Wait-Free Global Virtual Time Computation in Shared Memory Time Warp Systems

Alessandro Pellegrini
DIAG, Sapienza Università di Roma
Via Ariosto 25 - 00185 Roma - Italy
Email: pellegrini@dis.uniroma1.it

Francesco Quaglia
DIAG, Sapienza Università di Roma
Via Ariosto 25 - 00185 Roma - Italy
Email: quaglia@dis.uniroma1.it

***Abstract*—Global Virtual Time (GVT) is a powerful abstraction used to discriminate what events belong (and what do not belong) to the past history of a parallel/distributed computation. For high performance simulation systems based on the Time Warp synchronization protocol, where concurrent simulation objects are allowed to process their events speculatively and causal consistency is achieved via rollback/recovery techniques, GVT is used to determine which portion of the simulation can be considered as committed. Hence it is the base for actuating memory recovery (e.g. of obsolete logs that were taken in order to support state recoverability) and non-revocable operations (e.g. I/O). For shared memory implementations of simulation platforms based on the Time Warp protocol, the reference GVT algorithm is the one presented by Fujimoto and Hybinette [1]. However, this algorithm relies on critical sections that make it non-wait-free, and which can hamper scalability. In this article we present a wait-free shared memory GVT algorithm that requires no critical section. Rather, correct coordination across the processes while computing the GVT value is achieved via memory atomic operations, namely compare-and-swap. The price paid by our proposal is an increase in the number of GVT computation phases, as opposed to the single phase required by the proposal in [1]. However, as we show via the results of an experimental study, the wait-free nature of the phases carried out in our GVT algorithm pays-off in reducing the actual cost incurred by the proposal in [1].**

I. Introduction

In the context of Discrete Event Simulation (DES), a classical technique for achieving high performance model execution is Parallel-DES (PDES) [2]. It is based on partitioning the simulation model into several distinct objects, also known as Logical Processes (LPs), which concurrently execute simulation events on top of a parallel/distributed computing system. The core aspect in designing/developing PDES platforms is synchronization, the goal of which is to ensure causally-consistent (e.g. timestamp-ordered) execution of simulation events at each LP. To cope with this aspect, several synchronization protocols have been proposed, among which the optimism-oriented ones, such as the Time Warp protocol [3], are recognized to be highly promising.

In Time Warp, block-until-safe policies for event processing at the LPs are avoided, thus allowing speculative computation, which is reflected into great exploitation of parallelism. At the same time, causal consistency is guaranteed through rollback/recovery techniques, which restore the simulation model to a correct state upon the a-posteriori detection of consistency violations. These are originated when LP_a schedules a new event destined to LP_b having a timestamp lower than the one of some event already speculatively processed by LP_b. In case this occurs, the rollback of LP_b might also require undoing the send operation of events that were produced by LP_b during the rolled back portion of the computation. This is usually achieved via so called anti-messages (carrying anti-events), which are aimed at annihilating the originally sent events, thus possibly causing cascading rollbacks across chains of LPs.

A core abstraction underlying Time Warp-based platforms is Global Virtual Time (GVT), which is defined as the smallest timestamp among those of events (or anti-events) that are still unprocessed, or that are currently being processed. It indicates the commitment horizon of the speculative simulation run (in fact no LP can ever rollback to simulation time preceding the GVT value [3]), and is used both to execute actions that cannot be subject to rollback, such as displaying/inspecting intermediate simulation results (see, e.g., [4], [5]), and to recover memory (see, e.g., [6]) ([1]).

Computing the GVT value (e.g. on a periodic basis) requires the employment of some GVT algorithm, and different algorithms have been devised (see, e.g., [7], [8]) depending on the specific features of the underlying platform (e.g. shared vs distributed memory) hosting the Time Warp system. For shared memory platforms, the reference GVT algorithm is the one provided by Fujimoto and Hybinette in [1]. This work exploits the so called *observability* property, commonly matched by shared memory implementations of Time Warp, in order to provide a GVT algorithm that does not rely on any message acknowledgment scheme.

Essentially, in an observable Time Warp system, the send operation of any message/anti-message leads to the direct incorporation of the sent information into the message-

[1]Events with timestamps lower than GVT will never need to be re-executed after a rollback, therefore they can be discarded. The same happens to obsolete state information, if any, maintained to support recoverability of the LP state. The action of recovering memory after GVT calculation is typically referred to as *fossil collection*.

queuing data structure of the receiver, which removes the notion of in-transit message. This feature is exploited in [1] to devise a GVT algorithm relying on a single phase in which, once aware of a new GVT computation request, each process simply (A) keeps track of the minimum timestamp of any message/anti-message it sent out (towards any other process) and (B) after incorporating into its event-queue any message/anti-message detected as incoming in its message-queuing data structure, it contributes to the GVT computation by writing its so called *local minimum*, namely the minimum across the timestamps of its sent out and incoming messages/anti-messages, into a proper memory location. The last process that ends the above tasks is also in charge of computing the *global minimum* (namely the GVT value to be adopted) across all the local minima.

In this approach, the start of the GVT computation phase takes place by atomically setting a GVT-flag to the value N, which corresponds to the number of participating processes. The GVT-flag is then decreased by one each time a process ends the activities in the above task B, and is atomically checked to have reached the value zero in order to trigger the computation of the global minimum value. Overall, task B is executed within a critical section (which possibly includes the actual computation of the global minimum), namely in a sequentialized fashion, in order to guarantee correctness and progress of the GVT algorithm. However, such a critical section may represent an impairment to scalability.

In this article we propose a GVT algorithm still coping with observable shared memory Time Warp systems, but which does not require any critical section, hence standing as a *wait-free algorithm* [9]. Our proposal is based on memory atomic operations, namely compare-and-swap, which are used to keep track of the advancement of any thread within the different phases of GVT computation. In fact, as opposed to [1], our GVT algorithm requires multiple phases (rather than one), but given the wait-free nature of any task carried out in any of its phases, our proposal can reduce the actual cost (and also the latency) for computing the new GVT value especially for deploys on non-minimal scale multi-core systems. Overall, our proposal stands as an alternative to the GVT algorithm in [1], which is aimed at better coping with scalability aspects of Time Warp platforms to be run on top of multi/many-core shared memory computing systems. Also, its wait-free nature allows our GVT algorithm to better cope with contexts where the computing system can be shared across Time Warp processes and other kinds of workload that may interfere on CPU usage, thus possibly stretching the time-span of the critical section required by the algorithm in [1] (e.g. in case a Time Warp thread currently running the critical section is context-switched off the CPU). With our proposal, no wait-phase is induced across Time Warp threads in case some or more of them are context-switched off the CPU while GVT computation is in progress, in fact all the other threads can continue processing simulation events.

We have implemented our GVT algorithm into the open source ROOT-Sim speculative PDES platform [10], exactly based on the Time Warp paradigm, and we have also performed tests assessing the effectiveness of the algorithm when running this platform on top of a 64-bit NUMA machine equipped with 32 CPU-cores using a version of the well-known PHOLD benchmark [11] as the application test-bed.

The remainder of this paper is organized as follows. In Section II we discuss related work. Our wait-free GVT algorithm is presented in Section III. Experimental results are provided in Section IV.

II. Related Work

GVT algorithms can be barely divided in two categories, depending in whether they can cope with distributed memory systems, or require tightly coupled nodes in a shared memory platform.

As for the first category, several proposals have been based on explicit message acknowledgment schemes [12], [13], [14] in order to determine which messages (or anti-messages) are still in transit and which processes are responsible for keeping into account the timestamps of in-transit messages while computing the new GVT value. Some of these algorithms (see, e.g., [12], [14]) opt for acknowledging individual messages, which reduces the time interval along which a message can result as still in-transit. On the other hand, other approaches (see, e.g., [13]) opt for acknowledging batches of messages (rather than individual ones) which allows for reducing the overhead due to acknowledgment messages, but stretches the interval of time along which a message still results in-transit (although being potentially already processed at the destination). This, in its turn, leads to worsening the approximation provided by the algorithms on the actual GVT, given that "obsolete" timestamps might be still considered in the global reduction while computing the new GVT value.

An approach where explicit message acknowledgments are not required has been provided in [7]. In this solution, messages are associated with kind of "phases" (represented by different message coloring schemes) so that it is possible for the processes in the system to determine whether the timestamp of any message (or anti-message) needs to be accounted for in the current GVT computation. However, this algorithm requires control messages to set-up the start of a new GVT computation.

The need for both control messages and acknowledgments is removed by the proposal in [8], which has been tailored to distributed memory clusters where specific bounds can be assumed on the message delivery transfer across the nodes, and the clocks of the different machines can be assumed to be (perfectly) synchronized. In this proposal, new GVT computations are triggered by specific timeouts, which occur

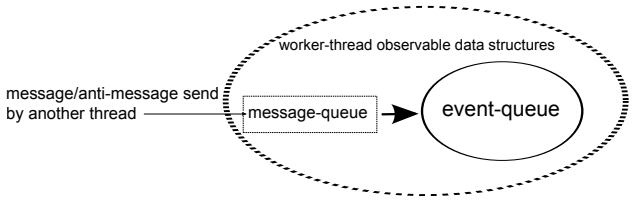

Figure 1. Data structures and send operations in observable Time Warp systems (the unique event-queue is the logical collection of the corresponding event-queues of the LPs managed by the worker-thread).

in synchronized way across all the nodes in the system. This gives rise to the scenario where all the nodes observe the start of the GVT algorithm at the same identical time instant, and are able to determine which messages (or anti-messages) can be still in transit since the start of the current GVT computation, given the knowledge on the upper bound delivery delay.

Differently from all the above proposals, we target shared memory Time Warp systems. Hence our GVT algorithm copes with orthogonal settings.

As pointed out before, for the case of tightly coupled shared memory systems, the reference GVT algorithm is the one by Fujimoto and Hybinette [1]. This algorithm requires the Time Warp system to be observable, a property that we have already pointed out, which expresses that no message (or anti-message) can ever be in-transit, given that the corresponding send operation leads to directly incorporating the message into the recipient message-queue. In this algorithm, the start of the GVT computation phase is instantaneously visible to all the processes, given that it simply requires setting the value of a flag into shared memory, namely a counter, to the number of participating processes. However, as we already pointed out, the computation of the local minimum at each process and the decrease of the counter in order to indicate that the contribution by the process has been made available, are executed within a critical section, which may represent a major impairment to scalability and resilience to interfering external workload (which may impact the duration of the critical section). In our approach we avoid any critical section, by trading-off in a completely different way synchronization costs and the number of phases required to compute the new GVT value.

Similar considerations apply to the work in [15], which presents a GVT algorithm for observable shared memory Time Warp systems (although observability was formally defined later in literature by [1]). In particular, this work is based on a critical section that is used to atomically update the entries of an array of elements, with size equal to the number of participating processes/threads. This leads the algorithm not to be wait-free, a property which is instead guaranteed by our proposal.

III. The Wait-Free GVT Algorithm

A. Algorithm Overview

In classical implementations of Time Warp, each worker-thread running within the simulation platform is in charge of managing a set of LPs. Particularly, it is in charge of handling the event-queues of these LPs, which are used to keep all the events that have been scheduled for them. In shared memory Time Warp versions (see, e.g., [16], [17]) the worker-thread is also associated with an input messaging data structure (the message-queue), where messages/anti-messages incoming from other worker-threads are directly buffered right upon the corresponding send operation (see Figure 1). The message-queue and the event-queue are accessible by the corresponding worker-thread in such a way that it can "observe" at any time the value of the timestamp of any message/anti-message that has been sent to the LPs it is managing. Hence, no sent message/anti-message is ever in-flight across worker-threads, thus being not accessible (in terms of ability to read its timestamp) by the destination worker-thread.

We consider the typical case where messages/anti-messages produced by any worker-thread are actually sent towards the destination while completing the current event processing loop. In this type of organization, the GVT value corresponds to the global minimum (across all the worker-threads) of the timestamps of messages/anti-messages that are either into the message-queue or that have already been incorporated into the event-queue. Hence, building a GVT algorithm actually means determining some right moment for the worker-thread to look at its data structures and to compute its local minimum, which will be then used for the calculation of the global minimum. With no loss of generality, in our approach the local minimum will be computed after having incorporated the already incoming messages/anti-messages into the event-queue, meaning that if an anti-message cancels a specific event, then the timestamps of both the canceled event and the anti-message will no longer have to be accounted for. This complies with observability operations as described in [1], while simplifying the computation procedure, given that the local minimum will correspond to the minimum value of the timestamps of events kept by the event-queue. Also, the incorporation of any message/anti-message is meant to leave the event-queue in a consistent state, with the meaning that if a message/anti-message incorporation leads some LP to be flagged for rollback, then the event-queue is refilled with the already processed events that need to be reprocessed after the LP rollback is finalized.

In our approach, we determine the right moment(s) for a worker-thfread to look at its data structures and compute the local minimum by having all the worker-threads participating in the GVT computation to pass through a sequence of different phases. No successive phase can be entered by any worker-thread unless all the other worker-threads have

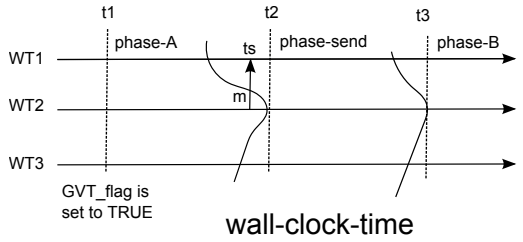

Figure 2. Phases example.

already executed the previous phase. An example of this type of behavior is shown in Figure 2, where we represent the start of the GVT algorithm as the atomic set of a special GVT-flag to the value $TRUE$, operation that occurs (at time t_1 in the example) as instantaneously visible to all the worker-threads, given the shared memory (and cache coherent) nature of our target platform. On the other hand, the conclusion of the different phases on different threads can occur at different time instants of wall-clock-time, as explicitly shown in the picture (although the actual system-wide end of the phase, and the begin of the successive phase, correspond to the latest wall-clock-time instant in which the end occurs on some thread).

Our algorithm is based on partitioning the computation of the GVT value in a sequence of phases according to which each worker-thread WT_i computes its local minimum two times, hence determining two values min_i^A and min_i^B. The actual local minimum provided by WT_i for the computation of the global minimum will then result as $min(min_i^A, min_i^B)$. Between the two computations of min_i^*, whose phases are referred to as phase-A and phase-B, we interpose an additional phase, marked as phase-send (see Figure 2). This phase is such that each worker-thread WT_i is requested to process at least one pending event (destined to some LP it is managing), if any, and to send newly scheduled events produced during such processing phase towards the destination worker-threads. phase-send starts right after all the worker-threads ended their tasks related to phase-A. For the example in Figure 2 this occurs at wall-clock-time t_2. We note that when the last one of the send operations by WT_i is performed while being in phase-send, any other message/anti-message previously produced by WT_i (destined to whichever worker-thread) is guaranteed to be already incorporated into the destination data structures (namely the message-queue of the destination worker-thread), given the intrinsic sequential nature of the activities at worker-thread WT_i and system observability.

At this point, indicating with $MINTS_i$ the minimum timestamp of any message/anti-message sent by WT_i up to the end of phase-send, we have the following two possibilities:

(A) $min_i^A \leq MINTS_i$, in this case min_i^A incorporates the lower bound on the logical time value that can be affected by any activity possibly occurring (or already occurred) at WT_i up to the end of phase-send.

(B) $min_i^A > MINTS_i$, in this case min_i^A does not represent the lower bound on the logical time value that can be affected by some activity occurred at WT_i.

However, given that any worker-thread WT_j recomputes min_j^B after phase-send is already over, all the messages/anti-messages sent by any worker-thread to WT_j up to the end of phase-send, and hence up to the end of phase-A, have been already incorporated into the data structures handled by WT_j (some of them might have been already processed, thus already belonging to the past of the computation). Hence, min_j^B represents the lower bound on the logical time value that can be affected by WT_j when also considering incoming information after phase-send is over. Therefore, $min(min_i^A, min_i^B)$ is the absolute lower bound on the logical time value that can be affected by the generic worker-thread WT_i after phase-send is over. By having each worker-thread WT_i writing $min(min_i^A, min_i^B)$ into a proper memory location, and then computing the absolute minimum across all the values kept by these locations we determine the GVT value. Such a computation is realized in our scheme via an additional phase, occurring after all the worker-threads have posted their local minimum into their associated memory locations (e.g. the entries of a shared array, each one associated with a specific worker-thread).

We note that, if the worker-threads were perfectly synchronized, thus computing their min_*^A values at the same identical time instant, then GVT would simply correspond to the global minimum across these values, given system observability. However, to avoid thread synchronization, min_*^A values are computed at different time instants, hence some message/anti-message might have not yet been incorporated into the message-queue of the recipient upon calculating its min_*^A, but might have been sent before the sender computes its min_*^A. An example is shown in Figure 2, where a message m is sent at wall-clock-time t_s from WT_2 to WT_1 after WT_1 already computed min_1^A, but before WT_2 computes min_2^A. The timestamp of this message/anti-message would therefore be missing in the global reduction. However, in our proposal, this timestamp gets recovered (if not yet belonging to the past of the computation) by having the worker-threads computing min_*^B when we are sure that any message sent by some worker-thread up to the end of phase-A is (or has been, if already processed) observable.

How to carry on the different phases in a wait-free manner and how to embed this GVT calculation scheme into a classical thread execution flow for Time Warp systems is discussed in the following section, where the pseudo-code of our algorithm is provided.

B. Algorithm Pseudo-code

To support the detection of the end of each phase, we use atomic counters. The startup of GVT computation is therefore handled according to what reported in Algorithm

1. It simply consists in setting the GVT_flag shared variable to the value $TRUE$ after having set to the value N (number of participating threads) five different atomic counters. The counters C_A, C_{send} and C_B directly map to the above presented algorithm execution phases, while the other atomic counters C_{aware} and C_{end} are used to identify the completion of two additional phases where the worker-threads actually become aware of the newly computed GVT value, so that the GVT algorithm is allowed to terminate and to be triggered again for a subsequent computation round.

Algorithm 1 GVT_INIT

$C_A = C_{send} = C_B = C_{aware} = C_{end} = N$; // shared atomic counters
$GVT\_flag = TRUE$; // shared flag

By the structure of Algorithm 2, in case GVT_flag is found to be set to $TRUE$, the generic worker-thread WT_i immediately ends its permanence in phase-A, which is done by computing min_i^A (see line 15), updating its local phase variable to $send$, and notifying to the other worker-threads that it ended this phase (see line 17). While executing the main loop, with the local phase variable set to $send$, as soon as all the threads have ended their execution in phase-A (namely C_A is found to be zero—see line 20), the thread is forced to execute at least one event, by also sending output messages/anti-messages (if any) towards the other threads. Then it again notifies the end of the current phase (by decrementing C_{send}—see line 25), and sets its local phase variable to B. When all the threads have done the same (hence C_{send} has reached the value zero), phase-B can start. At this point, each thread WT_i incorporates the incoming messages into the event-queue, and then computes min_i^B and its local minimum, which gets then stored in memory (see lines 28–31). It also decreases C_B and moves its phase to $aware$, so that when this counter reaches the value zero, all the local minima are already stored, and any worker-thread can compute the GVT (see lines 36–43), by also decrementing C_{aware}, so as to indicate awareness of the new GVT value. When this counter reaches the value zero, GVT_flag is reset via a compare-and-swap (CAS) atomic operation, which leads only one of the worker-threads to succeed in the reset task. At this point, the worker-threads trap into the code block in lines 5-10, hence re-initializing their phase to A (so as to allow the GVT algorithm to correctly restart for a subsequent round). When all of them have re-initialized their phase control-variable, the C_{end} counter reaches the value zero, so that the **if** condition in Algorithm 3 can be satisfied. Hence the GVT algorithm can be re-triggered for a subsequent round, which is done atomically via a CAS operation involving a global (shared) round-counter $current\_GVT\_round$, and a local one $local\_GVT\_round_i$. Only one worker-thread will be allowed to update the shared round-counter, also triggering the restart of the GVT algorithm. On the other hand, any

Algorithm 2 Main simulation loop (worker-thread WT_i)

$current\_GVT\_round \leftarrow 0$; //shared round-counter
$my\_phase_i \leftarrow A$; // thread local
$my\_GVT\_round_i \leftarrow 0$; //thread local

while (not end){
1 incorporate messages into event-queue;
2 execute next-event (if any);
3 send output messages/anti-messages (if any);

4 switch(GVT_flag){

5 **case** FALSE:
6 **if**($my\_phase_i = end$){
7 $my\_phase_i \leftarrow A$; //back to phase-A for next GVT round
8 $atomic\_dec(C_{end})$;
9 }
10 **break**;

11 **case** TRUE:
12 $my\_GVT\_round_i \leftarrow current\_GVT\_round$;

13 **if** ($my\_phase_i = A$){
14 incorporate messages into event-queue;
15 compute min_i^A;
16 $my\_phase_i \leftarrow send$; // entering phase-send
17 $atomic\_dec(C_A)$; // notify finalization of phase-A
18 **break**;
19 }

20 **if** ($my\_phase_i = send$ && $C_A = 0$){
21 incorporate messages into event-queue;
22 execute next-event (if any);
23 send output messages/anti-messages (if any);
24 $my\_phase_i \leftarrow B$; // entering phase-B
25 $atomic\_dec(C_{send})$; // notify finalization of phase-send
26 **break**;
27 }

28 **if** ($my\_phase_i = B$ && $C_{send} = 0$){
29 incorporate messages into event-queue;
30 compute min_i^B;
31 store $min(min_i^A, min_i^B)$;
32 $my\_phase_i \leftarrow aware$; // entering phase-aware
33 $atomic\_dec(C_B)$; // notify finalization of phase-B
34 **break**;
35 }

36 **if** ($my\_phase_i = aware$ && $C_B = 0$){
37 compute GVT as the global min of all stored local min;
38 $my\_phase_i \leftarrow end$; // entering phase-end
39 $atomic\_dec(C_{aware})$;//notify finalization of phase-aware
40 **if** (C_{aware} = 0){
41 $CAS(GVT\_flag, TRUE, FALSE)$;
42 }
43 **break**;
44 }

}//end switch
}//end while

Algorithm 3 GVT algorithm start (worker-thread WT_i) - this might be triggered by timeout

if $(GVT\_flag = FALSE$ && $C_{end} = 0$ && $CAS(current\_GVT\_round, local\_GVT\_round_i, local\_GVT\_round_i + 1))$
goto GVT_INIT;

worker-thread re-aligns its local round-counter to the shared one as soon as it becomes aware that the GVT algorithm has been started by some thread (see line 12 of Algorithm 2).

With this scheme, no worker-thread enters any wait phase due to delays in the processing of specific GVT-computation steps by any other thread (e.g. because of a context-switch off the CPU for any reason like, e.g., a page fault), which is what leads our algorithm to stand as a wait-free solution.

IV. Experimental Data

A. Experimental Platform

We have integrated the presented wait-free GVT algorithm within the ROOT-Sim simulation platform [10]. This is a C-based open source simulation package targeted at POSIX systems, which implements a general-purpose simulation environment based on the Time Warp synchronization paradigm. It offers a very simple programming model relying on the classical notion of simulation-event handlers (both for processing events and for accessing a committed and globally consistent state image upon GVT calculations), to be implemented according to the ANSI-C standard, and transparently supports all the services required to parallelize the execution. More in detail, we integrated the wait-free GVT algorithm in the symmetric multi-threaded version of ROOT-Sim that has been presented in [17]. This version adheres to the observability property of Time Warp systems given that message/anti-message exchange across different worker-threads is supported by directly en-queuing the sent information within a bottom-half queue, which is accessible by the destination worker-thread at any time. On the other hand, the partitioning of the message send/receive across top/bottom-half operations allows for high scalability of the simulation platform.

This platform has been run on top of a 32-core HP ProLiant server equipped with 64GB of RAM and running Debian 6 on top of the 2.6.32-5-amd64 Linux kernel.

B. Test-bed Application

This experimental study has been based on the PHOLD benchmark application [11]. Particularly, we have run the implementation of this benchmark presented in [18], which is based on homogeneous LPs that perform memory allocation/deallocation operations, across lists of buffers of different sizes, as well as memory read/write operations onto these buffers. In the configuration of the benchmark that we have used, each LP schedules for itself memory-deallocation events that, once processed, lead the LP to deallocate some buffer belonging to its state. On the other hand, when deallocating the buffer, a memory-allocation event is scheduled for some other LP in the system. The concept underlying this implementation/configuration of PHOLD is to have a stable value of the global amount of memory used to represent the state of the whole simulation model, by also having continuous variations of the amount of memory used for the local states of individual LPs. Also, scheduled simulation events tend to be clustered (along simulation time) across LPs exhibiting higher memory usage for their states. This is useful in our tests given that we may have short-lived execution phases where a few worker-threads may have a few events to process, which possibly increases the likelihood for these threads to compete for the access to critical sections (if any) used to support specific housekeeping operations, such as GVT computation (which is especially true when fixing the size of the simulation model, and running with larger numbers of worker-threads). This is therefore a good test case for evaluating wait-free implementations of Time Warp-suited housekeeping algorithms like the GVT algorithm we are presenting.

C. Results

We have compared the run-time behavior of our wait-free GVT algorithm (which we refer to as WF in the plots) with the one of the algorithm by Fujimoto and Hybinette (referred to as FH). The latter has been integrated within ROOT-Sim as an alternative to WF.

We have fixed the size of the used PHOLD model to 32 LPs (with a total amount of live memory for the corresponding states of the order of 1 GB — about 32 MB per-LP on the average), with read (resp. write) operations touching 20% (resp. 10%) of the current LP state size, and we have performed experiments in two different configurations.

In the first configuration, the computing platform has been reserved for the simulation runs, and we have varied the number of worker-threads used for running the PHOLD benchmark between 2 and the maximum number of available hardware-cores, namely 32. We recall that, having the size of the PHOLD model been fixed to 32 LPs, variation of the number of worker-threads towards the maximum value of 32 leads to scenarios with increased parallelism. Therefore we get an increased likelihood of the aforementioned phenomenon where, for short-lived phases, some worker-thread may not have simulation work to be performed (being the scheduled events temporarily clustered to occur on LPs hosted by other worker-threads, just depending on how the size of the memory used to keep the LPs' states varies over time in relation to deallocation/allocation operations occurring upon processing the events). For this experiment, we fixed the timeout for triggering a new GVT computation to 1 sec (this value looks reasonable for allowing prompt

recovery of memory, while not making GVT computation a predominant housekeeping task), and we measured the wall-clock-time required to complete the run of the PHOLD model in the selected configuration. The reported values refer to the average wall-clock-time observed over 10 different runs, all done with different pseudo-random seeds. However, the same seed has been used for the corresponding runs with the two different GVT algorithms so as to allow the same trajectory for the evolution of the simulation model when taking each individual wall-clock-time sample for the two algorithms. For FH, we also report the number of spin-lock tries per Wall-Clock-Time (WCT) unit experienced by the worker-threads while attempting to enter the critical section proper of this GVT algorithm. This parameter provides an indication on how the worker-threads tend to compete in the access to GVT-support structures of FH, in mutual exclusion. The outcoming results are shown in Figure 3, and we observe that, while increasing the number of worker-threads, WF allows for reducing the wall-clock-time required to complete the run up to 50% when compared to FH, a phenomenon which is strictly related to the higher overhead paid by FH in terms of CPU-time requested for running tasks related to GVT computation. In fact, as shown by the data on the bottom of Figure 3, as soon as the degree of parallelism increases (and the aforementioned short-lived phases with no simulation event to be processed at some worker-thread materialize), the likelihood of concurrent execution of the worker-threads in housekeeping mode increases, with consequent increase of the incidence of the critical-section access delay when running the FH algorithm. This is avoided by WF due to its wait-free nature.

We have also measured the execution time required to run the same identical PHOLD model (same code) in a serial fashion, by relying on a calendar-queue scheduler for storing the events and selecting the next-to-be processed one. It was on the order of 62 secs, which leads to the observation that the parallel runs provide speedup while increasing the number of worker-threads (up to about 10, which is achieved when running with 24 worker-threads and WF as the GVT algorithm). Hence, the presented results refer to the case of competitive parallel executions, which further strengthen the relevance of the improvements provided by our algorithm.

On the other hand, we note that wall-clock-time does not scale down when running with more than 24 worker-threads, which is due to two main reasons. One is that rollbacks tend to increase while increasing the level of parallelism, thus increasing the likelihood of performing non-useful work. Second, the aforementioned short-lived phases with no event to be processed at some worker-thread (which especially materialize when increasing the number of worker-threads) prevent fully exploiting the computing power offered by the underlying platform. However, by the data we see much higher resilience of the WF algorithm towards performance degradation phenomena caused by these "over-parallelism"

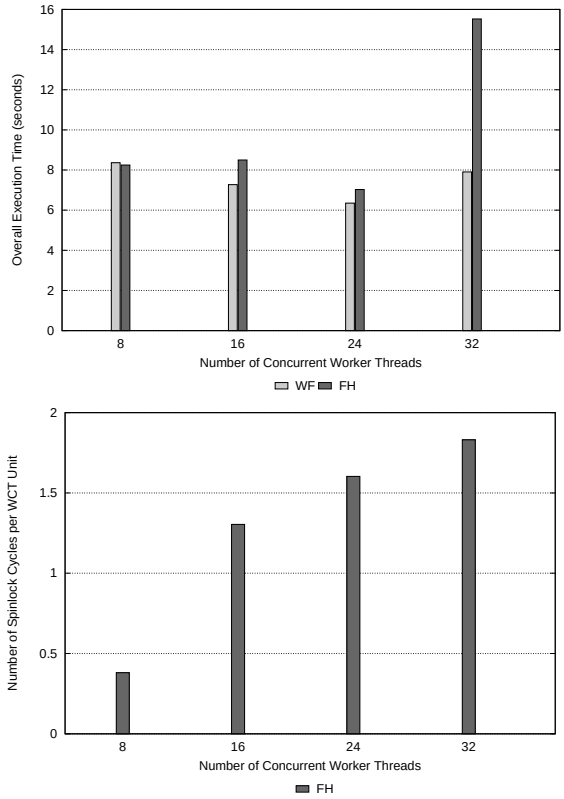

Figure 3. Results with the computing platform reserved for the simulation runs.

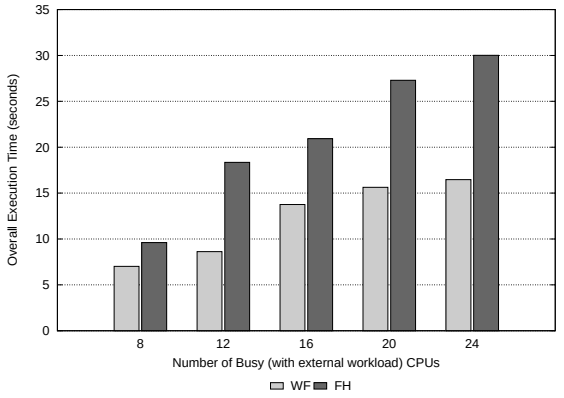

Figure 4. Results with interfering workload.

scenarios.

In the second configuration, we ran the same PHOLD model by fixing the number of worker-threads to 24 (which was the best parallelism level observed in the previous configuration). However, we injected external workload on the computing platform, which is made up by CPU-bound processes simply executing a busy loop. The number of such processes has been varied between 8 and 24. In this scenario, we have that ROOT-Sim worker-threads compete for CPU usage with the external workload (at least when the number of interfering CPU-bound processes oversteps

the value 8), a phenomenon which can give rise to delays in the completion of the critical section employed by FH in case the worker-thread running the critical section is context-switched off the CPU, and to consequent delays in the access to the same critical section by the other worker-threads. This experiment is therefore intended as a means to assess the resilience of WF (vs FH) to performance degradation in case of interference by external workload on the execution of the simulation application. By the data shown in Figure 4, we see that FH rapidly leads to performance degradation as soon as the number of interfering processes tends to increase. This phenomenon is noted also for WF. However, with this algorithm, the performance degradation is mainly expected to be generated by the reduced computing power exploited by the Time Warp system and by secondary costs, such as (a) those related to cache invalidation and refill in case of context-switch between a ROOT-Sim worker-thread and an interfering process, and (b) those related to the increase of rollback due to more skewed advancement in logical time of the different worker-threads in case of interference. In any case, WF still provides 30%-50% reduction of the wall-clock-time to complete the simulation run when compared to FH, a gain which is noted as soon as minimal interference by the external workload occurs.

V. Conclusions

In this article we have presented a wait-free GVT algorithm suited for shared memory "observable" Time Warp systems. We have also reported the results of an experimental study comparing the run-time behavior of our wait-free proposal with the state of the art GVT algorithm for shared memory Time Warp platforms, namely the one by Fujimoto and Hybinette [1]. By the experimental analysis we have observed a reduction of the overhead due to GVT computation by our wait-free proposal, and increased resilience to performance degradation when running with over-parallelism levels and/or with external workload interfering with the Time Warp simulation platform on the usage of hardware resources.

References

[1] Richard M. Fujimoto and Maria Hybinette. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, 7(4):425–446, 1997.

[2] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[3] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, 1985.

[4] Francesco Antonacci, Alessandro Pellegrini, and Francesco Quaglia. Consistent and efficient output-stream management in optimistic simulation platforms. In *Proceedings of the 2013 ACM Conference on Principles of Advanced Discrete Simulation*, pages 315–326, 2013.

[5] Diego Cucuzzo, Stefano D'Alessio, Francesco Quaglia, and Paolo Romano. A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation. In *Proceedings of the 11th International Symposium on Distributed Simulation and Real Time Applications*, pages 227–234, 2007.

[6] Samir R. Das and Richard M. Fujimoto. Adaptive memory management and optimism control in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 7(2):239–271, 1997.

[7] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[8] David W. Bauer, Garrett Yaun, Christopher D. Carothers, Murat Yuksel, and Shivkumar Kalyanaraman. Seven-o'clock: A new distributed GVT algorithm using network atomic operations. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation*, pages 39–48, 2005.

[9] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[10] HPDCS Research Group. ROOT-Sim: The ROme OpTimistic Simulator - v 1.0. http://www.dis.uniroma1.it/ hpdcs/ROOT-Sim/, October 2012.

[11] Richard M. Fujimoto. Performance of Time Warp under synthetic workloads. In *Proceedings of the Multiconference on Distributed Simulation*, pages 23–28. Society for Computer Simulation, 1990.

[12] Tel Gerard. *Topics in Distributed Algorithms*. Cambridge University Press, 1991.

[13] Yi-Bing Lin and Edward D. Lazowska. Determining the global virtual time in a distributed simulation. In *Proceedings of the 19th International Conference on Parallel Processing*, pages 201–209, 1990.

[14] Ten-Hwang Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[15] Zhonge Xiao, Fabian Gomes, Brian Unger, and John G. Cleary. A fast asynchronous GVT algorithm for shared memory multiprocessor architectures. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 203–208, 1995.

[16] Samir Ranjan Das, Richard Fujimoto, Kiran S. Panesar, Don Allison, and Maria Hybinette. GTW: a Time Warp system for shared memory multiprocessors. In *Winter Simulation Conference*, pages 1332–1339, 1994.

[17] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation*, pages 211–220, 2012.

[18] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Benchmarking memory management capabilities within ROOT-Sim. In *Proceedings of the 13th International Symposium on Distributed Simulation and Real Time Applications*, pages 33–40, 2009.