Journal of Parallel and Distributed Computing 140 (2020) 37–51

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Prime clock: Encoded vector clock to characterize causality in distributed systems

Ajay D. Kshemkalyani [*], Min Shen, Bhargav Voleti

Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, USA

ARTICLE INFO

Article history:
Received 9 July 2019
Received in revised form 21 November 2019
Accepted 13 February 2020
Available online 22 February 2020

Keywords:
Causality
Vector clock
Prime numbers
Encoding
Happened-before relation

ABSTRACT

The vector clock is a fundamental tool for tracking causality in distributed applications. Unfortunately, it does not scale well to large systems because each process needs to maintain a vector of size n, where n is the total number of processes in the system. To address this problem, we propose the prime clock, which is based on the encoding of the vector clock using prime numbers and uses a single number to represent vector time. We propose the operations on the encoded vector clock (EVC). We then show how to timestamp global states and how to perform operations on the global states using the EVC. Using a theoretical analysis and a simulation model, we evaluate the growth rate of the size of the EVC. The EVC is seen to grow very fast and hence it does not appear to offer a general purpose practical replacement of vector clocks. To address this drawback, we propose several scalability techniques for the EVC that can allow the use of the EVC in practical applications. We then present two case studies of detecting memory consistency errors in MPI one-sided applications and of dynamic race detection in multi-threaded environments, that use a combination of two of these scalability techniques. The results show that the EVC 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.

1. Introduction

The ordering of events and states is a fundamental operation in the analysis of distributed executions. It is used in distributed applications such as checkpointing and rollback recovery, mutual exclusion, debugging, and replication-based data stores [22,36]. For example, in replication-based data stores, the ordering of reads and updates to a shared object is required to determine the object's most recent value. Logical clocks have been proposed to order events without the need for tightly synchronized physical clocks. These logical clocks order events based on the *causality* relation on events, defined by Lamport [27]. The ordering of events based on the causality relation is also required for enforcing causal consistency in data stores. Thus, tracking causality and evaluating causality between different events and between different states of a distributed execution is a fundamental challenge.

The simplest form of logical clocks, proposed by Lamport [27], uses a scalar clock at each process in the system. If two events are related by causality, their scalar clock values are so ordered. However, the causality relation between events cannot be inferred from the values of the scalar clocks of events. To overcome this drawback, vector clocks have been proposed [10,28]. The vector clock is a fundamental tool for tracking causality in distributed applications. Unfortunately, vector clocks do not scale well to large systems because each process needs to maintain a vector of size n, where n is the total number of processes in the system. This has been shown to be a lower bound [5]. Several works in the literature attempted to reduce the size of vector clocks [25,29,38,39], but they had to make some compromises in accuracy or alter the system model, and in the worst-case, were as lengthy as vector clocks.

To address the above scalability problem, we propose the encoding of the vector clock using prime numbers to use a single number to represent vector time. Thus, we get the properties of the vector clock by maintaining only a single number – a big integer – at each process. We propose the tick, merge, and comparison operations on the encoded vector clock (EVC). We then show how to timestamp global states and how to perform operations – namely, the union, intersection, common causal past computation, and comparison – on the global states using the EVC. All these operations on the EVC values of events and on EVC values of global states have equal or lower time complexity than the corresponding operations on traditional vector clocks in the uniform cost model. As the EVC values are big integers, we also express the time complexities of the operations on EVCs in the logarithmic cost model. However, these complexities are

[*] Corresponding author.
E-mail address: ajay@uic.edu (A.D. Kshemkalyani).
URL: http://www.cs.uic.edu/~ajayk (A.D. Kshemkalyani).

https://doi.org/10.1016/j.jpdc.2020.02.008
0743-7315/© 2020 Elsevier Inc. All rights reserved.

incomparable with the complexities of operations on traditional vector clocks in the uniform cost model.

Although the EVC is a single big integer rather than a vector of integers, the drawback of the EVC is that it grows very fast. Using a theoretical analysis and a simulation model, we evaluate the growth rate of the size of the EVC. Assuming that the integer data type used by programming languages is represented in 32 bits, we compute how many system events it takes until the size of the single big integer number EVC at some process becomes $32n$. Our simulation results confirm the intuition and theoretical analysis that the single number EVC grows very fast. Thus, although the EVC is mathematically elegant, it does not appear to offer a general purpose practical replacement of vector clocks due to its high growth rate.

To overcome the drawback that the EVC grows very fast, we then propose four techniques. These are: (i) ticking the clock only at application-relevant events, (ii) the use of detection regions within which the EVC is tracked, (iii) resetting the EVC when the size of the EVC at some process reaches a predefined threshold such as $32n$ or when a global synchronization point is reached, and (iv) using logarithms of the EVC rather than the EVC itself. Depending on the application, a judicious use of these scalability techniques can control the size of the EVC and guarantee that the size of the EVC never exceeds the size of the traditional vector clock. We then present two case studies: (i) detecting memory consistency errors in MPI one-sided applications [8,9], and (ii) dynamic race detection in multi-threaded environments [32], that use a combination of two of the proposed scalability techniques. The results show that the EVC in conjunction with the scalability techniques 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. In conjunction with resetting, the EVC is seen to be designed with scalability and adaptability to very different scenarios. We believe these achievements are promising and 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.

An earlier version of this paper appeared as [21]. The present paper is a revision that contains new material including theorem/claims with proofs, a theoretical analysis of the growth rate of the EVC, and sections on the system model and on related works. A simulation section to evaluate the growth of the EVC and the case study of detecting memory consistency errors in MPI one-sided applications are based on [23].

In Section 2, we give the system model and present preliminaries. In Section 3, we give the encoding of the vector clock, and the operations on the encoded vector clock. In Section 4, we give mechanisms to timestamp global states and operations on the global states using EVC. In Section 5, we give simulation results on the growth of the EVC. In Section 6, we propose scalability techniques for EVC. In Section 7, we give our two case studies. In Section 8, we discuss related work. We give concluding remarks in Section 9.

2. System model

A distributed system is modeled as an undirected graph (P, L), where P is the set of processes and L is the set of communication links connecting them. Let $n = |P|$ and let d denote the degree of the graph. Between any two processes, there may be at most one logical channel over which the two processes communicate asynchronously. A logical channel from P_i to P_j is formed by paths over links in L. We do not assume FIFO logical channels; thus the messages may be delivered out of order. Let c denote the number of logical channels in the system.

The execution of process P_i produces a sequence of events $E_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$, where e_i^k is the kth event at process P_i. An event at a process can be an *internal* event, a *message sending* event, or a *message reception* event. Let $E = \bigcup_{i \in P} \{e \mid e \in E_i\}$ denote the set of events in a distributed execution. The causal precedence relation between events induces an irreflexive partial order on E. This relation is defined as Lamport's "happened before" relation [27], and denoted as $\rightarrow$. An execution of a distributed system is thus denoted by the tuple $(E, \rightarrow)$. Lamport designed the scalar clock, which is a function C that assigns integer timestamps to events such that if $e \rightarrow f$, then $C(e) < C(f)$. However, the drawback of scalar clocks is that $C(e) < C(f)$ does not imply that $e \rightarrow f$.

Mattern [28] and Fidge [10] designed the vector clock which assigns a vector V to each event such that: $e \rightarrow f \iff V(e) < V(f)$. This is called the *strong clock consistency condition*. Thus, the vector clock overcomes the drawback of the scalar clock. Each process P_i maintains a vector clock V. Events are timestamped by the current clock value. The vector clocks, initialized to the 0-vector, are updated by the following rules.

1. Before an internal event happens at process P_i, $V[i] = V[i] + 1$ (local tick).
2. Before process P_i sends a message, it first executes $V[i] = V[i] + 1$ (local tick), then it sends the message piggybacked with V.
3. When process P_i receives a message piggybacked with timestamp U, it executes
   $\forall k \in [1 \ldots n]$, $V[k] = \max(V[k], U[k])$ (merge);
   $V[i] = V[i] + 1$ (local tick)
   before delivering the message.

The vector clock is a fundamental tool to characterize causality in distributed executions [22,36]. Each process needs to maintain a vector of size n, where n is the total number of processes in the system, to represent the local vector clock. Unfortunately, this does not scale well to large systems. Several works in the literature attempted to reduce the size of vector clocks [25,29,38,39], but they had to make some compromises in accuracy or alter the system model, and in the worst-case, were as lengthy as vector clocks. To address this problem, we propose the encoding of the vector clock using prime numbers to use a single number to represent vector time.

3. Encoded vector clock

Charron-Bost has shown that to capture the partial order $(E, \rightarrow)$, the size of the vector clock is the dimension of the partial order [5], which is bounded by the size of the system, n. She mentioned that vector clocks could be encoded using prime numbers. We propose and develop the technique for the vector clock to be encoded into a single number using n distinct prime numbers. The encoding of vector clocks using primes was used for detecting locality-aware conjunctive predicates in large-scale systems [37]. A vector clock containing n elements, $V = [v_1, v_2, \ldots, v_n]$, can be encoded by n distinct prime numbers $p_1, p_2, \ldots, p_n$ as:

$$Enc(V) = p_1^{v_1} * p_2^{v_2} * \cdots * p_n^{v_n}$$

However, only being able to encode a vector clock into a single number is insufficient to track causal relations. We develop the EVC technique to show how to implement the basic operations of a vector clock. The EVC at each process P_i is initialized to 1. For a vector clock to work, it needs three basic operations: local tick, merge, and compare. Below, we implement these basic operations using EVC.

Table 1
Correspondence between vector clocks and EVC.

Operation	Vector clock	Encoded vector clock
Representing clock	$V = [v_1, v_2, \dots, v_n]$	$Enc(V) = p_1^{v1} * p_2^{v2} * \cdots * p_n^{vn}$
Local tick (at process P_i)	$V[i] = V[i] + 1$	$Enc(V) = Enc(V) * p_i$
Merge	Merge V_1 and V_2 yields V where $V[j] = \max(V_1[j], V_2[j])$	Merge $Enc(V_1)$ and $Enc(V_2)$ yields $Enc(V) = LCM(Enc(V_1), Enc(V_2))$
Compare	$V_1 < V_2$: $\forall j \in [1, n]$, $V_1[j] \le V_2[j]$ and $\exists j$, $V_1[j] < V_2[j]$	$Enc(V_1) \prec Enc(V_2)$: $Enc(V_1) < Enc(V_2)$ and $Enc(V_2) \bmod Enc(V_1) = 0$

1. Initialize $t_i = 1$.
2. Before an internal event happens at process P_i,
   $t_i = t_i * p_i$ (local tick).
3. Before process P_i sends a message,
   it first executes $t_i = t_i * p_i$ (local tick),
   then it sends the message piggybacked with t_i.
4. When process P_i receives a message piggybacked with timestamp s, it executes
   $t_i = LCM(s, t_i)$ (merge);
   $t_i = t_i * p_i$ (local tick)
   before delivering the message.

Fig. 1. Operation of EVC t_i at process P_i.

3.1. Encoded vector clock operations

Local Tick: Whenever the logical time advances locally at P_i, the local component of the vector clock needs to tick. This increases the local component in the vector by 1:

$V[i] = V[i] + 1$

While using EVC, this operation is equivalent to multiplying the EVC timestamp by the local prime number p_i,

$Enc(V) = Enc(V) * p_i$

Merge: Whenever one process sends a message with a piggybacked vector clock timestamp to another process, the recipient of the message needs to merge the piggybacked vector clock timestamp with its own local vector clock. For two vector clock timestamps

$V_1 = [v_1, v_2, \dots, v_n]$ and $V_2 = [v'_1, v'_2, \dots, v'_n]$

merging them yields:

$U = [u_1, u_2, \dots, u_n]$, where $u_i = \max(v_i, v'_i)$

The encodings of V_1, V_2, and U are:

$$Enc(V_1) = p_1^{v_1} * p_2^{v_2} * \cdots * p_n^{v_n}$$
$$Enc(V_2) = p_1^{v'_1} * p_2^{v'_2} * \cdots * p_n^{v'_n}$$
$$Enc(U) = \prod_{i=1}^{n} p_i^{\max(v_i, v'_i)}$$

We do not have access to the vector components v_i and v'_i ($i = 1 \dots n$) to generate $Enc(U)$. Furthermore, it would be better to merge $Enc(V_1)$ and $Enc(V_2)$ into $Enc(U)$ without knowing the n prime numbers. (One advantage of this is protection against attacks.) We can compute $Enc(U)$ using the Fundamental Theorem of Arithmetic which states that any positive integer can be represented as a product of prime numbers, and except for their order, this representation is unique. In our context, any EVC can be uniquely represented as

$$\prod_{k=1}^{n} p_k^{v_k}, \text{ where } v_k \text{ is a non-negative integer.}$$

Applying the definition of LCM (Lowest Common Multiple) in terms of the Fundamental Theorem of Arithmetic, we have the following claim.

Claim 1.

$$Enc(U) = LCM(Enc(V_1), Enc(V_2)) = \prod_{i=1}^{n} p_i^{\max(v_i, v'_i)}$$

So, by computing the LCM of two EVC timestamps, these two timestamps can be merged without knowing the n prime numbers.

Comparison: A mechanism to compare two vector clock timestamps is needed. Let $\mathcal{V}^{\mathcal{E}}$ denote the set of vector timestamps of events. Then $(\mathcal{V}^{\mathcal{E}}, <)$ is isomorphic to $(E, \to)$ [10,28]. To compare two distinct vector clock timestamps, a component-wise comparison between the corresponding elements of two vectors is needed. The comparison has two results (the tests $V_1 < V_2$ and $V_2 < V_1$ are symmetrical):

(i) $V_1 < V_2$ if $\forall j \in [1, n]$, $V_1[j] \le V_2[j]$ and $\exists j$, $V_1[j] < V_2[j]$
(ii) $V_1 \parallel V_2$ if $V_1 \not< V_2$ and $V_2 \not< V_1$

Let $\mathcal{ENCV}^{\mathcal{E}}$ denote the set of encoded vector timestamps of events. To compare two (distinct) EVC timestamps, it is only necessary to test if $Enc(V_j) \bmod Enc(V_i) = 0$. Thus,

(i) $Enc(V_1) \prec Enc(V_2)$ if
$$Enc(V_1) < Enc(V_2) \text{ and } Enc(V_2) \bmod Enc(V_1) = 0$$

(*ii*) $Enc(V_1) \parallel Enc(V_2)$ if

$$Enc(V_1) \not\prec Enc(V_2) \text{ and } Enc(V_2) \not\prec Enc(V_1)$$

The correspondence between the three basic operations of the vector clock and EVC is shown in Table 1. Thus, the encoded vector clock t_i (initialized to 1) is operated at process P_i as shown in Fig. 1. To manipulate the EVC, each process needs to know only its own prime and not the primes of other processes. Merging two EVCs requires computing the LCM, which does not require factorization (see Section 3.2.2). Our technique encodes vector clocks of events and the encoded vector clock/timestamps of events, $(\mathcal{ENCV}^{\mathcal{E}}, \prec)$ is isomorphic to $(E, \rightarrow)$ and to $(\mathcal{V}^{\mathcal{E}}, <)$.

The operations using EVC are illustrated in Fig. 2 using an example execution over three processes.

We now prove that the EVC satisfies the strong clock consistency condition. The proof uses the Fundamental Theorem of Arithmetic.

Theorem 3.1. *The EVC operation given in Fig. 1 satisfies the strong clock consistency condition, namely, for distinct events e and f,*

$$e \rightarrow f \iff EVC(e) \prec EVC(f)$$
$$\iff \frac{EVC(f)}{EVC(e)} \in N$$

Proof. (Part 1 ($\Rightarrow$)) Assume $e \rightarrow f$. Consider the events $e = e_{k_0}, e_{k_1}, e_{k_2}, \ldots, e_{k_j} = f$ along any causal path from e to f.

We prove by induction that $EVC(f)$ is divisible by $EVC(e)$.

Induction hypothesis: $EVC(e_{k_x})$, for $0 \leq x \leq j$, is divisible by $EVC(e_{k_0})$.

The hypothesis is clearly true for the base case $x = 0$. Assume the hypothesis for any x. To prove the hypothesis for $x + 1$, we proceed as follows. Let the process at which event $e_{k_{x+1}}$ occurs be l. Observe the following from the code of Fig. 1.

1. $EVC(e_{k_{x+1}}) = EVC(e_{k_x}) * p_l$, or
2. $EVC(e_{k_{x+1}}) = LCM(EVC(e_{k_x}), EVC(g)) * p_l$, where g is some other event.

In both cases, $EVC(e_{k_{x+1}})$ is divisible by $EVC(e_{k_x})$. By the induction hypothesis, $EVC(e_{k_x})$ is divisible by $EVC(e)$ and by transitivity, $VC(e_{k_{x+1}})$ is divisible by $EVC(e)$.

(Part 2 ($\Leftarrow$)) Assume $e \not\rightarrow f$, and let event e occur at process P_i. Let e' be the latest event at P_i such that $e' \rightarrow f$. If such an e' does not exist, set it to the dummy initial event e_i^0, which by definition is assumed to happen before all actual events at all processes. Clearly, $e' \rightarrow e$. Let $EVC(e_i) = \prod_{k=1}^{n} p_k^{v_k}$ and let $EVC(e_i') = \prod_{k=1}^{n} p_k^{v_k'}$. We have $p_i^{v_i'} < p_i^{v_i}$ due to a local tick at P_i for each event from e_i' to e_i. Consider the events $e' = e_{k_0}, e_{k_1}, e_{k_2}, \ldots, e_{k_j} = f$ along any causal path from e' to f. Our proof proceeds by induction.

Induction hypothesis: For any $EVC(e_{k_x})$, where $0 \leq x \leq j$, the largest factor that is a power of p_i is $p_i^{v_i'}$.

The hypothesis is clearly true for the base case $x = 0$. Assume the hypothesis for any x. To prove the hypothesis for $x + 1$, we proceed as follows. Let the process at which event $e_{k_{x+1}}$ occurs be P_l. Note that $l \neq i$ because of our choice of e'. Observe the following from the code of Fig. 1.

1. For a send or internal event $e_{k_{x+1}}$, $EVC(e_{k_{x+1}}) = EVC(e_{k_x}) * p_l$. As $p_l \neq p_i$, from the induction hypothesis, the largest factor of $EVC(e_{k_{x+1}})$ that is a power of p_i is $p_i^{v_i'}$.
2. For a receive event $e_{k_{x+1}}$, denote the corresponding send event as s, and the event preceding the receive event as r'.

   $$EVC(e_{k_{x+1}}) = LCM(EVC(s), EVC(r')) * p_l$$

   $$= \prod_{k=1}^{n} p_k^{\max(v_k^s, v_k^{r'})} * p_l$$

   Irrespective of whether e_{k_x} is s or r', it follows from the induction hypothesis and the definition of e' that for both $EVC(s)$ and $EVC(r')$, their largest factor that is a power of p_i is less than or equal to $p_i^{v_i'}$. That is, $p_i^{v_i^s}, p_i^{v_i^{r'}} \leq p_i^{v_i'}$. As one of s and r' is e_{k_x} and $p_l \neq p_i$, hence, the largest factor of $EVC(e_{k_{x+1}})$ that is a power of p_i is $p_i^{v_i'}$.

As a consequence of the proof by induction, the largest factor of $EVC(f)$ that is a power of p_i is $p_i^{v_i'} < p_i^{v_i}$. Stated equivalently, $p_i^{v_i^f} < p_i^{v_i^e}$.

We are now ready to show that $\frac{EVC(f)}{EVC(e)} \notin N$. From the Fundamental Theorem of Arithmetic, we have

$$\frac{EVC(f)}{EVC(e)} = \frac{p_1^{v_1^f} * \ldots * p_i^{v_i^f} * \ldots * p_n^{v_n^f}}{p_1^{v_1^e} * \ldots * p_i^{v_i^e} * \ldots * p_n^{v_n^e}}$$
$$= \frac{p_1^{v_1^f} * \ldots * p_{i-1}^{v_{i-1}^f} * p_{i+1}^{v_{i+1}^f} * \ldots * p_n^{v_n^f}}{p_1^{v_1^e} * \ldots * p_i^{v_{i-1}^e} * p_i^{v_i^e - v_i^f} * p_i^{v_{i+1}^e} * \ldots * p_n^{v_n^e}}$$

As $p_i^{v_i^f} < p_i^{v_i^e}$, the denominator has a factor $p_i^{v_i^e - v_i^f}$ that does not divide any of the prime factors of the numerator. Hence, $\frac{EVC(f)}{EVC(e)} \notin N$. □

3.2. Complexity

We compare the vector clock with the EVC in time and space complexity. Each process only needs to store and transmit a single number EVC.

Theorem 3.2. *A vector clock with n entries and a maximum of α events in each entry has bit complexity $O(n \log_2 \alpha)$ whereas an EVC has bit complexity $\Omega(n \cdot \alpha)$.*

Proof. The bit complexity of the vector clock is self-evident.

The EVC uses an initial sequence of primes, one per process, and powers them to the event count at each process. We give a non-tight lower bound on the EVC size by using the same smallest prime (2) for each process. Clearly, $2^{v_1} * 2^{v_2} * \ldots * 2^{v_n} < 2^{v_1} * 3^{v_2} * \ldots * p_n^{v_n} = \prod_{i=1}^{n} p_i^{v_i}$. As

$$2^{v_1} * 2^{v_2} * \ldots * 2^{v_n} = 2^{\sum_{i=1}^{n} v_i} \leq 2^{n \cdot \alpha} < \prod_{i=1}^{n} p_i^{\alpha},$$

we can establish in general a lower bound for n processes and up to α events per process as $2^{n \cdot \alpha}$. Thus, EVC has a bit complexity $\Omega(n \cdot \alpha)$. □

Viewing Theorem 3.2 differently, for a system with n processes, if EVC is allowed to use $n \cdot 32$ bits, then 32 events across n processes each ($32n$ events) would fill the available $32n$ bits. Here, the $\Omega(n \cdot \alpha)$ bound acts as an upper bound $O(n \cdot \alpha)$ for the number of events that can be registered. In contrast, vector clocks with $32n$ bits can register up to 2^{32} events per process, so for a total of $n \cdot 2^{32} = 4294967296n$ events.

However, if we assume that the local space for storing and transmitting the EVC number is bounded, in conjunction with resetting the EVC, then the storage cost and message space overhead is $O(1)$ in the uniform cost model.

In general, we analyze the complexity of vector clocks and EVC assuming bounded storage using the uniform cost model, which is suitable for analysis when the numbers fit into a single machine word. We analyze the complexity of EVC assuming unbounded

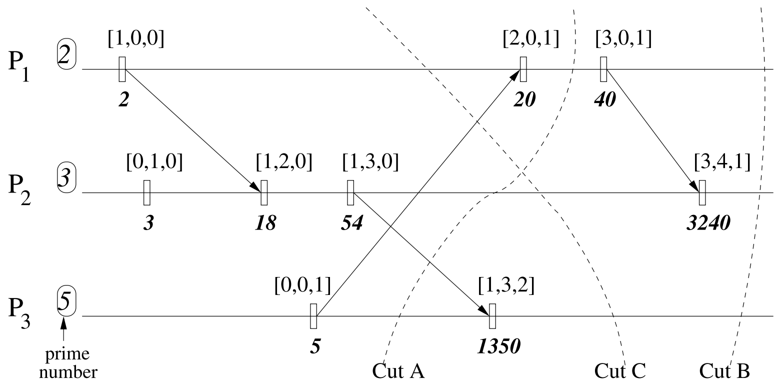

Fig. 2. Illustration of using EVC. The vector timestamps and EVC timestamps are shown above and below each timeline, respectively. In real scenarios, only the EVC is stored and transmitted.

storage using the logarithmic cost model, which assigns a cost to every machine operation that is a function of the number of bits involved, and is suitable when the numbers are unbounded. The logarithmic cost model is used to compute the bit complexity. For a EVC value H, we use h to denote the number of bits or digits in H. Thus, $h = \log H$. Note that H is at least $2^{n\cdot\alpha}$ and thus h is greater than $n \cdot \alpha$ and hence much higher than n.

3.2.1. Local tick

- **Unbounded numbers:** If we assume the EVC has unbounded storage, the bit complexity of multiplying two numbers of size H is $O(h^2)$ using naive multiplication. This becomes $O(h(\log h)(\log\log h))$ using the Schonhage–Strassen or other modern algorithms. However, for the local tick, the prime number that the EVC value H is being multiplied with is assumed to have bounded storage (fits in one machine word). Hence, the multiplication has bit time complexity $O(h)$ in the logarithmic cost model.
- **Bounded numbers:** If we assume the EVC has bounded storage, the multiplication of EVC with the prime number has $O(1)$ time complexity in the uniform cost model.

3.2.2. Merge or computing LCM

To compute $LCM(a, b)$, we have:

$$LCM(a, b) = \frac{a * b}{GCD(a, b)}$$

Here, GCD is the Greatest Common Divisor. By applying the Euclidean algorithm, we can compute $GCD(a, b)$ without factoring the two numbers. The time complexity is the number of steps in Euclid's algorithm, multiplied by the computational cost of each step. Let $\hbar$ be number of digits of the smaller number in base 10. (Note that $\hbar$ and h are of the same order.) It is well known that the number of steps required is never more than five times the number $\hbar$ [13,15,26].

- **Unbounded numbers:** In the logarithmic cost model, it is well known that the overall bit time complexity of the Euclidean algorithm for GCD is $O(h^2)$ [7,17] by employing the Euclid algorithm together with a classical mod operation. This can be reduced using recursive reduction techniques, and be brought down to $O(M(h)\log h)$, where $M(h)$ is the bit complexity of multiplication of two h-bit integers [7]. The best-known bound for $M(h)$ is $O(h(\log h)(\log\log h))$, as derived from modern transform and convolution techniques based on the Schonhage–Strassen or other algorithms for fast large integer multiplication [7,30]. Thus, the complexity of the recursive GCD algorithms is:

  $O(h(\log^2 h)(\log\log h))$

  This results in quasilinear algorithms for the GCD and LCM.
- **Bounded numbers:** In the uniform cost model, each step of the Euclid algorithm takes constant time. The total running time for GCD is $O(\hbar)$; this can be expressed as $O(1)$ because $\hbar$ is bounded by the word size. So we can compute GCD and LCM in $O(1)$ time.

3.2.3. Compare

- **Unbounded numbers:** The complexity of a mod operation modulo H is asymptotically the same as a size-H multiply. The time to check $Env(V_2) \bmod Enc(V_1) = 0$ is the same time taken to multiply two large numbers of h bits. The best-known bit complexity can be calculated using the Schonhage–Strassen bound, as $O(h(\log h)(\log\log h))$ [7].
- **Bounded numbers:** In the uniform cost model, the time complexity to check $Env(V_2) \bmod Enc(V_1) = 0$ is $O(1)$.

3.2.4. Storage

- **Unbounded numbers:** The storage complexity is $O(h)$ in the logarithmic cost model.
- **Bounded numbers:** In the uniform cost model, the storage complexity is $O(1)$. The only drawback for assuming a bounded space for storing the numbers is that eventually it will overflow. When overflow happens, we can adapt the vector clock resetting techniques [2,44] or the EVC resetting technique [32] which enable us to reuse the smaller numbers. The clock resetting algorithm [44] will incur an $O(c)$ message count complexity and an $O(d)$ storage cost at each process.

In Table 2, we compare the time complexity of the three basic operations (local tick, merge, compare), and the storage cost, for vector clock and EVC. Note that the logarithmic cost model for computing the complexities for the unbounded EVC storage case is different from the uniform cost model used to compute the complexities for the bounded EVC storage case and for vector clocks.

3.3. Resilience to churn

Churn refers to the dynamic joining and departure of processes. EVCs (and the operations on them) can operate correctly without any change and without any overhead in the face of churn, because the prime number of each process is independent of the others, and the EVC timestamp of an event also encodes its causal history into a single number. Optimizations, such as reducing the EVC values by a factor corresponding to the component of the departed process, require an engineered solution.

In comparison, vector clocks can also handle churn but may require some adaptation of the basic protocol and/or the operations and incur a corresponding overhead. In the simple approach,

Table 2
Comparison of the time complexity of the three basic operations and the space complexity, for vector clock and EVC.

	Vector clock (bounded storage) (uniform cost model)	Encoded vector clock (unbounded storage) (logarithmic cost model)	Encoded vector clock (bounded storage) (uniform cost model)
Local tick	$O(1)$	$O(h)$	$O(1)$
Merge	$O(n)$	$O(h(\log^2 h)(\log\log h))$	$O(1)$
Compare	$O(n)$	$O(h(\log h)(\log\log h))$	$O(1)$
Storage	$O(n)$	$O(h)$	$O(1) + O(d)$ (with resetting)

when a process joins, the vector size is increased and when a process departs, the vector size is not reduced [10]. Approaches that reduce the vector size when a process departs incur a change to the protocol [34,40].

4. EVC timestamps of cuts

4.1. Cuts

A cut is a prefix of the execution $(E, \rightarrow)$ and the state after the events of a cut represents a global state [4]. A downward closed prefix of $(E, \rightarrow)$ represents a consistent global state, and is a meaningful observable state of the execution [4]. The set of consistent cuts $CCuts$ forms a lattice $(CCuts, \subset)$ under the set inclusion relation [28]. Vector timestamps are assigned to cuts in order to reason with cuts [18,28].

Let $\downarrow e = \{f \mid f \rightarrow e \bigwedge f \in E\} \cup \{e\}$ denote the causal history of event e. $\downarrow e$ is a consistent cut. In general, the union of the causal histories of any subset X of events is a consistent cut. Thus, $cut(X) = \bigcup_{e \in X} \downarrow e$ is consistent even if the events in X form a cut that is not consistent.

The surface of a cut $S(cut)$ is the set that contains the last event of the cut cut at each process. Formally, $S(cut) = \{e_i^k \mid e_i^k \in cut \bigwedge e_i^{k+1} \notin cut\}$. For a cut cut, we define $\widehat{cut} = \bigcup_{e_i \in S(cut)} \downarrow e_i$ to be the smallest consistent cut that is larger than or equal to the cut cut. If cut is consistent, then $cut = \widehat{cut}$, whereas if cut is not consistent, then $cut \subset \widehat{cut}$.

4.2. EVC timestamp of a cut

Vector timestamp of a cut cut, $V(cut)$, is defined as

$$\begin{aligned}\forall k \in [1, n],\ V(cut)[k] &= V(e_k)[k], \text{ for } e_k \in S(\widehat{cut}) \\ &= \max_{e_i \in S(cut)} V(e_i)[k]\end{aligned}$$

Let event $e_i \in S(cut)$ occur at process P_i and let the vector timestamp of e_i, $V(e_i) = [v_1^i, v_2^i, \ldots, v_n^i]$. Likewise, let event $\hat{e}_i \in S(\widehat{cut})$ occur at process P_i and let the vector timestamp of $\hat{e}_i$, $V(\hat{e}_i) = [\hat{v}_1^i, \hat{v}_2^i, \ldots \hat{v}_n^i]$. We can then observe that

$$\begin{aligned}Enc(V(cut)) &= \prod_{i=1}^{n} p_i^{\hat{v}_i^i} \\ &= \prod_{i=1}^{n} p_i^{\max(v_i^1, v_i^2, \ldots, v_i^n)}\end{aligned}$$

To compute $Enc(V(cut))$, we do not have access to the individual components of vector timestamps of events in $S(cut)$. Moreover, it would be better to combine $Enc(V(e_1))$, $Enc(V(e_2))$, …, $Enc(V(e_n))$ into $Enc(V(cut))$ without knowing the n prime numbers. Using the definition of LCM in terms of the Fundamental Theorem of Arithmetic, we have the following claim.

Claim 2.

$$Enc(V(cut)) = LCM(Enc(V(e_1)), Enc(V(e_2)), \ldots, Enc(V(e_n))).$$

So, by computing the LCM of n EVC timestamps of events, the encoded timestamp of the consistent cut can be computed without knowing the n prime numbers. The LCM of n numbers can be computed iteratively, and its complexity is $n-1$ times the complexity of a single LCM. By extending the results of Section 3.2.2, the time complexity of computing the LCM of n EVC timestamps is $O(n \times h(\log^2 h)(\log\log h))$ assuming unbounded storage for EVCs and $O(n)$ assuming bounded storage for EVCs.

Example 1. For Cut A in Fig. 2, for events $e_i \in S(CutA)$, $V(e_1) = [2, 0, 1]$, $V(e_2) = [1, 3, 0]$, and $V(e_3) = [0, 0, 1]$. We have $V(CutA) = [2, 3, 1]$.

- Using prime numbers,
  $Enc(V(CutA)) = 2^{\max(2,1,0)} \times 3^{\max(0,3,0)} \times 5^{\max(1,0,1)} = 4 \times 27 \times 5 = 540$.
- We have $Enc(V(e_1)) = 20$, $Enc(V(e_2)) = 54$, and $Enc(V(e_3)) = 5$.
  Without using prime numbers,
  $Enc(V(CutA)) = LCM(Enc(V(e_1)), Enc(V(e_2)), Enc(V(e_3))) = LCM(20, 54, 5) = 540$.

Thus, $Enc(V(CutA))$ is the same value with and without using the prime numbers.

4.3. EVC timestamp of cut representing common past

For a cut cut, we can define its common past $CP(cut)$ to be the execution prefix such that the prefix is in the causal history of every element in $S(cut)$. $CP(cut) = \bigcap_{e_i \in S(cut)} \downarrow e_i$ [18]. The common-past of a cut is useful for discarding obsolete information in distributed databases, checkpointing, and designing protocols for the replicated log and replicated dictionary problems [20,43]. We define the vector timestamp of $CP(cut)$, $V(CP(cut))$, as

$$\forall k \in [1, n],\ V(CP(cut))[k] = \min_{e_i \in S(cut)} V(e_i)[k]$$

As before, let event $e_i \in S(cut)$ occur at process P_i and let the vector timestamp of e_i, $V(e_i) = [v_1^i, v_2^i, \ldots, v_n^i]$. We can then observe that

$$Enc(V(CP(cut))) = \prod_{i=1}^{n} p_i^{\min(v_i^1, v_i^2, \ldots, v_i^n)}$$

To compute $Enc(V(CP(cut)))$, we do not have access to the individual components of vector timestamps of events in $S(cut)$. Moreover, it would be better to combine $Enc(V(e_1))$, $Enc(V(e_2))$, $\ldots Enc(V(e_n))$ into $Enc(V(CP(cut)))$ without knowing the n prime numbers. Using the definition of GCD in terms of the Fundamental Theorem of Arithmetic, we have the following claim.

Claim 3.

$$Enc(V(CP(cut))) = GCD(Enc(V(e_1)), Enc(V(e_2)), \ldots, Enc(V(e_n))).$$

So, by computing the GCD of n EVC timestamps of events, the encoded timestamp of the consistent cut can be computed without knowing the n prime numbers. The GCD of n numbers can be computed iteratively, and its complexity is $n-1$ times the complexity of a single GCD. By extending the results of Section 3.2.2, the time complexity of computing the GCD of n EVC timestamps is $O(n \times h(\log^2 h)(\log\log h))$ assuming unbounded storage for EVCs and $O(n)$ assuming bounded storage for EVCs.

For unbounded storage, an alternate bound for $GCD(a_1, \ldots, a_n)$, based on the Euclidean algorithm, also computes the GCD iteratively but counts the total number of division operations. The derivation leverages the fact that after each GCD calculation, the GCD reduces by a factor of at least two, (else if it remains the same, only one division is used). There are $\log_2 a_1$ (rather than n) terms in the series $\sum_{k=1}^{\log_2 a_1} \log a_{i_k}$. The total number of division operations is less than $O((\log a_1)(\log a_n))$ or simply $O(h^2)$. As each division costs $O(h(\log h)(\log\log h))$, this bound is better if

$$O(h^3(\log h)(\log\log h)) < O(nh(\log^2 h)(\log\log h))$$

which may not be true for large numbers a_i.

Example 2. For Cut B in Fig. 2, for events $e_i \in S(CutB)$, we have $V(e_1) = [3, 0, 1]$, $V(e_2) = [3, 4, 1]$, and $V(e_3) = [1, 3, 2]$. $V(CutB) = [3, 4, 2]$, whereas we have $V(CP(CutB)) = [1, 0, 1]$.

- Using prime numbers, $Enc(V(CP(CutB))) = 2^{\min(3,3,1)} \times 3^{\min(0,4,3)} \times 5^{\min(1,1,2)} = 2 \times 1 \times 5 = 10$.
- We have $Enc(V(e_1)) = 40$, $Enc(V(e_2)) = 3240$, and $Enc(V(e_3)) = 1350$.
  Without using prime numbers, $Enc(V(CP(CutB))) = GCD(Enc(V(e_1)), Enc(V(e_2)), Enc(V(e_3))) = GCD(40, 3240, 1350) = 10$.

Thus, $Enc(V(CP(CutB)))$ is the same value with and without using the prime numbers.

Matrix clocks, first defined by Wuu and Bernstein [43], use a $n \times n$ matrix M of clock values, where the $M[j, k]$th entry at P_i denotes P_i's knowledge of P_j's knowledge of the latest local clock value at P_k. Note that $M[j]$, the jth row of the matrix timestamp of an event e, corresponds to the vector timestamp of the event at P_j in the surface of the cut $\downarrow e$, denoted by $V((S(\downarrow e))_j)$, and this can be encoded by EVC as shown above. Thus, the matrix clock can be encoded as a vector of length n of EVCs. The common past of events $(S(\downarrow e))_j$, for all j, identifies the execution prefix that is known to all processes and thus can be discarded from the local log at event e.

Example 3. For the event e with EVC = 3240 in Fig. 2, $Enc(M(e)) = [40, 3240, 5]$. We have $V(\downarrow e) = [3, 4, 1]$, $V((S(\downarrow e))_1) = [3, 0, 1]$, $V((S(\downarrow e))_2) = [3, 4, 1]$, $V((S(\downarrow e))_3) = [0, 0, 1]$, whereas we have $V(CP(\downarrow e)) = [0, 0, 1]$. By applying a logic similar to Example 2, it follows that:

- Using prime numbers, $Enc(V(CP(\downarrow e))) = 2^{\min(3,3,0)} \times 3^{\min(0,4,0)} \times 5^{\min(1,1,1)} = 1 \times 1 \times 5 = 5$.
- $Enc(V((S(\downarrow e))_1)) = 40$, $Enc(V((S(\downarrow e))_2)) = 3240$, $Enc(V((S(\downarrow e))_3)) = 5$.
  Without using prime numbers, $Enc(V(CP(\downarrow e))) = GCD(40, 3240, 5) = 5$.

The EVC of the execution prefix that can be safely discarded is 5.

4.4. Other operations on cuts

Intersection and Union: For two vector clock timestamps of (consistent) cuts $cut1$ and $cut2$, let

$$V(cut1) = [v_1, v_2, \ldots, v_n] \text{ and } V(cut2) = [v'_1, v'_2, \ldots, v'_n]$$

We have that

$$V(cut1 \bigcap cut2) = [u_1, u_2, \ldots, u_n], \text{ where } u_i = \min(v_i, v'_i)$$
$$V(cut1 \bigcup cut2) = [u_1, u_2, \ldots, u_n], \text{ where } u_i = \max(v_i, v'_i)$$

The encodings of $V(cut1)$, $V(cut2)$, $V(cut1 \bigcap cut2)$, and $V(cut1 \bigcup cut2)$ are:

$$Enc(V(cut1)) = p_1^{v_1} * p_2^{v_2} * \cdots * p_n^{v_n}$$
$$Enc(V(cut2)) = p_1^{v'_1} * p_2^{v'_2} * \cdots * p_n^{v'_n}$$
$$Enc(V(cut1 \bigcap cut2)) = \prod_{i=1}^{n} p_i^{\min(v_i, v'_i)}$$
$$Enc(V(cut1 \bigcup cut2)) = \prod_{i=1}^{n} p_i^{\max(v_i, v'_i)}$$

To compute the encodings of the vector timestamps of the intersection and union cuts, we do not have access to the individual components of the vector timestamps of $cut1$ and $cut2$. Moreover, it would be better to compute $Enc(V(cut1 \bigcap cut2))$ and $Enc(V(cut1 \bigcup cut2))$ without knowing the n prime numbers. Using the definition of GCD and LCM in terms of the Fundamental Theorem of Arithmetic, we have the following claim.

Claim 4.

$$Enc(V(cut1 \bigcap cut2)) = GCD(Enc(V(cut1)), Enc(V(cut2)))$$
$$Enc(V(cut1 \bigcup cut2)) = LCM(Enc(V(cut1)), Enc(V(cut2)))$$

So, by computing the GCD and the LCM of two EVC timestamps of cuts, the EVC timestamps of the intersection and the union cuts, respectively, can be computed without knowing the n prime numbers. The time complexity is $O(1)$, namely the cost of a single GCD or LCM operation, assuming bounded storage, or $O(h(\log^2 h)(\log\log h))$ assuming unbounded storage to represent the EVCs. This assumes the encoded vector timestamps of $cut1$ and $cut2$ are available. These should be available since we are operating in the EVC domain.

Example 4. Consider the intersection and union of Cut A and Cut C shown in Fig. 2. Using vector timestamps of cuts and prime numbers, we have:

- $V(CutA) = [2, 3, 1]$; $V(CutC) = [1, 3, 2]$.
- $V(CutA \bigcap cutC) = [1, 3, 1]$; $V(CutA \bigcup CutC) = [2, 3, 2]$
- $Enc(V(CutA \bigcap CutC)) = 2^1 * 3^3 * 5^1 = 270$;
  $Enc(V(CutA \bigcup CutC)) = 2^2 * 3^3 * 5^2 = 2700$

Using the encodings of the vector timestamps of Cut A and Cut C and without using prime numbers, we have by using the expression from Section 4.2, $Enc(V(CutA)) = LCM(20, 54, 5) = 540$ and $Enc(V(CutC)) = LCM(2, 54, 1350) = 1350$. We then have:

- $Enc(V(CutA \bigcap CutC)) = GCD(Enc(V(CutA)), Enc(V(CutC))) = GCD(540, 1350) = 270$.
- $Enc(V(CutA \bigcup CutC)) = LCM(Enc(V(CutA)), Enc(V(CutC))) = LCM(540, 1350) = 2700$.

Table 3
Correspondence between operations on cuts using vector clocks and EVC.

Operation	Vector clock	Encoded vector clock
Cut	$\forall k \in [1, n]$, $V(cut)[k] = \max_{e_i \in S(cut)} V(e_i)[k]$ (cut may not be consistent) $\forall k \in [1, n]$, $V(cut)[k] = V(e_k)[k]$ for $e_k \in S(cut)$ (cut is consistent)	$Enc(V(cut)) = LCM(Enc(V(e_1)), Enc(V(e_2)), \ldots, Enc(V(e_n)))$, where $e_i \in S(cut)$
Common past	$\forall k \in [1, n]$, $V(CP(cut))[k] = \min_{e_i \in S(cut)} V(e_i)[k]$	$Enc(V(CP(cut))) = GCD(Enc(V(e_1)), Enc(V(e_2)), \ldots, Enc(V(e_n)))$, where $e_i \in S(cut)$
	If $V(cut1)[j] = v_j$ and $V(cut2)[j] = v'_j$,	For $Enc(V(cut1))$ and $Enc(V(cut2))$,
Intersection	$V(cut1 \bigcap cut2)[j] = \min(v_j, v'_j)$	$Enc(V) = GCD(Enc(V(cut1)), Enc(V(cut2)))$
Union	$V(cut1 \bigcup cut2)[j] = \max(v_j, v'_j)$	$Enc(V) = LCM(Enc(V(cut1)), Enc(V(cut2)))$
Compare	$V(cut1) < V(cut2)$: $\forall j \in [1, n]$, $V(cut1)[j] \le V(cut2)[j]$ and $\exists j$, $V(cut1)[j] < V(cut2)[j]$	$Enc(V(cut1)) \prec Enc(V(cut2))$: $Enc(V(cut1)) < Enc(V(cut2))$ and $Enc(V(cut2)) \bmod Enc(V(cut1)) = 0$

Table 4
Comparison of the time complexity of the operations on cuts using vector clocks and EVC.

	Vector clock (bounded storage) (uniform cost model)	Encoded vector clock (unbounded storage) (logarithmic cost model)	Encoded vector clock (bounded storage) (uniform cost model)
Computing timestamp	$O(n^2)$ (cut may not be consistent) $O(n)$ (cut is consistent)	$O(nh(\log^2 h)(\log\log h))$	$O(n)$
Computing common past	$O(n^2)$	$O(nh(\log^2 h)(\log\log h))$	$O(n)$
Intersection and union	$O(n)$	$O(h(\log^2 h)(\log\log h))$	$O(1)$
Compare	$O(n)$	$O(h(\log h)(\log\log h))$	$O(1)$

Thus, $Enc(V(CutA \bigcap CutC))$ is the same value with and without using the prime numbers. Likewise for $Enc(V(CutA \bigcup CutC))$.

Comparison: The comparison of two distinct consistent cuts $cut1$ and $cut2$ in $CCuts$ results in one of two outcomes: (i) $cut1 \subset cut2$ (or symmetrically, $cut2 \subset cut1$), or (ii) $cut1 \not\subset cut2$ and $cut2 \not\subset cut1$, i.e., $cut1 \parallel cut2$.

To compare two EVC timestamps of cuts $cut1$ and $cut2$, it is only necessary to test if $Enc(V(cut2)) \bmod Enc(V(cut1)) = 0$. Thus,

$$(i)\ Enc(V(cut1)) \prec Enc(V(cut2)) \text{ if } Enc(V(cut1)) < Enc(V(cut2)) \text{ and } Enc(V(cut2)) \bmod Enc(V(cut1)) = 0$$

$$(ii)\ Enc(V(cut1)) \parallel Enc(V(cut2)) \text{ if } Enc(V(cut1)) \nprec Enc(V(cut2)) \text{ and } Enc(V(cut2)) \nprec Enc(V(cut1))$$

For unbounded storage, the time to check $Env(V(cut2)) \bmod Enc(V(cut1)) = 0$ is asymptotically the same as the time taken to multiply two large numbers of h bits. The best-known bit complexity is based on the Schonhage–Strassen bound, as $O(h(\log h)(\log\log h))$. For bounded numbers, the time complexity to check $Env(V(cut2)) \bmod Enc(V(cut1)) = 0$ is $O(1)$.

The encoded vector clock timestamps of consistent cuts, denoted $(\mathcal{ENCV}^{cc}, \prec)$, is isomorphic to $(\mathcal{V}^{cc}, <)$, the vector clock timestamps of consistent cuts, and to $(CCuts, \subset)$.

Table 3 gives the correspondence between the operations on cuts using vector clocks and using EVC. In Table 4, we compare the time complexities between the operations on cuts using vector timestamps and using EVCs.

5. Simulation results

For n processes in the system and α_i events at each process P_i, the maximum EVC timestamp across all processes is $O(\prod_{i=1}^{n} p_i^{\alpha_i})$. This is because at each event (send, receive, or internal) at P_i, the EVC gets multiplied by p_i, and in addition, at receive events, an LCM computation over two EVCs may significantly increase the EVC. From this observation and analysis towards the start of Section 3.2 (Theorem 3.2), we can see that EVC timestamps grow very fast. We ran simulations to test the growth rate of EVCs. The simulations were done in Rust and used the GMP library. We simulated distributed executions with a random communication pattern. As parameters, we used the number of processes and the probability of send (versus internal) events. The destination of a message from a send event was chosen at random. We timestamped events using EVCs, and measured the size of the EVC in bits. We used the first n prime numbers for the n processes.

We define the *overflow process* to be that process which is earliest to have its EVC size exceed $32n$ bits. The size $32n$ was chosen for comparison because this is the constant size used by traditional vector clocks, assuming each integer in the vector clock is represented by 4 bytes.

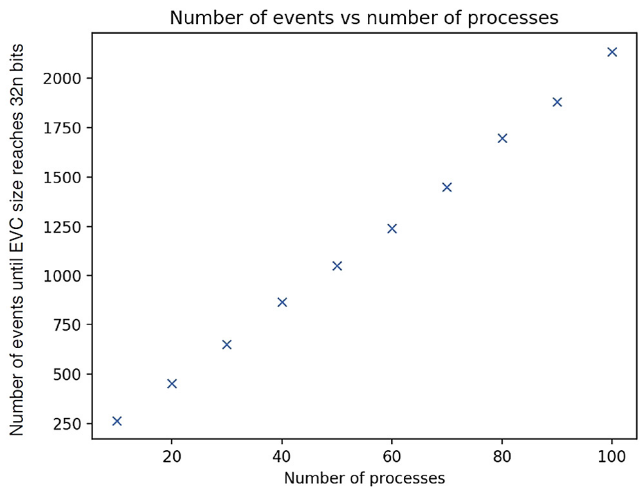

Fig. 3. Number of events needed for EVC to reach a size of $32n$ as a function of the number of processes n in the system.

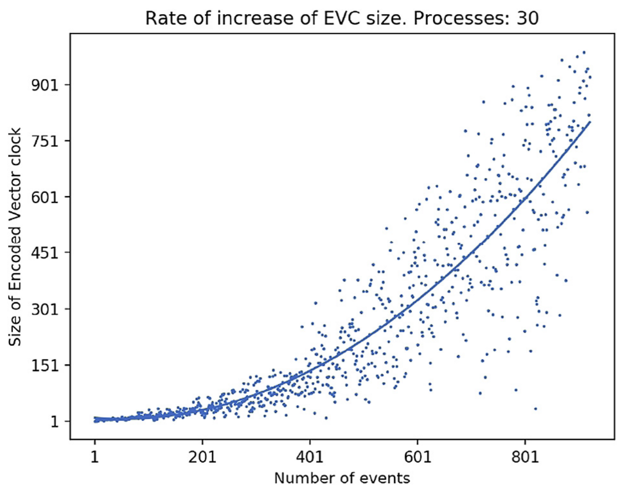

Fig. 4. Scatter-plot of the size of EVC in bits as a function of the number of events in the system. $n = 30$.

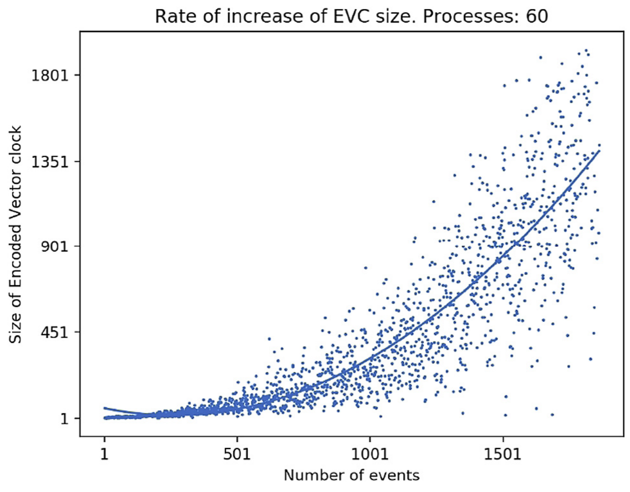

Fig. 5. Scatter-plot of the size of EVC in bits as a function of the number of events in the system. $n = 60$.

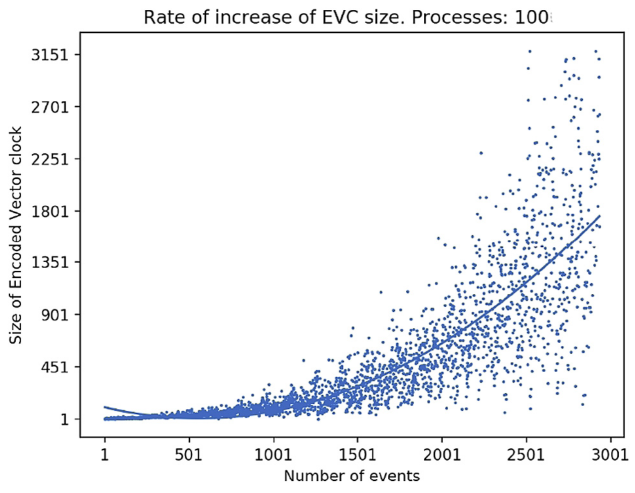

Fig. 6. Scatter-plot of the size of EVC in bits as a function of the number of events in the system. $n = 100$.

5.1. Number of events until EVC size becomes $32n$ as a function of n

Fig. 3 shows the number of events executed in the system until the EVC size reaches $32n$ bits at the overflow process, as a function of n. We varied n from 10 to 100, and plotted the average of 10 runs for each setting, assuming that pr_s, the probability of send events (versus internal events), was 0.6. The plot turns out to be almost a straight line.

For the range of n tested (10–100), typically 21 to 25 events were executed at some process before the EVC size exceeded $32n$ at the overflow process. As this number in the interval $[21, 25]$ appears small, we conduct a worst-case strawman analysis to show that this number is reasonable. As $pr_s = 0.6$, $probability(send\, event) = probability(receive\, event) = 0.6/1.6$. We can approximate this as assuming that every third event is a receive event. Now consider, for example, $n = 60$. The simulation uses the 60 lowest prime numbers, and a significant number of them need 8 bits for representation. At each event, we multiply t_i by p_i, so the size of the EVC increases by 8 bits. In addition, at every third event (a receive event), the size of the EVC can double in the worst case due to the LCM operation. (Doubling of the size of the EVC due to LCM computation is more likely in the initial part of the execution because the LCM is likely to be computed over relative prime numbers.) So the worst-case progression of the size of the EVC in bits at a process P_i can be approximated as:

$8, 16, 32$ and 40 (event e_i^3),
$48, 56, 112$ and 120 (event e_i^6),
$128, 136, 272$ and 280 (event e_i^9),
$288, 296, 592$ and 600 (event e_i^{12}),
$608, 616, 1232$ and 1240 (event e_i^{15}),
$1248, 1256, 2512$ and 2520 (event e_i^{18})

At the 18th event at P_i, the EVC size exceeds $60 \times 32 = 1920$ bits. As per the simulation, the overflow happens at the 1250/60th event, which is the 21st event, at the overflow process, so this worst-case analysis is reasonably accurate.

This analysis indicates that receive events cause the EVC to grow very fast due to the LCM computation.

5.2. Size of EVC as a function of number of events

In our next experiment, we measured the size of the EVC in bits as a function of the number of events executed in the

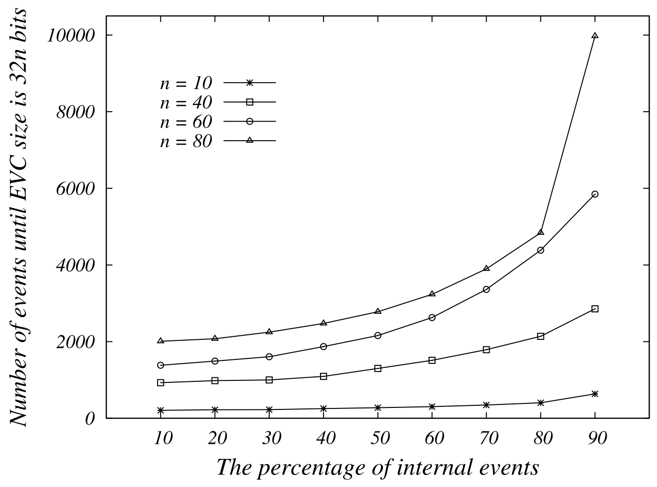

Fig. 7. Total number of events until EVC size reaches $32n$ bits, for different n and different percentages of internal events.

system. Figs. 4, 5, and 6, show the scatter-plots for a system with $n = 30, 60, 100$ processes, respectively. For these executions, pr_s, the probability of send event (versus internal event) was chosen as 0.5. In these plots, the number of events on the X-axis is such that the size of the EVC in bits is always less than that of the traditional vector clocks. The Y-axis shows the size of the EVC in bits until the size equals $32n$. The maximum size $32n$ was chosen because this is the constant size used by traditional vector clocks, assuming each integer in the vector clock is represented by 4 bytes.

Consider for example, Fig. 5, which uses parameters $n = 60$ and $pr_s = 0.5$. There were about 1800 events in the systemwide execution (or an average of $1800/60 = 30$ events at a process) until the EVC size reached 1920 ($= 60 \times 32$) bits at the overflow process.

5.3. *Number of events until EVC size becomes* 32*n* *as a function of ratio of event types*

We also varied the percentage of internal events (where the total number of events included send, receive, and internal events), and varied n, and observed the total number of events in the system until the EVC size reaches $32n$ bits at the overflow process. The observations are plotted in Fig. 7. For a given n, as the percentage of internal events increased, symbolizing an increasingly smaller proportion of receive events (and hence fewer LCM computations), the rate of increase of the total number of events until the EVC size reached $32n$ bits kept increasing. In particular, when $probability(internal\,event) > 0.8$, there was a more noticeable rate of increase of the total number of events (until the EVC size reached $32n$ bits at the overflow process). This shows that as the proportion of send events and corresponding receive events decreases progressively, particularly below 10%, due to the fewer resulting LCM computations at receive events, the EVC grows much less rapidly, thereby resulting in a much larger number of system events until the EVC size reaches $32n$ bits. This corroborates the earlier observation that receive events cause the EVC to grow very fast due to the LCM computation.

Consider, for example, the value for $n = 60$, $probability(internal\,event) = 0.9$ which implies that $probability(receive\,event) = 0.05$. We again conduct a strawman analysis. We assume the prime numbers take up to 8 bits representation. As before, let us assume each LCM computation causes the EVC size (in bits) to double because the execution has just begun and the LCM is likely to be computed over relative prime numbers. Then, the receive event occurs every 20 events, at which time the EVC size increases by a factor of 2. So the worst-case progression of the size of the EVC in bits at a process P_i can be approximated as:

$8, \ldots, 152, 304$ and 312 (event e_i^{20}),
$320, \ldots, 464, 928$ and 936 (event e_i^{40}),
$944, \ldots, 1088, 2176$ and 2184 (event e_i^{60})

At the 60th event at P_i, or equivalently at around the $60 \times 60 = 3600$th event in the execution, the EVC size exceeds $60 \times 32 = 1920$ bits. As per the simulation graph (Fig. 7), the overflow happens at around the 6000th event in the execution, and this can be justified by applying a correction to the worst-case strawman analysis. Note that in the simulation, there is a delay for the message transmission. Hence, in the small initial window (before steady state) that the results in Fig. 7 depict, actually $probability(receive\,event) < 0.05$ and hence there are more than 20 non-receive events per receive event. Hence, there are more than 60 events at the overflow process P_i and hence more than 3600 events in the system until overflow occurs. This supports the simulation result of 6000 events.

6. Scalability

As seen in Section 5, the EVC timestamps grow very fast and eventually they will exceed the size of vector clocks. However, we can use several strategies to alleviate this problem and control the maximum size of the EVC. In particular, these strategies can be used to guarantee that the EVC size is always less than the vector clock size.

6.1. *Relevant events*

It suffices if the local clock does not tick at every event but only at events that are relevant to the application. Thus, the EVC does not grow so fast. This strategy is explained in the context of predicate detection [37]. The local clock should tick only when the variables in the predicate alter the truth value of the predicate. As another example, the local clock ticks only at synchronization events in MPI application programs [8,9]; and at the synchronization events, viz., *lock*, *unlock*, *fork*, and *join* events, in dynamic race detection in multi-threaded environments [32]; see the case studies in Section 7.

6.2. *Detection regions*

In large-scale systems, the application requiring a vector clock may be confined to only a subset of m processes, where $m < n$. An example of this is locality-aware predicate detection [37]. The subset of m processes forms a detection region. Processes within the detection region maintain a single number for the EVC. More importantly, for processes outside the detection region, we can cut down the storage cost and make the solution more practical for large-scale systems. For a process P_j outside the region, when it first receives a message piggybacked with an EVC timestamp, it simply stores this single number. Although P_j will not tick the EVC locally since there is no corresponding component in the vector clock for P_j, it may still receive multiple messages. Each time this happens, P_j simply executes the merge operation by calculating the LCM of two numbers. (P_j needs to store the EVC and to do the merge because it may later send messages back into the detection region, directly or transitively.)

6.3. Resetting EVC

We can adapt the clock resetting technique [44] to solve the problem when the clock overflows. This technique divides the execution of a distributed system into multiple phases. Each time the clock overflows at any process, the resetting algorithm terminates the current phase by sending control messages while ensuring there is no computation message sending from the current phase to the next phase, nor from the next phase to the current phase. The reset protocol involves a period of send inhibition of messages, and the local clock gets reset in a strongly consistent (i.e., transitless) global state [1,19]. This technique introduces false causality dependencies when comparing timestamps across phases. To deal with this, the phase number needs to be maintained along with the EVC and an auxiliary function (to be adapted from [44]) needs to be used. Alternately, the idea of asynchronous and uncoordinated reset, as used in [2] for vector clocks and in [32] for EVC can be applied.

It is up to the application to determine when the EVC overflows. If we say that the clock overflows when the size of the EVC equals $32n$ bits at some process, then we can guarantee that the size of EVC is always less than that of traditional vector clocks.

We can also reset the EVCs globally when there is a naturally occurring global system state in which all previous events are ordered (as per the "happened before" relation $\rightarrow$) before all subsequent events. For example, such global synchronization occurs at a global barrier or fence instruction in MPI programs [14]; see the case study in Section 7.1. The system state immediately after a global synchronization is a transitless global state.

6.4. Using logarithms of EVC

As the EVC technique uses exponentiation, we propose the use of logarithms to store and transmit the EVCs. This can result in a significant reduction in the size of EVCs. We note that since logarithms involve finite-precision arithmetic, their use is subject to the introduction of errors due to the limited precision. In related work [35], a detailed analysis of the error rates introduced by the use of logarithms of concise version vectors was given. This analysis showed that the error rates are quite low, and can be further decreased by increasing the number of mantissa bits of the logarithms.

7. Case studies

We review two case studies of the use of EVC. The first case study is of detecting memory consistency errors in MPI one-sided applications using EVC [8,9]. The second case study is of the resettable EVC (REVC) with an application to dynamic race detection in shared memory multi-threaded environments [32].

7.1. Detecting memory consistency errors in MPI one-sided applications

7.1.1. MPI one-sided communication

MPI one-sided communication, also known as MPI remote memory access, does not require sends to be matched with corresponding receive instructions [14]. Only one process takes part in the data movement (using unilateral instructions such as `MPI_Put` and `MPI_Get` rather than matching pairs of `MPI_Send` and `MPI_Receive`). It decouples data transfer between processes from synchronization between the processes. This eliminates overhead from unneeded synchronization and allows for greater concurrency. This also eliminates message matching and buffering overheads that are incurred in traditional two-sided communication, leading to significant reduction in communication costs. These advantages of one-sided communication come at a cost — the programs are more prone to synchronization bugs, such as memory consistency errors.

7.1.2. Memory consistency errors in MPI one-sided communication

In simple terms, a memory consistency error is a write to a location (through a local store instruction or through a remotely issued `MPI_Put`) that is concurrent with another write or a read (through a local load instruction or a locally or remotely issued `MPI_Get`) to the same memory location at the same process [6]. We elaborate on "is concurrent with" semi-formally. The $\xrightarrow{hb}$ "happened before" relation between events a and b is the transitive closure of the union of the program order and synchronization order. The program order at a process specifies that a previous instruction is executed before a later instruction. The synchronization order across processes orders events by the order in which synchronization instructions are executed (e.g., `MPI_Send` at a source process completes before `MPI_Receive` at the destination process). The consistency order $\xrightarrow{co}$ on events a and b guarantees that the memory effects of a are visible before b [14]. This order is necessary because synchronization instructions such as `MPI_Win_lock/unlock` order memory accesses but do not synchronize processes. For example, if a is nonblocking, and a and b both access overlapping buffers, there is no consistency order because of a potential race condition due to a being nonblocking. Now, the $\xrightarrow{cohb}$ relation on events is the transitive closure of the intersection of the $\xrightarrow{co}$ and $\xrightarrow{hb}$ relations [14]. If the $\xrightarrow{cohb}$ does not hold between a pair of events, that pair of events is concurrent under $\xrightarrow{cohb}$. Thus, two memory operations are concurrent if there are no $\xrightarrow{co}$ and $\xrightarrow{hb}$ between them. If there are two concurrent events accessing the same memory location and at least one of them is an update operation (whether local or remote), then there is a memory consistency error in an MPI one-sided program execution. Note that a memory consistency error may be of two types: either within an epoch at the same process, or across processes.

Although MPI one-sided communication calls may cause memory consistency errors with other such calls or load/store operations, not every pair of operations will cause such errors. This is because MPI applications use synchronization calls (such as `MPI_Barrier` and `MPI_Win_fence`) to enforce $\xrightarrow{co}$ and/or $\xrightarrow{hb}$ between two operations. Only when two operations fall within a concurrent program region may memory consistency errors arise. A *concurrent program region* is defined as a group of program regions across multiple (all) processes, that can be executed concurrently without $\xrightarrow{co}$ and $\xrightarrow{hb}$ ordering relations, i.e., program regions that are not ordered by $\xrightarrow{cohb}$ [6]. Each *program region* is formed of one or multiple epochs, where an *epoch* is formed by a pair of one-sided synchronization calls.

7.1.3. The MC-Checker tool

MC-Checker [6] is a tool for identifying memory consistency errors. Using trace files, it generates a dynamic data access DAG whose nodes are the events and edges represent the "happened before" relation. The DAG represents a set of concurrent regions. A concurrent region begins and ends with a global synchronization operation (such as `MPI_Barrier` and `MPI_Win_fence`). In the general case, the set of concurrent regions forms a partial order. However, the set of concurrent regions is totally ordered, assuming a single MPI communicator. Each concurrent region is modeled as a graph: the set of nodes are the events in MPI one-sided programs and the edges are the $\xrightarrow{cohb}$ relation. Each concurrent region is (independently) analyzed to detect memory consistency errors — such an error exists between each pair of conflicting operations that are not ordered by $\xrightarrow{cohb}$. MC-Checker detects conflicting operations within each epoch of a program region, and across processes within the concurrent region.

Typically, two-sided communication is used along with one-sided communication in high-performance computing applications. In order to detect memory consistency errors, transitive dependencies between processes, such as those induced by send and receive operations by several different processes, need to be captured. MC-Checker [6] suffers the drawback that it does not take into account such transitive dependencies, because capturing such dependencies would require building a complete DAG of dependencies between events for analysis, which would require maintaining vector clocks. However, vector clocks do not scale and they impose high overheads; as a result MC-Checker did not use vector clocks and this led to the introduction of false positives in reporting memory consistency errors.

7.1.4. The MC-CChecker tool using EVC

The MC-CChecker tool [8,9] overcame this drawback by using the EVC, thereby eliminating the false positives reported by MC-Checker while still maintaining low overheads. As the $\xrightarrow{cohb}$ relation is specified only on synchronization events within and across processes (these are the relevant events), the EVC scheme also needs to timestamp only such events. MC-CChecker adapted the EVC rules of Fig. 1 [21] to MPI one-sided communication system as follows [8,9].

R1 For two consecutive synchronization events, if $e_i^x \xrightarrow{cohb} e_i^{x+1}$, then $t_i^{x+1} = t_i^x * p_i$.

R2. If e_i^x is `fence` (or `barrier`) and e_j^y is the corresponding `fence` (or `barrier`), then a message m from e_i^x to e_j^y is timestamped $tm = t_i^x$. On receipt at P_j, $t_j^y = LCM(tm, t_j^y)$.

R3/R4/R5. If e_i^x is `post`/ `complete`/ `send` and e_j^y is the corresponding `start`/ `wait`/ `receive`, then a message m from e_i^x to e_j^y is timestamped $tm = t_i^x$; and then a local tick is executed at P_i. On receipt at P_j, $t_j^y = LCM(tm, t_j^y)$.

For simplicity, it is assumed that `post` $\xrightarrow{cohb}$ `start` and `complete` $\xrightarrow{cohb}$ `wait`. Only synchronization operations are timestamped as the goal is to represent an area (termed as a *separate region*) formed between two consecutive synchronization operations, including the former but excluding the latter; the timestamps of all events within the separate region equal the timestamp of the representing (former) synchronization event's timestamp.

Along the lines of the test in [21], $e_i^x \xrightarrow{cohb} e_j^y$ if and only if t_i^x divides t_j^y. The two events are concurrent under $\xrightarrow{cohb}$ if and only if the EVC timestamp of neither event divides that of the other. MC-CChecker considers concurrent regions like MC-Checker, but using EVC timestamped information built after analyzing the trace files. MC-CChecker loads concurrent regions one by one from trace files. Once MC-CChecker loads one concurrent region, it detects memory consistency errors within each epoch similar to MC-Checker. However, for errors across processes, it examines the concurrency of each pair of separate regions for each concurrent region. If two separate regions are executed concurrently, MC-CChecker checks the accessed memory of each pair of operations belonging to the two separate regions to flag memory consistency errors (if the two operations are concurrent under $\xrightarrow{cohb}$, conflict, and access the same location).

7.1.5. Performance benefits of MC-CChecker using EVC

Experiments [8,9] run on HPC platforms using three different MPI applications showed that MC-CChecker used low processing time and memory usage, when checked for up to 128 processes. The scalability study compared MC-CChecker using EVC and using traditional vector clocks, for systems ranging from 512 up to 8192 processes. The study showed that with EVC, execution time and memory usage are linear (with respect to n), whereas with traditional vector clocks, both execution time and memory usage were significantly higher and increased in much larger proportion.

7.1.6. Analysis and summing up proof of concept

In this case study, the relevant events were the synchronization events; only these were timestamped by MC-CChecker using EVCs. Further, each concurrent region contained a program region from a different process. All the concurrent regions were totally ordered, assuming a single MPI communicator. (Without this assumption, the concurrent regions form a partial order.) The boundary between two adjacent concurrent regions was implemented by global synchronization calls such as `MPI_Barrier` and `MPI_Win_fence`. The start of each concurrent region corresponded to a global synchronization where there was no concurrency between events in the previous concurrent region and in the following one. Each concurrent region was a unit of computation [1,19], and the boundary between two adjacent/consecutive concurrent regions corresponded to a global transitless state. MC-CChecker safely reset the EVC of each process to 1 at the start of each concurrent region. Using the combination of these two scalability techniques, viz., tick at relevant event, and reset at the start of each concurrent region, the size of the EVCs at the processes remained small and grew linearly (with n), as the MC-CChecker scalability study showed.

7.2. Dynamic race detection in multi-threaded environments

7.2.1. Resettable Encoded Vector Clock (REVC)

Pozzetti proposed and formalized the concept of *Resettable Encoded Vector Clock* (REVC), a logical clock implementation, which builds on the EVC to tackle its very high growth rate (and, under given conditions, place an upper bound on its storage requirements,) while maintaining the desirable properties of the EVC [32]. REVC can be applied in both shared memory systems and message passing systems to achieve a consistent logical clock. The advantage of REVC's growth rate with respect to EVC's growth rate was shown through practical examples. Then, a practical application of the REVC to the dynamic race detection problem in multi-threaded environments using the RoadRunner [12] dynamic analysis framework was shown. The tool built was compared to the currently existing vector clock based tool DJIT$^+$ [31] to show how the REVC can help in achieving higher performance with respect to the vector clock.

The core idea of the REVC is that of performing a reset operation at an EVC location every time such value overflows a predefined number of bits at that location. The reset operation brings the EVC value back to the initial one, allowing the system to continue its operations until the following overflow event. The REVC exploits asynchronous local resets at each process. Each time a local reset occurs, a new local *frame* is generated. Experiments using several benchmark programs from benchmark suites [3,16] showed that the growth rate of the REVC is linear with respect to the total number of events in the system (as opposed to the exponential growth rate of the EVC). The REVC framework also provides optimizations that have a bounded memory solution for causality analysis, i.e., the number of frames stored is bounded. The formulation of the REVC contains several intrinsic trade-offs between space, time, and accuracy 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 different scenarios, which cannot be found in other vector clock implementations.

7.2.2. Evaluation

The evaluation of the REVC with respect to the traditional vector clock was shown for the dynamic race detection problem in multi-threaded environments [32]. Specifically, a modified version of the DJIT$^+$ protocol that exploits the REVC instead of the vector clock to track causality relations among events was built. The solution was analyzed and evaluated in terms of performance, by comparing it to the other tools that have already been developed for dynamic race detection. The tool built was implemented as a backend tool in the dynamic analysis framework RoadRunner [12]. In order to be able to provide a fair evaluation of proposed system with respect to the traditional DJIT$^+$ protocol, an implementation of DJIT$^+$ on the same framework was also developed. FastTrack [11], on the other hand, was already implemented on top of RoadRunner, as it has been studied and developed by the same team of Flanagan and Freund.

The applications that were chosen for evaluation were a subset of the applications that are found in the DaCapo Benchmarking Suite [3] and the Java Grande Benchmarking Suite [16]. 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 were chosen exhibit multi-threaded behavior 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. Results showed that the REVC-based application is able to outperform by up to 1.6 times the traditional vector clock 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. (FastTrack 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.) However, this result is a promising achievement as it allows us to show that the EVC can have practical applications, which are competitive for scenarios in which no alternatives to a traditional vector clock implementation are available.

7.2.3. Analysis and summing up proof of concept

This case study used the two scalability techniques: ticking at relevant events, and resetting the EVC. The relevant events were the synchronization events, viz., the *lock*, *unlock*, *fork*, and *join* events, in the shared memory multi-threaded environment. Resetting the EVC was done asynchronously and locally using the formalism developed [32]. Using a combination of these two scalability techniques, the size of the REVC remained small, and the REVC-based application was able to outperform the traditional vector clock based DJIT$^+$ protocol by up to 1.6 times.

8. Related work

In the field of distributed systems, Raynal used prime numbers as a tool for designing distributed algorithms [33]. Specifically, he proposed a termination detection algorithm and a mutual exclusion algorithm using the properties of prime numbers. Shen et al. [37] previously used the encoding of vector clocks using prime numbers to detect locality-aware conjunctive predicates in large-scale systems.

Singhal and Kshemkalyani [38] proposed a mechanism to reduce the size of the vector timestamp piggybacked on messages sent over FIFO channels. The key idea is to transmit only incremental changes to previously transmitted timestamp components. This message overhead reduction is achieved by having each process maintain two arrays of size n integers to track the incremental changes to be sent to each potential destination. Meldal et al. [29] consider applications where it is important to determine the causality between two messages sent to a common destination process. In this context, they achieve a reduction in the size of timestamps because they do not need to capture the causality or the happened-before relation between all possible pairs of events. Their algorithms exploit information about the paths over which messages may be propagated.

Torres-Rojas and Ahamad [39] proposed a class of logical clocks, called plausible clocks, that can be implemented with a constant number of components. Under certain circumstances, determined by factors such as the number of sites in the system, communications patterns, size of the global history, level of concurrency in the system, and frequency of communications, these plausible clocks provide ordering accuracy close to that of vector clocks. Several implementations of constant size plausible clocks are presented. REV is a variant of vector clocks where R-sized vectors are used. As $R < n$, several entries are shared by more than one site of the distributed system and therefore a mapping between sites and entries in the vector needs to be defined. KLA is an extension of Lamport clocks where each site keeps a standard Lamport clock together with a collection of the maximum timestamp of any message received by itself and by the $K-2$ previous sites that directly or indirectly have had communications with this site. The authors also develop rules to compose known plausible clocks to produce more accurate clocks. The accuracy levels of the proposed instances of constant-sized plausible clocks are tested using simulations, and shown to be relatively high. However, false positives occur (i.e., the mechanism sometimes orders events that are mutually concurrent) even if there are no false negatives (i.e., the mechanism always orders events that are related by causality).

Ward and Taylor [41] proposed a timestamping mechanism that is defined assuming the processes are organized as a hierarchy of clusters. They maintain two types of timestamps per event: one, 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. The latter type of timestamps is a recognition of the fact that events within a cluster can only be causally dependent on events outside the cluster through receive events from send events that occurred outside the cluster. Such receive events are called "cluster-receive" events. In the hierarchical scheme, the key idea is to maintain a set of cluster-receive events that have timestamps only to the next level in the hierarchy. This 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 uses static, pre-determined clusters. An extension algorithm that allows a dynamic, self-organizing selection of clusters is then presented [42].

Kulkarni et al. [24] proposed a clock scheme that combines physical clocks (such as NTP) with logical clocks. Their clock scheme deals only with weak causality, i.e., $e \to f \Rightarrow C(e) < C(f)$. Their scheme is meant for applications where only (weak) causal relations in the recent immediate past are relevant, and assumes that (weak) causal relations beyond a configurable parameter ϵ are already reflected.

Kulkarni and Vaidya [25] reduced the size of the vector timestamp by exploiting the underlying logical topology (assuming, not all processes communicate to each other) and deferring the assignment of a timestamp to an event for a suitably determined duration of time. They showed that for a graph with vertex cover VC, it is possible to assign timestamps containing only $2|VC|+2$ integer elements. In particular, assuming α events per process, they showed that the size of a timestamp for any event is at most $\log_2 n + (2|VC|+1)\log_2(\alpha+1)$ bits.

9. Conclusions

We proposed the encoding of the vector clock using prime numbers, to use a single number to represent vector time. We gave the operations on the EVC. To manipulate the EVC, every process only needs to know its own prime, and not the primes of other processes. Further, to compute the equivalent of the maximum of two vector clocks, a process needs to find the largest common multiple of their EVCs, which does not require factorization. We also showed how to timestamp global states using EVC, and various operations – namely, common causal past computation, union, intersection, and comparison – on these global states using EVC.

A serious drawback of EVCs is that they grow very fast and overflow, i.e., exceeding the space used by traditional vector clocks soon occurs. We examined using a theoretical analysis and using simulations how fast the EVC grows. We then proposed four scalability approaches for the EVC to deal with the high growth rate of the EVC. These included (i) ticking the clock only at application-relevant events and only at processes where such events occur, and (ii) resetting the EVC throughout the system at a global synchronization or at a transitless global state, or resetting the EVC locally and independently (the REVC), when it overflows at some process. A judicious use of these scalability approaches can guarantee that the size of the EVC never exceeds the size of the traditional vector clock. We considered two case studies of using EVC. The first case study was for detecting memory consistency errors in MPI applications that use one-sided communication. Using the combination of two scalability approaches, viz., ticking at relevant event, and resetting at the start of each concurrent region, the size of the EVCs at the processes remained small, grew linearly, and was significantly much less than that using traditional vector clocks. The second case study was for dynamic race detection in multi-threaded environments. Using the combination of two scalability approaches, viz., ticking at relevant event, and resetting the EVC locally and asynchronously when an overflow occurred locally, the size of the EVCs at the processes remained small, and the time overhead was less than that using traditional vector clocks of the DJIT$^+$ tool.

In summary, while EVC is mathematically elegant, it does not appear to offer a general purpose practical replacement of vector clocks due to its very high growth rate. However in conjunction with the scalability techniques, the EVC shows much potential. There are application areas, such as those discussed in the case studies, where the EVC offers a definite practical advantage. These results show how the EVC 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. In conjunction with resetting, the EVC is seen to be designed with scalability and adaptability to different scenarios, which cannot be found in other vector clock implementations. We believe these achievements are promising and 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.

As future work, it is a challenge to identify other such application areas where EVC outperforms vector clock. The EVC timestamps in the first case study were assigned after analyzing the program traces. It would be interesting to determine whether they can be assigned in an on-line manner efficiently. Another direction for further work is to examine the length of the causal chain of relevant events/messages in social platforms (e.g., Twitter and Facebook) where the number of users is potentially large. This could provide evidence as to whether or not EVCs are advantageous on social platforms.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2020.02.008.

CRediT authorship contribution statement

Ajay D. Kshemkalyani: Conceptualization, Formal analysis, Investigation, Methodology, Supervision. **Min Shen:** Conceptualization. **Bhargav Voleti:** Software, Validation, Visualization.

Acknowledgments

We thank Bhaskar Dasgupta for suggesting the alternate bound on computing the GCD of n numbers in the unbounded storage case. Thanks also to Rahul Sathe for his help with the simulations.

References

[1] M. Ahuja, A.D. Kshemkalyani, T. Carlson, A basic unit of computation in distributed systems, in: 10th International Conference on Distributed Computing Systems, ICDCS 1990, May 28–June 1, 1990, Paris, France, 1990, pp. 12–19, http://dx.doi.org/10.1109/ICDCS.1990.89327.

[2] A. Arora, S.S. Kulkarni, M. Demirbas, Resettable vector clocks, J. Parallel Distrib. Comput. 66 (2) (2006) 221–237, http://dx.doi.org/10.1016/j.jpdc.2005.07.001.

[3] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The dacapo benchmarks: Java benchmarking development and analysis, SIGPLAN Not. 41 (10) (2006) 169–190, http://dx.doi.org/10.1145/1167515.1167488, URL http://doi.acm.org/10.1145/1167515.1167488.

[4] K.M. Chandy, L. Lamport, Distributed snapshots: Determining global states of distributed systems, ACM Trans. Comput. Syst. 3 (1) (1985) 63–75, http://dx.doi.org/10.1145/214451.214456, URL http://doi.acm.org/10.1145/214451.214456.

[5] B. Charron-Bost, Concerning the size of logical clocks in distributed systems, Inf. Process. Lett. 39 (1) (1991) 11–16, http://dx.doi.org/10.1016/0020-0190(91)90055-M.

[6] Z. Chen, J. Dinan, Z. Tang, P. Balaji, H. Zhong, J. Wei, T. Huang, F. Qin, MC-Checker: Detecting memory consistency errors in MPI one-sided applications, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16–21, 2014, 2014, pp. 499–510, http://dx.doi.org/10.1109/SC.2014.46.

[7] R. Crandall, C. Pomerance, Prime Numbers: A Computational Perspective, first ed., Springer, New York, NY, USA, 2001.

[8] T.-D. Diep, K. Fürlinger, N. Thoai, MC-CChecker: A clock-based approach to detect memory consistency errors in MPI one-sided applications, in: Proceedings of the 25th European MPI Users' Group Meeting, EuroMPI'18, ACM, New York, NY, USA, 2018, pp. 9:1–9:11, http://dx.doi.org/10.1145/3236367.3236369, URL http://doi.acm.org/10.1145/3236367.3236369.

[9] T. Diep, K.T. Pham, K. Fürlinger, N. Thoai, A time-stamping system to detect memory consistency errors in MPI one-sided applications, Parallel Comput. 86 (2019) 36–44, http://dx.doi.org/10.1016/j.parco.2019.04.013.

[10] C.J. Fidge, Logical time in distributed computing systems, IEEE Comput. 24 (8) (1991) 28–33, http://dx.doi.org/10.1109/2.84874.

[11] C. Flanagan, S.N. Freund, Fasttrack: efficient and precise dynamic race detection, in: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15–21, 2009, 2009, pp. 121–133, http://dx.doi.org/10.1145/1542476.1542490, URL http://doi.acm.org/10.1145/1542476.1542490.

[12] C. Flanagan, S.N. Freund, The roadrunner dynamic analysis framework for concurrent programs, in: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, ACM, New York, NY, USA, 2010, pp. 1–8, http://dx.doi.org/10.1145/1806672.1806674, URL http://doi.acm.org/10.1145/1806672.1806674.

[13] H. Grossman, On the number of divisions in finding a G.C.D., Amer. Math. Monthly 31 (9) (1924) 443.

[14] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K.D. Underwood, Remote memory access programming in MPI-3, TOPC 2 (2) (2015) 9:1–9:26, http://dx.doi.org/10.1145/2780584, URL http://doi.acm.org/10.1145/2780584.

[15] R. Honsberger, Mathematical Gems II, The Mathematical Association of America, 1976.

[16] Java Grande Forum, Java Grande benchmarking suite, 2008, http://www.javagrande.org/, [Online]. (Accessed 31 July 2019).

[17] D.E. Knuth, The Art of Computer Programming, Vol. 2, third ed., in: Seminumerical Algorithms, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[18] A.D. Kshemkalyani, Causality and atomicity in distributed computations, Distrib. Comput. 11 (4) (1998) 169–189, http://dx.doi.org/10.1007/s004460050048.

[19] A.D. Kshemkalyani, A framework for viewing atomic events in distributed computations, Theoret. Comput. Sci. 196 (1–2) (1998) 45–70, http://dx.doi.org/10.1016/S0304-3975(97)00195-3.

[20] A.D. Kshemkalyani, The power of logical clock abstractions, Distrib. Comput. 17 (2) (2004) 131–150, http://dx.doi.org/10.1007/s00446-003-0105-9.

[21] A.D. Kshemkalyani, A.A. Khokhar, M. Shen, Encoded vector clock: Using primes to characterize causality in distributed systems, in: Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4–7, 2018, 2018, pp. 12:1–12:8, http://dx.doi.org/10.1145/3154273.3154305, URL http://doi.acm.org/10.1145/3154273.3154305.

[22] A.D. Kshemkalyani, M. Singhal, Distributed Computing: Principles, Algorithms, and Systems, Cambridge University Press, 2011.

[23] A.D. Kshemkalyani, B. Voleti, On the growth of the prime numbers based encoded vector clock, in: Distributed Computing and Internet Technology - 15th International Conference, ICDCIT 2019, Bhubaneswar, India, January 10–13, 2019, Proceedings, 2019, pp. 169–184, http://dx.doi.org/10.1007/978-3-030-05366-6_14.

[24] S.S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, M. Leone, Logical physical clocks, in: Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina D'Ampezzo, Italy, December 16–19, 2014. Proceedings, 2014, pp. 17–32, http://dx.doi.org/10.1007/978-3-319-14472-6_2.

[25] S.S. Kulkarni, N.H. Vaidya, Effectiveness of delaying timestamp computation, in: Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25–27, 2017, 2017, pp. 263–272, http://dx.doi.org/10.1145/3087801.3087818, URL http://doi.acm.org/10.1145/3087801.3087818.

[26] G. Lame, Note sur la limite du nombre des divisions dans la recherche du plus grand commun diviseur entre deux nombres entiers, C. R. Acad. Sci. 19 (1844) 867–870.

[27] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.

[28] F. Mattern, Virtual time and global states of distributed systems, in: Proceedings of the Parallel and Distributed Algorithms Conference, 1988, pp. 215–226.

[29] S. Meldal, S. Sankar, J. Vera, Exploiting locality in maintaining potential causality, in: Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, PODC '91, ACM, New York, NY, USA, 1991, pp. 231–239, http://dx.doi.org/10.1145/112600.112620, URL http://doi.acm.org/10.1145/112600.112620.

[30] N. Moller, On Schonhage's algorithm and subquadratic integer gcd computation, Math. Comp. 77 (261) (2008) 589–607.

[31] E. Pozniansky, A. Schuster, MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs, Concurr. Comput.: Pract. Exper. 19 (3) (2007) 327–340, http://dx.doi.org/10.1002/cpe.1064.

[32] T. Pozzetti, Resettable Encoded Vector Clock for Causality Analysis with an Application to Dynamic Race Detection (M.S. thesis), University of Illinois at Chicago, 2019.

[33] M. Raynal, Prime numbers as a tool to design distributed algorithms, Inf. Process. Lett. 33 (1) (1989) 53–58, http://dx.doi.org/10.1016/0020-0190(89)90187-7.

[34] G.G. RichardI.I.I., Efficient vector time with dynamic process creation and termination, J. Parallel Distrib. Comput. 55 (1) (1998) 109–120, http://dx.doi.org/10.1006/jpdc.1998.1493.

[35] H. Roh, M. Jeon, E. Seo, J. Kim, J. Lee, Log' version vector: Logging version vectors concisely in dynamic replication, Inf. Process. Lett. 110 (14–15) (2010) 614–620, http://dx.doi.org/10.1016/j.ipl.2010.04.026.

[36] R. Schwarz, F. Mattern, Detecting causal relationships in distributed computations: In search of the holy grail, Distrib. Comput. 7 (3) (1994) 149–174, http://dx.doi.org/10.1007/BF02277859.

[37] M. Shen, A.D. Kshemkalyani, A.A. Khokhar, Detecting unstable conjunctive locality-aware predicates in large-scale systems, in: IEEE 12th International Symposium on Parallel and Distributed Computing, ISPDC 2013, Bucharest, Romania, June 27–30, 2013, 2013, pp. 127–134, http://dx.doi.org/10.1109/ISPDC.2013.25.

[38] M. Singhal, A.D. Kshemkalyani, An efficient implementation of vector clocks, Inf. Process. Lett. 43 (1) (1992) 47–52, http://dx.doi.org/10.1016/0020-0190(92)90028-T.

[39] F.J. Torres-Rojas, M. Ahamad, Plausible clocks: Constant size logical clocks for distributed systems, Distrib. Comput. 12 (4) (1999) 179–195, http://dx.doi.org/10.1007/s004460050065.

[40] X. Wang, J. Mayo, W. Gao, J. Slusser, An efficient implementation of vector clocks in dynamic systems, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2006, Las Vegas, Nevada, USA, June 26–29, 2006, Vol. 2, 2006, pp. 593–599.

[41] P.A.S. Ward, D.J. Taylor, A hierarchical cluster algorithm for dynamic, centralized timestamps, in: Proceedings of the 21st International Conference on Distributed Computing Systems, ICDCS 2001, Phoenix, Arizona, USA, April 16–19, 2001, 2001, pp. 585–593, http://dx.doi.org/10.1109/ICDSC.2001.918989.

[42] P.A.S. Ward, D.J. Taylor, Self-organizing hierarchical cluster timestamps, in: Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28–31, 2001, Proceedings, 2001, pp. 46–56, http://dx.doi.org/10.1007/3-540-44681-8_8.

[43] G.T. Wuu, A.J. Bernstein, Efficient solutions to the replicated log and dictionary problems, in: Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, PODC '84, ACM, New York, NY, USA, 1984, pp. 233–242, http://dx.doi.org/10.1145/800222.806750, URL http://doi.acm.org/10.1145/800222.806750.

[44] L. Yen, T. Huang, Resetting vector clocks in distributed systems, J. Parallel Distrib. Comput. 43 (1) (1997) 15–20, http://dx.doi.org/10.1006/jpdc.1997.1330.

Ajay D. Kshemkalyani received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987, and the MS and Ph.D. degrees in computer and information science from The Ohio State University 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 in the Department of Computer Science at the University of Illinois at Chicago. His research interests are in distributed computing, distributed algorithms, computer networks, and concurrent systems. In 1999, he received the US National Science Foundation Career Award. He has served on the editorial board of the Elsevier journal *Computer Networks*, and the *IEEE Transactions on Parallel and Distributed Systems*. He has coauthored a book entitled *Distributed Computing: Principles, Algorithms, and Systems* (Cambridge University Press, 2008). He is a distinguished scientist of the ACM and a senior member of the IEEE.

Min Shen received the B.S. degree in computer science from Nanjing University in 2009, and the Ph.D. degree in computer science from University of Illinois at Chicago in 2014. His research interests include distributed algorithms, predicate detection and wireless sensor networks. He is currently working at LinkedIn on various aspects of Hadoop ecosystem.

Bhargav Voleti received his M.S. in computer science from the University of Illinois at Chicago in 2019. His research interests include distributed systems, operating systems and high-performance concurrent systems. He is currently working at Amazon on scalable action frameworks.