Exploiting Inter-Processor-Interrupts for Virtual-Time Coordination in Speculative Parallel Discrete Event Simulation

Emiliano Silvestri
silvestri@diag.uniroma1.it
Sapienza, University of Rome
Italy

Cristian Milia
miliacristian4@gmail.com
University of Rome "Tor Vergata"
Italy

Romolo Marotta
marotta@diag.uniroma1.it
Sapienza, University of Rome
Italy

Alessandro Pellegrini
pellegrini@diag.uniroma1.it
Sapienza, University of Rome
Italy

Francesco Quaglia
francesco.quaglia@uniroma2.it
University of Rome "Tor Vergata"
Italy

ABSTRACT

Reducing the waste of resource usage (e.g., CPU-cycles) when a causality error occurs in speculative parallel discrete event simulation (PDES) is still a core objective. In this article, we target this objective in the context of speculative PDES run on top of shared-memory machines. We propose an Operating System approach that is based on the exploitation of the Inter-Processor-Interrupt (IPI) facility offered by off-the-shelf hardware chipsets, which enables cross-CPU-core control of the execution flow of threads. As soon as a thread T produces a new event placed in the past virtual time of a simulation object currently run by another thread T', our IPI-based support allows T to change the execution flow of T'—with very minimal delay—so to enable the early squash of the currently processed (and no longer consistent) event. Our solution is fully transparent to the application level code, and is coupled with a lightweight heuristic-based mechanism that determines the actual goodness of killing thread T' via the IPI (rather than skipping the IPI send) depending on the expected residual execution time of the incorrect event being processed. We integrated our proposal within the speculative open-source USE (Ultimate Share Everything) PDES package, and we report experimental results obtained by running various PDES models on top of two shared-memory hardware architectures equipped with 32 and 24 (48 Hyper-threads) CPU-cores, which demonstrate the effectiveness of our proposal.

CCS CONCEPTS

• **Computing methodologies → Discrete-event simulation**; *Shared memory algorithms*; • **Theory of computation** → Parallel computing models.

KEYWORDS

Parallel Event Discrete Simulation, Shared memory, Multi core, Operating Systems

ACM Reference Format:
Emiliano Silvestri, Cristian Milia, Romolo Marotta, Alessandro Pellegrini, and Francesco Quaglia. 2020. Exploiting Inter-Processor-Interrupts for Virtual-Time Coordination in Speculative Parallel Discrete Event Simulation. In *Proceedings of the SIGSIM Principles of Advanced Discrete Simulation (SIGSIM-PADS '20), June 15–17, 2020, Miami, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3384441.3395985

1 INTRODUCTION

Speculative execution of parallel discrete event simulations is a mean to achieve high performance and scalability, especially for models with limited (or null) lookahead [13]. On the downside, it may suffer from waste of computation, caused by the rollback mechanism used to overcome mis-speculation (i.e., out of timestamp order processing). To cope with the reduction of the resource waste, several approaches have been proposed in the literature, which range from (1) load balancing [1, 4, 28]—used to reduce the likelihood of divergence in the advancement of concurrent simulation objects along virtual time—to (2) optimized CPU-scheduling [25]—used to punctually select the best suited simulation object to be advanced in virtual time—to (3) throttling [26]—used to artificially delay the processing of events so as to avoid uncontrolled speculation—to (4) optimized rollback management schemes [2, 23]—aimed at providing suited tradeoffs between the overheads associated with the rollback support and the actual rollback tasks.

However, none of the above-mentioned solutions entails the capability to "promptly react" to the occurrence of rollback. Hence, none of them is based on the timely interruption of the execution of events that are no longer causally consistent, but are still being processed by some thread when the causality error materializes. More recently, some solution has been proposed [22] which is aimed at preempting the event processing phase when the event is detected as no longer consistent. However, the preemption mechanism is based on a polling scheme actuated via the exploitation of hardware-level timer interrupts, which periodically checks the event consistency.

The limitation of this approach is that it cannot show an arbitrarily fine granularity—otherwise the polling overhead would reveal unaffordable. Furthermore, just a reduced percentage of polls will actually reveal useful in the early detection of a causality inconsistency. On the other hand, a larger grain tends to reduce the benefits in performance since late detection of inconsistencies increases the amount of CPU time spent for doomed to be rolled-back work[1]—especially in scenarios with non-minimal incidence of rollbacks.

In this article we explore an orthogonal approach to the early abort of no longer consistent work, which is based on the Inter-Processor-Interrupt (IPI) facility, supported by conventional chipsets by all the major vendors (Intel, AMD and ARM). Our approach is suited for speculative PDES run on top of shared memory multi-core machines, where an IPI can be sent by a CPU-core (or hyper thread) selectively to any other CPU-core on board of the same machine. The advantages of our proposal, compared to preempting the execution of a no longer consistent event in the basis of a polling scheme, are multiple:

- We pay the cost of hardware level cross-CPU-core coordination only if the CPU-core, which is the destination of the IPI, is actually running inconsistent work. In other words, if no causality error materializes, then no overhead at all is payed for virtual-time coordination via the IPI support. In contrast, the hardware-timer interrupts used by the polling approach produce overhead even with rollback free runs.
- The IPI-based mechanism allows for an "almost immediate" preemption of the doomed to be rolled-back work, since the delivery of the IPI to the CPU-core that is running the inconsistent event takes place in very few microseconds. In contrast, in the polling approach the thread running the inconsistent event will detect the inconsistency (and will preempt the event) only upon the expiration of the polling interval.
- Compared to the polling approach, we reduce the effect of the event-preemption support on the locality of the access to memory, thus indirectly improving performance. In fact, the polling mechanism is based on a control flow variation, passing control to the consistency check routine, which is unfavorable to locality, even if no causality violation is in place. In contrast, the IPI-based scheme does not induce any thread control flow variation if no causality violation occurs.

Furthermore, our mechanism is coupled with a heuristic-based decision model that allows detecting whether sending the IPI towards a CPU-core running an inconsistent event can be actually fruitful, compared to letting the thread go ahead and complete the event processing routine. The decision relies on threshold based mechanisms that take into account both the expected residual CPU-time for completing the event processing phase, as well as runtime collected statistics related to the types (and expected granularity) of the events being processed by the threads running within the PDES platform.

Our solution is targeted at x86 processors and Linux, although the approach we take can be easily ported to other architectures and Operating Systems. Also, our software, which has been integrated within the USE (Ultimate Share Everything) simulation platform [12], is available for free download[2].

The remainder of this article is structured as follows. In Section 2 we discuss related work. Our IPI-based approach and its integration within the USE platform are presented in Section 3. Experimental data for an assessment of the proposed solution are provided in Section 4. Conclusions are drawn in Section 5.

2 RELATED WORK

As hinted, several proposals in the literature have addressed the issue of reducing the negative incidence of rollbacks on the performance of speculative PDES systems. Well balancing the workload across concurrent threads [1, 4, 12, 28] relies on the idea that diminishing the divergence in the advancement of concurrent simulation objects in virtual time provides the expectation of a reduction of the frequency of causality errors. However, these proposals do not target the reduction of the resource waste when a rollback occurs. In other words, they try to prevent rollbacks, but they do not entail mechanisms to preempt doomed to be rolled-back work carried out by threads. A similar consideration applies to all the proposals based on throttling (e.g., [26]) or those based on bounding speculation via synchronization schemes that are not purely optimistic (e.g., [7, 21]). Since the artificial delay, or the temporary block, is imposed prior to processing events that are identified (or estimated) to be more far from the commit horizon of the simulation run, these solution are not complemented with highly reactive mechanisms to be triggered when an already CPU-dispatched event, which is still being processed, becomes no longer causally consistent. All these solutions are essentially orthogonal to what we propose in this article.

Another way of limiting the overhead caused by rollback is based on adopting smart strategies for selecting what event (hence what simulation object) should be dispatched along a thread as soon as this thread completes the processing phase of another event. Lowest-Timestamp-First scheduling [17] is a classical reference, which is however agnostic of the CPU-demand by the events. Therefore it can lead to suboptimal choices in scenarios with non-minimal variation of the CPU-time required to process events that have very close timestamps. The proposal in [25] takes into account the event granularity, and does not favor coarser grain events in the CPU-scheduling decision when compared to finer-gain events that have timestamps falling in a given (short) virtual time window left delimited by the lowest-timestamp pending event. This method is essentially based on less promptly starting the execution of events that, once CPU-dispatched, might produce a longer CPU-burst of activities that can in any case be invalidated by a causality error. However, this solution has been devised for scenarios where, once CPU-dispatched, the event cannot be preempted. Rather, we propose a solution where preempting (and squashing) the processing of an event with close-to-zero delay with respect to the actual materialization of its causality inconsistency is the core target.

The usage of event-preemption to avoid continuing the processing of an event that becomes causally inconsistent has been adopted in [22]. However, this solution is based on polling, which is activated through hardware-level timer interrupts. In more detail, when a thread running in the speculative PDES platform is CPU-scheduled by the Operating System, the system timer is set to issue interrupts with a fine grain period. Each interrupt changes the execution flow of the thread so that a consistency-check routine is executed, to verify if the currently processed event is no longer consistent. The drawback of this solution is that it pays the cost

[1]With "doomed to be rolled-back work" we indicate an event that is no longer causally consistent, but is currently being processed.
[2]Available at https://github.com/HPDCS/USE

of hardware level activities, and control flow variation along the thread, even in scenarios where no causality error occurs. Also, the period of the system-timer interrupt cannot be set arbitrarily short, otherwise there would be a risk of thrashing, which would impair the progress of the execution. Overall, in the proposal in [22] preemption cannot be actuated with extreme reactivity with respect to the instant at which the event becomes inconsistent. Our approach solves all these issues since we resort to hardware level specific activities—the sending of the IPI—and to the activation of the thread control flow variation only if the causality violation occurs. Further, the delay for propagating the IPI to the destination CPU-core and for handling it at the destination thread is very minimal, and can largely pay off especially for simulation models with events (or event types) showing non-minimal granularity. Saving "almost all" the residual execution cost of these events when they are no longer consistent can enable a significant reduction of the resource waste. Hence, with our proposal, the resources within the computing platform can be more effectively exploited for useful work.

The proposal in [14] is based on the notion of preemption. However, it is not applied to events being processed, rather it is applied to simulation objects. In particular, in this solution the simulation platform gives control to a simulation object for processing its events, up to the point where an incoming event with lower timestamp is detected. In this case, the simulation object is preempted, with the meaning that it is not granted the CPU for processing additional events, and the CPU is reassigned to the simulation object destination of the lowest-timestamp event—or to that same simulation object if it needs to rollback. In this solution, an inconsistent event being processed is never promptly interrupted via hardware level mechanisms to save doomed to be rolled-back work, as instead we do in our solution.

The approach in [19] is based on message broadcasts into groups of simulation objects. These are used as a sort of signaling mechanism to notify that some event that can have generated a chain of other events destined to the group members, is no longer consistent. Therefore, that chain of events is no longer consistent too. The broadcast allows the recipient simulation objects to early rollback, if they have already been affected by a causality error along that chain of events, or to avoid processing inconsistent events belonging to the chain. This solution still tackles the issue of reducing the impact of causality errors, and the associated waste of resources. However, it does not support the interruption of a doomed to be rolled-back event that has been already CPU-dispatched, which is instead the target of our proposal.

Other mechanisms exist for optimizing speculative PDES, which typically include the combined optimization of the costs for the rollback tasks and for all the tasks that enable correct rollbacks. Here, we find proposal in the fields of checkpointing and reverse computation [2, 5, 23, 24], as well as the design and the implementation of housekeeping tasks that are typical of speculative PDES platforms (like Global Virtual Time computation [10], messaging [3, 27] and memory management [6, 23]). Our proposal is still orthogonal with respect to these solutions, and can be combined with them.

The IPI technology is largely used in Operating System kernels (e.g., the Linux kernel) and is exploited to manage various kinds of tasks, such as (1) making the address space of a process consistently accessible by all its threads (in this case the IPI is used to notify other CPU-cores of changes in the mapping or access rules to virtual memory caused by a thread running on another CPU-core; this leads the killed cores to flush their TLB for rejuvenating its content on the basis of page-table updates), (2) enabling all (or a subset of) the CPU-cores to be notified that a given function needs to be executed by them (this is the classical SMP-function call kernel-level support), or (3) to kill a CPU-core in order to enable it to execute a CPU-reschedule (this is typical of scenarios where the Operating System kernel needs to promptly CPU-reschedule a thread on a given CPU-core because of something that happened on another CPU-core, namely the sender of the IPI). However, to the best of our knowledge the IPI technology has not been exploited to provide solutions useful for a specific application context—PDES in our case—or more generally for coordinating the execution of threads in the presence of speculative computation. Overall, we provide a solution exploiting IPI, which is different in nature with respect to existing ones.

Finally, we note that the achievement of our same objective would have not been possible by relying on the conventional Operating System support for changing the execution flow of threads asynchronously, like POSIX signals. In fact, the delivery of a signal to a target thread would lead it to change its execution flow—and possibly squash its current activity if it were no longer causally consistent—only upon the next kernel-to-user mode passage. This may not happen promptly since the thread might not call system-calls or be interrupted by hardware devices (including the system timer) for a while. Instead, our solution meets the requirements of reduced overhead and close to immediate reaction by the destination thread currently executing doomed to be rolled-back work.

3 OUR APPROACH

3.1 Outline

We provide the outline of our solution by relying on x86/Linux terminology. However, as already hinted, our design can be easily ported to processors supporting a different Instruction Set Architecture (ISA) and to other Operating Systems. At the architectural level, the IPI-based approach we propose spans across both user and kernel code. Each of the two parts is discussed in what follows.

3.1.1 Kernel-level Facilities. At kernel level, we have setup the following two facilities:

- a new system call to allow a thread *A*—the one that detects the inconsistency of the event currently executed by another thread *B*—to send the IPI towards the CPU-core (or hyper thread) running *B*. This activity cannot be executed at user level since the send of the IPI requires executing a machine instruction that writes a specific bitmask on a Model-Specific Register (MSR) of the x86 processor, which is only accessible when running in kernel mode;
- a new interrupt handler used to process the IPI at the destination side, which is in charge of changing the execution flow of the hit thread *B*. This brings it to run a routine, included in the user space PDES platform, which can lead to squashing the context related to the processing of the inconsistent event.

The above two points have been solved via a loadable Linux-kernel module. The first one has been addressed by searching one free entry (i.e., an entry pointing to the `sys_ni_sys_call`) within the system-call-table, and linking this entry to the kernel-level routine included in our module, which actually implements the new system-call called `sys_hit_by_ipi`. This approach is perfectly compatible with the recent Page-Table-Isolation (PTI) patch [11] that contrasts security attacks based on hardware-level speculation, like Meltdown [18] and Spectre [16]. In fact, our new system-call is still dispatched by the original dispatcher used for all the other system-calls[3].

More tricky is the solution we adopted for addressing the second point listed above. In particular, simply linking a new interrupt-handler to some entry of the Interrupt-Descriptor-Table (IDT) used by x86 processors would not lead to a solution compatible with the above mentioned security oriented patch. The new handler is in fact contained in the external module—hence it is not compiled in the kernel image—therefore it is not included in the region of virtual addresses that are left mapped by the security patch on the page-table used by an active process when running in user mode. Therefore, as soon as thread *B* would be hit by the IPI when running some no longer consistent simulation event in user mode, an unrecoverable error will occur. To avoid this, we adopted a binary-patching approach that exploits the modular interrupt architecture used in the Linux kernel. In particular, as shown in Figure 1, the actual handling of interrupts in Linux is based on a two level organization where a top-level routine is linked to the IDT—this is the routine that is started when the interrupt is really accepted by the processor—while the actual logic for handling the interrupt is not directly linked to the IDT and is only triggered by the top-level routine (if this gets executed). The top-level routines for all the predefined handles are included in the kernel compilation and are left mapped to the page-table of any process also when running in user mode, so that these routines will be reachable when an interrupt occurs on a CPU-core while running in user mode. We exploited the top-level routine for the so called spurious interrupts—which are never used by the kernel—and applied a binary-patch where the original call instruction to the spurious interrupt logic is substituted with a call to the logic implemented in our module, which applies the control flow variation. This is also shown in Figure 1. With this solution, the top-level routine is left in charge of handling the actions related to the security oriented patch.

Clearly, it is possible that in the interval of clock cycles between the decision of thread *A* to hit thread *B* via the IPI, and the send of the IPI to the destination CPU-core through the new system-call, *B* gets temporarily descheduled in favor of another thread by the Operating System. Hence, we embedded in our architecture a mechanism to let the handler of the IPI know if the thread currently running on the hit CPU-core needs to really change its execution flow—or is a thread running outside the PDES platform. The mechanism we exploited has been based on a special device-file driver, still supported by our loadable Linux-kernel module, which offers an `ioctl` operation that enables all the threads living within the PDES platform to register their ID at kernel level, as well as on

[3]The security oriented patch that implements PTI for the case of access to the kernel via system-calls is in fact fully managed by this dispatcher.

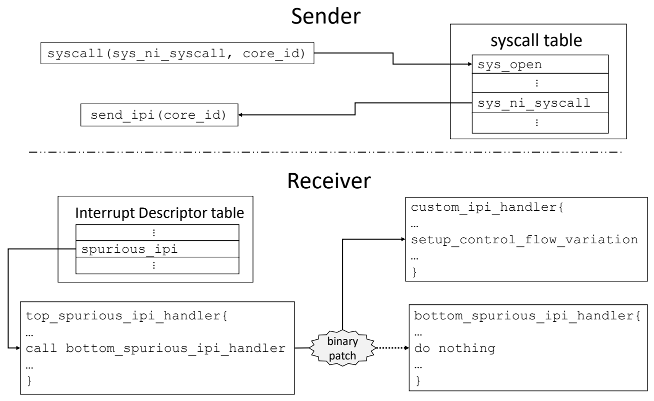

Sender
syscall(sys_ni_syscall, core_id)
syscall table
sys_open
sys_ni_syscall
send_ipi(core_id)
Receiver
Interrupt Descriptor table
spurious_ipi
custom_ipi_handler{
…
setup_control_flow_variation
…
}
top_spurious_ipi_handler{
…
call bottom_spurious_ipi_handler
…
}
binary patch
bottom_spurious_ipi_handler{
…
do nothing
…
}

Figure 1: The patching scheme of the spurious interrupt handler.

what CPU they are pinned. This way, the handler of the IPI can check if the currently running thread is registered and pinned to the IPI recipient CPU-core. In the negative case it does not apply the control flow variation—rather, it sets a flag associated with the real destination thread. In our solution, this flag is checked when this thread is rescheduled on CPU by the Operating System—this is achieved by installing a hook at the tail of the kernel `schedule` function via the Linux `kprobe` service. A positive check by the hook will lead to the control flow variation enabling the thread to squash the inconsistent work it was running prior to its CPU-deschedule by the Operating System.

3.1.2 User-level Facilities. While devising the user-level facilities that exploit the IPI support, a core problem to solve has been to avoid the send of multiple un-needed IPIs in scenarios where multiple threads concurrently detect that the event *e* being processed by a thread *X* is no longer consistent. Further, we also need to correctly identify what CPU-core we need to hit with the IPI[4] and to setup fresh information to allow concurrent threads to capture the current activities of other threads. To cope with these issues, we devised the usage of metadata associated with each individual simulation object, which allow:

- identifying the CPU-core that is currently hosting the thread which is processing the event *e*, as well as the timestamp of *e*;
- identifying if some other concurrent thread has already detected that the event e is no longer consistent, and has taken care of triggering the IPI send towards the destination CPU-core;
- keeping information useful for determining whether an IPI can currently lead to a safe squash of a no longer consistent event—namely whether the control flow variation possibly induced by the IPI arrival is compatible with the actual execution flow of the hit thread.

In Figure 2 we show the structured variable we associated with each individual simulation object. The variable represents the simulation-object *control table* within the IPI-based solution. We

[4]We recall that the IPI technology works at the level of the processor firmware and is agnostic of threads and threads' IDs.

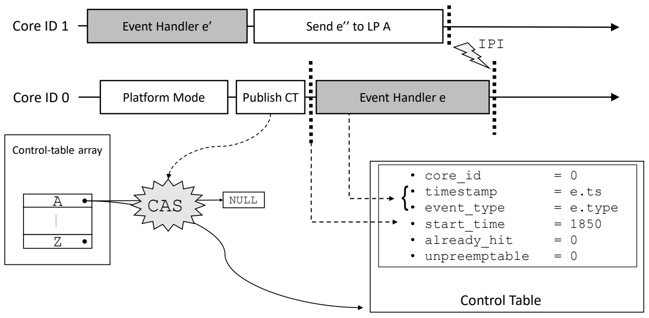


Figure 2: The control-table data structure.

recall that our approach works in shared-memory multi-core machines, so the control table associated with each individual simulation object is accessible in read/write mode by all the threads running in the multi-threaded PDES platform. Its fields are as follows:

- The `core_id` field is simply the numerical ID—according to ACPI indexing—of the CPU-core that is currently running an event destined to the simulation object.
- The `timestamp` field denotes the timestamp of the simulation event that is currently being processed at that simulation object (or the one that needs to be processed after this simulation object has finished a rollback phase).
- The `event_type` field denotes the type of the event being processed—useful to determine whether to hit it via IPI or not, as we shall discuss.
- The `start_time` field keeps the wall-clock-time at which the event e has been started processing.
- The `already_hit` field is used to denote if some thread has already revealed that the current event being processed by that simulation object (the one associated with the current content of the control table) is no longer consistent.
- The `unpreemptable` field is a control field—particularly a counter—used to determine whether the processing of e can be safely squashed—an aspect that we shall discuss in a while.

With each simulation object O we associate a pointer `ct[`O`]` into the array `ct[]`, which can be either `NULL` or can point to O's control table. When a thread picks the event e to be processed at the object O then it initially fills all the fields of a fresh instance of the control table, and right before invoking the event handler with e as input, it updates the pointer `ct[`O`]` to point to the fresh control table via an atomic memory update instruction, which essentially acts as a memory-fence. In particular, we used the Compare-and-Swap (CAS) `x86` instruction (named `CMPXCHG`). This ensures that the control table has a consistent content visible to all the other concurrent threads as soon as they catch a not null value of the `ct[`O`]` pointer. Also, the thread that updates the control table after picking the event e is able to publish towards all the other threads the control table by avoiding lock protected critical sections, hence enabling scalability. Clearly, as soon as the thread finishes working on the event e, it resets the pointer to the value `NULL` still via a CAS instruction and the buffer hosting the no longer active control table can be collected for later usage according to well-known schemes, such as Epoch-Based Reclamation [8] or Hazard Pointers [20].

As for `core_id`, we already hinted that in our solution we resorted on CPU-affinity mechanisms which pin each thread working in the PDES platform to a different CPU-core. Therefore, the ID of the CPU-core to be posted by a thread on the control table is deterministically known at startup of the PDES run. However, we are currently working on the definition of efficient mechanisms enabling the dynamic update of the control-table information under scenarios where the Operating System is allowed to migrate threads dynamically among different CPU-cores. As for `start_time`, we populate it by reading the real-time-clock register on board of the `x86` processor—also known as Timestamp Counter Register (TSC)—via the `RDTSC` machine instruction. This is the very last machine instruction we execute prior to the CAS that publishes the control table and the actual call to the event handler, so that the value registered in `start_time` is likely a good approximation of the real wall-clock-time at which the processing phase of the event has started.

As hinted, the shared-memory nature of our target speculative PDES system enables any other thread that has processed an event which generated some other event e' in the past of the event e currently being processed at the simulation object O to know that the causality error has happened. In fact, this can be simply tracked by comparing the timestamp of e' with the value of the `timestamp` field registered in the control table associated with the object O. In this case, the thread that produced e' attempts to update the `already_hit` field in the control table via a CAS machine instruction. If it does not fail, it means that this thread is the one that needs to notify the causality violation to the destination core via the apposite system-call we introduced. Otherwise, a failure in the CAS indicates that some other thread already notified the destination core of the causality inconsistency via an IPI, and there will be no need to re-send the IPI in order to preempt the processing of the inconsistent event.

After the control table has been published, it is possible that the execution flow will pass through functions—called by the event handler—that need to be executed according to an all-or-nothing semantic. An example is the one of memory allocation functions managed by (recoverable) memory allocators, which need to correctly manage metadata in a non-preemptable manner, otherwise the memory allocator would be left in an inconsistent state (possibly not able to guarantee rollbackability of the allocation/deallocation operations). Here, the `unpreemptable` counter plays a crucial role. For all these functions, we offer wrappers such that at the entrance of whichever of them we increment the `unpreemptable` counter, still atomically via the Fetch-and-Add (FAD) instruction. Then, we decrement the counter (again atomically via FAD) right before returning. In this case, when an IPI hits the currently processed event (which is no longer consistent) the first action done by the interrupted flow is the one of checking if the event can be immediately squashed. This is done by checking if the `unpreemptable` counter is currently set to zero. If the check is negative, it means that we are currently running within one (or a chain) of calls to non-preemptable functions and we cannot squash the event execution immediately. However, the function-return wrapper is structured in such a way to check, after decrementing `unpreemptable` via FAD, if its value is zero, and if the `already_hit` field has been set. If both checks are true, it means that the event is no longer consistent and we can perform the preemption just upon the return from the currently executed non-preemptable function—which allows saving the whole remaining portion of the doomed to be rolled-back work.

In this case the wrapper squashes the event processing context, making the thread land to a PDES platform context which allows it to resume housekeeping operations at the level of the simulation kernel. A scheme where this "deferred squash" of a no longer consistent event is adopted is provided in Figure 3. It shows an example time line where the core with id 0 is executing the event handler targeting a specific simulation object. When the corresponding thread invokes an unpreemptable function (e.g., a standard library routine), it increases the `unpreemptable` counter by one. Consequently, if it receives an IPI since another core has detected a causality violation, no control flow variation takes place. However, the `already_hit` bit in the control table of the simulation object dispatched on core 0 will be set to 1. As soon as the thread completes the call to an unpreemptable routine, our wrapper is invoked. Here, the counter will be set again to 0 and, since the `already_hit` bit is set, it applies the control flow variation, squashing the current event management and giving control again to the simulation kernel.

As a last note, if the IPI arrives late at the destination thread, with respect to the processing of the event e to be hit, the varied execution flow still works correctly, since it simply checks whether an event is currently being processed—possibly different from e—and if this event is still consistent (this information is in the control table of the currently dispatched simulation object). In the positive case, the interrupted execution flow is simply resumed. Otherwise, it is squashed anyhow.

3.2 Runtime IPI-send Decisions

An additional relevant point in our solution has been the introduction of a runtime decision support to determine the actual usefulness of sending the IPI towards a CPU-core that is currently running doomed to be rolled-back work. Clearly, such usefulness depends on several factors, among which we can consider the following:

A) the expected granularity of the inconsistent event currently being processed;
B) the expected residual processing time of this event.

Even though the two metrics above are somehow correlated, they have been exploited in differentiated manners in our solution. In more detail, we keep track of the average CPU-time required to process events of any given type, as defined by the simulation model programmer. This is done at the simulation kernel level by still relying on the TSC register and the `RDTSC` instruction, used to determine both start and end time of the event handler routine so as to take an individual sample of the CPU-time demand by the event of that type. For each event-type we have therefore a tuple $< type, expected\_granularity >$, where the expected granularity is computed as the Exponential Moving Average (EMA) over the collected samples (with parameter α, the weight of old sample, set to 0.2). The usage of EMA allows us to capture scenarios where the activities at the level of the simulation model implementation are non-stationary, so that the granularity of the events of a given type can change along the simulation run. Also, the above tuple can be associated with each individual simulation object, so that we can estimate the expected granularity of events of a same given type occurring at different simulation objects. This helps in scenarios where the simulation model is not symmetric, so that simulation objects may perform computations with different CPU-demand when an event of a given type takes place at different objects.

Given that the control table for the management of the IPI-based solution keeps the `event_type` field, when some thread determines that an event e currently processed on a given CPU-core—which is destined to simulation O—is no longer consistent, we can exploit the above statistical information to determine the expectation in terms of CPU-demand by the event, denoted as $CPU(e)$, and we can compare it with a threshold value TR.

If $CPU(e) \leq TR$, then we can skip sending the IPI towards the CPU-core currently processing the event e. The motivation behind this choice is essentially related to the fact that hitting a CPU-core that is processing a fine grain event e to save the residual execution cost of e will probably not pay off—being the event too fine grain. Clearly, this approach requires the determination of a meaningful value for TR, which in our experiments has been set to 10 times the delay for delivering the IPI to the destination core. However, we note that the IPI delivery delay (including the cost for calling the system-call that issues the IPI) is of the order of 1/2 microseconds for common chipsets (we obtained these values for the case of both Intel Xeon and AMD Opteron, which are the platforms used for the experiments whose results are reported in Section 4). Hence, opting for sending the IPI if the expected granularity of the event to be preempted is at least of 10 microseconds seems reasonable to possibly save a non-minimal amount of CPU-waste.

On the other hand, while the solution above allows avoiding the send of the IPI when we are confident that no significant revenue will come out, we cannot be sure that for events with CPU-demand larger than TR we can actually take advantage by sending the IPI. In fact, the event to be hit might have already been executed almost completely. Hence the attempt to squash it via the IPI would mostly result in overhead. To cope with this problem, we exploit the `start_time` field within the control table, which allows us, in conjunction with the expected event granularity, to compute the expected residual CPU-time of the event.

Specifically, the thread that detects the inconsistency of e can read the TSC register, to assess the current wall-clock-time and what portion of the event e has been already processed. Consequently the expected residual can be easily computed. If the residual, which we denote as $RES(e)$, is larger than another threshold TR', then the IPI is sent. As for TR', we suggest the following setup:

$$TR' = \beta \times TR \ with \ \beta \geq 1 \quad (1)$$

This way, the expected CPU-time residual of the event to be hit via the IPI needs to be at least equal to the minimum granularity of the events that we consider eligible for being hit—so they are not too fine grain—if they are found to be inconsistent.

As a final note, the timing information used in our scheme is based on a hardware-level timer (the TSC register) which does not account for time-sharing of threads on the same CPU-core. Therefore, when a thread decides to send the IPI at a given wall-clock-time T it is possible that it gets descheduled by the Operating System, and then rescheduled after a delay, so that the IPI send will

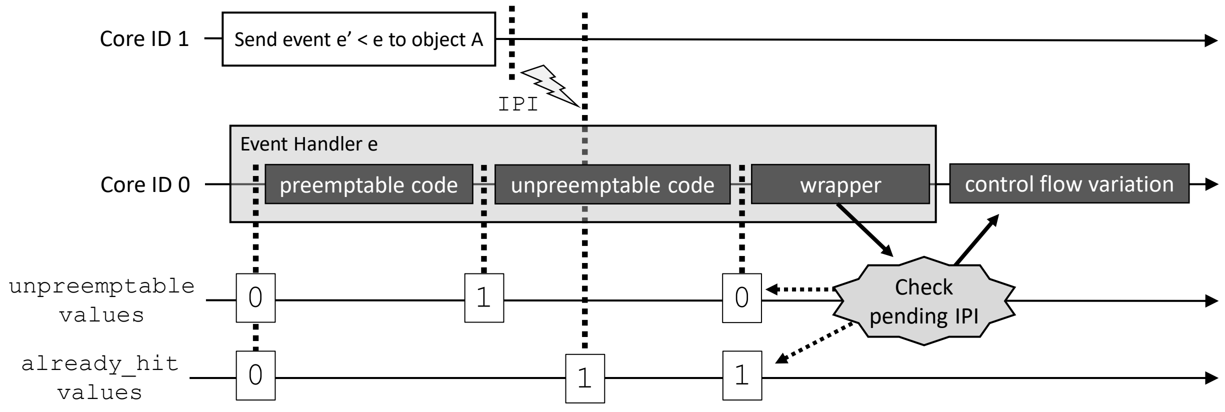


Figure 3: A timeline with IPI and unpreemptable functions executed while processing the no longer consistent event.

not occur on time. In this case, we are simply paying the cost of an IPI that will not arrive on time for hitting the target inconsistent event. However, in scenarios where there is not strong competition for CPU usage by the threads running in the PDES platform and other threads, this situation is expected to be unlikely. The same happens to the thread that is running the event e to be hit, since it is possible that it gets descheduled by the Operating System while in this processing phase. In this case, we have a favorable scenario to the IPI since there will be more time for the IPI to arrive and to flag the unscheduled thread (see Section 3.1.1) before this thread is actually given the opportunity to execute the residual portion of the event.

4 EXPERIMENTAL EVALUATION

In order to assess the effectiveness and the costs associated with our proposal, we have performed an extensive experimental evaluation that includes both synthetic and real-world simulation models. A description of the hardware/software environment used for our experiments is provided in Section 4.1. The simulation models used to evaluate different aspects of our proposal, along with the associated performance results, are presented in Section 4.2.

4.1 Test-bed Platform

Our proposal has been embedded into the *Ultimate Share-Everything Simulator* (USE) [12], a highly-optimized PDES engine for shared-memory multi-core machines, which provides non-blocking progress in both virtual and wall-clock time. The latter is guaranteed by exploiting fine-grain synchronization techniques that confine critical sections to the execution of individual atomic instructions, ensuring scalability while accessing shared data and metadata within the simulation engine. The former is achieved by having an innovative implementation of the Time-Warp protocol, which is highly optimized for maximum exploitation of the capabilities of shared-memory machines. The key characteristic of USE is the presence of a unique pool of events to be processed, which is shared among all the compute units (worker threads) of the runtime environment. This guarantees that the computing power is always assigned to the processing of highest-priority events, thus reducing the occurrence of causality violations, and consequently

Table 1: Hardware Evaluation Platforms

	Platform	
	A	**B**
Processors	4 x AMD Opteron Processor 6128	2 x Intel Xeon E5-2650v4
Cores (Logical)	32	24 (48)
NUMA Nodes	8	2
RAM	64GB	128GB
Operating System	Debian 5.4.19	Ubuntu 19.04
Linux Kernel	v5.4.0	v5.0.0
IPI latency	$1\mu s$	$1\mu s$

improving the overall efficiency. Therefore, adopting USE as test-bed platform allows us to evaluate benefits and shortcomings of our approach in a worst case scenario, namely when it is deployed within an engine characterized by a generally low incidence of rollback and, hence, to be rolled-back work. On the other hand, the two hardware platforms A and B, whose details are given in Table 1, provide an adequate level of parallelism (32 and 48 hardware threads respectively).

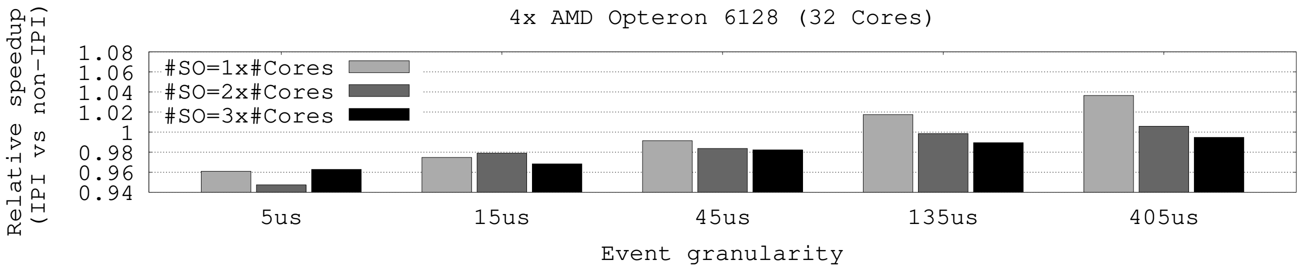
The Operating System running on the machines exposes the IPI API through a loadable kernel module[5] that we have developed according to the patching scheme described in Section 3.1.1.

The parameters β and TR of the decision model have been set so as to avoid preemption of events/processing tasks whose residual time is ten times the IPI latency. Finally, the parameter α used in the EMA computation has been set to 0.2.

4.2 Benchmark Applications and Results

In order to evaluate the effectiveness of our solution, we have performed an experimental evaluation that relies on two different simulation models: PHOLD [9] and Personal Communication System (PCS). The first one has been exploited to estimate potential overheads of our approach under different configurations in terms of event granularity. Conversely, PCS allows us to provide an example of the benefits given by reducing the amount of doomed to

[5]Available at https://github.com/HPDCS/ipi4user



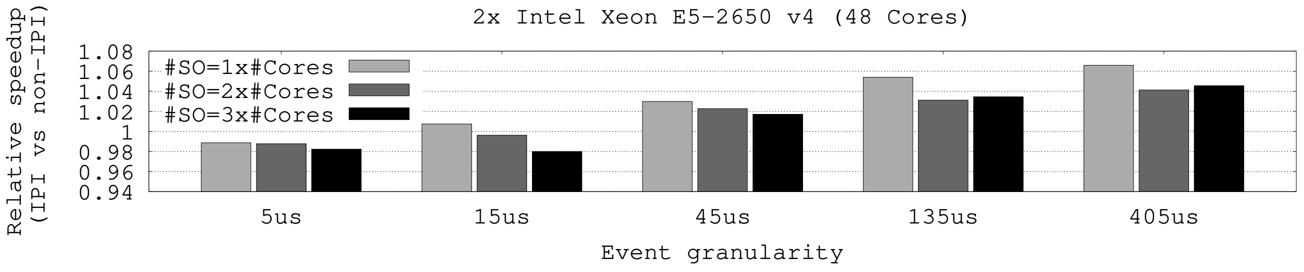


Figure 4: Results with PHOLD - each bar represents the speed up/slowdown obtained by our solution while running on top of the two different hardware platforms.

be rolled-back work in a real-world simulation model. Both models, whose detailed description and results are given in the following two sections, have been configured with a number of simulation objects (SOs) equal to 1, 2 and 3 times the number of CPU cores. The reduced ratio between the number of simulation objects and the number of cores, as well as the absence of lookahead, make the execution characterized by a high degree of simulation-object execution parallelism, which is considered a challenging scenario for speculative simulation.

4.2.1 *PHOLD.*

The first test-bed application is the classical PHOLD benchmark [9]. The execution of an event leads to updating the state of the target simulation object, which keeps track of statistics related to simulation advancement such as the number of processed events and average simulation time advancement observed by simulation objects. It also leads to executing a classical CPU busy-loop for the emulation of a given event granularity. There are two types of events: i) regular events, whose processing generates new events of any type; ii) diffusion events that do not generate new events when being processed. The number of diffusion events generated by regular ones (denoted as Fan-Out) is set to 1 in our evaluation. This event pattern leads to scenarios where the average number of events in the event pool is stable, but there are punctual fluctuations. The timestamp increments are drawn from an exponential distribution with mean set to one simulation-time unit. Finally, the busy loop proper of PHOLD event processing generates a different event granularity in different tests, namely $5\mu s$, $15\mu s$, $45\mu s$, $135\mu s$ and $405\mu s$, in order to emulate low to high granularity events proper of a large variety of discrete event models.

The results of this benchmark are presented in Figure 4, which reports the speedup/slowdown introduced by our solution. This has been computed as the ratio between the average throughputs for different thread counts. For platform A, the plot shows that our support introduces a sensible overhead (5%) for very fine-grain events ($5\mu s$), even though the occurrence of causality violations is very infrequent—the percentage of straggler events is below 1% in all configurations. Such overhead is even smaller (up to 2%) when running on a different and more recent architecture, as for the case of platform B. This result shows that our approach is non-intrusive, while being capable of providing speed up (i.e., 4% and 6% for platforms A and B, respectively) in very adverse scenarios—namely when rollbacks involve larger-grain events.

4.2.2 *PCS.*

PCS models a mobile network adhering to GSM technology. Each simulation object models the evolution of an individual hexagonal cell. Each cell can handle a number N of channels, which are modeled via power regulation and interference/fading phenomena, according to the results in [15]. The records associated with channels are dynamically allocated/released upon start/end of calls. Upon call setup, power regulation is performed, which involves scanning the aforementioned list of records to compute the minimum transmission power allowing the current call setup to achieve the threshold-level signal-to-interference ratio (SIR). Each record is released when the corresponding call ends or is handed off towards an adjacent cell. In the latter case, a similar call-setup procedure is executed at the destination cell. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a model defining meteorological conditions (and their variations). The set of configurable parameters entails:

- τ_A, which is the inter-arrival time of subsequent calls to any target cell;
- τ_D, which expresses the expected call duration;

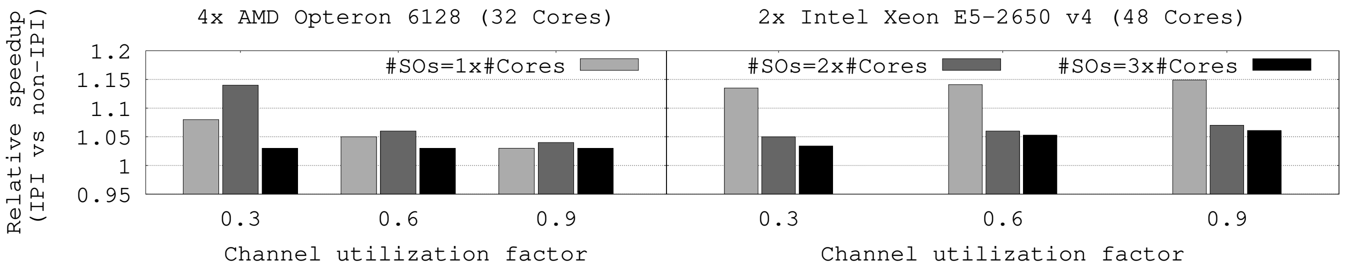


Figure 5: Results with PCS.

- τ_H, which expresses the residual residence time of a mobile device into the current cell.

These parameters affect the channel utilization factor, expressed as $\rho = \frac{\tau_D}{\tau_A \cdot N}$. The value ρ impacts the granularity of the events, since the more the busy channels, the more power-management records are allocated and consequently scanned/updated while processing events. At the same time, higher values of the channel utilization factor lead to higher memory requirements for the state image of individual simulation objects. Also, CPU and memory demands are bounded depending on the total number N of per-cell managed channels. In fact, when a call-setup operation is requested due to a call arrival (or a hand off arrival), if all the channels are already busy, then the call is dropped, mimicking the real-world scenario where communication is interrupted whenever the base station has no available resources to support the communication.

For this model, we have studied a configuration resembling high mobility of the devices involved in communication activities, like during morning hours around a commercial or business area, with many people moving towards their office or work place. Hence, we set the parameters to provide a non-minimal likelihood that an ongoing call is handed off between cells, in particular $\tau_D = 300s$ and $\tau_H = 120s$. N and τ_A have been set in order to evaluate scenarios where the channel utilization factor is equal to 0.3, 0.6 and 0.9.

Figure 5 shows the results achieved while running on top of the two hardware platforms. Coherently with what observed with the PHOLD model, the benefits from the IPI-based mechanism are more evident when running with lower counts of simulation objects because of the higher degree of actual simulation-object execution parallelism, leading to more pronounced speculation. For instance, the maximum performance is in most of the cases achieved when running with a number of simulation objects tied to the number of cores, giving at least 1.05x and 1.12x speedup in platforms A and B respectively—and providing a maximum speedup of 1.15x with platform B.

Even though the event granularity is enough large compared to the IPI latency (it varies from to $87\mu s$ to $195\mu s$ and from $45\mu s$ to $100\mu s$ in platform A and B respectively, depending on the channel utilization factor), the benefit introduced by our solution is strongly related (and increases) with the probability of rollbacks. As for this aspect, we report in Figure 6 data related to the frequency of causality violations for the executions on platform B. By the curves we see that (as expected) the frequency of causality violations decreases from about 12/13% to 3/5% when reducing the actual level

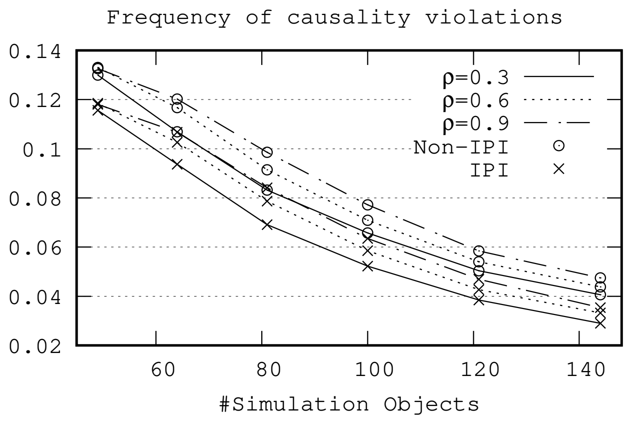


Figure 6: Results with PCS - frequency of causality violations.

of execution parallelism (namely, when increasing the number of simulation objects). Also, we see a slightly reduced frequency of casuality violations when relying on the IPI approach, indicating that this approach can early prevent the spreading of inconsistent computation. In fact, early interrupting the execution of an inconsistent event e avoids injecting in the system additional inconsistent events that would otherwise have been generated by the processing of e.

Still for runs on platform B, we show in Figure 7 the frequency of early interruptions of inconsistent events achieved with the IPI mechanism. By the results we see how this frequency ranges from about 40% to a maximum of 55%, depending on the different application settings. This confirms the fact that a good percentage of the inconsistent events can be actually early aborted, saving waste of resources.

Finally, in Figure 8 we offer a different view on the final performance data for executions of the PCS model on top of platform B. In particular, we report the throughput curves for the two scenarios—IPI vs non-IPI—including the confidence interval calculated on the basis of the different outcomes achieved over the 10 runs per data-point we carried out. The data show that the advantages by the IPI approach are statistically consistent across all the simulation object counts.

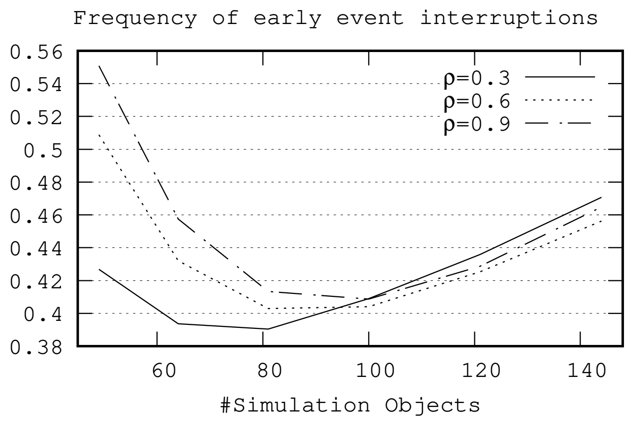


Figure 7: Results with PCS - frequency of early event interruptions.

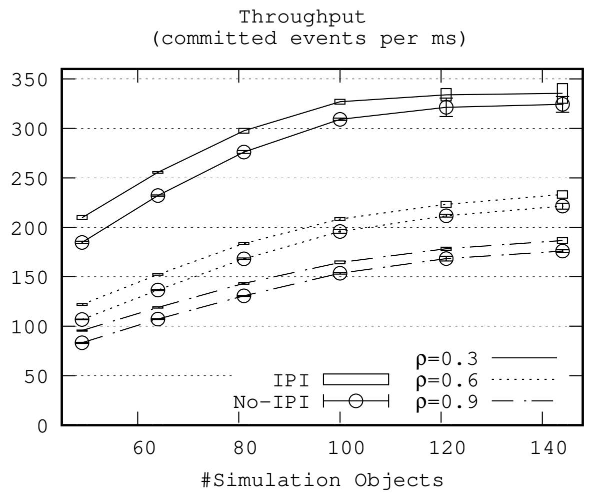


Figure 8: Results with PCS - throughput curves.

5 CONCLUSIONS

Inter-Processor-Interrupts (IPIs) are a fundamental technology in modern software/hardware systems. They allow a CPU-core to notify other cores about important tasks to be promptly executed. IPIs are widely exploited in Operating System technology, as an example to correctly drive the execution of multi-threaded applications, where something occurring on a given CPU-core because of the activities of the hosted thread needs to be reflected on the state of the hardware or shared data structures seen by another thread possibly running on another CPU-core. However, to the best of our knowledge, IPIs have not yet been exposed to the application programmers, and have not been exploited for optimizing specific application scenarios. In this article we have proposed a solution where IPIs are used to drive the evolution, in virtual time, of simulation objects running within a speculative parallel discrete event simulation environment hosted by a multi-core machine. In more detail, our solution enables the close-to-zero delay notification to a CPU-core of the inconsistency of the speculative work it is carrying out. This enables the on-demand early squash of incorrect portions of the computation, with benefits on CPU-cycles savings and (indirectly) energy savings to run the speculative simulation. We have embedded our solution within the Ultimate-Share-Everything (USE) open source simulation platform, and report in this article data for an assessment of the proposed solution with both synthetic and real-world simulation models. Our proposal is in the direction of introducing an innovative and unconventional system-level support offering runtime capabilities to be exploited by last generation parallel simulation systems, oriented to high performance and to the exploitation of scaled-up hardware parallelism in shared-memory platforms.

REFERENCES

[1] Christopher D Carothers and Richard M Fujimoto. 2000. Efficient execution of Time Warp programs on heterogeneous, NOW platforms. *IEEE Transactions on Parallel and Distributed Systems* 11, 3 (2000), 299–317.

[2] Christopher D Carothers, Kalyan S Perumalla, and Richard M Fujimoto. 1999. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation* 9, 3 (1999), 224–253.

[3] Malolan Chetlur, Nael B. Abu-Ghazaleh, R Radhakrishnan, and Philip A. Wilsey. 1998. Optimizing Communication in Time-warp Simulators. In *Proceedings of the Twelfth Workshop on Parallel and Distributed Simulation (PADS '98)*. IEEE Computer Society, Washington, DC, USA, 64–71. https://doi.org/10.1145/278008.278017

[4] Myongsu Choe and Carl Tropper. 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.

[5] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2017. Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. *ACM Trans. Model. Comput. Simul.* 27, 2, Article Article 11 (May 2017), 26 pages. https://doi.org/10.1145/3077583

[6] Samir R Das and Richard M Fujimoto. 1997. Adaptive Memory Management and Optimism Control in Time Warp. *ACM Transactions on Modeling and Computer Simulation* 7, 2 (1997), 239–271. https://doi.org/10.1145/249204.249207

[7] Phillip M. Dickens, David M. Nicol, Paul F. Reynolds, and J. M. Duva. 1996. Analysis of bounded time warp and comparison with YAWNS. *ACM Transactions on Modeling and Computer Simulation* (1996). https://doi.org/10.1145/240896.240913

[8] Keir Fraser. 2003. *Practical lock freedom*. Ph.D. Dissertation. Cambridge University Computer Laboratory.

[9] Richard M Fujimoto. 1990. Performance of Time Warp Under Synthetic Workloads. In *Proceedings of the Multiconference on Distributed Simulation*. Society for Computer Simulation, 23–28.

[10] Richard M. Fujimoto and Maria Hybinette. 1997. Computing Global Virtual Time in Shared-Memory Multiprocessors. *ACM Transactions on Modeling and Computer Simulation* 7, 4 (1997), 425–446. https://doi.org/10.1145/268403.268404

[11] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 161–176. https://doi.org/10.1007/978-3-319-62105-0_11

[12] Mauro Ianni, Romolo Marotta, Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2018. The Ultimate Share-Everything PDES System *(SIGSIM-PADS '18)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/3200921.3200931

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

[14] David R. Jefferson, Brian Beckman, Frederick Wieland, Leo Blume, Mike Di Loreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, L. Van Warren, John J. Wedel, Herb Younger, and Steve Bellenot. 1987. Distributed Simulation and the Time Warp Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987*, Les Belady (Ed.). ACM, 77–93.

[15] Sunil Kandukuri and Stephen Boyd. 2002. Optimal Power Control in Interference-Limited Fading Wireless Channels with Outage-Probability Specifications. *IEEE Transactions on Wireless Communications* 1, 1 (2002), 46–55.

[16] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. https://doi.org/

10.1109/SP.2019.00002 arXiv:1801.01203

[17] Yi-Bing Lin and Edward D Lazowska. 1991. Processor Scheduling for Time Warp Parallel Simulation. In *Advances in Parallel and Distributed Simulation*. IEEE Computer Society, 11–14.

[18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*.

[19] Vijay K. Madisetti, Jean C. Walrand, and David G. Messerschmitt. 1988. Wolf: a rollback algorithm for optimistic distributed simulation systems. In *Proceedings of the 20th conference on Winter simulation, WSC 1988, San Diego, California, USA, December 12-14, 1988*, Michael A. Abrams, Peter L. Haigh, and John Craig Comfort (Eds.). ACM, 296–305.

[20] Maged M Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[21] D.M. Nicol and Jason Liu. 2002. Composite synchronization in parallel discrete-event simulation. *IEEE Transactions on Parallel and Distributed Systems* 13, 5 (may 2002), 433–446. https://doi.org/10.1109/TPDS.2002.1003854

[22] Alessandro Pellegrini and Francesco Quaglia. 2017. A Fine-Grain Time-Sharing Time Warp System. *ACM Transactions on Modeling and Computer Simulation* 27, 2 (may 2017), 1–25. https://doi.org/10.1145/3013528

[23] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2015. Autonomic State Management for Optimistic Simulation Platforms. *IEEE Transactions on Parallel and Distributed Systems* 26, 6 (jun 2015), 1560–1569. https://doi.org/10.1109/TPDS.2014.2323967

[24] Bruno R Preiss, Wayne M Loucks, and D MacIntyre. 1994. Effects of the Checkpoint Interval on Time and Space in Time Warp. *ACM Transactions on Modeling and Computer Simulation* 4, 3 (1994), 223–253.

[25] Francesco Quaglia and Vittorio Cortellessa. 2002. On the processor scheduling problem in time warp synchronization. *ACM Transactions on Modeling and Computer Simulation* 12, 3 (2002).

[26] S Srinivasan and P F Reynolds Jr. 1998. Elastic Time. *ACM Transactions on Modeling and Computer Simulation* 8, 2 (apr 1998), 103–139.

[27] Brian Paul Swenson and George F Riley. 2012. A New Approach to Zero-Copy Message Passing with Reversible Memory Allocation in Multi-core Architectures.. In *PADS*. 44–52.

[28] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2012. Load sharing for optimistic parallel simulations on multi core machines. *ACM SIGMETRICS Performance Evaluation Review* 40, 3 (jan 2012), 2–11. https://doi.org/10.1145/2425248.2425250