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

BY

TOMMASO POZZETTI
B.S., Computer Engineering, Politecnico di Torino, Torino, Italy, 2017

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2019

Chicago, Illinois

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, professor Kshemkalyani, for his guidance and support throughout this research work and the time he has dedicated to helping me achieve this result. Also, I would like to thank professor Sisto for following my work as my home university's advisor and my home university itself, Politecnico di Torino, for giving me this incredible opportunity to finish my studies in Chicago through this exchange program.

A special thanks goes to my mom, my dad, my sister and my brother, my grandparents and all of my family for their constant love and support throughout all the years of study and the many life lessons. I would never be where I am today without them, and I will be grateful for the rest of my life.

I would also like to thank my girlfriend Rosita, for supporting me through these experiences, even if it meant being apart for long periods of time. You have always been there for me and have given me the strength to be the best version of myself, even during the difficult times.

Last, but certainly not least, I would like to thank all of the friends from Torino's and Milano's 2018/2019 Chicago experience that have shared with me this incredible journey, making it the wonderful experience that it has been. In particular, I would like to express my gratitude and love for the people that have become my closest family this year: Alessio, Arturo and Riccardo, and also our upstairs neighbors Gabriele and Federico. Thank you for making me feel at home when we were 7000 km away from home.

TP

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

DMT	Differential Merge Technique
EVC	Encoded Vector Clock
FSFW	Fixed Size Frame Window
OOM	Out Of Memory
REVC	Resettable Encoded Vector Clock
VC	Vector Clock

SUMMARY

Causality tracking among events is a fundamental challenge in distributed environments. Most previous work on this subject has been focused on designing an efficient and scalable protocol to represent logical time. Several implementations of logical clocks have been proposed, most recently the Encoded Vector Clock (EVC), a protocol to encode Vector Clocks in scalar numbers through the use of prime numbers, to improve performance and scalability.

In this thesis work we propose and formalize the concept of **Resettable Encoded Vector Clock** (REVC), a new logical clock implementation, which builds on the EVC to tackle its very high growth rate issue. We show how our REVC can be applied in both shared memory systems and message passing systems to achieve a consistent logical clock. We show, through practical examples, a comparison between REVC's and EVC's growth rates, to demonstrate our clock's advantages. Finally, we show a practical application of the REVC to the Dynamic Race Detection problem in multi-threaded environments. We compare our tool to the currently existing VC-based tool DJIT$^+$ to show how the REVC can help in achieving higher performance with respect to the Vector Clock.

CHAPTER 1

INTRODUCTION

1.1 Background and motivation

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

Several implementations of logical clocks have been proposed, such as *Scalar Clock* [1], *Vector Clock* (VC) [2] [3] and *Encoded Vector Clock* (EVC) [4]. The first is a lightweight, simple and efficient system to track causality through the use of scalars, but it does not allow to obtain the causality relation from the timestamps (no **strong consistency** property). The

VC, on the other hand, is the commonly employed technique to implement a logical clock which can achieve strong consistency. Its main drawback, however, is its poor scalability as it is dependent on the number of processes for all its operations and storage requirements. Finally, the EVC is a technique to represent a vector clock using a single scalar number, to improve scalability of the VC while maintaining strong consistency and it is based on the use of prime numbers to achieve such encoding. The main drawback of such representation has been shown to be the extremely high growth rate which quickly causes an overflow at the locations that are storing the EVC values [5].

1.2 Contributions

In this thesis work, we propose and formalize the concept of **Resettable Encoded Vector Clock** (REVC), a new logical clock implementation, which builds on the EVC to reduce its growth rate and, under certain conditions, place an upper bound on its storage requirements, while maintaining the scalability and performance properties of the EVC. We show how our REVC can be applied in both shared memory systems and message passing systems to achieve a consistent logical clock. We show, through practical examples, a comparison between REVC's and EVC's growth rates, to demonstrate our clock's advantages.

Finally, we show a practical application of the REVC to the Dynamic Race Condition Detection problem in multi-threaded environments using the RoadRunner [6] dynamic analysis framework. We compare our tool to the currently existing VC-based tool DJIT$^+$ [7] to show how the REVC can help in achieving higher performance with respect to the VC.

1.3 Document's structure

The next chapter, *Chapter 2*, details previous and related work that has been carried out on logical clocks and on dynamic race detection. *Chapter 3* presents the models for the shared memory systems and message passing systems on which we apply our Resettable Encoded Vector Clock, which is detailed in *Chapter 4*. *Chapter 5* shows a practical application of the REVC to the Dynamic Race Detection problem and its results. Finally, *Chapter 6* concludes our work and presents ideas for future developments.

CHAPTER 2

RELATED WORK

In this thesis work we present the *Resettable Encoded Vector Clock*, which is a logical clock implementation built on top of the *Encoded Vector Clock* [4]. The EVC itself is an encoding of the traditional Vector Clock [2] [3] using prime numbers.

We also present an application of the REVC to Dynamic Race Detection, which we implement using the dynamic analysis framework RoadRunner [6]. Finally, we compare our REVC tool for Dynamic Race Detection to a VC-based protocol, DJIT$^+$ [7], and to FastTrack [8], an adaptive tool that can dynamically switch between a scalar clock and a vector clock.

2.1 Vector Clock

The Vector Clock is a logical clock developed independently by Fidge [2] and Mattern [3]. Differently from the Scalar Clock presented by Lamport [1], the VC achieves the property of strong consistency, at the expense of scalability and performance. The Vector Clock is based on keeping at each process an array of size n, where n is the number of processes in the system. The i^{th} component of such array at process x denotes the value of the scalar clock of process i as known by process x at that time.

The Vector Clock defines an implementation of the three basic operations of a logical clock, the *local tick* operation, the *merge* operation and the *comparison* operation.

When performing a local tick operation, process x will execute the following instruction (assuming V is x's VC):

$$V[x] = V[x] + 1 \tag{2.1}$$

When merging Vector Clock V_1 into Vector Clock V_2, the process executes:

$$\forall i : V_2[i] = \max(V_1[i], V_2[i]) \tag{2.2}$$

Finally, the purpose of any logical clock is that of inducing a partial order on the set of events labeled with a timestamp according to Lamport's *happens-before* relation [1] (represented with the operator $\rightarrow$). To be able to test any two events a and b according to such relation, the comparison operation is defined:

$$a \rightarrow b \iff (\forall i : V_a[i] \leq V_b[i]) \wedge (\exists i : V_a[i] < V_b[i]) \tag{2.3}$$

The main drawback of the Vector Clock is its impact on scalability and performance. Given that both the *merge* operation and the *comparison* operation are dependent on n, along with the data structure used to store the VC, as the number of processes increases, the overhead increases with it.

2.2 Encoded Vector Clock

Kshemkalyani et al. proposed the use of prime numbers to encode the values of the Vector Clock in one single scalar number [4], which allows for higher scalability while maintaining strong consistency. The EVC is based on assigning each process a unique prime number p, which is the initialization value of its local clock. The three basic operation are then modified as follows.

When process x, which has been assigned prime number p, performs a local tick operation, it executes (assuming E is x's EVC):

$$E = E * p \tag{2.4}$$

In order to merge EVC E_1 into EVC E_2, the following is executed:

$$E_2 = \mathrm{lcm}(E_1, E_2) \tag{2.5}$$

Finally, to establish the relationship between two events a and b labeled with EVCs E_a and E_b, the comparison operation is defined:

$$a \to b \iff E_a < E_b \wedge E_b \bmod E_a == 0 \tag{2.6}$$

The Encoded Vector Clock achieves a higher scalability than the Vector Clock as it is not dependent on the number of processes and all its operations have a theoretical complexity of $O(1)$. The main drawback of the EVC is its extremely high growth rate [5]. By relying on multiplications and *least common multiple* operations, EVC values grow with an exponential rate, which can quickly require a very large number of bits in order to be able to represent it, and therefore store it.

Despite this drawback, the EVC has found practical applications, such as detecting memory consistency errors in MPI one-sided applications [9], due to the use of resets at global barrier synchronizations.

2.3 Dynamic Race Detection

One of the most common types of errors that is found in parallel shared-memory environments is the so called **race condition**. A race condition is present when two or more threads try to make use of the same resource without properly synchronizing among themselves, and are therefore in a *race* to access the resource. Clearly, if at least one of the threads is trying to modify the resource, errors can arise depending on the relative speed among the threads, and therefore depending on who *wins* the race at any given execution.

Several protocols for race condition detection have been studied, which aim at reporting such bugs to the developer to allow for an easier debugging. There are static tools (examples include [10], [11], [12]) which perform source code analysis or dynamic tools (examples include [13], [7], [14]) which track a given execution to determine the presence of races. Those tools can be further divided in tools that report probable races (and can include false positives) and tools that report only actual races that are present in the software. **DJIT**$^{+}$ [7] is a dynamic race condition detection protocol that exploits vector clocks to track causality among accesses to memory locations and therefore precisely analyze a program execution and report actual races. Given that it is a dynamic protocol, it runs during the actual program execution and as such it provides some overhead to the execution itself. In order for such protocol to be usable in practice, the overhead should be reduced as much as possible, and therefore performance is key.

2.3.1 The RoadRunner Framework

Dynamic race condition detection protocols execute alongside the application that is being tested for race conditions, and track the interesting events (memory accesses, synchronization instructions, thread creation and termination instructions) in order to store and retrieve information related to such events that can be useful to identify races. In order to achieve such results, a framework needs to be deployed that is capable of interacting to some extent with the running program and implement the dynamic protocol.

In our evaluation, we exploited the Java framework RoadRunner, developed by Flanagan and Freund [6]. RoadRunner is a dynamic analysis framework which is designed to help in the implementation of tools that perform dynamic analysis of concurrent Java programs. It is composed of two main parts: a Java bytecode instrumenter and a backend which can be extended to implement any dynamic analysis tool.

The first component's task, as the name suggests, is to instrument the Java bytecode of the application which is being analyzed. Upon class loading, the system dynamically modifies the bytecode to include callbacks and extra variables that are used to produce an event stream that will then be intercepted and processed by one or more backend tools. In practice, every time the application under scrutiny executes an action which is considered an *interesting* event by the framework, a callback is issued to notify the backend tool of the execution of such event. A payload of information regarding the event is also attached to the callback, to allow the backend to correctly understand the context of the event and the actors that are participating. Among the set of *interesting* events that are tracked by the system are the synchronization operations provided by the Java programming language, **acquire** and **release** of lock objects, **fork**, **start**, **join**, **wait**, **sleep**, **notify** and access events to class methods, class fields, static fields, arrays and volatile variables. For a subset of such events, two callbacks are available, one firing before the event is executed and one after the event is executed, to allow the backend tool maximum flexibility on the management of such events.

The second component, as mentioned, is the backend tool which implements the dynamic analysis protocol that is being deployed. RoadRunner defines a basic abstract class that contains all the possible callbacks to handle the event stream and the implementation of a backend tool starts with the creation of a new class that extends the basic abstract class and implements all, or a subset, of the callbacks to handle the events that are of interest to the protocol. Once the events are fired, the callbacks defined by the tool are called and complete flexibility is given to the tool to implement any behaviour that is required.

Each event, when fired, carries with it a payload of information, which are specialized based on the event type. For example, among the information conveyed to the tool for a *fork* event, are the identifiers of both the thread that executes the fork operation and the new thread that is created, while for an *acquire* operation, the identifier of the executing thread and the identifier of the lock object being acquired are reported.

Furthermore, RoadRunner associates special data structures to each thread that is created, lock object that is used and memory location that is accessed. Such data structures have the goal of storing additional information that can be retrieved every time an event is fired concering such objects, by accessing special methods from the callbacks. Those data structures can be further customized by the backend tool to store any additional information that it requires in order to achieve the desired results.

Finally, given that memory access operations constitute the majority of operations among the tracked events, RoadRunner offers the possibility of creating specialized *fast path* implementations of *read* and *write* operations, by defining two special methods in the tool class. Such fast implementations are optimized with respect to the usual memory access event, because they are inlined by the instrumenter directly inside the application's bytecode, thus skipping the overhead of event creation and dispatch.

2.3.2 DJIT$^+$

DJIT$^+$ is a dynamic race condition detection protocol developed by Pozniansky and Schuster [7], which employs vector clocks to track causality relations among access operations performed on a shared memory location, to identify races among threads during the program execution and report them.

Each thread maintains a vector clock from the beginning of the execution, and uses it to label each access to a memory location. Each synchronization object (for simplicity we consider here only simple locks, with no loss of generality as any other form of more complex synchronization can be implemented on top of such basic primitives) is also attached a vector clock. Each clock is advanced and modified following the basic operations of a multi-threaded environment on a shared memory model as briefly described by the following rules:

- **Lock acquisition**: when a thread acquires a lock object, it **merges** the vector clock of the lock into its own vector clock
- **Lock release**: when a thread releases a lock object, it **merges** its own clock into the lock's vector clock. Then, the thread performs a **local tick** operation to create a new fresh timestamp value to be used for following memory accesses
- **Fork operation**: when a new thread t_2 is created by thread t_1, t_1 sets the vector clock of t_2 equal to its own vector clock, and then it performs a **local tick** operation
- **Join operation**: when thread t_2 terminates and joins with thread t_1, t_1 **merges** t_2's clock into its own

When the program performs an access to a memory location, the previous vector clock timestamp that was stored at that memory location is checked against the current thread's vector clock and, based on the two operations and their relationship according to the *happened-before* operator, a possible race condition might be detected and reported.

More precisely, each memory location actually stores two vector clocks vr and vw which represent the latest read and write operations. Supposing that t is the current thread's vector clock, upon a memory access, races can be detected by the following rules:

- **Read-Write race**: if the current operation is a write operation at the accessed memory location and it is not true that $vr \rightarrow t$, then a **Read-Write** race is detected and reported, as the current write operation is concurrent with the previous read operation

- **Write-Read race**: if the current operation is a read operation at the accessed memory location and it is not true that $vw \rightarrow t$, then a **Write-Read** race is detected and reported, as the current read operation is concurrent with the previous write operation
- **Write-Write race**: if the current operation is a write operation at the accessed memory location and it is not true that $vw \rightarrow t$, then a **Write-Write** race is detected and reported, as the current write operation is concurrent with the previous write operation

Clearly Read-Read races do not need to be reported as they don't really constitute a *race condition*, as multiple concurrent reads are allowed at any given memory location without the risk of returning inconsistent values.

The authors of the DJIT$^+$ protocol show proof that this system is precise to the point of never reporting false positives and being able to detect any race condition that happens during a given program execution. However, given that DJIT$^+$ is a dynamic race condition detection system, it is possible that not all races that are present are reported as some race condition might happen only due to a specific relative speed among threads which doesn't happen in the actual program executions in which the protocol is employed.

2.3.3 FastTrack

FastTrack [8] is the state-of-the-art Dynamic Race Detection protocol. It builds on DJIT$^+$ employing several optimization techniques that allow it to reduce the information that is stored for each memory location from a Vector Clock to a Scalar Clock for most operations. Only when the timestamp that is generated by using the Scalar Clock is not sufficient to correctly characterize the accesses, it dynamically switches to the traditional Vector Clock representation.

These optimization techniques allow FastTrack to achieve a very high speedup over DJIT$^+$. Furthermore, the storage requirements for the application are consistently reduced, as most instances do not require the storage of a full Vector Clock.

CHAPTER 3

SYSTEM MODEL

Distributed systems are usually divided into two categories: **shared memory** systems and **message passing** systems [15]. As the names suggest, this division is based on the type of communication that the different components of the distributed system use to synchronize. We will now briefly analyze each of the two models along with the types of events that make it up and the rules that ensure a consistent relationship among such events.

3.1 Shared Memory System

A shared memory system is composed of several threads that execute concurrently and communicate via shared memory. The execution is assumed to be asynchronous, therefore the relative speed among the threads is not fixed. The threads can synchronize using the operating system provided operations *lock*, *unlock*, *fork* and *join*. Other more complicated means of synchronization are not analyzed here, but the model can be easily extended to include them by building on these basic primitives.

The execution of each thread consists in a series of events that can be either one of the operations previously presented or what here is called an *internal event*. Internal events can be any type of action that is performed by a thread, which does not require any synchronization with other threads, and can therefore be seen as an event local to the thread itself. These actions may include (but are not limited to) read and write operations to arbitrary memory locations, function calls or compound statements composed of several operations.

A partial order on this set of actions can be induced by Lamport's *happened-before* relation [1], which can characterize causality among such events. In this context, such relation is characterized by the following rules:

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

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

3.2 Message Passing System

A message passing system is composed of several processes that execute concurrently and communicate via message passing. We assume the presence of a logical communication channel among each pair of processes. We do not require that such channels be physically present in the system, nor do we make any assumption on the nature of such channels, such as requiring them to be FIFO. The execution is again assumed to be asynchronous, therefore no fixed relative speed among the processes exists. Each process can synchronize with the others by means of the communication primitives **send** and **receive**. The **fence** operation is also available, which allows to create barrier synchronizations among multiple processes.

The execution can again be characterized as a sequence of events. In message passing systems, four types of events can be distinguished: *send* events, *receive* events, *fence* events and *internal* events. The first three correspond to the execution of the basic communication primitives that have just been described, while an internal event can be any other operation, or compound statement composed of multiple operations, which does not require any synchronization with other processes.

Lamport's *happened-before* relation [1] can be used to induce a partial order on this set of events to characterize causality among them. In message passing systems the following rules apply:

- **Program order**: if a and b are two events that are executed by the same process, and a is executed before b, then $a \rightarrow b$

- **Message synchronization**: if s is a *send* operation for message m and r is the receive operation for the same message m, then $s \to r$
- **Barrier synchronization**: if f_i and f_j are two corresponding *fence* operations executed by two processes and e is an event executed by any of the two processes before reaching the barrier, then $e \to f_i \wedge e \to f_j$. Also if e' is an event executed by any of the two processes after executing the *fence* operation, then $f_i \to e' \wedge f_j \to e'$
- **Transitive property**: if $a \to b$ and $b \to c$, then $a \to c$

Finally, concurrent events can be characterized following the same definition that was used for shared memory systems, i.e. events a and b are said to be concurrent if and only if $\neg(a \to b) \wedge \neg(b \to a)$.

CHAPTER 4

THE RESETTABLE ENCODED VECTOR CLOCK

The first contribution of this thesis work, as previously described, is to detail the components and rules that make up the Resettable Encoded Vector Clock, and analyze its applicability both in shared memory systems and message passing systems. Finally, we also compare the REVC protocol with the EVC protocol to study the benefits in terms of growth rate and storage requirements that it is able to obtain.

The Resettable Encoded Vector Clock is a logical clock implementation which builds on top of the Encoded Vector Clock with the aim of tackling its high growth issue. The idea that the protocol is built on is that of performing a reset operation at the EVC location every time such value overflows a predefined number of bits n. The reset operation brings the EVC value back to the initialization value, allowing the system to restart its operations until the following overflow event.

One possible approach to such reset is to require that, when the first process overflows the available number of bits for storing the EVC, the system is *paused* and all processes synchronize to execute a local reset of their EVC. A practical way to achieve such result is to use a modified version of Lamport's global snapshot algorithm [16], which requires processes to exchange control messages to achieve a global synchronization point. Such technique has been studied and implemented as a protocol for resetting Vector Clocks by Yen and Huang [17].
However, an important consideration is that, for the REVC to be usable in place of the EVC, it

should be equivalent to the EVC, and thus return the same results as the EVC would. Therefore, any solution that proposes to reset the EVC value by means of a global synchronization point, not only introduces a high overhead into the system caused by the added synchronization, but also breaks the equivalence among EVC and REVC. We can in fact show that, if a synchronization event e is added to the system to perform a global reset, the underlying causal order is changed. Let's take two concurrent events a and b and let's suppose that, in the current execution of the program, after executing a but before executing b, an overflow occurs and therefore a reset is required. Event e is now executed, as the global synchronization event in which all processes perform the local reset. Given that e is a global synchronization event executed by all processes, the intrinsic program ordering states that $a \to e$. After the reset, the system can restore normal operations and event b is now executed. Again, given that e is a global synchronization event, it follows that $e \to b$. But because of the transitive property of Lamport's happened-before relation, if $a \to e$ and $e \to b$, we can state that $a \to b$ which is in contrast with the result that would be obtained by the EVC, in which a and b are two concurrent events.

We have showed with this example how a solution that exploits added synchronization among the processes introduces external events in the normal program execution, which change the causality relation in the set of all events. Furthermore, let it be noted that such an approach would in practice introduce a considerable overhead into the system as each overflow event would require a global pause of the system to execute the reset protocol.

Therefore, the solution that is proposed in the Resettable Encoded Vector Clock exploits asynchronous local resets at each process, an idea that was previously explored in the context of Vector Clocks by Arora et al. [18]. Thus, we avoid both introducing the overhead of a global synchronization event with the corresponding loss of performance, and breaking the equivalence among EVC and REVC. Clearly, however, this approach presents a number of challenges. As an example, let us consider a scenario in which, during an execution, given processes p_1 and p_2, p_1 has performed a local reset as a consequence of an overflow while p_2 is still able to fit its EVC value in n bits. Furthermore, let us assume that, before resetting, p_1 had executed event e_1 and p_2 had executed event e_2 in such a way as to establish the relationship $e_2 \rightarrow e_1$. Because of the local reset happened at p_1, p_1 will now label a new event x that it executes, with a new fresh timestamp, which therefore does not reflect the relationship $e_2 \rightarrow x$. However, because of the intrinsic *program order*, we can state that $e_1 \rightarrow x$, as x is executed after e_1. Furthermore, because of the transitive property, given that $e_2 \rightarrow e_1$ and $e_1 \rightarrow x$, we can state that $e_2 \rightarrow x$, which is in contrast with the result that is obtained by comparing the REVC timestamps of the two events.

Therefore, the local asynchronous reset is not enough to guarantee the equivalence between REVC and EVC. The Resettable Encoded Vector Clock will need additional components and rules to ensure that a timestamp that is produced by such protocol can account for the scenario that has been presented, returning a consistent result when investigating the relationship among events. The following sections will detail all the components that are needed to achieve such result, along with the rules that guarantee the REVC consistency when applied to a shared memory system or a message passing system.

4.1 Components of the REVC

The Resettable Encoded Vector Clock is built on top of the Encoded Vector Clock, adding the necessary rules to allow for resets that avoid unbounded growth of the EVC itself. We have however showed how adding an asynchronous local reset protocol is not sufficient to achieve a consistent logical clock that can substitute the EVC. The REVC is therefore composed of additional components.

Firstly, we introduce the concept of **frame**. The *frame* is a counter that keeps track of the number of local resets that have been performed at that process. Each REVC instance contains an integer variable, which defines the current *frame* of such REVC. By using the *frame*, a more precise comparison among two REVC timestamps is possible. In fact, it is now easy to understand whether one of the two is directly comparable with the other with the standard EVC comparison operation, by checking whether they are part of the same execution frame.

The second component that we introduce is the **frame history**. Each REVC instance needs to be comparable with any other REVC instance, irrespectively of whether the other instance is in the same execution frame or not. In order to achieve that, each REVC instance needs to keep track of the EVC values of previous frames before the local resets. The *frame history* component, therefore, achieves this objective by storing a structure that can map a given frame to a given EVC value.

Based on the components that we have detailed, we can represent an instance of the Resettable Encoded Vector Clock as the following tuple:

$$(f, e, m) \tag{4.1}$$

where f is the current frame, e is the current EVC value and m the map containing the frame history.

Finally, given that the REVC is built on the Encoded Vector Clock, each process needs to be assigned a unique prime number that is used by the EVC to encode the timestamp of the process itself.

We have detailed in section 2.2 the three basic operations characterizing the EVC: the *local tick* operation, the *merge* operation and the *comparison* operation. In the following sections, we analyze and detail the three corresponding operations for the Resettable Encoded Vector Clock.

4.1.1 Local tick operation

The first operation that is analyzed is the *local tick* operation. This operation is used to increment the local clock of a process to create a new fresh timestamp value that can be assigned to label any new event. This operation, performed on the standard EVC, requires to multiply its value by the process' unique prime number. If a certain threshold of n bits has been defined to represent the EVC value, clearly such operation can cause the current value to overflow the given threshold.

In the Resettable Encoded Vector Clock, the overflow event constitutes the trigger that requires a thread to perform a local reset and therefore perform a transition to the following *frame*. The frame counter is updated and the value of the EVC is reset to the process' unique prime number. However, the old value of the EVC before the overflow event is not lost, but is rather saved in the history map, along with the previous frame value. Supposing that the current REVC of the process can be represented as the tuple (f, e, m), supposing that p is the process' unique prime number and supposing the presence of the T operation which performs the local tick operation on an EVC, as detailed in Section 2.2, and returns the new EVC value, Algorithm 1 contains the *local tick* operation that updates the REVC instance and Algorithm 2 shows the reset function.

Algorithm 1: Local tick operation

```
Input  : An REVC instance (f, e, m) and a prime number p
Output: The updated REVC instance
temp = T (e);
if overflow(temp) then
    (f, e, m) = reset(f, e, m, p);
else
    e = temp;
end
return (f, e, m);
```

Algorithm 2: reset function

```
Input  : An REVC instance (f, e, m) and a prime number p
Output: The reset REVC instance
1 m.put(f, e);
2 f = f + 1;
3 e = p;
4 return (f, e, m);
```

4.1.2 Merge operation

The second operation that makes up a logical clock is the *merge* operation. It is used to generate a new clock value starting from the two values that need to be merged together. Just like the local tick operation, this operation performed on two EVC values can lead to overflow as it requires to find their *least common multiple*, which can be a bigger number than the two inputs. Furthermore, new complications arise when performing such operation on a Resettable Encoded Vector Clock. Clearly, it only makes sense to perform the EVC merge operation between two EVCs that belong to the same execution frame. Furthermore, given that the REVC also contains a history of previous frames, it is not sufficient to merge the current frame's EVCs, but it is also necessary to merge the corresponding previous EVCs. Therefore, two different scenarios can arise. If the two REVCs to be merged are on the same current frame, the two current EVCs can be merged and, in case of overflow, a local reset can be performed following the same rules detailed in the local tick operation. Then, the two history maps need to be merged by merging the EVCs of corresponding frames. If instead the two REVC instances are not on the same current frame, the newer frame is kept untouched along with its EVC value, while the older is merged in the history map of the newer REVC. Finally the two history maps can be merged as previously explained. It should be noted that, while merging the history maps' EVCs, overflow could occurr again. However, we suppose that the data structures for the history maps allow for a sufficiently high number of bits such that these overflows can never happen. Given that all values in the history maps before any merge cannot be bigger than n bits by definition (as this is the threshold that causes overflow of the current EVC value), it is in fact possible to place an upper bound on the maximum number of bits required to store any possible result of all the merge operations. Let it be noted that when multiplying two numbers that are representable on n bits, the result is representable on $2n$ bits in the worst case scenario. Furthermore, when calculating the *least common multiple* of two numbers, in the worst case scenario the result will be the multiplication of the two numbers. Therefore, when performing a merge operation between two EVC values represented on n bits each, in the worst case the result will be representable on $2n$ bits. Finally, in a system with k processes, in the worst case scenario a process will merge with all other processes for a given frame in the history map. Therefore, we can state that the upper bound in terms of bits of each entry in the history map of the REVC for the worst case scenario is:

$$\text{maximum number of bits to represent an EVC} * \text{number of processes in the system} = n * k \tag{4.2}$$

Supposing that (f_2, e_2, m_2) is the REVC that needs to be merged into (f_1, e_1, m_1), that p represents the prime number of the current thread and M represents the merge operation for EVCs, Algorithm 3 contains the *merge* operation that updates the REVC (f_1, e_1, m_1) while Algorithm 4 shows the function that performs the merge of the two history maps.

Algorithm 3: Merge operation

```
Input  : Two REVC instances ($f_1$, $e_1$, $m_1$) and ($f_2$, $e_2$, $m_2$), and a prime number p
Output: The updated REVC instance
if $f_1 > f_2$ then
    $m_1$.put($f_2$, M($m_1$.get($f_2$), $e_2$));
    $m_1$ = historyMerge($m_1$, $m_2$);
else if $f_2 > f_1$ then
    $m_1$.put($f_1$, $e_1$);
    $f_1 = f_2$;
    $e_1 = e_2$;
    $m_1$ = historyMerge($m_1$, $m_2$);
else
    temp = M($e_1$, $e_2$);
    if overflow(temp) then
        ($f_1$, $e_1$, $m_1$) = reset($f_1$, temp, $m_1$, p);
    else
        $e_1$ = temp;
    end
    $m_1$ = historyMerge($m_1$, $m_2$);
end
return ($f_1$, $e_1$, $m_1$);
```

Algorithm 4: historyMerge function

```
Input  : Two history maps $m_1$ and $m_2$
Output: The merged history map
foreach (f, e) in $m_2$ do
    if $m_1$.contains(f) then
        $m_1$.put(f, M($m_1$.get(f), e));
    else
        $m_1$.put(f, e);
    end
end
return $m_1$;
```

4.1.3 Comparison operation

Given two timestamped events, the main purpose of a logical clock is to be able to determine their relationship according to the Lamport's *happened-before* relation. This is the aim of the third and last operation that is analyzed, the comparison operation. Given any two events a_1 and a_2, with their respective REVC timestamps (f_1, e_1, m_1) and (f_2, e_2, m_2), we again distinguish two scenarios. If the two frames f_1 and f_2 are equal, it is possible to test whether $a_1 \rightarrow a_2$ by performing the usual EVC comparison operation between e_1 and e_2. However, if the two frames are not equal, the causality relation can be tested by performing the EVC comparison operation between e_1 and the value of the EVC stored in the history map of a_2 for frame value f_1. This latter operation is equivalent to investigating whether, during frame f_1, the process that has produced the timestamp for a_2 had acknowledged the existence of event a_1 before producing such timestamp. Clearly, if this is true, then $a_1 \rightarrow a_2$. Let it be noted that it could also be true that f_1 is greater than f_2, in which case no comparison is necessary to conclude that $a_1 \rightarrow a_2$ is false as the process executing a_2 cannot have acknowledged a_1 as it is not even reached the same execution frame yet.

Supposing the two events for which the causality relation needs to be established are a_1 and a_2 and their REVCs are respectively (f_1, e_1, m_1) and (f_2, e_2, m_2), and supposing that C is the EVC comparison operation returning *true* if the event passed as first parameter happens before the event passed as second parameter or *false* otherwise, Algorithm 5 contains the comparison operation that tests whether the relation $a_1 \rightarrow a_2$ is true or false:

Algorithm 5: Comparison operation

```
Input  : Two REVC instances ($f_1$, $e_1$, $m_1$) and ($f_2$, $e_2$, $m_2$)
Output: true if ($f_1$, $e_1$, $m_1$) → ($f_2$, $e_2$, $m_2$), false otherwise
if $f_1 > f_2$ then
    return false
else if $f_2 > f_1$ then
    return C($e_1$, $m_2$.get($f_1$))
else
    return C($e_1$, $e_2$)
end
```

4.2 REVC for Shared Memory Systems

In section 3.1 we described the system model for a shared memory environment in which several threads run in parallel and synchronize by means of a set of operations. This section aims at showing how the three basic operations of the Resettable Encoded Vector Clock, which have been previously presented, can be applied to the system's events to obtain the causality relations that describe the system model.

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

The initialization of the system provides each thread with a unique prime number p and each Resettable Encoded Vector Clock is initialized so that $f = 1$, $e = p$ and m is an empty map.

4.2.1 Internal events

Internal events happen locally at a single thread. For the purposes of this analysis, it is supposed that each internal event requires a fresh new timestamp that can uniquely identify it. Let it be noted that this assumption comes with no loss of generality, as an internal event has been defined also as a compound statement composed of several operations. In this latter case, multiple operations end up being labeled with the same timestamp as long as they are part of the same event. Given that no restrictions have been placed on how these compound statements are defined, any implementation is free to choose which operations to label with a new fresh timestamp and which to label with the same previous timestamp.

When a thread executes an internal event, it labels it with the current value of its REVC before performing a *local tick* operation to update its local clock.

It is easy to show how, by following such behaviour, all timestamps of events that happen locally at a thread are consistent with the *program order* rule, such that for any two events a and b, $a \rightarrow b$ is true if and only if a has been executed before b. In fact, by definition, the *local tick* operation applied on REVC value r_1 creates a new REVC value r_2 that satisfies the property $r_1 \rightarrow r_2$ when tested using the *comparison* operation.

4.2.2 Unlock events

Unlock operations are executed when a thread needs to release a given lock object that it has previously acquired. The lock-unlock pattern is one of the basic patterns to allow synchronization among threads, and therefore it introduces a causal relationship among events executed at different threads before and after the synchronization event. In order to correctly represent such relations using the Resettable Encoded Vector Clock, the thread performing the unlock operation needs to label the lock object with its current clock value. The information stored in the lock object can be used by the thread performing the following lock operation to acknowledge the causality, as presented in the following section. Finally, the thread performing the unlock operation should also execute a local tick operation to update its current REVC with a fresh new value for future events, thus ensuring that no other event executed by the same thread following the synchronization point will mistakenly show a causality relationship with the following acquisition event of the lock object.

Clearly, given that all lock objects need to be able to store a REVC value, they should also be initialized with an initial value at the beginning of the program execution. The REVC that can be used for such initialization has $f = 1$, $e = 1$ and m initialized as an empty map.

4.2.3 Lock events

We showed what the behaviour of the Resettable Encoded Vector Clock should be when the thread performs an unlock operation on a lock object. Clearly, whenever no thread is holding the lock of a specific lock object, any thread can perform a lock operation to acquire such lock. The property that needs to be satisfied with this pattern from a causality point of view is the one described by the *lock synchronization* rule from Section 3.1. In practice, the lock event, and any local event that follows in the locking thread, must happen after the previous unlock event. This property can be realized by ensuring that the thread performing the lock operation merges the REVC value, that has been stored on the lock object during the previous unlock operation, into its own current REVC instance. By executing this operation, if l is the lock event and u is the unlock event, the relationship $u \to l$ is enforced, thus satisfying the *lock synchronization* rule.

4.2.4 Fork events

Fork events are executed when a thread starts a new thread during the execution of the program. The fork-join pattern, similarly to the lock-unlock pattern, introduces synchronization events among the threads that are running asynchronously in parallel and therefore it also introduces causality relations. In particular, a fork event should satisfy the property detailed in the *thread creation* rule from Section 3.1. In order to enforce such rule, when the new thread is created, a new unique prime number p should be assigned to it and a new temporary REVC instance should be created and set to (f', e', m'). Supposing that the thread executing the fork operation has current timestamp (f, e, m), the new temporary REVC can be initialized as $f' = f$, $e' = p$ and m as an empty map. Finally, the REVC of the new thread (f'', e'', m'') should be set as the result of the merge operation between (f, e, m) and (f', e', m').

The reason for this set of operations is that, by definition of the merge operation, the new REVC instance associated with the created thread will have acknowledged both the creation event of such thread, given by the new unique prime number, and the fork event itself, given by the timestamp of the thread executing the fork operation.

Clearly, if the new REVC is set as detailed, any new event a executed by the new thread generated by the fork event f will satisfy the relation $f \rightarrow a$, therefore being consistent with the *thread creation* rule.

4.2.5 Join events

The last type of events that we analyze for shared memory systems are join events. Whenever a thread needs to wait for the termination of another thread before resuming its execution, it can do so by executing a join operation. Just as explained in the previous section, this is a synchronization event that introduces a causality relation which should satisfy the *thread termination* rule detailed in Section 3.1. To enforce this, when a join operation terminates, the thread issuing it should set its own REVC as the result of performing a merge operation between its own old REVC value and the terminating thread's REVC value.

This operation clearly advances the thread's clock to acknowledge the termination of the joining thread. Therefore, by performing it, any event a that was executed by the terminating thread before termination will necessarely satisfy the relation $a \rightarrow j$, where j is the join event, thus remaining consistent with the *thread termination* rule.

4.3 REVC for Message Passing Systems

In Section 3.2 we described the system model for a message passing environment in which several processes run concurrently and asynchronously and communicate by means of exchanging messages. This section aims at showing how the three basic operations that have been presented for a Resettable Encoded Vector Clock can be applied at any of the events that can be executed in such environment, in order to implement the causality relations of this model.

Four types of events have been presented in relation to a message passing environment: *internal events*, *send events*, *receive events* and *fence events*. In the following we will detail the behavior of the REVC for each of those events, similarly to what we have done for a shared memory environment. The objective is that, by following the presented rules, the causality relation that is obtained through the application of the comparison operation of the REVC to two timestamped events is consistent with the one that has been described for this model.

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

4.3.1 Internal events

Internal events, similarly to what happens in shared memory environments, are events that are executed locally at the process without any need for synchronization with other processes. Once again we notice that not all events that are executed might need to be labeled with a new fresh timestamp, but this is easily achievable, with no loss of generality, as we defined *internal event* also as a compound statement made up of several operations.

In order to satisfy the *program order* rule detailed in Section 3.2, it is sufficient for the process to execute a local tick operation after labeling the current internal event e_1, with the current REVC timestamp. The new value of the REVC will be used to label the following event e_2, which will execute after e_1. By definition of *local tick* operation, the predicate $e_1 \rightarrow e_2$ will be true, thus remaining consistent with the rule of the model.

4.3.2 Send events

Send events are executed when a process needs to send a message to another process. Similar to the lock-unlock pattern for shared memory systems, the send-receive pattern is a common pattern that is used by processes to synchronize, and thus it introduces causality relations among the events that happen before and after such operations.
In order to consistently represent this relationship with the Resettable Encoded Vector Clock, when a process sends a message to another process it needs to attach to such message its own current REVC timestamp, which will be used by the receiving process to acknowledge the execution of previous events.
Finally a *local tick* operation is also performed by the process executing the send event. This is necessary to avoid that subsequent events at that process will be labeled with the same timestamp that has been used for the send event, thus showing the same causality relationship with the corresponding receive event even though they have been executed afterwards.

4.3.3 Receive events

Receive events are executed when a process is expecting to receive a message from another process. The reception of such message, as previously explained, creates a causal chain among the two processes that is used as a synchronization point and needs to be acknowledged by the Resettable Encoded Vector Clock.

When the process receives the message, it will extract the REVC timestamp that has been attached to it as explained in the previous section. Such timestamp will then be merged into the process' current REVC instance. By executing this, by definition of merge operation, if r is the receive event and s is the corresponding send event, the relationship $s \to r$ is enforced, thus maintaining consistency with the *message synchronization* rule detailed in Section 3.2.

4.3.4 Fence events

Fence events are executed when a process needs to synchronize with several other processes at a barrier, to pause the execution until all participating processes have reached the same barrier. Since it is a synchronization operation, it introduces causality relations among the events happening before and after reaching the barrier at the various participating processes.

In order to consistently acknowledge such relations using the Resettable Encoded Vector Clock, each process participating to the barrier synchronization needs to merge into its own REVC instance all the REVC timestamps of the other participants. By executing such operations, the clocks of all processes at the barrier are aligned to the same timestamp, thus acknowledging all the events previously executed at each process. Therefore, the *barrier synchronization* rule of the model is enforced.

4.4 Analysis and evaluation

In the previous sections we have presented the components and characteristics of the Resettable Encoded Vector Clock along with the rules that define its applicability in both shared memory and message passing environments. The aim of the REVC was to create a valid alternative to the EVC, that would be able to address its very high growth rate and its unbounded growth issue. The solution that has been presented up to this section, however, is able to properly achieve only the first of the results. Furthermore, in achieving such result it introduces a tradeoff in terms of performances which is not compatible with our aim of maintaining the scalability properties of the EVC.

In the following we will analyze and evaluate both objectives, introducing some further considerations where needed, to understand if the Resettable Encoded Vector Clock can be enhanced to achieve both results and what are the tradeoffs needed to do so. We will also present some experimental results to show the practical implications of our reasoning.

4.4.1 Growth rate

The first objective of our Resettable Encoded Vector Clock was to tackle the high growth rate drawback of the EVC. The REVC is built on the EVC and internally makes use of EVC values and operations to implement the logical clock. The EVC has been shown to have exponential growth, due to its use of multiplication and *least common multiple* operations [5]. The exponential behavior of the EVC derives from the fact that the exponents of the prime numbers, used to encode the traditional vector clock values, are monotonically increasing. However, the REVC introduces the use of a reset operation that does not allow the EVC value to grow past a certain number of bits. In practice, this operation allows the REVC to periodically reset the exponents of the prime numbers, therefore avoiding the exponential explosion of the EVC value. The consequence of the application of the reset technique on the number of bits used by the EVC value is that such number fluctuates between a minimum given by the number of bits required to represent the process' prime number, and a maximum of n bits, where n represents the threshold that has been chosen as the overflow value that triggers the reset operation.

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

The experimental data that we collected confirm our claim. In Section 5.4 we present the benchmark suites that we have used for evaluation of the Dynamic Race Detection application. We have exploited this practical application of the Resettable Encoded Vector Clock to track its growth in an actual real-world scenario rather than using a less realistic random simulation.

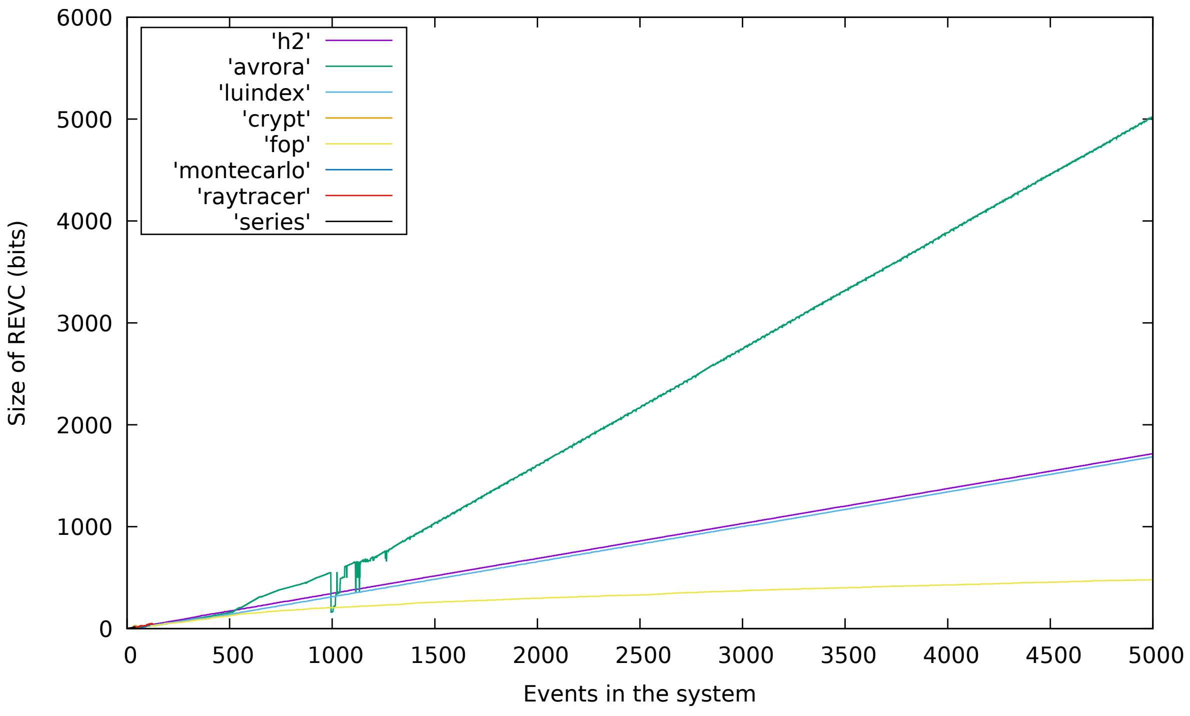

Figure 1. Growth rate for the Resettable Encoded Vector Clock

Figure 1 shows the results of the experiments performed using an REVC-based application run on several benchmark programs. The behavior of the growth rate is linear as predicted, confirming how the use of a reset operation can consistently reduce the exponential explosion of the EVC values.

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

Let it be noted that lines for different benchmark programs show varying slopes. The experimental data shows how such slopes can be related to the number of threads that are present in the system. A bigger number of threads is related to a higher slope in the linear growth, while a lower number of threads is connected to a less steep line. This behavior can be explained as each thread is assigned a unique prime number in an increasing order. Multiplication operations produce numbers that grow faster as the base prime numbers are bigger. Furthermore, as the number of threads increases, merge operations, which are characterized by *least common multiple* operations, produce numbers that on average are composed by a higher number of factors.

Table I details the number of threads that are created by each of the benchmark programs used in our evaluations.

The linear growth rate achievement of the Resettable Encoded Vector Clock guarantees a much more scalable solution that can be concretely used in practical applications requiring a non-trivial number of events to be executed and labeled in the system. However, such achievement is obtained at the expense of performance, introducing a tradeoff among the two. It is easy to notice, in fact, how, while the *local tick* operation and the *comparison* operation

Application	Number of threads
avrora	7
crypt	4
fop	2
h2	3
lufact	4
luindex	3
lusearch	4
moldyn	4
montecarlo	4
raytracer	4
series	4
sor	4
sparsematmult	4

TABLE I: NUMBER OF PARALLEL THREADS EXECUTED BY EACH BENCHMARK APPLICATION

maintain a time complexity $O(1)$, the *merge* operation does not, as it requires to execute the EVC merge operation for f times, where f is the current number of frames, which can grow unboundedly. This is not acceptable in practical applications that require a high number of events to be executed, as the performance of the *merge* operations consistently degrades as the number of execution frames stored in the history map increases. Therefore, further optimization is needed to ensure that the scalability properties that characterized the EVC are not lost. The next section performs a more detailed analysis of the unbounded growth problem, proposing scenarios to adapt the Resettable Encoded Vector Clock to a bounded solution both in time and space requirements. Furthermore, Section 5.2.2 details our proposal for an optimization technique that was developed in our practical implementation of the REVC and that is able to concretely reduce the average complexity of the *merge* operation in practical cases.

4.4.2 Bounded growth

The second focus point of the analysis of the Resettable Encoded Vector Clock is its unbounded growth. As was discussed in the previous section, the REVC has achieved a reduction of the growth rate with respect to the EVC, from an exponential one to a linear one. However, the issue of unbounded growth has not been resolved as the solution that has been presented, without adding any further components, is not sufficient to achieve a time and space bounded logical clock implementation. It is immediate to see that the current EVC value inside the REVC has been correctly set up in such a way that it will never overflow a given threshold of n bits, however the problem has been moved to the history map, as the structure can now grow unboundedly. Furthermore, this solution has also introduced a new tradeoff with performance, given by the new *merge* operation, whose time requirements are now dependent on the number of frames stored in the history map.
Therefore, being able to enhance the REVC to achieve a bounded solution is of paramount importance to ensure realistically acceptable time and space requirements.

We note that the current formulation, which is not space and time bounded, is necessary because the general use case of the REVC requires to be able to perform the comparison operation between any two events of a given program execution. We claim, however, that this requirement can be safely relaxed in many real world applications, where constraints can be added to reduce the required comparisons, without changing at all the semantics of the applications.

It can be found that many real world applications of logical clocks, require to perform causality analysis only on a subset of all events, for example just the ones that have been *recently* executed. If this constraint can be expressed as a maximum number of frames F in the REVC framework, the presented solution is immediately transformed into a bounded one both in time and in space. Each instance of an REVC (f, e, m), in fact, does not need to store any information about frames that are older than $f - F$. Therefore, an upper bound can be placed on the size of the m data structure, and the merge operations only need to merge F EVCs in the worst case scenario. This solution would still allow any pair of events along the whole program execution to be compared to study their causality relationship, as long as they do not happen more than F frames apart. Given that F can be arbitrarily chosen, based on the performance requirements of the REVC implementation, this constraint is likely easily satisfiable by most real world applications. We can therefore stipulate a contract with the application that exploits the Resettable Encoded Vector Clock, under which the Bounded REVC has an equivalent behavior to the Unbounded REVC. This contract can be formalized and expressed as a predicate that establishes whether two timestamped events, a_1 and a_2, with respective timestamps (f_1, e_1, m_1) and (f_2, e_2, m_2), can be compared under the Bounded REVC, given a maximum amount of stored frames F:

$$a_1 \textit{ comparable } a_2 \iff |f_1 - f_2| \leq F \tag{4.3}$$

Following the same procedures that have been detailed in the previous section, we have gathered experimental data to support our claim that this formulation leads to a bounded implementation of the REVC. We will here show the results that demonstrate the bounded growth in terms of space of the REVC. Furthermore, Section 5.2.1 will present an optimization technique built on this bounded formulation of the REVC that we have developed in our practical implementation, which will be evaluated in terms of time performance in Section 5.4.

Figure 2 and Figure 3 show the same benchmark programs that have been evaluated in the previous section, run using the new REVC formulation with respectively $F = 30$ and $F = 5$. The bounded growth is clearly visible as the linear increase in size of the REVC instance reaches a plateau when the maximum number of stored frames reaches F. The growth rate is then reduced to an average of 0, as the actual size of the REVC instance fluctuates around the plateau. This fluctuation is caused by the EVC value of the current execution frame, whose size in bits oscillates between a minimum given by the number of bits needed to represent its prime number, and a maximum given by the chosen threshold of n bits which triggers the reset operation. When the execution reaches the plateau, however, at each reset operation a new value is added to the history map, as an old one is removed, therefore maintaining a constant total size.

We note again that the plateaus of different applications differ in the size in bits. We can again conclude that the number of threads that are present in the given application is directly

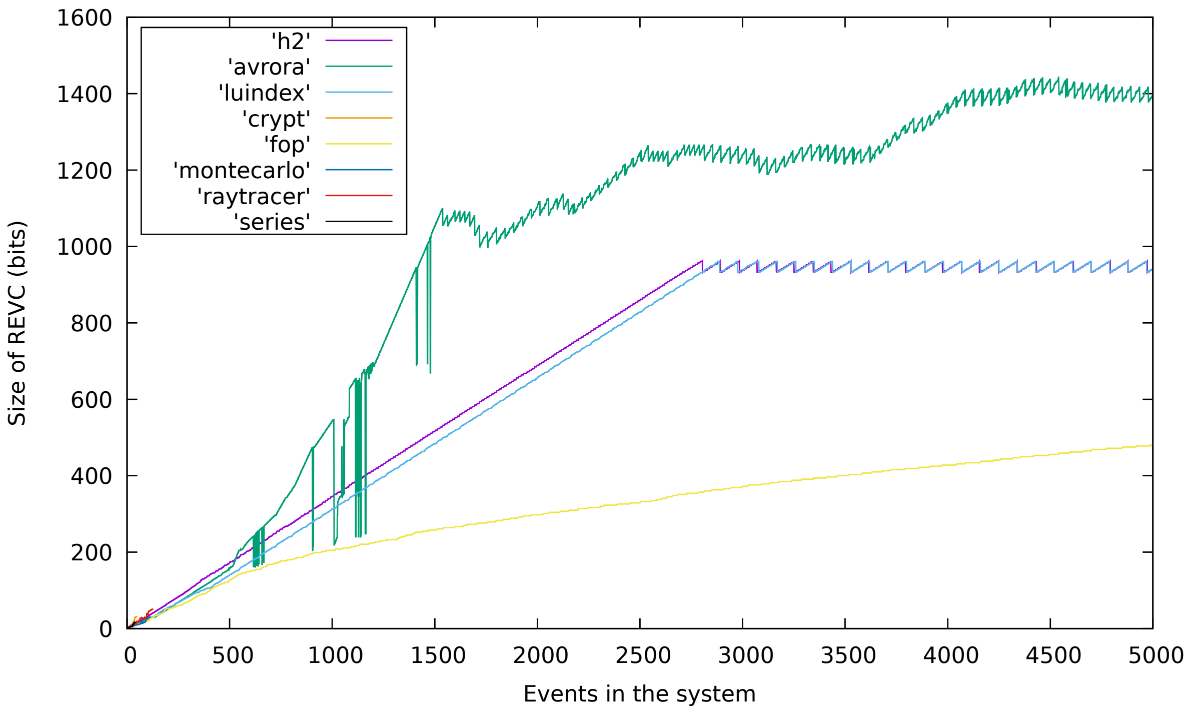

Figure 2. Growth rate for the Bounded REVC with $F = 30$

proportional to the size of the plateau. This is a direct consequence of the formula that we have defined in section 4.1.2 to calculate the upper bound for an entry in the history map of the REVC. Clearly, the total upper bound of the size of the history map when the plateau is reached will be:

$$\text{upper bound for one entry} * F \tag{4.4}$$

Since the upper bound for one entry is dependent on the number of threads, it follows that the position of the plateau in the graph must also depend directly on the number of threads.

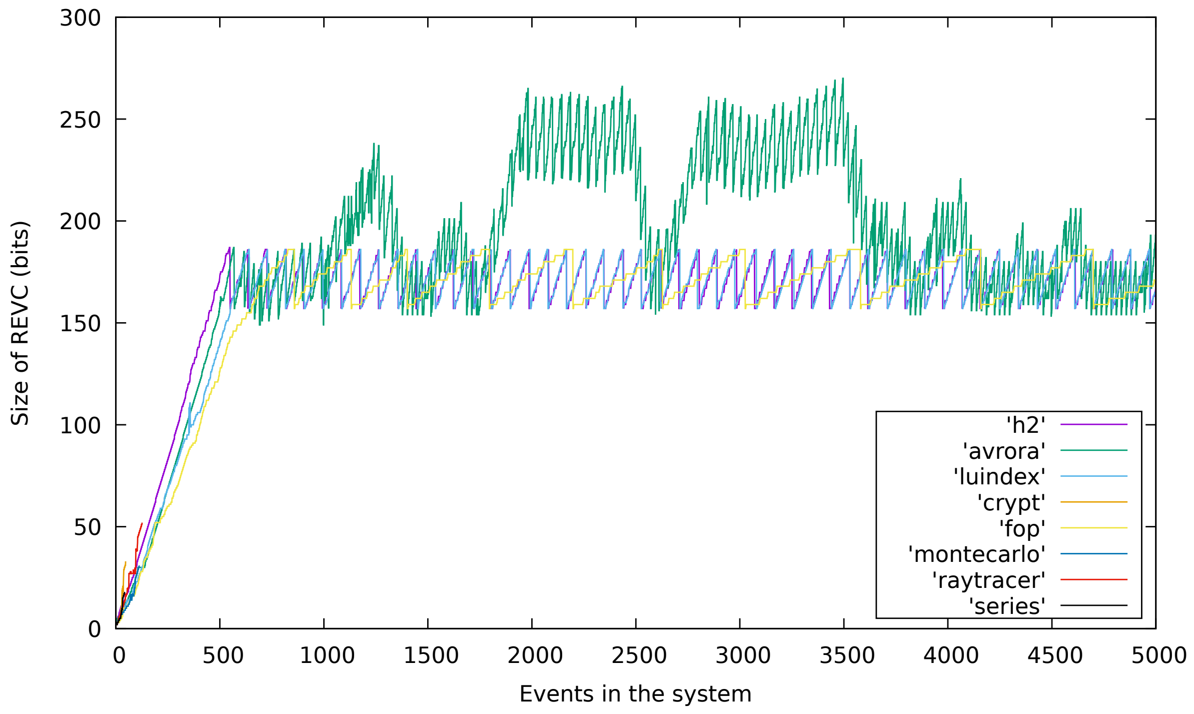

Figure 3. Growth rate for the Bounded REVC with $F = 5$

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

CHAPTER 5

REVC FOR DYNAMIC RACE DETECTION

We have presented the Resettable Encoded Vector Clock as an alternative logical clock implementation to track causality relations among events. We have shown its advantages with respect to the Encoded Vector Clock and we have discussed possible variants that allow to tune and improve performance.

In this chapter we explore a practical application of logical clocks, Dynamic Race Detection, with the objective of showing an application of our REVC and evaluating its performance against traditional logical clock implementations.

Section 2.3 presented the Dynamic Race Detection problem and the DJIT$^+$ protocol, which exploits traditional Vector Clocks to achieve its results. We will present a modified version of the DJIT$^+$ protocol that exploits the REVC in place of the VC to track causality relations among events. Finally, we will propose two optimization techniques for the REVC that can achieve significant performance gain, as will be presented in the analysis of the results.

5.1 DJIT$^+$ exploiting the REVC

Section 2.3.2 presented the DJIT$^+$ protocol for Dynamic Race Detection. Our modified version of this protocol exploits the Resettable Encoded Vector Clock with the objective of obtaining a more performant solution. We have previously presented rules to apply the REVC both on message passing systems and on shared memory systems. Our work on Dynamic Race

Detection will be carried out on a shared memory system, therefore the assumption in the following, unless otherwise noted, will be that of a multi-threaded parallel environment on a shared memory system.

In order to apply the REVC to the DJIT$^+$ protocol, instead of using vector clocks, each thread maintains an instance of the Resettable EVC, characterized by the tuple (f, e, m) where f is the current frame, e is the current EVC value and m is the history map of past EVC values for past frames. Each thread is also assigned a unique prime number that will be used for operations on the REVC.
Furthermore, each lock object is assigned an instance of the REVC as well, which is initialized to what we call the *basic REVC*, defined as (f_0, e_0, m_0) where $f_0 = 1$, $e_0 = 1$ and m_0 is an empty map.

The rules for updating and propagating the values of the Resettable Encoded Vector Clock in a multi-threaded environment on a shared memory model are the ones that have been presented in Section 3.1, which are very similar to the rules that are used by the original DJIT$^+$ protocol to propagate and update the Vector Clock.
Our system differs, however, in the information that are stored regarding memory location accesses and in the way that races are detected making use of the REVC.

Each memory location is labeled with a status which can be either *READ*, *WRITE* or *READ-SHARED*. As we have previously stated, all write operations to a specific memory location need to *happen-after* any other read or write operation to that same memory location that has happened up to that moment in the execution. No write operation can be concurrent with any other operation on the same memory location without generating a *race condition*. However, as we have pointed out, multiple read operations can be concurrent without generating races, therefore we need to address the possibility of memory locations which are in *read-shared* mode.

The status of the memory location traces the latest access operation which has been performed on it. The difference between the *READ* and the *READ-SHARED* modes is essentially that when a memory location is in *READ* mode, the latest operation has been a read which has surely *happened-after* all the operations that have preceded it. On the other hand, when the status is *READ-SHARED*, multiple concurrent read operations have been performed on the memory location, and as such they are all being tracked to ensure that no subsequent write operation can happen concurrently to any of those read operations.

Along with the described status, each memory location is labeled with the information of the REVC that are necessary to detect race conditions, as follows:

- **READ or WRITE status**: when the status of a memory location is labeled with either *READ* or *WRITE*, the latest operation performed on that memory location has been found to have *happened-after* all other previous operations, therefore only the current frame and EVC value of the REVC of the thread that has performed such operation are needed in order to be able to detect a possible concurrent operation and consequently a possible race condition
- **READ-SHARED status**: when the status of a memory location is labeled with *READ-SHARED*, multiple concurrent read operations have been performed, therefore a structure

needs to be saved containing for each read operation the current frame and evc value from the REVC of the thread performing it. Clearly, EVC values pertaining to the same frames can be merged together to reduce the amount of information that needs to be stored.

The first representation that has been described can be actually defined as a special instance of a Resettable EVC. Given the REVC representation as the tuple (f, e, m), the structure that needs to be attached to a *READ* or *WRITE* status is simply an REVC that has f and e respectively set as the current frame and EVC value of the thread, while m is set as an empty map. The structure that needs to be attached to a *READ-SHARED* status, on the other hand, is a slightly modified instance of an REVC that contains the result of merging several REVCs of the previously described type. Therefore, the frame f and EVC value e of the result will contain the newest frame with the corresponding result of merging all the EVC values of that frame. The history map, on the other hand, will contain all previous frames that have been merged, with the corresponding EVC values that derive from merging all the EVC values of corresponding frames.

We also define a slightly different behavior for the *comparison* operation when applied to this latter structure. In such a case, in fact, it is necessary to apply the usual REVC comparison operation to all the frames contained in the defined structure, rather than just to one of them. This ensures that all the operations that are being tracked by the REVC instance have happened before the timestamp that is being checked.

We will now show how races can be identified using the Resettable Encoded Vector Clock. Let's suppose that thread t performing operation o with REVC (f, e, m) accesses a memory location which is labeled with status s and REVC (f_1, e_1, m_1). The following rules to detect race conditions apply:

- **Read-Write race**: if s is set as *READ* or *READ-SHARED* and the current operation being performed by t is a write operation, a *Read-Write* race is detected if it is not true that $(f, e, m) \rightarrow (f_1, e_1, m_1)$. Formally

$$\begin{aligned} \textit{R-W race} \iff & (s == \textit{READ} \vee s == \textit{READ-SHARED}) \wedge \\ & o == \textit{WRITE} \wedge \neg((f, e, m) \rightarrow (f_1, e_1, m_1)) \end{aligned} \tag{5.1}$$

- **Write-Read race**: similarly, if s is set as *WRITE* and the current operation being performed by t is a read operation, a *Write-Read* race is detected if it is not true that $(f, e, m) \rightarrow (f_1, e_1, m_1)$. Formally

$$\textit{W-R race} \iff s == \textit{WRITE} \wedge o == \textit{READ} \wedge \neg((f, e, m) \rightarrow (f_1, e_1, m_1)) \tag{5.2}$$

- **Write-Write race**: finally, if s is set as *WRITE* and the current operation being performed by t is a write operation, a *Write-Write* race is detected if it is not true that $(f, e, m) \rightarrow (f_1, e_1, m_1)$. Formally

$$\textit{W-W race} \iff s == \textit{WRITE} \wedge o == \textit{WRITE} \wedge \neg((f, e, m) \rightarrow (f_1, e_1, m_1)) \tag{5.3}$$

Finally, after the thread performing an access to the memory location has tested the operation for possible races, it can update the information that are stored for that memory location. The update operations that are performed depend on the current status of the memory location and the type of access. Three different scenarios can be distinguished with different update rules, and they are detailed in the following.

- **READ or WRITE status with following READ access**: if the access type is *READ* and the operation that is being performed *happens-after* the previous operation stored at that memory location, a new REVC (f_1, e_1, m_1) can be created with f_1 and e_1 set as the thread's current frame and EVC values and m_1 set as an empty map. The status of the memory location can then be set as *READ* and the new REVC can be stored in place of the previous one.
- **READ, READ-SHARED or WRITE status with following WRITE access**: similarly, if the access type is *WRITE* and the operation that is being performed *happens-after* the previous operation stored at that memory location, the same process can be followed. The new REVC is created as previously explained, but the status of the memory location is now set to *WRITE*.
- **READ or READ-SHARED status with concurrent READ access**: A slightly different update needs to be performed when the current status of the memory location is *READ* or *READ-SHARED* and the current access is a *READ* operation which is concurrent with the previous operation stored at the memory location. As we have stated previously, concurrent read operations are allowed and do not constitute a race condition,

however they all need to be tracked in order to be able to correctly identify future races in case a subsequent *WRITE* access is performed. This can be easily achieved using the *merge* operation of the REVC. A new Resettable EVC can be created with the same rules described for the other two scenarios and it can then be merged into the current REVC that is stored for that memory location. The merge operation will create a new REVC which will contain all the information needed to be able to track both the accesses. Finally, the memory location status is set to *READ-SHARED*

5.2 REVC optimization techniques

The Resettable Encoded Vector Clock has been defined as a bounded solution, both in time and space requirements, only if its implementation can satisfy a contract that sets an upper bound on the maximum amount of information that is stored per REVC instance. In the previous section we presented the rules that allow the REVC to be used in place of a traditional vector clock in the DJIT^{+}algorithm, but we made no assumptions on any constraint that needs to be enforced. Therefore, without adding any further modification to the protocol, the size of the REVC instances is allowed to grow unboundedly and possibly overflow the maximum amount of memory that is allocated for the application or incur significant overhead for *merge* operations, which, as it has been previously stated, is the only operation that depends on the size of the history map.

In practical cases, this might still be acceptable, especially for applications that exhibit a behaviour in which the REVCs have a very low growth rate, for example applications that have few synchronization points among the threads, and concentrate mainly on parallel work. Given the rules presented in the previous section, it can easily be seen that only synchronization instructions contribute to the growth of the clocks, while all other operations, particularly memory accesses, simply execute *comparison* operations, without modifying the values of the REVCs.

However, in general, the application of the protocol as it has been described in the previous sections carries with it the risk of incurring high memory growth and high overhead which can lead to poor performances. Therefore, two optimization techniques are proposed that aim at reducing both space and time overheads and improving overall performance of the implementation. Both techniques can be employed by themselves or in conjunction with each other and present a tradeoff that needs to be carefully evaluated to achieve the desired results.

5.2.1 Fixed Size Frame Window

The first optimization that is presented to the REVC protocol is the **Fixed Size Frame Window** (FSFW). This optimization derives directly from the contract that has been outlined in section 4.4.2 and allows to place an upper bound on the maximum number of frames that are stored in the history map of REVCs. This technique allows to reduce both time and space requirements. It is intuitive to see how, by storing only a limited amount of frames, the space requirement becomes bounded and can be arbitrarily reduced by choosing a smaller window value. Time requirements, on the other hand, are improved because the *merge* operation of REVCs is dependent on the number of frames that are present in the history map, and therefore, by ensuring that such number never grows past a given threshold, an upper bound can be placed also on the amount of instructions required to perform such operation.

The contract outlined by the REVC also claims that the results of using a *Fixed Size Frame Window REVC* with window size F are equivalent to those of an Unbounded REVC, as long as all pairs of timestamped events that need to be analyzed to establish causality relations, happen at most F frames apart one from the other. This has been formalized with Equation 4.3, which provides a boolean predicate that can establish whether any pair of events is comparable under a Fixed Size Frame Window REVC.

The practical consequences of such a constraint on our utilization of the REVC in the Dynamic Race Detection protocol, are that, given a certain frame window size F, the protocol will only be able to detect races that happen within the last F frames. This is derived easily from the rules that have been previously presented. The comparison operation of the REVC is used in our modified version of the DJIT$^+$ protocol only for race detection purposes, and any comparison operation always uses as one of the operands a thread's current clock value. Therefore, only when the two access events under investigation are comparable, as given by the defined predicate, a possible race condition can be analyzed. This means that races can be detected only if the previous access to the memory location has happened within the last F frames from the thread's current frame.

This optimization technique, therefore, clearly presents a tradeoff between performance and precision. If a lower the value of F is chosen, a lower number of frames needs to be stored and therefore a lower overhead is imposed on the *merge* operation. However, on the other hand, a lower window is available to detect races, and therefore races happening outside of such window cannot be detected. Viceversa, a higher value of F will incur higher penalties performance-wise but will allow a higher precision when detecting race conditions.

The intuition that stands behind the adoption of such an optimization, however, is the fact that a race condition, by definition, is a tentative access to a memory location by two threads which are accessing it **concurrently**, and is therefore highly probable that the two operations will happen within a low number of instructions one from the other, and most likely within a low number of synchronization points, which we have shown to be the operations that increment the REVC timestamps. By selecting a fairly low value of F, we still allow a relatively high number of synchronizations to happen between the two accesses, and therefore we should still maintain a relatively high precision.

This intuition is confirmed by the data that we collected during our practical experimentation. In Section 5.4 we will present the results of our analysis and we will show that, despite choosing very low values of F, our system is still capable of detecting races with 100% precision over the suite of applications that we used for evaluation.

5.2.2 Differential Merge Technique

The second optimization that is presented to the Resettable Encoded Vector Clock is the **Differential Merge Technique** (DMT). This optimization aims at reducing the impact that a high number of stored frames has on the *merge* operation. It derives from the observation that, based on the standard definition of the merge operation for the REVC, each time an instance of the Resettable EVC is merged into another, all the frames that are stored in the first timestamp's history map need to be merged with the corresponding frames in the second timestamp's history map. Merging such frames practically translates to performing the EVC *merge* operation, which is nothing more than finding the least common multiple among the two numbers. However, if the two values to be merged are e_1 and e_2, and e_2 is already a multiple of e_1, then merging e_1 into e_2 will not actually change the value of e_2, as the result of $lcm(e_1, e_2) = e_2$.

Let us now analyze the following scenario: suppose that REVC (f_1, e_1, m_1) is merged into REVC (f_2, e_2, m_2), then a number of instructions are executed and finally (f_1, e_1, m_1) needs to be merged again into (f_2, e_2, m_2). When the *merge* operation addresses the history maps, most likely only a subset of the frames of m_1 actually needs to be considered for merging. This is because, as stated, only the frames of m_1 for which the EVC values have changed since the previous merge will produce a new result. For all other frames, however, the EVC values contained in m_2 are already multiples of the corresponding values contained in m_1.

Building from this observation, we can also intuitively state that, as an execution progresses, older frames tend to not get updated anymore and therefore performing the least common multiple operation at every merge for all frames introduces an unnecessary overhead. A smarter merge operation can be defined, to reduce such overhead. We can also state, based on such observations, that its time complexity would become dependent on the number of stored frames only in the worst case scenario, while on average only a subset of such frames would need to be merged.

Clearly, however, to define such a smarter merge operation, each thread would also need to keep track of what frames inside its history map have been modified since the last merge operation with *each* other thread, which would result in a higher storage requirement. Furthermore, a new data structure would need to be defined, capable of tracking such information.

Once again, this optimization technique presents a tradeoff between space and performance. Let it be noted, however, that this optimization does not affect the precision of the protocol, as no assumptions are made on the events that can be compared, and therefore we claim that the results of using a *REVC with Differential Merge* are always equivalent to those of a traditional Resettable Encoded Vector Clock.

We now define an enhanced version of the REVC, ready to be employed with the Differential Merge Technique. This extended REVC is defined as the tuple (f, e, m, d) where f is the current frame, e is the current EVC value, m is the history map of past frames and corresponding EVC values and finally d is a **difference map**, containing for each other thread a list of frames whose EVC value contained in m has been modified since the last merge into the REVC of such thread. Supposing that REVC (f_2, e_2, m_2, d_2) is to be merged into REVC (f_1, e_1, m_1, d_1), p represents the prime number of the current thread and its unique identifier, and M represents the merge operation for EVCs, Algorithms 6 and 7 define the new merge operation. The reset function remains unchanged and is defined in Algorithm 2.

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

It is worth noting that both the optimization techniques that have been described can be combined into a Resettable Encoded Vector Clock which has a Fixed Size Frame Window and also adopts only Differential Merging, so that, even if a lower number of frames is kept, they are still updated upon a merge operation only in case they have been modified since the previous merge operation with the same thread.

Algorithm 6: Merge operation with Differential Merge Technique

```
Input  : Two REVC instances ($f_1$, $e_1$, $m_1$, $d_1$) and ($f_2$, $e_2$, $m_2$, $d_2$), and a prime
         number p
Output: The updated REVC instance
if $f_1 > f_2$ then
    $m_1$.put($f_2$, M($m_1$.get($f_2$), $e_2$));
    $m_1$ = historyMerge($m_1$, $m_2$, $d_2$, p);
else if $f_2 > f_1$ then
    $m_1$.put($f_1$, $e_1$);
    $f_1 = f_2$;
    $e_1 = e_2$;
    $m_1$ = historyMerge($m_1$, $m_2$, $d_2$, p);
else
    temp = M($e_1$, $e_2$);
    if overflow(temp) then
        ($f_1$, $e_1$, $m_1$) = reset($f_1$, temp, $m_1$, p);
    else
        $e_1$ = temp;
    end
    $m_1$ = historyMerge($m_1$, $m_2$, $d_2$, p);
end
return ($f_1$, $e_1$, $m_1$, $d_1$);
```

5.3 Implementation

In Section 2.3.1 we presented the RoadRunner framework for dynamic analysis of Java programs. We showed how such framework is capable of instrumenting Java bytecode upon class loading, to enable the generation of an event stream during program execution that can be processed by an arbitrary backend tool to implement any dynamic analysis behavior.

Algorithm 7: historyMerge function with Differential Merge Technique

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

```
1 foreach f in d.get(p) do
2     if $m_1$.contains(f) then
3         $m_1$.put(f, M($m_1$.get(f), $m_2$.get(f)));
4     else
5         $m_1$.put(f, $m_2$.get(f));
6     end
7 end
8 d.get(p).empty();
9 return $m_1$;
```

Our implementation of the modified DJIT$^+$ protocol using the Resettable Encoded Vector Clock was built on top of the RoadRunner framework. We extended the software to build a new backend tool capable of processing the event stream to exercise the behaviour that was described in Section 5.1.

The tool defines callbacks for all the events that need to be monitored for the protocol to work correctly. In particular, synchronization operations that exploit lock objects are processed, to allow the tool to update the information contained in the REVC instance connected with such objects as previously explained. The synchronization primitives provided by the Java programming language that make use of lock objects are *acquire* and *release* of locks, *wait* and *notify* operations which are used by threads to implement condition variables. Furthermore, *fork*, *start* and *join* operations are used to create new threads or wait for their termination.

Finally, access operations to all types of memory locations clearly need to be tracked, as they represent the events that are under scrutiny by the tool to identify race conditions.

RoadRunner provides each thread object, lock object and memory location with a data structure to store tool-specific information. Our tool customizes such data structures to include the possibility of storing an REVC instance. In particular, for threads, a complete instance of the REVC is necessary, along with a unique prime number that allows the thread to perform all needed operations on the REVC, while the program is executing. Lock objects need a complete instance of the REVC as well, but they don't need to be assigned a unique prime number as they only need to store specific timestamps, but never need to perform *local tick* operations on the REVCs that they are storing. Finally, memory locations need to store a timestamp as well, like lock objects, but they also need to track the current status of that memory location, either *READ*, *WRITE* or *READ-SHARED*, as it has been detailed in previous sections. As a further optimization, when the status is either *READ* or *WRITE*, the identifier of the thread performing the latest operation is also stored. This information allows for a slightly faster processing, as, when an access operation performed by thread t is under scrutiny, if the previous operation had also been performed by thread t, no further checks on the values of the REVCs are necessary as the operation necessarily *happens-after* the previous one, because of the intrinsic *program order*.

A class defining the REVC and its operations has been created, and instances of such class are attached to each data structure that needs to be enabled to manage REVCs. We defined the EVC value e contained in each REVC instance to be representable as an integer over 32 bits. Therefore, the threshold of n bits, whose overflow triggers the reset operation, is set as 32 bits. When a callback is invoked, the payload containing information on the event is accessed and the data structures attached to the actors of the event are retrieved. Finally, the callback performs the requested operations to implement the behavior of the modified DJIT$^+$ protocol. Algorithms 8 and 9 detail simplified versions of the *acquire* and *release* operations, to present an example of such callbacks.

Finally, simplified code for the management of the *memory access* operation is presented by Algorithm 10 to show how the tool can analyze and detect race conditions.

Algorithm 8: The callback for the acquire event

```
Input : An event object event
function acquire(event)
    threadREVC = getREVCForThread(event.getThreadId());
    lockREVC = getREVCForLock(event.getLock());
    threadREVC.merge(lockREVC);
end
```

Algorithm 9: The callback for the release event

```
Input   : An event object event
function release(event)
    threadREVC = getREVCForThread(event.getThreadId());
    lockREVC = getREVCForLock(event.getLock());
    lockREVC.merge(threadREVC);
    threadREVC.localTick();
end
```

5.4 Evaluation

In this section we aim at analyzing and evaluating our solution in terms of performance, by comparing it to the other tools that have already been developed for Dynamic Race Detection.

In order to be able to provide a fair evaluation of our system with respect to the traditional DJIT$^+$ protocol, we also developed an implementation of DJIT$^+$ on the RoadRunner framework. FastTrack, on the other hand, was already implemented on top of RoadRunner, as it has been studied and developed by Flanagan and Freund as well.

The applications that were chosen for evaluation are a subset of the applications that are found in the DaCapo Benchmarking Suite [19] and the Java Grande Benchmarking Suite [20]. Those two suites are composed of Java programs that have been designed to emulate non-trivial loads and be representative of intensive calculations. The applications that have been chosen exhibit multi-threaded behaviour and very diverse semantics, in order to be able to test the system on a relatively complete set of programs ranging from a very high usage of synchronization to a high parallelization of the workload.

Algorithm 10: The callback for the access event

```
Input   : An event object event
function access(event)
    threadREVC = getREVCForThread(event.getThreadId());
    memInfo = getDataStructureForMemoryLocation(event.getMemoryLocation());
    if event.isWrite() then
        eType = Type.WRITE;
    else
        eType = Type.READ;
    end
    if memInfo.tId == event.funcGetThreadId then
        memInfo.type = eType;
        memInfo.revc = threadREVC.copyCurrentFrame();
        return;
    end
    if eType == Type.WRITE then
        if threadREVC.happensAfter(memInfo.revc) then
            memInfo.type = eType;
            memInfo.revc = threadREVC.copyCurrentFrame();
            memInfo.tId = event.funcGetThreadId;
        else
            reportRace(memInfo.type + " - WRITE race");
        end
    else
        if threadREVC.happensAfter(memInfo.revc) then
            memInfo.type = eType;
            memInfo.revc = threadREVC.copyCurrentFrame();
            memInfo.tId = event.funcGetThreadId;
        else if memInfo.type == Type.READ or memInfo.type ==
          Type.READ-SHARED then
            memInfo.type = Type.READ-SHARED;
            memInfo.revc.merge(threadREVC.copyCurrentFrame());
            memInfo.tId = -1;
        else
            reportRace("WRITE - READ race");
        end
    end
end
```

All experiments have been carried out on a system with a Dual-Core Intel i7 2.8 GHz processor and 12 GB of RAM running Linux Ubuntu 18.04 and Java 8.
Table I details the number of threads that are run in parallel by each of the chosen applications. Our modified version of DJIT$^+$ that exploits the Resettable Encoded Vector Clock has been tested with several configurations, aimed at comparing the performance of the basic version versus the performance of the versions employing the *Fixed Size Frame Window* (FSFW), the *Differential Merge Technique* (DMT) or a combination of the two. Furthermore, several different sizes of the frame window have also been tested, to assert the impact on the precision of the tool and on the performance. Table II shows the execution time in milliseconds of each application, for each tool that has been tested.
The first column shows the execution time of the application without the overhead of a dynamic analysis tool, while the following columns show the execution times of the same applications when they are instrumented by RoadRunner and processed by a dynamic analysis tool. Our modified version of DJIT$^+$ which exploits the REVC is tested in various flavors, first without any optimization technique, then using the sizes 30, 5 and 1 for the FSFW, in varying combinations with and without the DMT.

As expected, when the REVC is used without any limitation to the amount of frames that are stored as part of the history, applications that perform frequent synchronization, and therefore produce high growth in the REVCs, use up all the available memory and return an out-of-memory error (OOM). This is the case for *avrora*, *h2*, *luindex* and *lusearch*. It is however interesting to notice that specific applications which make frequent use of comparison

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

TABLE II: EXECUTION TIME COMPARISON

operations, but do not require high growth of the logical clocks, perform the best without any form of optimization, because the overhead of the additional processing or maintenance of the required data structures for the optimization techniques is higher than the performance benefit that is obtained. This behaviour can be noticed in particular for *raytracer* and *crypt*.

However, on average, a reduced size of the frame window results in higher overall performances as would be reasonable to expect. Furthermore, by bounding the maximum amount of memory that can be used, no more out-of-memory errors arise and all applications can be successfully analyzed by the protocol. Similarly, the results show that the Differential Merge Technique, when activated, is able to increase the overall performances by reducing the overhead of merge operations.

Table III clearly presents the performance comparison, by showing the speedup achieved over the DJIT$^+$ protocol for each application and the average speedup across all benchmarks. The configurations that caused out-of-memory errors do not present an aggregate average measurement as it would not be comparable to the other configurations due to the lack of data for such applications.
It is clear by looking at the average speedup data that our REVC-based application is able to outperform by up to 1.6 times the traditional DJIT$^+$ protocol, even if it is not able to achieve performance that is comparable to the state-of-the-art FastTrack protocol that employs Scalar Clocks.

We claim, however, that this result is a promising achievement as it allows us to show that the REVC can have practical applications, which are competitive for scenarios in which no alternatives to a traditional Vector Clock implementation are available. Furthermore, such results allow us to rank the performance that can be obtained by such protocol with respect to the most commonly used logical clock implementations, laying the ground work for future developments that can employ our protocol to obtain new state-of-the-art tools applying it to other problems, as suggested in Section 6.2.

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

Application	REVC DJIT$^+$	REVC DJIT$^+$ DMT	REVC DJIT$^+$ FSFW 30	REVC DJIT$^+$ FSFW 5	REVC DJIT$^+$ FSFW 30 + DMT	REVC DJIT$^+$ FSFW 5 + DMT	REVC DJIT$^+$ FSFW 1	FastTrack
avrora	-	-	0.996	1.465	1.499	1.603	1.711	2.678
crypt	0.547	0.513	0.529	0.539	0.536	0.540	0.543	3.740
fop	0.399	1.520	1.445	1.460	1.549	1.551	1.483	1.817
h2	-	-	0.967	0.947	1.035	1.042	1.062	2.389
lufact	1.844	1.684	1.693	1.732	1.895	1.856	1.898	2.898
luindex	-	-	1.470	1.768	1.805	1.822	1.794	3.411
lusearch	-	-	1.240	1.457	1.334	1.299	1.542	2.736
moldyn	1.648	1.740	1.622	1.676	1.758	1.697	1.725	2.898
montecarlo	3.228	3.498	2.904	3.170	3.242	3.257	3.331	6.437
raytracer	3.330	1.947	2.591	1.774	1.773	1.906	2.112	4.690
series	1.000	1.010	0.995	0.996	1.005	1.010	0.982	1.018
sor	1.953	2.086	1.908	1.990	1.990	2.000	1.910	2.579
sparsematmult	0.745	0.777	0.758	0.732	0.764	0.756	0.719	1.608
AVERAGE	**-**	**-**	**1.470**	**1.520**	**1.553**	**1.564**	**1.601**	**2.992**

TABLE III: SPEEDUP OVER TRADITIONAL DJIT$^+$ COMPARISON

CHAPTER 6

CONCLUSION

6.1 Achievements

In this thesis work we have presented the Resettable Encoded Vector Clock, a logical clock implementation that is built on top of the Encoded Vector Clock with the objective of tackling its main drawbacks: high growth rate and unbounded growth.
We have defined and formalized the REVC, and showed its applicability in both shared memory systems and message passing systems.

We have presented experimental results that show how we have been able to achieve both objectives. In particular, the REVC presents a linear growth rate, with respect to the exponential growth rate presented by the EVC. Furthermore, the REVC's definition allows to easily tune the implementation to place an upper bound on the storage requirements, at the theoretical expense of precision. We have however claimed that practical applications rarely require unconstrained comparability among the events, and we have showed how there exist real-world scenarios in which the Bounded REVC can achieve no loss in precision.

Finally, we have explored a practical application of the REVC by applying it to the Dynamic Race Detection problem, with the aim of evaluating its performance with respect to other traditional logical clock implementations, and showing its practical applicability in real-world scenarios.

6.2 Future work

Our aim in this thesis work was to go beyond the theoretical presentation of the Resettable Encoded Vector Clock as a sound logical clock implementation, by also showing practically its applicability to a real problem. We chose the Dynamic Race Detection problem, as its literature presented both a traditional Vector Clock approach and a state-of-the-art approach capable of employing Scalar Clocks for most operations and dynamically adapt to Vector Clocks only if needed.

This application allowed us to provide experimental data to better analyze and compare the REVC with other famous logical clock implementations. Given the results of our work, we have shown that the REVC can provide better performance than a traditional Vector Clock, even if it is not able to achieve performance that can compete with a Scalar Clock. The logical continuation of our work would be to identify one or more other applications that make use of logical clocks but, however, do not present any alternative to Vector Clock implementations. It is reasonable to believe that, if a protocol can be studied to apply the REVC in such scenarios, better performance might be achieved, thus creating new state-of-the-art tools.

In our work, we have mainly experimented with applications that require a low number of threads, as we have never applied the REVC on programs with more than 7, as detailed in Section 5.4. However, one of the strengths of the REVC with respect to the EVC and the traditional Vector Clock, is its better scalability with respect to the number of processes or threads in the system. Therefore, another logical continuation of our work could be expanding it in scenarios requiring high number of parallel processes or threads, to measure and quantify its robustness and possibly its higher performance with respect to other logical clock implementations.

Finally, the Resettable Encoded Vector Clock's main drawback is given by the performance hit and memory consumption generated by the necessity of storing the history map. Further theoretical work could be carried out to understand whether a more compact representation of such structure can be defined, or an extension of our Differential Merge Technique can be studied to further reduce the time requirements for merge operations of the history maps. This achievement would greatly benefit the REVC protocol, possibly allowing it to compete with Scalar Clock implementations in real-world applications.

6.3 Final remarks

The main result of this thesis work has been the theoretical presentation of the Resettable Encoded Vector Clock as a valid alternative to the Encoded Vector Clock and other logical clock implementations. However, our results have also been able to show how the REVC is not just a theoretical concept, but it is applicable to practical problems and can compete in terms of both space and time requirements with other known protocols.

Finally, the REVC has been designed with scalability and adaptability in mind. Its formulation contains several intrinsic tradeoffs that can be easily tuned by enabling or disabling optimization techniques, and choosing between Bounded and Unbounded implementations. These configurations provide the REVC with a much higher adaptability to very different scenarios, which cannot be found in other logical clock implementations.

These achievements are very promising and, as we have showed, 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.

CITED LITERATURE

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

2. Fidge, C.: Logical time in distributed computing systems. Computer, 24(8):28–33, 1991.

3. Mattern, F.: Virtual time and global states of distributed systems. In PARALLEL AND DISTRIBUTED ALGORITHMS, pages 215–226. North-Holland, 1988.

4. Kshemkalyani, A. D., Khokhar, A., and Shen, M.: Encoded vector clock: Using primes to characterize causality in distributed systems. In Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN '18, pages 12:1–12:8, New York, NY, USA, 2018.

5. Kshemkalyani, A. D. and Voleti, B.: On the growth of the prime numbers based encoded vector clock. In Distributed Computing and Internet Technology, eds. G. Fahrnberger, S. Gopinathan, and L. Parida, pages 169–184. Springer International Publishing, 2019.

6. Flanagan, C. and Freund, S. N.: 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, PASTE '10, pages 1–8, New York, NY, USA, 2010.

7. Pozniansky, E. and Schuster, A.: Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. Concurr. Comput. : Pract. Exper., 19(3):327–340, 2007.

8. Flanagan, C. and Freund, S. N.: Fasttrack: Efficient and precise dynamic race detection. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, pages 121–133, New York, NY, USA, 2009.

9. Diep, T.-D., Trung Pham, K., Fürlinger, K., and Thoai, N.: A time-stamping system to detect memory consistency errors in mpi one-sided applications. Parallel Computing, 86:36–44, 2019.

10. Abadi, M., Flanagan, C., and Freund, S. N.: Types for safe locking: Static race detection for java. ACM Trans. Program. Lang. Syst., 28(2):207–255, 2006.

11. Boyapati, C. and Rinard, M.: A parameterized type system for race-free java programs. SIGPLAN Not., 36(11):56–69, 2001.

12. Naik, M., Aiken, A., and Whaley, J.: Effective static race detection for java. SIGPLAN Not., 41(6):308–319, 2006.

13. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst., 15(4):391–411, 1997.

14. Schonberg, D.: On-the-fly detection of access anomalies. SIGPLAN Not., 24(7):285–297, 1989.

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

16. Chandy, K. M. and Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst., 3(1):63–75, 1985.

17. Yen, L. and Huang, T.: Resetting vector clocks in distributed systems. J. Parallel Distrib. Comput., 43(1):15–20, 1997.

18. Arora, A., Kulkarni, S., and Demirbas, M.: Resettable vector clocks. In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '00, pages 269–278, New York, NY, USA, 2000.

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

20. Java Grande Forum: Java Grande benchmarking suite. http://www.javagrande.org/, 2008. [Online; accessed 31/07/2019].

VITA

NAME	Tommaso Pozzetti
EDUCATION	M.S., Computer Engineering, Politecnico di Torino, 2019, Italy
	M.S., Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2019, USA
	B.A., Computer Engineering, Politecnico di Torino, 2017, Italy
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	2017 – IELTS examination (8.0)
	A.Y. 2018/2019 One year of study abroad in Chicago, Illinois, USA
	A.Y. 2015/2016 One year of study abroad in Shanghai, China
SCHOLARSHIPS	
Fall 2018	Italian scholarship for final project (thesis) at UIC
Fall 2018	Italian scholarship for students of the TOP-UIC project
Fall 2015	Italian scholarship for students of the POLITONG project
WORK EXPERIENCE	
03/17 – 06/18	Software Engineer, Intervieweb, Turin, Italy
	Designed and implemented, using PHP and Javascript, a new search feature for In-recruiting, by integrating it with the Elasticsearch engine. Helped the company during the process of transitioning to Amazon Web Services by designing part of the new infrastructure with focus on security and availability. Worked on the analysis and implementation of an automated software deployment strategy to maximize the benefits of cloud technologies and achieved CI/CD by designing pipelines exploiting Gitlab, Docker and AWS resources.