View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Archivio della ricerca- Università di Roma La Sapienza

Time-Sharing Time Warp via Lightweight Operating System Support

Alessandro Pellegrini and Francesco Quaglia
DIAG – Sapienza, University of Rome
Via Ariosto 25, 00185 Rome, Italy
{pellegrini, quaglia}@dis.uniroma1.it

ABSTRACT

The order according to which the different tasks are carried out within a Time Warp platform has a direct impact on performance, given that event processing is speculative, thus being subject to the possibility of being rolled-back. It is typically recognized that not-yet-executed events having lower timestamps should be given higher CPU-schedule priority, since this contributes to keep low the amount of rollbacks. However, common Time Warp platforms usually execute events as atomic actions. Hence control is bounced back to the underlying simulation platform only at the end of the current event processing routine. In other words, CPU-scheduling of events resembles classical batch-multitasking scheduling, which is recognized not to promptly react to variations of the priority of pending tasks (e.g. associated with the injection of new events in the system). In this article we present the design and implementation of a time-sharing Time Warp platform, to be run on multi-core machines, where the platform-level software is allowed to take back control on a periodical basis (with fine grain period), and to possibly preempt any ongoing event processing activity in favor of dispatching (along the same thread) any other event that is revealed to have higher priority. Our proposal is based on an ad-hoc kernel module for Linux, which implements a fine grain timer-interrupt mechanism with lightweight management, which is fully integrated with the modern top/bottom-half timer-interrupt Linux architecture, and which does not induce any bias in terms of relative CPU-usage planning across Time Warp vs non-Time Warp threads running on the machine. Our time-sharing architecture has been integrated within the open source ROOT-Sim optimistic simulation package, and we also report some experimental data for an assessment of our proposal.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Threads*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete Event, Parallel*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGSIM-PADS'15, June 10–12, 2015, London, United Kingdom.
Copyright © 2015 ACM ISBN 978-1-4503-3583-6/15/06...$15.00.
DOI: http://dx.doi.org/10.1145/2601381.2601398.

General Terms

Algorithms, Performance

Keywords

PDES; Speculative Processing; Preemptive Scheduling

1. INTRODUCTION

Time Warp [10] is the reference synchronization protocol for optimistic parallel processing of discrete event simulation models. It allows the worker threads operating within the simulation platform to process simulation events speculatively, with no preliminary assessment of their safety. On the other hand, in case processed events are eventually revealed as non-causally consistent, their effects on the simulation model execution trajectory are undone via rollback schemes based on state recovery techniques exploiting either checkpointing [19, 21, 18] or reverse computing [5, 24]. The relevance of Time Warp (and of simulation platforms based on this synchronization paradigm) lies in that it allows for extremely high scalability. In fact, as recently shown by the results provided in [12], Time Warp systems exhibit the potential for scaling up to millions of processing units.

As for software development, beside some historical proposal along the path of developing Time Warp as a special-purpose operating system oriented at supporting discrete event applications [11], the common trend is the one according to which Time Warp systems are built as user-space platforms, to be hosted on top of general-purpose operating systems (see, e.g., [4, 17, 14, 6]). As a consequence, the platform-level software within the whole Time Warp architecture is seen by the application-level code (namely, the simulation model) as a library offering a specific API (e.g., for injection and CPU-dispatching of simulation events) and, in the most advanced cases (see [18]), providing application transparent support for recoverability[1].

The major consequence by the library-based approach is that control is bounced back to the platform-level software (along any worker thread) only upon the occurrence of specific run-time events, such as the end of the execution of the last CPU-dispatched simulation event or the interception of, e.g., memory update operations that trigger platform-level recoverability capabilities ultimately allowing to restore a

[1]As an example, in some proposals compile/link time (instrumentation) directives are used to allow the platform-level library to intercept memory updates issued by the application code so as to transparently support log/recovery of the application-level data structures.

previous simulation model's state. In other words, all the simulation events processed within conventional Time Warp systems are actually CPU-dispatched according to a classical batch-multitasking scheme, where the platform-level software is not allowed to take back control independently of the activities that are carried out by the application code. As a consequence, the platform-level software is not allowed to re-evaluate CPU assignment until the completion of the last-dispatched simulation event. Therefore, it is not able to CPU-dispatch any other simulation event that may have been produced in the system, which may have a higher priority (e.g., a lower timestamp) compared to the one currently being processed by the CPU [22].

Regaining control frequently, with re-evaluation of the assignment of the CPU, can be a relevant means for improving performance (thanks to the reduction of the incidence of rollback) given that the CPU capacity can be dynamically assigned to simulation events that currently require more prompt execution, e.g. due to their lower timestamps, and thus can more likely give rise to dependencies affecting the virtual time window that is currently covered by the worker threads processing activities. We note that this aspect is also related to improving the energy efficiency of Time Warp platforms, given that reducing the amount of rollbacks in the parallel run means reducing the overall energy used for any individual productive unit of work done (namely, eventually committed events) [20].

Clearly, the period according to which the platform-level software re-gains control needs to be fine grain, especially in contexts where CPU-requirements for processing simulation events are on the order of (tens of) microseconds. Hence classical timer-interrupt settings supported by conventional operating systems do not suffice for this purpose. As an example, common Linux configurations lead the timer to interrupt the current thread running on any CPU-core with period on the order of 1 to 4 milliseconds (higher values are typically used on machines with larger amounts of CPU-cores given that CPU preemption and reassignment is less critical on these systems compared to those with reduced number of cores). Also, bouncing control back to the platform-level software via standard temporized signals, such as the POSIX `SIGALRM` signal, would be unfeasible because of the overhead, given that this approach would require the whole chain of signal management mechanisms to be passed through at kernel level.

In this article we present the design and implementation of a time-sharing Time Warp system, to be hosted on top of multi-core machines running Linux, where the platform-level software is allowed to take back control on a periodical basis, with very fine grain period (e.g. on the order of tens of microseconds) and is allowed to re-evaluate CPU assignment, and to dynamically schedule higher priority simulation events. This is achieved with minimal overhead thanks to the capabilities of an ad-hoc Linux module for timer management that we have developed, which allows for (periodical) control flow variations along any running thread with no intervention by the chain of kernel mechanisms used for supporting POSIX signals. Our proposal is fully compliant with the conventional and scalability oriented top/bottom-half timer-interrupt management offered by Linux, and does not create any bias in terms of actual CPU-assignment across the threads (including kernel-level threads) operating in the system. In other words, Time Warp threads are allowed to see their original CPU ticks (those natively assigned by the operating system) as partitioned into sub-intervals (with proper control flow management at the end of each sub-interval), while any other active thread in the system (which is not a Time Warp worker thread) is not subject to any partitioning of its assigned ticks. This prevents to impair fairness, which is an essential pre-requisite given that the time-sharing Time Warp platform might run on a multi-user conventional platform. This aspect is clearly relevant also when considering fairness across Time Warp worker threads and kernel-level threads used by the operating system for housekeeping.

Besides the ability to optimize CPU assignment depending on the (dynamic) priority of the tasks to be performed within the Time Warp system, 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 [15]. These loops can be (timely) broken thanks to our time-sharing approach which can be exploited for supporting preemptive rollback operations leading to the squash of the non-admissible state trajectory.

The fully featured time-sharing architecture we have developed has been integrated within the open source ROOT-Sim package[2] [17, 9], and operates in a fully transparent way to the overlying application code. Hence, the benefits from it come with no explicit intervention by the application programmer. We also report experimental data for an assessment of the time-sharing Time Warp proposal in terms of both overhead by the extra-ticks and final delivered performance with a real-world case study application in the area of simulation of wireless systems.

The remainder of the article is structured as follows. In Section 2 we discuss related work. The whole time-sharing architecture is presented in Section 3. Experimental data are provided in Section 4.

2. RELATED WORK

In the wide area of High-Performance-Computing (HPC) systems, some literature studies exist on the relation between performance and timer-interrupt frequency. The common idea underlying most of the outcoming performance optimization proposals is that the lower the timer-interrupt frequency, the better the final delivered performance [7, 8, 25]. The exacerbation of this approach led to defining *tickless* operating systems (namely, with extremely reduced frequency of the timer-interrupt) as the best configuration for hosting HPC applications. However, these studies have been tailored to the case of non-speculative processing, where any work carried out by the thread running on whichever CPU-core is actually useful, hence there is no actual 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 optimistic 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 actual 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)

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

in HPC contexts, not at introducing ad-hoc software modules for exploiting timer events, which is instead the path we followed. In fact, our proposal puts in place a special (and lightweight) mechanism for handling timer-interrupts. Overall, we follow an approach which 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 optimistic PDES systems, the only work we are aware of which deals with the relation between performance and timer-interrupt configuration is the one in [3]. 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 workload on the system, 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 time, thanks to an appropriate periodic variation of the control flow along Time Warp threads. Also, while the proposal in [3] 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 across 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 (as hinted, with proper control flow management at the end of each sub-interval). This leads our proposal to be suited also for context where the computing platform is shared across different users and applications.

As pointed out, our time-sharing Time Warp proposal also entails the capability of supporting preemptive rollback of the currently (incorrectly) executed simulation event. The topic of preemptive rollback in optimistic discrete event simulation has been studied in literature, mainly in [6, 23]. The Time Warp platform presented in [6] supports event preemptive rollback for optimistic 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 after preemption of the last dispatched one) independently of the actual materialization of a causality violation, rather when we also must give higher priority to a different simulation event, possibly to be executed at a simulation object different from the currently running one, anyhow currently bound to the worker thread (e.g., in order to reduce the likelihood of future rollback generation). This is basically due to the fact that our time-sharing Time Warp system is not limited to the support for preemptive rollback. As for the preemptive rollback approach in [23], it is suited for distributed memory systems (based on Myrinet interconnection) while we deal with shared memory multi-core machines. Also, the solution in [23] is based on polling, and the polling code to periodically verify the causal consistency of the current event needs to be nested by the application programmer in proper points of the application native code. Instead, our proposal is based on interrupts, and is fully transparent to the application code (and thus to the developer). Finally, similarly to [6], the solution in [23] 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 currently executed along the thread) that are not currently giving rise to a rollback operation.

As a matter of fact, our time-sharing Time Warp approach is based on a kind of dual-mode execution, where control is periodically pushed back to platform mode via the kernel-level extra ticks' management system. Dual-mode execution in Time Warp systems has been already studied in [16], however this work is focused on the management of different memory views (via operating system-level ad-hoc facilities) so that when running in application mode, only a sub-portion of the whole address space is accessible by the Time Warp worker thread, namely the sub-portion keeping the memory layout of the 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 the access to any valid memory location by any event processing routine (as for classical sequential style coding of DES models). Rather, 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) and to the dynamical assignment of CPU to these events. Still, like the proposal in [16], we retain application transparency.

Finally, the present proposal is clearly related to the seminal work in [11], where Time Warp is instantiated as a special-purpose operating system (although operating in user space), destined to host discrete event applications according to a speculative processing paradigm. The core difference between what we are presenting and the proposal in [11] lies in that such an approach is still based on batch-processing of the events, with preemption only used in case of causality errors affecting the currently-dispatched simulation event. Rather, our approach is a truly time-sharing one, which is achieved thanks to the employment of kernel-level ad-hoc (and lightweight) modules specifically oriented to the management of control flow variations (hence preemption in its wide usage) on a fine-grain periodical basis.

3. THE TIME-SHARING ARCHITECTURE

3.1 Basics on Linux Timer-Interrupts

As for our target machines, namely `x86` ones, modern processors are equipped with various timer facilities, among which one is ultimately exploited to drive the passage of time on each individual CPU-core. This is the LAPIC-timer component supported by APIC (Advanced Programmable

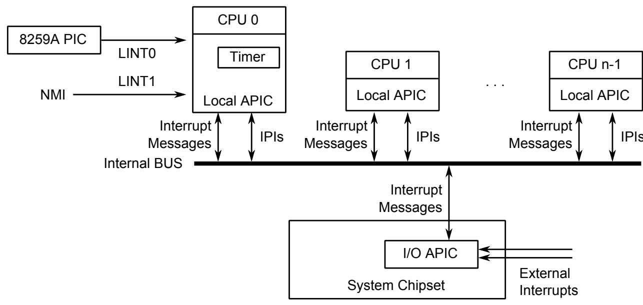

Figure 1: x86 interrupt system.

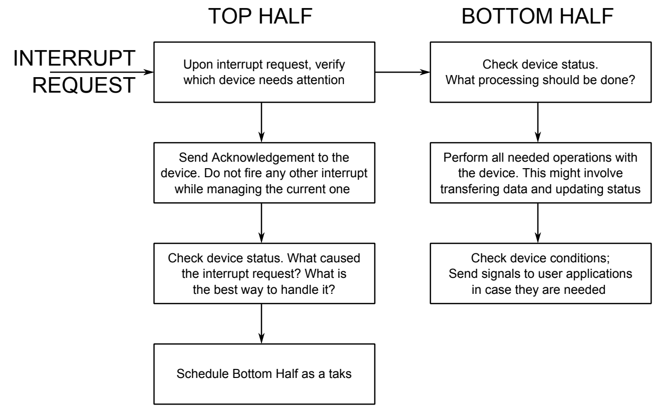

Figure 2: Top/bottom-half general scheme.

Interrupt Controller), which is a timer-component local to any CPU-core in the system. The general hardware organization on x86 architectures regarding interrupt management is depicted in Figure 1.

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. More 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 at kernel compile time. As hinted, 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 at startup, the configuration of the LAPIC-timer is never changed, thus the same interrupt period is always used at system steady state.

As for software-side management of the interrupt, we have that the actual management scheme supported by Linux is based on the top/bottom-half paradigm. More specifically, upon the receipt of the LAPIC-timer interrupt, a very minimal piece of code is executed by the CPU-core, 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 (that switches this thread off the CPU) can take place. Actually, a thread that has been dispatched on CPU is typically allowed to run for various timer-ticks before being flagged for re-schedule. The general scheme for top/bottom-half management, for the sake of clarity, is reported in Figure 2.

All the above (fine grain) actions are representative of the top-half portion of the timer-interrupt handler. On the 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[3]. 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-rescheduled, an operation that may occur after an unpredictable amount of time (also depending on system workload and relative thread priorities).

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

3.2 The Extra-Tick Logic

As hinted, our time-sharing architecture is based on a kernel-level differentiation between Time Warp threads and other kinds of threads (running generic applications), since only the former ones need to be managed according to the lightweight extra-tick scheme. To this end, we have developed a Linux module offering 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 of its worker threads;
- a thread can register itself as a Time Warp worker-thread by issuing as simple `ioctl` call towards the device file.

Registering a thread on the special device file allows the kernel to know that the thread needs to undergo the extra-tick policy. Actually, registration means that the thread identifier (as seen by the kernel, not by the `pthread` library, since the two are typically different), is registered 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 and needs ad-hoc tick management are two: the kernel scheduler, and the top-half of the timer-interrupt. The external module implementing the management logic for the `dev_extra_tick` device file is also in charge of redefining the actual behavior of the kernel scheduler and of the top-half of the timer-interrupt so that their logic is modified so as to become compliant with extra-tick management requirements.

Rather than providing a recompiled version of the kernel with the aforementioned changes already implemented, our module adopts a *dynamic patching* approach where parts of the executable image of the kernel are rewritten at startup of the external module implementing the device file.

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 the `scheduler()` would execute its finalization part (e.g. stack realignment and return). A scheme of this patching approach is shown in Figure 3, which has been tested on Linux kernels from 2.6 to 3.2. The red block of code implementing the finalization part of original `schedule()` function is initially sampled and copied at the end of the `schedule_hook()` function by also adjusting relative memory references (if present) in the copy. Then we replace the red block of code in the original version of `schedule()` with a block of machine code (the yellow part) which allows passing control to `schedule_hook()` so that the actual 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's `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[4] is already finalized. Hence, we know what thread will have control of the CPU-core for the current set of 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. More in 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. Note that 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 time a non-registered thread is

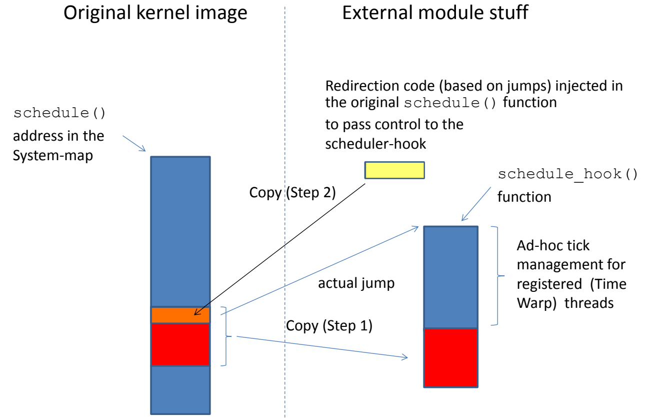

Figure 3: Dynamic patching of the kernel scheduler.

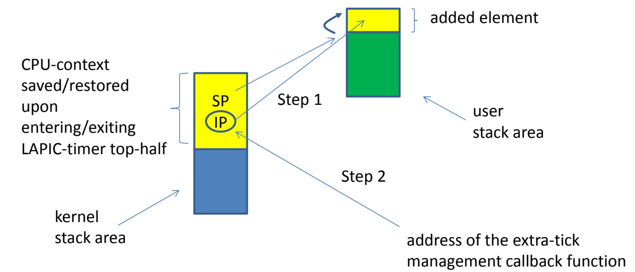

Figure 4: Stack and CPU context management by the LAPIC-timer top-half hook.

(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 has been patched in our architecture, so that the extra-ticks can be actually exploited for control flow variations of the `dev_extra_tick` registered threads, namely the ones that in our overall architectural organization run within the Time Warp platform. 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 then of calling the actual top-half module. Such a snapshot also includes the CPU-context to be restored once the interrupt top-half procedure ends. This includes the stack pointer (SP) and the instruction pointer (IP) associated with the interrupted execution flow, which will play a major role in our time-sharing architecture.

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

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

external module that we have developed, which therefore fully replaces the original top-half procedure. This 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 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 pointed out, this is the counter that is set upon the reschedule of any thread registered on the `dev_extra_tick` device file).

(ii) If the counter reaches the value zero, then it means that a whole originally-sized tick-period has expired (hence the thread consumed all the extra-ticks granted to it in its current tick period). In this case, the top-half hook calls the actual kernel function used to update kernel-level timing information (in most of the recent kernel versions this work is carried out via the `local_apic_timer_interrupt()` kernel 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, hence upon any LAPIC-timer interrupt when using the classical timer calibration.

(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 (namely, when returning from LAPIC-timer interrupt). 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 then is also 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 4.

(iv) Finally, in case 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.

In our time-sharing Time Warp architecture, the address of the code block that will take control thanks to the instruction pointer variation in point (iii) represents an ad-hoc callback function of the Time Warp platform, which will periodically (namely, at each extra-tick expiration) bring control to the platform-level software along any Time Warp worker thread. This address is posted to the kernel when calling the same `ioctl` system call that is used for registering the thread in 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

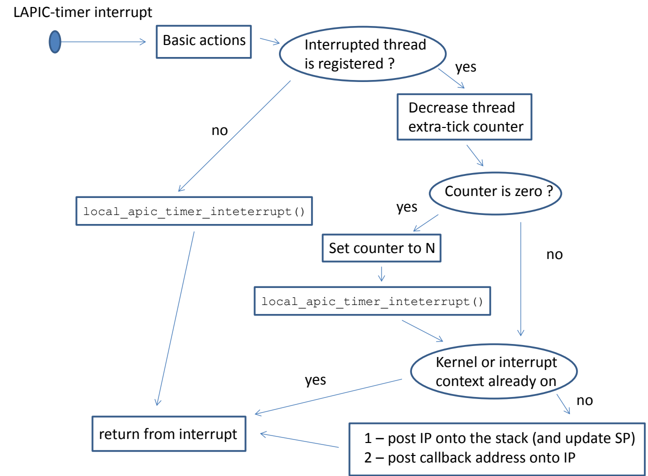

Figure 5: Behavior of the top-half hook for the LAPIC-timer interrupt.

callback 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 actual scheme according to which our top-half hook for the LAPIC-timer interrupt works is depicted in Figure 5. As one may observe, the hook version of the LAPIC-timer interrupt manager is still lightweight given that the additional actions it performs (compared to the original version of the top-half procedure) 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 (see point 1 in the bottom-right box of Figure 5, where the IP present in the snapshot of the user level CPU-context, which is already logged in the system level stack upon entering the top-half hook, is saved onto the user level 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).

As an additional note, care must be taken when receiving an extra-tick along a thread which is already working at kernel level. This may happen in case the extra-ticked thread called a system call, or if it entered kernel mode because of the receipt of a generic interrupt by some device (see Figure 1) or even because of a generic trap (e.g. an empty-zero memory access requiring the physical allocation of the requested page). In this case, variation of the flow control must not be actuated given that it would violate any, e.g., atomicity rule for the execution of already activated kernel-level code. This is reflected in our scheme by having that the control flow variation is actuated only if the interrupted thread was not already working in kernel mode or was not already subject to an interrupt. Such a check has been implemented in our top-half hook by simply checking whether the IP to be restored contains a kernel-level address, and by also checking whether an interrupt context was already active on the current CPU-core (a check that is anyhow already assessed by the basic actions that are carried out also by the original top-half procedure).

The periodic execution flow variation induced via extra-tick management, although being similar in spirit to the

one adopted for handling POSIX signals, is still much more lightweight, given that on demand usage of temporized signals would require passing through the whole scheduling process (and also trough system calls for requesting the activation of each individual alarm). Also, conventional signal handlers for temporized signals cannot operate with the same time granularity we can impose in our architecture (namely the extra-tick period) just because, in the worst case, a whole tick period might be requested in order for the kernel to take control back and determine that some alarm has expired for the thread.

As a final note, we ensure execution safety while mounting the kernel module[5] by synchronizing all the CPU-cores during the mounting operation via the `smp_call_function()` kernel function. This can be used to trigger the execution of a same code block on all the cores. We have used it to implement a master/slave protocol, similar in spirit to the one used while booting the Linux kernel, where a single CPU-core executes the actual patching of the kernel code (by also temporarily disabling write protection on the kernel image) during module startup, while the other CPU-cores wait for the master to finish. In this way, no critical race will take place, preventing any CPU-core from accessing the not yet finalized image of the kernel.

3.3 Detection and Management of Event and Task Priority Variations

The extra-tick architecture presented in the previous section is of general applicability, in terms of its ability to periodically bring control back to a specific portion of the platform-level software, and to dynamically (re)schedule on CPU higher-priority tasks. On the other hand, how to exploit it within an optimistic PDES platform depends on proper platform internals.

In this section we discuss the integration we performed between the extra-tick manager and the ROOT-Sim open source optimistic simulation platform. However, given that this platform implements some relevant reference architectural solutions specifically tailored for multi-core environments (see [27]), the presented integration, beyond giving rise to a specific time-sharing Time Warp solution, can also be seen as one providing reference guidelines for time-sharing organizations of optimistic PDES.

The core ROOT-Sim aspect that is of interest in this discussion is the management of the message (or anti-message) exchange across different worker threads operating within the platform. More in detail, given that ROOT-Sim manages any subset of simulation objects, say S, by (temporarily) binding them to a specific worker thread, say t, and the adopted CPU-dispatching rule is the classical Lowest-Timestamp-First (LTF), the CPU-dispatched event associated with the objects in S is always the one with highest priority.

The only exception is when messages, or anti-messages, produced while concurrently running other simulation objects along other worker threads, and destined to some object belonging to the set S handled by t, will carry a timestamp which is lower than the one associated with the last-dispatched lowest-timestamp event. Clearly, this cannot be known before the CPU-schedule operation along thread t.

[5]We recall that during the mount operation, the module initialization function rewrites some parts of the kernel code at run-time, which is a critical procedure.

In ROOT-Sim, 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, still oriented to scalability. Particularly, each worker thread manages a set of bottom-half queues (one for each simulation object 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. Checking whether some new data is present into a bottom-half queue, and actual processing of the data with incorporation into the destination event queue, is carried out exclusively by the worker thread in charge of (currently) handling the destination object. This scheme is complemented by a constant-time management of the critical section for inserting/deleting elements into/from any bottom-half queue, which leads actual thread synchronization to scale.

The above scheme has been extended in order to perform the integration with the extra-tick management architecture along the following lines. First, each worker thread t operating in the Time Warp platform has been associated with a BH_min_t record, which represents at any time instant the minimum timestamp of any 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 information in transit (if any), which is destined to some simulation object handled by t.

This record is initialized to the special macro `INFTY` (via a single atomic assignment operation) when the worker thread t accesses its bound bottom-half queues an 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 bound to t to execute a conflicting critical section[6].

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 in case thread t is not currently processing any event (e.g. it is running housekeeping operations within the Time Warp platform).

The values of $current_time_t$ and BH_min_t are used by the callback function that takes control via the extra-tick mechanism in order to determine whether some higher priority task (compared to the one currently processed by the CPU along thread t) needs to be CPU-dispatched. Particu-

[6]In fact, each of them needs to temporarily lock a different bottom-half queue for data insertion, which helps not hampering concurrency [27].

larly, the callback function executing along t has the following simple structure:

---

void **tick-manager()**
1. **if** ($current_time_t \leq BH_min_t$)
2. **return**;
3. **else**
4. `switch_to_platform_context();`

The above structure allows changing the current execution flow along thread t 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 anti-message in its past (namely 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 just based on the time-sharing organization and on the (periodic) regain of control at the Time Warp platform level.

2) Some generic simulation object 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 lowest-timestamp CPU-scheduled event. 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 the CPU-dispatching of the events by any worker thread t.

Overall, in either case, control must return to the Time Warp platform layer, so that the higher priority task (either a rollback operation or not) is promptly executed. In the pseudo-code this is achieved via the invocation of the `switch_to_platform_context()` function, whose actual support, as well as the support for correct management of individual (and separate) simulation object contexts, is described in the next section.

On the other hand, in case no higher priority task needs to be executed, the forward execution of the last CPU-dispatched event is immediately resumed given that the tick-manager callback function simply returns, thus taking control back to the point where the original execution flow was interrupted by the extra-tick at the heart of the time-sharing architecture.

3.4 Support for Context Switches

The management of different per-simulation-object contexts (as well as the platform context) is based on the ROOT-Sim support that has been introduced in [16] to create stack separation across the different simulation objects. This is achieved by locating the stack of each object in a portion of memory destined for object usage (e.g. when memory chunks are dynamically requested while executing events at that object) via proper (and application transparent) allocation layers [16, 26]. On the other hand, each worker thread t also has a platform context associated with it, which is in turn associated with a proper stack area located in a different, and disjoint, memory region.

Execution resume in the different stacks, such as when `switch_to_platform_context()` is executed, has been supported via `setjump` and `longjump` POSIX APIs. They have also been used as the support for, e.g. squashing the stack image of the currently-executing simulation object in case a preemptive rollback occurs within the time-sharing Time Warp system (which eventually leads the object to resume execution with a different context, namely a logged one that is then put back in place).

3.5 Support for Safe Platform Mode Execution

A final core aspect deals with the fact that in an advanced Time Warp platform applications are supported by allowing the actual application code to live in a piece-wise-deterministic environment which is able, transparently to the application code, to support recovery of any incorrect trajectory of the application state possibly caused by causality errors in the speculative execution path.

A classical way to achieve this is the one where any interaction between the application code and external services, such as those offered by standard third-party programming libraries, is intercepted by the Time Warp environment and is handled according to proper rules. Classical examples are the ones where recoverable dynamic memory services and/or recoverable I/O services are exposed to the application via standard interfaces (e.g. `malloc` or `printf`) but are handled (to make them actually recoverable) via proper logic at the level of the Time Warp platform. Such a kind of application-transparent intervention by the platform-level software may reach a granularity so fine that even a single machine instruction can be intercepted and made recoverable, as when relying on binary code instrumentation to run-time track memory writes, and to dynamically log recoverability data so as to retain the possibility to undo the update [18].

According to this view, independently of the presence of any time-sharing support like the one we provide, an advanced Time Warp platform can be already seen as a system working according to a dual-mode execution model, where the application can trap into platform mode just because of some (seamless) access to one of the above mentioned services. On the other hand, the actual implementation of such platform-level services, for being correct, may require atomicity. Just to mention an example, locks on specific data structures or memory regions might be acquired by some worker thread once the application has trapped into platform mode along that thread in order to correctly manage the triggered service [1].

This atomicity must be guaranteed also in case of the time-sharing architecture we provide. Hence, if the tick-management callback is triggered while the running thread has already trapped into platform mode in a seamless manner on behalf of the application code processing the current simulation event, our choice is to avoid any variation of the control flow (independently of the actual presence of higher priority tasks), so as not to interfere with the already entered platform-level code block. In order to achieve this target in ROOT-Sim, we have reorganized the platform-level software such in a way that any entry point for actual platform operations has been augmented by a wrapper that atomically sets a flag indicating that the thread is running in platform mode. The reverse action, which resets the flag to application mode, is actuated via wrappers intercepting the return

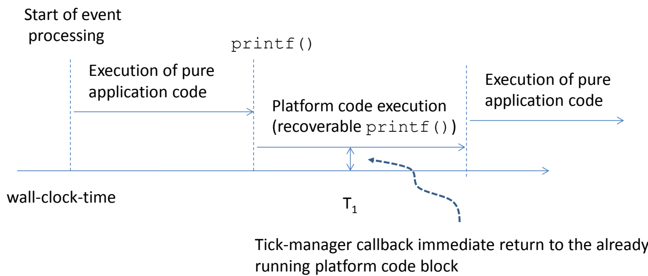

Figure 6: Management of extra-ticks in the interleave between application and platform code blocks within an event processing wall-clock-time window.

of any platform-level service possibly activated while processing some event via the aforementioned trap/interception mechanism. In case a callback for managing some extra-tick is received while running a platform-mode phase during the processing of an event in the application software, then the callback simply returns control to the interrupted execution point.

A schematization of this behavior is provided in Figure 6, where we show the arrival of an extra-tick at wall-clock-time T_1, with consequent activation of the extra-tick management callback function, and where the callback simply returns given that at the same time instant the platform-level software was already handling, possibly via proper recoverability rules, some I/O operation invoked by the event processing code implemented at the application level.

We also note that the approach where no control flow variation is actuated in case the platform mode has been already entered, e.g., because of an interception of some third-party library call while processing the current event by some worker thread, also allows safety of the execution of any platform-level facility ultimately relying on external user space libraries, such as `libc-xx.so`, given that the flow control in these libraries is never changed by any extra-tick arrival.

4. EXPERIMENTAL RESULTS

We have executed experiments with the time-sharing version of ROOT-Sim by running this platform of top of a 32-core 64-bit NUMA machine, namely an HP ProLiant 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 augmented with our extra-tick management architecture. In the original configuration of the kernel, the LAPIC-timer was set to issue an interrupt each 1 millisecond. When running in extra-tick mode we configured the LAPIC-timer to send an interrupt 10 times more frequently, thus each 100 microseconds.

As the benchmark for assessing the effectiveness of our time-sharing proposal, we used a real-world cellular system simulator, which has already been used as a reference benchmark application in a number of other studies oriented to optimistic PDES (see, e.g., [18]). In this application, each simulation object models a wireless cell, and we selected a total number of 1024 cells (represented as hexagons), each one managing 1000 wireless channels, which provide coverage to mobile devices in a squared region. The model is high fidelity in terms of how interference across different channels within a same cell, and power management upon call setup/handoff is captured/actuated. Particularly, the application handles power management simulation according to the results in [13]. The 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 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 for simulating the concurrently active calls in any cells, 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 on-going communication, in which case the wireless channel at the source cell is released, and a new one in the destination cell is attempted to be reserved.

On 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 value 5 min, while the average call duration was set to 2 min. Both these values have been set to follow 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 (fairly distributed on 32 worker threads operating within the ROOT-Sim platform on top of the 32 CPU-core machine), and with periodic recalculations of the fading coefficients of active channels. This settings gives rise to variations of the simulation event's average CPU requirement from about 70/80 microseconds, to about 150 microseconds. This way we achieved differentiated configurations in terms of the relation between the event execution granularity, and the granularity of the extra-ticks' interval (recall this has been set to 100 microseconds). In other words, the adopted settings allowed us to determine different actual likelihoods for an extra-tick to interrupt an on going event (in fact such a likelihood is higher when the event granularity is greater), which gave us the possibility to assess our time-sharing architecture when changing the likelihood that a higher priority task can be detected as standing while the execution of an event is in progress. Further, the configuration with finer granularity of the events (namely the one with 25% channel utilization factor) looks also good for assessing the overhead by our proposal, just given the reduced likelihood for the extra-tick to occur while an event is in progress, rather while we are running any platform-level housekeeping operations, which leads to a case where the extra-tick simply returns control to the original point (but cost has anyhow been spent for delivering it, hence for delivering control to the callback offered by the Time Warp system).

In this experimentation we compare the original execution dynamics of ROOT-Sim, namely those based on the classical batch-multitask paradigm for processing the events (where control is returned to the underlying platform for dispatching other events/tasks only at the end of the event-handler processing routine) with those achieved via the integration

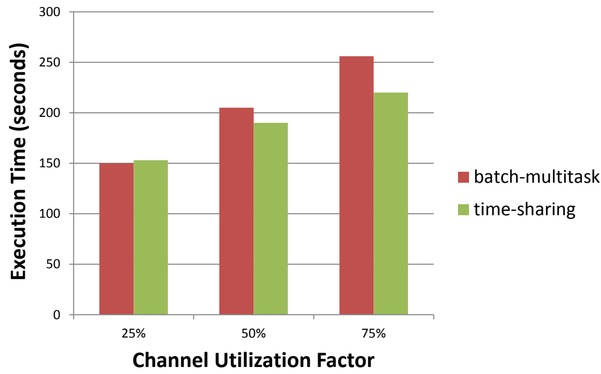

Figure 7: Execution time results.

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

Table 1: Performance of the serial simulator.

of the time-sharing support (which allows for passing control to events/tasks that dynamically reveal as higher priority ones). Particularly, in Figure 7 we report the execution time (computed as the average over 5 different samples, each referring to a different settings of the pseudo-random seeds used for stochastically driving model execution) for simulating a specific virtual time interval in the different configurations of the benchmark application and with either batch-multitask or time-sharing support. Also, we decided to run with checkpoint interval fixed to the value 10 for all the simulation objects in order not to introduce fluctuations and/or variations in performance possibly caused by some adaptive mechanism to select the checkpoint frequency, which might interfere with the actual performance variations natively imputable to the two different execution schemes we are comparatively studying, namely time-sharing vs batch-multitask. By the data we can draw the following main observations. First, the results related to the configuration with 25% channel utilization factor, which as hinted before is essentially useful for overhead assessment, actually show a minimal overhead by the time-sharing support, given that the execution times for the two different supports is essentially the same. On the other hand, as soon as the event granularity is increased (namely for higher values of the channel utilization factor) we get an actual reduction of the execution time achieved with the time-sharing support (as compared to the traditional batch-multitask support). Specifically, the gain in performance is of the order of 8% for the case of utilization factor set to 50%, and of the order of 15% when the utilization factor is further increased up to 75%. For completeness, we also report in Table 1 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 [2], which allows determining the speedup of the parallel runs, and hence whether this study refers to competitive parallel performance.

By the data in Figure 8, we get the explanation of the actual source of the performance gain by the time-sharing support compared to the batch-multitask one. Specifically,

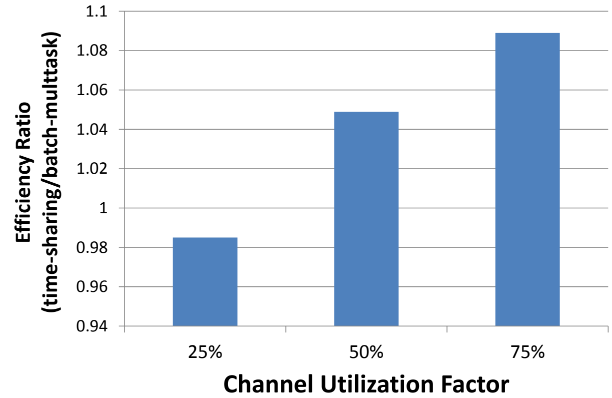

Figure 8: Time-sharing efficiency over batch-multitask efficiency.

we report in this figure the variation of the ratio between the optimistic run efficiency observed with the time-sharing support and the one observed with the batch-multitask support. We recall that the efficiency of an optimistic PDES run represents the percentage of productive work executed (processed simulation events that are not eventually rolled back). Hence it is an expression (and a derivation) of the actual rollback pattern (and amount). By the data we see that, for the configuration with 25% channel utilization factor, the two different supports provide in practice the same efficiency values. This is aligned with our previous observations in relation to the application configuration with finer-event granularity, which exhibits a reduced likelihood of actual interruption of an event processing phase by the extra-ticks, with consequent reduction of the possibility to change the rollback pattern in the time-sharing support by passing control to some higher priority task (e.g. a simulation event with smaller timestamp) dynamically injected in the system. On the other hand, increasing the event granularity leads to scenarios with increased ability by the time-sharing support to actually track (while an event is already being processed in CPU) whether higher priority activities need to be carried out along the same worker thread and dynamically reschedule them in CPU, which leads to limit the negative impact of, e.g. out-of timestamp order processing. This, in its turn, leads the relative efficiency by the time-sharing support to increase, compared to the batch-multitask one, up to 9%. Overall, the time-sharing configuration shows higher likelihood to perform useful (not eventually rolled back) work, especially for the case of larger granularity events, and this is achieved with negligible overhead (by the time-sharing support), with positive effects on performance.

In Figure 9 we provide an additional plot where we show the variation of the amount of actual event preemptions (hence execution flow variations) per execution time unit achieved while running in time-sharing mode for the three different configurations of the channel utilization factor we have considered in this study. Aligned with the previous results, the data reported in this plot show how the configurations with larger event granularity (namely, higher channel utilization factor) manifest a larger percentage of actual preemptions per wall-clock-time unit. On the other hand, the interesting point in this plot is that it does not scale linearly, which is a reflection of the fact that greater event

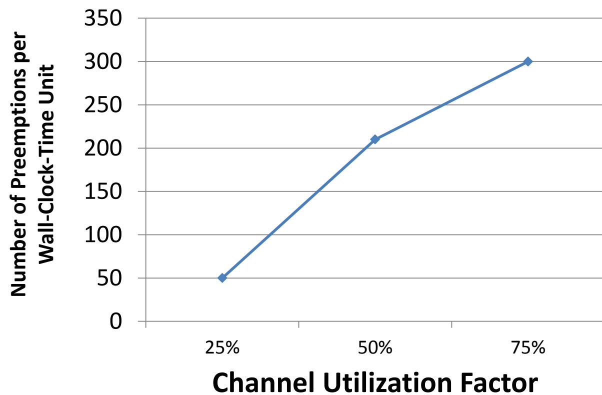

Figure 9: Number of preemptions per wall-clock-time unit when running in time-sharing mode.

granularity on the one hand allows for more opportunities of preemption (since extra-ticks more likely will be delivered while simulation-event processing is in progress), but on the other hand, parallel runs with larger grain events are typically (at least for balanced models like the ones we are considering) less subject to divergence of the simulation clocks of the different concurrent simulation objects, which leads to reduced likelihood that the extra-tick callback function actually finds some higher priority task (e.g. the rollback of the currently running simulation object) to be carried out. However, this somehow no linear shape of the curve in Figure 9 looks intrinsic to the nature of Time Warp dynamics, hence not being a specific limitation of the time-sharing Time Warp approach we have presented.

5. CONCLUSIONS

In this article we have presented a lightweight support for allowing time-shared execution of Time Warp platforms on top of conventional operating systems, such as Linux, and multi-core machines.

In our proposal, any individual worker thread operating within the Time Warp system can be interrupted with high frequency (a frequency much higher than conventional timer interrupts in classical operating systems' configurations) and with low interrupt-management overhead, in order to determine whether any higher priority task (e.g. a rollback to be promptly processed) or event (e.g. with lower timestamp compared to the currently executed one) needs to be CPU-dispatched. This allows for reacting to actual changes of the priorities of the activities to be carried out within the Time Warp run, with consequent (possible) advantages in terms of reduction of the amount of wasted work.

Our operating system based support for time-sharing has been integrated into an open source Time Warp platform, and we also report experimental data for an assessment of both the overhead by our proposal and its final effectiveness in terms of improvements of the execution speed of Time Warp simulations on multi-core machines.

6. REFERENCES

[1] F. Antonacci, A. Pellegrini, and F. Quaglia. Consistent and efficient output-streams management in optimistic simulation platforms. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, (SIGSIM-PADS), Montreal, QC, Canada, May 19-22, 2013*, pages 315–326, ACM Press, 2013.

[2] R. Brown. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[3] C. D. Carothers. Xsim: real-time analytic parallel simulations. In *Proceedings of the Workshop on Parallel and Distributed Simulation (PADS), Washington, D.C., USA, May 12-15, 2002*, pages 27–34, IEEE Computer Society, 2002.

[4] C. D. Carothers, D. W. Bauer, and S. Pearce. ROSS: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[5] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, 1999.

[6] S. R. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: a time warp system for shared memory multiprocessors. In *Proceedings of the 26th Winter Simulation Conference*, pages 1332–1339. Society for Computer Simulation International, 1994.

[7] P. De, R. Kothari, and V. Mann. Identifying sources of operating system jitter through fine-grained kernel instrumentation. In *Proceedings of the IEEE International Conference on Cluster Computing, 17-20 September 2007, Austin, Texas, USA*, pages 331–340, IEEE Computer Society, 2007.

[8] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC), November 15-21, 2008, Austin, Texas, USA*, pages 19:1–19:12, ACM Pess, 2008.

[9] HPDCS Research Group. ROOT-Sim: The ROme OpTimistic Simulator - v 1.0. *http://www.dis.uniroma1.it/∼hpdcs/ROOT-Sim/* (last accessed: May 2015).

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

[11] D. R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. D. Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, L. V. Warren, J. J. Wedel, H. Younger, and S. Bellenot. Distributed simulation and the Time Warp operating system. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles (SOSP), Austin, Texas, November 8–11*, pages 77–93, ACM Press, 1987.

[12] P. D. B. Jr., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp speed: executing Time Warp on 1,966,080 cores. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, (SIGSIM-PADS), Montreal, QC, Canada, May 19-22, 2013*, pages 327–336, ACM Press, 2013.

[13] S. Kandukuri and S. Boyd. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications*, 1(1):46–55, 2002.

[14] D. E. Martin, T. J. McBrayer, and P. A. Wilsey. WARPED: A Time Warp simulation kernel for analysis and application development. In *Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS), Volume 1: Software Technology and Architecture, January 3-6, 1996, Maui, Hawaii, USA*, pages 383–386. IEEE Computer Society, 1996.

[15] D. M. Nicol and X. Liu. 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), Lockenhaus, Austria, June 10-13, 1997*, pages 188–195, IEEE Computer Society, 1997.

[16] A. Pellegrini and F. Quaglia. Transparent multi-core speculative parallelization of DES models with event and cross-state dependencies. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, (SIGSIM-PADS), Denver, CO, USA, May 18-21, 2014*, pages 105–116, ACM Press, 2014.

[17] A. Pellegrini, R. Vitali, and F. Quaglia. The ROme OpTimistic Simulator: Core internals and programming model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools), Barcelona, Spain, March 22 - 24*, pages 96-98, ICST, 2011.

[18] A. Pellegrini, R. Vitali, and F. Quaglia. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Parallel and Distributed Systems (preprint)*, May 2014, doi:10.1109/TPDS.2014.2323967.

[19] B. R. Preiss, W. M. Loucks, and D. MacIntyre. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, 1994.

[20] P. Putnam, P. A. Wilsey, and K. V. Manian. Core frequency adjustment to optimize Time Warp on many-core processors. *Simulation Modelling Practice and Theory*, 28:55–64, November 2012.

[21] F. Quaglia. A cost model for selecting checkpoint positions in Time Warp parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):346–362, 2001.

[22] F. Quaglia and V. Cortellessa. On the processor scheduling problem in Time Warp synchronization. *ACM Transactions on Modeling and Computer Simulation*, 12(3): 143-175, 2002.

[23] A. Santoro and F. Quaglia. Software supports for event preemptive rollback in optimistic parallel simulation on myrinet clusters. *Journal of Interconnection Networks*, 6(4):435–457, 2005.

[24] S. K. Seal and K. S. Perumalla. Reversible parallel discrete event formulation of a tlm-based radio signal propagation model. *ACM Transactions on Modeling and Computer Simulation*, 22(1):4:1–4:23, 2011.

[25] S. Seelam, L. L. Fong, A. N. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea. Extreme scale computing: Modeling the impact of system noise in multicore clustered systems. In *Proceedngs of 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS), Atlanta, Georgia, USA, 19-23 April 2010*, IEEE Computer Society, pages 1–12, 2010.

[26] R. Toccaceli and F. Quaglia. 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 (PADS), Roma, Italy, June 3-6*, pages 163–172. IEEE Computer Society, 2008.

[27] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS), Zhangjiajie, China, July 15-19*, pages 211–220. IEEE Computer Society, Aug. 2012.