Optimizing Memory Management for Optimistic Simulation with Reinforcement Learning

Alessandro Pellegrini
DIAG–Sapienza, University of Rome
pellegrini@dis.uniroma1.it

***Abstract*—Simulation is a powerful technique to explore complex scenarios and analyze systems related to a wide range of disciplines. To allow for an efficient exploitation of the available computing power, speculative Time Warp-based Parallel Discrete Event Simulation is universally recognized as a viable solution. In this context, the rollback operation is a fundamental building block to support a correct execution even when causality inconsistencies are a posteriori materialized. If this operation is supported via checkpoint/restore strategies, memory management plays a fundamental role to ensure high performance of the simulation run. With few exceptions, adaptive protocols targeting memory management for Time Warp-based simulations have been mostly based on a pre-defined analytic models of the system, expressed as a closed-form functions that map system's state to control parameters. The underlying assumption is that the model itself is optimal.**

In this paper, we present an approach that exploits reinforcement learning techniques. Rather than assuming an optimal control strategy, we seek to find the optimal strategy through parameter exploration. A value function that captures the history of system feedback is used, and no a-priori knowledge of the system is required. An experimental assessment of the viability of our proposal is also provided for a mobile cellular system simulation.

I. Introduction

A traditional way to achieve high performance simulations is the employment of discrete parallelization techniques [1]. They are based on the partitioning of the simulation model into N objects, called Logical Processes (LP), that can execute events in parallel on different CPUs and/or different cores. LPs are associated with a simulation state S_i, and rely on synchronization mechanisms to achieve causally-consistent execution of simulation events at each LP. The operations that are associated with events happen instantly in logical simulation time, and have an impulsive duration.

The Time Warp optimistic synchronization protocol presented in [2] is based on the rollback of already-executed events, to recover possible timestamp-order violations due to the absence of block-until-safe policies for event processing. As it is well recognized, this protocol is likely to favor the speedup in general application contexts. In particular, it has been shown to exhibit a performance which is relatively independent of both the specific simulation model and the message delivery latency (even when non-minimal).

In this context, the design and development process of optimized techniques supporting state recoverability is a major obstacle for the construction of efficient optimistic simulation systems. When complete transparency towards the application layer is pursued, this process is even harder. In particular, in order to reduce the cost associated with log management (both in terms of memory usage and latency to take possibly unnecessary logs), an important aspect is how often a checkpoint is taken—a concept related to the checkpointing interval χ [3]. The best-suited time instant when this operation should be executed strongly depends on the runtime dynamics of the simulation model, and must take into account both the cost associated with the operation itself, and the probability that the taken state snapshot will be useful to restore a previous simulation state due to the occurrence of a rollback.

Several works have addressed recoverability issues via the implementation of hardware/software architectures offering simulation states' log/restore facilities (e.g., [4], [5], [6], [7]), each providing some specific transparency level, and/or the use of models aimed at identifying the best suited tuning of the parameters associated with the selected log/restore policy, in order to optimize performance (e.g., [8], [9], [10]).

An additional aspect is related to the way a simulation state snapshot is taken. In particular, to reduce the amount of data copied into a state snapshot, incremental state saving techniques can be employed. According to this scheme, the simulation engine is able to detect at runtime what are the portions of the simulation state which have been modified by the execution of a set of events, and therefore it is able to pack into the snapshot only the necessary (modified) data. Of course, due to this memory update tracking, in case the system offers full transparency to the model developer, an additional overhead is paid. This overhead could not be paid off, e.g. in scenarios where a very large portion of the simulation state is updated by one or multiple events.

To the best of our knowledge, no one has studied the feasibility of Reinforcement Learning (RL) techniques [11] targeting recoverability in optimistic simulation systems yet. Although the works in [12], [13] have explored the possibility of exploiting RL in Time Warp simulations, there is no explicit focus on state recoverability.

In this paper, we present a log/restore architecture designed according to a Q-Learning schema, which addresses performance and transparency issues by:

1) Transparently enabling the adoption of incremental and non-incremental log/restore of system's state, in an interleaved fashion.
2) Dynamically switching to the operating mode (incremental vs. non-incremental) that likely provides the best

performance, despite plausible variations of the runtime dynamics, adding only a little overhead to the execution of the system. This entails tuning the value of the checkpointing interval χ, although this is not done explicitly.

3) Allowing the programmer to use standard constructs for dynamic memory allocation/deallocation operations, thus allowing the state of a simulation object to be scattered across non-contiguous memory chunks.

Our RL-based log/restore architecture builds on our previous results in [14], providing a comprehensive solution for memory management in optimistic simulation. This solution allows to scatter simulation states on dynamically allocated buffers, and to take full and incremental state snapshots. The work in [14] offers as well an analytic model to switch at runtime between incremental and full checkpointing schemes, yet this is replaced in this work by our RL-based approach.

The Q-Learning log/restore layer has been integrated into the ROme OpTimistic Simulator (ROOT-Sim)[1], an open source general-purpose simulation platform based on the Optimistic Synchronization approach, thus making it available within an operating environment. Also, we report experimental results for an assessment of the viability and efficiency of our proposal for a case study related to GSM coverage along two different wireless cell configurations.

The remainder of this paper is structured as follows. In Section II we discuss related work. In Section III, the architecture which the RL agent is based upon is described. The agent itself is presented in Section IV. Experimental data are reported in Section V.

II. Related Work

As mentioned, most of the proposals in the literature target autonomic optimization of memory management subsystems in the context of Time Warp-based simulation leveraging closed-form models to fine tune the runtime behaviour of simulation engines. In particular, in [3] the optimal checkpointing interval χ^{OPT} is selected by relying on an analytic model based on LP execution time, by assuming that the execution of events is non-preemptive, and by assuming that the rollback length is independent of each other. Differently, in [10] the total number of rollback operations executed within a certain wall-clock time interval, and the number of executed (both committed and uncommitted) events are exploited to derive an optimal checkpointing interval χ^{OPT}.

The latter scheme does not take into account the fact that the execution time of different classes of events can vary. This aspect is captured in [15], where the Event Sensitive State Saving (ESSS) is proposed. This technique emphasizes that it is convenient to take a state snapshot when the granularity of the next event increases. In [16], the LP's event history is taken into account, considering the variations between the timestamp of two consecutive events, to determine which is the best moment for taking a snapshot.

[1]Source code available at https://github.com/HPDCS/ROOT-Sim.

The works in [17], [18] base the selection of the best-suited χ value on heuristic algorithms, the latter explicitly taking into account the usefulness of a checkpoint with respect to the probability that it will be used for a restore operation.

We keep the ability of all the above works, allowing the system to determine an optimal checkpointing interval χ. Nevertheless, we do not ground our decision on a closed-form model, thus allowing the runtime to take more flexible decisions, and possibly allowing it to capture secondary effects (not explicitly modeled by the above proposals) which could be relevant for the selection of an optimal strategy. Moreover, all the above results do not allow to switch between an incremental vs non-incremental checkpointing mode.

This latter aspect is covered by the work in [14] which presents the design of an actual architecture to support dynamic switching among the two execution modes, as well in the presence of dynamically-allocated memory. While we borrow the architectural organization from [14], this work still bases all the dynamic optimization on closed-form models.

A work which is close in spirit with our proposal is presented in [19]. The authors propose a dynamic selection between incremental vs non-incremental execution modes, along with the selection of a proper checkpointing interval χ, by relying on a genetic algorithm. Due to the nature of this kind of algorithms, the solution space that can be explored by the proposal is sparse and limited, since the genes allow to select only a limited amount of values of χ, thus possibly leading to suboptimal choices.

Explicit machine-learning runtime optimization using reiforcement learning has been used in the context of Time Warp-based simulation in [12], [13]. Nevertheless, the authors explicitly address the optimism window and the global virtual time (GVT) recomputation interval respectively, which are optimization parameters orthogonal to the goal of the present work.

III. Reference Architecture

We implement our autonomic optimization strategy within ROOT-Sim, an open source general purpose platform developed using `C` technology, which is based on a multi-threaded simulation-kernel layer. The platform transparently supports all the mechanisms associated with parallelization (e.g., mapping of simulation objects on different threads) and optimistic processing.

ROOT-Sim's memory management subsystem (called DyMeLoR [14]—Dynamic Memory Logger and Restorer) can be seen as wrapper of `malloc` and `free` `ANSI-C` standard services, interposed at linking time between the simulation model's software and the standard `malloc` library. This approach allows the application programmer to use dynamically-allocated memory within the simulation software in a way transparent to the lower-level memory management tasks, such as log/restore operations, actuated by the ROOT-Sim kernel.

DyMeLoR is based on the concepts of memory preallocation and memory partition. Each LP's simulation state is coupled with a metadata table which is used to manage a

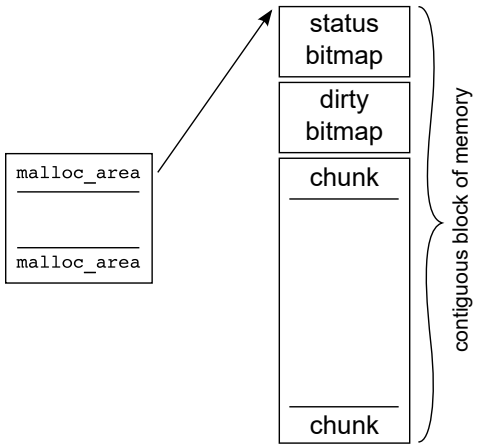

Fig. 1. Di-DyMeLoR Data Structures

block of contiguous memory chunks. The chunks within a block have the same size, while different blocks host chunks with size corresponding to different powers of 2. Each entry of the metadata table (which is called *malloc area*) holds a pointer to an (eventually allocated) block of chunks, used to serve memory requests from the simulation model software. For both time and space efficiency, each chunk within a block is associated with a single bit that indicates its current status, namely used to serve a request or available. This solution avoids the usage of a costly per-chunk header. In addition, the bitmap of status bits (called the *status bitmap*) is placed at the top of the pre-allocated block of chunks, therefore it can be managed directly only in case that a chunk's status changes. A second bitmap, called the *dirty bitmap*, keeps information about what chunks have been involved in a memory write operation, since the last checkpoint was taken (those chunks are referred to as *dirty chunks*). The overall organization of DyMeLoR's data structures is shown in Figure 1.

A log operation is simply performed by analyzing the status/dirty bitmaps. In case of full logs, all the in-use chunks are packed into a contiguous log buffer appropriately sized. Instead, for an incremental log, only those chunks that have been dirtied by event execution are packed. Similarly, a restore operation is performed by iteratively backward traversing the log chain, searching for logged chunks that have not been restored yet, until a full log is found (i.e., until all the *malloc areas* are restored).

To dynamically detect at runtime what are the chunks which are involved in a memory update, DyMeLoR relies on Hijacker [20], an open-source static binary instrumentation tool which allows, via the specification of a set of xml rules, to manipulate an executable object before the final linking stage. In particular, we use Hijacker to intercept any memory-update instruction, compute its final address target and its size, and call an ad-hoc routine which checks whether this update operation belongs to any *malloc area*. In the positive case, the corresponding bits in the dirty bitmap are set before the actual memory-update operation takes place.

To minimize the runtime overhead, specifically when non-incremental checkpoints should be taken, we revert to the non-instrumented version of the original simulation model relying on the *multiversion binary* feature offered by Hijacker. This feature allows multiple versions of the original simulation model to coexist in the same binary image, explicitly sharing the same data sections. Therefore, since two versions of the same model's code exist in the same binary image (with different names), ROOT-Sim can activate either version by simply invoking the instrumented vs non-instrumented simulation event handler at runtime. This solution allows for a fast switch between the two versions with zero overhead. Deciding whether to call an instrumented or a non-instrumented handler is the orthogonal problem which we explicitly study in this paper.

IV. Reinforcement Learning agent

As already mentioned, the work in [14]—which we borrow the architectural organization from—adopts a closed-form model to determine which of the aforementioned schemes (namely, incremental vs non-incremental) should be used to autonomically maximize overall simulation performance, taking into account stability as well. We present here an orthogonal approach, where a RL agent is used to determine what execution scheme should be used.

A. Overview of Reinforcement Learning

In reinforcement learning [11], agents learn the best-suited action in a given scenario by try and error, taking account of the amount of rewards received. The environment is uncertain and it can be changed probabilistically by the actions of agents. Further, the reward is probabilistic and is dependent on the environment change. RL has deep roots in Markov Decision Process (MDP) [21]. The Markov Decision Problem is the problem of optimizing an MDP. Details on common MDP solutions can be found in [21], [11], [22], [23].

A stochastic process is called a Markov chain if it has the Markov property, namely, the state transition of the process is dependent only on the current state, regardless of the process history. In a Markov decision process, for each state there is a set of possible actions that can be taken by an external decision maker. The process still possesses the Markov property, but the state transition is also dependent on the actions. In fact, each action defines a state transition matrix. In addition, there is reward matrix for each action.

A MDP is completely specified by the following [22]:

- Transition probability matrices (TPM). These are the one-step state transition probability matrices. For each action, there is a corresponding matrix.
- Transition reward matrices (TRM). Like the TPMs, there is also one matrix per action. The element of the matrix is the reward received when taking a certain action in a given state.
- Objective function (performance metric). This function provides a quantitative measurement of the performance of a policy.

As mentioned above, at each decision epoch of a MDP an action is taken and a reward is received as a result of the action. The sequence of rewards received after decision epoch t is $r_{t+1}, r_{t+2}, \ldots$. In a Markov decision problem, the goal is to maximize the expected value of some function of the reward sequence. This function is known as the *return function*, or simply the *return*. Here we shall focus on the *discounted reward function*, which is a function of the following *discounted return*:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$$

where γ is a number, $0 \leq \gamma \leq 1$, called the *discount rate*. The purpose of the discount rate is twofold. First, it gives more weight to recent rewards than to future rewards in the determination of the current return. Second, it makes it possible to have a single definition of return for both episodic tasks, where the task breaks naturally into subsequences with a final state (such as playing chess), and continuing tasks with a long life span.

The Transition Probability (TP) and the Transition Reward (TR) are collectively known as the *model* of a Markov decision problem. In [22] it is stated that when the model is known, a MDP can be solved with Dynamic Programming. Insted, in our solution, RL techniques are adopted, which were created to solve the Markov decision problems when the model is unknown by using estimates of the TP and TR.

A RL system is defined as a five-tuple: $\{S, A, \pi, RF, VF\}$, where:

- S is a set of states of the environment,
- A is a set of actions that the agent can take,
- $\pi : S \rightarrow A$ is a mapping from a state of the environment to an action to be taken by the agent,
- $RF : S \times A \rightarrow R$ is the *Reward Function*, a mapping from a state or state-action pair of the environment to a numerical value called a reward signal, which is an indication of the desirability of the state or state-action pair.
- VF is the *Value Function* of a given policy, which defines the expected return an RL agent can receive. In RL the action-value function is most often used, and is known as the *Q-factor*. The best action to take in a state is obtained from the best *Q-factor* associated with that state.

For problems with more than one state, we need to find out which action is the best for each state in order to maximize the expected return, by using *Q-factors*. Given a policy π, the expected return would be:

$$Q^{\pi}(s,a) = E_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s, a_t = a\right\}$$
$$+E_{\pi}\left\{r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1}))\right\}$$

but, since the the model of the system is unknown, we have to estimate the expected immediate return from the sequence of rewards. This can be done by using the Robbins-Monro algorithm:

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a')] \quad (1)$$

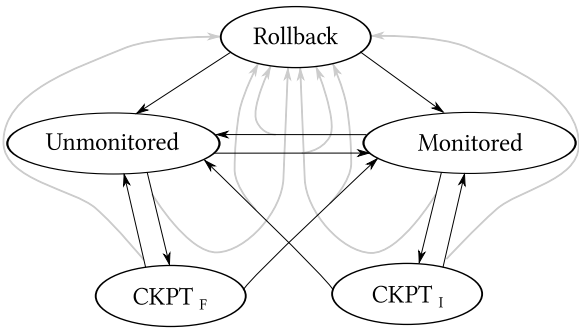

Fig. 2. State-Transition Diagram on the System

with $0 < \alpha \leq 1$ being the *learning rate*. We can see that, when updating $Q(s,a)$, instead of using $Q(s_{t+1}, \pi(s_{t+1}))$, the best *Q-factor* from the next state, $\max_{a'} Q(s_{t+1}, a')$ is used.

B. System Model and RL Agent

The checkpointing model we are presenting here, is a semi-Markov Decision Process model (SMDP). We have identified the following system states: $X = \{$Rollback, Non-Incremental, Incremental, CKPT$_F$, CKPT$_I\}$. When the system is processing events, its state is Non-Incremental or Incremental, depending on which execution mode has been selected. When the system is performing housekeeping operations, it is in any other state, depending on what kind of operation is being performed. Figure 2 shows the state transtion diagram on the checkpointing scheme, with the gray arrows representing transitions which are not under the control of the agent, but which are due to external events (i.e., the reception of a causally unordered message) and can occur whenever the agent decides to perform a certain action.

Further, we define the state space and the action space in SMDP as follows. The state space is:

$$I = \{s_1,\ s_2,\ \ldots\}$$

while the action space is:

$$A = \{a_{inc}, a_{full}, a_{ckpt}\} \qquad \forall s \in I$$

which represent the action to execute the next event in an incremental or full fashion, or the action to take a checkpoint, respectively. We note that by deciding whether to take or not a checkpoint, our RL agent is indirectly optimizing the value of χ. In fact, while traditional approaches explicitly use the runtime history to determine an optimal value of χ, we take a decision after the execution of each event. This approach can therefore respond more promptly to sudden runtime dynamics' variation with respect to traditional approaches, which should wait for the next moment at which the value of χ^{OPT} is recomputed.

No action to switch to the $Rollback$ state is present, since switching to $Rollback$ only depends on events that are external to the system. Further, one single action for performing a checkpoint has been provided. This has been done because, recalling the fact that each type of checkpointing requires

information collected during the events' execution, the sole checkpoint mode available when the decision to take the snapshot is done, is the one related to the execution mode adopted up to that moment.

The goal of the RL agent is to reduce the time spent by the kernel performing tasks that are not actual forward computation. To describe this, we define the following function, $\{X(t),\ t \geq 0\}$, which is dependent on the current system state $x \in X$, given by:

$$X(t) = \begin{cases} 0 & \text{if } x = \text{Non-Incremental} \\ \frac{\delta_M}{(\delta_e+\delta_M)} & \text{if } x = \text{Incremental} \\ 1-\gamma & \text{if } x = \text{CKPT}_I \\ 1 & \text{if } x = \text{CKPT}_F \\ 1 & \text{if } x = \text{Rollback} \end{cases} \quad (2)$$

By using this function, we can define the *expected computation loss* as a performance criterion:

$$\Gamma = E\left[\int_0^T X(t)dt\right] \quad (3)$$

where T is the time when the simulation terminates. Of course, the termination time T cannot be controlled, so the minimization of Γ is essentially equal to minimizing the *overhead ratio* presented in [24], which is defined by

$$\lim_{t\to\infty} \frac{E[\text{amount of computation loss in } [0,t)]}{t}$$

To completely define the RL agent, we need to define the following quantities:

- $Q(s,a)$: expected computation loss when action a is taken in the state s.
- $V(s)$: minimum expected computation loss until the termination of process, when the initial state is s.

The optimality equation is given by:

$$V(s_n) = \min_a\{Q(s_n,a)\} \qquad \forall a \in A$$

where the initial *Q-factors* are defined as follows, $\forall s_n \in S$:

$$Q(s_n, a_{unm}) = Pr\left(S_F\delta_{RB} + \frac{\chi-1}{2}\delta_e\right)$$

$$Q(s_n, a_{mon}) = Pr\left[S_F\delta_{RB} + \frac{\chi-1}{2}(\delta_e+\delta_M)\right]$$

$$Q(s_n, a_{ckpt}) = \begin{cases} \delta_{LB}S_F & \text{if } x = \text{Non-Incremental} \\ \delta_{LB}S_P & \text{if } x = \text{Incremental} \end{cases}$$

where:

- δ_e is the average event execution cost.
- δ_M is the cost for running the memory-update tracking module.
- S_F is the average size of a full (non-incremental) log.
- S_P is the average size of a partial (incremental) log.
- δ_{LB} is the average cost for logging a single byte belonging to the state image.
- δ_{RB} is the average cost for restoring a single byte from the log.
- Pr is the rollback probability.
- χ is the average log interval.

Those initial *Q-factors* are then exploited to select the best action to perform by the RL agent, and at the same time they are updated in order to improve the knowledge of the system.

To build our checkpointing scheme, we define the following four elements for the RL:

1) **agents**: Checkpointing Manager (CM), one per LP
2) **Environment**: System States, as presented in Figure 2
3) **Rewards**: Computation loss (negative reward)
4) **Actions**: 'Non-Incremental', 'Incremental', 'Checkpoint'

The purpose of CM is to reduce the computation loss represented by Equation 3. The checkpointing scheme based on the Q-learning is developed as follows.

Step 1: Choose an action a_t based on the *Q-factors*.

Step 2: After taking the action a_t, CM observes the state s_{t+u} in the next decision epoch[2]. Until the next decision epoch, CM records the computation loss, $X(v),\ 0 \leq v \leq u$, where $X(v)$ is given by Equation 2.

Step 3: Update the *Q-factor* by using Equation 1, where the reward r_{t+1} is given by:

$$r_{t+1} = \int_0^u X(v)dv$$

In Step 1, we have to choose an action. The ϵ-greedy method presented in [11] has been used. With this method, CM chooses an action at random with probability $0 < \epsilon < 1$. Otherwise, the action is chosen by exploiting the best *Q-factor* at the state s. With small values of ϵ, the system is more conservative, acting as if the available information are already representative of the runtime dynamics. On the other hand, if ϵ is large, the system explores more the solution space.

V. Experimental Data

To assess the validity of our RL agent, we rely on a test-bed simulation model of GSM coverage along the ring highway running around the city of Rome (named GRA—Grande Raccordo Anulare). The length of GRA is 68 km and GSM connectivity is guaranteed via 8 GSM cells, each offering up to 9 km coverage along the highway. As in the actual system organization supported by the Telecommunication Company,

[2]A decision epoch is the time between two different moments when the agent is asked to take a decision.

each cell hosts 1000 radio channels [25]. In our simulations, communication channels are modeled in a high fidelity fashion via explicit simulation of power regulation/usage and interference/fading phenomena on the basis of the current state of the corresponding cell (also expressed as a function of current meteorological conditions). The interest in simulating this type of system is related to the need for assessing whether current dimensioning is adequate for supporting both normal workload conditions, as well as peak workload conditions related to traffic bursts along the highway, possibly leading to traffic congestions. At the same time, assessing the availability of radio channels, and determining the power consumption, for serving sub-urban areas close to the ring highway even in case of workload peaks associated with traffic rushes is another aspect of interest. The power regulation model has been implemented according to the results in [26]. Specifically, each modeled GSM cell tracks, via dynamically-allocated data structures, channel allocation and power management information for ongoing calls. Upon the start of a call destined to a mobile device currently hosted by a given GSM cell, a call setup record is instantiated within the simulation model via dynamically allocated data structures, which is linked to a list of already active records. Each record is released when the corresponding call ends or is handed-off towards a different cell along the ring highway. In the latter case, a similar call-setup procedure is executed at the destination GSM 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 Signal/Interference Ration (SIR) value, according to GSM technology. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a meteorological model defining current climatic conditions (and related variations) around the city of Rome. The climatic model accounts for variations of the climatic conditions (e.g. the current wind speed) with a minimum time granularity of ten seconds. We have simulated a whole week of operativity of the GSM coverage system along the highway, by explicitly accounting for dynamic day-time traffic variations, and differentiated climatic conditions. Statistics about the vehicle traffic variations have been derived from [27]. Simulated night-time periods are characterized by near-zero utilization factors (correspondingly, by the data from [27], less than 800 vehicles run along the high-way in night periods), while rush hours may lead to definitely higher channel utilization factors. For non-weekend days, we have a whole day split into a night-time period, with minimal channel utilization factor, and the remaining part of the day into alternate rush and normal traffic hours. Day-time normal/rush periods lead in our simulations to an increase in the call arrival frequency per cell, and hence to an increase in the channel utilization factor, which depends on the relative density of vehicles along the ring high-way on the basis of the statistics in [27], and on how the mean of an exponential distribution for the call inter-arrival time varies according to that density. Specifically, the average channel utilization factor gets up to 30% in rush periods considering an average call duration of 60 seconds, with oscillations that can lead to even higher peaks. According to [27], weekend days have a different workload, which exhibits a behavior in between normal and night ones.

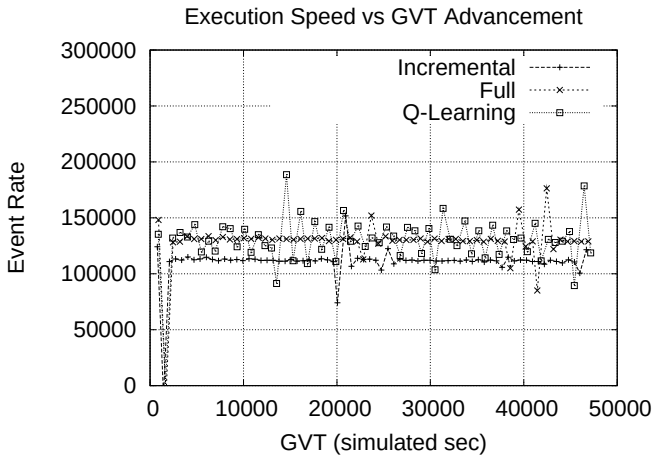

Fig. 3. Simulation Execution

The hardware architecture used for testing our proposal is a 64-bit NUMA machine composed by two AMD Opteron 6174 processors and 32GB of RAM memory. Each processor is equipped with 12 cores that share a 12MB L3 cache, each core has a 512KB private L2 cache and 2200MHz speed. The software architecture consists of 64-bit Suse Enterprise 11, with Linux Kernel, version 2.6.32.13. The compiling and linking tools used are gcc 4.3.4 and binutils (as and ld) 2.20.0.

In Figure 3 we present the variations of the amount of committed events per wall-clock-time second (event rate) achieved while simulating specific virtual time periods, represented by the variation of the GVT on the x-axis. This parameter indicates the speed according to which a given virtual time period is simulated. The higher the event rate, the faster is the execution while simulating a given virtual time period. We report three curves referring to (i) the case in which the Q-Learning layer is active (ii) the case in which the Q-Learning layer is active, but we always force the incremental log/restore mode, and (iii) the case in which the layer is active but the non-incremental (full) log/restore mode is forced. The plots for cases (ii) and (iii) express performance levels that could be achieved via an optimized log/restore mode based on either the incremental or the non-incremental log mode, but not allowing autonomic switch between the two modes on the basis of run-time dynamics, as typical of a wide set of literature solutions.

By the results, we see that, most of the time the Q-Learning has an event rate which is higher than the Forced-Incremental and the Forced-Full configurations, due to the selection of those actions which allow to benefit from the best operating mode available during that particular part of the simulation, depending of the actual dynamics (e.g. in terms of state size, event granularity, memory update pattern and so on). Low peaks are due to the ϵ-greedy policy or to non-complete

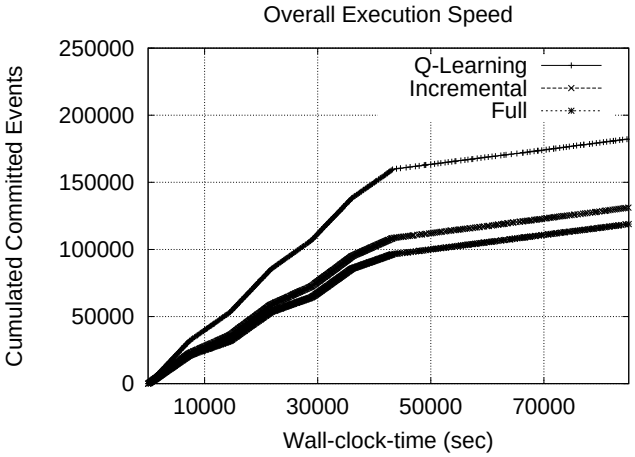

Fig. 4. Simulation Throughput

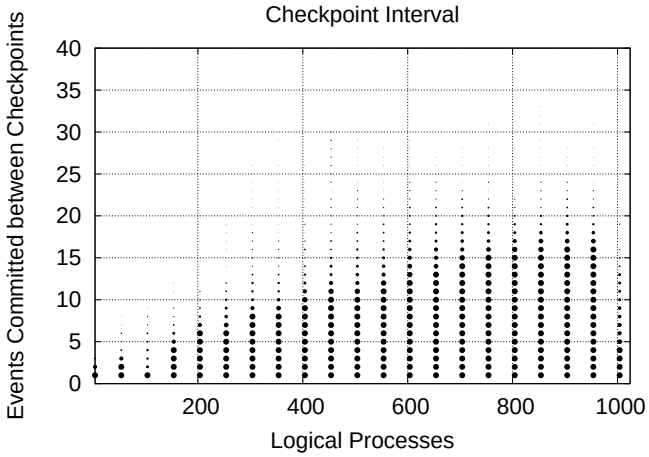

Fig. 5. Chekpoint Intervals

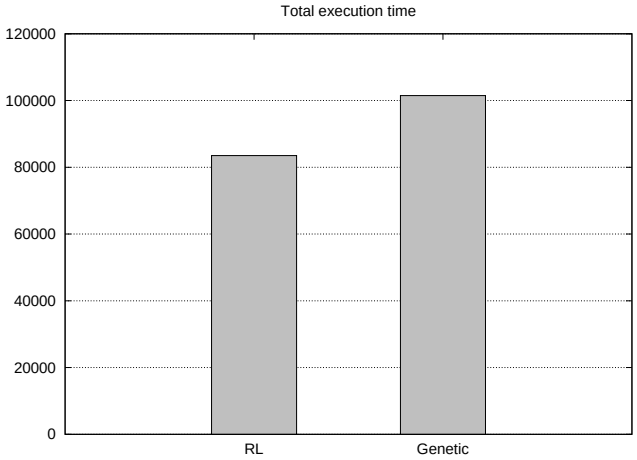

Fig. 6. Execution time comparison

exploration of the space, which can be as well due to changes in the exection pattern of the application-level software. The parameter ϵ has been set to 0.1, a non-minimal value which allows to account for dynamics variation.

Nevetheless, the final effect on performance by the above optimized behavior is expressed by the plots in Figure 4, where we draw the cumulated amount of committed events vs. wall-clock time for the simulation run. These curves express the ability of each log/restore configuration to commit events (and hence to carry out useful simulation work) while wall-clock time goes ahead. Hence we have a representation of how fast the simulation model is executed vs wall-clock-time, which is a representation of the perceived execution speed. By the results, the ability of the Q-Learning configuration to always switch to the best suited mode is reflected in the fact that its cumulated event rate curve always exhibits the best slope. In other words, it allows the model execution to be carried out in a significantly faster manner, compared to the other to schemes. Given that these modes are anyway optimized, thanks to the dynamic selection of well-suited log intervals, this is a significant result.

To study whether our proposal is able to change the estimation of χ^{OPT}, we observe the checkpoint intervals (i.e., the number of events executed before a checkpoint is taken) selected by our RL agent. We run a set of simulations varying the number of macro-cells in between 4 and 1024, evenly distributed on four simulation kernels, each managing up to 1000 wireless channels. We have taken into account only those checkpointing intervals related to a real exploit of the *Q-factors* learnt by the agent, thus we have discarded all those intervals related to a checkpoint taken because of the ϵ-greedy approach. Results are presented in figure 5, showing the checkpoint intervals selected by our agent, with the relevance of a selection (i.e., the checkpointing-interval frequency) represented by the thickness of the points. As we can see, with a small number of LPs, the system selects small intervals. This is reasonable, because when few processes run by a simulation kernel, there is the possibility that one process goes faster in the simulation, thus receiving a large number of order-violating messages, which cause rollbacks. Since the length of the rollback is proportional to the checkpoint interval, the agent tries to reduce this overhead by taking checkpoints more often. On the other hand, when a large number of LPs is used in the simulation, the overall workload is much more well distributed, thus allowing the agent to select as well larger checkpoint intervals. Nevertheless, overloaded periods still suffer from the high number of rollbacks, thus requiring small intervals when the simulator is under stress.

To complete the study, we compare the execution of our RL-based engine with the genetic algorithm-based previously presented in [19], namely relying on the implemetation provided by the same ROOT-Sim version. The configuration of the benchmark is the same, and we have configured the two different runs using the same seeds for the pseudo-random generators. In Figure 6 we report the execution times for the two autonomic optimization strategies. By the results, we can see that the RL-based solution offers a performance increase

on the order of 20%. This is related to the finer granularity according to which the RL-based optimization strategy is able to select the checkpointing interval χ. In fact, as we have shown by the data in Figure 5, the RL-based optimization scheme actually selects checkpointing intervals in a range [1, 35]. In the same range, the genetic algorithm presented in [19] can select only the values 1–8, 10, 12, 15, 20, 25, 30, 35, 40. Therefore, the genetic algorithm could more likely select a sub-optimal strategy for a certain execution phase.

VI. Conclusion

In this paper we have presented the design and implementation of an autonomic optimization system for log/restore layers targeting checkpoint-based optimistic simulation. Our solution allows to rely on standard dynamic memory services to implement simulation models, and transparently supports both incremental and non-incremental log/restore modes (in time-interleaved fashion) depending on current execution dynamics. The selection of the best suited log mode is based on a Reinforcement Leaning agent, exploring and updating *Q-factors* to determine the best mode to execute the simulation with, using an ϵ-greedy policy to alternate exploitation and exploration. The effectiveness of the approach has also been tested with a real-work case study related to wireless connectivity along a ring high-way and in a general-case mobile phone cell distribution.

References

[1] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.

[2] D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and System*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[3] A. C. Palaniswamy and P. A. Wilsey, "An analytical comparison of periodic checkpointing and incremental state saving," in *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, ser. PADS. ACM, 1993, pp. 127–134.

[4] R. M. Fujimoto, J.-J. Tsai, and G. C. Gopalakrishnan, "Design and evaluation of the rollback chip: Special purpose hardware for Time Warp," *IEEE Trans. Comput.*, vol. 41, no. 1, pp. 68–82, 1992.

[5] F. Quaglia and A. Santoro, "Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 6, pp. 593–610, Jun. 2003.

[6] R. Ronngren, M. Liljenstam, R. Ayani, and J. Montagnat, "Transparent incremental state saving in Time Warp parallel discrete event simulation," in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, May 1996, pp. 70–77.

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

[8] A. C. Palaniswamy and P. A. Wilsey, "An analytical comparison of periodic checkpointing and incremental state saving," in *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 1993, pp. 127–134.

[9] B. R. Preiss, W. M. Loucks, and D. MacIntyre, "Effects of the checkpoint interval on time and space in Time Warp," *ACM Transactions on Modeling and Computer Simulation*, vol. 4, no. 3, pp. 223–253, Jul. 1994.

[10] R. Ronngren and R. Ayani, "Adaptive checkpointing in Time Warp," in *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*. Society for Computer Simulation, Jul. 1994, pp. 110–117.

[11] R. S. Sutton and A. G. Barto, "Reinforcement Learning I: Introduction," 1998.

[12] J. Wang and C. Tropper, "Optimizing time warp simulation with reinforcement learning techniques," in *WSC '07: Proceedings of the 39th conference on Winter simulation*. Piscataway, NJ, USA: IEEE Press, 2007, pp. 577–584.

[13] ——, "Selecting gvt interval for time-warp-based distributed simulation using reinforcement learning technique," in *SpringSim '09: Proceedings of the 2009 Spring Simulation Multiconference*. San Diego, CA, USA: Society for Computer Simulation International, 2009, pp. 1–7.

[14] A. Pellegrini, R. Vitali, and F. Quaglia, "Autonomic state management for optimistic simulation platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1560–1569, Jun. 2015.

[15] S. Skold and R. Rönngren, "Event Sensitive State Saving in Time Warp Parallel Discrete Event Simulation," in *Proceedings of the 1996 Winter Simulation Conference*. Society for Computer Simulation, 1996, pp. 653–660.

[16] F. Quaglia, "Event History Based Sparse State Saving in Time Warp," in *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 1998, pp. 72–79.

[17] J. Fleischmann and P. A. Wilsey, "Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators," in *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 1995, pp. 50–58.

[18] F. Quaglia, "A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 4, pp. 346–362, 2001.

[19] A. Pellegrini, R. Vitali, and F. Quaglia, "An evolutionary algorithm to optimize log/restore operations within optimistic simulation platforms," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools. SIGSIM, 2011.

[20] A. Pellegrini, "Hijacker: Efficient static software instrumentation with applications in high performance computing (poster paper)," in *Proceedings of the 2013 International Conference on High Performance Computing & Simulation*, ser. HPCS. IEEE Computer Society, Jul. 2013, pp. 650–655,

[21] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*, ser. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. New York: John Wiley & Sons Inc., 1994, a Wiley-Interscience Publication.

[22] A. Gosavi, *Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[23] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[24] N. H. Vaidya, "Impact of checkpoint latency on overhead ratio of a checkpointing scheme," *IEEE Trans. Comput.*, vol. 46, no. 8, pp. 942–947, 1997.

[25] "Private communication."

[26] S. Kandukuri and S. Boyd, "Optimal power control in interference-limited fading wireless channels with outage-probability specifications," *IEEE Transactions on Wireless Communications*, vol. 1, no. 1, pp. 46–55, 2002.

[27] "http://traffico.octotelematics.it/."