Consistent and Efficient Output-Streams Management in Optimistic Simulation Platforms

SAPIENZA
UNIVERSITÀ DI ROMA

Francesco Antonacci
Alessandro Pellegrini
Francesco Quaglia

High Performance and Dependable Computing Systems Group
Sapienza, University of Rome

PADS 2013

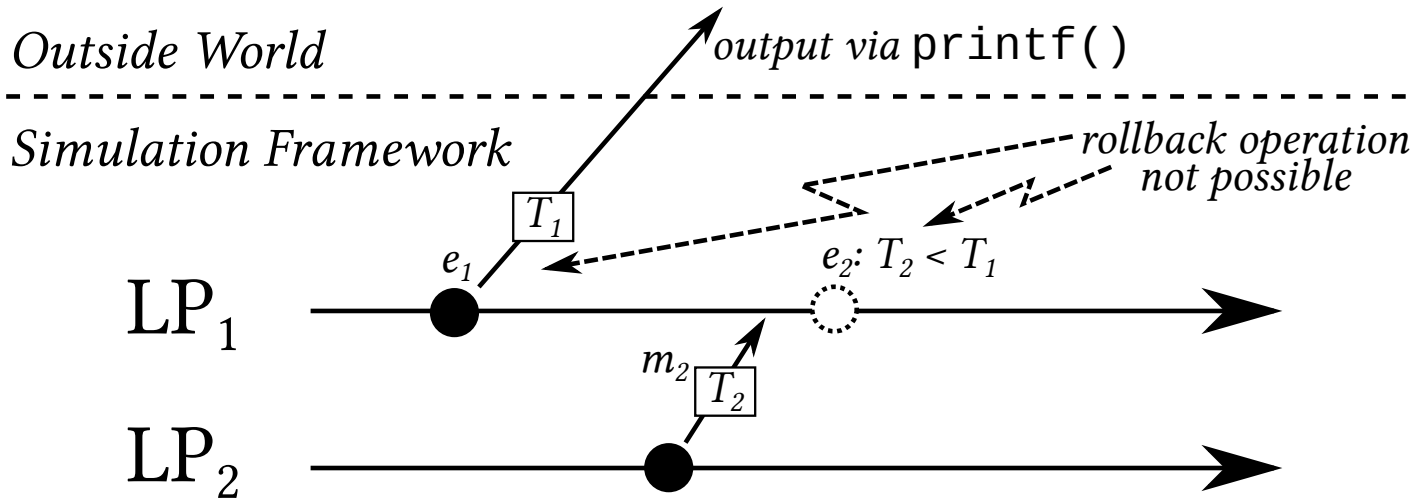
Motivations

- (Distributed) Parallel Discrete Event Simulation and Optimistic Synchronization are great for supporting highly efficient simulations
- Nevertheless, rollbacks pose some limitations

Motivations

- (Distributed) Parallel Discrete Event Simulation and Optimistic Synchronization are great for supporting highly efficient simulations
- Nevertheless, rollbacks pose some limitations
- Interactions with the *outside world* can be tricky: is it aware of rollbacks?
- Yet, (timely) output generation could be vital in simulation:
  - Interaction with the user
  - Evaluation of global parameters
  - Real-time visualization of the simulation

Motivations (2)



Motivations (3)

- Several viable solutions have been proposed:
  1. Ad-hoc output-generation APIs provided by simulation frameworks
  2. Temporary suspension of processing activities until output generation is safe (*delay until commit*)
  3. Storing output messages in events, and materializing during fossil collection

Motivations (3)

- Several viable solutions have been proposed:
  1. Ad-hoc output-generation APIs provided by simulation frameworks
  2. Temporary suspension of processing activities until output generation is safe (*delay until commit*)
  3. Storing output messages in events, and materializing during fossil collection

- These solutions have drawbacks:
  1. Programming model is not transparent to the user
  2. Overall simulation performance might be degraded (especially when there is a dense output flow)
  3. Output is not system-wide ordered

Goals

- We propose a general approach to output generation in (distributed) PDES
- Simulation execution is never stopped
- The model writer can rely on standard output generation libraries
- Output is system-wide ordered
- Inconsistent output is never shown to the user
- Overall, provides the illusion of a sequential programming model

Goals

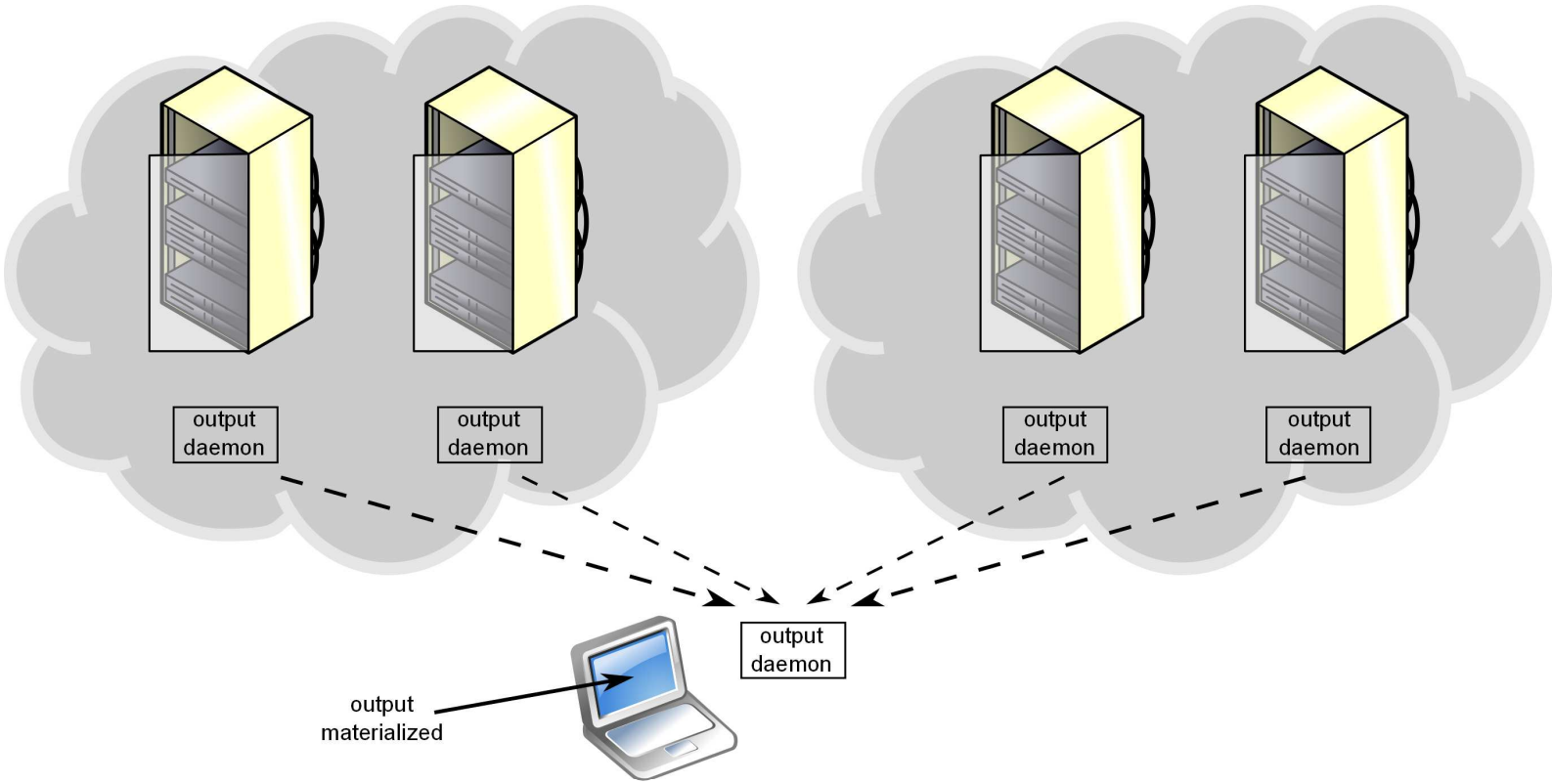
- We propose a general approach to output generation in (distributed) PDES
- Simulation execution is never stopped
- The model writer can rely on standard output generation libraries
- Output is system-wide ordered
- Inconsistent output is never shown to the user
- Overall, provides the illusion of a sequential programming model

- We have targeted:
  - Output generation via `printf()` family functions
  - ANSI-C programming language
- We have implemented our proposal within the ROme OpTimistic Simulator (ROOT-Sim)

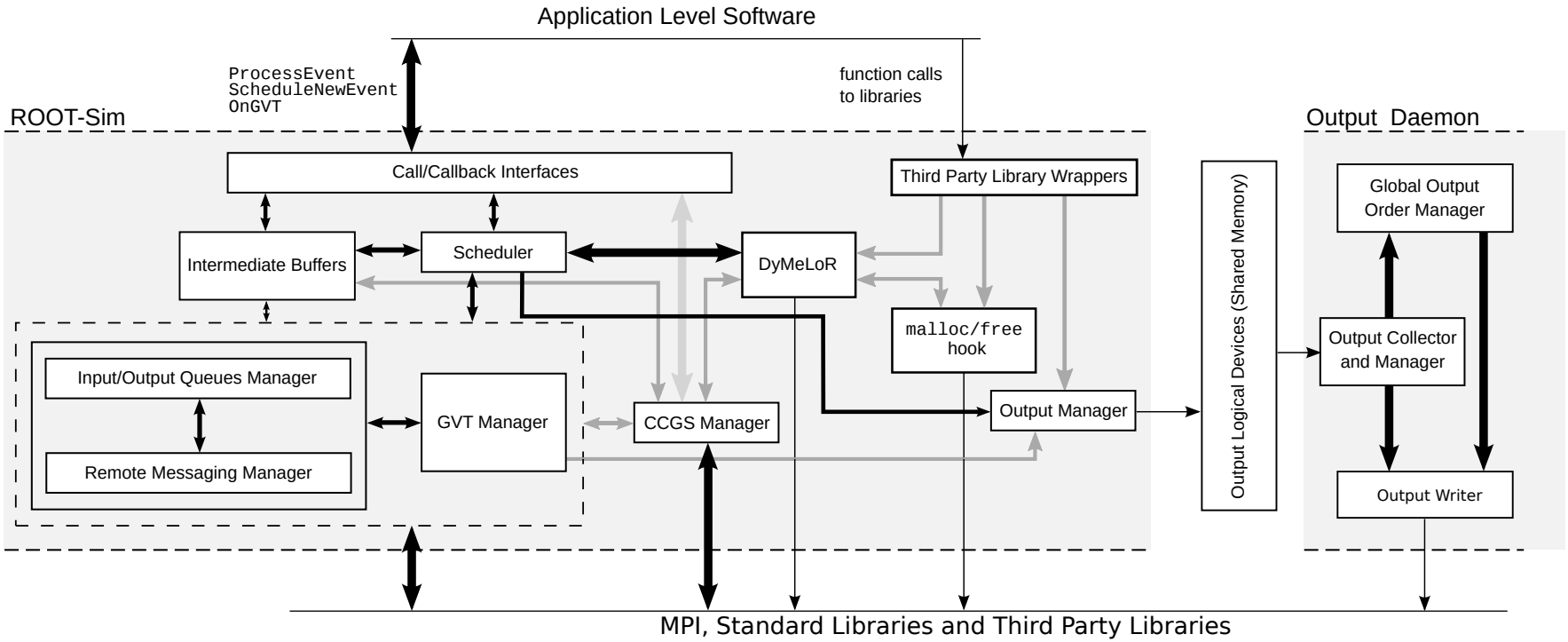
General Overview

- We target simulation-dedicated distributed environments relying on multicore CPUs
- We base our solution on an *output daemon*
  - a user-space process separated from the actual simulation framework
  - It is *not* given a dedicated processing unit
- Communication with kernel instances is achieved via a *logical device*
  - A (per-kernel) non-blocking shared memory buffer, accessed circularly
  - If it gets filled, a new (double-sized) buffer gets chained
  - Once empty, the older buffer gets destroyed

General Overview (2)



General Overview (3)



Output Generation

- We rely on linking-time redirection of primitives:
  - Linker-script directives to redirect calls to *output manager* facilities
  - Produced output buffers are not immediately materialized, rather are written on the logical device
- This solution is suitable for most machines and most of the available compilers
- Performing an output activity is logically considered as the generation of an *output message*
- Logical device messages are marked with a header specifying the nature of the content (an output message is marked as `OUTPUT`)

Non-blocking logical device

- A logical-device is a per-kernel channel
- The device is written by one kernel, and is read by one output daemon: we can implement a non-blocking access algorithm

Non-blocking logical device

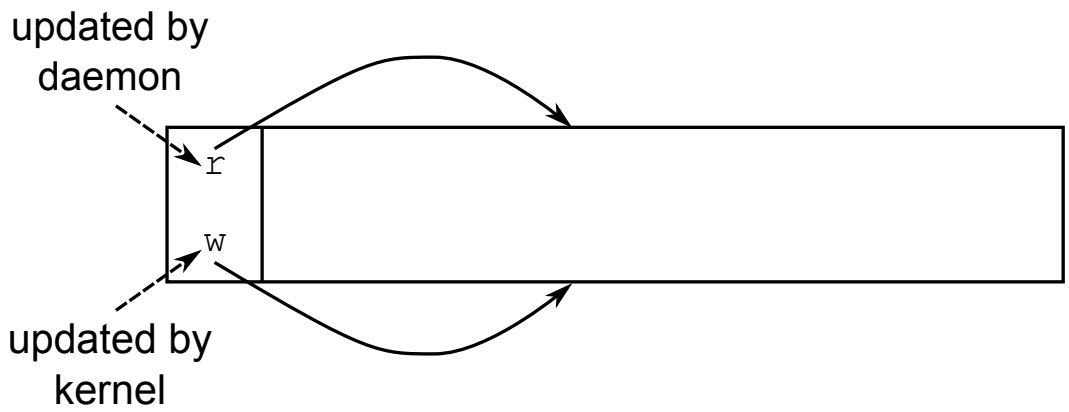
- A logical-device is a per-kernel channel
- The device is written by one kernel, and is read by one output daemon: we can implement a non-blocking access algorithm

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

Non-blocking logical device

- A logical-device is a per-kernel channel
- The device is written by one kernel, and is read by one output daemon: we can implement a non-blocking access algorithm

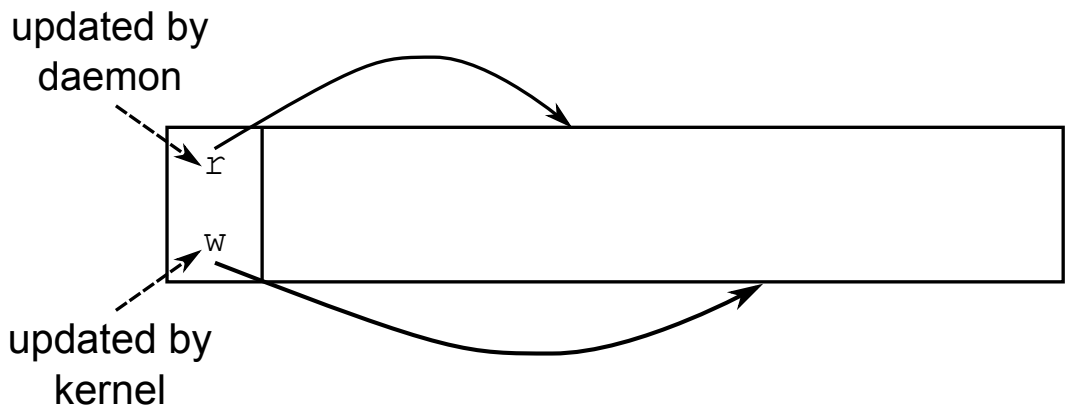
```
struct logical_device_t {
   size_t size;
   unsigned int written;
   unsigned int read;
   unsigned char buffer[];
}
```



Non-blocking logical device

- A logical-device is a per-kernel channel
- The device is written by one kernel, and is read by one output daemon: we can implement a non-blocking access algorithm

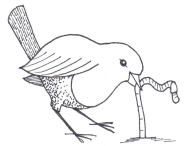
```
struct logical_device_t {
   size_t size;
   unsigned int written;
   unsigned int read;
   unsigned char buffer[];
}
```



Output Message Ordering

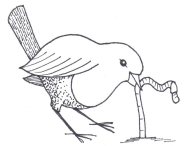
- An output message read from a device is stored into a *Calendar Queue*
  - This gives fast $O(1)$ access for output message insertion and materialization
- Messages inserted in the calendar queue are timestamp-ordered
- Depending on the actual configuration:
  - If the simulation runs on a single machine, this is enough to get system-wide ordering
  - If the simulation is distributed, the daemon forwards the locally-ordered messages to other remote daemon instances

It's the early worm that is caught by the bird!



- We're running distributed: sending messages to other instances and then rollbacking would be too much costly
- We thus don't want to forward messages to other daemons before they are committed
- Rollbacks *must* be processed locally!

It's the early worm that is caught by the bird!



- We're running distributed: sending messages to other instances and then rollbacking would be too much costly
- We thus don't want to forward messages to other daemons before they are committed
- Rollbacks *must* be processed locally!

- Upon computation of the GVT, the output subsystem is notified about the newly computed value
- This information is placed on the logical device, in a special message marked as `COMMIT`
- The Calendar Queue is then queried to retrieve messages falling before that value

Output Message Rollback

- Uncommitted output messages are stored in the local machine only
- When executing a rollback, a `ROLLBACK` message is written to the logical device, piggybacking:
  - a [*from*, *to*] interval
  - the involved *LP*
  - an *era*, a monotonic counter updated by every simulation kernel upon the execution of a rollback operation on a per-LP basis
- Calendar Queue's buckets are augmented with a Bloom filter, storing eras of messages contained
- *from* and *to* are mapped to buckets
- A linear search is performed in between the two buckets, checking only the ones which are expected to contain an element by the Bloom filter

Output Daemon Wakeup

- All this might require much computing time
  - We want a timely materialization!
  - Yet we want an efficient simulation!

Output Daemon Wakeup

- All this might require much computing time
  - We want a timely materialization!
  - Yet we want an efficient simulation!

- Processing time of each type of message: t_o, t_c, t_r
- Mean events' number written to device in a GVT phase: $\bar{c}_o$, $\bar{c}_c$, $\bar{c}_r$
- Expected execution time to empty the logical device:

$$\mathbb{E}(T) = \sum_{x \in (o,c,r)} \bar{t}_x \cdot \bar{c}_x$$

- If larger than a compile-time threshold (smaller than GVT interval), it is forced to that value
- Final value is divided into several time slices and a sleep time is computed, so to create an activation/deactivation pattern

Final Glance at Output Subsystem APIs

`commit_time(GVT)` : the simulation kernel notifies the newly computed GVT value

`set_LVT(LP, timestamp)`: the simulation kernel's scheduler notifies the identity of the dispatched LP, and the timestamp of the dispatched event

`rollback(from, to, LP)`: the simulation kernel's scheduler notifies the reception of a straggler message or an anti-message

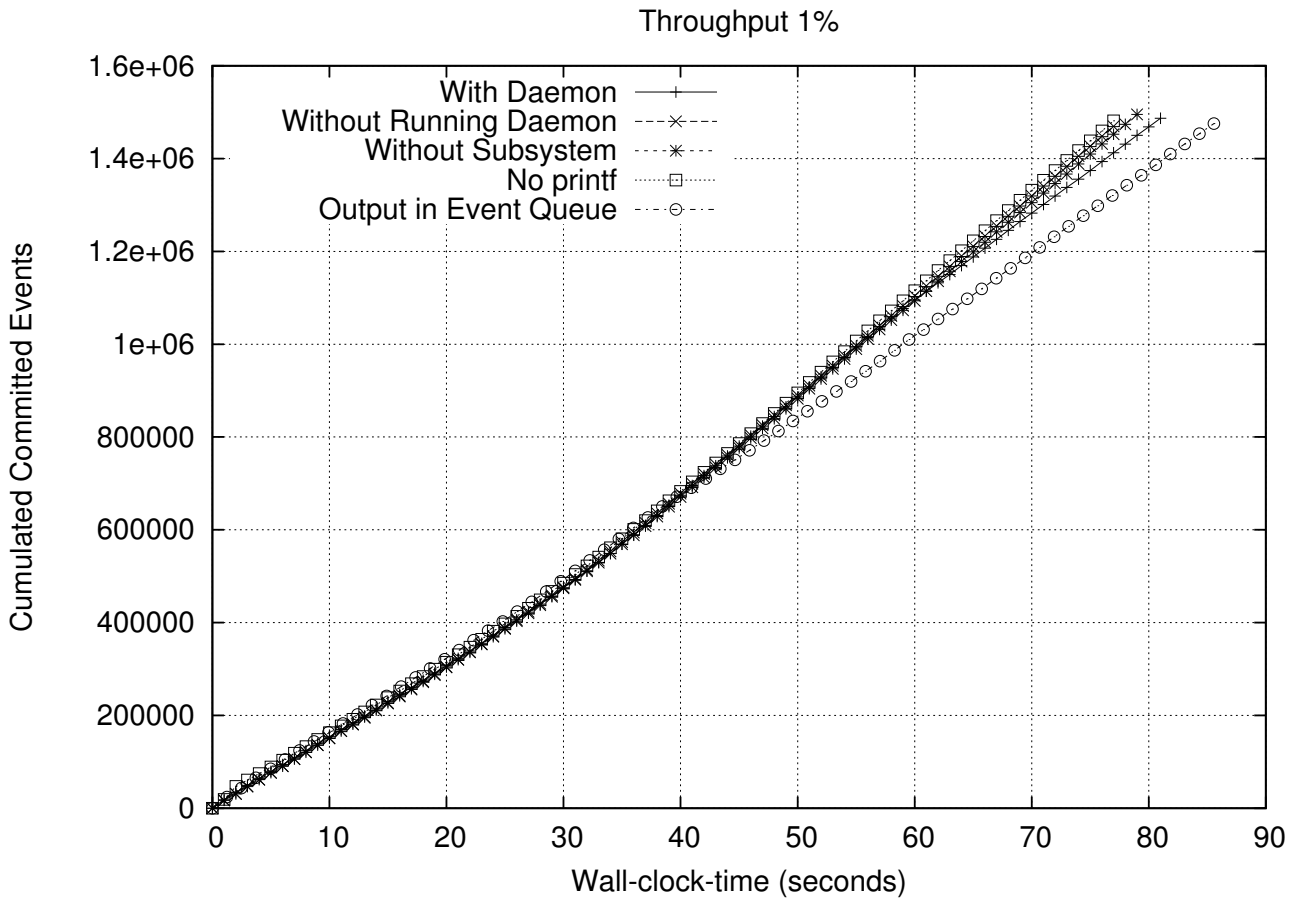
`out_msg(LP, stamp, msg, stream)`: used by the kernel to transfer an output message to the output subsystem

`autocommit(flag)`: If `flag` is true, every output message received is considered as non-rollbackable, in order to support the integration with conservative simulation engines.

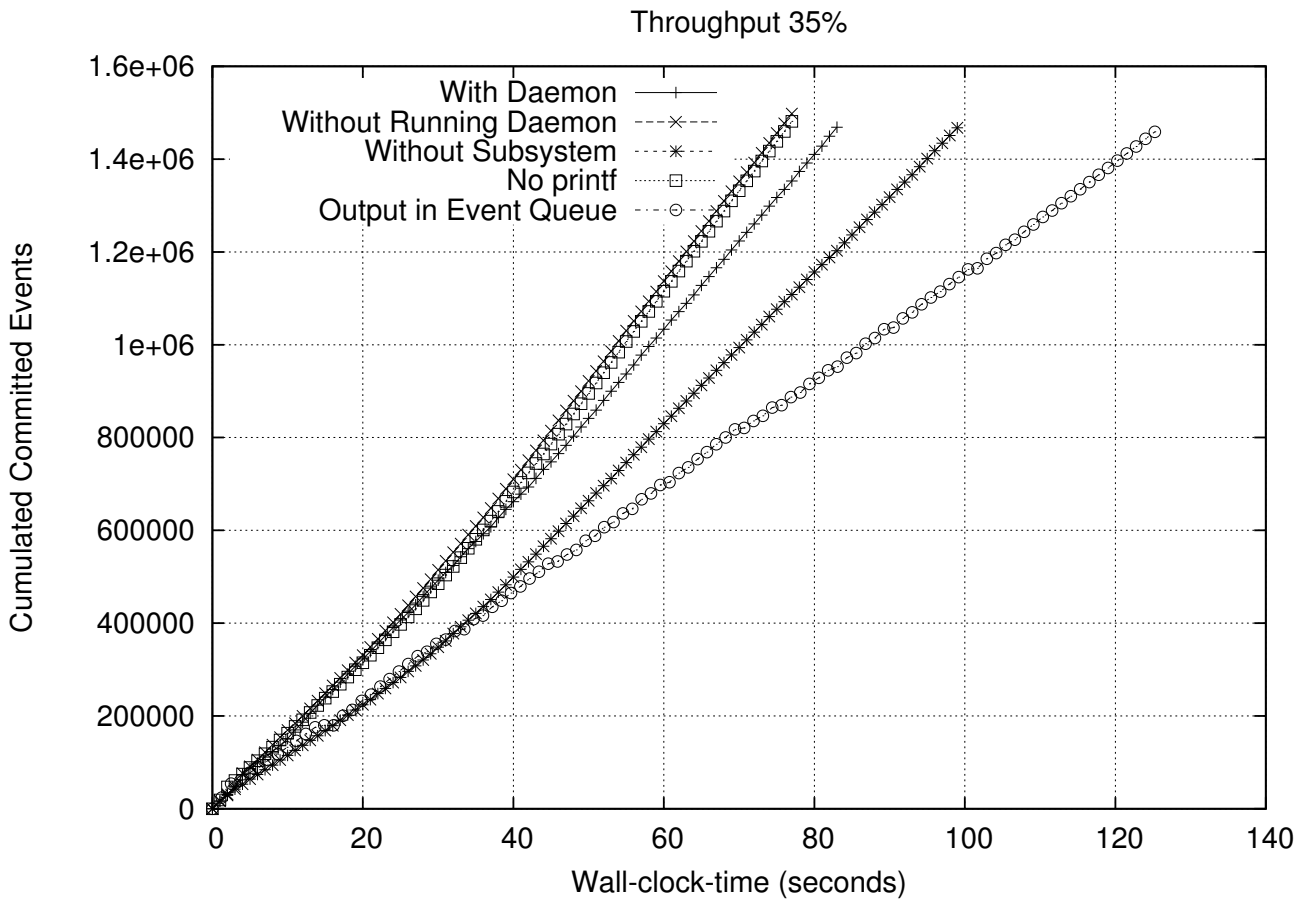
Test-Bed Scenario and Settings

- *Personal Communication System* (PCS) benchmark
  - 1024 wireless cells, each one having 1000 channels
  - simulation statistics printed periodically, with frequency $f \in [1\%, 35\%]$ of total events
  - that's one output message produced $[200, 7000]$ times per second
- Run on an HP Proliant server:
  - 64-bits NUMA machines
  - four 2GHz AMD Opteron 6128 processors and 32GB of RAM
  - Each processor has 8 CPU-cores (for a total of 32 CPU-cores)

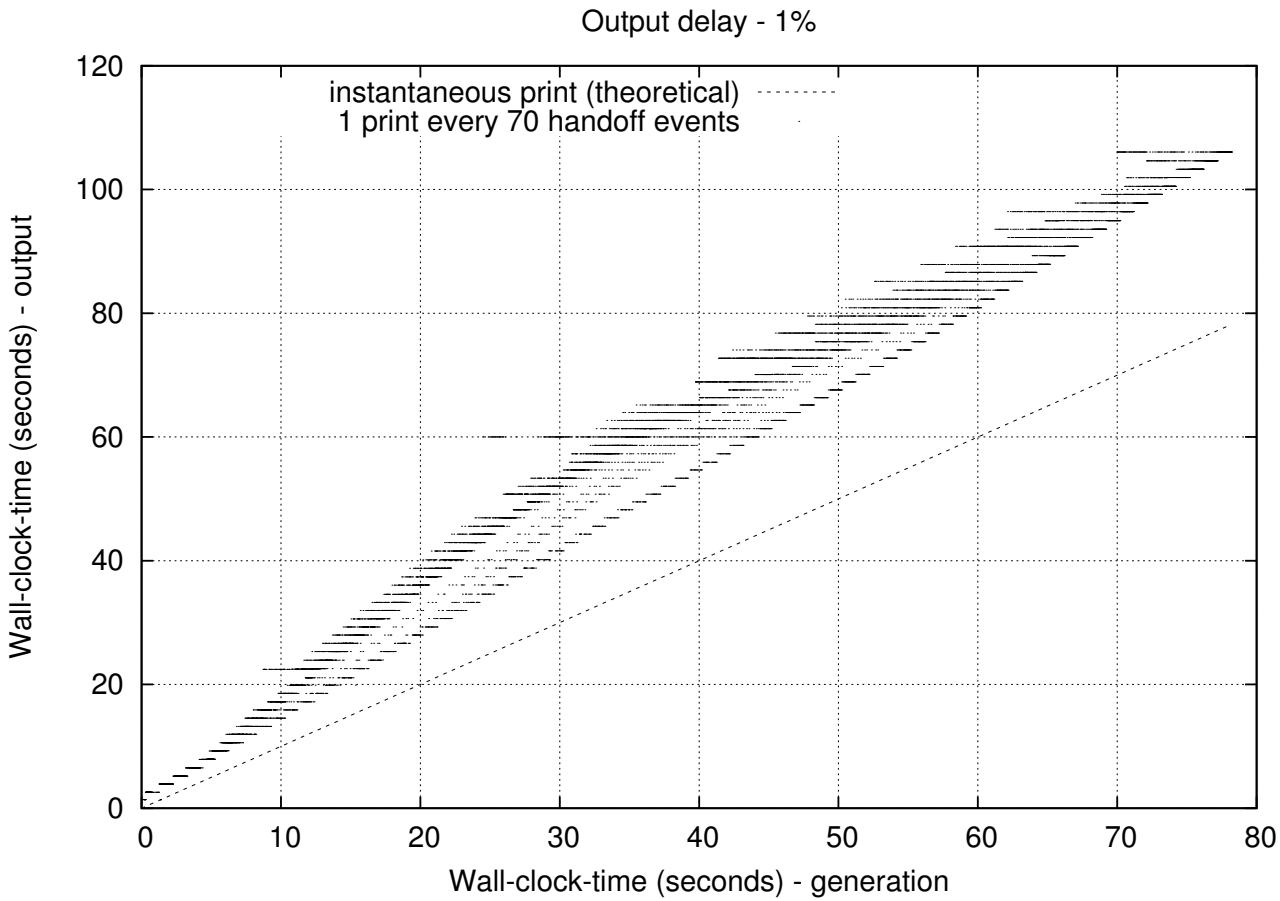
Simulation Throughput



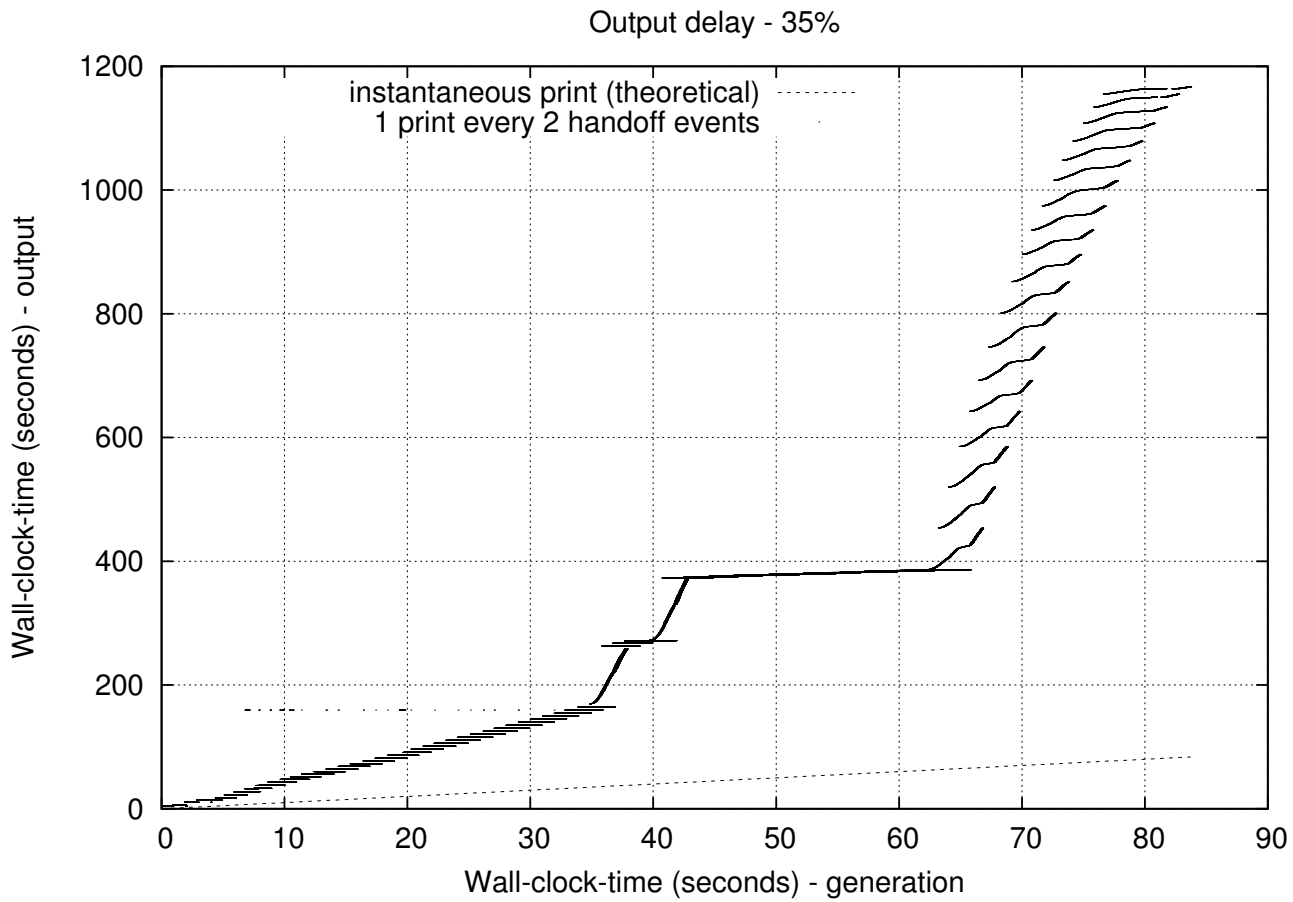
Simulation Throughput (2)



Output Materialization Delay



Output Materialization Delay (2)



Thanks for your attention

Questions?

pellegrini@dis.uniroma1.it
http://www.dis.uniroma1.it/∼pellegrini
http://www.dis.uniroma1.it/∼ROOT-Sim