The ROme OpTimistic Simulator: Core Internals and Programming Model[*]

Alessandro Pellegrini, Roberto Vitali and Francesco Quaglia
DIS, Sapienza, Università di Roma

ABSTRACT

In this article we overview the ROme OpTimistic Simulator (ROOT-Sim), an open source C/MPI-based simulation package targeted at POSIX systems, which implements a general-purpose parallel/distributed simulation environment relying on the optimistic (i.e., rollback based) synchronization paradigm. It 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 (e.g. CPU scheduling and state log/restore protocols) aimed at minimizing the run-time overhead of the platform, thus allowing for high performance and scalability. Here we overview the core internal mechanisms provided by ROOT-Sim, together with the offered APIs and the programming model that is expected to be agreed in order to produce simulation software that can be transparently run, in a concurrent fashion, on top of the ROOT-Sim layer.

1. ROOT-SIM ARCHITECTURE

The internal architecture of ROOT-Sim can be schematized as in Figure 1. At the core of the architecture, there is an event-queue manager that maintains multiple input/output queues storing incoming (or already processed) and outgoing simulation events. Each pair of input/output queues is logically associated with a same locally-hosted simulation object. The interaction between the event-queue manager and the MPI layer, in order to support event notification across different instances of the ROOT-Sim kernel, is mediated by a messaging manager which multiplexes ROOT-Sim defined message tags (e.g. `EVENT` or `ANTI_EVENT`) travelling across different ROOT-Sim instances over the same MPI channel. The scheduling sub-system gives control to the application layer along the same thread running the scheduler. Hence, simulation events (and the associated simulation objects) are dispatched for execution according to a classical time-interleaved mode, where the scheduling priority for the next-to-be-executed event across all the hosted simulation objects is based on the Lowest-Timestamp-First (LTF) algorithm. The scheduler can run in two differentiated modes. The first one is a stateless $O(n)$ mode, resembling the linux-2.4 scheduler, which queries the event-queue manager at each dispatch operation for getting information about the next-event timestamp (and hence the scheduling goodness) of all the simulation objects. The second mode (see [3]) operates in constant time (at least statistically), and is based on pre-populated meta-data that are constantly kept updated by reflecting the updates of the state input queues of the locally hosted simulation objects. It results well suited for (very) large models, for which the advantages in terms of reduced scheduling latency oversteps any overhead for scheduler-state maintenance.

[*]http://www.dis.uniroma1.it/~quaglia/software/ROOT-Sim/

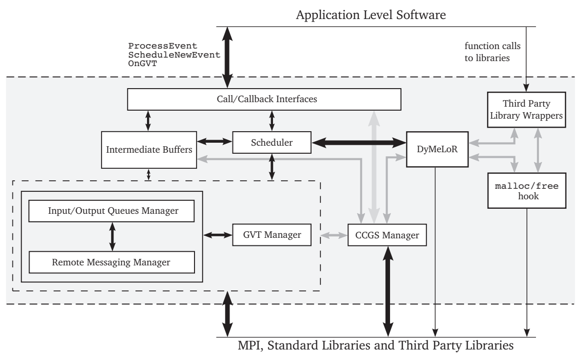

Figure 1: ROOT-Sim internals.

As for state recoverability of the simulation objects (and hence of data structures maintained at the application level), which is a crucial aspect for the design of effective optimistically synchronized environments, two main architectural approaches have been adopted. First, dynamic memory allocation and release via the standard malloc library are *hooked* by the kernel and redirected to a wrapper. Second, the simulation platform is "*context-aware*", i.e., it has an internal state which distinguishes whether the current execution flow belongs to the application-level code or the platform's internals. In the former case, the hooked calls are redirected via the wrapper to an internal Memory Map Manager (called DyMeLoR), which handles allocation/deallocation operations by maximizing memory locality for the state layout for each single simulation object, and by maintain-

SIMUTOOLS 2011, March 21-25, Barcelona, Spain
Copyright © 2011 ICST 978-1-936968-00-8
DOI 10.4108/icst.simutools.2011.245551

ing meta-data allowing the memory map to be recoverable to past values (see [4]). In addition, application level software can be compiled via an ad-hoc light instrumentation tool (see [2]) so to transparently provide the Memory Map Manager with the ability to track at runtime what memory areas are modified. This facility is offered on IA-32/x86-64 architectures, and targets the standard ELF object file format. Through this feature, ROOT-Sim is able to provide both incremental and full checkpointing capabilities. Such an instrumentation tool has been recently expanded (see [5]) in order to generate an executable version of the application layer based on a dual-coding approach, where two different versions of the application modules co-exist, one instrumented and one not. Thanks to the dual-coding approach, the ROOT-Sim kernel has been expanded so to embed logics that transparently switch to the best suited checkpointing mode, also minimizing the overhead while executing application modules.

Transparent incremental vs non-incremental checkpointing and state restore capabilities are guaranteed also in case the application interacts with third-party libraries. Even though libraries are not instrumented, they are handled via compile/link-time parsing/hooking techniques so that allocation/deallocation of memory belonging to the state layout of the simulation object, occurring within a library, is again redirected to the Memory Map Manager. Also, updates occurring within the state buffers due to the execution of a library function are captured through the call parameters via the wrapping technique.

At current date, although global variables are allowed to be used in the application-level code, in case of a rollback operation their value will not be restored. This is true also for global variables belonging to statefull third party libraries. This is related to that the original design of ROOT-Sim has been tailored to full compliance with the Time Warp paradigm, according to which the global simulation state is partitioned into disjoint object states. Hence, state sharing approaches, representative of the employment of global variables or statefull libraries will be the object of future extensions. Overall, ROOT-Sim complies to a model where a simulation object is a (dynamically allocated) set of data structures updated by subsequent calls to an event handler.

Concerning GVT calculation ([1]), ROOT-Sim relies on an optimized asynchronous approach based on a message acknowledgment scheme to solve the well-known transient message problem. Within this scheme, each kernel instance keeps track of all the messages sent to the other instances in an aggregate manner (i.e. via counters). Also, to reduce the communication overhead, each instance sends cumulative acknowledgment messages according to a window-based approach. Finally, to overcome the simultaneous reporting problem, each kernel instance temporarily stops sending acknowledgment messages during the execution of the GVT protocol.

ROOT-Sim also supports a very peculiar service that, once a new GVT value is available, transparently rebuilds a Committed and Consistent Global Snapshot (CCGS), formed by a collection of individual simulation object states (see [1]). This occurs via update operations applied to local committed checkpoints of individual simulation objects so to eliminate mutual dependencies among the final-achieved state values. The checkpoint update operation is completely transparent since ROOT-Sim realigns the logged state images by triggering the execution of event handlers natively present within the application code, by passing in input already committed events not yet discarded by memory recovery procedures. Once the CCGS is built, each simulation object gains control via an ad-hoc callback within the APIs, by also having access to the copy of its state image belonging to the CCGS. Such a service can support, e.g., termination detection schemes based on global predicates evaluated on a committed and consistent global snapshot.

[1]GVT - Global Virtual Time - represents the commitment horizon. No causality violation can even occur for optimistically processed events whose timestamp falls before the current GVT value. GVT updates typically trigger memory recovery procedures, e.g. of obsolete checkpoints.

1.1 Exposed APIs

Beyond support functions, e.g. for accessing recoverable random number generators, ROOT-Sim supports the following APIs: (A) `int ProcessEvent(int me, time_type now, int event_type, void *content, int size, void *state)` - a callback to be implemented within the application layer, which provides control to the application for the actual processing of simulation events. `me` is the identifier of the simulation object being dispatched, `now` is the current value for the local clock, `event_type` is the numerical code for the event to be processed, `content` is the buffer maintaining the event payload (made of `size` bytes), and `state` is the pointer to the top data structure forming the simulation object state layout. (B) `int ScheduleNewEvent(int where, time_type timestamp, int event_type, void *content, int size)` - this function allows injecting a new simulation event within the system, to be destined to whichever simulation object identified via `where` (the other parameters have the above described meaning). (C) `int onGVT(int me, void *snapshot)` - this callback passes control to the application by providing the simulation object snapshot belonging to the CCGS.

2. CODE SNIPPETS AND PROFILING

In this section we present code snippets for the implementation of a Personal Communication System (PCS) simulation model, which we used as test-bed in several studies.

```
 1 #include <ROOT-Sim.h>
 2 ...
 3 typedef struct _state_type{
 4     int free_channels;
 5     int incoming_calls;
 6     int completed_calls;
 7     int blocked_incoming_calls;
 8     int blocked_handoff_calls;
 9     int out_handoffs;
10     int in_handoffs;
11     unsigned long event_counter;
12     unsigned int channel_state[CHANNELS_PER_CELL / 32 + 1];
13     struct _channel *active_channels_list;
14     ...
15 } state_type;
```

By the above snippet, the definition of the data structure representing basic information about the state of each PCS cell, modeled by means of a separate simulation object, is based on a transparently rollbackable dynamic-memory based list keeping information (e.g., power setup) related to radio channels actively involved in conversation (see line 13).

```
16 void ProcessEvent(int me, time_type now, int event_type,
17         void *content, void *state) {
18  ...
19  switch(event_type) {
20    case INIT:
21      state = (state_type*)malloc(sizeof(state_type));
22      SetState(state);
23      state−>free_channels = CHANNELS_PER_CELL;
24      state−>active_channels_list = NULL;
25      state−>in_calls = 0;
26      state−>completed_calls = 0;
27      state−>out_handoffs = 0;
28      state−>blocked_in_calls = 0;
29      state−>blocked_handoff_calls = 0;
30      state−>in_handoffs = 0;
31      state−>event_counter=0;
32      for (w = 0; w < CHANNELS_PER_CELL / 32; w++)
33              state−>channel_state[w] = 0;
34      ...
35      TimeStamp = (time_type) (10 ∗ rand()/RAND_MAX);
36      ScheduleNewEvent(me,TimeStamp,START_CALL,NULL,0);
37
38      break;
39     case START_CALL:
40              ...
41     case END_CALL:
42              state−>event_counter++;
43              state−>free_channels++;
44              state−>completed_calls++;
45              deallocation(state, event_content−>channel);
46              break;
47     case HANDOFF_LEAVE:
48              ...
49     case HANDOFF_RECV:
50          ...
51     }
52  }
```

The above portion of the logic shows the handling of a special type of event called `INIT`, which is automatically scheduled by the ROOT-Sim environment once for each simulation object at simulation startup. It is useful for performing initialization operations, which in this code example correspond to the allocation, via a traditional `malloc()` call, of the main data structure implementing the simulation object state (see line 21), and to the setup of its fields. One interesting point relates to line 22, where we list a statement that exploits the ROOT-Sim support function `SetState()`. It is used to explicitly notify the simulation kernel that the next event handler activation for this same simulation object wants to receive the memory address of its own freshly allocated data structure as input value for locating itself in memory. Then, a `START_CALL` event for an initial incoming call is scheduled for the current simulation object around the initial simulation time. As for the code fragment related to `END_CALL` events, here we update basic statistics and then call a function de-allocating the record keeping track of the ongoing call (on a specific channel), which also updates finer grain statistics related to the power regulation model.

```
53  int onGVT(int me, void ∗snapshot) {
54   if (snapshot−>completed_calls < COMPLETE_CALLS)} return 0;
55   ...
56   return 1;
57  }
```

The above snippet shows how a simulation object can get control by accessing its most recent committed snapshot (belonging to the globally consistent and committed state image), and can evaluate a predicate based on the number of completed calls for replying to the simulation kernel with a flag value indicating whether (according to what computed by this simulation object) the simulation run can be stopped.

We finally conclude this paper by reporting run-time results for a concise view of the efficiency of the ROOT-Sim environment. In particular, we report the Wall-Clock-Time for simulating half an hour of operativity of the PCS for the case of a coverage area of 256 and 1024 cells, evenly distributed on the 24 cores of an HP DL385 G7 Server. We have run the simulations by setting to 1000 the number of radio channels per cell, to 120 sec the mean call duration, and by varying the mean residence time of mobiles within a cell in between 600 sec and the extremely reduced value of 150 sec (which would tend to model macro vs micro cells. Decreasing the residence time, handoff events become increasingly relevant, so that cross-cell events become increasingly more frequent (compared to cell-internal events). In other words, we increase the level of coupling of the cells, thus forcing more tight constrainment while advancing in simulation time. Rollbacks tend to become more frequent (although shorter), and the various sub-systems supporting the optimistic synchronization paradigm (e.g. the state restore sub-system) tend to become subject to increased (or varied) load. We have set the call inter-arrival time in order to provide channel utilization factor of about 50%, which gives rise to small to medium event granularity and non-minimal state size, thus making a good scenario for the evaluation of the underlying platform overheads. Figure 2 shows the results, which indicate how the variation of the Wall-Clock-Time is bounded by about 25%, which indicates good performance stability vs variations of the execution pattern for the parallel run. Also, although not shown, speedup over the corresponding sequential runs ranges from 7 to 12.

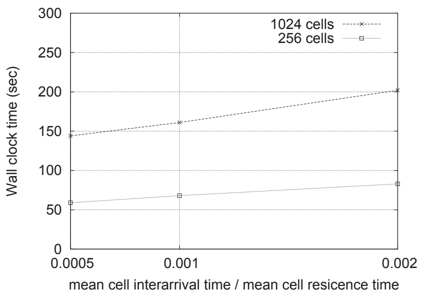

Figure 2: Run-time profiling.

3. REFERENCES

[1] D. Cucuzzo, S. D'Alessio, F. Quaglia, and P. Romano. A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation. In *Proceedings of the 11th International Symposium on Distributed Simulation and Real-Time Applications*, pages 227–234, Chania, Crete Island, 2007. IEEE Computer Society.

[2] A. Pellegrini, R. Vitali, and F. Quaglia. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 45–53, Lake Placid, New York, USA, 2009. IEEE Computer Society.

[3] T. Santoro and F. Quaglia. A low-overhead constant-time ltf scheduler for optimistic simulation systems. In *Proceedings of the 15th Symposium on Computers and Communications (ISCC)*, page 948–953, Riccione, Italy, 2010. IEEE Computer Society.

[4] R. Toccaceli and F. Quaglia. DyMeLoR: Dynamic memory logger and restorer library for optimistic simulation objects with generic memory layout. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 163–172, Rome, Italy, 2008. IEEE Computer Society.

[5] R. Vitali, A. Pellegrini, and F. Quaglia. Autonomic log/restore for advanced optimistic simulation systems. In *Proceedings of the 18th International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, pages 319–327, Miami, Florida, USA, 2010. IEEE Computer Society.