A Fine-Grain Time-Sharing Time Warp System

ALESSANDRO PELLEGRINI, Sapienza Università di Roma
FRANCESCO QUAGLIA, Sapienza Università di Roma

Although Parallel Discrete Event Simulation (PDES) platforms relying on the Time Warp (optimistic) synchronization protocol already allow for exploiting parallelism, several techniques have been proposed to further favor performance. Among them we can mention optimized approaches for state restore, as well as techniques for load balancing or (dynamically) controlling the speculation degree, the latter being specifically targeted at reducing the incidence of causality errors leading to waste of computation. However, in state of the art Time Warp systems, events' processing is not preemptable, which may prevent the possibility to promptly react to the injection of higher priority (say lower timestamp) events. Delaying the processing of these events may, in turn, give rise to higher incidence of incorrect speculation. In this article we present the design and realization of a fine-grain time-sharing Time Warp system, to be run on multi-core Linux machines, which makes systematic use of event preemption in order to dynamically reassign the CPU to higher priority events/tasks. Our proposal is based on a truly dual mode execution, application vs platform, which includes a timer-interrupt based support for bringing control back to platform mode for possible CPU reassignment according to very fine grain periods. The latter facility is offered by an ad-hoc timer-interrupt management module for Linux, which we release, together with the overall time-sharing support, within the open source ROOT-Sim platform. An experimental assessment based on the classical PHOLD benchmark and two real world models is presented, which shows how our proposal effectively leads to the reduction of the incidence of causality errors, as compared to traditional Time Warp, especially when running with higher degrees of parallelism.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems; D.4.1 [**Operating Systems**]: Process Management —*scheduling*; I.6.8 [**Simulation and Modeling**]: Types of Simulation — *discrete event*; *parallel*

General Terms: Theory, Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Optimistic synchronization

1. INTRODUCTION

Parallel Discrete Event Simulation (PDES) is a universally recognized methodology for speeding up the execution of (very) large/complex discrete event models via the exploitation of hardware parallelism [Fujimoto 1990a]. It is based on partitioning the model into multiple simulation objects, historically referred to as Logical Processes (LPs), whose events are concurrently dispatched for execution.

One core problem in PDES is how to ensure causally consistent (say, timestamp ordered) execution of the events at all the simulation objects, which is not trivial due to the dependencies that arise when different objects schedule events for each other.

Authors' addresses: A. Pellegrini, Dipartimento di Ingegneria Informatica, Automatica e Gestionale "Antonio Ruberti", Sapienza Università di Roma, Via Ariosto 25, 00185 Roma, Italy, email quaglia@dis.uniroma1.it; F. Quaglia, Dipartimento di Ingegneria Informatica, Automatica e Gestionale "Antonio Ruberti", Sapienza Università di Roma, Via Ariosto 25, 00185 Roma, Italy, email quaglia@dis.uniroma1.it

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

This is also known as the *synchronization* problem, for which different approaches and protocols have been provided in literature. The various proposals differ from each other by whether they entail (or not) the possibility to speculate along the simulated trajectory. If speculation is allowed, events are dispatched for processing at any simulation object as soon as they are available (at the underlying PDES environment) with no preliminary assessment of their safety. If causal inconsistencies arise, their effects are undone via rollback schemes. This is the Time Warp approach introduced in the seminal article by Jefferson [1985].

The relevance of speculative synchronization for PDES lies in that it allows for extremely high scalability. Recent results [Barnes-Jr. et al. 2013] have indeed shown how Time Warp systems exhibit the potential for scaling up to millions of processing units. However, on the down side, building Time Warp-based PDES platforms is far from being a trivial task because of two main reasons. One is related to the need for including the support for reversibility of the simulation model execution trajectory, an objective that should be pursued as transparently as possible to the overlying simulation application. Second, the actual performance delivered by the Time Warp-based PDES platform can be strongly affected by the rules according to which its worker threads CPU-dispatch the events to be processed.

The common literature trend is to build Time Warp systems as user-space platforms that are seen by the application-level code as run-time environments offering a specific API (e.g., for cross-simulation-object scheduling of events) and, in the most advanced cases (see, e.g., [Pellegrini et al. 2015]), providing application transparent support for reversibility of the actions performed by both the native application code and the invoked third party (standard) libraries. Invocations to the latter, or side effects on the simulation state natively produced by the application code, are in fact transparently intercepted by the platform-level code (via wrapping and/or instrumentation [Pellegrini et al. 2015; Rönngren et al. 1996; West and Panesar 1996]) which runs reversible versions of the corresponding tasks.

Nonetheless, another common way of implementing Time Warp PDES systems is the one where each CPU-dispatched simulation event is executed in *non-preemptable manner*. Consequently, the platform-level software is not allowed to re-evaluate CPU assignment until the completion of the last-dispatched event. This approach is not able to promptly react to the (system wide) dynamic generation and injection of events with higher priority, say lower timestamps, compared to the one currently being processed by some CPU-core. Consequently, it is not fully optimized given that the generation of rollbacks, and the associated waste of computation, tends to increase when events are CPU-dispatched and processed according to a rule that does not fully fit the priorities associated with the dynamic generation of timestamped events [Quaglia and Cortellessa 2002]. We note that the reduction of rollback incidence cannot be fully tackled by solely relying on load balancing/sharing strategies (see, e.g., [Carothers and Fujimoto 2000; Choe and Tropper 1999; Glazer and Tropper 1993; Vitali et al. 2012]) since they operate as long term planners for fruitful CPU usage, thus being not suited for "prompt" response to punctual variations of the event priorities along time.

Clearly, the ideal approach to preempt the execution of a CPU-dispatched event would be to interrupt the thread execution flow right upon the delivery of some higher priority event (or anti-event), destined to one of the simulation objects managed by the same thread. This would require a mechanism to reflect the arrival of a new event (say of a new message) into a change of the state of the CPU-core (e.g. the instruction

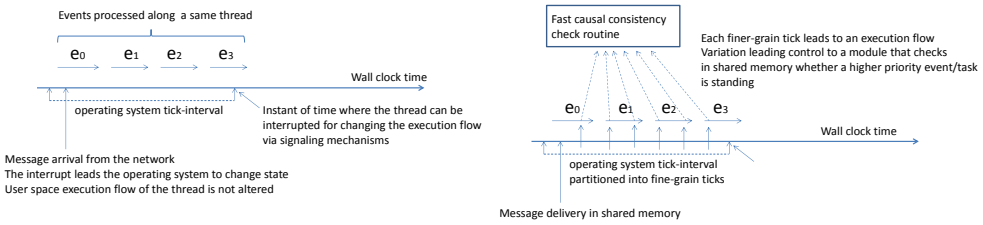

Fig. 1. Thread reactions to the injection of higher priority events.

pointer) so that the running thread can change its execution path, thus enabling the higher priority event to be actually dispatched[1].

For the case of network based message passing in distributed memory systems, the arrival of a message from a network interface is reflected into a change of the operating system state, which makes the message accessible on I/O channels (e.g. via polling). But the thread itself does not change its execution flow, except if asynchronous signaling mechanisms are adopted. However, these would operate according to the time-granularity of conventional timer-interrupt mechanisms, say 1 to 4 milliseconds on conventional operating system configurations. This is a granularity level that does not allow prompt preemption of events with common PDES workloads, where CPU requirements for processing simulation events are well below the order of milliseconds. An example scenario illustrating this problem is shown in Figure 1 (left).

A similar problem still appears for the case of Time Warp systems running on top of shared-memory platforms. In more detail, if user space shared-memory support is used for exchanging messages across the threads, a sending thread will only post the new message on a shared-buffer, which will be checked by the destination thread according to a polling mechanism operating before (or after) the processing of an event. In fact, pure shared-memory based communication provides no effective mechanism for interrupting the event execution at some destination thread right upon the post of the new message. Even if a signaling mechanism were used by the source thread, such as Posix user-defined signals, the time-granularity for the signal delivery to the destination thread would be still bound to the conventional operating system timer-interrupt interval configuration, thus resulting not adequate. Also, this approach would require the whole chain of signal management mechanisms to be passed through at the level of the operating system kernel, with consequent non-minimal overhead.

In this article we cope with preemptive events' processing in Time Warp systems to be run on shared-memory multi-core machines. Also, we target C-based application programming and Linux/x86-64 computing systems. Our solution overcomes the above depicted limitations by enabling the platform-level software to take back control and to re-evaluate CPU assignment with very fine grain period (e.g. on the order of tens of microseconds). To achieve this target we designed and developed an ad-hoc timer management Linux module which allows for (periodical) control flow variations along any running thread with no intervention by the chain of kernel-level mechanisms used for supporting Posix signals, hence leading to minimal run-time overhead. This is achieved by dividing each operating system tick assigned to the thread into sub-ticks, each one leading to an execution flow variation that brings control to a fast causality check routine implemented at the Time Warp platform level. The latter accesses compact data posted on shared-memory to determine whether the currently

[1]For the case of an incoming higher priority anti-event, the thread would dispatch the corresponding management operations, including the rollback of the target simulation object.

processed event is still the highest priority one (across those bound to the simulation objects managed by that same thread). In the negative case, the thread preempts the execution of the current event and switches to the execution of some higher priority task, according to a *fine-grain time-sharing* approach. The time-line of the execution of a thread with our approach is schematized in Figure 1 (right). Clearly, the higher the frequency of fine-grain ticks' delivery, the higher the likelihood of prompt switch to some higher priority task, if any. But the overhead associated with the management of fine-grain ticks should be anyhow kept to a minimum value. To cope with this issue, we also provide a benchmark program that can be used to configure the frequency of fine-grain ticks in the target computing platform where our fine-grain time-sharing Time Warp system would be installed.

In our proposal, CPU assignment is also re-evaluated right before returning control back to the application code after the execution of an event has trapped into a platform level service, either explicitly or because of interception (aimed at making some action by the application modules reversible). Overall, the return to application code from platform level execution and the fine-grain ticks are exploited in a synergistic manner to maximize the opportunities to preempt events if higher priority ones are delivered.

Our proposal does not create any bias in terms of CPU assignment across the threads (including kernel-level threads) running in the Linux system. In fact, the fine-grain tick mechanism we adopt does not alter the original operating system planning in terms of overall CPU time to be assigned to the worker threads running within the Time Warp PDES platform and to any other thread. This prevents impairing fairness when running our fine-grain time-sharing Time Warp system on top of a multi-user conventional platform.

Besides the ability to optimize CPU assignment depending on the (dynamic) priority of the tasks to be performed, our proposal has also the capability to address some specific liveness problems related to the speculative nature of Time Warp, such as application-level infinite loops that may arise when reaching an application non-admissible state due to out of order events' executions caused by speculation [Nicol and Liu 1997]. These loops can be (timely) broken thanks to our fine-grain time-sharing approach which can be exploited for supporting preemptive rollback operations leading to the squash of the non-admissible state trajectory.

The fine-grain time-sharing Time Warp architecture we have developed has been integrated within the open source ROOT-Sim package [2] [Pellegrini et al. 2011; HPDCS Research Group 2012], and operates in a fully transparent way to the overlying application code. Hence, the benefits from it come with no intervention by the application programmer. We also report experimental data for an assessment of our proposal, which have been collected by running three different test-bed applications—the PHOLD benchmark, a data store model and a personal communication system model—on top of a 32-core machine equipped with 64 GB of RAM.

While presenting our proposal, we assume the reader is already familiar with Time Warp concepts, and we refer the less familiar readers to, e.g., [Jefferson 1985; Jafer et al. 2013] for background information.

The remainder of this article is structured as follows. The fine-grain time-sharing Time Warp architecture is presented in Section 2. Experimental data are provided in Section 3. Related work is discussed in Section 4.

[2]Available at `http://github.com/HPDCS/ROOT-Sim`.

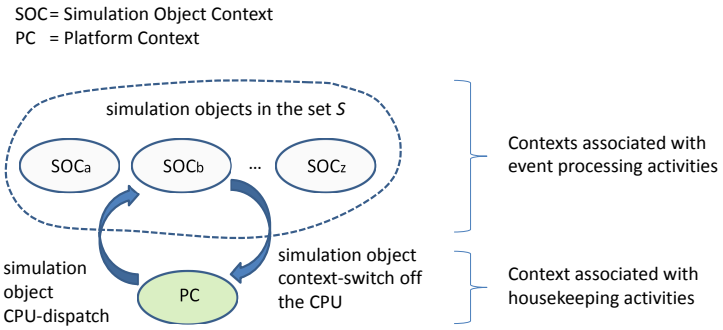

Fig. 2. Time-sharing Time Warp basics: execution contexts for an individual worker thread.

2. THE TIME-SHARING ARCHITECTURE

2.1. Basics

We assume the organization of the Time Warp PDES platform to adhere to the multi-thread paradigm, which has been recently shown to offer benefits (compared to counterpart implementations based on separate processes) for several aspects, such as the avoidance of simulation object migrations for well balanced usage of resources [Vitali et al. 2012] and optimized data-exchange [Wang et al. 2014]. With this type of organization, any subset S of the simulation objects is (temporarily) bound to a specific worker thread, which is in charge of managing the corresponding event queues (each one associated with an object) and of CPU-dispatching its bound objects for event processing. Further, all the worker threads share platform level data structures, which plays a central role in how our fine-grain time-sharing architecture handles the detection of event priority variations at run-time, as we shall discuss.

The basic organization of the fine-grain time-sharing Time Warp system is schematized in Figure 2. Each simulation object belonging to the set S managed by a given worker thread has its own execution context (e.g. its own stack). Additionally, a platform context is included, thus each worker thread operates, at any time instant, either in the context of some simulation object or in platform context. In the real implementation these contexts, including the switch between them, are managed by relying on context management functions inspired to classical `setjump` and `longjump` functions provided by the Posix API, whose detailed description is provided in the appendix.

When running in platform context, the worker thread carries on housekeeping tasks, such as the check for incoming events (anti-events) destined to the simulation objects it is currently managing, and the actual CPU-dispatch of the simulation objects. The latter operation takes place according to the Lowest-Timestamp-First (LTF) policy [Lin and Lazowska 1991], which leads the worker thread to CPU-dispatch the simulation object (belonging to its bound set S) whose next to be processed event is the one with the minimum timestamp among those already delivered, thus already known to exist.

Third party library functions accessible by the application code (e.g. `malloc`, `free` and `printf`) are transparently intercepted via wrapping schemes, which enables running the corresponding reversible instances supported by the run-time environment[3], conforming to what suggested by a few literature works (see, e.g., [Antonacci et al.

[3]In our view, reversibility also means that the behavior of the intercepted libraries is guaranteed to be piece-wise-deterministic, so as to allow optimized state restore schemes based on infrequent checkpointing and coasting forward.

2013; Rönngren et al. 1996]). The same is true for the case of application transparent code injection (say instrumentation) aimed at intercepting memory updates by the application code so as to make them reversible [Pellegrini et al. 2015; West and Panesar 1996]. The injected software brings control to platform level in a manner that is logically equivalent to the interception of external libraries' invocations by the application code. As for this aspect, in our fine-grain time-sharing Time Warp system, reversibility of the updates occurring within the (dynamically allocated) memory chunks forming the live state image for the simulation object is supported by relying on the checkpoint support offered by the DyMeLoR library [Toccaceli and Quaglia 2008]. In what follows we focus our discussion on the interception of external libraries, with the implicitly assumption that the discussion also covers scenarios based on application level instrumentation.

In our time-sharing architecture, each time one of the third party library functions is invoked, we say that the execution switches to platform mode, and then switches back to application mode as soon as the function returns. Clearly, when the worker thread operates in platform context, it also operates in platform mode. On the other hand, when it runs within the context of some simulation object, it can switch from application to platform mode multiple times, depending on the interactions between the application code and the intercepted external libraries. The wrapper that in our proposal encapsulates any intercepted function has the following structure:

```
return_type _function_name_wrapper (.. params ..){
            return_type ret;
            _enter_platform_mode;
            ret = function_name_reversible (.. params ..);
            _try_leave_platform_mode;
            return ret;
}
```

where the preamble `_enter_platform_mode` and the tail `_try_leave_platform_mode` are macros that set/unset a per-worker thread flag indicating the current running mode. Further, as we shall discuss, the `_try_leave_platform_mode` macro is also used for implementing part of the event preemption logic leading to switch the current execution context, if needed. This is the reason for the "try" prefix in the macro.

The switch between application and platform mode (and vice versa) occurs in our architecture not only because of synchronous invocations to intercepted external functions by the application level software (when running in the context of some simulation object). Rather, a timer-interrupt handler operating in user space is used to bounce control to platform mode periodically. We refer this handler to as **extra-tick-manager**, given that, as hinted, the time-sharing architecture leads a single operating system tick interval assigned to a worker thread to be partitioned into multiple fine-grain tick intervals just leading to extra-tick events for the target thread. Overall, the state diagram for any worker thread operating within the fine-grain time-sharing Time Warp system is the one depicted in Figure 3.

The execution of **extra-tick-manager** is triggered by the ad-hoc timer management logic we have embedded within our Linux module, which allows delivering fine-grain ticks to any worker thread running within the time-sharing Time Warp platform. The details of the implementation are provided in the appendix. For the abstraction level of the discussion in the main body of this article the important point is that a thread that wishes to be interrupted according to fine-grain ticks needs to register itself via an `ioctl` command on the `dev_extra_tick` file we support with our Linux module.

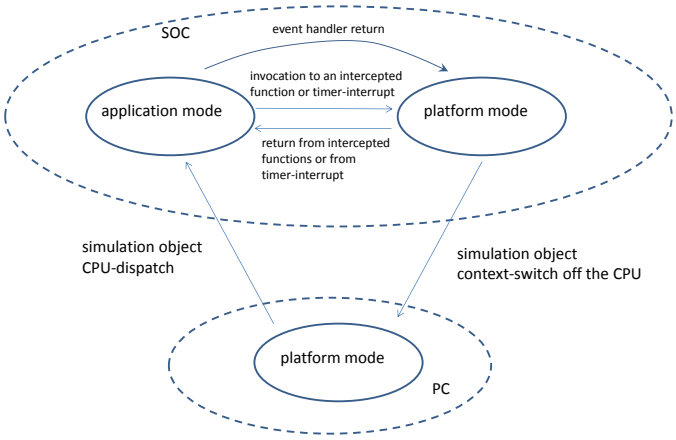

Fig. 3. Worker thread state diagram.

2.2. Run-time Detection of Priority Variations and Event Preemption Logic

As well known, when CPU-dispatching of the simulation objects in S is carried out by the worker thread according to the LTF algorithm, priority variations of the currently executed events (say, a decrease of the priority of the last CPU-dispatched event caused by the delivery of some event–or anti-event–with a lower timestamp destined to some object belonging to S) may only arise due to communication between simulation objects belonging to different subsets, say S and S', which are bound to different worker threads. Consequently, the architectural organization of the communication facility within the multi-thread Time Warp platform plays a relevant role in the run-time determination of the priority variation (if any) of the currently processed event.

As hinted we focus on shared-memory communication, and we consider a scenario conforming to the indications in [Vitali et al. 2012], where the exchange of messages/anti-messages across different worker threads does not take place by directly incorporating the corresponding information into the destination object event queue. Rather, messages are exchanged according to a top/bottom-half approach oriented to scalability. In particular, each worker thread manages a set of bottom-half queues (one for each simulation object belonging to the set S it is currently handling) such that any other worker thread in the system can notify the presence of new data to be ultimately incorporated into the destination object's event queue via the corresponding bottom-half queue. This is done via the execution of a top-half data record (tail) insertion into the bottom-half queue. Checking whether some new data is present into a bottom-half queue, and actual processing of the data with (timestamp-ordered) incorporation into the destination event queue, is carried out exclusively by the worker thread in charge of (currently) handling the destination object.

In our time-sharing organization of the multi-thread Time Warp system, the above scheme has been extended along the following lines, in order to support early detection of priority variations. First, each worker thread t has been associated with a BH_min_t record, which represents at any time instant the minimum timestamp of a message/anti-message that has been recorded in any of the bottom-half queues associated with the simulation objects that t is currently managing, since the last flush operation of these queues. In other words, BH_min_t represents the minimum value

among the timestamps of data in transit (if any), destined to some simulation object belonging to the set S handled by t.

This record is initialized to a special macro `INFINITE` when the worker thread t accesses its bound bottom-half queues and flushes the data into the corresponding event queues. Whenever a different worker thread inserts a bottom-half record into any of the bottom-half queues associated with the simulation objects managed by t, the reduction $BH\_min_t = Min(BH\_min_t, T)$ is performed, where T represents the timestamp of the message/anti-message that is being placed into the destination bottom-half queue. In our implementation, this reduction is performed via an atomic Compare-And-Swap (CAS) instruction. This allows manipulating BH_min_t while not requiring worker threads that concurrently access two distinct bottom-half queues associated with t to execute a conflicting critical section[4].

Another record, called $current\_time_t$, is associated with each worker thread t. It is used to keep track of the timestamp of the current simulation event, if any, that has been CPU-dispatched along t—this is the lowest-timestamp event according to LTF. The value of $current\_time_t$ is set to the special value -1 if thread t is not currently processing any event, which means that it is running housekeeping operations in platform context. The values of $current\_time_t$ and BH_min_t are used in combination to determine whether some higher priority task (compared to the one currently processed along thread t) needs to be CPU-dispatched. In particular, the platform level function that executes the check and determines whether some higher priority task needs to be executed along thread t, which needs therefore context switch between simulation object and platform contexts (thus enabling CPU reassignment via platform level actions), is structured as follows :

```
void check-and-switch()
1. if (current_time_t ≤ BH_min_t)
2.     return;
3. else
4.     _enter_platform_mode;
5.     switch_to_platform_context();
```

The above structure allows changing the current execution flow along thread t, by pushing it to platform-context (and also to platform mode, if not already operating with this mode), in case:

1) The simulation object currently dispatched for event execution along t needs to rollback, since it is the recipient of a message or an anti-message in its past—BH_min_t corresponds to the timestamp of a message/anti-message destined to the currently running simulation object. In this case the rollback operation will take place according to a preemptive mode.
2) Any simulation object belonging to the set S managed by t dynamically gains a priority higher than that of the currently running one, since it becomes the recipient of some message or anti-message with a timestamp lower than that of the last event that has been CPU-dispatched according to LTF. The case of an incoming anti-message is again representative of a causal inconsistent execution at the destination simulation object, given the adopted LTF rule for CPU-dispatching the events.

In either case, control must return to the Time Warp platform layer, so that the higher priority task (either a rollback operation or not) can be promptly executed. On

[4]In fact, each of them needs to temporarily lock a different bottom-half queue for data insertion, which helps not hampering concurrency [Vitali et al. 2012].

the other hand, if no higher priority task needs to be executed, the **check-and-switch** function simply returns control back to its caller. Clearly, if the simulation object that is context-switched off the CPU still runs long a consistent path, the preempted event will be resumed (with no loss of already performed work) when LTF will again find it as the highest priority one.

The last aspect to discuss is related to how the **check-and-switch** function is integrated with the other parts of the fine-grain time-sharing Time Warp support. In particular, how it is integrated with the `_try_leave_platform_mode` macro and with the **extra-tick-manager**, which is used to handle the interrupts delivered by the timer. The integration with `_try_leave_platform_mode` takes place as follows:

```
_try_leave_platform_mode
1. check-and-switch();
2. _leave_platform_mode;    //reset of the flag indicating platform mode running
                            //regular return from an intercepted function
3. return;
```

By the above pseudo-code, the **check-and-switch** function (possibly leading to context-switching off the CPU the currently running simulation object) is invoked upon the finalization of any application external function that has been intercepted by the corresponding wrapper, and is then executed in platform mode (for reversibility purposes). If the invocation to **check-and-switch** in line 1 leads to no switch to platform context, then the flag indicating whether we are running in application or platform mode is correctly aligned with the return to application mode.

As for the integration between **check-and-switch** and the timer-interrupt handling function **extra-tick-manager**, we have the following structure:

```
void extra-tick-manager()
1. if(platform_mode)
2.     return;     //already platform mode running - no control flow variation
3. else
4.     check-and-switch(); // do we need an execution flow variation?
```

If the timer-interrupt handler is activated while already running in platform mode, then no control flow variation needs to take place. In fact, if platform mode is currently associated with simulation object context, it means that the check on whether some higher priority task needs to be CPU-dispatched will be carried out right before returning to application mode via the `_try_leave_platform_mode` macro. If the current mode is not the platform one, then the handler triggers **check-and-switch** to verify whether higher priority tasks need to be carried out. This may lead to context-switching the currently processed event (hence the running simulation object) off the CPU.

The check in line 1 and the avoidance of the variation of the current execution flow if the worker thread is already in platform mode guarantee that whichever platform level block of code is executed along any worker thread as a non-preemptable action, which is a fundamental prerequisite. In fact, locks on data structures or memory regions might be acquired by some worker thread once the application has trapped into platform mode along that thread, which might be necessary in order to correctly manage the triggered service (see, e.g., [Pellegrini and Quaglia 2015]). Hence, the in place critical section cannot be context-switched off the CPU[5].

[5]A way to cope with the interruption of platform level code blocks would be to design the platform level software according to the concept of "safe places", which characterizes preemptable, e.g. real-time, operating system kernels. However, this type of design is aside of the core focus of our time-sharing proposal.

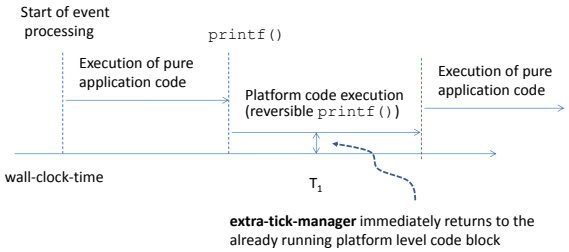

Fig. 4. Management of extra-ticks in the interleave between application and platform code blocks within an event processing wall-clock-time window.

A schematization of the behavior of our fine-grain time-sharing Time Warp system in relation to the delivery of timer-interrupts while already running in platform mode is provided in Figure 4, where we show the arrival of an extra-tick at wall-clock-time T_1, with consequent activation of the **extra-tick-manager**. In this scenario, the interrupt handler simply returns given that at the same time instant the thread was already running a platform-level reversible version of the `printf` function, via interception. However, the check on whether higher priority tasks would need to be dispatched by preempting the current event is anyhow carried out in our architecture as soon as the interrupted platform level function will attempt to return to application mode, which is done via the `_try_leave_platfrom_mode` macro.

2.3. Overall Configuration of the Time-sharing Support

By the architectural organization of the event preemption support described in Section 2.2, in our Time Warp system the platform level software has two different triggers for context-switching a simulation object off the CPU: (a) timer-interrupts and (b) returns from platform mode. However, while return from platform mode events are intrinsically related to the activities (say the execution profile) of the application level software (since they are triggered depending on the interaction between application level modules and the intercepted external libraries), timer-interrupts (and the cost/benefits they induce) depend on the configuration of the extra-tick interval. The shorter the length of the extra-tick period, which we denote as ΔET, the higher the expected overhead caused by timer-interrupts. However, shorter ΔET values can provide more opportunities for event preemption and prompt CPU reassignment to higher priority events/tasks, thus likely improving the effectiveness of the fine-grain time-sharing approach in reducing the amount of rollback.

To cope with the selection of ΔET and to optimize the synergy between the above two triggers for event preemptions, we devise the following scheme. We denote with $\widehat{\Delta ET}$ the minimum length of the extra-tick interval, which still induces negligible overhead due to extra-tick delivery to the Time Warp platform. As we shall discuss in Section 3, the value of $\widehat{\Delta ET}$ can be determined by running an ad-hoc benchmark in the early phase of the installation of our fine-grain time-sharing Time Warp system. Once determined $\widehat{\Delta ET}$, synergistic exploitation of the two different triggers for event preemption is based on run-time estimation of (i) the average event granularity for the specific application, which we refer to as Δ_e, and (ii) the average number of switches to (and then back from) platform mode while processing an individual event. We recall that these switches (if any) take place while running in simulation object context (see the state diagram in Figure 3). We denote such an average number of switches as $APMS$

(Application/Platform Mode Switches), hence the expression

$$T = \frac{\Delta_e}{APMS} \tag{1}$$

represents the average wall-clock-time interval after which the application software spontaneously provides control back to the platform software while an event processing phase is in place. The actual computation of Δ_e and $APMS$ has been based on the exponential mean of samples. The samples for computing Δ_e are taken by monitoring, via `gettimeofday`, the CPU time spent for processing individual events. The samples for computing $APMS$ are taken by counting the number of times the macro `_leave_platform_mode` is executed while processing any individual event[6]. Given that, according to the rules specified in Section 2.2, this already provides opportunities for CPU reassignment, the combination of the two different triggers for event preemption is based, in our design, on the relation between T and $\widehat{\Delta ET}$. Specifically, we dynamically switch on/off the extra-tick delivery along a thread depending on whether the following inequality is verified (or not)

$$\frac{T}{\widehat{\Delta ET}} \geq (1 + \alpha) \tag{2}$$

where α is a tunable parameter whose value falls in the interval $[-1, \infty]$. If α is set to the minimum value -1, then the Time Warp worker thread registers itself on the `dev_extra_tick` device file, thus being interrupted by the timer each $\widehat{\Delta ET}$ time units, independently of the simulation objects' run-time behavior (in terms of switches between application and platform modes). However, this settings might lead to bring control back to the platform level software excessively frequently (with respect to the benefits we would expect from CPU reassignment), especially for low values of T. Greater values of α would reduce the impact of this phenomenon. In fact, for $\alpha \to \infty$, Equation 2 would not be satisfied (except for $\widehat{\Delta ET}$ tending to zero, which is not realistic), leading the Time Warp worker thread to deregister itself from the `dev_extra_tick` device file, thus fully renouncing to be periodically interrupted for possible CPU reassignment.

A baseline settings for α could be represented by the value zero, leading the Time Warp thread to register itself as one to be extra-ticked with period $\widehat{\Delta ET}$ if the frequency according to which the execution of an event (taking place in simulation object context) switches to/from platform mode does not overstep the one of the extra-tick delivery. However, this setting would lead to reduced opportunities for event preemption if the switches to/from platform mode were not uniformly distributed along the lifetime of a simulation event. More conservative values of α, say in the interval $[-0.5, -0.1]$, would likely avoid this phenomenon. With this settings, even if T is less than $\widehat{\Delta ET}$, meaning that the execution in simulation object context spontaneously and frequently switches to platform mode, thus providing opportunity for event preemption, we still retain the possibility to achieve the same objective via the timer-interrupt scheme, which is done in order to avoid having a portion of the event processing interval uncovered by switches to platform mode.

As a last note, the quantity T in Equation 1 can be estimated at run-time on a per worker thread basis, so that each worker thread can operate its decision on whether to dynamically register or deregister itself as one to be extra-ticked (on the basis of the evaluation of Equation 2) independently of the other ones. This would allow coping with scenarios with simulation objects exhibiting different (heterogeneous) execution

[6]In our implementation the counter is directly updated by the macro upon its execution.

profiles, possibly giving rise to different Δ_e and/or $APMS$ values across the simulation objects' sets managed by the different threads.

3. EXPERIMENTAL RESULTS

3.1. Determining $\widehat{\Delta ET}$

As hinted, to determine the value of $\widehat{\Delta ET}$, an ad-hoc benchmark can be run during the early phase of installation of our fine-gain time-sharing Time Warp system on the target computing platform. Since PDES engines based on Time Warp are CPU-bound applications (given the absence of wait/block phases, at least in the presence of tasks/events to be actually carried on), our ad-hoc benchmark is simply made up by a multi-thread application where each thread executes a busy loop of a given duration. We initially run this benchmark in a baseline configuration with no registration of the threads on `dev_extra_tick`, hence with no extra-tick delivery. Then we run the benchmark again by registering the threads on `dev_extra_tick`, thus leading them to be periodically interrupted by the extra-tick logic. We run these experiments on a 32-core 64-bit HP ProLiant NUMA server, equipped with four 2GHz AMD Opteron 6128 processors (each one equipped with 8 CPU-cores) and 64 GB of RAM. The operating system is Linux SUSE, kernel version 3.16.7. This is the computing platform we used for all the experiments whose outcomes are reported in this section.

In the original configuration of the Linux kernel, the timer was set to issue an interrupt (a tick) each 1 millisecond. When running our benchmark we experimented with different values of the extra-tick interval achieved by scaling the original tick by a factor between 2 and 20, leading to experiment with extra-tick periods in the interval between 500 and 50 microseconds, hence being able to observe how the overhead caused by extra-tick delivery scales vs the length of the extra-tick period. In our benchmark, the extra-tick handler only increments a counter of delivered extra-ticks, and then returns control to the execution flow interrupted by the extra-tick arrival. This is aligned with the objectives of this benchmarking phase given that we only aim at evaluating the cost for delivering extra-ticks, independently of the usefulness of such a delivery (hence independently of any real action to be taken upon extra-tick arrival). We also varied the number of running threads between 1 and 32, thus studying how the overhead varies as a function of the number of threads managed according to the extra-tick logic supported by our Linux module.

We show in Figure 5 the inverse ratio between the execution time of the baseline configuration (no extra-tick), which we roughly report to be on the order of 30 seconds, and the execution time achieved with extra-ticks delivered to the application according to the selected scaling factor. Each sample has been computed as the average over 5 different runs of a same configuration; nonetheless, very minimal variation has been observed among the different sampled values. By the plot we see that the overhead induced by the extra-tick operating mode is less than 4% even when the scaling factor of the original tick interval (which we recall is of 1 millisecond) is set to the value 20, meaning that the extra-tick is delivered with granularity of 50 microseconds. Another interesting trend is that the overhead appears to be slightly higher when running the benchmark with larger number of threads. This is due to the slightly increased interference by common kernel level threads automatically started up by the operating systems (e.g. `kworker` threads), which is naturally induced when the benchmark tends to run on an increased number of CPU-cores. In fact, kernel level threads, although not being CPU-bound, lead anyhow to periodic operating system context switches, which in turn force our extra-tick management logic to more frequently interact with the timer, in terms of setting the requested interrupt period (depending on whether a `dev_extra_tick` registered thread, or not, is CPU-dispatched by the kernel).

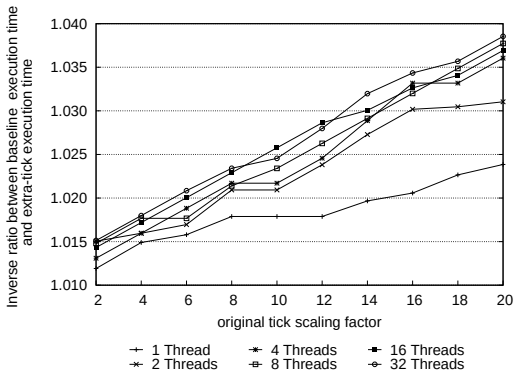

Fig. 5. Extra-tick overhead vs variations of both the extra-tick period and the number of threads.

Scaling factors of the original tick lower than the maximum value 20 lead the overhead by extra-tick delivery to be further reduced, at the expense of reduced opportunities for timer-interrupt based preemptions in the time-sharing Time Warp systems (in case of adoption of such lower scaling factors). Also, extra-tick interval length of 50 microseconds, beyond still providing minimal overhead, looks a suited value (in terms of opportunities for preempting an event currently being processed) when considering complex PDES workloads characterized by event granularity well above the order of a few (or a few tens of) microseconds. These workloads can be considered as typical targets for Time Warp synchronization, and more generally for classical PDES methodologies. We intrinsically target this category of workloads with our fine-grain time-sharing Time Warp proposal, at least for time-interrupt triggered preemptions. On the other hand, discrete event models with (very) fine grain events spontaneously bring control back to platform mode (after the CPU-dispatch of some event) in a prompt manner. Hence they naturally allow the platform to promptly react to priority variations even when events are processed in non-preemptable manner. Still, for these workloads, our time-sharing Time Warp system offers the possibility to preempt a CPU-dispatched simulation object by relying on switches back from platform mode.

Finally, the overhead determined via this benchmark can be considered a worst case reference value since the busy loop run by the threads is not interfered by factors such as memory access latency (and its variation as a function of locality of the accesses), which would tend to reduce the relative per-instruction cost of extra-tick delivery.

3.2. Performance Results with the PHOLD Benchmark

In this section we provide performance data that have been collected by relying on the well known PHOLD benchmark [Fujimoto 1990b]. The relevance of using PHOLD lies in that it entails events that are loosely coupled with the underling Time Warp platform. In fact, they are simple CPU busy loops, which lead to no invocation of application-external libraries possibly intercepted (for reversibility purposes) by the underlying platform. Therefore, event preemptions will never take place via switches back from platform mode. This kind of workload allows assessing the benefits from our fine-grain time-sharing Time Warp system in scenarios where preemptions can only be triggered by fine-grain timer-interrupts.

In order to improve the representativeness of this study, the PHOLD benchmark configuration we selected entails three different execution phases having the same virtual time duration, which we refer to as A, B and C. PHASE-A is lightweight, being it characterized by event duration of the order of 30 microseconds. PHASE-B is

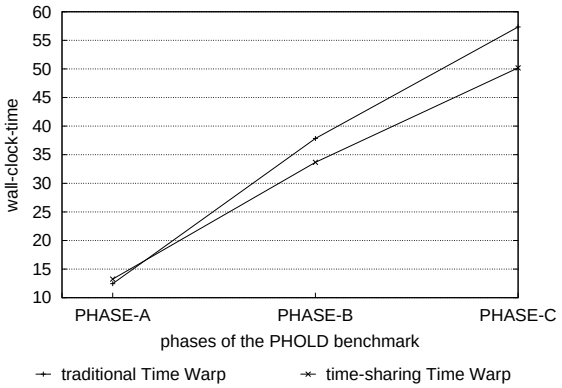

Fig. 6. Results with the PHOLD benchmark.

middleweight, being it characterized by event granularity of the order of 150 microseconds. Finally, PHASE-C is heavyweight, since the events have granularity of the order of 300 microseconds. We configured the benchmark with 2025 simulation objects connected as a bi-dimensional mesh, which have been equally distributed among 32 worker threads operating in the Time Warp system. Five events (say jobs) per simulation object have been inserted in the system. The jobs are routed randomly among neighbors by scheduling new events with exponential timestamp increment. This configuration of PHOLD (coupled with the selected level of parallelism in the underlying platform, say 32 threads) gave rise to a speculative execution pattern characterized by infrequent rollbacks undoing large numbers of events. The overall efficiency (say the ratio between the number of committed events, and the total number of processed events, say committed plus rolled back) that has been observed for the case of execution with a traditional configuration of the Time Warp system was on the order of 50%, with minor variations in the different execution phases.

In Figure 6 we show the execution time of the different phases of the PHOLD benchmark for the case of both traditional Time Warp and the time-sharing version[7]. Both these configurations rely on the same core PDES engine, namely ROOT-Sim, within which the time-sharing support has been integrated. The runs have been carried out by setting the extra-tick period to 50 microseconds. Also, the parameter α in Equation 2 has been set to the baseline value zero. This led time-sharing executions to have worker threads (dynamically) registered on the `dev_extra_tick` device file along PHASE-B and PHASE-C, but not along the lightweight PHASE-A. Hence, the time-sharing version behaves like a traditional one during this phase given that, as hinted, PHOLD events do not undergo switches to/from platform mode except for timer-interrupts. On the other hand, the two different platform configurations, time-sharing and traditional, were run with either the `dev_extra_tick` device file active or not, respectively. Therefore, the execution time of PHASE-A can help assessing the overhead by the device file logic, in relation to checking whether a thread that has been CPU-dispatched by the operating system kernel is a registered one or not, plus the overhead for managing simulation object contexts.

By the data we see that the time-sharing Time Warp version introduces about 4% overhead during PHASE-A, along which it provides no advantage due to absence of preemptions (since extra-ticks are not delivered during PHASE-A). However, when switching to PHASE-B and then to PHASE-C, the time-sharing version allows for in-

[7]All the reported samples have been computed as the average over 10 runs.

Table I. Ratio between the efficiency of traditional execution and the efficiency of time-sharing execution.

execution phase	$\frac{\text{efficiency of traditional Time Warp}}{\text{efficiency of time}-\text{sharing Time Warp}}$
PHASE-A	1.003
PHASE-B	0.935
PHASE-C	0.917

creasing reduction of the execution time. In particular, the time-sharing Time Warp system is between 9% and 11% faster than the traditional one. This happens thanks to the delivery of extra-ticks during both PHASE-B and PHASE-C, possibly triggering CPU reassignment to higher priority events/tasks, which leads the time-sharing version to reduce the amount of wasted computation. In particular, we show in Table I the ratio between the efficiency of the traditional execution, and the efficiency of the time-sharing execution in the different phases. By the data, time-sharing Time Warp provides about 7% better efficiency along PHASE-B and about 8% better efficiency along PHASE-C. This result, in combination with the particular rollback pattern with unfrequent but long rollbacks, allows for boosting the final performance gain. Such a gain is not only originated by the reduction of the amount of events that are eventually undone, but also by the reduction of rollback management costs, such as the cost for managing anti-messages, which may not scale linearly, e.g., for locality reasons[8].

3.3. Performance Results with a Data Store Model

As an alternative workload to PHOLD, in this section we consider a real world discrete event model of an in-memory key/value data store system. This type of models have recently become attractive (e.g. for capacity planning purposes) since real platforms based on this data storing paradigm have become a first class technology in modern (e.g. cloud based) infrastructures.

We simulated a distributed data store with 64 nodes, each one modeled by a separate simulation object, where data are partitioned and the partitions are distributed across the nodes. Batches of transactional data access requests are delivered to each node by proper simulation events (that are self-generated by the same simulation object modeling the node), which are scheduled following an exponential distribution of their timestamps. The transactions may entail accessing the local partition or remote partitions, and the access to remote data partitions leads to cross-simulation-object exchange of simulation events, carrying as payload the set of transactional requests that require access to the remote partition. The batching factor determines the actual workload to be simulated, hence the resource requirements for executing the simulation. We have considered two different configurations of this model, a lightweight configuration characterized by batching factor set to 10, and a heavy one characterized by batching factor set to 20. The transactional requests within each batch are processed (in the simulation) by having them managed via a round-robin scheme, resembling the assignment of real CPU resources in a multi-thread data management system. For the lightweight configuration, the average CPU requirement for simulating the event delivering the batch of transactions is of the order of 300 microseconds. Instead, the heavy configuration has CPU requirement of the order of slightly less than 500 microseconds. One primary objective of this type of simulation is the determination of

[8]In the configuration of PHOLD we have used, each undone event by a rollback operation requires sending a corresponding anti-message, which leads to costs for both send and receive tasks, including the cost for scanning output/input queues of the simulation objects, operations that lead to reduced locality especially for longer rollbacks. On the other hand, being the PHOLD benchmark an application with almost no state, the checkpoint/restore cost for the data structure representing the state of the simulation objects does not influence performance significantly vs variations of the amount of rollback.

the data store performance (e.g. its throughput) while varying the size of data partitions, the transaction access pattern (which may give rise to aborts depending on the materialization of data conflicts), and the locality of the accesses to the partitions.

In this study we still set the extra-tick interval to the value 50 microseconds, while selecting a more conservative value for the parameter α, which has been set to -0.5. This choice is motivated by the fact that this simulation model makes use of dynamic memory allocation/deallocation services for keeping the meta-data representing the transactional requests. Therefore it generates a non-negligible amount of interactions with the underlying Time Warp platform since the dynamic memory services are intercepted to make them reversible (as hinted, via the DyMeLoR library [Toccaceli and Quaglia 2008]). Since this already gives rise to switches to/from platform mode while processing the events, which provide opportunities for event preemptions, setting α to a conservative value allows us to still exploit opportunities for preemptions thanks to timer-interrupts. In fact, with such a conservative value, according to Equation 2, the worker threads operating in the time-sharing configuration of the Time Warp system register themselves on the `dev_extra_tick` device file.

Beyond being focused on a real world simulation model, this study complements the one with PHOLD for a few additional aspects. First, for the data store model we varied the degree of execution parallelism by varying the number of worker threads between 4 and 32. This has been done in order to compare traditional and time-sharing executions of the Time Warp system with different concurrency degrees, which in turn give rise to different amounts of rollback. Second, the type of interactions among the simulation objects in the data store model gives rise to a rollback pattern that is opposite to the one observed with PHOLD. In fact it is made up by frequent rollback occurrences, each one undoing a reduced number of events.

We report in Figure 7 (left side) the variation of the efficiency of the simulation run[9] while varying the number of worker threads for both traditional and time-sharing Time Warp systems. When running with low parallelism degree (say 4 worker threads), both the systems show relatively high efficiency, which is slightly less than 70%. The traditional version gives rise to a bit reduced efficiency limited to the case of the heavy model configuration. However while scaling up the degree of parallelism, the efficiency provided by the traditional Time Warp system rapidly degrades, falling just below 50% when running with 32 worker threads for both heavy and lightweight model configurations. Instead, the time-sharing version allows keeping the efficiency value significantly higher, leading to the order of 57% efficiency for the case of 32 worker threads, for both model configurations. Overall, the efficiency provided by the time-sharing Time Warp system stands up to 14% better than the one provided by the traditional Time Warp system. This advantage is reflected into a reduction of the simulation model execution time, as shown in Figure 7 (right side), which improves when increasing the degree of parallelism. Particularly, when relying on 32 worker threads the performance gain by the time-sharing Time Warp system is of the order of 15% for the heavy model configuration, and of the order of 11% for the lightweight model configuration. As compared to PHOLD, this time the advantage provided by the time-sharing configuration on the side of efficiency does not further boost in terms of final performance, which is due to the different rollback pattern. With more frequent rollback occurrences, the time-sharing Time Warp system has improved chances for early detection of (potential) causality violations. However, the gain by reducing the overall amount of rollback is essentially due to the avoidance of processing events that would be eventually undone, rather than to significant reductions of the cost for managing rollback phases (given that the rollback length is very short). Still, the gain by

[9]Also for this study we report average values over 10 different samples.

Table II. Ratio between the execution times of the two tested configurations of the heavy data store model with time-sharing Time Warp

number of used threads	$\frac{\text{execution time of the original configuration}}{\text{execution time of the varied configuration}}$
4	0.989
8	0.985
16	0.979
32	0.976

the time-sharing version, at the point of maximum parallelism, is significant. Also, the maximum speedup by the time-sharing Time Warp system compared to a sequential execution of the same data store models has been observed to be of the order of 20.

In Figure 8 we analyze the run-time dynamics of the time-sharing Time Warp system from a finer grain perspective. In particular, we report the number of event preemptions per wall-clock-time unit triggered either by timer-interrupts or by switches back from platform mode, and the sum of the two (marked as "total" in the plots). On the left side we show these data for the case of the original version of the data store model (heavy configuration), while on the right side we show the values achieved by running a modified version that is functionally equivalent, but where the instantiation of the meta-data for simulating the transactional requests does not take place following the round-robin scheme according to which the advancement of transactions execution is simulated. Rather, we instantiate these meta-data right upon starting the simulation phase of the whole transaction batch. This variant only anticipates the instantiation operation at the begin of the event that delivers the batch of transactions to be simulated. Hence, differently from the original version, the interaction with the underlying platform, which intercepts the dynamic memory management requests for instantiating the meta-data, are much more clustered along the execution phase of simulation events. By the data in Figure 8 we can see that the amount of preemptions triggered by timer-interrupts definitely increases in the second configuration, which also shows a reduction of the incidence of preemptions originated by switches back from platform mode. However, the important message that is convoyed by these data is that the total amount of preemptions per wall-clock-time unit is very similar in the two scenarios, independently of the number of used worker threads. This is a support to the robustness according to which timer-interrupts and switches to/from platform mode can be combined in complex workloads especially when relying on conservative values of the parameter α. In fact, by these results we see that the fine-grain time-sharing Time Warp system does not degrade its ability to early detect priority variations independently of the actual pattern of interaction between application and platform level software. Also, the version of the data store model with clustered allocation of meta-data for simulating the transactional requests has shown execution times very close to the ones observed for the case of the original version with either traditional or time-sharing configurations of the Time Warp system, which is somehow expected given that no relevant change in the actual run-time dynamics were induced. Relative performance values of the two versions of the data store model for the case of time-sharing executions are shown in Table II. Overall, the ability of our fine-grain time-sharing Time Warp system to robustly provide opportunities for event preemptions (as shown in Figure 8) is reflected into performance improvements independently of the interaction pattern between application and platform software.

3.4. Performance Results with a Personal Communication System Model

As a thirds alternative workload, in this section we consider a personal communication system model, say a real-world application that has already been used in a number of studies for assessing optimizations in PDES platforms (see, e.g., [Cingolani et al.

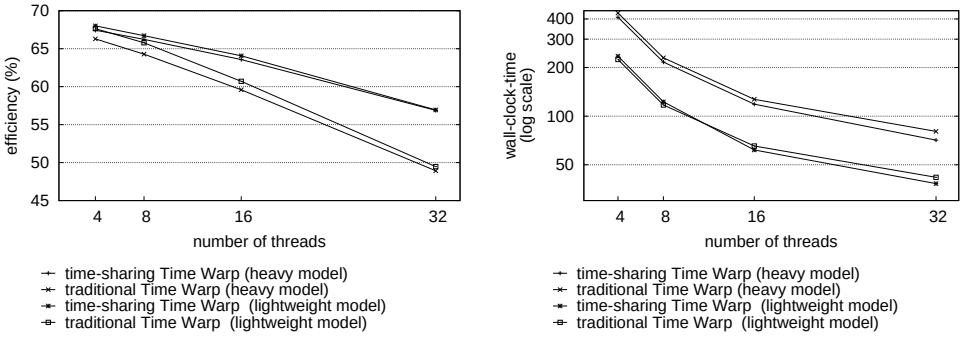

Fig. 7. Results with the data store model.

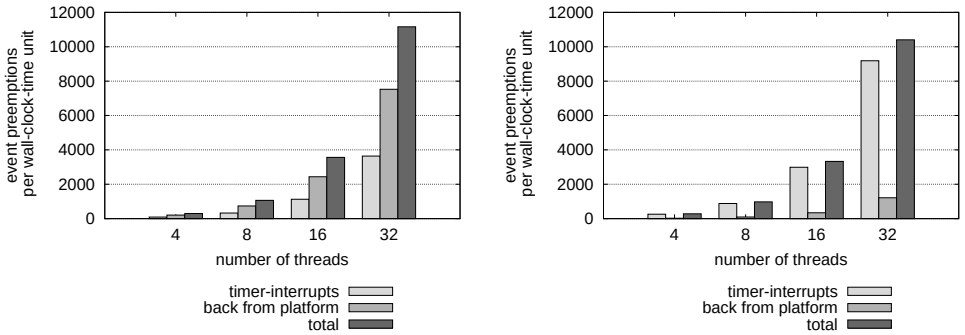

Fig. 8. Frequency of event preemptions for the heavy data store model - Original (left) and varied (right) memory allocation patterns.

2015]). In this application, each simulation object models a wireless cell—we selected a total number of 1024 cells organized into a hexagonal grid—each one managing 1000 wireless channels, which provide coverage to mobile devices in a squared region. The model represents the cells in high fidelity, addressing both interference across different channels within a same cell, and power management upon call setup/handoff. Particularly, the application handles power management simulation according to the results in [Kandukuri and Boyd 2002]. This application is also highly parameterizable by allowing the recalculation of fading coefficients and actual Signal-to-Interference Ratio (SIR) both on the occurrence of specific events (e.g. the startup of a call) and periodically (so as to account for, e.g., changes of weather conditions in the coverage area). Also, the inter-arrival of calls to mobile devices residing in the coverage area can be configured, thus leading to different values for the wireless channels' utilization factor. This, in its turn, affects both memory and CPU demand by the simulation given that higher utilization factors lead to the need for keeping more records (stored on dynamically allocated buffers) for simulating the concurrently active calls in any cell, and also to more costly operations for scanning and (possibly) updating these records. As a final preliminary note, the interaction across the different simulation objects takes place upon the occurrence of a handoff of a mobile device involved in an ongoing communication, in which case the wireless channel at the source cell is released, and a new one is attempted to be reserved at the destination cell.

In our experimentation we set the average residual residence time in the current cell for a mobile device involved in an on-going call to the 5 minutes, while the average call duration was set to 2 minutes. Both these parameters have been set to follow

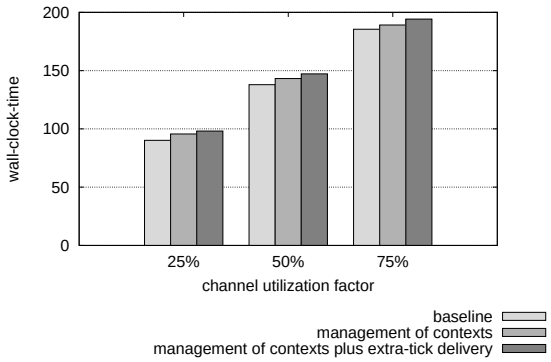

Fig. 9. Execution time with no cross-simulation-object events scheduling.

exponential distributions. Also, we have run this model with three different settings for the channel utilization factor, namely 25%, 50% and 75%, determined by different call inter-arrival rates, with balanced workload on all the simulation objects, and with periodic recalculations of the fading coefficients of active channels. This settings gives rise to variations of the average CPU requirement for simulation events from about 70/80 microseconds to about 150 microseconds.

For this study we still set the extra-tick interval to 50 microseconds, and α to the value -1, say to its lower bound. With this setting the worker threads operating within the time-sharing Time Warp system keep on being registered on the `dev_extra_tick` device file for the whole lifetime of the simulation in all the tested configurations (say for any value of the channel utilization factor). This leads to maximal exploitation of timer-interrupts for event preemptions. This choice is motivated by the fact that, unlike the data store model, the processing of an event in the personal communication system model leads to reduced interactions with platform level reversible implementations of memory allocation/deallocation services (since the number of buffer allocations/deallocations per event is much lower than the one characterizing the data store model). Hence, returns from platform mode can play a reduced role in triggering preemptions. On the other hand, compared to the data store model, the event processing routine shows a very different profile, much more based on floating point operations.

For this test-bed application we initially run a modified version, with the aim to assess the overhead imposed by the core facilities offered by the fine-grain time-sharing Time-Warp system, which enable systematic exploitation of event preemptions. These facilities are (i) the support for managing contexts, and (ii) the delivery of extra-ticks to the PDES platform. This overhead study is someway complementary to the one associated with PHASE-A of the execution of the PHOLD benchmark, since in that phase we only assessed the cost for managing contexts. In fact during that phase of the execution of PHOLD, the worker threads did not register themselves on the `dev_extra_tick` device file, hence no extra-ticks were delivered.

In order to assess the overhead by the aforementioned two facilities we run the personal communication system model by always enforcing a call to complete with the same wireless cell where it was originated. In this way, no interaction at all by the different simulation objects is ever generated, and events are processes by the worker threads always according to non-decreasing values of their timestamps. In such a scenario, the delivery of extra-ticks provides no revenue (since the event being processes will always represent the one with the highest priority bound to a given worker thread), just like the management of separate simulation object contexts (since no simulation object will be ever context-switched off the CPU while processing an

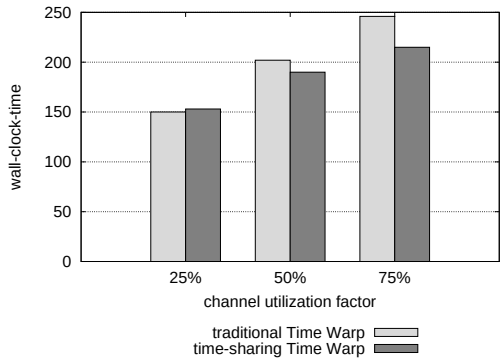

Fig. 10. Results with the personal communication system model.

event). Also, the absence of rollback in such scenarios allows us to assess the overhead by the two facilities with no interference by rollback management operations (which might lead to, e.g., changes in the locality of the execution due to the access to both checkpoints of the simulation object states and already processed event buffers). In Figure 9 we report the execution time values[10]—achieved with 32 worker threads—when excluding both the management of contexts and the delivery of extra-ticks, or when including these facilities. The former settings represents the common one for Time Warp systems not embedding the support for fine-grain time-sharing execution of the simulation objects. The reported data show how the overhead by the core facilities enabling fine-grain time-sharing is very limited, except for 25% channel utilization factor, case in which it reaches 7%. In fact, as soon as the event granularity (say the channel utilization factor) increases, we observe a decrease of the overhead, especially in relation to the management of contexts. This is somehow expected given that longer running events lead to reduced frequency of context switch operations across different simulation objects over time in scenarios with no actual preemptions.

In Figure 10 we show execution time results when reintroducing handoff events across cells, say cross-simulation-object scheduling of events. With this settings, the performance gain provided by the fine-grain time-sharing Time Warp system, compared to the traditional Time Warp execution, increases when increasing the channel utilization factor. The gain is of the order of 7% for the case of utilization factor set to 50%, and of the order of 13% when the utilization factor is further increased up to 75%. For channel utilization factor set to 25% we observe no relevant gain from time-sharing, just because of the reduced potentiality of extra-ticks exploitation (given the reduced wall-clock-time required for processing events in this configurations). This tren is confirmed by data we report in Figure 11, showing the variation of the amount of event preemptions per execution time unit achieved while running in time-sharing mode for the different configurations of the channel utilization factor.

For completeness, we also report in Table III the corresponding execution times for the case of a serial execution of the same identical application code on top of a sequential scheduler based on the Calendar-Queue data structure [Brown 1988], which allows determining the speedup of the parallel runs—hence whether the reported data refer to competitive parallel performance.

[10]Still based on the average over 10 runs.

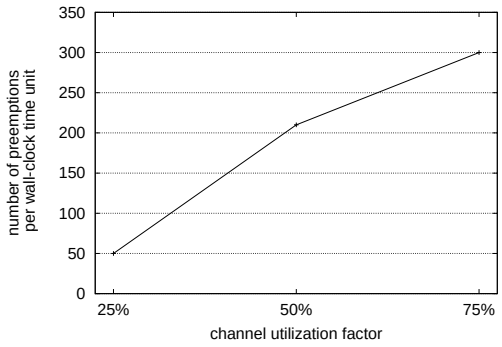

Fig. 11. Number of preemptions per wall-clock-time unit.

Table III. Performance of the serial simulator

	Channel Utilization Factor		
	25%	50%	75%
execution time (sec)	3500	5145	7610

4. RELATED WORK

Approaches Focusing on Performance vs Timer-interrupt Trade-offs. In the wide area of High-Performance-Computing (HPC), some literature studies exist on the relation between performance and timer-interrupt frequency. The common idea underlying most of the performance optimization proposals is that the lower the timer-interrupt frequency, the better the final delivered performance [De et al. 2007; Ferreira et al. 2008; Seelam et al. 2010]. The extremization of this approach led to defining *tick-less* operating systems (namely, with extremely reduced frequency of timer-interrupt) as the best configuration for hosting HPC applications. However, these studies have been tailored to the case of non-speculative processing, where the work carried out by any thread running on whichever CPU-core is ever useful, and there is no need to change the software execution flow (e.g. periodically) in order to optimize synchronization dynamics in terms of reduction of wasted computation, which is instead the case of speculative Time Warp systems. Also, the above studies have been tailored to evaluate the effects of the variation of the timer-interrupt frequency in contexts where the management of the timer-interrupt is still based on the native rules applied by the operating system kernel. In other words, the above proposals have been aimed at simply configuring the timer-interrupt behavior (limited to its frequency) in HPC contexts, not at introducing ad-hoc software modules for exploiting timer events, which is instead the approach we followed. In fact, our proposal puts in place a special (and lightweight) mechanism for handling timer-interrupts. Overall, our approach is completely different from the one dealt with by those literature studies, in terms of both reference scenario (speculative vs non-speculative processing) and architectural impact on the system organization.

In the context of speculative PDES systems, the only work we are aware of which deals with the relation between performance and timer-interrupt configuration is the one by Carothers [2002]. Here the author proposes an approach which is opposite to ours, where the Time Warp threads are allowed to take CPU control for longer periods (thus being not interrupted for a while) in order to be able to fully execute a simulation model with no interference by other workloads, and to deliver the output in real-time. This solution is still along the path of tick-less operating systems, with the difference that the tick-less behavior is triggered on-demand (namely whenever a

time-critical parallel simulation needs to be executed), hence it is not a static configuration of the underlying operating system. Our approach is fully orthogonal to this one because our target is the reduction of wasted computation, thanks to an appropriate periodic variation of the control flow along Time Warp threads. Also, while the proposal by Carothers [2002] is based on reserving the computing capacity for Time Warp programs—thus excluding the possibility for other tasks to be run on the system for a while—in our approach we do not create any bias in the usage of the computing system by Time Warp threads and other kinds of threads. We only allow the Time Warp threads to see their own ticks as partitioned into sub-intervals (with proper control flow management at the end of each sub-interval).

Approaches Targeting Preemptive Rollback. Our time-sharing Time Warp proposal supports preemptive rollback, a topic that has been somehow studied in literature, mainly in [Das et al. 1994; Santoro and Quaglia 2005]. The solution in [Das et al. 1994] targets parallel simulation on shared-memory machines, and is based on direct manipulation of the event list of the recipient simulation object by the thread along which the generation of a new event is handled. With this solution, the sender thread is able to determine the current simulation time of the recipient simulation object and whether any message/anti-message being sent to that object violates causality. If this is the case, the sender thread notifies the violation to the thread handling the recipient object, which is done to timely interrupt any in-progress activity in order to execute rollback operations. Our solution is different since it does not rely on cross-thread signaling. Also, in our approach, any Time Warp thread is allowed to change its current flow (and dynamically dispatch a different simulation event, or simulation object, after preempting the last dispatched one) independently of the materialization of a causality violation, but rather when any need arises to process a higher priority task, bound to a simulation object possibly different from the currently running one. This is done in order to reduce the likelihood of future rollback generation, not only to react via preemption to an already materialized causality violation. This is basically due to the fact that our fine-grain time-sharing Time Warp system is not limited to the support for preemptive rollback.

As for the preemptive rollback approach in [Santoro and Quaglia 2005], it is suited for distributed memory systems while we deal with shared memory multi-core machines. Also, it is based on polling, and the polling code to periodically verify causal consistency of the current event needs to be nested in the application code by the programmer. Instead, our proposal is fully transparent, and exploits back from platform mode and timer-interrupt events, rather than polling. Finally, similarly to [Das et al. 1994], the solution in [Santoro and Quaglia 2005] does not cope with control flow variations associated with the dynamic generation of higher-priority events (namely with timestamps lower than that of the event being executed along the thread) that do not directly give rise to a causality violation.

Approaches Based on Operating Systems Concepts. Dual-mode execution in Time Warp systems, which we exploit in our proposal, has been also investigated in [Pellegrini and Quaglia 2014]. In this proposal, when the worker thread runs in application mode, only a sub-portion of the whole address space is made accessible, namely the sub-portion keeping the memory layout of the CPU-dispatched simulation object. Any access to the state of another object generates a trap that gives control back to the platform code, which actuates proper thread synchronization mechanisms so as to allow cross-state processing of the events. Unlike [Pellegrini and Quaglia 2014], the present proposal is tailored to variations of the control flow in order to react to the generation of higher priority simulation events or tasks (such as rollbacks to be processed). Still, we retain application transparency just like [Pellegrini and Quaglia 2014].

Our proposal is clearly related to the work in [Jefferson et al. 1987], where Time Warp is instantiated as a special-purpose operating system destined to host discrete event applications to be executed according to the speculative processing paradigm. The core difference between what we are presenting and the proposal in [Jefferson et al. 1987] lies in that such an approach uses preemption only in case of causality errors affecting the currently-dispatched simulation event. Rather, we exploit preemption anytime a higher priority task needs to be processes, independently of the actual materialization of causality errors. Thus our solution also tends to anticipate the generation of causality errors.

Our proposal has also relations with recent approaches based on operating system scheduling to support virtual time synchronized advancement in emulation/simulation scenarios (see, e.g., [Lamps et al. 2015; Lamps et al. 2014; Yoginath et al. 2012]). These solutions provide scheduling policies of Linux Containers (LXCs) or Virtual Machines (VMs) allowing the emulated components to adjust their speed of operation in order to align it to the advancement of simulation time. This is typically achieved by scaling up/down the CPU capacity assigned to the different LXCs or VMs. Our solution is orthogonal to these approaches since we do not work at the level of the operating system scheduling policy. Rather, we customize the operating system management of timer-interrupts, so as to deliver them with fine granularity and at low cost to the speculative PDES simulation platform, so as to enable optimized CPU assignment within a fine-grain time-sharing scheme among multiple simulation objects run on top of a same thread.

Approaches Directly Targeting the Reduction of Rollback. Given that our core target is the reduction of the incidence of causality errors, our proposal is naturally related to literature solutions directly targeting rollback reduction in speculative PDES. We can roughly classify these works in two main categories: (a) the ones based on balanced resource usage (see, e.g., [Carothers and Fujimoto 2000; Choe and Tropper 1999; Glazer and Tropper 1993; Vitali et al. 2012]) and (b) the ones based on bounded optimism (see, e.g., [Dickens et al. 1996; Srinivasan and Reynolds 1998]). In the former case, the target is the one of reducing the skew in the advancement of simulation time at the different simulation objects, which is typically achieved via simulation objects' periodical migration (for balanced execution) across the Time Warp worker threads. As hinted, these proposals act as long term planners for CPU usage, and do not entail capabilities of reacting to punctual variations of the priority of the events, as instead we do via preemptive CPU reassignment. Overall, we can see the two approaches as orthogonal to each other, hence being ideally combinable. Finally, the solutions based on bounded optimism opt for artificially delaying the execution of events within the speculative processing scheme with the aim at increasing the likelihood of performing useful (not eventually rolled back) work. Some proposal (see, e.g. [Srinivasan and Reynolds 1998]) can even dynamically select per-event delays, thus attempting to reduce the incidence of rollback on a fine grain basis. We retain this same capability, but we still fully exploit the available computing power since we admit truly speculative preemptive event processing, with no artificial delay. Although different in spirit, we can still think of these two approaches as orthogonal and potentially usable in synergy.

5. CONCLUSIONS

It is typical that PDES platforms process simulation events in non-preemptive manner. For the case of Time Warp PDES systems, which exploit speculative processing and rollback techniques for causality maintenance, a preemptive approach would provide the possibility to dynamically reassign the CPU to events (or tasks, such as rollback operations) standing in the past logical time of the currently processed event. This would

allow for reducing the incidence of causality errors along the speculated execution path, and to more promptly react to the actual generation of the errors. To cope with this issue, have presented a fine-grain time-sharing version of a Time Warp system, which makes systematic use of event preemption just for the purpose of making the system run, at any time, those events/tasks that are dynamically determined to have the highest priority—they refer to past logical time values compared to the last CPU-dispatched ones. Our proposal is targeted at multi-core machines and Linux/x86-64 platforms. We integrated the fine-grain time-sharing Time Warp architecture, including the ad-hoc Linux module supporting timer-interrupt based preemptions, within an open source speculative PDES platform. Further, we have reported experimental data supporting the effectiveness of our proposal. Indications on how to configure the presented fine-grain time-sharing Time Warp system, in relation to core parameters driving its internal logic, have also been provided, which should help fruitful employment of our solution with workloads aside of the ones used in our experiments.

REFERENCES

ANTONACCI, F., PELLEGRINI, A., AND QUAGLIA, F. 2013. Consistent and efficient output-streams management in optimistic simulation platforms. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 315–326.

BARNES-JR., P. D., CAROTHERS, C. D., JEFFERSON, D. R., AND LAPRE, J. M. 2013. Warp speed: executing time warp on 1, 966, 080 cores. In *Proc. of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation'*. 327–336.

BROWN, R. 1988. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM 31*, 1220–1227.

CAROTHERS, C. D. 2002. Xsim: real-time analytic parallel simulations. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation, PADS 2002, Washington, D.C., USA, May 12-15, 2002*. 27–34.

CAROTHERS, C. D. AND FUJIMOTO, R. M. 2000. Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms. *IEEE Transactions on Parallel and Distributed Systems 11,* 3, 299–317.

CHOE, M. AND TROPPER, C. 1999. On learning algorithms and balancing loads in time warp. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*. Springer Verlag, 101–108.

CINGOLANI, D., PELLEGRINI, A., AND QUAGLIA, F. 2015. Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES. In *Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation, June 10 - 12, 2015*. 211–222.

DAS, S. R., FUJIMOTO, R. M., PANESAR, K., ALLISON, D., AND HYBINETTE, M. 1994. GTW: a time warp system for shared memory multiprocessors. In *WSC '94: Proceedings of the 26th conference on Winter simulation*. Society for Computer Simulation International, 1332–1339.

DE, P., KOTHARI, R., AND MANN, V. 2007. Identifying sources of operating system jitter through fine-grained kernel instrumentation. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing, 17-20 September 2007, Austin, Texas, USA*. 331–340.

DICKENS, P. M., NICOL, D. M., JR., P. F. R., AND DUVA, J. M. 1996. Analysis of bounded time warp and comparison with YAWNS. *ACM Transactions on Modeling and Computer Simulation 6,* 4, 297–320.

FERREIRA, K. B., BRIDGES, P., AND BRIGHTWELL, R. 2008. Characterizing application sensitivity to os interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. IEEE Press, Piscataway, NJ, USA, 19:1–19:12.

FUJIMOTO, R. M. 1990a. Parallel discrete event simulation. *Communications of the ACM 33,* 10, 30–53.

FUJIMOTO, R. M. 1990b. Performance of Time Warp under synthetic workloads. In *Proceedings of the Multiconf. on Distributed Simulation*. Society for Computer Simulation, 23–28.

GLAZER, D. W. AND TROPPER, C. 1993. On process migration and load balancing in time warp. *IEEE Transactions on Parallel and Distributed Systems 4,* 3, 318–327.

HPDCS RESEARCH GROUP. 2012. ROOT-Sim: The ROme OpTimistic Simulator - v 1.0. http://www.dis.uniroma1.it/ hpdcs/ROOT-Sim/.

JAFER, S., LIU, Q., AND WAINER, G. A. 2013. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory 30*, 54–73.

JEFFERSON, D. R. 1985. Virtual Time. *ACM Transactions on Programming Languages and System 7,* 3, 404–425.

JEFFERSON, D. R., BECKMAN, B., WIELAND, F., BLUME, L., LORETO, M. D., HONTALAS, P., LAROCHE, P., STURDEVANT, K., TUPMAN, J., WARREN, L. V., WEDEL, J. J., YOUNGER, H., AND BELLENOT, S. 1987. Distributed simulation and the time wrap operating system. In *SOSP*. 77–93.

KANDUKURI, S. AND BOYD, S. 2002. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications 1,* 1, 46–55.

LAMPS, J., ADAM, V., NICOL, D. M., AND CAESAR, M. 2015. Conjoining emulation and network simulators on linux multiprocessors. In *Proceedings of the 3rd ACM-SIGSIM Conference on Principles of Advanced Discrete Simulation, June 10 - 12, 2015*. 113–124.

LAMPS, J., NICOL, D. M., AND CAESAR, M. 2014. Timekeeper: a lightweight virtual time system for linux. In *Proceedings of the 2nd ACM-SIGSIM Conference on Principles of Advanced Discrete Simulation, May 18-21, 2014*. 179–186.

LIN, Y.-B. AND LAZOWSKA, E. D. 1991. Processor scheduling for Time Warp parallel simulation. In *Proceedings of the 23rd SCS Multiconference on Advances in Parallel and Distributed Simulation*. IEEE Computer Society, 11–14.

NICOL, D. M. AND LIU, X. 1997. The dark side of risk (what your mother never told you about time warp). In *Proceedings of the Eleventh Workshop on Parallel and Distributed Simulation, PADS '97, Lockenhaus, Austria, June 10-13, 1997*. 188–195.

PELLEGRINI, A. AND QUAGLIA, F. 2014. Transparent multi-core speculative parallelization of DES models with event and cross-state dependencies. In *SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS '14, Denver, CO, USA, May 18-21, 2014*. 105–116.

PELLEGRINI, A. AND QUAGLIA, F. 2015. NUMA time warp. In *Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation, London, United Kingdom, June 10 - 12, 2015*. 59–70.

PELLEGRINI, A., VITALI, R., AND QUAGLIA, F. 2009. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 45–53.

PELLEGRINI, A., VITALI, R., AND QUAGLIA, F. 2011. The ROme OpTimistic Simulator: Core internals and programming model. In Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques. *Proceedings of the 4th ICST Conference of Simulation Tools and Techniques (SIMUTools)*.

PELLEGRINI, A., VITALI, R., AND QUAGLIA, F. 2015. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Paralle and Distributed Systems 26,* 6, 1560–1569.

QUAGLIA, F. AND CORTELLESSA, V. 2002. On the processor scheduling problem in time warp synchronization. *ACM Transactions on Modeling and Computer Simulation 12*.

RÖNNGREN, R., LILJENSTAM, M., AYANI, R., AND MONTAGNAT, J. 1996. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 70–77.

SANTORO, A. AND QUAGLIA, F. 2005. Software supports for event preemptive rollback in optimistic parallel simulation on myrinet clusters. *Journal of Interconnection Networks 6,* 4, 435–457.

SEELAM, S., FONG, L. L., TANTAWI, A. N., LEWARS, J., DIVIRGILIO, J., AND GILDEA, K. 2010. Extreme scale computing: Modeling the impact of system noise in multicore clustered systems. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. 1–12.

SRINIVASAN, S. AND REYNOLDS, JR., P. 1998. Elastic time. *ACM Transactions on Modeling and Computer Simulation 8,* 2, 103–139.

TOCCACELI, R. AND QUAGLIA, F. 2008. DyMeLoR: Dynamic Memory Logger and Restorer library for optimistic simulation objects with generic memory layout. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 163–172.

VITALI, R., PELLEGRINI, A., AND QUAGLIA, F. 2012. Load sharing for optimistic parallel simulations on multi core machines. *SIGMETRICS Performance Evaluation Review 40,* 3, 2–11.

WANG, J., JAGTAP, D., ABU-GHAZALEH, N. B., AND PONOMAREV, D. 2014. Parallel discrete event simulation for multi-core systems: Analysis and optimization. *IEEE Transactions on Parallel and Distributed Systems 25,* 6, 1574–1584.

WEST, D. AND PANESAR, K. 1996. Automatic incremental state saving. In *Proceedings of the Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 78–85.

YOGINATH, S. B., PERUMALLA, K. S., AND HENZ, B. J. 2012. Taming wild horses: The need for virtual time-based scheduling of vms in network simulations. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, August 7-9, 2012*. 68–77.

APPENDIX

A. MANAGEMENT OF CONTEXTS

In the most general case, an *execution context* is the program state at a certain point of the lifetime of the application. This state is composed of the *CPU image* and *program variables*. The former entails all CPU registers, which allow to determine what is the next instruction to be executed (the program counter), and which keep values commonly computed by the currently executed function, and all control registers which allow the hardware architecture and the operating system to correctly interact with the available hardware resources (e.g., MMU-related registers to correctly drive a virtual-to-physical address translation). The latter entails all the variables kept in the data sections, in the heap, and in the stack.

In our time-sharing Time Warp environment, multiple contexts should be available at the same time within the same multi-threaded application. Moreover, the multi-threaded application should be able to switch from one execution context to another at any point. This means that there are no "safe points" in which one context could be saved, rather the context switch might happen after the execution of any machine instruction, e.g., since the fine grain timer interrupts we deliver are independent of the flow of the application's code. The contexts we need to manage cannot therefore be associated with multiple concurrent operating system's threads. Rather, they are logically bound to a single operating system's thread, and we rely on multiple User-Level Threads (ULTs), each one associated with a simulation object, which can be activated by any worker thread—namely, at any time a worker thread can "jump" to the execution flow of any ULT. Given that simulation objects' program variables stored in the head/data section are transparently managed by the multi-threaded nature of the simulation engine and state management facilities (see, e.g., [Pellegrini et al. 2009]), we rely on the following code snippet to setup a new execution context for a certain simulation object:

```
void context_create(LP_context_t *context, void (*entry_point)(void *), void *args,
                                                                    void *stack, size_t stack_size) {
struct sigaction sa;
    struct sigaltstack ss;
    struct sigaltstack oss;

    bzero((void *)&sa, sizeof(struct sigaction));
    sa.sa_handler = context_create_trampoline;
    sa.sa_flags = SA_ONSTACK;
    sigfillset(&sa.sa_mask);
    sigdelset(&sa.sa_mask, SIGUSR1);
    sigaction(SIGUSR1, &sa, NULL);

    ss.ss_sp = stack;
    ss.ss_size = stack_size;
    ss.ss_flags = 0;
    sigaltstack(&ss, &oss);

    context_creat = context;
    context_creat_func = entry_point;
    context_creat_arg = args;
    context_called = false;
    raise(SIGUSR1);
    sigaltstack(&oss, NULL);

    context_switch_create(&context_caller, context);
}
```

The `context_create` function stores the information related to this new execution flow (namely, the CPU context and the stack's location in memory) within the `LP_context_t` structure. To setup this new context, we rely on Posix signals raised by the worker thread which is in charge of setting up the context for a certain simulation

object. To allow the new context to live in a different stack, we rely on the Posix-compliant `sigaltstack()` API, which asks the underlying operating system to run a signal handler within a separate stack. This stack can be allocated using any memory allocator, and is passed to the `context_create` function as an argument. Then, the `context_create` function stores a function pointer (`entry_point`) and a pointer to a vector of arguments (`args`) both passed as arguments to the `context_create` function into two TLS variables. This allows to concurrently run the creation of multiple ULTs by multiple worker threads, which allows to reduce the overhead to startup the simulation.

The worker thread then issues a call to the Posix `raise()` API, to kill itself with the `SIGUSR1` signal. Given that the `context_create` function earlier posted the `context_create_trampoline()` function as the signal handler, control is passed to it. The source of the `context_create_trampoline` function is as follows:

```
static void context_create_trampoline(int sig) {
    (void)sig;

    if(context_save(context_creat) == 0)
        return;

    context_create_boot();
}
```

The behavior of this function is quite straightforward if we correctly catch what the `context_save()` call does. This function (which we shall discuss later) is similar in spirit to the traditional Posix `setjmp()` API. In particular, the purpose of this function is to store the whole CPU state of the running thread. We recall that this signal handler is run using a different stack (the one passed via the `stack` parameter to `context_create`), so the image saved by `context_save` actually stores in `%rsp` a pointer to the top of the new stack. Similarly to `setjump`, the `context_save` function returns 0 when called directly, while it returns a user-specified value whenever the context is later restored. Since at this time we are directly calling `context_save`, the return value is 0 and the `return` statement is executed so that control returns to `context_create`. We note that this return statement completes the execution of the manually-activated signal handler, so the operating system returns the control to `context_create` in a state which is no longer related to the signal which was raised.

At this point, `context_create` relies again on `sigaltstack` to restore the previous setting (namely, new signals are not run in the previously-specified different stack) and it issues a call to `context_switch_create`, which relies on special versions of the Posix-compliant `setjmp()`/`longjmp()` API we have developed, and is implemented as the following macro:

```
#define context_switch_create(context_old, context_new) \
    if(set_jmp(context_old) == 0) \
        long_jmp(context_new, 1)
```

Basically, the purpose of this macro is to store the current execution context in the `context_old` variable, and to restore the context stored in `context_new`. This can be done safely by checking again for the return value 0 from `set_jmp`, which is returned exactly when `context_switch_create` is referenced for the first time (i.e., we don't restore `context_new` when we return from `context_new` due to an additional context switch). Therefore, once `context_switch_create` is referenced by `context_create`, control is returned again to `context_create_trampoline`. At this time, since the second argument of `long_jmp` in `context_switch_create` is 1, the check for 0 is not satisfied, and the execution flow continues. We emphasize that this invocation to `long_jmp` already changes the stack, bringing back the stack which was setup by the operating system due to the

sigaltstack call earlier in context_create. Nevertheless, the execution is not related to a signal handler now.

Next a call to context_create_boot() is issued. This implements the last step in the construction of the ULT context upon starting up the simulation, and is declared as a noreturn function. This is due to the fact that the final goal of ULTs in our system is to run simulation objects, which have a lifetime as long as the whole simulation, and therefore they are realized using a private main loop which never ends—if a simulation object has no scheduled events, then its context is simply not reactivated by the worker thread in charge of its execution. The implementation of context_create_boot is as follows:

```
static void context_create_boot(void) __attribute__ ((noreturn));
static void context_create_boot(void) {

    void (*context_start_func)(void *);
    void *context_start_arg;

    context_start_func = context_creat_func;
    context_start_arg = context_creat_arg;

    // Go back where the thread was created, being ready to restart from here when the ULT is scheduled!
    context_switch(context_creat, &context_caller);

    // Magically start the thread
    context_start_func(context_start_arg);

    // you should never reach this!
    abort();
}
```

By relying on it, the worker thread makes a copy of the ULT entry point and its arguments (which were temporarily stored in a TLS variable) in a couple of local variables. These local variables are actually persistent to further context changes exactly because each ULT has its own stack. At this point, everything is ready to activate the ULT, but this is postponed to the time instant the simulation object has to be activated for event processing—this will be done by calling the local function pointer context_start_func(). Control is now returned to the worker thread by issuing a call to context_switch, which restores the previous context, namely the one at the end of context_create. The latter function then returns, and the normal execution flow is restarted. Nevertheless, the argument context of context_create can now be used to reactivate the ULT (with a separate stack) whenever an event bound to the simulation object should be CPU-dispatched. Figure 12 summarizes the steps which we have just described.

We shall now discuss how it is possible to explicitly switch among two different execution contexts, in our architecture. The main problem we had to face is related to context-switch management upon timer-interrupts delivered to the Time Warp platform. To understand this issue, we must consider the fact that computing architectures rely on *calling conventions* to change the control flow between functions. Calling conventions usually divide general purpose registers between *caller-save* and *callee-save* registers. The former category encloses all the registers which, upon a function call, are not guaranteed to be saved by the called function—after the function's return point, their content might be clobbered. On the other hand, callee-save registers are those which are guaranteed to be saved by the called function before using them, so that the previous content can be restored before the function returns. It is the responsibility of compilers to coherently handle calling conventions, allowing as well for code reuse (e.g., in the context of libraries) on a same architecture. On x86-64, which is the target of our work, calling conventions are dictated by the System V AMD64 ABI. In

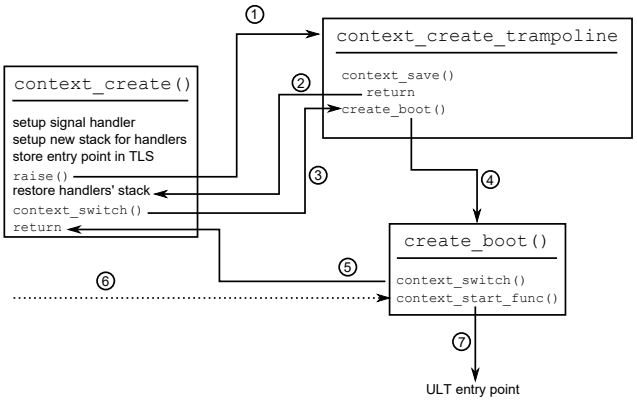

Fig. 12. Steps taken to setup a ULT—①: a signal is raised towards a signal handler using a separate stack, ②: the CPU context is saved, and control is returned to complete the signal handler, ③: context is changed explicitly, returning to the signal handler although in regular execution mode, ④: a function is called to store a local copy of the ULT entry point and arguments, ⑤: context is switched back again, to restore the original execution flow, ⑥: eventually the context is switched again, to activate the ULT, ⑦: ULT's entry point is called.

this calling convention, callee-save registers are %rbp, %rbx, and %r12–%r15, any other general purpose register is caller-save.

Since compilers are assumed to be consistent with calling conventions, traditional implementations of setjmp/longjmp Posix API functions leverage them to reduce their internal execution time. In particular, a setjmp is regarded by the compiler as a function call, therefore any required caller-save register is pushed before issuing that call. This allows setjmp to store the execution context keeping only callee-save registers %rbp, %rbx, %r12–%r15, along with the program counter %rip and the status register %rflags. No other register should be copied, since when control is returned after the call to setjmp, any required caller-save register is restored by the code generated by the compiler. Once again, we emphasize that this (correct) behaviour is triggered by the fact that the compiler sees a function call, and enforces calling conventions.

In our scenario, such a function call cold be missing. In particular, we might change the flow of the program at any point, due to the interception of a timer-interrupt delivered to the Time Warp plaform. Therefore, the compiler has no clue about what registers could be caller-save, so it never emits instructions to store their value. Anyway, they might be required to be saved at any point in the program. Therefore, traditional setjmp/longjmp API simply do not work in our time-sharing Time Warp environment.

We therefore rely on special versions of these functions, which are calling convention-agnostic, meaning that they save the whole CPU state. Although this is a bit more costly, it is mandatory to enforce correctness in the execution. As an additional note, x86 CPUs offer a plethora of additional registers, e.g. those related to floating-point instructions. By the definition of the calling conventions, they are all caller-save, so they must be explicitly saved/restored by our context switch facilities. We report in the following code snippet the source for the _set_jmp() function, which is at the heart of the implementation of the set_jmp which we have previously shown. It is directly implemented in assembly, due to efficiency reasons and because it explicitly breaks traditional x86-64 calling conventions.

To understand the organization of this function, we note that its C-like signature is long long _set_jmp(exec_context_t *env), thus it takes as its only argument a pointer to an exec_context_t structure, which keeps enough space to store the whole CPU state. According to System V AMD64 ABI, the first argument of a function (in

case it is a pointer) is passed via the %rdi register—when we enter the _set_jmp function, thus, we have already lost the possibility to save its value, so this should be handled in a different way.

```
.align 4
.globl _set_jmp
.type _set_jmp, @function
_set_jmp:
pushq %rax          # save rax, it will point to the context
pushq %r11          # save r11, it will be used as the source

lahf                # Save status flags on stack
seto %al
pushq %rax

# Save the context
movq %rdi, %rax     # rax points to the context
movq 16(%rsp), %r11 # r11 keeps the 'old' rax
movq %r11, (%rax)   # rax is the first field of the context
movq %rdx, 8(%rax)
movq %rcx, 16(%rax)
movq %rbx, 24(%rax)
movq %rsp, 32(%rax)
addq $16, 32(%rax)  # saved rsp must point one quadword above the old return address
movq %rbp, 40(%rax)
movq %rsi, 48(%rax)
movq 32(%rsp), %r11 # old 'rdi' was pushed by the surrounding macro
movq %r11, 56(%rax)
movq %r8, 64(%rax)
movq %r9, 72(%rax)
movq %r10, 80(%rax)
movq 8(%rsp), %r11  # r11 keeps the 'old' r11
movq %r11, 88(%rax) # r11 is the 12-th field of the context
movq %r12, 96(%rax)
movq %r13, 104(%rax)
movq %r14, 112(%rax)
movq %r15, 120(%rax)
movq (%rsp), %rdx   # (%rsp) is flags
movq %rdx, 136(%rax)

movq 24(%rsp), %r11 # Save the original return address
movq %r11, 128(%rax)

# Now save other registers. fxsave wants memory aligned to 16 byte.
# The context structure is aligned to 16 bytes. We have 18 8-byte
# registers, so the next address is exactly the 'others' buffer.
fxsave 144(%rax)

addq $24, %rsp
xorq %rax, %rax # return 0 because the context is being created
ret
```

The idea behind our construction of this context-save facility is to move the content of all CPU registers in a memory buffer. To this end, we must use some register as a "pointer". This register is %rax, so in order to preserve its value we first push its content to the stack. We then use %rax to save the status register. We can then save all registers to the buffer: this is done via a couple of mov instructions (we recall that the pointer to the buffer was passed via %rdi). Some register's content must be reconstructed, such as the stack pointer %rsp, since we executed within this function a couple of push instructions. Nevertheless, since we know the number of pushes, we can determine the offset to apply to the current value of %rsp. Similarly, we clobbered the %rax register to save the content of the status flags, but we pushed it beforehand, so we can retrieve it from the stack. Similarly, since we issued a call to _set_jmp, we can retrieve the original program counter's value from the stack.

To save the remainder of the CPU state, we must save all other caller-save registers which are used to support all floating-point instructions. Manually saving them could be difficult and performance-unfair, as the number of these registers is large, and they cannot be accessed using traditional `mov` instructions. Therefore, we rely on the `fxsave` instruction, which is offered by x86 architectures to save very quickly all registers used for floating-point and vectorized instructions.

To complete the execution of our context-save procedure, we return 0, to be compliant with the original semantic of `setjmp`—by the calling convention the return value is always stored into `%rax`. Nevertheless, we still have to discuss how we can save the original value of `%rdi`, which is clobbered by the function call due to calling conventions. In fact, `context_switch` does not call `_set_jmp` directly, rather it relies on the `set_jmp` macro which is defined as follows:

```
#define set_jmp(env)  ({\
    int _set_ret;\
    __asm__ __volatile__ ("pushq %rdi"); \
    _set_ret = _set_jmp(env); \
    __asm__ __volatile__ ("add $8, %rsp"); \
    _set_ret;\
})
```

This macro evaluates the return value of `_set_jmp`, making it compliant in its turn with the original `setjmp`, but before issuing the call to `_set_jmp`, it pushes the value of `%rdi` on the stack (and similarly removes it from the stack after the call returns). In fact, looking at the code of `_set_jmp`, we explicitly retrieve the original value of `%rdi` from the stack, since it was pushed by the surrounding macro.

To discuss how we can restore a previously-saved context, we first emphasize that the actual program counter's value that we have saved in the CPU context is the address of the first instruction after the call to `_set_jmp`, namely the instruction that stores `_set_jmp`'s return value into `_set_ret` in the macro. Therefore, to discriminate whether we are returning from a `_set_jmp` to a `longjmp` (or equivalent) we can play with this return value. Nevertheless, care must be taken to restore as well the actual original value of `%rax`, which is used to store the return value. First, we introduce the C-like signature of our `_long_jmp` (which is presented in the next code snippet) as `__attribute__ ((__noreturn__)) void _long_jmp(exec_context_t *env, long long val)`, where `val` allows to specify the return value that we want to use as the return value of our "fake" invocation of `_set_jmp` upon a context restore. This value is stored into the `%rsi` register, as per the calling conventions.

```
.align 4
.globl _long_jmp
.type _long_jmp, @function
_long_jmp:
movq %rdi, %rax # rax points to the context

movq 128(%rax), %r10 # This is the old return address
movq 32(%rax), %r11 # r11 is the old rsp
movq %r10, 8(%r11) # restore the old return address

movq %rsi, (%r11) # Put on the old stack the desired return value

movq 8(%rax), %rdx # rdx is the second field of the context
movq 16(%rax), %rcx
movq 24(%rax), %rbx
movq 32(%rax), %rsp
movq 40(%rax), %rbp
movq 48(%rax), %rsi
movq 64(%rax), %r8
movq 72(%rax), %r9
movq 80(%rax), %r10 # Finish to restore GP registers
movq 88(%rax), %r11
movq 96(%rax), %r12
```

```
movq 104(%rax), %r13
movq 112(%rax), %r14

# Restore FLAGS
movq 136(%rax), %rax # this is flags
addb $0x7f, %al # Overflows if OF was set
sahf

# Restore remaining rdi and r15
movq %rdi, %rax # rax now points again to context
movq 56(%rax), %rdi
movq 120(%rax), %r15

fxrstor 144(%rax) # Restore other registers

movq 32(%rax), %rsp # (possibly) change stack
popq %rax # Set the desired return value
ret # do the long jump
```

In our implementation, we play a bit with the "other" stack, namely the stack of the destination ULT. This can be done since we can extract from the stored execution context the value of `%rsp` before having restored it in the current CPU state. In our code, register `%r11` is used to this end. In this way, while executing on the current stack, we can read/write values to the destination stack. In fact, we use this facility to make a copy of `val` (stored in the `%rsi` register) on the destination stack. This frees the `%rsi` register, which can be restored along with all other registers. To move the return value from the stack to `%rax`, we just issue a `pop %rax` instruction just before returning from `_long_jmp`. To restore `%rflags`, we use a triplet of instructions in a way similar to what we did in the context-save procedure, considering that `addb $0x7f, %al` generates an overflow only if the overflow flag was set during the context-save procedure. After this point, we must ensure that no arithmetic/logical instructions are executed. To restore floating-point registers, we use the companion `fxrestor` instruction.

To actually perform the context switch, we use a trick similar to what we did to restore `%rax`. In particular, we read from the saved context the old program counter's value, and we push that on the destination stack. Therefore, once the stack is changed, on the top of the stack we find exactly the address of the instruction next to the call to the corresponding `_set_jmp`. A `ret` instruction will "jump" to the program counter's value of the destination CPU context. In order to have the `_long_jmp` restore the actual original value of `%rax`, it can be called as `_long_jmp(context_new, (context_new)->rax)`.

B. ENABLING TIMER-INTERRUPT DELIVERY TO THE TIME WARP PLATFORM

In this section we initially discuss basics on how Linux manages timer-interrupts, and then provide the description of our timer-interrupt management module enabling the delivery of timer-interrupts (with fine granularity) to the Time Warp platform.

B.1. Basics on Linux Timer-Interrupts

x86 processors are equipped with various timer facilities, among which one is ultimately exploited to drive the passage of time on each CPU-core. This is the LAPIC-timer supported by APIC (Advanced Programmable Interrupt Controller), which is a timer-component local to the CPU-core (see Figure 13).

The LAPIC-timer can be configured to operate in different modes, among which the one used by the Linux kernel is the periodic-interrupt mode. Specifically, at kernel startup, a so called calibration procedure is executed such that the LAPIC-timer is setup (in terms of its internal hardware counter, upon the expiration of which the interrupt is issued towards the associated CPU-core) so as to periodically generate interrupts according to the frequency established by the `CONFIG_HZ` parameter defined

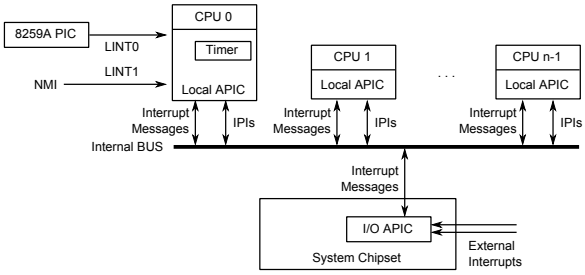

Fig. 13. x86 interrupt system.

at kernel compile time. Classical interrupt periods for entry level to medium-end machines range from 1 to 4 milliseconds, which is reflected to values of `CONFIG_HZ` ranging from 1000 to 250. Once setup, the configuration of the LAPIC-timer is never changed, thus the same interrupt period (which we also refer to as original timer-tick) is always used at operating system steady state.

The timer-interrupt management scheme supported by Linux is still based on the top/bottom-half paradigm. In more detail, upon the receipt of the LAPIC-timer interrupt, a very minimal portion of code (the top-half) is executed, which is only used to update timing information and to possibly flag the current thread in such a way that eventually the scheduler is called and a context switch (leading the thread off the CPU) can take place. A thread that has been CPU-dispatched is typically allowed to run for various timer-ticks before being flagged for re-schedule. On the other hand, the call to the kernel `schedule()` function for performing actual context switches (if requested) is actuated right prior to leaving kernel mode[11]. Therefore, the `schedule()` function, which represents the core part of the bottom-half of the timer-interrupt manager, is executed only in case no kernel-level critical task is being executed by the thread. This allows for scalability on multi-core processors, given that de-scheduling a thread during the execution of any kernel-level critical task, such as a spinlock-protected kernel-level critical section, would lead the critical section to be locked up to the point in time where this same thread will be CPU-dispatched again, an operation that may occur after an unpredictable amount of time (also depending on workload and thread priorities).

B.2. The Extra-Tick Logic

The extra-tick logic at the core of our time-sharing architecture is based on a kernel-level differentiation between Time Warp threads and other kinds of threads (running generic applications or kernel level housekeeping tasks), since only the former ones need to be managed according to the lightweight extra-tick scheme. To this end, the Linux module we developed offers the support for a special device file called `dev_extra_tick` such that:

— this special device file is single instance, hence no two different concurrently-opened I/O sessions on it are allowed. This is compliant with the idea that a single process—namely the multi-thread Time Warp platform running on the multi-core machine—needs to use the facilities offered by the special device file for supporting the execution of all its worker threads;
— a thread can register itself as a Time Warp worker thread by issuing an `ioctl` call towards the device file.

[11] A minor variation is in place for the case of kernel threads, which never leave kernel mode operations.

Registering a thread on the special device file allows the kernel to know that the thread needs to undergo the extra-tick policy. Upon registering, the kernel-level thread identifier is recorded into a fast access hash table, which is installed as part of the kernel module data structures implementing the special device file driver. At this point, the portions of the whole kernel architecture that need to know whether some thread is registered, thus requiring ad-hoc tick management, are the following two: (i) the kernel scheduler, and (ii) the top-half of the timer-interrupt handler. The external module implementing the `dev_extra_tick` device file is also in charge of redefining the behavior of the kernel scheduler and of the top-half of the timer-interrupt handler, thus leading them to become compliant with extra-tick management requirements. Specifically, our module adopts a *dynamic patching* approach that rewrites parts of the executable image of the kernel upon being loaded. This leads to avoid kernel recompilation.

To patch the kernel `schedule()` function, we retrieve the memory position of the corresponding machine instructions block from the system-map (typically available in Linux installations from the `/boot` directory of the root file system), and we inject into this routine an execution flow variation such that control goes to a `schedule_hook()` routine offered by the external module right before `schedule()` would execute its finalization part (e.g. stack realignment and return). A scheme of this patching approach is shown in Figure 14, which has been tested on Linux kernel versions from 2.6 to 3.2. According to this patching scheme, the `schedule()` function will never return, rather it will pass control to `schedule_hook()` so that the final part of the scheduling process is under the control of our external module. In the end, the `schedule_hook()` function will simply execute the same return actions originally planned by the kernel `schedule()` function. However, patching the original scheduler in this way allows the hook to take control when the decision about what thread needs to take control of the CPU-core[12] is already finalized. Hence, we know what thread will have control of the CPU-core for the current set of operating system assigned ticks. As a consequence, the hook is able to check whether the thread is a registered one (so that it needs to be extra-ticked) by consulting the aforementioned fast access hash table implementing the registration record, and in the positive case it executes the following additional steps:

A) It changes the LAPIC-timer period by scaling it on the basis of a configuration parameter supported by our kernel module. The scaling factor is what determines the length of the extra-tick interval.
B) It records in a per CPU-core entry of a proper control table (still managed by the module) that the current CPU-core is working in extra-tick mode.
C) It records in a proper per registered-thread entry of a control table (again managed by the module) a counter of extra-ticks not yet consumed by such a thread within the current tick period.

Clearly, the information recorded in point B is also used in order to revert the LAPIC-timer configuration to the original one. In more detail, if the scheduler passes control to a non-registered thread, and the current CPU-core is registered as operating in extra-tick mode, then the LAPIC-timer is restored to its initially configured counter value, thus the scheduled thread will run with a classical tick length, and the control record associated with the CPU-core is reset in order to reflect that the CPU-core is no longer operating in extra-tick mode. This approach works also in scenarios where the thread registered within the `dev_extra_tick` device file looses control of the CPU-core because of a passage into a sleep state (e.g. for an I/O interaction). Overall the above scheme allows restoring the LAPIC-timer configuration to the original one each

[12] It has actually already taken control of the CPU-core, since we are returning from the scheduling process.

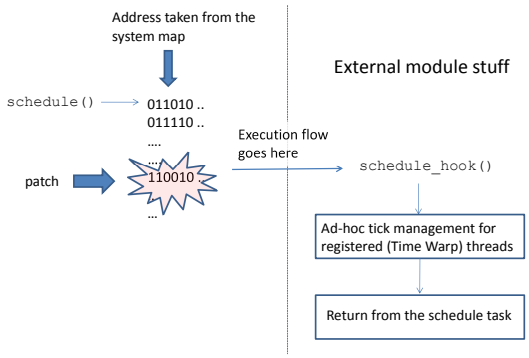

Fig. 14. Dynamic patching of the Linux kernel scheduler.

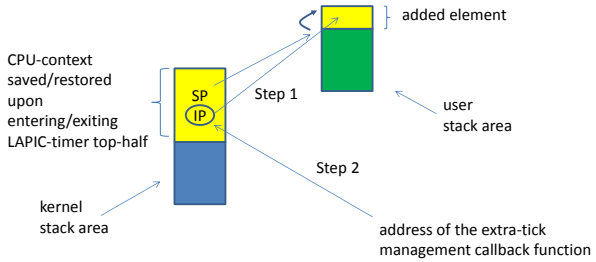

Fig. 15. Stack and CPU context management by the LAPIC-timer top-half hook.

time a non-registered thread is (re)scheduled independently of any state-transition of registered (hence extra-ticked) threads in the operating system state diagram.

Let us now analyze how the original top-half of the LAPIC-timer interrupt handler has been patched so as to exploit extra-ticks for control flow variations of the `dev_extra_tick` registered threads, say the Time Warp worker threads. The patch has been developed by targeting kernel version 3.16.7, but it is of general use (except for a few minor modifications that might be required for other kernel versions depending on the exact path of execution of, e.g., very basic actions in the preamble of the actual timer-interrupt management logic—details on this aspect will come shortly).

Top-half modules in conventional Linux configurations are made up by two different code blocks, a launcher and an actual top-half procedure. The launcher takes control when the CPU-core firmware accepts the interrupt. It is in charge of aligning the kernel-level stack of the interrupted thread to a proper snapshot and of calling the actual top-half module. Such a snapshot also includes the CPU-context to be restored once the interrupt handling top-half procedure ends. This includes the stack pointer (SP) and the instruction pointer (IP) associated with the interrupted execution flow.

In our patching approach of the LAPIC-timer interrupt management logic, we have still exploited the system-map to locate the launcher code block in the kernel memory image, and then we patched it by replacing the call to the original top-half with one to a top-half hook function offered by the external module that we have developed, which therefore fully replaces the original top-half procedure. The top-half hook is in charge of executing the same identical basic actions as those executed by the original top-half procedure (such as acknowledging the accepted interrupt). However, it discriminates

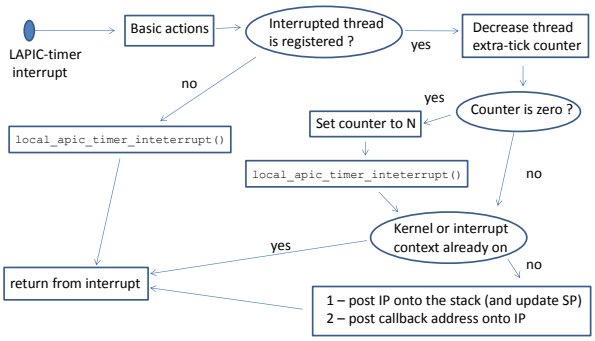

Fig. 16. Behavior of the top-half hook for the LAPIC-timer interrupt.

if the interrupted thread is a `dev_extra_tick` registered one (namely, one subject to extra-tick management), and in the positive case it executes the following actions:

(i) It decreases the extra-tick counter associated with the thread (as hinted, this counter is set upon the reschedule of any thread registered on `dev_extra_tick`).
(ii) If the counter reaches the value zero, then a whole originally-sized tick-period has expired (i.e., the thread consumed all the extra-ticks granted to it in its current tick period). In this case, the top-half hook calls the kernel function used to update kernel-level timing information (in most of the recent Linux kernel versions this work is carried out via the `local_apic_timer_interrupt()` function). This mimics the behavior of the original top-half manager execution path, given that it would trigger the timing information update function exactly at the end of each originally-sized tick-period.
(iii) The top-half hook changes the IP kept by the processor image registered into the system stack upon interrupt acceptance, so that the interrupted thread will gain control in a proper machine code block upon the restore of that image onto the CPU-core when returning from LAPIC-timer interrupt. In our design, this code block corresponds to the **extra-tick-manager** function, which we have presented in Section 2.2 of the main body of this article. Consequently, the top-half hook also changes the application-level stack layout of the thread by adding a program-counter return value that will allow that code block to exactly return control to the instruction interrupted by the extra-tick (namely, the original IP value logged into the CPU-context snapshot on the system stack). This is done by exploiting the SP value from the logged CPU-context, which is then modified in order to reflect the insertion of a new element at the top of the user level stack. A schematization of the performed operations is provided in Figure 15.
(iv) Finally, if the extra-tick counter of the thread registered within the `dev_extra_tick` device file reached the value zero—see point (ii)—the thread is again filled with the number of extra-ticks (say N) it is allowed to receive in the next tick period.

The address of the **extra-tick-manager** code block that will take control as a callback from the kernel thanks to the instruction pointer variation in point (iii) is posted to the kernel when calling the same `ioctl` system call that is used for registering the thread on the `dev_extra_tick` device file as one to be extra-ticked. Overall, a Time Warp thread can atomically register itself for being subject to extra-ticks and post the address of the function whose execution is activated thanks to the actions by the top-half hook of the LAPIC-timer interrupt we provide within our module.

The behavior of our top-half hook for the LAPIC-timer interrupt is schematized in Figure 16. It is still lightweight given that the additional actions it performs (compared

to the original top-half version) have constant time and are mostly related to decrementing and (possibly) setting a counter (the per-registered-thread extra-tick counter) and setting a few memory locations, one in the application-level thread stack and the other ones in the user level CPU-context logged in the stack (namely IP and SP values, that will be then restored upon exiting the interrupt procedure).

Although our software architecture is available for free download (at the URL `https://github.com/HPDCS/ROOT-Sim`), for the sake of the reader's convenience we report below a snippet of code showing the structure of our hook of the original LAPIC-timer interrupt handler:

```
void smp_apic_timer_interrupt_hook(struct pt_regs* regs) {

    unsigned long auxiliary_stack_pointer;
    unsigned long flags;
    unsigned int timer_cycles;


    struct pt_regs *old_regs = set_irq_regs(regs);

    .... //ack of the timer interrupt removed from this snippet of code

    if(current->mm == NULL) goto normal_LAPIC_timer_interrupt;  /* this is a kernel thread */

    if(registered_in_dev_extra_tick(current)) goto extra_tick_LAPIC_timer_interrupt;
    //this is a Time Warp thread - need extra tick management

    normal_LAPIC_timer_interrupt:

    //CPU-core not working in extra tick mode - need to update software timers
    if(extra_tick_flag[smp_processor_id()] == 0 || CPU_extra_ticks[smp_processor_id()]<=0){
            local_apic_timer_interrupt();
    }

    //realign the timer interrupt period if needed
    //no additional cost (except for the predicate evaluation) in non-extra-tick scenarios
    if(extra_tick_flag[smp_processor_id()] == 1){
        local_irq_save(flags);
        CPU_extra_ticks[smp_processor_id()] = 0;
        extra_tick_flag[smp_processor_id()] = 0;
        timer_cycles = (*original_calibration) ;
        setup_APIC_LVTT(timer_cycles, 0, 1);//reset timer original calibration
        local_irq_restore(flags);
    }

    my_irq_exit();
    set_irq_regs(old_regs);

    return;

    extra_tick_LAPIC_timer_interrupt:

    if( CPU_extra_ticks[smp_processor_id()] <= 0 ){//original tick expired
        //reassign fine-grain ticks to the Time Warp thread
        CPU_extra_ticks[smp_processor_id()] = EXTRA_TICK_SCALING_FACTOR;
        local_apic_timer_interrupt();
    }
    else{
        CPU_extra_ticks[smp_processor_id()] -= 1;//one less fine grain-tick to spend
    }

    if( old_regs != NULL ){//interrupted while in kernel mode running
        goto extra_tick_APIC_interrupt_kernel_mode;//cannot run user space timer handler
    }

    if((regs->ip >= data_section_address) ){//interrupted while running outside the Time Warp system code
        goto extra_tick_APIC_interrupt_kernel_mode;//cannot run user space timer handler
    }
```

```
    if(extra_tick_handler() != NULL){//do we have a valid address for the extra-tick-manager?
        //actual manipulation of user space stack and processor state to run the handler
        local_irq_save(flags);
        auxiliary_stack_pointer = regs->sp;
        auxiliary_stack_pointer -= sizeof(regs->ip);
        copy_to_user((void *)auxiliary_stack_pointer,(void *)&regs->ip, sizeof(regs->ip));
        regs->sp = auxiliary_stack_pointer;
        regs->ip = extra_tick_handler();
        local_irq_restore(flags);
    }

    extra_tick_APIC_interrupt_kernel_mode:

    local_irq_save(flags);
    extra_tick_flag[smp_processor_id()] = 1; //still running a Time Warp thread
    timer_cycles = (*original_calibration) /  EXTRA_TICK_SCALING_FACTOR;
    setup_APIC_LVTT(timer_cycles, 0, 1); //post the fine-grain tick to the LAPIC-timer
    local_irq_restore(flags);
    irq_exit();
    set_irq_regs(old_regs);

    return;

}
```

As an additional note, our approach to modify the execution flow of Time Warp worker threads is based on modifying the user space stack of the thread just above the current stack pointer address. So we do not allow software to use the so called *red zone* of the stack[13], which is achieved by simply compiling both application and platform software with red zone exclusion directives. Making our proposal compliant with the reliance on red zones of the stack would require putting in place in point (iii) a stack management logic similar to the one used by operating system kernels for the activation of signal handlers. However, this is an issue aside of the core aspects characterizing our design.

[13]The red zone is the stack region above the current stack frame. It is typically exploited by conventional compilation tool-chains so as to allow a leaf function to use the stack with no explicit storage reserving–via decrease of the stack pointer– within the stack frame