Consistent and Efficient Output-Streams Management in Optimistic Simulation Platforms

Francesco Antonacci Alessandro Pellegrini Francesco Quaglia
DIAG – Sapienza, University of Rome

ABSTRACT

Optimistic synchronization is considered an effective means for supporting Parallel Discrete Event Simulations. It relies on a speculative approach, where concurrent processes execute simulation events regardless of their safety, and consistency is ensured via proper rollback mechanisms, upon the a-posteriori detection of causal inconsistencies along the events' execution path. Interactions with the outside world (e.g. generation of output streams) are a well-known problem for rollback-based systems, since the outside world may have no notion of rollback. In this context, approaches for allowing the simulation modeler to generate consistent output rely on either the usage of ad-hoc APIs (which must be provided by the underlying simulation kernel) or temporary suspension of processing activities in order to wait for the final outcome (commit/rollback) associated with a speculatively-produced output. In this paper we present design indications and a reference implementation for an output streams' management subsystem which allows the simulation-model writer to rely on standard output-generation libraries (e.g. stdio) within code blocks associated with event processing. Further, the subsystem ensures that the produced output is consistent, namely associated with events that are eventually committed, and system-wide ordered along the simulation time axis. The above features jointly provide the illusion of a classical (simple to deal with) sequential programming model, which spares the developer from being aware that the simulation program is run concurrently and speculatively. We also show, via an experimental study, how the design/development optimizations we present lead to limited overhead, giving rise to the situation where the simulation run would have been carried out with near-to-zero or reduced output management cost. At the same time, the delay for materializing the output stream (making it available for any type of audit activity) is shown to be fairly limited and constant, especially for good mixtures of I/O-bound vs CPU-bound behaviors at the application level. Further, the whole output streams' management subsystem has been designed in order to provide scalability for I/O management on clusters.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSIM-PADS'13, May 19–22, 2013, Montréal, Québec, Canada.
Copyright 2013 ACM 978-1-4503-1920-1/13/05 ...$15.00.

Categories and Subject Descriptors

I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete Event, Parallel*; B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems)—*Parallel I/O*; D.2.8 [**Software Engineering**]: Metrics—*Performance Measures*

General Terms

Theory, Design, Experimentation, Performance

Keywords

PDES, Output Commit, Transparency, Consistency, Rollback, ROOT-Sim

1. INTRODUCTION

Parallel Discrete Event Simulation [7] is based on the partitioning of the simulation model into several (distinct) simulation objects, whose execution is undertaken by Logical Processes (LP), which are hosted (and scheduled) by one or more simulation kernel instances. Given that LPs are allowed to run concurrently, the PDES approach is natively prone to exploit the computing power offered by multi-core and/or clustered architectures in order to speedup the simulation run. The actual speed is determined by the synchronization scheme that is adopted in order to ensure causally consisted (e.g. timestamp ordered) execution of the events at each LP.

The optimistic synchronization protocol, as described in [9], has established itself as a successful support to the execution of efficient simulation runs. In particular, it entails speculative event processing schemes, coupled with rollback/recover mechanisms for squashing causally-inconsistent event processing, which are prone to maximal exploitation of the parallelism intrinsic to the simulation model. On the other hand, the optimistic protocol is well known to be difficult to deal with in terms of transparency to the modeler, given that its complexity (e.g. in terms of state restore operations and annihilation of inconsistent events' spreading across the LPs) may require the application layer to entail synchronization oriented code blocks, to be employed in synergy with the facilities offered by the underlying simulation kernel. On the other hand, masking synchronization at all to the modeler, while not sacrificing performance due to the avoidance of application vs kernel layer synergic programming schemes, would require very advanced methodologies/algorithms, and related implementations, which nowadays represent a major challenge for the diffusion of the optimistic PDES paradigm in the large. Some solutions along

this direction, which have been oriented to full (or partial) transparency for what concerns state recovery of the LPs and/or mutual consistency of LPs' states and shared data can be found in [3, 16, 20, 23]. They represent a step ahead along the path of bringing the power of the optimistic PDES paradigm into the hands of ordinary modelers (namely programmers).

In this paper, we cope with the aforementioned transparency vs performance issue for what concerns the production of output streams in optimistic simulation runs. Particularly, we focus on how to efficiently and transparently tackle the output commitment problem [5] in optimistic simulation, which is related to the possibility for a simulation model (executed optimistically) to interact with the outside world, e.g. by producing output on a screen, on a network device, on a printer, or on any other device which might be unable to undo the output materialization in case a rollback operation is triggered because of a-posteriori causality inconsistency detection (see Figure 1). This timely and consistent interaction can be significant for all those simulation scenarios where the evaluation of (possibly unstable) predicates should be done on a global scale, via audit on individual LPs' state trajectories. With our approach, we explicitly free the simulation-model developer from facing the notion of *event commitment*, and allow her to rely on output calls (to be invoked within the modeler-defined event handlers), which are properly managed by the simulation kernel. More precisely, the application-level developer can rely on standard libraries calls (e.g., on the `printf()` family functions) to produce output on some streams, and the burden of materializing the actual output is placed on a specifically-targeted runtime subsystem which, at the same time, provides:

(i) *consistency*, by finalizing the actual output operation only if it is associated with an event which gets committed, and by system-wide ordering the output messages along the simulation time axis (as if they were produced by a sequential run of the same model);

(ii) *efficiency*, by inducing only a small (negligible) overhead at the side of the engine running the simulation model; and

(iii) *timeliness*, by placing only a reduced delay from the generation to the materialization of the output on the specified stream, unless for applications constantly exhibiting I/O-bound profile along the whole run.

Traditionally, PDES simulation frameworks (see, e.g., [8, 14, 19]) have relied on ad-hoc APIs for delivering consistent output related to the (progressively advancing) state trajectory while executing the simulation model, in order to (incrementally) provide, e.g., model statistics. This forces the model writer to look more deeply into the internal details of simulation supports, making the development of the application more cumbersome and more constrained to the specific programming model supported by the framework. On the other hand, in case the modeler would rely on non-consistent output production via the exploration of standard I/O libraries, the output streams would need to be post-processed for cleaning them from messages produced by events that got eventually rolled-back, and for providing a unique globally timestamp-ordered trace, if needed.

Our proposal is based on the concept of an ad-hoc *daemon*, i.e. a separate user-space process with respect to the actual simulation framework. Once equipped with our I/O-management-stub, the simulation kernel instances within the framework can communicate with the daemon process via in-memory, fast-access, shared data structures. They are used to support an additional level of temporary buffering which is biased to reduce the cost for managing output streams (and their consistency) at the side of the framework (as opposed to the typical buffering rules/schemes used by standard I/O libraries). Particularly, via this additional buffering subsystem we achieve: (i) *reduced interaction time*, by developing a non-blocking algorithm for accessing shared memory segments by the simulation kernel instances and the output daemon (hence materializing the scenario where a near-to-zero-cost (transparently consistent) logical output device is accessible by the simulation framework), and (ii) *enhanced scalability* (even at a geographical scale), allowing to process the commitment of the output on a multi-level basis; particularly, the daemon exploits Global-Virtual-Time (GVT) computation already natively performed by the framework in order to finalize the production of the output streams, which avoids the need for I/O specific consensus protocols to be run. As an additional advantage, the output daemon can run as a *stream forwarder*, allowing the user to retrieve output on a separate (dedicated) machine with respect to the cluster of machines used for model execution.

The efficiency of our proposal is daemonstrated via an extensive experimental study carried out in the context of a wireless communication system model, hosted on top of the ROOT-Sim open source PDES optimistic simulation framework [8].

The remainder of this paper is structured as follows. In Section 2, related work is discussed. Section 3 deals with theoretical aspects of output management in optimistic simulation, presenting involved issues, proposing a general design approach, and depicting our architecture/implementation, also showing how it could be simply interconnected with any existing simulation engine. Experimental data, for assessing the effectiveness of our proposal, are reported in Section 4.

2. RELATED WORK

The issue of output commitment in rollback recovery systems has been thoroughly studied in literature. An excellent survey on the proposed approaches can be found in [5]. These solutions have been oriented to fault tolerance, where rollback occurrence is due to failures, potentially causing the loss of volatile state information causally related to the output message (which might lead to un-recognize the relation between the delivered output and the actual system state, after rollback). Overall, these schemes are aimed at supporting reproducibility of state trajectories (and related outputs) in order to keep the system state aligned with what is actually observed by the external world. Instead, in optimistic PDES systems we experience the problem of masking the outcome of specific state trajectories (those that are rolled back) towards the external world. Further, the approaches surveyed in [5] deal with the Lamport's causality model [13], where no predetermination of events' ordering is imposed by pre-computed event timestamps, as instead it occurs in PDES systems. On the basis of the above considerations, our work can be considered orthogonal to these solutions.

Still in the context of fault tolerance, the work in [11] presents an innovative protocol for supporting output commitment specifically aimed at promptly delivering output data, which is one of the target of our work. It places special attention on state-information copies on stable storage, to overcome the failure of processes and minimizing the time

for restarting the execution of the system in case of failure. Despite the goal differences (already discussed above), this work is based on a communication protocol which tries to minimize the time from the output generation and its materialization. Differently from this proposal, we do not explicitly rely on a special communication protocol for committing the output, rather we exploit the GVT computation protocol (endemically required by optimistic PDES systems) to get information about which portion of the generated output can be considered committed, avoiding any additional communication cost.

The work in [22] addresses the problem of external-world interactions in the context of Transactional Memories (TM). In particular, the work focuses on what a TM system (either software or hardware) should support to allow the execution of irrevocable actions, such as I/O and system calls, when the side effects cannot be rolled back. The proposed solution allows the programmer to mark a specific transaction as *irrevocable*, i.e. the interactions with the external world are immediately finalized, and the correctness is guaranteed by avoiding to abort such transactions. Whenever a transaction is marked as irrevocable, the runtime support checks whether another irrevocable transaction is currently being run. In the positive case, the new transaction's execution is delayed until the other irrevocable transaction commits. In the negative case, the transaction is immediately executed, and in case some conflict on accessed data is detected, the contention manager only aborts other transactions which are not marked as irrevocable. Differently from this proposal, we offer complete transparency with respect to the application-level programmer (i.e. non-rollbackable operations are not required to be marked), and furthermore we allow multiple interactions with the external world to run concurrently, providing ad-hoc facilities to correctly order the outcome of these operations, and to support rollback of no-longer-consistent operations by an efficient and optimized multi-level buffering architecture.

In the context of Operating Systems supports, the work in [24] proposes an architecture for enabling applications to define program-specific custom policies to carry on speculative execution, e.g. for I/O operations. Despite the clear lack of transparency (in fact, the programmer is required to manually specify policies to determine which speculative branch is correct, which ones should be rolled back, and to manually mark portions of code which can be executed speculatively, like in the TM case), this work mostly relies on system calls to communicate between the application and the Operating System, requiring a large number of mode/context changes which can significantly affect the overall performance, while we work completely in user space, thus avoiding costly mode/context changes. Additionally, speculation is implemented via an ad-hoc version of the `fork()` system call, in order to create a new speculative copy of the process, while in optimistic simulation speculation is already intrinsic in the execution model.

As hinted in the Introduction section, in the context of optimistic PDES engines, one approach to deal with I/O is based on ad-hoc API. One example along this direction is the work in [17], where the optimistic engine is able to provide the user-level software with past, committed (and globally consistent) state images. This is done by passing control (and the snapshot) to the applications via a proper callback. Once taken control, the callback can invoke output operations providing data related to the committed portion of the simulation. Compared to this approach, the present proposal does not require specific callbacks within the API. Also, in that solution the frequency of I/O production is bounded by the time period for the identification of the committed part of the computation (typically the GVT period). Hence not all the states passed through while executing the model can be observed (in terms of output stream production). Instead, with the present proposal we allow the modeler to produce output any time the event handler takes control, hence ideally at each state transition within the simulation. Further, the output is system-wide ordered, while in the approach in [17] it is ordered on a per-LP basis (hence post processing is required in order to provide a unique ordered trace along the simulation time axis).

Recently, the issue of transparency in optimistic synchronization in the context of HLA based simulations has been tackled [18], where the problem of direct external interactions (e.g. with the underlying Operating System) is also addressed. However, the provided solution is based on temporary suspension of processing activities at the federates until the interaction is conservatively (namely safely) detected to be committable, while our approach is based on post-filtering and post-manipulation of the output streams with no suspension of the actual run. Overall, we sustain pure optimism as opposed to the throttling scheme provided in [18].

A similar approach, but less transparent, has been adopted in the PDES framework in [4], where an ad-hoc API is provided to the application programmer in order to signal the kernel that a non-revocable I/O operation needs to take place. The effect on run-time is that a blocking period is experienced until the I/O is detected to be safe.

The approach provided in [10] has some resemblances with our proposal in that a special object (resembling our daemon) is used for I/O management. Any output message is routed towards this object in the form of an event, which gets processed only after GVT advancement (namely when the event is detected to be committed). However, this approach is based on ad-hoc APIs used to trigger the interaction with the special object, while we cope with the API provided by standard libraries. Also, the approach in [10] performs the I/O-event management within the simulation engine (hence leading to some I/O management latency along the execution path of the simulation kernel), while we rely on the orthogonal approach of removing most of the work for materializing the output from the simulation engine execution path.

Finally, a work related to the present proposal can be found in [6], where a scheme for optimizing checkpointing is provided in order to allow an interactive end-user to inject events prior to the commit horizon, and explore alternative paths (by resuming the computation from proper snapshots), as compared to those already established as committed. Orthogonality with our proposal lies in that we target the management of output streams, thus intrinsically coping with the typical scenarios where the input to the simulation run, e.g., in terms of simulation parameters, is provided at simulation startup and what-if analysis via simulation relies on batch processing schemes. Instead, that work targets the management of input streams, which is typical of, e.g., interactive what-if analysis approaches.

3. OUTPUT MANAGEMENT

3.1 Involved Issues

Consistent output management in optimistic simulations is a non-trivial task, due to the existence of the rollback

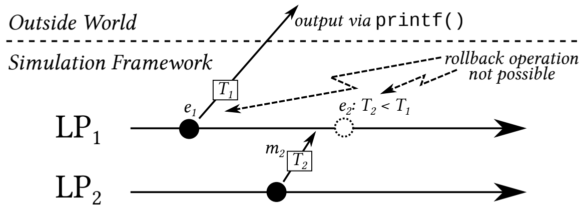


Figure 1: Interaction with the Outside World

operation, which is not handled by common libraries for managing output streams. Let us consider a simulation scenario where two LPs are present, namely LP_1 and LP_2. LP_1 is scheduled for the execution of event e_1, associated with timestamp T_1. During the execution of e_1, some output is produced via a call to the standard library's `printf()` function. After e_1's execution is completed, LP_2 generates a new event e_2 destined to LP_1, and associated with timestamp $T_2 < T_1$. According to the optimistic synchronization scheme presented in [9], e_1 is undone and the execution is restarted from e_2. However, the `printf()` invocation is already completed, the output has been already materialized on the standard output stream, and it is therefore inconsistent, as it is associated with an undone event. This is depicted in Figure 1.

The task of generating consistent output is even more complicated if we consider two different points of view on the same issue. On the one hand, the application-level programmer is interested in the *ease of use*, i.e. she does not want to rely on complex APIs which force her to be aware of the notion of rollback, and to look at the internal organization of the simulation engine. On the other hand, consistently managing output streams must not make the simulation engine pay a relevant performance penalty, in order to minimize the overhead on the actual simulation execution. This tradeoff between *application transparency* and *simulation performance* must be well balanced, in order to effectively and efficiently support the management of output streams.

In order to address *transparency*, the architecture which we hereby propose is based on compile/linking time facilities that, before creating the final simulation-model executable, redirect every output call in the application-level code from the standard `printf()` family functions to an ad-hoc simulation kernel's module, which represents the I/O-management-stub within the architecture. These facilities are present in most compiling tool-chains, and can be triggered via ad-hoc directives/scripts.

On the other hand, in order to enforce a *high performance* of the simulation execution, all the operations associated with the management of the output must exhibit an overhead—both direct and indirect—as small as possible. As for direct overhead, we have explicitly designed our proposal in order to minimize the use of system calls during the execution of the simulation for buffering the produced output until the generation-associated event is committed. In particular, at simulation startup, we pre-allocate a set of shared memory segments which are used by simulation kernel instances to store any output on any stream produced by the application-level code. These buffers are read, during the simulation run, by a separate user-space process, which we refer to as *output daemon*. This daemon is in charge of collecting all the output produced by the LPs, sort it, and—in case of a rollback operation—remove the involved strings. Whenever a set of events gets committed, the output daemon is able to use this information in order to actually materialize the relevant (committed) output on the associated streams.

As it will be thoroughly discussed later, this explicit design choice gives us several benefits: (i) output buffering is handled completely in user space—by relying on shared memory—avoiding costly mode/context changes during the execution of the simulation; (ii) simulation efficiency is preserved, as the operations for producing the output (independently of its eventual commitment) does not involve costly procedures, giving most of the available CPU time to actual (relevant) computation; (iii) considering that each simulation kernel is assigned a private shared memory segment, and considering that kernel instances only write on it, and the output daemon only reads from it, a completely non-blocking algorithm for the interaction between kernel instances and the daemon can be implemented, providing additional benefits due to the avoidance of contention on logical data (i.e. no locking primitives must be exploited to allow kernels–daemon communication).

On the indirect-overhead side, two main issues have been addressed, namely data locality and CPU sharing. In the simulation-kernel instances, the output is produced on a contiguous buffer (i.e the private shared-memory segment), avoiding costly cache-invalidation secondary effects during the generation of output on whichever stream. The segment is used in a cache-aligned fashion, avoiding at the same time false cache sharing effects. The daemon, on the other hand, handles the output materialization relying on an efficient modified version of a calendar queue [2], which is in turn able to minimize the computing time required for sorting the output strings, and it is additionally featured with an autonomic agent which is able to determine the best activation interval for reducing CPU-sharing effects on the actual simulation, and to reduce the delay from the generation of the output to its actual materialization.

3.2 Design Approaches and Reference Implementation

Without any loss of generality, we can say that a simulation framework comprises K simulation-kernel instances, scattered across M machines, and each machine m hosts a set of K_m simulation kernel instances. Of course, K_m can be different for each m, i.e. simulation kernel instances must not be necessarily evenly distributed across the machines, depending on, e.g., the actual computing power offered by each of them. Upon simulation startup, on each machine m a separate output-daemon instance is started, and K_m shared-memory segments are created (and shared with each kernel instance). These can be seen as per-kernel private *logical devices*, on which the simulation-kernel instances write their LPs' output messages, rather than on the requested stream. In this scenario, considering the whole system, D_M^K logical devices will be installed, and each kernel k hosted on machine m will have is own device D_m^k. In particular, as mentioned before, upon the invocation of some function of the `printf()` family (which is redirected to the kernel instance's I/O-management-stub at compile time), the output subsystem stores output-related information in a variable-sized data structure for keeping the information until the associated event gets committed (or rolled back). The structure used for this intermediate buffering is as follows:

```
struct output_msg_t {
    size_t size;
    int fd;
    unsigned int era;
    unsigned int LP;
    double timestamp;
    char buffer[];
};
```

where `size` keeps the total size of the entry, `LP` associates the output generation with a specific LP in the simulation, `timestamp` keeps information about the Local Virtual Time (LVT) at which the LP has generated the output, `fd` is the file descriptor associated with the output stream, and `buffer` is the actual output payload[1]. This entry is then written—before returning control to the application-level software—into the per-kernel logical device D_m^k. We note that `timestamp` is not required to be explicitly provided by the application layer within the output message (namely as a parameter of the `printf() call`). Rather, we assume the corresponding LVT value (as well as the LP-identifier) must be provided by the simulation engine to the I/O-management-stub right before passing control to the LP for actual event processing. In this way, the application modeler is not forced to tag all (or part) of the output messages with timestamping information. She will be anyhow transparently provided with an output-stream content matching (system-wide) the advancement of simulation time.

The logical device can be implemented as a circular buffer, where kernels' output subsystems write new output messages (which are later processed by the daemon in a FIFO order) with an ad-hoc header for describing the content of the buffer. The organization of this device is described via the following structure:

```
struct logical_device_t {
   size_t size;
   unsigned int wrote;
   unsigned int read;
   unsigned char buffer[];
};
```

where `size` identifies the (current) size of the circular buffer, `wrote` is an offset pointing to the last byte in the device written by the kernel output subsystem, and `read` is an offset identifying the last byte read by the output daemon. Since these fields are updated in isolation (i.e. the field written by the kernel instance is only read by the daemon, and vice versa), a non-blocking algorithm has been developed for managing the logical device. In particular, by checking if `wrote` and `read` store the same value (either in modular or non-modular arithmetic), the daemon and the kernel are able to determine whether the buffer is full or empty, and can immediately access via displacement the correct position on which to perform a read/write operation, without the need for any synchronization primitive. If the buffer is full, then the simulation kernel must wait before finalizing the output generation procedure, unless some resize procedure, like the non-blocking one described in Section 3.3, is adopted.

On a periodic basis, the output daemon wakes up and checks whether some output message has been produced by some locally-hosted simulation-kernel instance on its private logical device (again, this can be done by comparing the values of `read` and `wrote`). In case a new output message is present, the output daemon creates a local copy of it (i.e. in its own address space), updates the `read` value, and inserts it in a specifically-modified version of a calendar queue [2], which guarantees the correct ordering of the output messages by all the output kernels. We also note that, in case the two counters match, but the actual content of a new message has already been inserted within the device (which does not impose atomicity with the update of the `wrote` counter), the demon will simply experience a false-negative on the presence of new messages, which will be resolved at subsequent iteration steps.

Overall, with the above approach, no spin-locks or other types of atomic operations (which are known to exhibit non-minimal direct and indirect costs, possibly impacting the execution speed at the side of the simulation engine) are used at all. This is done by trading-off with the actual delay for the delivery of the message at the demon.

In order for the output daemon to correctly support the execution of rollback operations, the traditional implementation of the calendar queue has been augmented with one additional operation, namely `delete`, which allows to remove elements falling in a given $[from, to]$ timestamp range. For efficiency reasons, upon the invocation of this operation, the buckets associated with $from$ and to are computed, and then a linear search for removing elements is performed, resembling the original direct search described in [2]. Of course, the delete operation removes only the elements which are associated with a particular LP, as specified in the `output_msg_t` structure. At the same time, we note that, depending on the actual simulation execution, scanning all the buckets in the $[from, to]$ range can be a relatively costly operation, if we consider that we do not want to affect the simulation's performance. To this end, the `output_msg_t` structure keeps track of the `era` field, namely a monotonic counter which is updated by every simulation kernel upon the execution of a rollback operation, on a per-LP basis. This can be regarded as a compressed information identifying every output message related to a given era comprised in between two rollback operations. The calendar queue bucket array is therefore augmented, keeping for every bucket a bloom filter [1] which is used to know whether output messages from a given era are present in the bucket. In this way, before starting to scan the list associated with any bucket in the $[from, to]$ interval, a check on the bloom filter is performed, to determine whether that particular bucket can be skipped.

To enable the execution of a rollback operation, and to enforce a timely materialization of output messages on the related streams, two control messages are placed by kernel k (namely by the corresponding I/O-management-stub) on device D_m^k, namely `ROLLBACK` and `COMMIT`. The former is a control message with a payload structured as $\langle from, to, LP, era \rangle$, which directly triggers the aforementioned `delete` operation on the calendar queue. The latter, notifies the output daemon on which portion of the per-kernel-generated output can be considered as committed[2]. Whenever a `COMMIT` control message is found on a device, the associated commitment timestamp is stored into a separate array, which keeps track of the last commitment time for each locally-hosted simulation kernel instance k. Periodically, the output daemon finds the minimum commitment time $T_{C_{min}}$ among the ones notified by the kernel instances, and invokes a `flush`

[1]We note that, if output generation entails processing a format string, this can be easily done by relying on the POSIX `sprintf()` library function.

[2]In fact, the `COMMIT` control message is generated immediately after the completion of the GVT reduction operation.

operation on the calendar queue, which iteratively dequeues elements from it until an element associated with a timestamp $T > T_{C_{min}}$ is found. Every output message dequeued during this flush operation is materialized on the associated stream, described by the `fd` field.

We note that these operations can be non-negligible in terms of CPU usage by the output daemon. In order to produce a minimal impact on the simulation's overall performance, the daemon does not process all the available messages (either output or control) from $D_m^k \forall k$ before returning to sleep. On the other hand, after having processed any message, it checks whether its execution has lasted more than a specific threshold value and, in the positive case, it returns to sleep. Of course, the overall performance can be affected by the sleep and working time. As it will be discussed later in Section 3.3, an ad-hoc solution can be adopted to autonomically self-tune these parameters for maximizing the simulation's performance, while ensuring a timely materialization of committed output messages.

As an additional note, since the proposed output message data structure identifies a particular stream using its file descriptor, the kernel instance's output subsystem intercepts calls to functions which actually open/close streams (e.g., `fopen()`-family calls) and produces on the logical device additional control messages (namely, `OPEN` and `CLOSE`) which tell the output daemon to process these operations. In this way, the daemon is the only user-space process controlling the actual output streams, for materializing committed output messages on them.

In case the simulation is executed on a distributed architecture (either a cluster, a desktop grid, or on the cloud), the daemon-based approach is able to communicate only with the kernel instances which are locally hosted on the same machine. In order to collect output messages from all the nodes in the system, the output daemon can be configured to act as a *forwarder*, i.e. whenever a set of output messages is detected to be committed, the messages are forwarded on the network to an additional (remote) instance of the daemon which acts as a *collector*. This additional instance, which can be even geographically far from the actual simulation architecture, receives all the committed messages from every daemon running in the simulation framework, and inserts them into an additional calendar queue. Whenever `COMMIT` messages are forwarded, the collector daemon relies on the same aforementioned logic for materializing the output messages on the associated streams. We note that this approach is effective in two ways: (i) over the network we transfer only committed output, thus we early process rollback operations, and avoid to pay costly network delays for exchanging output messages which might not be committed, enabling for a considerable scalability even at a geographical scale, and (ii) we can finalize the materialization of the output messages on a remote (possibly not dedicated to simulation) machine, allowing both an enhanced simulation performance (in fact, materializing output on a stream can be a non-minimal-cost operation) and the possibility to easily monitor the execution of the simulation by the user on a separate machine, without suffering a considerable delay.

As a final note, we mention that the output daemon can be flagged to work in *autocommit* mode. Using this facility, the daemon (running in the aforementioned forwarding mode) can be connected to a conservative distributed simulation framework, where the rollback issue is not present, but nevertheless for producing consistent output a timestamp-based ordering must be performed on the output messages produced by the various nodes in the system. Therefore, in this scenario, the facilities provided by our proposal would never trigger a `delete` operation, but upon receiving a message, its timestamp is considered as the last committed timestamp for the specific LP. By simply performing a local reduction across the current LVT of the LPs (which, as said is communicated by the simulation engine to the I/O-management-stub while the simulation proceeds), the set of (timestamp-ordered) output messages to be flushed is identified as the set of those messages with timestamp less than the reduction value, which are therefore immediately flushed (i.e. forwarded) for output materialization.

To give the final picture, we list the core set of APIs that are provided by the I/O-management-stub, which can be used for integration with differentiated simulation kernel layers:

`commit_time(GVT)`: This function is exposed by the output subsystem in order to allow the GVT reduction subsystem to notify the output daemon of a newly-computed GVT value. This information is used by the subsystem to generate a `COMMIT` control message to notify the output daemon.

`set_LVT(LP, timestamp)`: Using this function, the simulation kernel's scheduler can notify the output subsystem about the identity of the dispatched LP, and the timestamp of the dispatched event (hence the logical time to be associated with any output message produced as a result of processing this event).

`rollback(from, to, LP)`: Using this function, the simulation kernel's scheduler can notify the output subsystem about the reception of a straggler message or an antimessage. Using this information, the output subsystem can generate a `ROLLBACK` message for the output daemon, and instantiate a new era.

`out_msg(LP, stamp, msg, stream)`: This API is used by the simulation kernel to transfer a particular output message destined to a specified stream to the output subsystem, which will in turn write it on the kernel's logical device D_m^k.

`autocommit(flag)`: If `flag` is true, every output message received is considered as non-rollbackable, in order to support the integration with a conservative simulation engine. If `flag` is true, the daemon does not wait for `COMMIT` messages before committing output messages. The commitment is triggered by the aforementioned internal logic.

We recall that, in order to provide the application-level developer with complete transparency, calls to output generation library functions must be properly wrapped, in order to correctly redirect them to `out_msg()`. This can be easily done, as already described, by relying on compile-time facilities provided by most linkers, and by applying small changes on the application-program build rules. The interconnection with the simulation kernel, particularly the ROOT-Sim kernel we are exploiting in this study, is depicted in Figure 2.

3.3 Optimizations

In case of an output-intensive simulation model (either because a large number of events involve the generation of some output destined to some stream, or because the output messages are considerably large) the logical device might become the bottleneck of the system. In particular, since we want to ensure *exactly-once* materialization of the output

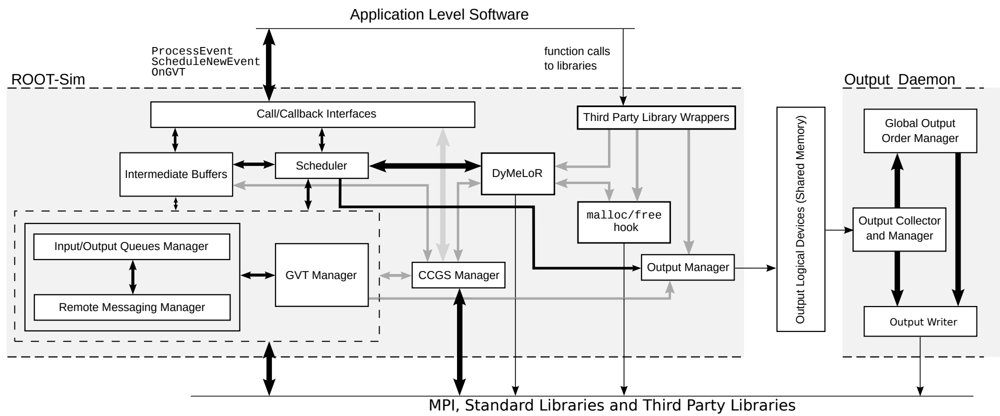


Figure 2: ROOT-Sim vs Output Daemon Architecture

messages, if the circular buffer which is used to implement the logical device gets filled, the kernel's output subsystem trying to write on it must wait for the output daemon to free some space for buffering the output.

Since this would be a non-affordable cost, we have augmented the `logical_device_t` data structure with an additional flag, namely `subst_id`, which is used by the output subsystem whenever the current circular buffer is full. In this case, the output subsystem allocates a new shared memory segment (of doubled size with respect to the old, full one), stores its id in this field, detaches from the old segment, and starts buffering output messages on this new device. Whenever the output daemon finds a logical device to be empty, it checks whether the `subst_id` field is set to a valid value. In the positive case, it attaches to the new logical device, releases the old one, and starts processing messages from the new one.

If on the one hand this approach can reduce the time spent during the execution of a simulation event inside the kernel's management subsystems, on the other it can affect memory usage if the simulation model's activity is highly output-bound. Although this can be regarded as a secondary problem, considering the large amount of memory available on modern architectures, optimizing the actual tradeoff between CPU usage by the output daemon and memory consumption by the logical device can provide benefits in terms of output materialization delay and problem feasibility.

We therefore propose an additional optimization, which addresses the amount of time the output daemon spends processing messages from the devices and sleeping, and allows the output manager to autonomically self tune its activation phase. In particular, as for output materialization delay, given that output messages are flushed on the associated streams after the GVT computation (which is, traditionally, a periodic operation), the best scenario arises when the kernels' output subsystems try to write the `COMMIT` message on their devices and find them empty. This situation happens whenever the total execution time of the output daemon is such that every message placed in the device in between two `COMMIT` messages has been processed. We therefore measure the execution time of each message (namely, output message's t_o, a `COMMIT` message's t_c, and a `ROLLBACK` message's t_r) and continuously update their mean values $\bar{t}_o$, $\bar{t}_c$, and $\bar{t}_r$ according to an exponential mean. Additionally, we rely on three counters, namely $\bar{c}_o$ $\bar{c}_c$, and $\bar{c}_m$ which describe the (exponential) average number of messages placed by simulation kernel instances in a GVT phase. By relying on these values, upon the reception of a `COMMIT` control message, the output daemon computes the expected total execution time $\mathbb{E}(T) = \sum_{x \in \{o,c,r\}} \bar{t}_x \cdot \bar{c}_x$ which is needed for emptying the logical devices during the next phase. If this value is lower than a compile-time specified threshold, it is then fragmented into several slots (resembling Operating Systems' time slices) and the sleep time is set accordingly in order to identify an execution phase which is equal to the GVT phase. The compile-time threshold ensures that, in case an application is extremely output-bound (at least in some phase of its execution), then the output daemon will not significantly affect the overall simulation performance, while trying to minimize the output materialization delay. We note that, by relying on this approach, the output daemon is able to capture variations in the actual output generation dynamics by the application-level software, thus adapting to execution phases which exhibit a higher/lower output generation rate.

The last optimization concerns timely processing of output-messages rollback, which can prevent the output daemon to process wrong data before noticing that it must be undone. In particular, upon initialization of the per-kernel logical device, each simulation-kernel instance installs a second channels for the delivery of high-priority control messages (namely the `ROLLBACK` ones) which are used to early inform the daemon that some information that it is about to process might be already uncommittable. Whenever the daemon notices that a `ROLLBACK` message is present in the high-priority channel, it starts scanning the associated logical device and marks as *don't care* every output message which should be rolled back. The circular buffer is scanned (without updating the `read` flag) until the `ROLLBACK` control message corresponding to the one found in the high-priority channel is found. When this operation is completed, the normal behavior is restored, with the exception that whenever a don't-care output message is found in the circular buffer, it gets simply discarded. We note that this rollback optimization avoids enqueueing and dequeueing the output messages which can be early detected as uncommittable, reducing the cost of the operations on the calendar queue[3]. If some events involved in the rollback operation were already

[3]This situation might result in a calendar queue's resize, which is a very costly operation.

in the calendar queue, when the processing of the logical device (which, we recall, stores messages emitted by the associated simulation kernel in FIFO order) reaches the `ROLLBACK` control message, the `delete` operation described in Section 3.2 is executed, which removes the already inserted output messages.

4. EXPERIMENTAL DATA

As hinted, we have integrated the proposed output-streams management architecture within ROOT-Sim, an open source C/MPI-based simulation package targeted at POSIX systems [8, 15], which implements a general-purpose parallel/distributed simulation environment relying on the optimistic synchronization paradigm. ROOT-Sim offers a very simple programming model based on the classical notion of simulation-event handlers, to be implemented according to the ANSI-C standard, and transparently supports all the services required to parallelize the execution. It also offers a set of optimized protocols aimed at minimizing the run-time overhead by the platform, thus allowing for high performance and scalability.

The output management subsystem has been integrated respecting the API described in Section 3.2, while the output daemon has been realized as a separate process written in ANSI-C. Upon the ROOT-Sim startup, if requested, the output daemon is launched and the output subsystem installs the logical devices (one for every simulation-kernel instance) .

The hardware architecture used for testing our proposal is a 64-bit NUMA machine, namely an HP ProLiant server, equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 cores (for a total of 32 cores) that share a 12MB L3 cache (6 MB per each 4-cores set), and each core has a 512KB private L2 cache. The operating system is 64-bit Debian 6, with Linux Kernel version 2.6.32.5. The compiling and linking tools used are `gcc` 4.3.4 and binutils (`as` and `ld`) 2.20.0. This hardware architecture has been used only for simulation (i.e. no other process, except for the system ones, where running during the execution), thus resembling a scenario where the hardware environment is dedicated to high performance simulation.

In order to evaluate different aspects of the proposed architecture, we have conducted experiments on a family of configurations of *Personal Communications Service* (PCS), a GSM wireless communication systems simulation model, where channels are modeled in high fidelity via explicit simulation of power regulation/usage and interference/fading phenomena on the basis of the current state of the corresponding cell. Also, the power regulation model has been implemented according to the results in [12]. Accurate descriptions of this model can be found in [21]. However, for the reader's convenience we report below some details related to the main features of the model.

Upon the start of a call destined to a mobile device currently hosted by a given wireless cell, a call-setup record is instantiated via dynamically-allocated data structures, which gets linked to a list of already active records within that same cell. Each record gets 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. Upon call-setup, power regulation is performed, which involves scanning the aforementioned list of records for computing the minimum transmission power allowing the current call-setup to achieve the threshold-level SIR value. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a meteorological model defining climatic conditions (and related variations). The employed simulation models have been developed for execution on top of ROOT-Sim in a way that each LP models a single wireless cell. Hence, the event-handler callback involves the update of individual cells' states, and cross-LP events are essentially related to hand-offs between different cells.

We have performed a set of experiments where each cell sustains the same workload of incoming calls, whose inter-arrival time is exponentially distributed, and whose average duration is set to 2 min. The expected rate for call inter-arrival has been set to achieve channel utilization factor on the order of 30%, while the residence time of an active device within a cell has a mean value of 5 min and follows the exponential distribution. For the above scenario, we have run experiments with 1024 wireless cells, modeled as hexagons covering a square region, each one managing 1000 wireless channels. These have been evenly distributed across 32 kernel instances running on the 32-core underlying machine.

For measuring the overall performance of the simulation runs, we have relied on the measurement of cumulated committed events over wall clock time advancement, i.e. a measure of how many events get committed while the simulation's execution is carried on.

To capture different aspects of the impact on execution dynamics by the output-management subsystem and the output daemon, we have modified the PCS benchmark in order to provide on the terminal statistics regarding the occurrence of the hand-off event. In particular, with a certain frequency f, the execution of the hand-off event entails the generation of an output string which tells the total amount of per-cell hand-off events so far, and the average duration of a handed-off call. We have explicitly varied the value of f so that, among all the events (not only hand-off events) executed in the simulation run, in between 1% and 35% of them involved some output generation (which corresponds, in our configuration, to a total number of generated string in between 5 millions and 35 millions).

We note that, for the above deploy/parameterization, the runs exhibited an efficiency on the order of 80%. Hence the experimentation has been carried out when considering well behaving optimistic runs (namely not affected by thrashing, which would lead the experimentation to be non-reliable), which however show an amount of rollback that is expected to provide a good test case for all the functionalities (e.g. output-message discarding functionalities) offered by our output management subsystem. Further, running the above model on top of ROOT-Sim (on the same multi-core machine used in this experimental study) has been already shown to give rise to super-linear speedup values (see the experimental data provided in [21]). Hence our experimentation is carried out via competitive parallel runs.

We have compared this execution with two different scenarios, one relying on the traditional ROOT-Sim framework (i.e. without the output subsystem and without the output daemon), and one with the I/O-management-stub working within the simulation kernel, but without any daemon listening on the other side of the logical device (in this case the device has been configured like a typical `/dev/null` device file, by simply discarding the incoming messages). We consider the former scenario as a good baseline situation, for assessing the overall overhead introduced by the output management architecture, while the latter allows us to evaluate what is the actual impact of having a separate process (the daemon) running in time-sharing on a dedicated sim-

ulation environment, for supporting the execution of additional housekeeping tasks. Further, we have also considered runs based on a modified version of ROOT-Sim where the output strings produced by the execution of an event are logged within the event queue (as a list associated with the event-buffer), and are then flushed when fossil collecting the event after GVT computation. This approach has resemblances with the solution provided in [10]. We note however that this approach does not provide global ordering of the output across the whole set of LPs.

By the results, shown in Figure 3 (obtained as the average over 10 runs done with different pseudo-random seeds), we note that the execution with the device configured like a `/dev/null` device file exhibits a reduced overhead with respect to the baseline configuration, showing that the operations internal to the I/O-management-stub (and the interaction between the simulation engine and the stub) do not impact significantly on the overall simulation performance. When the output daemon is running, on the other hand, the simulation throughput decreases when the application-level software exhibits more output-bound behavior. This is reasonable, considering that the CPU time required by the daemon for a timely processing of the messages placed into the logical device gets increased. However, the worst case for the overhead is on the order of 22%. Further, the overhead well scales vs the increase of the frequency of output production (in fact the overhead is quite similar for the cases where f is set to 12% and 35%). On the other hand, for very reduced output-message frequency (say f set to 1% or 7%), as typical of when the application is configured for primarily matching performance requirements via audit reduction, the overhead introduced by our output-management architecture is quite bounded (namely between 2% and 11%). It is interesting to notice that our proposal shows a better performance than the scenario with no output architecture activated, but with I/O calls performed while processing the events. This phenomenon is due to the fact that the `stdio` library is not optimized for integration with high performance computing, while our architecture has been oriented exactly to this scenario. Further, standard I/O calls during event processing would even give rise to non-consistent output, due to the fact that output materialization associated with rolled back events is finalized as well (in the case of optimistic synchronization), or should require an additional post processing to correctly order the output (in case of the conservative synchronization), thus requiring additional time for the end user to be able to perform its audit activities. Similar considerations can be made when considering I/O management via logging of the strings with the processed events and flush operations after GVT computation. Particularly, while this approach introduces negligible overhead with bounded frequency of output production, the overhead gets significantly increased for higher frequency of output message generation.

Another aspect which requires attention regards the overhead induced by the evaluation of parameters within a format string being passed to the output subsystem. In Figure 4, we present three different curves (whose samples are again computed as the average over 10 runs), one entailing the evaluation of a float, one of an integer, and a case where no parameters should be evaluated at all. We recall that, in our proposal, parameters are evaluated immediately within the I/O-management-stub (directly called by the application-level code through library call redirection) via the POSIX-compliant `sprintf()` library function. By the results, we can see that the overall execution time is

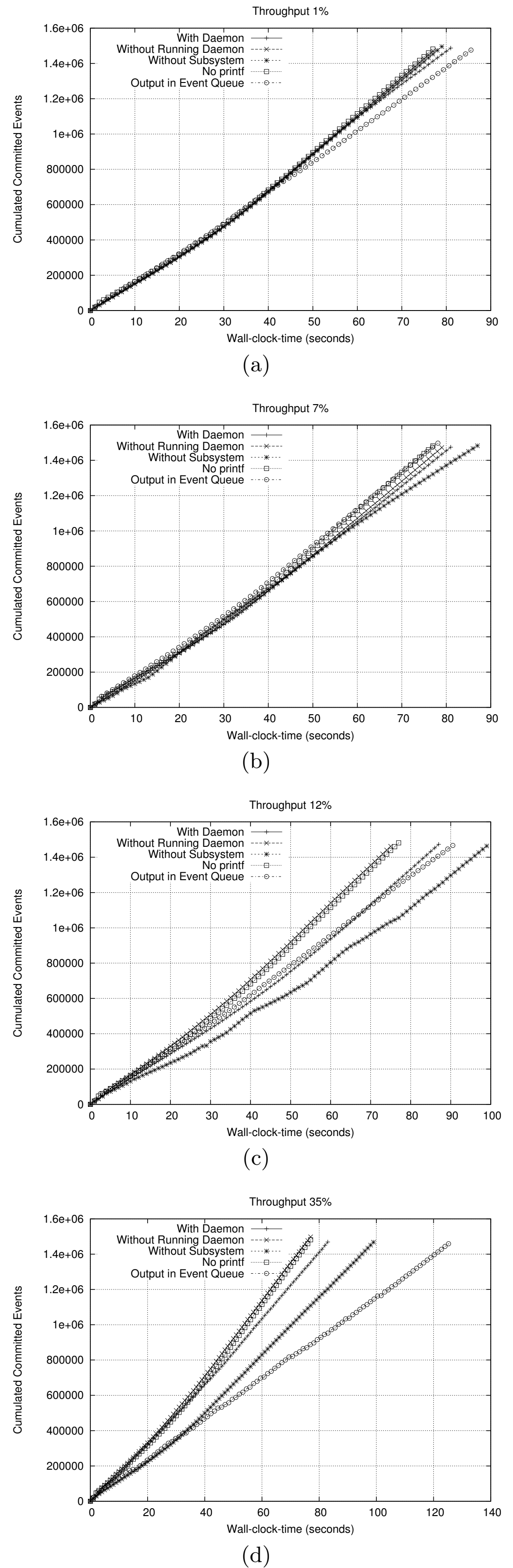


Figure 3: Throughput for Different Values of f

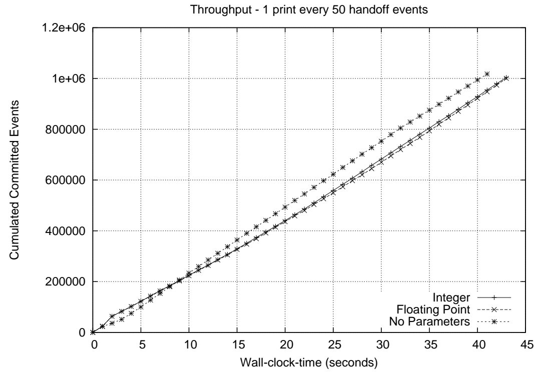
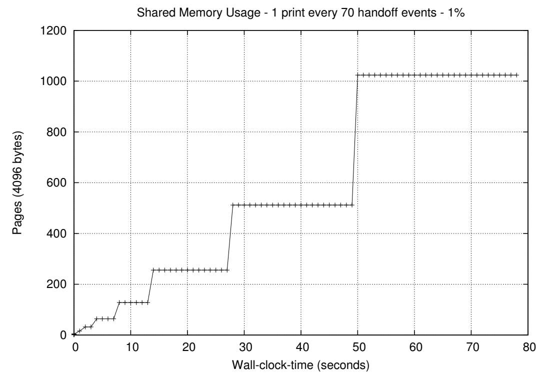


Figure 4: Throughput with Different Data Types

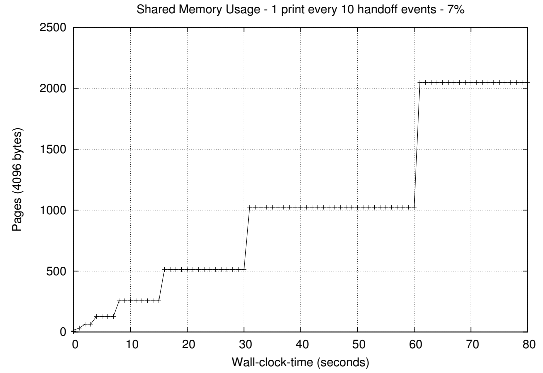
not significantly affected by the presence of additional (more complex) parameters to be evaluated, showing the effectiveness of our proposal even in the case of relatively complex outcome-messages from a simulation model.

To show how the CPU-usage/memory tradeoff is addressed by the output daemon, in Figure 5 we report the total amount of shared memory allocated for the logical I/O-devices during the execution of the simulation (these plots refer to one of the 10 runs, where similar behavior is anyway observed). As in the case of the throughput discussed earlier, we note that (as expected) the amount of required shared memory is increased when the application exhibits a more output-bound behavior. However, the reached bound (on the order of 16MB) represents a relatively reduced absolute value (especially when considering that optimistic simulation is known to be memory consuming on the side of the engine). Further, being such a memory virtualized by the underlying Operating System, and thanks to the fact that the shared memory segments implementing the logical I/O-device are used according to the circular rule (not in scattered mode), we may expect a reduced impact on the actual locality while the frequency of output-message generation gets increased.

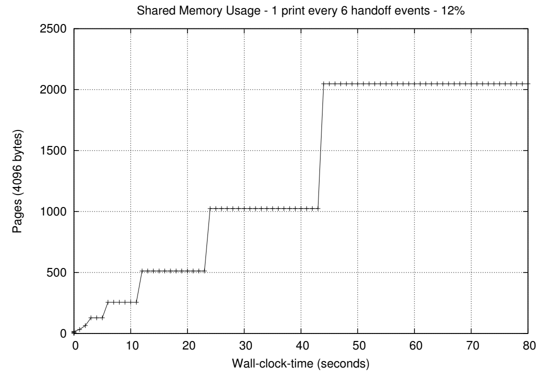
In Figure 6 we present a plot which shows what is the actual output–materialization delay exhibited by the output daemon (again for one of the ten runs, which is anyhow representative of what observed in the different runs). Both the x and the y axis represent wall clock time. To the x axis we associate the time at which a specific committed output message was produced, while to the y axis we associate the time at which the same output message was materialized. For the sake of clarity, we show a 45-degree curve which represent a (theoretical) instantaneous materialization time, i.e. a situation where there is no actual delay between the generation and materialization. In this plot, the steeper the slope, the higher is the materialization delay induced by the operations by the output daemon. It is interesting to notice, by the plots, that the output materialization advances in steps. This is related to the fact that the commitment operation of output messages can be started only after a GVT calculation, which is a periodic operation. It is interesting to note that in Figures 6 (a), and (c), the autonomic self-tuning subsystem for CPU/memory tradeoff optimization is able to capture the best configuration to minimize the materialization delay. In fact, the slope of the curve, during the simulation execution, tends to get gentler. The case in Figure 6 (b) is quite different, as the



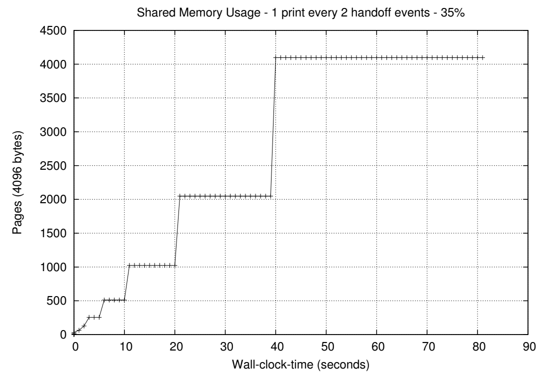
(a)



(b)



(c)



(d)

Figure 5: Shared Memory Size

daemon tries to minimize the output materialization delay, but then the CPU threshold is hit, and the daemon is not able to increase its computing power usage, in order not to significantly affect simulation performance, and therefore the curve diverges. The scenario in Figure 6 (d) shows that, considering the large amount of output messages, the calendar queue becomes the bottleneck of the system (due to an upper bound on the number of buckets, as described in [2]), and the rollbacks which are encountered during the simulation determine an amount of operations on the calendar queue to delete messages which were already stored. However, we note that for values of f from 1% to 7% (which, as said, would represent a case of orientation of the application layer to performance due to reduced audit on model execution), we get that upon run termination the output stream has been already materialized at the 75% or, at least, the 35%, which would enable pipelined treatment of the output data while the run is still in progress. The careful reader might notice that with a given generation wall-clock time, more materialization times are associated. This is related to the fact that these plots present system-wide materialization delays, where different kernel instances at the same WCT value might generate output messages from LPs running at different LVT values. This skew is therefore reflected in the commitment (wall-clock) time at which a set of messages can be safely materialized on the associated output stream.

To assess the effects of the autonomic self-tuning mechanism for daemon activation, described in Section 3.3, we present in Figure 7 a scenario where the parameter f (which describes the frequency of statistics generation on standard output) is not constant over the simulation run, rather varies in the range $[1\%, 30\%]$ in an interleaved fashion. In particular, f is set to 1% at the beginning of the simulation, and is then incremented until it reaches the value of 30%, and then again decreased. The plots show that the autonomic self-tuning system is able to cope well with the dynamics variations. In fact there is not any significant skew in the simulation throughput, despite the output daemon continuously requests for more or less computing power, depending on the actual load phase on the output architecture.

5. CONCLUSION AND FUTURE WORK

In this work we have presented design indications for the development of an efficient shared memory based output streams management subsystem targeted at optimistic simulations. It allows a simulation model to rely on standard library calls for producing output on different streams during the execution of not-yet-committed events, and to materialize this output only when the corresponding events get committed, in a system-wide timestamp ordered manner. Further, reduced overhead is introduced at the side of the simulation engine, which allows for not hampering the speedup that is expected to be achieved when executing simulation models on optimistic PDES systems. Future work entails the development of ad-hoc policies for limiting the execution's optimism in case the simulation environment is running out of memory. To complement the results of the experimental study, which has been tailored to the analysis for shared-memory multi-core architectures, we plan to investigate on the effects of our proposal in the context of distributed memory systems (e.g. clusters).

6. REFERENCES

[1] B. H. Bloom. Space/time trade-offs in hash coding

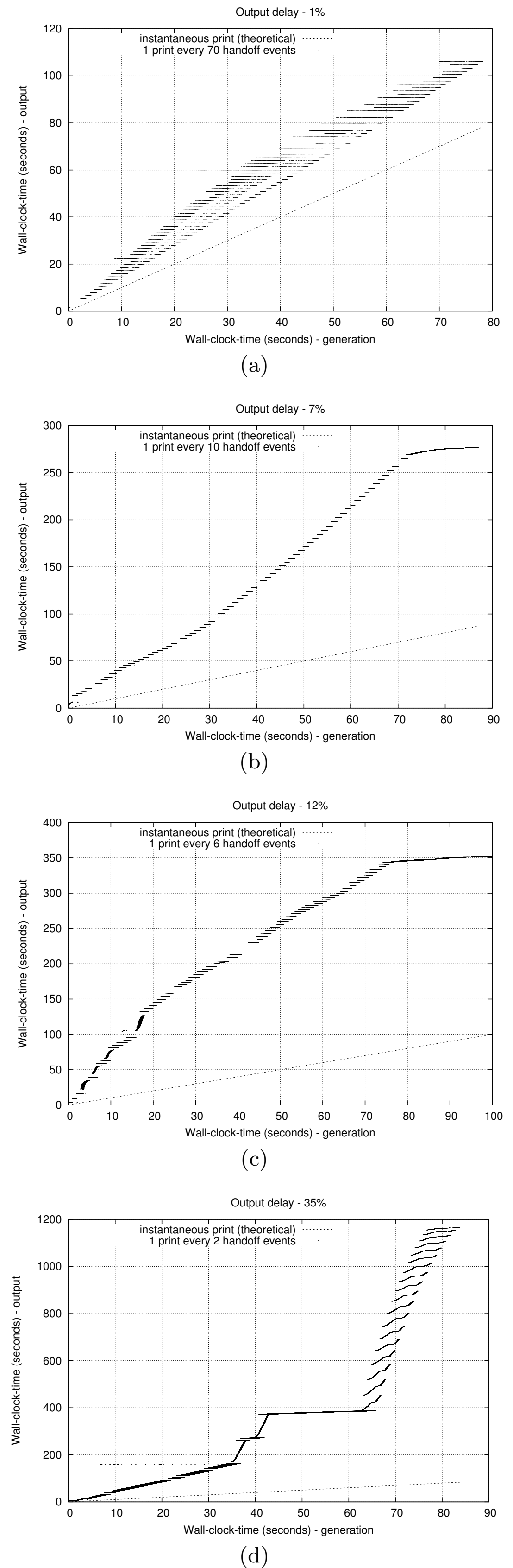


Figure 6: Generation/Materialization Delay

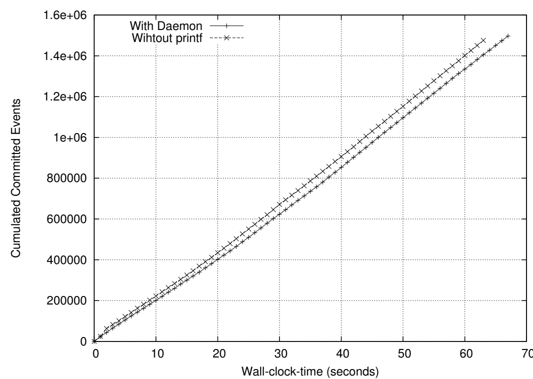


Figure 7: Frequency Variation

with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

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

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

[4] S. R. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: a time warp system for shared memory multiprocessors. In *WSC '94: Proceedings of the 26th conference on Winter simulation*, pages 1332–1339. Society for Computer Simulation International, 1994.

[5] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, sept 2002.

[6] S. Franks, F. Gomes, B. Unger, and J. G. Cleary. State saving for interactive optimistic simulation. In *Workshop on Parallel and Distributed Simulation*, pages 72–79, 1997.

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

[8] H. R. Group. ROOT-Sim: The ROme OpTimistic Simulator - v 1.0. `http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/`, Oct. 2012.

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

[10] D. R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. D. Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, L. V. Warren, J. J. Wedel, H. Younger, and S. Bellenot. Distributed simulation and the time wrap operating system. In *SOSP*, pages 77–93, 1987.

[11] D. B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, RDS, pages 86–95, Oct. 1993.

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

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[14] D. E. Martin, T. J. McBrayer, and P. A. Wilsey. WARPED: A time warp simulation kernel for analysis and application development. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, page 383. IEEE Computer Society, 1996.

[15] A. Pellegrini, R. Vitali, and F. Quaglia. The ROme OpTimistic Simulator: Core internals and programming model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools. ICST, 2011.

[16] A. Pellegrini, R. Vitali, and F. Quaglia. Transparent and efficient shared-state management for optimistic simulations on multi-core machines. In *Proceedings 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 134–141. IEEE Computer Society, Aug. 2012.

[17] F. Quaglia. On the construction of committed consistent global states in optimistic simulation. *International Journal of Simulation and Process Modelling*, 5(2):172–181, 2009.

[18] A. Santoro and F. Quaglia. Transparent optimistic synchronization in the high-level architecture via time-management conversion. *ACM Trans. Model. Comput. Simul.*, 22(4):21, 2012.

[19] SPEEDES. `http://www.speedes.com`, 2005.

[20] R. Vitali, A. Pellegrini, and F. Quaglia. Autonomic log/restore for advanced optimistic simulation systems. In *Proceedings of the Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 319–327. IEEE Computer Society, 2010.

[21] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 26th International Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 211–220. IEEE Computer Society, Aug. 2012.

[22] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 285–296. ACM, 2008.

[23] D. West and K. Panesar. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 78–85. IEEE Computer Society, May 1996.

[24] B. Wester, P. M. Chen, and J. Flinn. Operating system support for application-specific speculation. In *Proceedings of the sixth conference on Computer systems*, EuroSys, pages 229–242. ACM, 2011.