*Proceedings of the 2019 Winter Simulation Conference*
*N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.*

# HARDWARE-ASSISTED INCREMENTAL CHECKPOINTING IN SPECULATIVE PARALLEL DISCRETE EVENT SIMULATION

Stefano Carnà
Serena Ferracci
Emanuele De Santis
Alessandro Pellegrini

DIAG
Sapienza, University of Rome
Lockless S.r.l.
Rome 00185, ITALY

Francesco Quaglia

DICII
University of Rome Tor Vergata
Lockless S.r.l.
Rome, 00133, ITALY

## ABSTRACT

Nowadays hardware platforms offer a plethora of innovative facities for profiling the execution of programs. Most of them have been exploited as tools for program characterization, thus being used as kind of program-external observers. In this article we take the opposite perspective where hardware profiling facilities are exploited to execute core functional tasks for the correct and efficient execution of speculative Parallel Discrete Event Simulation (PDES) applications. In more detail we exploit them—specifically, the ones offered by Intel x86-64 processors—to build a hardware-supported incremental checkpointing solution that enables the reduction of the event-execution cost in speculative PDES compared to the software-based counterpart. We integrated our solution in the open source ROOT-Sim runtime environment, thus making it available for exploitation.

## 1 INTRODUCTION

Speculative Parallel Discrete event Simulation (PDES) is known to be a core method for deliverying high performance of model execution (Jefferson 1985), and for enabling the full exploitation of the available computing resources in both distributed and shared memory settings (Barnes et al. 2013; Ianni et al. 2018). At the same time, a core aspect to be considered when building speculative PDES platforms is the ability to reconstruct past simulation states whenever a causality violation—caused by wrong speculation paths—needs to be undone.

Several literature proposals exist, envisaging different state-reconstruction methodologies, which can be roughly classified as checkpoint-based (Preiss et al. 1994) or reverse computing-based (Carothers et al. 1999)—although in Cingolani et al. (2017) a solution to mix the two strategies has been presented. In any case, the actual implementations of the state-reconstruction support mostly rely on pure software-based approaches, and do not exploit hardware-level operations that are nowadays commonly available in modern processors.

In this article we explore the alternative approach where the support for state reconstruction is devised as a hardware-assisted facility. Particularly, we focus on the checkpoint-based methodology, and present an architectural design of the speculative PDES engine where hardware-level program profiling capabilities offered by modern Intel processors are exploited to detect (with no software intervention) the state updates that occur while processing an event at the simulation object. This hardware-based detection is exploited to identify the portions of the object's state that have been updated along a sequence of events, which need to be logged to build a new incremental checkpoint. In this work, we explicitly target the x86-64 architecture

and the Linux operating system, as a test case for the viability of our hardware-assisted checkpointing support.

One core aspect in our proposal is that the detection of memory updates occurs at the exact granularity of the machine-instruction that performs the update. This provides a big advantage over memory-update detection actuated with other more traditional hardware-level mechanisms, such as the paging-firmware commonly exploited at the level of the operating system. In fact, the latter is known to work with page-based granularity—corresponding to 4KB in standard configurations on x86-64 processors—which can be clearly proven suboptimal for finer grain updates, possibly scattered across multiple operating-system pages.

We also note that our solution retains application transparency, since the activation of the hardware-level facility that enables memory update detection does not require any particular action by the user-defined event handler that implements the simulation logic used to process the events at the simulation objects. In fact, all the job of coordinating the hardware-level profiler and the execution of the event handler is carried out by the PDES runtime environment, which manages the activation of the application-level event handlers.

Clearly, detecting memory updates with no software intervention determines a drastic reduction of the cost of incremental checkpointing, thush making speculative PDES prone to delivering ever increasing speedup. Overall, our proposal is along the path of using hardware-level facilities as accelerators in the contexts of speculative PDES. However, this is done in an unconventional manner since we do not accelerate the execution of the event-handler logic—as it occurs when porting this logic to GPGPU architectures (Lysenko and D'Souza 2008; Liu and Andelfinger 2017). Rather, we avoid the inclusion of additional machine instructions that would otherwise be needed to intercept the memory updates natively coded within the event handler by the application programmer. In other words, we virtualize the presence of these instructions replacing them via "accelerated" harware-level tasks.

Our solution has been integrated in the open source ROOT-Sim speculative PDES runtime environment (Pellegrini and Quaglia 2014), and is available for download—all the sources and models used in this paper are available in the ROOT-Sim official online repository. In this paper we also provide experimental evidence of the effectiveness of our proposal when employed to support the execution of a synthetic model derived from the classical PHOLD.

The remainder of this article is structured as follows. In Section 2 we discuss related work. The off-the-shelf hardware support which can be used to implement our hardware-assisted checkpointing strategy is introduced in Section 3. The innovative architecture offering hardware-assisted incremental checkpointing capabilities in PDES is described in Section 4. Experimental data for the assessment of our proposal are provided in Section 5.

## 2 RELATED WORK

How to enable state reconstruction in speculative PDES in an efficient manner is a long-lived research topic. Literature proposals are very disparate and tackle a wide spectrum of aspects. The proposals in Preiss et al. (1994), Rönngren and Ayani (1994), Fleischmann and Wilsey (1995), Soliman and Elmaghraby (1998), Quaglia (2001) are agnostic of the way a single checkpoint is built; rather, they only need to know what is the (average) latency for taking the checkpoint in order to determine how frequently (or at what points along simulation time) checkpoints should be taken in order to optimize the tradeoff between the checkpointing cost and the state reconstruction cost—we recall that an un-checkpointed state needs to be recontructed by reloading the latest checkpoint preceeding that state and then reprocessing intermediate events. Essentially, these proposals provide performance models (sometimes used as run-time decision models) which do not directly address the issue of reducing the cost of each single checkpoint operation. Hence, they can be considered as orthogonal to what we propose in this article.

How to reduce the cost of the individual checkpoint operation has been tackled by other studies. Quaglia and Santoro (2003) present a hardware-assisted solution where programmable DMA engines are used to implement data-copy operations which allow to fill checkpoint buffers with the current content of

the simulation object state—hence offloading the checkpoint operation from the CPU. This solution does not offer the support for incremental checkpointing, since the DMA based data copy operation always stores the entire simulation object state, a limitation that appears to be relevant given the memory-wall phenomenon in modern processors. Also, it can be used only under the constraint that the simulation object state (which represents the source of the DMA operation) is stored in a contiguous memory buffer. Our porposal removes both these limtations since we enable incremental checkpointing and we support simulation objects' states that are scattered in memory.

Still by the side of hardware-assisted solutions, Santoro and Quaglia (2012) present an architecture where incremental checkpointing in the context of HLA-based simulations is achieved transparently via operating-system services. However, the granularity according to which memory accesses are tracked to determine what portion of the state to log within the incremental checkpoint corresponds to an entire operating-system page. We avoid this limitation since our hardware-assisted mechansm for tracking memory updates operates at the memory-granularity of each single memory-write machine instruction.

Fujimoto et al. (1992) have proposed the "rollback-chip", which is a specialized hardware component used to store state variables according to a multiversion scheme. This enables keeping within this hardware component checkpointed versions of the state of the simulation object. State restoration is achieved by instructing the rollback chip to realign the live state to the selected checkpoint. Differently from this proposal we do not rely on specialized hardware. Instead, we exploit conventional faclities offered by Intel CPUs, which makes our proposal applicable in a wider spectrum of contexts.

As for software based implementations of incremental checkpointing, optimized approaches rely on (semi-)transparent instrumentation of the event-handler code to inject additional instructions used to identify state updates (West and Panesar 1996; Rönngren et al. 1996; Pellegrini et al. 2015). These approaches have been shown to work well especially with read intensive workloads, where the fraction of instructions that perform memory updates is relatively small. Our objective is the one of avoiding at all the instrumenting instructions and migrating the task of tracking state updates directly to the hardware. This will make incremental checkpointing viable even in scenarios with more write-intensive workloads, for which the cost of tracking memory updates via additional software instructions could not pay off—with respect to taking non-incremental checkpoints of the whole state of the simulation object.

Our work is (less closely) related to studies targeted at the exploitation of specific hardware capabilities (or accelerators)—such as GPGPUs or FPGAs—for improving the execution speed of PDES platforms (Liu and Andelfinger 2017; Yang et al. 2018; Rahman et al. 2019). Compared to these proposal, our objective is the one of accelerating the execution of specific tasks carried out by the PDES platform, while still keeping the application executing on a conventional CPU architecture.

Finally, our proposal is fully orthogonal to the solutions based on reverse computing, such as the proposal by Schordan et al. (2018) or the one by Cingolani et al. (2017). These proposals still rely on the usage of infrequent checkpoints to optimize the delivered performance. In fact, our hardware-assisted checkpointing architecture can be used as a black-box checkpointing support in such combined techniques.

## 3 DETAILS ON THE EXPLOITED HARDWARE FACILITIES

Most modern processors are provided with hardware-based profiling features granted by specialized performance monitoring units (PMUs). The most common implementation of such units is represented by Performance Monitor Counters (PMCs), available since old CPU architectures such as the Intel Pentium or the AMD Athlon, which have been largely improved over time. This improvement has been driven by the possibility to profile the effects of software execution in a more precise way, but has often created incompatibilities at the software level with respect to previous families of firmware interfaces.

In the context of Intel CPUs, PMC operations are based on the concept of *hardware events*. Examples of hardware events are the occurrence of a write operation in memory, a cache miss or the fact that a conditional branch in the execution flow of the program has been taken. The events that a processor is aware of (and that can be therefore triggered) are highly coupled with the actual processor architectural family.

This is due to the fact that the generation of a hardware event is physically triggered by data paths or control signals implemented in the actual control unit of the CPU, which is often subject to partial or complete re-implementation across different families of processing units. In the literature on hardware profiling, this extremely low-level information is often referred to as *micro-architectural events*. The benefit of relying on micro-architectural events is that they can be extremely optimized, and they can work at the speed of the CPU. If we compare this capability with the cost that it is often incurred into when implementing incremental checkpointing supports via pure software approaches—see, for example, the works by West and Panesar (1996) or by Pellegrini et al. (2009)—it is evident that this speed is desirable. Fortunately enough, the complexity of the very low-level micro-architectural events is sometimes masked by CPU vendors, which try to select a set of events (called *architectural events*) which are deemed common to different architectural models. Relying on architectural events increases the level of portability of solutions based on these supports.

Modern CPUs support hundreds of events, but only few PMCs are available per processor (modern off-the-shelf CPUs commonly offer up to 8). Each PMC can be configured to track exactly one architectural event. The software interface to control a PMC is implemented by relying on a couple of *model-specific registers* (MSRs):

1. a *selector* (or *control*) register which is used to specify the PMC operating mode, and the architectural event to be observed;
2. a *counter* register, which is incremented every time the associated architectural event is triggered—of course, the counter can overflow.

The *control* register can be used to activate two different operating modes. When the PMC is configured to work in *counting mode*, the *counter* register simply accumulates the number of target architectural events observed while any program is running on the CPU. On the other hand, when the PMC is working in *sampling mode*, the system generates a hardware interrupt as soon as the *counter* register overflows. The operating system handler associated with this interrupt can be programmed to take a CPU snapshot so as to understand what is currently going on in the profiled application.

Nevertheless, when the interrupt service routine is activated, the program counter stored in the interrupt frame on the stack might not be associated with the actual instruction whose execution triggered the overflow of the counter, possibly making the whole approach useless, if extremely precise information is desired for profiling. This problem is largely exacerbated in modern super-scalar architectures which rely on out-of-order execution engines to speedup the execution of applications. Here, multiple operations are executed concurrently—as an example, the AMD Fam 10h processor can have up to 72 in-flight operations at any time. Therefore, due to the high number of concurrent pipelines and the different order associated with micro-operation execution and instruction retirement, the actual sampling notification may be late with respect to its generation point. This delay is called *skid*.

The accuracy of PMCs has been extensively studied by Korn et al. (2002), Weaver and McKee (2008), Zaparanuks et al. (2009), and we refer the reader to their work for a more comprehensive discussion. Anyhow, skid is a significant impairment to the effective usage of PMCs, as it might produce low-precision results. This is the reason behind additional capabilities which have been embedded within PMUs. In particular, more modern Intel CPUs extended the PMC capabilities with the *Precise Event-Based Sampling* (PEBS) support. It introduces the notion of *precise events*, beyond the aforementioned non-precise events which can be observed by relying on PMCs.

PEBS cannot be applied to all the events supported by the PMU, yet it provides a new hardware-based mechanisms that automatically saves the processor context when the counter overflows. This solution, called *PEBS assist*, is implemented at the firmware level, and it avoids any code interruption to gather extra processor information related to the event itself—no hardware interrupt is required to save the CPU context, which can be therefore inspected at a later time. PEBS is still based on the usage of the standard counters, so they can work in a combined manner. In particular, a PMU can still be configured to work

in sampling mode. In this way, when the counter reaches the configured threshold, a hardware interrupt is fired.

When gathering the event-related data, the information is packed into a structure called *PEBS record* which represents the base element of the *PEBS buffer*. This buffer is located in the *Debug store* (DS) save area, whose size can be defined at setup time by writing into the DS model-specific register. Every time an architectural event monitored using the PEBS support is triggered, a data sample is produced to snapshot the whole CPU state. In the PEBS buffer, the next record to be filled with a newly-generated sample is identified thanks to the *PEBS index*. When the PEBS index reaches the *PEBS threshold*, which can be configured at setup time, a hardware interrupt informs the operating system that the buffer is almost full and a read operation should be carried out as soon as possible, so as to avoid loosing samples. The overall organization and the relation between the data structures used to support PEBS execution are depicted in Figure 1.

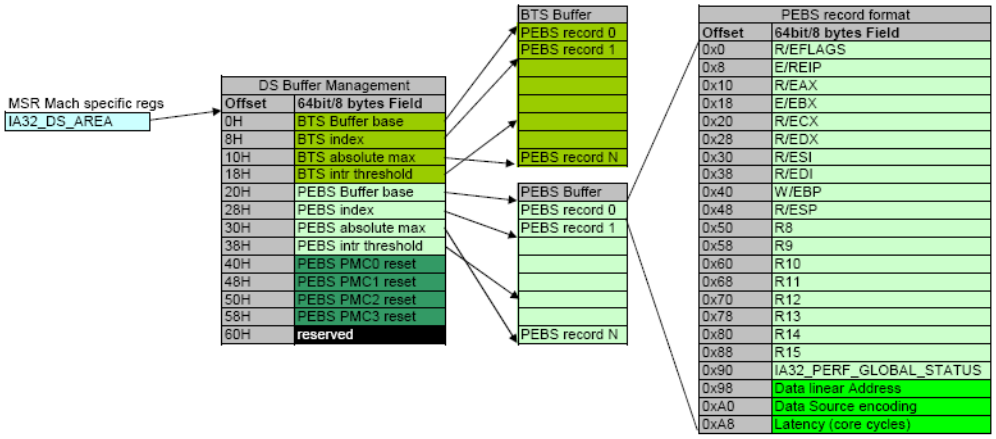


Figure 1: Organization and relation of the data structures used in the context of PEBS.

## 4 THE HARDWARE-ASSISTED INCREMENTAL CHECKPOINTING ARCHITECTURE

The hardware-assisted incremental checkpointing architecture which we have devised is composed of two different parts. On the one hand, we must be able (thanks to the discussed hardware support) to detect what are the portions of the simulation state which are touched in write mode during the execution of simulation events targeted at each specific simulation object involved in the simulation run. On the other hand, we must be able to exploit and organize this information in a way that allows to effectively handle state saving and restore operations, carried out to recover from causal violations due to the speculative nature of the simulation. In this section, we discuss separately both aspects.

### 4.1 Hardware-assisted Memory Write Tracing

In our hardware-assisted incremental checkpointing architecture we have relied on the PEBS support. In particular, we have implemented a Linux kernel module which exposes some services to the userspace PDES runtime environment to activate the tracing of memory write architectural events. The facilities offered by this kernel module are activated via proper `ioctl()` calls, towards a special device file which is created upon module load.

A first aspect to take into account when dealing with the tracing of memory write operations is that, despite the fact that the actual tracing is implemented via firmware facilities, a small overhead is anyhow introduced. This is related to the fact that the firmware has to pack PEBS records and write their content in main memory into the PEBS buffer, upon any memory access in write mode. This buffer is located in

RAM, therefore writing PEBS records consumes a certain amount of memory bandwidth. On more recent NUMA architectures, the penalty associated with writing into memory could be also exacerbated by the fact that a CPU core might be required to write to memory banks associated with remote NUMA nodes. This scenario could also lead to interference with the activities of other cores, as the remote memory access is carried out by relying on inter-cache controller messages, as in the case of CPU interconnect based on the Intel QuickPath.

It is therefore fundamental to limit the tracing operation only to the execution of event handlers associated with the simulation model. In this way, any memory write operation carried out by the PDES runtime environment to support its internal scheduling, checkpointing, or any other housekeeping operation will be executed with no active tracing operation, thus significantly reducing the overall overhead. Nevertheless, a general-purpose PDES runtime environment must account for simulation events which can have any execution time granularity. It is therefore fundamental to devise a solution which can quickly activate/deactivate the tracing facility based on PEBS, therefore we have decided not to rely on an `ioctl()` call for this very specific activity. Indeed, although the transition to kernel mode is quite fast on modern architectures (thanks to the introduction of the `syscall` assembly instruction), the activation of the proper `ioctl()` handler involves the execution of several kernel-level software trampolines (one associated with the system call dispatcher, and one associated with the `ioctl()` dispatcher, which is part of the Virtual-File-System layer). Also, more modern versions of the Linux kernel, implement protection mechanisms against the Meltdown security attack—in particular the Kernel-level Page Table Isolation mechanism, KPTI—which involve a modification of the page table upon any mode switch from userspace to kernel space, thus introducing an additional overhead associated with the flush of the TLB. Overall, paying all these costs just to enable hardware-assisted incremental checkpoinfing while executing a single simulation event that could last only a handful of microseconds could be too much performance unwise.

To limit the overhead to activate/deactivate PEBS-based tracing, we rely on a dedicated software trap. Upon module load, we modify the Interrupt Descriptor Table (IDT) of every CPU core by nesting an ad-hoc interrupt handler on interrupt vector `0xf0`. This is a vector which is not used by the Linux kernel for any of its internal interrupt service routines. The routine which we hook at this vector is extremely simple: it simply writes into a MSR register a value to activate/deactivate the PEBS-based tracing. The PDES runtime environment can therefore execute an ad-hoc software trap (using the `int` assembly instruction) which generates an execution path of a few assembly instructions in kernel mode. This execution flow is also KPTI-compliant, and therefore introduces a very negligible overhead.

When the PEBS-based tracing support is active, the firmware will write PEBS records into the PEBS buffer. This buffer is allocated in kernel space upon module load. We use a buffer of 16 contiguous (in physical memory) 4KB pages, taken directly from the Linux buddy system. If the PEBS index reaches the PEBS-buffer occupancy threshold, the firmware will activate a second ad-hoc handler (also hooked in the IDT upon module load) which simply moves the current buffer into a pool of buffers to be consumed by the PDES runtime environment in userspace, and allocates a new buffer to be used as the PEBS buffer.

Another aspect to take into account is the time-sharing nature of the Linux operating system. In particular, the PEBS-based hardware mechanism is completely unrelated to the scheduling activities of the kernel. It is therefore perfectly possible that, while the PDES runtime environment is executing a simulation event, the time quantum allocated to it expires. In this scenario, the scheduler might give the CPU to a different thread, completely unrelated to the PDES simulation engine, with the PEBS-based support still active. The firmware writes in the PEBS buffer virtual addresses, and it is therefore possible that many samples will be associated with addresses which are either unmapped in the PDES simulation process, or are associated with buffers which are valid, yet not related with the simulation model. To avoid this phenomenon, the kernel module attaches an ad-hoc routine at the end of the execution flow of the scheduler of the Linux kernel—this is done by relying on the `kprobes` facilities offered by Linux. In particular, every time that the scheduler determines that a new thread should be scheduled on some CPU core, we check the pid of that thread against all the pids of the PDES runtime environment—these pids are registered

at simulation startup via an ad-hoc `ioctl()` call. If the about-to-be-scheduled thread does not belong to the PDES runtime environment, we perform a write operation on a MSR to disable PEBS-based tracing. In the opposite case, we explicitly enable PEBS-based tracing. By relying on this approach, we limit a possible system-wide performance penalty, and we avoid the introduction of any noise in the PEBS buffer, with respect to memory write operations not associated with the PDES run.

The last aspect which we have dealt with is how to transfer the information from the PEBS buffer to the userspace application. As mentioned, the PEBS buffer is composed of physically-contiguous pages of memory, which can be obtained only at kernel level—contiguousness is a requirement for the firmware to be able to correctly generate PEBS records. Moreover, the content of PEBS records is not completely of interest for the PDES runtime environment, which is interested only in the information associated with simulation state memory write operations, rather than in a full CPU snapshot. We have therefore implemented an `ioctl()` call which allows to retrieve a set of tuples in the form $\langle$*base address*, *size*$\rangle$, where *base address* is the initial address of the memory area touched in write mode by the simulation model, and *size* is the amount of bytes touched by the operation. This `ioctl()` call is similar in spirit to the `readdir()` Linux system call: it returns from kernel space a stream of structures describing a set of the aforementioned tuples, which can be used to construct the metadata used to later build the incremental log. This call essentially consumes the pool of PEBS buffers which have been filled during the execution of one or more simulation events—once the data from a buffer in the pool is completely transferred to the userspace PDES runtime environment, the buffer is returned to the buddy system.

To summarize, the actions which the PDES runtime environment executes to lever the PEBS-based memory-write tracing are:

1. before scheduling any simulation event to any simulation object, the hardware-assisted tracing is activated (thanks to a fast dedicated software trap via the `int 0xf0` assembly instruction);
2. when the event is completely executed, the hardware-assisted tracing is disabled (again via the `int 0xf0` assembly instruction);
3. the information about what memory areas have been accessed in write mode by one or multiple events are queried from the module, via an `ioctl()` call.

This simple scheme allows to identify what is the portion of the simulation state which has been updated while processing events.

### 4.2 Managing Incremental Checkpoints

The information retrieved from the kernel module should be used by the PDES runtime environment to build and manage incremental checkpoints. To this end, we borrow the baseline approach from Pellegrini, Vitali, and Quaglia (2015). In this proposal, each simulation object's state is managed thanks to a userspace memory manager which serves dynamic memory allocations via `malloc()` calls. In particular, the PDES runtime environment relies on a set of bitmaps to describe the current state of each simulation object, in terms of buffer allocations and memory updates.

The PDES runtime environment uses the $\langle$*base address*, *size*$\rangle$ tuples retrieved via `ioctl()` to flag all the corresponding memory chunks in the dynamic memory allocator metadata as dirtied. This information is used to pack a checkpoint which only has the chunks that have been updated since the last checkpoint (and the associated metadata).

This checkpoint is then linked to the checkpoint queue, in event timestamp order, as in the traditional Time Warp proposal by Jefferson (1985). We note that this approach allows for great flexibility in the checkpointing scheme. Indeed, we are only required to update the metadata describing what portions of the simulation state has been updated after the execution of every event. The actual incremental checkpoint can be taken also after the execution of any number of simulation events. In this sense, our proposal is perfectly compatible with all previous literature results based on the usage of checkpointing intervals (rather

than checkpointing at each event). Indeed, it is possible to rely on sparse state saving (Lin and Lazowska 1990; Bellenot 1992), or on any form of adaptive state saving (Palaniswamy and Wilsey 1993; Rönngren and Ayani 1994; Fleischmann and Wilsey 1995; Skold and Rönngren 1996; Quaglia 1998; Quaglia 2001).

To reconstruct a previous simulation state upon a rollback operation, the chain of logs is backward traversed, starting from the first checkpoint appearing in the simulation time axis before the restoration point—this is again a classical means to support state restoration. From each incremental log that is found in the chain, the PDES runtime environment puts back in the live image of the simulation state all the chunks that are available and have not yet been restored in possible previous iterations. In this way, only the "newest" chunks are restored, and multiple memory write operation for the same chunk are not executed (this would be the case if the state was reconstructed by traversing the chain in reverse order). Also, all the metadata describing which chunks are currently in use are restored.

The iterative restore procedure stops when all the in-use chunks that have been dirtied are restored. Although in principles this could entail an indefinite number of iterative backward steps along the log chain, in practice the restore operation can be immediately finalized once we find a full log while backward re-traversing the log chain. In fact, all the in-use chunks that have not yet been restored are immediately available inside the full log for copy-back operations. Full logs can be explicitly interleaved to incremental ones, according to the original proposal by Pellegrini, Vitali, and Quaglia (2015).

## 5 EXPERIMENTAL RESULTS

We have implemented the hardware-assisted incremental checkpointing architecture described in Section 4 within the open source ROOT-Sim speculative PDES runtime environment (Pellegrini and Quaglia 2014). All the simulations have been run on an octo-core Intel i5-8250U CPU, equipped with 16 GB of RAM, running Linux 4.9.1.

To assess the effectiveness of our approach, we rely on a baseline configuration which is based on the more traditional pure software-based incremental state saving technique. In this baseline configuration, we have relied on a custom `gcc` plugin which, during the compilation of the simulation model, inserts in a completely transparent way ad-hoc function calls before every memory write instruction. In this way, the PDES runtime environment can be notified of the pending write operation, and it can flag any relevant metadata to mark a portion of memory as dirtied, since the last checkpoint. Both the hardware-based implementation and the software-based implementation rely on the incremental state saving mechanism proposed by Pellegrini, Vitali, and Quaglia (2015).

To study the performance of the hardware-assisted incremental checkpointing architecture, we have relied on a synthetic benchmark derived from the well known PHOLD benchmark presented by Fujimoto (1990), explicitly embedding parameterizable memory operations' patterns. In this benchmark, each simulation object executes fictitious events which only involve the advancement of the local simulation clock to the event timestamp. Every time that an event is executed, a new fictitious event is scheduled, destined to any simulation object, with a timestamp increment following some exponential distribution. Implementations of this benchmark have been already used in the literature to study the performance of incremental state saving approaches (West and Panesar 1996; Quaglia and Santoro 2003). The execution of an event includes a busy loop (which emulates a specific CPU delay for event processing, and hence a specific event granularity) and/or read/write mode access to a fictitious, memory contiguous state buffer of a given size $S$. Large values for $S$ would mimic applications with large memory requirements. On the other hand, the spanning of read/write operations across the state buffer determines the specific locality inside the object state, associated with the event execution.

We have configured the model to use 16 simulation objects, and we have run the model on 4 concurrent worker threads. Each simulation object has been associated with a state of 16 KB, and we have varied in the range $[3KB, 15KB]$ the average amount of memory touched in write mode during the execution of each simulation event at each simulation object. Due to the scattered nature of the buffers, a higher amount of total memory accessed in write mode increases the number of memory writes to be intercepted (both

in the software and the hardware-based case). However, we have adopted an approach where the update operations are based on memory-block writing machine instructions, such as `stos`, which gives rise to a configuration not unfavorable to the software based solution (since a single interception via software allows capturing both smaller and larger memory updates).

As a first consideration, the duration of checkpoint/restore operations is not directly affected by using hardware- or software-based tracing of the memory-write operations. In particular, when updating 15 KB of simulation state per event, in both configurations we have observed an average checkpoint creation time on the order of 0.5 $\mu s$, and a recovery time on the order of 4.5 $\mu s$. Therefore, the data that we compare in the remainder of this section exactly allows to capture the effects on memory tracing of the two different supports (hardware vs software).

In Figure 2 we report the average duration (in $\mu s$) of a simulation event when varying the size of the memory accesses. From the result, we can observe that the software-based solution (referred to as SW in the plots) is always outperformed by the hardware-assisted solution (HW in the plots)—although as discussed above it is not penalized by larger memory updates given the memory-block approach of writing machine instructions. This is an expected result, because (as we have discussed) the hardware-assisted solution relies on highly optimized firmware operations to trace a single memory access. On the other hand, for the same operation, the software-based solution requires activating a dedicated software function. This has also the drawback of requiring to save the content of several CPU registers for the execution of the model to continue correctly.

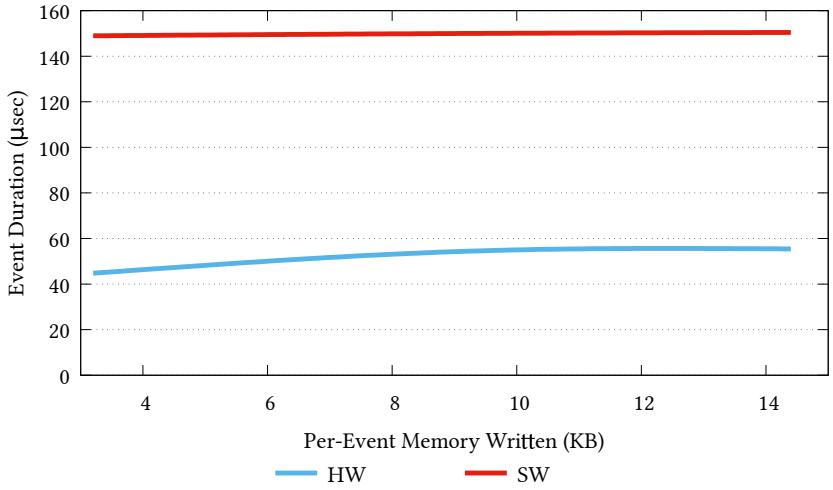


Figure 2: Average Event Duration.

This result is also related to the data reported in Figure 3. In particular, the reliance on a highly-optimized firmware increases the overall efficiency of the simulation—a reduced number of rollbacks is always observed when relying on the hardware-assisted solution. We believe that this phenomenon is related to a secondary effect on the usage of caches, ultimately related to the variance of the event execution times. Indeed, PEBS buffers are always contiguous in memory, and therefore accessing them to store PEBS samples will likely have a more circumscribed negative effect on the data used by the models and on the variance in the event execution time. Therefore, the reduction of variance of the duration of an event is also strengthened, an effect which is already known in the literature to have a possible positive effect on the rollback probability.

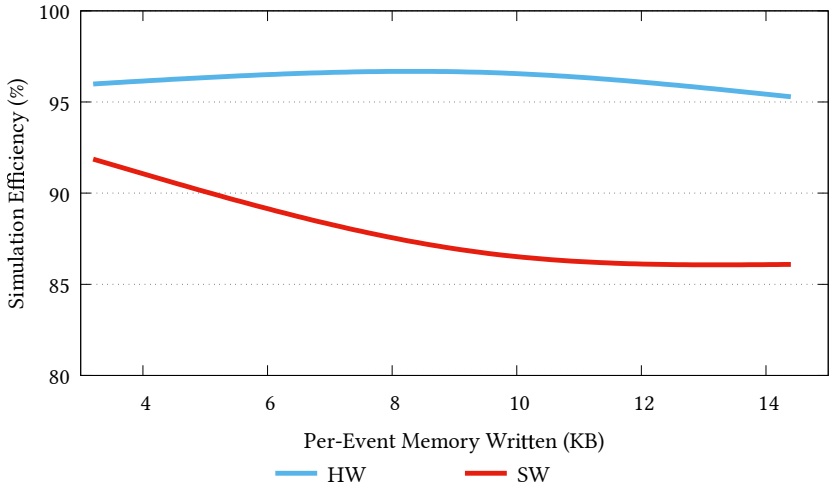


Figure 3: Simulation Efficiency.

## 6 CONCLUSIONS AND FUTURE WORK

In this paper we have explored the effectiveness of hardware profiling facilities used in the context of PDES runtime environments to intercept memory-write operations performed by simulation models, thus enabling the possibility to support incremental state saving. We have compared experimentally our implementation relying on the Intel PEBS support with more traditional software instrumentation based approaches. Our experimental assessment shows that the possibility of relying on hardware-assisted checkpointing is viable, as it reduces the intrusiveness of the incremental state saving support, traditionally associated with software instrumentation.

As future work, we plan to study the reliability and the effectiveness of the hardware support under differentiated workloads, also by using multiple real-world simulation models.

## REFERENCES

Barnes, P. D., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. 2013. "Warp Speed: Executing Time Warp on 1,966,080 Cores". In *Proceedings of the 2013 ACM SIGSIM conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS, 327–336. New York, NY: Association for Computing Machinery.

Bellenot, S. 1992. "State Skipping Performance with the Time Warp Operating System.". In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, PADS, 53–64.

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. 1999. "Efficient Optimistic Parallel Simulations Using Reverse Computation". *ACM Transactions on Modeling and Computer Simulation* 9(3):224–253.

Cingolani, D., A. Pellegrini, and F. Quaglia. 2017. "Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES". *ACM Transactions on Modeling and Computer Simulation* 27(2):1–26.

Fleischmann, J., and P. A. Wilsey. 1995. "Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators". In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, 50–58. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Fujimoto, R. M. 1990. "Parallel Discrete Event Simulation". *Communications of the ACM* 33(10):30–53.

Fujimoto, R. M., J. J. Tsai, and G. Gopalakrishnan. 1992. "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp". *IEEE Transactions on Computers* 41(1):68–82.

Ianni, M., R. Marotta, D. Cingolani, A. Pellegrini, and F. Quaglia. 2018. "The Ultimate Share-Everything PDES System". In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, 73–84. New York, NY: Association for Computing Machinery.

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

Korn, W., P. Teller, and G. Castillo. 2002. "Just How Accurate are Performance Counters?". In *Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference*, 303–310. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Lin, Y.-B., and E. D. Lazowska. 1990. "Reducing the Saving Overhead for Time Warp Parallel Simulation". Technical Report. Department of Computer Science and Engineering, University of Washington, Seattle, Washington.

Liu, X., and P. Andelfinger. 2017. "Time Warp on the GPU: Design and Assessment". In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS, 109–120. New York, NY: Association for Computing Machinery. New York, NY, USA: ACM.

Lysenko, M., and R. M. D'Souza. 2008. "A Framework for Megascale Agent Based Model Simulations on the GPU". *Journal of Artificial Societies and Social Simulation* 11(4):10.

Palaniswamy, A. C., and P. A. Wilsey. 1993. "An Analytical Comparison of Periodic Checkpointing and Incremental State Saving". In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, PADS, 127–134. New York, NY: Association for Computing Machinery.

Pellegrini, A., and F. Quaglia. 2014. "The ROme OpTimistic Simulator: A Tutorial". In *Proceedings of the Euro-Par 2013: Parallel Processing Workshops*, edited by D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Constan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, PADABS, 501–512. LNCS, Berlin, Germany: Springer-Verlag.

Pellegrini, A., R. Vitali, and F. Quaglia. 2009. "Di-DyMeLoR: Logging Only Dirty Chunks for Efficient Management of Dynamic Memory Based Optimistic Simulation Objects". In *Proceedings - Workshop on Principles of Advanced and Distributed Simulation*, PADS, 45–53. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Pellegrini, A., R. Vitali, and F. Quaglia. 2015. "Autonomic State Management for Optimistic Simulation Platforms". *IEEE Transactions on Parallel and Distributed Systems* 26(6):1560–1569.

Preiss, B. R., W. 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):223–253.

Quaglia, F. 1998. "Event History Based Sparse State Saving in Time Warp". In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, PADS, 72–79. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Quaglia, F. 2001. "A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation". *IEEE Transactions on Parallel and Distributed Systems* 12(4):346–362.

Quaglia, F., and A. Santoro. 2003. "Non-Blocking Checkpointing for Optimistic Parallel Simulation: Description and an Implementation". *IEEE Transactions on Parallel and Distributed Systems* 14(6):593–610.

Rahman, S., N. B. Abu-Ghazaleh, and W. A. Najjar. 2019. "PDES-A: Accelerators for Parallel Discrete Event Simulation Implemented on FPGAs". *ACM Transactions on Modeling and Computer Simulation.* 29(2):12:1–12:25.

Rönngren, R., and R. Ayani. 1994. "Adaptive Checkpointing in Time Warp". In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, PADS, 110–117. San Diego, California: The Society for Computer Simulation, International.

Rönngren, R., M. Liljenstam, R. Ayani, and J. Montagnat. 1996. "Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation". In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, PADS, 70–77. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Santoro, A., and F. Quaglia. 2012. "Transparent Optimistic Synchronization in the High-Level Architecture via Time-Management Conversion". *ACM Transactions on Modeling and Computer Simulation* 22(4):21:1—-21:26.

Schordan, M., T. Oppelstrup, D. R. Jefferson, and P. D. B. Jr.. 2018. "Generation of Reversible C++ Code for Optimistic Parallel Discrete Event Simulation". *New Generation Computing* 36(3):257–280.

Skold, S., and R. Rönngren. 1996. "Event Sensitive State Saving in Time Warp Parallel Discrete Event Simulation". In *Proceedings of the 1996 Winter Simulation Conference*, edited by J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, 653–660. San Diego, California: The Society for Computer Simulation, International.

Soliman, H. M., and A. S. Elmaghraby. 1998. "An Analytical Model for Hybrid Checkpointing in Time Warp Distributed Simulation". *IEEE Transactions on Parallel and Distributed Systems* 9(10):947–951.

Weaver, V. M., and S. A. McKee. 2008. "Can Hardware Performance Counters be Trusted?". In *2008 IEEE International Symposium on Workload Characterization*, 141–150. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

West, D., and K. Panesar. 1996. "Automatic Incremental State Saving". In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, PADS, 78–85. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Yang, M., P. Andelfinger, W. Cai, and A. Knoll. 2018. "Evaluation of Conflict Resolution Methods for Agent-Based Simulations on the GPU". In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS, 129–132. New York, NY: Association for Computing Machinery.

Zaparanuks, D., M. Jovic, and M. Hauswirth. 2009. "Accuracy of Performance Counter Measurements". In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 23–32. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

## AUTHOR BIOGRAPHIES

**STEFANO CARNÀ** is currently enrolled in the PhD program in Computer Engineering at Sapienza, university of Rome. He has received the Master's Degree in Computer Engineering at Sapienza, University of Rome in 2017, and a Bachelor's Degree in Computer Engineering at University of Calabria. His main research interest are in transparent performance monitoring techniques using hardware-based support. His work is mainly focused at the level of operating-system kernels, with the ultimate goal of delivering comprehensive supports to study different performance aspects, with extremely reduced interference and performance penalties. His email address is carna@diag.uniroma1.it.

**SERENA FERRACCI** is a Master Student in Computer Engineering at Sapienza, University of Rome. Her research interest are focused on Cyber Security detection systems, levering hardware-based supports. Her email address is ferracci.1649134@studenti.uniroma1.it.

**EMANUELE DE SANTIS** is a Master Student in Computer Engineering at Sapienza, University of Rome. His research interest are focused on software instrumentation and compiler supports. His email address is desantis.1664777@studenti.uniroma1.it.

**ALESSANDRO PELLEGRINI** has received the PhD degree in Computer Engineering from Sapienza, University of Rome in 2014. His research interests are simulation on parallel and distributed architectures, compilers, operating systems, high-performance computing systems, and distributed and concurrent algorithms. On these topics he has more than 70 publications on books, journals and conference proceedings. In 2018 he has won the HiPEAC Technology Transfer Award, while in 2015 he has won the Best PhD Thesis Award from Sapienza, University of Rome. He has worked as a researcher in many national/international research institutes, such as ISSNOVA, CINI, CINFAI, IRIANC, and BSC. He has successfully contributed to the development of open-source applications currently used in several international research institutes. His email address is pellegrini@diag.uniroma1.it.

**FRANCESCO QUAGLIA** has received the Laurea degree (MS level) in Electronic Engineering in 1995 and the PhD degree in Computer Engineering in 1999 from Sapienza University of Rome. Since 2017 he works as a Full Professor at the School of Engineering of the University of Rome Tor Vergata. His research interests include distributed systems and protocols, middleware platforms, parallel/distributed and federated simulation systems, parallel computing, fault-tolerance, transactional systems and Web-based systems. In these areas, he has been an author of more than 180 technical articles, and gained five best paper awards. He has acted as coordinator within several EU and international/national projects. Currently he is an associate editor of ACM Transactions on Modeling and Computer Simulation. His email address is francesco.quaglia@uniroma2.it.