propose and formalize the concept of **Resettable Encoded Vector Clock** (REVC), a new logical clock implementation, which builds on the EVC to tackle its very high growth rate issue. We show how our REVC can be applied in both shared memory systems and message passing systems to achieve a consistent logical clock. We show, through practical examples, the advantage of REVC's growth rate with respect to EVC's growth rate. Finally, we show a practical application of the REVC to the dynamic race detection problem in multi-threaded environments. We compare our tool to the currently existing VC-based tool DJIT$^+$ to show how the REVC can help in achieving higher performance with respect to the Vector Clock.

Index Terms—Causality, vector clock, prime numbers, encoding, dynamic race detection, clock reset protocol, performance

1 Introduction

A FUNDAMENTAL concept in distributed systems is that of tracking causality among events that occur at different processes in the system [1], [2]. The causality relation between events is given by Lamport's *happens-before* operator [3] (represented as $\rightarrow$). On a single-threaded process, causality can be tracked by means of attaching a timestamp to each event, thus establishing a precedence relation among them. However, this problem becomes much harder to solve in a multi-threaded or multi-processor environment, as concurrency among threads and processes depends on their relative speed and can therefore change at any different execution. Consequently, a timestamp from a global clock is not sufficient to correctly track causality relations among events. Furthermore, in a multi-processor system, global time is not even available unless processors' clocks are tightly synchronized (which is not achievable in real systems). Therefore, a different notion of time and causality becomes necessary for such environments. One solution to the problem is to substitute the physical time with logical time [2], given by a logical clock at each thread or process.

Several implementations of logical clocks have been proposed, such as *Scalar Clock* [3], *Vector Clock* (VC) [4], [5], and *Encoded Vector Clock* (EVC) [6], [7]. The scalar clock has the drawback that causality between events cannot be inferred from the timestamps of events. The vector clock allows such inference, but it requires each process to maintain a vector of size equal to the number of processes, and this has been shown to be a lower bound [8]. The EVC is a technique to represent a Vector Clock using a single scalar number, to improve scalability of the VC and it is based on the use of prime numbers to achieve such encoding. The main drawback of the EVC has been shown to be the extremely high growth rate which quickly causes an overflow at the locations that are storing the EVC values [9].

In this paper, we propose and formalize the concept of *Resettable Encoded Vector Clock* (REVC), a new logical clock implementation, which builds on the EVC to reduce its growth rate and, under given conditions, place an upper bound on its storage requirements, while maintaining the desirable properties of the EVC. The basic idea of the REVC is that a process can unilaterally and asynchronously reset its EVC when the EVC value overflows, i.e., reaches a predetermined fixed number of bits in size. We give the basic operations of the REVC (*tick*, *merge*, and *compare*) and then we show how our REVC can be applied in both shared memory systems and message passing systems to achieve a consistent logical clock. We prove that the REVC's growth rate is linear in the number of events executed, as opposed to the exponential growth rate of the EVC. We show, through practical examples, the advantage of REVC's growth rate with respect to EVC's growth rate. We then give a bounded growth implementation of the REVC, called Fixed Size Frame Window (FSFW), that is applicable whenever a contract between the application and the system is satisfied. We then give the Differential Merge Technique (DMT) optimization that further reduces the cost of the merge operation of the REVC.

We then show a practical application of the REVC to the dynamic race detection problem in multi-threaded

• *The authors are with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607 USA. E-mail: pozzetti.tommaso@gmail.com, ajay@uic.edu.*

Manuscript received 6 Nov. 2019; revised 6 Aug. 2020; accepted 13 Oct. 2020. Date of publication 20 Oct. 2020; date of current version 12 Nov. 2020. (Corresponding author: Ajay D. Kshemkalyani.) Recommended for acceptance by S. K Prasad. Digital Object Identifier no. 10.1109/TPDS.2020.3032293

environments using the RoadRunner [10] dynamic analysis framework. We compare our tool to the currently existing VC-based tool DJIT$^+$ [11] to show how the REVC can help in achieving higher performance with respect to the VC. For the range of applications tested, our REVC approach achieves an average speedup of 1.6 over the DJIT$^+$ tool.

Our results show how the REVC is not just a theoretical concept, but it is applicable to practical problems and can compete in terms of both space and time requirements with other known protocols. The REVC has been designed with scalability and adaptability in mind. Its formulation contains several intrinsic tradeoffs that can be easily tuned by enabling or disabling optimization techniques, and choosing between bounded and unbounded implementations. These configurations provide the REVC with a much higher adaptability to very different scenarios, which cannot be found in other logical clock implementations.

In Section 2 we detail previous and related work that has been carried out on logical clocks and on dynamic race detection. In Section 3 we present the models for the shared memory systems and message passing systems on which we apply our Resettable Encoded Vector Clock. In Section 4 we detail the components and rules for the REVC. In Section 5 we give an analysis of the growth rate of REVC along with its evaluation with respect to the EVC. We also propose the FSFW bounded growth rate framework and the DMT optimization. Section 6 shows a practical application of the REVC to the dynamic race detection problem and the performance results. Finally, in Section 7, we conclude our work.

2 Related Work

Several logical clock implementations have been proposed in the literature such as Scalar Clock [3], Vector Clock [4], [5] and the EVC [6], [7] on which our protocol is based. The first is a lightweight, simple and efficient system to track causality through the use of scalars, but it does not allow to obtain the causality relation from the timestamps (no *strong consistency*). The Vector Clock achieves the property of strong consistency, at the expense of scalability and performance. It is based on keeping at each process an array of size equal to the number of processes in the system, and in the general case, this is a lower bound [8]. The ith component of such array at process x denotes the value of the scalar clock of process i as known by process x at that time. Several works in the literature attempted to reduce the size of the Vector Clock [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], but they had to make some compromises in accuracy, and/or had limited applicability, and/or had to alter the system model, and in the worst-case, were as lengthy as the Vector Clock. We review these next.

A mechanism to reduce the size of the vector timestamp piggybacked on messages sent over FIFO channels was given in [12]. The local storage for vector clocks is not decreased. For applications where it is important to determine the causality between two messages sent to a common destination, a mechanism to reduce the size of the vector timestamp was given [13]. This mechanism cannot capture causality between all possible pairs of events. Plausible clocks are approximation clocks that use fewer entries in their vector than vector clocks [14]. Plausible clocks show false positives (but no false negatives) and thus compromise accuracy. The scheme in [15], [16] requires processes to be organized as a hierarchy of clusters. Two types of timestamps are maintained–short internal timestamps for (send) events that occur from within the same cluster and longer external timestamps for send events that occur from outside the cluster. A receive event in a cluster corresponding to a send event outside the cluster is termed a cluster-receive event. This scheme allows a trade-off between the size of the cluster, the number of cluster-receives, and the size of the cluster-receive timestamp. This scheme has limited applicability.

In data stores, dotted version vectors [17] improve on version vectors (which are based on vector clocks) by using size proportional to the number of servers rather than number of clients. This idea is orthogonal to our work and is applicable to data storage systems. In dynamic settings where the number of processes varies with time, interval tree clocks [18] allow the size of the vector clock to also dynamically adjust to the number of processes. This is an orthogonal idea to our work.

Logical physical clocks combine physical clocks (such as NTP) with logical clocks [19], [20]. This scheme deals only with weak causality, i.e., $a \rightarrow b \implies Clock_a < Clock_b$, and is meant for applications where only weak causal relations in the immediate past are relevant. In [21], a scheme to reduce the size of vector timestamps is given; this scheme is applicable when not all processes communicate to each other and when the assignment of a timestamp to an event is deferred for a suitably determined period of time. In multithreaded systems, the size of the vector clock is either the number of threads or the number of objects. It was shown that when the thread-object bipartite graph of an execution has low density, a vector clock of size less than the number of threads and less than the number of objects can be used [22]. Bloom clocks which are based on Bloom filters reduce the size of vector clocks but show false positives, and thus compromise accuracy [23], [24].

The EVC is an encoding of the VC using prime numbers, which results in the use of one single scalar number, thus allowing for higher scalability while maintaining strong consistency [6], [7], [25]. The EVC, initialized to 1 at each process, is characterized by three main operations.

1) the first is the *local tick* (denoted *T*) and is used to increment the value of the clock. This is achieved by multiplying the current value by the current process' unique prime number.
2) The second operation is used to *merge* (denoted *M*) two clock values and requires to perform the *least common multiple* operation (LCM) between the two values.
3) Finally, the *comparison* operation (denoted *C*) allows to establish the causal relation between events *a* and *b* as given by Lamport's *happens-before* operator [3] (represented as $\rightarrow$). Such operation can be described as $[a \rightarrow b \Leftrightarrow EVC_a < EVC_b \wedge EVC_b \bmod EVC_a == 0]$.

The main drawback of the EVC has been shown to be the extremely high growth rate which quickly causes an overflow at the locations that are storing the EVC values [9]. By relying on multiplications and *least common multiple* operations, EVC

values grow at an exponential rate, which can quickly require a very large number of bits in order to be able to represent it, and therefore store it. Despite this, the EVC has found a practical application to detect memory consistency errors in MPI one-sided applications [26], [27], due to the use of resets at global barrier synchronizations. The EVC was also used to detect weak conjunctive predicates in distributed systems [25]. The idea of resetting the EVC when it overflows was suggested in [6], [7], [25] but the details were not given. Other techniques to slow the growth of the EVC suggested in [6], [7], [25] include: ticking only at relevant events, the use of detection regions, and the use of logarithms of EVCs.

To address the problem of overflow in vector clocks, two approaches have been explored. One approach to such reset is to require that, when the first process overflows the available number of bits for storing the VC, the system is *paused* and all processes synchronize to execute a local reset of their VC. A practical way to achieve such result is to use a modified version of Lamport's global snapshot algorithm [28], which requires processes to exchange control messages to achieve a global synchronization point. Such technique has been studied and implemented as a protocol for resetting Vector Clocks [29]. However, we note that such a technique not only introduces a high overhead into the system caused by the added synchronization, but would also break equivalency of resettable VC and VC. We can in fact show that, if a synchronization event c is added to the system to perform a global reset, the underlying causal order is changed. Let us take two concurrent events a and b and let us suppose that after executing a but before executing b, an overflow occurs and therefore a reset is required. Event c is now executed, as the global synchronization event in which all processes perform the local reset. Given that c is a global synchronization event executed by all processes, the intrinsic program ordering states that $a \rightarrow c$. After the reset, the system can restore normal operations and event b is now executed. Again, given that c is a global synchronization event, it follows that $c \rightarrow b$. But because of the transitive property of Lamport's happened-before relation, if $a \rightarrow c$ and $c \rightarrow b$, we can state that $a \rightarrow b$ which is in contrast with the result that would be obtained by the VC, in which a and b are two concurrent events.

The second approach to address overflow of vector clocks exploits asynchronous local resets at each process [30]. The authors observed that many applications are structured in phases and track causality only within a bounded number of adjacent phases [31], [32], [33], [34]. So they reused timestamps by augmenting VC with a nonblocking reset operation that allows a process to locally reset its own local clock when it moves from one phase to another. Thus, a bounded number of clock values suffice provided no two timestamps of different incarnations exist. Applications that use this bounded implementation of resettable VC need to satisfy a contract having two parts: (i) a comparison predicate over events whose causality needs to be tracked, and (ii) a communication pattern between the processes. The bounded space version of our solution of the REVC is related to this latter approach and uses only a simple comparison predicate in the contract.

Dynamic race detection in multi-threaded environments has been a topic of interest for multiple works. Our protocol for dynamic race detection that adopts our REVC is a modification of DJIT$^+$ [11] which is a dynamic race detection protocol that exploits vector clocks to track memory accesses. The state-of-the-art dynamic race detection protocol is FastTrack [35], which is an optimization over DJIT$^+$ that is able to exploit primarily a Scalar Clock representation, switching to Vector Clock only in the few instances in which the information stored by the Scalar Clock is not enough.

RoadRunner [10] is the dynamic analysis framework that was used to develop FastTrack and that we used to implement our modified version of DJIT$^+$ exploiting the REVC. It is made up by a Java bytecode instrumenter which modifies Java code to create a stream of events during program execution, which is consumed by a backend tool implementing the dynamic analysis protocol.

3 System Model

Distributed systems are usually divided into two categories: *shared memory* systems and *message passing* systems [2]. As the names suggest, this division is based on the type of communication that the different components of the distributed system use to synchronize.

A shared memory system is composed of several threads that execute concurrently and communicate via shared memory. The execution is assumed to be asynchronous, therefore the relative speed among the threads is not fixed. The threads can synchronize using the operating system provided operations *lock*, *unlock*, *fork* and *join*. Other more complicated means of synchronization are not analyzed here, but the model can be easily extended to include them by building on these basic primitives. The execution of each thread consists in a series of events that can be either one of the operations previously presented or what here is called an *internal event*. Internal events can be any type of action that is performed by a thread, which does not require any synchronization with other threads, and can therefore be seen as an event local to the thread itself. A partial order on this set of actions can be induced by Lamport's *happened-before* relation [3], which can characterize causality among such events. In this context, such relation is characterized by the following rules:

- *Program order*: if a and b are two events that are executed by the same thread, and a is executed before b, then $a \rightarrow b$
- *Lock synchronization*: if u is an *unlock* operation and l is a *lock* operation on the same lock object, then $u \rightarrow l$
- *Thread creation*: if a is a *fork* operation and b is an event executed by the thread that is created by a, then $a \rightarrow b$
- *Thread termination*: if j is a *join* operation and a is an event executed by the thread that will terminate at j, then $a \rightarrow j$
- *Transitive property*: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

A message passing system, on the other hand, is composed of several processes that execute concurrently and communicate via message passing. We assume the presence of a logical communication channel among each pair of processes, although we do not require that such channels be physically present in the system. The execution is again assumed to be asynchronous, therefore no fixed relative

speed among the processes exists. Each process can synchronize with the others by means of the communication primitives *send* and *receive*. The *fence* operation is also available, which allows to create barrier synchronizations among multiple processes. The execution can again be characterized as a sequence of events. In message passing systems, four types of events can be distinguished: *send* events, *receive* events, *fence* events and *internal* events. The first three correspond to the execution of the basic communication that have been just described, while an internal event is defined equivalently as in a shared memory environment. Lamport's *happened-before* relation [3] can be used to induce a partial order on this set of events to characterize causality among them. In message passing systems the following rules apply:

- *Program order*: if a and b are two events that are executed by the same process, and a is executed before b, then $a \rightarrow b$
- *Message synchronization*: if s is a *send* operation for message and r is the *receive* operation for the same message, then $s \rightarrow r$
- *Barrier synchronization*: if b_i and b_j are two corresponding *fence* operations executed by two processes and a is an event executed by any of the two processes before reaching the barrier, then $a \rightarrow b_i \wedge a \rightarrow b_j$. Also if a' is an event executed by any of the two processes after executing the *fence* operation, then $b_i \rightarrow a' \wedge b_j \rightarrow a'$
- *Transitive property*: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

If $\neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$ then a and b are said to be *concurrent events*. It can be intuitively understood that if $a \rightarrow b$ then in any execution of the program, a will always be executed before b as b is causally dependent on a. On the other hand, if a and b are concurrent, they can be executed in any order on a given execution of the program.

4 The Resettable Encoded Vector Clock

The Resettable Encoded Vector Clock's core idea is that of performing a reset operation at the EVC location every time such value overflows a predefined number of bits n. The reset operation brings the EVC value back to the initial one, allowing the system to restart its operations until the following overflow event.

4.1 Components of the REVC

To maintain the consistency of the logical clock through the use of the reset operation, new components need to be added.

- The first additional component we introduce is the concept of *frame*. The *frame* is a counter that keeps track of the number of local resets that have been performed at each process. Each REVC instance contains an integer variable, which defines its current *frame*. By using the *frame*, a more precise comparison between two REVC timestamps is possible. In fact, it allows to understand whether one of the two is directly comparable with the other through the standard EVC comparison operation, by checking whether they are part of the same execution frame.
- The second component that we introduce is the *frame history*. Each REVC instance needs to be comparable with any other REVC instance, irrespective of whether the other instance is in the same execution frame or not. In order to achieve that, each REVC instance needs to keep track of the EVC values of previous frames before the local resets. The *frame history* component, therefore, achieves this objective by storing a structure that can map a given frame to a given EVC value.

Based on the components that we have detailed, we can represent an instance of the Resettable Encoded Vector Clock as the tuple (f, e, m) where f is the current frame, e is the current EVC value and m the map containing the frame history. Finally, given that the REVC is built on the Encoded Vector Clock, each process needs to be assigned a unique prime number that is used by the EVC to encode the timestamp of the process itself.

We have detailed in Section 2 the three basic operations characterizing the EVC: the *local tick* operation T, the *merge* operation M and the *comparison* operation C. In the following, we analyze and detail the three corresponding operations for the Resettable Encoded Vector Clock.

4.1.1 Local Tick Operation

The *local tick* operation is used to increment the local clock of a process to create a new fresh timestamp value that can be assigned to label any new event. This operation, performed on the standard EVC, can cause the current value to overflow the given n bits threshold. In the REVC, the overflow event constitutes the trigger that requires a local reset and a transition to the following *frame*, saving the old EVC value in the history map. If T is the operation that performs the local tick on an EVC, Algorithm 1 shows the pseudocode of the corresponding operation for the REVC, while Algorithm 2 shows the pseudocode for the reset function.

Algorithm 1. Local Tick Operation

Input: An REVC instance (f, e, m) and a prime number p
Output: The updated REVC instance

```
1: temp = T(e);
2: if overflow (temp) then
3:    (f, e, m) = reset (f, e, m, p);
4: else
5:    e = temp;
6: end
7: return (f, e, m);
```

Algorithm 2. `reset` Function

Input: An REVC instance (f, e, m) and a number q
Output: The reset REVC instance

```
1: m.put(f, e);
2: f = f + 1;
3: e = q;
4: return (f, e, m);
```

4.1.2 Merge Operation

The *merge* operation is used to generate a new clock value that acknowledges two previous values. Just like the local tick operation, this operation performed on two EVC values

can lead to overflow. Furthermore, new complications arise when performing such operation on a REVC. Clearly, it only makes sense to perform the EVC merge operation between two EVCs that belong to the same execution frame. Furthermore, given that the REVC also contains a history of previous frames, it is not sufficient to merge the current frame's EVCs, but it is also necessary to merge the corresponding previous EVCs. Therefore, different scenarios can arise depending on whether the two instances are on the same execution frame.

It should be noted that, while merging the history maps' EVCs, overflow could occur again. However, we assume that the data structures for the history maps allow for a sufficiently high number of bits such that these overflows can never happen. Let there be k processes in the system. Given a n bit threshold for overflow of an EVC, it is in fact possible to place an upper bound on the maximum number of bits required to store any possible result of all merge operations for any frame of a history map as $n * k$.

Supposing that (f_2, e_2, m_2) is the REVC that needs to be merged into (f_1, e_1, m_1) and that M represents the merge operation for EVCs, Algorithm 3 shows the pseudocode of the corresponding operation for REVCs while Algorithm 4 shows the pseudocode of the function that performs the merge of the two history maps.

Algorithm 3. Merge Operation

Input: Two REVC instances (f_1, e_1, m_1) and (f_2, e_2, m_2)
Output: The updated REVC instance

if $f_1 > f_2$ **then**
  m_1.put(f_2, M(m_1.get(f_2), e_2));
  m_1 = historyMerge(m_1, m_2);
else if $f_2 > f_1$ **then**
  m_1.put(f_1, e_1);
  $f_1 = f_2$;
  $e_1 = e_2$;
  m_1 = historyMerge(m_1, m_2);
else
  $temp$ = M(e_1, e_2);
  if overflow($temp$) **then**
    (f_1, e_1, m_1) = reset(f_1, $temp$, m_1, 1);
  else
    $e_1 = temp$;
  end
  m_1 = historyMerge(m_1, m_2);
end
return (f_1, e_1, m_1);

Algorithm 4. historyMerge Function

Input: Two history maps m_1 and m_2
Output: The merged history map

foreach (f, e) **in** m_2 **do**
  if m_1.contains(f) **then**
    m_1.put(f, M(m_1.get(f), e));
  else
    m_1.put(f, e);
  end
end
return m_1;

4.1.3 Comparison Operation

Given two timestamped events, the main purpose of a logical clock is to be able to determine their relationship according to the Lamport's *happened-before* relation. This is the aim of the comparison operation. Given any two events a_1 and a_2, with their respective REVC timestamps (f_1, e_1, m_1) and (f_2, e_2, m_2), we again distinguish different scenarios based on whether the two instances are on the same execution frame.

Supposing that C is the EVC comparison operation returning *true* if the event passed as first parameter happens before the event passed as second parameter or *false* otherwise, Algorithm 5 contains the pseudocode for the corresponding REVC operation that tests whether the relation $a_1 \rightarrow a_2$ is true or false.

Algorithm 5. Comparison Operation

Input: Two REVC instances (f_1, e_1, m_1) and (f_2, e_2, m_2)
Output: true if (f_1, e_1, m_1) $\rightarrow$ (f_2, e_2, m_2), **false** otherwise

if $f_1 > f_2$ **then**
  return false;
else if $f_2 > f_1$ **then**
  return (C(e_1, m_2.get(f_1)) **or** e_1 = m_2.get(f_1));
else
  return C(e_1, e_2);
end

4.2 REVC for Shared Memory Systems

In Section 3 we described the system model for a shared memory environment in which several threads run in parallel and synchronize by means of a set of operations. We now show how the three basic operations of the REVC can be applied to the system's events to obtain the causality relations that describe the system model.

The events that are present in the shared memory system have been shown to be either *internal events* or one of the following synchronization operations: *lock*, *unlock*, *fork*, *join*. In the following, we detail the behavior of the REVC for each of those operations, to ensure that the causality relation that is obtained is consistent with the one that has been described. If such behavior is implemented correctly, taken any two events *a* and *b* that have been labeled with timestamps, it will be possible to apply the *comparison* operation on their REVCs to establish their relationship.

The initialization of the system provides each thread with a unique prime number *p*, each REVC is initialized so that $f = 1$, $e = 1$ and *m* is an empty map, and each lock object is initialized so that $f = 1$, $e = 1$ and *m* is an empty map.

- *Internal events* happen locally at a single thread and require a fresh new timestamp that can uniquely identify them. When a thread executes an internal event, it (i) performs a *local tick* operation to update its current REVC with a fresh new value, and labels the event with the current value of its REVC.
- *Unlock events* are executed when a thread needs to release a given lock object that it has previously acquired. The lock-unlock pattern introduces a causal relationship among events executed at different threads before and after the synchronization event. In order to correctly represent such relations

using the REVC, the thread performing the unlock operation needs to (i) perform a *local tick* operation to update its current REVC with a fresh new value, and (ii) label the lock object with its current clock value.

- *Lock events* can happen whenever no thread is holding the lock of a specific lock object. In such case, any thread can perform a lock operation to acquire such lock. This event, and any local event that follows in the locking thread, must happen after the previous unlock event. This property can be realized by ensuring that the thread performing the lock operation (i) merges the REVC value, that has been stored on the lock object during the previous unlock operation, into its own current REVC instance, and (ii) performs a *local tick* operation to update its current REVC with a fresh new value.
- *Fork events* are executed when a new thread is started. The fork-join pattern, similarly to the lock-unlock pattern, introduces new causality relations. To acknowledge them, when the new thread is created, (i) the thread executing the *fork* performs a *local tick* operation to update its current REVC with a fresh new value. Furthermore, (ii) a new unique prime number p should be assigned to the new thread, (iii) the new thread's REVC's e variable should be initialized to 1, (iv) the frame of the REVC of the new thread should be set to the frame of the forking thread, and (v) finally, the forking thread should merge its REVC into the one of the new thread.
- *Join events* happen whenever a thread needs to wait for the termination of another thread before resuming its execution. When such operation terminates, the thread issuing the *join* should (i) set its own REVC as the result of performing a merge operation between its own REVC value and the terminating thread's REVC value, and (ii) perform a *local tick* operation to update its current REVC with a fresh new value.

For the dynamic race detection [11], [35] application considered in Section 6, the above rules are adapted as follows.

- Internal events: no *local tick*.
- Lock, and join events: no *local tick* if no reset is invoked by the merge invocation.
- Unlock and fork events: The *local tick* is executed as the last action (not the first).
- Initialization: The *vector time frame* (which is different from our frame f) initialization requires us to initialize e of the thread's REVC to p, both at startup and for a newly forked thread.

4.3 REVC for Message Passing Systems

In Section 3 we described the system model for a message passing environment in which several processes run concurrently and asynchronously and communicate by means of exchanging messages. We will now detail how the three basic operations that have been presented for a REVC can be applied to the events that can be executed in such environment, in order to implement the causality relations of this model.

Four types of events have been presented in relation to a message passing environment: *internal events*, *send events*, *receive events* and *fence events*. We notice that internal events follow the same rules in both environments, therefore in the following we will detail the behavior of the REVC for send, receive and fence events. The objective is that, by following the presented rules, the causality relation that is obtained through the application of the comparison operation of the REVC to two timestamped events is consistent with the one that has been described for this model.

The initialization of the system provides each process with a unique prime number p and each REVC instance is initialized so that $f = 1$, $e = 1$ and m is an empty map.

- *Send events* are executed to initiate a communication with another process. Similarly to the lock-unlock pattern for shared memory systems, the send-receive pattern introduces causality relations among the events that happen before and after such operations. In order to consistently represent such relations with the REVC, the sending process needs to (i) perform a *local tick* operation to update its current REVC with a fresh new value, and (ii) attach to the message its own current REVC timestamp.
- *Receive events* are executed when a process is expecting to receive a message from another process. To correctly acknowledge the causal chain that is created by such event, the receiving process (i) extracts the REVC timestamp that has been attached to the message and merges it into its own current REVC instance, and (ii) performs a *local tick* operation to update its current REVC with a fresh new value.
- *Fence events* are executed when a process needs to synchronize with several other processes at a barrier, to pause the execution until all participating processes have reached the same barrier. Since it is a synchronization operation, it introduces causality relations among the events happening before and after reaching the barrier at the various participating processes. To correctly represent such relations using the REVC, each participating process needs to (i) perform a *local tick* operation to update its current REVC with a fresh new value, and (ii) merge into its own REVC instance all the REVC timestamps of the other participants, thus aligning the clocks of all processes at the barrier to the same timestamp.

5 Analysis and Evaluation

5.1 Growth Rate

The REVC is built on the EVC and internally makes use of EVC values and operations to implement the logical clock. The EVC has been shown to have exponential growth, due to its use of multiplication and *least common multiple* operations [7], [9]. However, the REVC introduces the use of a reset operation that does not allow the EVC value to grow past a certain number of bits. The consequence of the application of the reset technique on the number of bits used by the EVC value is that such number fluctuates between a

minimum given by the number of bits required to represent the process' prime number, and a maximum of n bits, where n represents the threshold that has been chosen as the overflow value that triggers the reset operation.

We have however showed that keeping only the EVC value along with the reset operation is not enough to ensure a consistent logical clock implementation. Therefore, the REVC also stores a history map containing old EVC values for past execution frames. Every time a reset operation is performed, the old EVC value is added to the history map. The practical implication of this operation on the number of bits needed to represent an REVC instance is the following: when the fluctuation of the current EVC value reaches its peak, the reset brings it back to the minimum number of bits, moving the old value to the history, thus linearly incrementing the total number of bits that are needed. Therefore, intuitively, the growth rate of an REVC instance will present a linear behavior with respect to the total number of events being executed in the system. We now present this analysis more formally.

Theorem 1. *The size of the REVC is linearly proportional to the number of events in the system.*

Proof. At a *receive* event, on average, the EVC value of the current frame may double in bit size assuming a worst-case scenario that the LCM operation for the *merge* operation is performed over relatively prime numbers. And at a *tick* operation, the EVC value of the current frame may increase by $\log p$ bits.

We first consider a particular case where we focus on the execution at one process and assume that at a receive event when (f_2, e_2, m_2) is being merged into (f_1, e_1, m_1), $f_2 \leq f_1$. Let N denote the total size of the REVC including the history map size, let v denote the count of the number of (local) events after the last (local) reset, and let V denote the total count of the number of (local) events since the start. Let $g(v)$ be the function for the size of the current EVC value since the last (local) reset. When $g(v) = n$ or $v = g^{-1}(n)$, overflow and hence a reset occurs; and V becomes $V + v$ while N becomes $N + n$. After x (local) resets, $V = x \times v$ and $N = x \times n$. In the history merge, EVC values in older frames may increase in size, but in Section 4.1 we showed that the bit size of the EVC value is bounded by $n \times k$. Hence, N is proportional to $x \times n \times k = x \times g(v) \times k$. Here, the nature of the function g over the incremental δ-range v is not important, and can be replaced by a linear function of v. Hence N is proportional to the number of events $x * v = V$.

In the general case, multiple new frames may be added at a receive event in the history merge (when (f_2, e_2, m_2) is being merged into (f_1, e_1, m_1), f_2 may be greater than f_1), thus increasing N by the size of multiple frames. Hence x (the number of resets) needs to be considered along the longest (global) causal path in the execution up to the present event. So v (respectively, V) needs to be defined as the count of the number of events since the last reset along such a longest causal path (total count of events over such a longest causal path). Hence and as for the particular case, N will still be a linear function of $x * v = V$ as defined here. V is bounded by the number of events in the execution. □

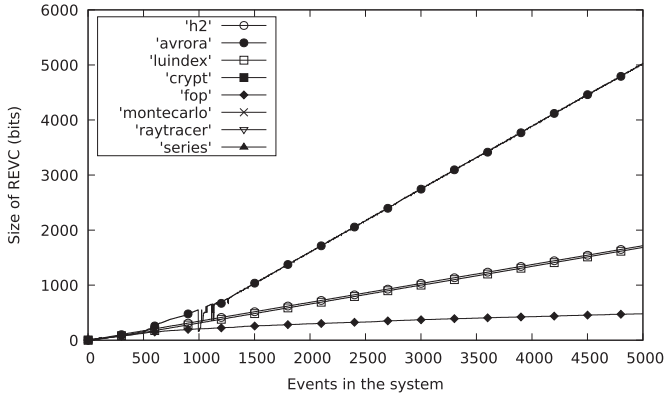

Fig. 1. Growth rate for the resettable encoded vector clock.

The experimental data that we collected confirm Theorem 1. The applications that were chosen for evaluation are a subset of the applications that are found in the DaCapo Benchmarking Suite [36] and the Java Grande Benchmarking Suite [37]. Those two suites are composed of Java programs that have been designed to emulate non-trivial loads and be representative of intensive calculations. The applications that have been chosen exhibit multi-threaded behaviour and very diverse semantics, in order to be able to test the system on a relatively complete set of programs ranging from a very high usage of synchronization to a high parallelization of the workload. In Section 6.2 we present these benchmark suites that we have used for evaluation of the dynamic race detection application. We have exploited this practical application of the REVC to track its growth in an actual real-world scenario rather than using a less realistic random simulation. Fig. 1 shows the results of the experiments performed using an REVC-based application run on several benchmark programs. The behavior of the growth rate is linear in the number of executed events as predicted, confirming how the use of a reset operation can consistently reduce the exponential explosion of the EVC values. As some applications require a small number of events and terminate, Fig. 2 shows the results zoomed in for the first 200 events.

The varying behavior of the different benchmark programs that have been used is clearly visible in the different lines. A subset of such applications, such as *avrora*, *fop*, *h2* and *luindex*, shows a high utilization of synchronization operations, therefore executing a high number of events, which surpasses our analysis window of 5000 system events. On the other hand, the other applications require a smaller number of events to be executed before the program is terminated. This can be linked to a lower utilization of synchronization operations, therefore higher parallelization of the workload, and it results in the lines terminating after less than 500 events have been executed in the system. The varying slopes of the lines can be related to the number of threads that are present in the system. A bigger number of threads is related to a greater slope, while a lower number of threads leads to a less steep line.

The linear growth rate achievement of the REVC is obtained at the expense of performance, introducing a trade-off between the two. In fact, the *merge* operation requires to

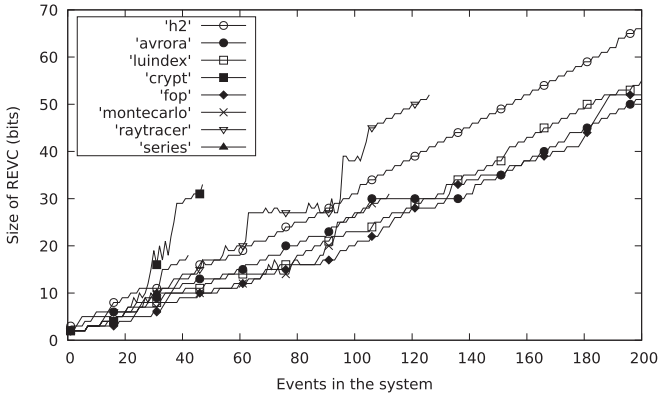

Fig. 2. Growth rate for the resettable encoded vector clock, zoomed-in results.

execute the EVC merge operation for f times, where f is the current number of frames, which can grow unboundedly, therefore its complexity is not constant. This is not acceptable in practical applications that require a high number of events to be executed, as the performance of the *merge* operations consistently degrades as the number of execution frames stored in the history map increases.

We note, however, that the time complexity of *merge* for REVC is lower than that of EVC. The LCM computation of *merge* is a function h that is super-linear in the number of bits of the operands, (using the logarithmic cost model) [6], [7]. For REVC, the time cost of the *merge* for f frames would be $O(f \times h(64))$. In EVC, the LCM computation of *merge* is done on big integers. As the operand sizes would be linearly proportional to f, a super-linear function h of this size would cost more time $O(h(64f))$ than the $O(f \times h(64))$ for REVC.

5.2 Bounded Growth

We have discussed how, despite achieving a reduction of the growth rate with respect to the EVC, the REVC still presents the issue of unbounded growth. It is immediate to see that the current EVC value will never overflow the given threshold of n bits, however the problem has been moved to the history map, as the structure can now grow unboundedly. Furthermore, this solution has also introduced a new tradeoff with performance, given by the new *merge* operation, whose time requirements are now dependent on the number of frames stored in the history map.

We note that the current formulation, which is not space and time bounded, is necessary because the general use case of the REVC requires to be able to perform the comparison operation between any two events of a given program execution. We observe, however, that this requirement can be safely relaxed in many real-world applications, where constraints can be added to reduce the required comparisons, without changing at all the semantics of the applications. We observe that many applications are structured as phases and track causality only within a bounded number of adjacent phases [31], [32], [33], [34]. Other examples are the following. In fair mutual exclusion which requires requests to be satisfied in their "happened before" timestamped order, processes compare their requests' timestamps only within a bounded number of requests of other processes [2]. In race detection, the racing instructions are executed in the temporal locality of one another. And in checkpointing, processes need to track causality only between a pair of consecutive checkpoints [2]. Many applications of logical clocks, in fact, require to perform causality analysis only on a subset of all events.

Thus, if this constraint to reduce the number of comparisons can be expressed as a maximum number of frames F in the REVC framework, the presented solution is immediately transformed into a bounded one both in time and in space. Each instance of an REVC (f, e, m), in fact, does not need to store any information about frames that are older than $f - F$. Therefore, an upper bound can be placed on the size of the m data structure, and the merge operations will only merge up to F EVC pairs in the worst-case scenario. We call this optimization technique *Fixed Size Frame Window (FSFW)*.

We can stipulate a contract with the application that exploits the REVC, under which the FSFW REVC has an equivalent behavior to the Unbounded REVC. This contract can be formalized and expressed as a predicate that establishes whether two timestamped events, a_1 and a_2, with respective timestamps (f_1, e_1, m_1) and (f_2, e_2, m_2), can be compared under the FSFW REVC, given a maximum amount of stored frames F

$$a_1 \; comparable \; a_2 \Leftrightarrow |f_1 - f_2| \leq F. \tag{1}$$

Given that F can be arbitrarily chosen, based on the performance requirements of the REVC implementation, this constraint is likely easily satisfiable by most real-world applications.

Note that unlike the resettable VC [30] which uses a two-part contract – a comparison predicate and a communication pattern requirement – our design of the REVC based on which the bounded space implementation is designed uses the above simple comparison predicate based on the frame count difference F.

The reason that the authors of [30] present an additional communication pattern contract is that they make the domain of the phase (their equivalent to our frame) bounded and equal to F. (As an example, if $F = 3$, then the phase will be 1, 2 or 3 and then will reset to 1.) Because of this, messages need to be received within the F window or their timestamp will collide with an equal timestamp of a previous window. This is a problem we are not concerned with as our REVC does not address the case of overflow in the frame number. We do not make the frame number bounded in the FSFW, we only make the amount of information in the history map bounded to F frames; the frame does not rollover after F numbers, therefore it can still function correctly even if a message from a frame previous to F frames is received. This older message will however not contain any new information to be stored in the current timestamp, as all events previous to F frames are no longer important to track because they become incomparable as given by Predicate 1. This older message will not be misinterpreted by our framework as a newer message, creating an inconsistency for new events, but rather correctly discarded as an old message. A problem would occur if we actually were to reach overflow of the frame, which (if we suppose f to be stored in a 64 bit unsigned integer number) would happen after $2^{64} - 1$ frames. Therefore if we were to address this concern practically, we would also need a

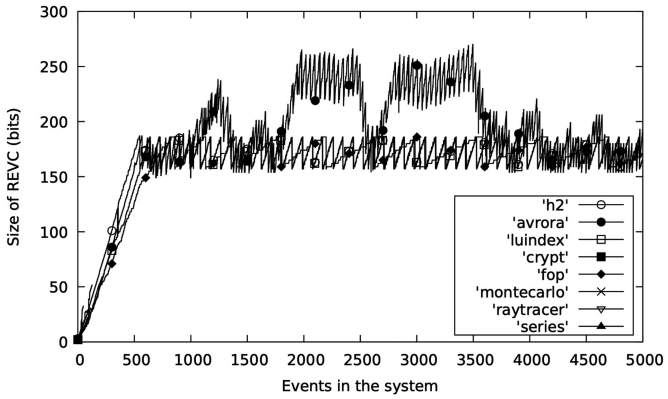

Fig. 3. Growth rate for the bounded REVC with $F = 5$.

similar communication predicate, stating that all messages need to be received within $2^{64} - 1$ frames, so no message can be received after $2^{64} - 1$ frames from when it is sent. This is a very particular case, which is comparable to what would happen if the vector clock were to overflow, which we do not concern ourselves with in our presentation of the REVC, as we assume that the system will not run for so long for that to be a real problem.

We have again gathered experimental data to support our claim that our formulation leads to a bounded implementation of the REVC. We show here the results that demonstrate the bounded growth in terms of space of the FSFW REVC. Furthermore, in Section 6.2 we show the results of our dynamic race detection application exploiting the FSFW REVC, evaluated in terms of time performance.

Fig. 3 shows the same benchmark programs that have been evaluated in Section 5.1, run using the new REVC formulation with $F = 5$. The bounded growth is clearly visible as the linear increase in size of the REVC instance reaches a plateau when the maximum number of stored frames reaches *F*. The growth rate is then reduced to an average of 0, as the actual size of the REVC instance fluctuates around the plateau. This fluctuation is caused by the EVC value of the current execution frame, whose size in bits oscillates between a minimum given by the number of bits needed to represent its prime number, and a maximum given by the chosen threshold of *n* bits which triggers the reset operation. Fig. 2 shows a zoomed-in view for the first 200 events; this does not change from the zoomed-in view for Fig. 1 as the plateau, as shown in Fig. 3, is not yet reached.

We note again the importance of the number of threads in the height of the plateaus. This is easily explained as we have stated in Section 4.1.2 that the upper bound for one entry in the history map is directly proportional to the number of threads. Therefore, the same must be for the upper bound on the total size of the history map when the plateau is reached, which will be equal to the upper bound for one entry multiplied by *F*.

We have showed how the formulation of the REVC allows to easily achieve a bounded solution both in time and space requirements, which can also be easily configured to meet the performance requirements of the application. This is of paramount importance to present the Resettable Encoded Vector Clock as a practically usable and scalable logical clock implementation in real-world applications. This achievement has been obtained at the expense of precision in the general case, as not all events are comparable under a FSFW REVC implementation. However, as we will show in our practical application detailed in Section 6, there are real-world scenarios in which the implementation can be tuned to eliminate the loss in precision while maintaining a high scalability.

5.3 Differential Merge Technique

We have stated how the merge operation's performance is related to the number of stored frames in the history map, and therefore it degrades as such number increases. We propose now a further optimization to the REVC which we call the *Differential Merge Technique (DMT)*. This optimization aims at reducing the impact that a high number of stored frames has on the *merge* operation. It derives from the observation that, based on the standard definition of the merge operation for the REVC, each time an instance is merged into another, all the frames that are stored in the first timestamp's history map need to be merged with the corresponding frames in the second timestamp's history map. Merging such frames practically translates to performing the EVC *merge* operation, which is nothing more than finding the *least common multiple* between the two numbers. However, if the two values to be merged are e_1 and e_2, and e_2 is already a multiple of e_1, then merging e_1 into e_2 will not actually change the value of e_2, as the result of $lcm(e_1, e_2) = e_2$. Next, we explain how to exploit this observation.

Let us analyze the following scenario: suppose that REVC (f_2, e_2, m_2) is merged into REVC (f_1, e_1, m_1), then a number of instructions are executed and finally (f_2, e_2, m_2) needs to be merged again into (f_1, e_1, m_1). When the *merge* operation addresses the history maps, most likely only a subset of the frames of m_2 actually needs to be considered for merging. This is because, as stated, only the frames of m_2 for which the EVC values have changed since the previous merge will produce a new result. For all other frames, however, the EVC values contained in m_1 are already multiples of the corresponding values contained in m_2. Therefore, performing the *least common multiple* operation at every merge for all frames introduces an unnecessary overhead, that can be avoided by defining a smarter merge operation. Clearly, however, each thread t would also need to keep track of whether each frame f inside its history map has been modified since the last merge operation with *each* other thread t', which would result in a higher storage requirement. Once again, this optimization technique presents a tradeoff between space and performance. Let it be noted, however, that this optimization does not affect the precision of the protocol, as no assumptions are made on the events that can be compared, and therefore we claim that the results of using a *REVC with DMT* are always equivalent to those of a traditional REVC.

We now define an enhanced version of the REVC, ready to be employed with the Differential Merge Technique. This extended REVC is defined as the tuple (f, e, m, d) where *f* is the current frame, *e* is the current EVC value, *m* is the history map of past frames and corresponding EVC values and finally *d* is a *difference map*, containing for each other thread a list of frames whose EVC value contained in *m* has been modified since the last merge into the REVC of that other thread. The difference map is a hash map that is implemented as a collection of lists. Therefore, a thread needs a

collection of lists to keep track of merges associated with k number of threads that communicate.

Supposing that REVC (f_2, e_2, m_2, d_2) is to be merged into REVC (f_1, e_1, m_1, d_1), p represents the prime number of the current thread (corresponding to (f_1, e_1, m_1, d_1)) and its unique identifier, and M represents the merge operation for EVCs, Algorithms 6 and 7 define the pseudocode for the new merge operation. (In Algorithm 7, we consider each frame of the second thread that corresponds to (f_2, e_2, m_2, d_2) that has been updated since the last merge with the first thread.) The reset function remains unchanged and its pseudocode was defined in Algorithm 2. Let it be noted how, when the DMT is used (Algorithm 7), the number of *lcm* operations depends only on the number of items present in the difference map for the merging thread, i.e., the frames updated after the last merge between the two threads. Such number is expected to remain constant as the program progresses and older frames do not get updated anymore, in contrast with the number of *lcm* operations performed by the pseudocode shown in Algorithm 4, which depends on the total number of frames stored in the history map.

Algorithm 6. Merge Operation With Differential Merge Technique

Input: Two REVC instances (f_1, e_1, m_1, d_1) and (f_2, e_2, m_2, d_2), and a prime number p
Output: The updated REVC instance

```
1: if f1 > f2 then
2:    m1.put(f2, M(m1.get(f2), e2));
3:    m1 = historyMerge(m1, m2, d2, p);
4: else if f2 > f1 then
5:    m1.put(f1, e1);
6:    f1 = f2;
7:    e1 = e2;
8:    m1 = historyMerge(m1, m2, d2, p);
9: else
10:   temp = M(e1, e2);
11:   if overflow(temp) then
12:      (f1, e1, m1) = reset(f1, temp, m1, 1);
13:   else
14:      e1 = temp;
15:   end
16:   m1 = historyMerge(m1, m2, d2, p);
17: end
18: return (f1, e1, m1, d1);
```

Algorithm 7. `historyMerge` Function With Differential Merge Technique

Input: Two history maps m_1 and m_2, a difference map d and a unique id p
Output: The merged history map

```
1: foreach f in d.get(p) do
2:    if m1.contains(f) then
3:       m1.put(f, M(m1.get(f), m2.get(f)));
4:    else
5:       m1.put(f, m2.get(f));
6:    end
7: end
8: d.get (p).empty();
9: return m1;
```

Finally, to complete the formulation of the Differential Merge Technique, the rules to update the difference maps need to be detailed. In practice, to maintain such maps consistent, it is sufficient that at each modification of an entry in the history map of a thread's REVC, that thread adds that frame to all lists that are present in the difference map. In this way, the thread is marking that frame as an updated one and therefore that frame will be picked up by any subsequent merge before being cleared again in the difference map.

6 REVC for Dynamic Race Detection

We now explore a practical application of logical clocks, dynamic race detection, with the objective of showing an application of our REVC and evaluating its performance with respect to traditional logical clock implementations. We present a modified version of the DJIT$^+$ protocol that exploits the REVC in place of the VC to track causality relations among events. Finally, we analyze the impact of the optimization techniques and show that they can achieve significant performance gain, as will be presented in the analysis of the results.

Our work on dynamic race detection is carried out on a shared memory system. In order to apply the REVC to the DJIT$^+$ protocol, instead of using vector clocks, each thread maintains an instance of the REVC and is assigned a unique prime number that will be used for operations on it. Furthermore, each lock object is assigned an instance of the REVC as well, which is initialized as $f = 1$, $e = 1$ and m as an empty map. The rules for updating and propagating the values of the Resettable Encoded Vector Clock in a multi-threaded environment on a shared memory model were presented in Section 4.2.

Each memory location is labeled with a status which can be either *READ*, *WRITE* or *READ-SHARED*. Such status traces the latest access operation which has been performed on the memory location. When the status is *READ-SHARED*, multiple concurrent read operations have been performed on the memory location, and as such they are all being tracked to ensure that no subsequent write operation can happen concurrently to any of those read operations.

Given the semantics of the status label, when the memory location is labeled with either *READ* or *WRITE*, the latest operation performed on it has been found to have *happened-after* all other previous operations. Therefore, only the current frame and EVC value of the REVC of the thread that has performed such operation are needed in order to be able to detect possible races. On the other hand, when the status is *READ-SHARED*, multiple concurrent read operations have been performed, therefore a structure needs to be saved containing for each read operation the current frame and EVC value from the REVC of the thread performing it. Clearly, EVC values pertaining to the same frames can be merged together to reduce the amount of information that needs to be stored. Finally, we define a slightly different behavior for the *comparison* operation when applied to this latter structure. In such case, in fact, it is necessary to apply the usual REVC comparison operation to all the frames contained in the defined structure, rather than just to one of them. This ensures that all the operations that are being tracked by the REVC instance have happened before the timestamp that is being checked.

Using the defined structures, we can now identify races using the Resettable Encoded Vector Clock. Let us suppose that thread t with REVC (f, e, m) accesses a memory location which is labeled with status s and REVC (f_1, e_1, m_1) due to access by another thread u. It follows from what we have previously defined that, if the access operation is a write and it is not true that $(f_1, e_1, m_1) \rightarrow (f, e, m)$, then either a *Read-Write race* or a *Write-Write race* is detected based on s. On the other hand, if the access operation is a read, a *Write-Read race* race can be detected only if s is labeled as *WRITE* and it is not true that $(f_1, e_1, m_1) \rightarrow (f, e, m)$.

Finally, after the thread performing an access to the memory location has tested the operation for possible races, it can update the information that is stored for that memory location with the data from this latest access.

6.1 REVC Optimization Techniques

With the current formulation of our modified DJIT$^+$ protocol, the size of the REVC instances is allowed to grow unboundedly and possibly overflow the maximum amount of memory that is allocated for the application or incur significant overhead for *merge* operations. In practical cases, this might still be acceptable, especially for applications that exhibit a behavior in which the REVCs have a very low growth rate, for example applications that have few synchronization points among the threads, and concentrate mainly on parallel work. However, in general, the optimization techniques that we presented in Sections 5.2 and 5.3 (FSFW and DMT) need to be employed to reduce both space and time overheads and improve overall performance of the implementation.

6.1.1 Practical Consequences of FSFW

When we presented the *Fixed Size Frame Window*, we stated that this optimization technique allows to reduce both time and space requirements, at the expense of precision. We now analyze the practical implications of this when applied to the dynamic race detection protocol.

Given a certain frame window size F, the protocol will only be able to detect races that happen within the last F frames. In fact, the comparison operation of the REVC is used in our modified version of the DJIT$^+$ protocol only for race detection purposes, and any comparison operation always uses as one of the operands a thread's current clock value. Therefore, only when the two access events under investigation are comparable (as given by Predicate 1), a possible race condition can be analyzed. This means that races can be detected only if the previous access to the memory location has happened within the last F frames from the thread's current frame. This optimization technique, therefore, clearly presents a tradeoff between performance and precision.

The intuition that stands behind the adoption of such optimization, however, is the fact that a race condition, by definition, is a tentative access to a memory location by two threads which are accessing it *concurrently*, and is therefore highly probable that the two operations will happen within a low number of synchronization points one from the other. By selecting a fairly low value of F, we still allow a relatively high number of synchronizations to happen between the two accesses, and therefore we should still maintain a relatively high precision. This intuition is confirmed by the data that we collected during our practical experimentation. We will show in our results that, despite choosing very low values of F, our system is still capable of detecting races with 100 percent precision over the suite of applications that we used for evaluation.

6.2 Evaluation

In this section we analyze and evaluate our solution in terms of performance, by comparing it to the other tools that have already been developed for Dynamic Race Detection. We have implemented our tool as a backend tool in the dynamic analysis framework RoadRunner [10]. In order to be able to provide a fair evaluation of our system with respect to the traditional DJIT$^+$ protocol, we also developed an implementation of DJIT$^+$ on the same framework. FastTrack, on the other hand, was already implemented on top of RoadRunner, as it has been studied and developed by Flanagan and Freund as well.

The applications that were chosen for evaluation are a subset of the applications that are found in the DaCapo Benchmarking Suite [36] and the Java Grande Benchmarking Suite [37]. Those two suites are composed of Java programs that have been designed to emulate non-trivial loads and be representative of intensive calculations. The applications that have been chosen exhibit multi-threaded behaviour and very diverse semantics, in order to be able to test the system on a relatively complete set of programs ranging from a very high usage of synchronization to a high parallelization of the workload. All experiments have been carried out on a system with a Dual-Core Intel i7 2.8 GHz processor and 12 GB of RAM running Linux Ubuntu 18.04 and Java 8, with multithreading enabled. Table 1 shows the execution time in milliseconds of each application, for each tool that has been tested. We ran the code twice for warmup of caches. Then for each configuration we took the readings of 10 runs and reported the average. As the same platform was used for REVC, DJIT$^+$, and FastTrack, the average speedup ratios in Table 1 should not be affected by the platform.

The first column shows the execution time of the application without the overhead of a dynamic analysis tool, while the following columns show the execution times of the same applications when they are instrumented by RoadRunner and processed by a dynamic analysis tool. Our modified version of DJIT$^+$ which exploits the REVC is tested in various flavors, first without any optimization technique, then using the sizes 30, 5 and 1 for the Fixed Size Frame Window, in varying combinations with and without the Differential Merge Technique. As expected, when the REVC is used without any limitation to the amount of frames that are stored as part of the history, applications that perform frequent synchronization, and therefore produce high growth in the REVCs, use up all the available memory and return an out-of-memory error (OOM). This is the case for *avrora*, *h2*, *luindex* and *lusearch*. It is however interesting to notice that specific applications which make frequent use of comparison operations, but do not require high growth of the logical clocks, perform the best without any form of optimization, because the overhead of the additional processing or maintenance of the required data structures for the optimization

TABLE 1
Execution Time Comparison

Application	No tool	DJIT$^+$	REVC DJIT$^+$	REVC DJIT$^+$ DMT	REVC DJIT$^+$ FSFW 30	REVC DJIT$^+$ FSFW 5	REVC DJIT$^+$ FSFW 30 + DMT	REVC DJIT$^+$ FSFW 5 + DMT	REVC DJIT$^+$ FSFW 1	FastTrack
avrora	10235	73687	OOM	OOM	74021	50290	49146	45968	43066	27514
crypt	73	5008	9148	9760	9474	9298	9343	9273	9217	1339
fop	440	6376	15988	4194	4413	4366	4117	4112	4298	3509
h2	6205	116125	OOM	OOM	120144	122619	112219	111438	109302	48606
lufact	53	6266	3398	3720	3701	3617	3307	3376	3302	2162
luindex	523	23172	OOM	OOM	15768	13106	12841	12719	12919	6794
lusearch	1680	87460	OOM	OOM	70554	60029	65552	67312	56703	31970
moldyn	414	67798	41133	38956	41789	40443	38563	39948	39311	23392
montecarlo	928	29003	8986	8291	9988	9150	8947	8905	8706	4506
raytracer	542	83680	25129	42969	32293	47168	47201	43909	39620	17841
series	1507	1663	1663	1647	1672	1670	1654	1647	1693	1634
sor	283	19752	10116	9469	10353	9924	9925	9878	10343	7658
sparsematmult	104	29851	40090	38394	39403	40782	39052	39503	41499	18563
AVG SPEEDUP OVER DJIT$^+$	-	-	-	-	**1.470**	**1.520**	**1.553**	**1.564**	**1.601**	**2.992**

techniques is higher than the performance benefit that is obtained. This behaviour can be noticed in particular for *raytracer* and *crypt*.

On average, however, a reduced size of the frame window results in higher overall performance as would be reasonable to expect. Similarly, the results show that the Differential Merge Technique is able to increase the overall performance by reducing the overhead of merge operations. The last line clearly presents the performance comparison, by showing the average speedup achieved over the DJIT$^+$ protocol across all benchmarks. The configurations that caused out-of-memory errors do not present an aggregate average measurement as it would not be comparable to the other configurations due to the lack of data.

Our REVC-based application is able to outperform by up to 1.6 times the traditional VC-based DJIT$^+$ protocol, even if it is not able to achieve performance that is comparable to the state-of-the-art FastTrack protocol that employs Scalar Clocks. We claim, however, that this result is a promising achievement as it allows us to show that the REVC can have practical applications, which are competitive for scenarios in which no alternatives to a traditional Vector Clock implementation are available.

We have stated, when presenting the Fixed Size Frame Window optimization, that activating such technique would improve performance at the theoretical expense of precision, as not all pairs of events remain comparable, and this can lead to missed race conditions. We also intuitively stated, however, that because of the nature of the race condition detection problem, most races would have probably derived from events that are very close (in terms of number of instructions) rather than several frames apart. We have now corroborated such intuition with some experimental data, as our results show that all configurations that we have tried, down to a Fixed Size Frame Window of just 1 frame, are able to detect the same number of races for all applications, which also corresponds to the same number of races found by DJIT$^+$ and FastTrack. Therefore, for the applications tested, the precision of our tool, in all its configurations, is not affected by the optimization techniques that are employed. Note that as this is a dynamic race detection system, it will be only able to detect races that happen in the current program run.

We list two limitations of our tool. Our practical implementation cannot be applied to any programming language that does not run on the JVM. The concepts, however, are general and should be replicable, but that will require some further research in that field. Another limitation of the bounded-space implementation of REVC is that for a new application, one would have to guess the optimal size F of the FSFW window so as to detect all data races, by a trial-and-error approach or by comparing results with DJIT$^+$ or FastTrack.

7 Conclusion

In this paper we have presented the Resettable Encoded Vector Clock, a logical clock implementation that is built on top of the EVC with the objective of tackling its main drawbacks: high growth rate and unbounded growth. We have defined and formalized such protocol, and showed its applicability in both shared memory systems and message passing systems. We have presented experimental results that show how we have been able to achieve the desired objectives. In particular, the REVC presents a linear growth rate with respect to the total number of events being executed in the system, which is in contrast with the exponential growth rate presented by the EVC. Furthermore, the REVC's definition allows to easily tune the implementation to place an upper bound on the storage requirements. We have showed how this result can be achieved through the use of the FSFW at the theoretical expense of precision. We have however argued that practical applications rarely

require unconstrained comparability among the events, and we have showed how there exist real-world scenarios in which the Bounded REVC can achieve no loss in precision. We have also presented the DMT optimization technique, to consistently mitigate the overhead of the merge operation derived from a high number of stored frames. Finally, we have explored a practical application of the REVC by applying it to the dynamic race detection problem, with the aim of evaluating its performance with respect to other traditional logical clock implementations, and showing its practical applicability in real-world scenarios.

Our results show how the REVC is not just a theoretical concept, but it is applicable to practical problems and can compete in terms of both space and time requirements with other known protocols. The REVC has been designed with scalability and adaptability in mind. Its formulation contains several intrinsic tradeoffs that can be easily tuned by enabling or disabling optimization techniques, and choosing between Bounded and Unbounded implementations. These configurations provide the REVC with a much higher adaptability to very different scenarios, which cannot be found in other logical clock implementations. We believe that these results are promising and conjecture that they can be the starting point for a number of developments that can help introduce new theoretical and practical tools to more efficiently tackle several problems in distributed systems, that require causality analysis as part of their solutions.

Lidbury and Donaldson [38] present a state-of-the-art dynamic race detector for C/C++11, that also uses vector clocks. Hence, the REVC idea proposed can be applied to their C/C++11 framework. It would be interesting to measure the performance improvement that may be obtainable by using REVC instead of vector clocks in their C/C++11 framework. This is a future research work item. Another future work item is to examine the benefits of REVC in parallel programming frameworks such as OpenMP.

REFERENCES

[1] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distrib. Comput.*, vol. 7, no. 3, pp. 149–174, 1994. [Online]. Available: https://doi.org/10.1007/BF02277859

[2] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge, U.K.: Cambridge Univ. Press, 2011.

[3] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[4] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, pp. 28–33, Aug. 1991. [Online]. Available: https://doi.org/10.1109/2.84874

[5] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. Workshop Parallel Distrib. Algorithms*, 1989, pp. 215–226.

[6] A. D. Kshemkalyani, A. Khokhar, and M. Shen, "Encoded vector clock: Using primes to characterize causality in distributed systems," in *Proc. 19th Int. Conf. Distrib. Comput. Netw.*, 2018, pp. 12:1–12:8. [Online]. Available: http://doi.acm.org/10.1145/3154273.3154305

[7] A. D. Kshemkalyani, M. Shen, and B. Voleti, "Prime clock: Encoded vector clock to characterize causality in distributed systems," *J. Parallel Distrib. Comput.*, vol. 140, pp. 37–51, 2020. [Online]. Available: https://doi.org/10.1016/j.jpdc.2020.02.008

[8] B. Charron-Bost, "Concerning the size of logical clocks in distributed systems," *Inf. Process. Lett.*, vol. 39, no. 1, pp. 11–16, 1991. [Online]. Available: https://doi.org/10.1016/0020-0190(91)90055-M

[9] A. D. Kshemkalyani and B. Voleti, "On the growth of the prime numbers based encoded vector clock," in *Proc. 15th Int. Conf. Distrib. Comput. Internet Technol.*, 2019, pp. 169–184. [Online]. Available: https://doi.org/10.1007/978-3-030-05366-6_14

[10] C. Flanagan and S. N. Freund, "The roadrunner dynamic analysis framework for concurrent programs," in *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, 2010, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/1806672.1806674

[11] E. Pozniansky and A. Schuster, "MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs: Research articles," *Concurrency Comput., Pract. Experience*, vol. 19, no. 3, pp. 327–340, Mar. 2007. [Online]. Available: https://doi.org/10.1002/cpe.v19:3

[12] M. Singhal and A. D. Kshemkalyani, "An efficient implementation of vector clocks," *Inf. Process. Lett.*, vol. 43, no. 1, pp. 47–52, 1992. [Online]. Available: https://doi.org/10.1016/0020-0190(92)90028-T

[13] S. Meldal, S. Sankar, and J. Vera, "Exploiting locality in maintaining potential causality," in *Proc. 10th Annu. ACM Symp. Princ. Distrib. Comput.*, 1991, pp. 231–239. [Online]. Available: http://doi.acm.org/10.1145/112600.112620

[14] F. J. Torres-Rojas and M. Ahamad, "Plausible clocks: Constant size logical clocks for distributed systems," *Distrib. Comput.*, vol. 12, no. 4, pp. 179–195, 1999. [Online]. Available: https://doi.org/10.1007/s004460050065

[15] P. A. S. Ward and D. J. Taylor, "A hierarchical cluster algorithm for dynamic, centralized timestamps," in *Proc. 21st Int. Conf. Distrib. Comput. Syst.*, 2001, pp. 585–593.

[16] P. A. S. Ward and D. J. Taylor, "Self-organizing hierarchical cluster timestamps," in *Proc. 7th Int. Euro-Par Conf. Parallel Process.*, 2001, pp. 46–56.

[17] P. S. Almeida, C. Baquero, R. Gonçalves, N. M. Preguiça, and V. Fonte, "Scalable and accurate causality tracking for eventually consistent stores," in *Proc. 14th IFIP WG 6.1 Int. Conf. Distrib. Appl. Interoperable Syst.*, 2014, pp. 67–81.

[18] P. S. Almeida, C. Baquero, and V. Fonte, "Interval tree clocks," in *Proc. 12th Int. Conf. Princ. Distrib. Syst.*, 2008, pp. 259–274.

[19] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *Proc. 18th Int. Conf. Princ. Distrib. Syst.*, 2014, pp. 17–32.

[20] S. Yingchareonthawornchai, D. N. Nguyen, S. S. Kulkarni, and M. Demirbas, "Analysis of bounds on hybrid vector clocks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 1947–1960, Sep. 2018. [Online]. Available: https://doi.org/10.1109/TPDS.2018.2818700

[21] S. S. Kulkarni and N. H. Vaidya, "Effectiveness of delaying timestamp computation," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2017, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/3087801.3087818

[22] X. Zheng and V. K. Garg, "An optimal vector clock algorithm for multithreaded systems," in *Proc. 39th IEEE Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 2188–2194.

[23] A. D. Kshemkalyani and A. Misra, "The bloom clock for characterizing causality in distributed systems," in *Proc. 23rd Int. Conf. Advances Netw.-Based Inf. Syst.*, 2020, pp. 269–279. [Online]. Available: https://doi.org/10.1007/978-3-030-57811-4_25

[24] L. Ramabaja, "The bloom clock," *CoRR*, vol. abs/1905.13064, 2019. [Online]. Available: http://arxiv.org/abs/1905.13064

[25] M. Shen, A. D. Kshemkalyani, and A. A. Khokhar, "Detecting unstable conjunctive locality-aware predicates in large-scale systems," in *Proc. IEEE 12th Int. Symp. Parallel Distrib. Comput.*, 2013, pp. 127–134.

[26] T.-D. Diep, K. Fürlinger, and N. Thoai, "MC-CChecker: A clock-based approach to detect memory consistency errors in MPI one-sided applications," in *Proc. 25th Eur. MPI Users' Group Meeting*, 2018, pp. 9:1–9:11. [Online]. Available: http://doi.acm.org/10.1145/3236367.3236369

[27] T.-D. Diep, K. Trung Pham, K. Furlinger, and N. Thoai, "A timestamping system to detect memory consistency errors in MPI one-sided applications," *Parallel Comput.*, vol. 86, pp. 36–44, Aug. 2019.

[28] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985. [Online]. Available: http://doi.acm.org/10.1145/214451.214456

[29] L. Yen and T. Huang, "Resetting vector clocks in distributed systems," *J. Parallel Distrib. Comput.*, vol. 43, no. 1, pp. 15–20, 1997. [Online]. Available: https://doi.org/10.1006/jpdc.1997.1330

[30] A. Arora, S. Kulkarni, and M. Demirbas, "Resettable vector clocks," in *Proc. 19th Annu. ACM Symp. Princ. Distrib. Comput.*, 2000, pp. 269–278. [Online]. Available: http://doi.acm.org/10.1145/343477.343628
[31] J. Couvreur, N. Francez, and M. G. Gouda, "Asynchronous unison (extended abstract)," in *Proc. 12th Int. Conf. Distrib. Comput. Syst.*, 1992, pp. 486–493.
[32] J. Misra, "Phase synchronization," *Inf. Process. Lett.*, vol. 38, no. 2, pp. 101–105, 1991. [Online]. Available: https://doi.org/10.1016/0020-0190(91)90229-B
[33] S. S. Kulkarni and A. Arora, "Multitolerant barrier synchronization," *Inf. Process. Lett.*, vol. 64, no. 1, pp. 29–36, 1997. [Online]. Available: https://doi.org/10.1016/S0020-0190(97)00143-9
[34] D. K. Panda, "Fast barrier synchronization in wormhole k-ary n-cube networks with multidestination worms," *Future Gener. Comput. Syst.*, vol. 11, no. 6, pp. 585–602, 1995. [Online]. Available: https://doi.org/10.1016/0167-739X(95)00026-O
[35] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *Proc. 30th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2009, pp. 121–133. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542490
[36] S. M. Blackburn *et al.*, "The daCapo benchmarks: Java benchmarking development and analysis," *SIGPLAN Notices*, vol. 41, no. 10, pp. 169–190, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1167515.1167488
[37] Java Grande Forum, "Java Grande benchmarking suite," Accessed: Jul. 31, 2019. [Online]. Available: http://www.javagrande.org/
[38] C. Lidbury and A. F. Donaldson, "Dynamic race detection for C++11," in *Proc. 44th ACM SIGPLAN Symp. Princ. Program. Lang.*, 2017, pp. 443–457.

Tommaso Pozzetti received the BS degree in computer engineering from the Polytechnic University of Turin, Italy, in 2017 and two MS degrees in computer engineering and computer science from the Polytechnic University of Turin, Italy, and the University of Illinois at Chicago, Chicago, Illinois, in 2019 respectively. His thesis work, under the advisement of professor Ajay D. Kshemkalyani, was focused on causality analysis in distributed systems and dynamic race detection in shared memory environments. He recently joined Vail Systems Inc., as a DevOps engineer.

Ajay D. Kshemkalyani (Senior Member, IEEE) received the BTech degree in computer science and engineering from the Indian Institute of Technology, Bombay, India, in 1987, and the MS and PhD degrees in computer and information science from The Ohio State University, Columbus, Ohio, in 1988 and 1991, respectively. He spent six years at IBM Research Triangle Park working on various aspects of computer networks, before joining academia. He is currently a professor with the Department of Computer Science, University of Illinois at Chicago, Chicago, Illinois. His research interests are in distributed computing, distributed algorithms, computer networks, and concurrent systems. In 1999, he received the National Science Foundation Career Award. He served on the editorial board of the Elsevier Journal, *the Computer Networks* and the *IEEE Transactions on Parallel and Distributed Systems*. He has co-authored a book entitled Distributed Computing: Principles, Algorithms, and Systems (Cambridge University Press, 2011). He is a distinguished scientist of ACM.