Autonomic Log/Restore for Advanced Optimistic Simulation Systems

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

***Abstract*—In this paper we address state recoverability in optimistic simulation systems by presenting an autonomic log/restore architecture. Our proposal is unique in that it jointly provides the following features: (i) log/restore operations are carried out in a completely transparent manner to the application programmer, (ii) the simulation-object state can be scattered across dynamically allocated non-contiguous memory chunks, (iii) two differentiated operating modes, incremental vs non-incremental, coexist via transparent, optimized run-time management of dual versions of the same application layer, with dynamic selection of the best suited operating mode in different phases of the optimistic simulation run, and (iv) determination of the best suited mode for any time frame is carried out on the basis of an innovative modeling/optimization approach that takes into account stability of each operating mode vs variations of the model execution parameters.**

I. Introduction

A traditional way to achieve high performance simulations is the employment of parallelization techniques [9]. They are based on the partitioning of the simulation model into Logical Processes (LPs) that can execute events in parallel on different CPUs and/or different CPU-Cores, and rely on synchronization mechanisms to achieve causally consistent execution of simulation events at every LP.

As it is well recognized, the optimistic synchronization approach (based on rollback for recovering possible timestamp order violations due to the absence of block until safe policies for event processing) is likely to favor speedup in general application/architectural contexts. On the other hand, the design/development process of optimized supports for state recoverability is a major obstacle for the construction of efficient optimistic simulation systems. This process is additionally hardened when complete transparency, vs the application layer, is pursued.

In this paper we present a log/restore architecture designed according to the autonomic-computing paradigm, which jointly addresses transparency and performance issues by:

- 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.
- Transparently enabling phase-interleaved adoption of incremental and non-incremental log/restore modes.
- Running each log/restore mode in a highly optimized fashion, via the adoption of classical schemes for the optimization of typical parameters determining the actual overhead for each mode.
- Dynamically (and transparently) switching to the best suited operating mode (incremental vs non-incremental) depending on proper execution dynamics of the optimistic run.

Our autonomic log/restore architecture has been developed using `C` technology, and has been tailored to Linux systems, the ELF standard and to IA-32/x86-64 compliant processors. It builds on our results in [11], [21], providing respectively, a solution for transparent non-incremental log/restore of simulation object states scattered across non-contiguous, dynamically allocated memory chunks, and a complementary extension based on transparent, light instrumentation techniques, allowing the tracking of memory updates occurring within the dynamic memory map, so to enable log/restore incrementality. However, the present work significantly extends these results by providing an optimized integration/coexistence of the two log modes (incremental vs non-incremental) according to the autonomic paradigm.

The autonomic layer has been integrated within the ROme OpTimistic Simulator (ROOT-Sim), 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 assessing the viability and efficiency of our proposal for a case study related to GSM coverage along the ring high-way running around the city of Rome.

The remainder of this paper is structured as follows. In Section II related work is discussed. The autonomic architecture is presented in Section III. Experimental data are reported in Section IV.

II. Related Work

Recoverability issues in optimistic simulation systems have been addressed by providing architectures capable of supporting log/restore tasks and/or defining performance models aimed at optimizing the overhead vs specific parameters characterizing the employed log/restore scheme. The proposals in [7], [12], [14], [15] address the case of non-incremental logging, while incremental schemes have been provided in [16], [20], [23]. Solutions based on a mixture of the two approaches (incremental vs non-incremental) can be found in [8], [19]. With the above solutions there is the need (i) to supply the necessary code to collect snapshots of the objects' state inside the application level software, or (ii) to employ calls to functions within the API of proper checkpointing libraries, or (iii) to statically identify (e.g. at compile-time) the portions of the address space that need to be considered part of the state. Consequently, perfect transparency is not supported since the programmer must necessarily be faced with issues related to state snapshots.

Also, static identification of the memory locations to be included within the snapshot is non-compatible with desirable programming approaches entailing dynamic memory allocation/deallocation at the simulation object level. Compared to all the above proposals, the autonomic architecture we present in this paper does not require log/restore modules to be embedded within the application code, or explicit interfacing with log/restore libraries. Also, it supports both incremental and non-incremental log/restore modes in an optimized combined manner, while allowing the simulation object state to be scattered on dynamically allocated non-contiguous memory chunks.

Log/restore transparency for general memory layouts has been tackled for optimistic synchronization in the context of HLA-based federated simulations [17], [18]. These solutions rely on kernel level memory protection mechanisms offered by the Operating System to detect memory accesses and to trigger incremental copies of the accessed pages. Our autonomic architecture supports incremental log/restore via a lightly instrumented version of the application modules, and allows tracking memory updates with arbitrary granularity. As a consequence, the overhead for tracking updates and incremental log operations in the above scheme is likely higher (e.g. since it exhibits page size granularity) and affordable only when comparable with the cost of interoperability services supported by HLA middleware. On the other hand, our work targets more traditional optimistic simulation engines, typically relying on a restricted set of services, where the relative cost of log/restore operations can represent a dominating performance factor.

The issue of dynamic memory based states for optimistic simulation objects has also been addressed by the optimistic simulation frameworks in [3], [6]. However, ad-hoc APIs are used to explicitly notify to the simulation kernel that specific allocation/deallocation operations, and, more in general, operations on data structures based on dynamic memory (e.g. lists), need to be rollbackable. Hence, differently from what happens with our autonomic architecture, dynamic memory based layouts via ANSI-C memory allocation/deallocation services are not supported.

Finally, some recent advances (see, e.g., [4]) have shown the viability of recoverability via reverse computation. However, in general simulation contexts (e.g., possibly exhibiting non-reversible execution paths), this approach still needs to be complemented via optimized log/restore techniques, like the one we provide in this work.

III. The Autonomic Log/Restore Architecture

A. Coexistence of Different Log/Restore Modes

As hinted, the autonomic architecture we present in this paper builds on our results in [11], [21], where an advanced log/restore subsystem has been incrementally developed. The building block component is a so called memory-map manager, acting as a wrapper of ANSI-C memory allocation/deallocation services, transparently interposed in between the application software and the malloc library via proper linking directives. The memory-map manager intercepts allocation/deallocation requests from the overlying application, and serves them via pre-allocation of contiguous blocks of chunks, each hosting chunks of different power-of-two sizes. A meta-data table, named `malloc_area`, is kept for each simulation object, which is used for identifying virtual addresses of the memory blocks that have been pre-allocated for serving allocation requests from that object, and for maintaining, via a concise bit-map representation, the state (busy or free) of each chunk within a block. Mechanisms for dynamic expansion of both the meta-data table and the amount/size of pre-allocated blocks are included. The memory-map manager supports log operations of the memory scattered state of the simulation object by packing all the allocated chunks within a contiguous log-buffer, together with minimal meta-data that are required to put the logged chunks back in place in case of restore of the objects state to a logged image. This originally occurred according to a non-incremental mode.

To enable incremental logging, the memory-map manager has been successively integrated with a compile/linking time instrumentation tool (which has been tailored for IA-32/x86-64 architectures and the ELF) allowing transparent integration of a lightweight tracking mechanism of update operations occurring within the scattered memory-map associated with the object state. Every application level module is instrumented via the insertion of a call to an assembly monitor right before each memory-write instruction. The monitor retrieves the exact address of the memory-write instruction via the Program-Counter return value registered within the monitor stack-frame, and uses it to access a hash table acting as a cache of disassembling records, which tells the monitor how to reconstruct (on the basis of the current value of CPU registers) the exact address/size of the memory area to be dirtied by the memory-write instruction ([1]). Once done this, the corresponding chunk is identified, and is flagged as dirty within an additional bitmap. Upon incremental log operations (which, similarly to full - non-incremental -logs, can occur according to a periodic approach), the memory-map manager packs within the log-buffer only the currently in-use chunks that are also registered as dirty, again together with the required meta-data to be used upon state restore of the logged state image. Incremental logs can be interleaved with full logs so to enable fossil collection upon GVT operations, since they bound the amount of backward traversing steps along the log chain in order to retrieve the chunks that have been dirtied up to the restore point, which need to be put back in place.

We have expanded the above design in order to provide an optimized coexistence of the two different log modes. One way to possibly achieve such a coexistence would have been to simply add a (per simulation object) flag indicating

[1] Third party libraries are not instrumented, and the memory locations updated by the execution of modules within third party libraries are determined by wrapping the library call and exploiting the corresponding actual parameters.

to the memory-update tracking monitor whether the logging layer is currently executing in incremental mode or not. In the latter case, the monitor does not actually need to perform identification of the memory chunk to be dirtied, and to flag the corresponding dirty-bit. It could simply return right after checking the flag value. However, this solution would actuate the non-incremental logging mode by running application level modules that actually experience part of the overhead associated with the memory-update tracking monitoring mechanism (caused by a set of machine instructions, which include an explicit call for flow control variation and the associated stack-frame setup). This would mean running a non-optimized application layer configuration vs the current operating mode of the underlying log layer.

We have instead adopted a different approach where automatic ELF rewriting schemes have been used in order to create, starting from the same set of application level modules, two different text sections within the ELF, one containing a non-instrumented version of the compiled modules, and the other one containing the instrumented counterpart. These two sections are then transparently placed within different virtual memory sections thanks to standard `ld` facilities. However, the corresponding symbol tables are modified by our preprocessing/instrumenting tool in order to expose the application interface requested by the underlying simulation kernel, namely the event handler callback, via differentiated symbols. The rodata sections corresponding to the two different text sections are modified in order to provide correct adjustment of the displacement information associated with the position of code and data within the virtual memory addressing. Also, the replicated data/BSS sections associated with the two versions of the application object code have been collapsed on the same virtual addressing range in order to provide a single actual copy of initialized and non-initialized data, accessible by both the generated code versions. A schematization of the whole process supporting such a dual-version code generation is provided in Figure 1, where we explicitly indicate the steps carried out by our ad-hoc automatic compile/linking time instrumentation tool. Once the executable is finally built and run, a kernel level switch between the two different log modes simply involves reassigning the event-handler callback pointer to the entry point symbol associated with the corresponding version of the duplicated application executable modules. Adopting this solution, each log mode is supported according to an optimized run-time scheme where any overheads are at all avoided while processing simulation events in case no tracking of memory update operations is requested by the currently active log mode.

The above scheme would only entail additional virtual addresses consumption due to the presence of two versions of the executable modules associated with the application layer. However, this should not represent a real problem when considering the tendency of vendors towards 64-bit processors, enabling extremely wide span of virtual memory addressing, and the fact that text sections usually fill a reduced percentage of the available virtual addresses.

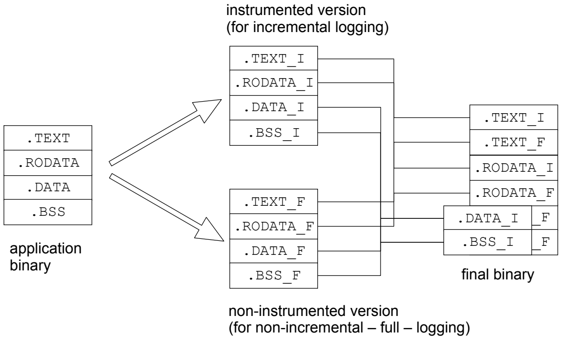

Figure 1. Dual-Version Code Generation via Compile/Linking Time ELF Rewriting.

B. Log/Restore Overhead Modeling

After having enabled the optimized coexistence of incremental and non-incremental log/restore modes, as explained in the previous section, we provide the models assessing the corresponding overhead per event (due to both log and restore operations). These models borrow from the one presented in [15] for periodic non-incremental logging, for which we provide both (i) a specialization to capture internal mechanisms proper of our advanced memory-map manager (i.e. the cost of managing meta-data identifying scattered memory layouts), and (ii) an extension to accommodate the case of incremental logging as supported by our architecture. Note that the model in [15] describes the log/restore overhead with per-simulation-object granularity. We inherit this feature in our modeling approach, thus providing an autonomic scheme allowing dynamic optimization of the log/restore mode on a per-simulation-object basis. Consequently, from now on, overhead modeling and autonomic optimization are implicitly referred to what experienced for each single simulation object.

For the non-incremental case, borrowing from [15] and recalling the aforementioned specialization, the log/restore overhead per event can be expressed as

$$OH_F = \frac{S_F}{\chi_F}\delta_{LB} + P_r(S_F\delta_{RB} + \frac{\chi_F - 1}{2}\delta_e) \qquad (1)$$

where

δ_e is the average event execution cost.

S_F is the average size of a full (non-incremental) log.

δ_{LB} is the average cost for logging a single byte belonging to the state image, which we consider to also include the per-byte cost for logging the meta-data maintained by the memory-map manager.

δ_{RB} is the average cost for restoring a single byte from the log, which we again assume to include the per-byte cost associated with the restore of the state layout meta-data.

P_r is the rollback probability (frequency of rollback occurrences over event executions).

χ_F is the selected log interval when operating according to the non-incremental mode, which determines the expected length of the coasting forward phase, occurring after the latest log preceding the causality violation is reloaded.

By the result in [15], the above overhead gets minimized for $\chi_F = \left\lceil \sqrt{2P_r \frac{\delta_{LB} S_F}{\delta_e}} \right\rceil$, and we denote as χ_F^{opt} the optimal non-incremental log-interval according to this equation.

For the incremental mode, as supported by our architecture, log operations in no way require to be forced at each simulation event, but can be taken periodically. In fact they are based on recognizing memory portions that have been dirtied since the last log, independently of the amount of events actually performing the dirtying operations. Accordingly, state reconstruction at whichever simulation time can be supported via a mix of state restore from the log, and classical coasting forward. Also, full logs can be (infrequently) interleaved with incremental logs to enable fossil collection of incremental log records with timestamp less than the timestamp of the latest committed full log. These full logs are anyway exploitable during recovery procedures since, while backward traversing the log chain, the restore operation of a complete state image gets finalized by extracting from the log all the in-use chunks that have not yet been restored via the scan of incremental logs, and putting them back in place within the state layout. To account for such optimized internal mechanisms offered by the memory-map manager, the above equation can be adapted as shown below to model the log/restore overhead for the incremental mode

$$\begin{aligned} OH_I &= \frac{S_P}{\chi_I}\delta_{LB} + \frac{(S_F - S_P)}{\chi_I \chi_{I,F}}\delta_{LB} + \\ & P_r\left[S_F\delta_{RB} + \frac{\chi_I - 1}{2}(\delta_e + \delta_m)\right] + \delta_m \end{aligned} \qquad (2)$$

where the additional/different terms in the equation have the following meaning

S_P is the average size of a partial (incremental) log.

X_I is the selected log-interval when operating according to the incremental mode, which again determines the expected length of the coasting forward phase after the reload of the latest valid state image from the log.

$X_{I,F}$ is the interleave step between full and incremental logs (number of incremental log operations after which a full-log is taken).

δ_m is the cost for running the memory-update tracking module.

In equation (2), the term $S_F\delta_{RB}$ accounting for the cost of state reload from the log is identical to the one in equation (1), due to the aforementioned mechanism, according to which all the in-use chunks belonging to the state image are restored (by retrieving them either from the incremental logs along the log chain, or the first full log found during the log chain backward traversing procedure). Further, each event is charged with the memory-update monitoring overhead δ_m, which also appears during costing forward. By exploiting the same arguments used in [15] for the minimization of the overhead vs the log interval in the context of non-incremental logging, we get that the optimum value for the interval of incremental logs can be computed as $\chi_I = \left\lceil \sqrt{2P_r \frac{\delta_{LB} S_P}{\delta_e + \delta_m}} \right\rceil$, and we denote as χ_I^{opt} the optimal interval according to this equation. Also, by the benchmarking results in [22], a well suited value for $\chi_{I,F}$, providing no significant additional overhead due to full logs, while ensuring efficient memory recovery during fossil collection, is on the order of 10. We have used such a value as a configuration setting for the autonomic log/restore layer.

C. Autonomic Optimization

By the analysis in the previous section we have, for each of the two coexisting log modes, the description of their overhead, together with the indication of the optimum value of the corresponding independent parameters, namely the log-intervals. In our autonomic log/restore architecture, these models are not used to simply select as the best operating log mode the one for which the corresponding expected overhead is minimal (once identified the best log-interval value). Instead, the best suited mode is identified as the one providing the best performance despite plausible fluctuations that can affect the parameters appearing within the overhead models (e.g. the expected event execution cost δ_e), which cannot be directly controlled since they depend on proper run-time dynamics related to the simulation model execution within the optimistic run. This set involves all the parameters appearing within the performance models, except the log-intervals χ_F and χ_I (or $\chi_{I,F}$), that can be controlled at run-time by the autonomic log/restore architecture.

Such an approach, actually aimed at pro-actively providing stability of the optimal performance, exactly matches characterizing aspects of the innovative autonomic-computing paradigm. Also, it well fits performance optimization when the set of possible operating modes is differentiated, each of them providing different overhead sensibility vs parameter fluctuations and/or variations. Literature approaches for log/restore optimization do not cope with such a multiple operating-mode scenario, which is the reason why sensibility of the a-priori uniquely selected operating mode vs parameter variations did not require to be addressed. Overall, our autonomic scheme for the selection of the best suited operating mode is based on a cost function $CF(\chi_F^{opt}, \chi_I^{opt})$ defined as

$$CF(\chi_F^{opt}, \chi_I^{opt}) = OH_F(\chi_F^{opt}) - OH(\chi_I^{opt}) \qquad (3)$$

and on the result of the integration of this cost function over a multi-dimensional domain defined by the values of the parameters $(\delta_e, \delta_m, \delta_{LB}, \delta_{RB}, P_r, S_F, S_P)$. The integral function allows us to take into account the possible fluctuations of the parameters the cost function is evaluated on.

For each parameter x defining a dimension of the integration domain, we integrate the cost function over the interval $\bar{x} \pm \alpha\bar{x}$, where we suggest $\alpha = 0.1$ to capture statistically relevant fluctuations of the parameters that can be envisaged at the time the dynamic selection is carried out. If the integration result is negative, then the selected operating mode is non-incremental (with the log-interval set to χ_F^{opt}), otherwise the incremental mode is selected (with log-interval set to χ_I^{opt}). Assuming the independence of the parameters defining the integration domain (which is reasonable in our approach since the mean values are operatively determined by direct sampling of the corresponding stochastic processes - see Section III-D), the integral function for $CF(\chi_F^{opt}, \chi_I^{opt})$ is a polynomial, having the following simple form, which allows non-costly evaluation

$$(\frac{2\alpha S_F^2}{\chi_F^{opt}} - \frac{2\alpha S_P^2}{\chi_I^{opt}} - \frac{2\alpha S_F^2 - 2\alpha S_P^2}{\chi_I^{opt}\chi_{I,F}})2\alpha\delta_{LB}^2 +$$

$$2\alpha Pr^2(\frac{\chi_F^{opt} - \chi_I^{opt}}{2}2\alpha\delta_e^2 - \frac{\chi_I^{opt} - 1}{2}2\alpha\delta_m^2) - 2\alpha\delta_m^2 \qquad (4)$$

Given that we only need to determine the sign of the above expressed value, we have finally divided it by 2α, in order to get rid of some machine instructions for multiplications.

The above optimization procedure requires defining a trigger for the evaluation of the integral function in order to dynamically actuate the selection of the best suited log-mode. In our autonomic system, we assume that the simulation run is partitioned into a startup phase and a normal phase. For the startup phase one of the two possible log modes is selected by default, and is kept until the end of that phase. Then, before starting the normal phase, the integral function is evaluated by using the mean $\bar{x}$ and the corresponding relevant statistical fluctuation $\alpha\bar{x}$ for the above parameters defining the integration domain, on the basis of samples observed during the startup phase. Actually, the mean can be computed in a very fast incremental manner not requiring the store of individual samples, thus not even impacting memory consumption.

Once the best suited log mode is selected at the end of the startup phase, subsequent re-selections can occur during the normal phase. The re-selection trigger is based on the current value of the mean $\bar{x}$ of any of the parameters defining the integration domain, and a predicate involving the values $\bar{x}^*$ and $\alpha\bar{x^*}$ that were used upon the last log mode autonomic selection. If for whichever parameter x the expression $|\bar{x} - \bar{x}^*| > \alpha\bar{x^*}$ becomes verified during the run, then the integral function is recalculated on the basis of current mean values. The reason for such a trigger is that the last dynamic selection of the best suited log mode has been actuated on the basis of statistical parameter values $\bar{x}^*$ and $\alpha\bar{x^*}$ that can be considered no more representative of actual run-time dynamics and related fluctuations. In case the current mean goes outside the integration interval for the corresponding parameter, it is likely that some relevant variation has actually occurred within the run time dynamics, which requires re-evaluating the decision about the best suited log/restore mode. In other words, fluctuations (around expected parameter values) accounted for in last log-mode selection step are no more representative of the current system behavior. As a last observation, instead of using the arithmetic mean, we relied on the exponential mean, with weighting parameter set to 0.1, which allows better reactiveness of the mean value vs variations of the corresponding stochastic process.

D. Run-time Parameter Sampling

As hinted, our approach relies on the mean value of the parameters appearing in the performance models in (1)-(2), which are used to define the integration boundaries within the corresponding multi-dimensional integration domain. We rely on a run-time sampling process for computing the mean of each parameter. One relevant difficulty is related to the fact that the mean value of every parameter x appearing in the performance models actually requires to be tracked by the sampling process over time, independently of the current operating mode of the log layer (incremental vs non-incremental). This is because the mean is used both to trigger the re-selection process of the best suited log mode, and to determine the actual outcome of the selection. Accordingly, the parameters δ_m and S_P, specifically used to capture run-time costs proper of the incremental log mode, require to be sampled even when the non-incremental mode is currently operating. Ad-hoc schemes to address this issue will be provided and discussed in this section. We do not explicitly address the issue of sampling the value of P_r since we rely on typical approaches (such as [15]) based on counting the number of rollbacks over a given interval of executed events.

1) Event and Memory-Update Tracking Costs: To determine event and memory-update tracking costs, our autonomic layer implements a sampling mechanism based on the `gettimeofday()` service, offering microsecond granularity. The intrusiveness of this approach is negligible given an overhead of less than 1 microsecond on current conventional machines for the couple of calls required to take start and end time of the interval defining the latency sample to be evaluated.

The used approach is based on a-priori knowledge of the cost $\delta_{tracking}$ for the execution of an instance of the memory-write tracking routine, which depends on the number of machine instructions required to monitor a memory-write access, and on the processor speed ([2]). Also, we have embedded in our architecture an ad-hoc benchmarking module that determines, in a transparent way to the users, the latency of the memory-write tracking routine upon installation of the software on the specific hardware platform.

A per-simulation-object counter $Count$ internal to the autonomic layer is kept, which is used to determine the number of invocations of the memory-write tracking routine occurring during the processing of each event. In case the current log mode is incremental, the application level modules whose execution is currently triggered with invocation of the proper callback entry point (according to the dual-version-code scheme presented in Section III-A) embed the memory-update tracking routine, which increments $Count$

[2] Non-determinism in the amount of instructions to be executed by the tracking modules, possibly arising due to iterative search operations within the meta-data table, is made statistically not relevant thanks to a software caching mechanism that, given the address touched by the memory-write, allows direct mapping to the entry of the meta-data table related to the touched chunk [11].

upon its execution. Denoting with Δ_e^i the wall-clock-time in between the start and the end of the execution of the i-th event by the simulation object, evaluated via the `gettimeofday()` service, we have

$$\delta_e^i = \Delta_e^i - \delta_m^i \tag{5}$$

with δ_e^i denoting the i-th sample for the event cost, and δ_m^i the i-th sample for the memory-update tracking cost, which can be computed as $Count^i \times \delta_{tracker}$.

The above equation boils down to $\delta_e^i = \Delta_e^i$ in case the log layer is currently adopting the non-incremental mode, since the corresponding application code version does not entail memory-update tracking modules. However, as pointed out above, we need the sample δ_m^i also in case the current log mode is non-incremental. To provide the value of δ_m^i in such a situation, we have slightly modified the differentiated dual-version code generation procedure in such a way that the code version running when non-incremental log/restore is active embeds a very light instrumentation scheme where each memory-write instruction is preceded by a single `ADD-r/m32,imm8` assembly instruction allowing the update of $Count$. In this way, we can infer the value of δ_m^i by simply multiplying the i-th sample of the counter value by the known in advance cost $\delta_{tracking}$, exactly as if the incremental mode were active. We note that this approach requires instrumenting memory-writes via a negligible overhead (just thanks to the single machine-instruction instrumentation approach), hence not altering the validity of the overhead model in expression (1), describing the case of non-incremental logging, which excludes costs associated with instrumenting instructions within the application code.

We note that the above mechanisms based on real-time clocks accessed by the `gettimeofday()` service directly fits cases where the computing platform is dedicated to the parallel simulation run, as typical of scenarios where performance is a critical factor. In case of time-sharing with other applications, such an approach needs to be complemented with solutions based on code pre-analysis and lightweight run-time profiling such as the one discussed in [13].

2) Size of Full and Partial Logs: Samples S_F^i of the size of full (non-incremental) logs can immediately be taken by the autonomic layer independently of the currently active log mode since the memory-map manager maintains meta-data (i.e. an accumulator) recording at any time the real memory occupancy of the object state image (in terms of the amount of bytes associated with currently allocated chunks). Hence, S_F^i samples can be taken by simple querying the memory-map manager for the value of the accumulator. In our implementation, the autonomic layer queries the memory-map manager each time a log (incremental or non-incremental) is taken.

A different approach is instead required for taking S_P^i samples of partial (incremental) logs. Specifically, when the currently active log mode is incremental, the memory-map manager updates a second accumulator accounting for the amount of bytes associated with chunks that have been dirtied since the last log. The accumulator is updated on the basis of actual memory-write operations that are tracked at run-time. This accumulator was already included within the design of the incremental log supports presented in [11] since it was used to determine the size of the buffer destined to keep the incrementally dirtied chunks. The value of this accumulator is therefore directly used as a valid S_P^i sample when the incremental log mode is active.

In case the current log mode within the autonomic scheme is non-incremental, the above accumulator does not get updated. Hence, we have decided to infer the value of S_P^i according to the following different approach. Each $K \times \chi_F^{opt}$ non-incremental log operations, we flag the corresponding simulation object so that, after the subsequent event is executed by the object, we compare chunk by chunk the current memory image content after the event with the last one packed within the log buffer. The comparison is carried out only over chunks that belong both to the memory image packed within the log buffer, and to the current memory image, hence taking into account the portion of the state layout that is stable across the two subsequent snapshots.

Obviously, the cost of this operation depends on the value of K and on specific optimizations for the comparison of each couple of chunks. As for the second aspect, we have not employed the traditional `memcmp()` service since, depending on the implementation, it might not provide early stop upon detection of the first different byte between the two memory chunks. We have therefore developed efficient, ad-hoc assembly modules that iteratively compare memory areas by fully exploiting the size of CPU registers at each compare-step, and that exactly implement the early stop procedure upon the detection of the first different byte between the two chunks. This matches the chunk-based granularity offered by the log/restore approach within the autonomic layer. Also, these modules are optimized in order to maximize the likelihood of actual early stop in case of different chunks between the two snapshots according to the following scheme. Small size chunks are checked within the comparison process by starting from the top byte, and then going towards the bottom. Instead, for large chunks, we have implemented a procedure that checks the bytes in an interleaved mode starting from the top and from 3/4 of the chunk size. The above approach well fits typical programming practices, which tend to structure records in such a way that the most frequently touched data are at the top of the record and/or at the bottom (see, e.g., pointers for linking between memory scattered dynamically allocated records). Hence, for large chunks, it is better to check top/bottom portions with higher priority. Also, starting from 3/4 of the chunk size accounts for internal chunk fragmentation, due to the typical un-correlation between the size of the record to be placed by the application software within the allocated chunk, and the actual size of the chunk that best fists the allocation request, among those managed by the memory management subsystem (defined according to power-of-two values). Once identified the dirty chunks according to the above scheme, on the basis of the aforementioned stable portion of the snapshot, the corresponding percentage p of dirty bytes is applied to the total current state size S_F^i to generate the j-th S_P sample as $S_P^j = p \times S_F^i$.

Concerning the value of K, namely the second factor determining the actual overhead due to the estimation of S_P samples when the non-incremental log mode is active, we have used a static approach where K is set to the value 20. Given that the cost associated with the estimation procedure for a single sample is, at worst, comparable with the one for a full-log operation ([3]), this would simply increase the real overhead experienced when the non-incremental mode is active by, at worst, 5% of the corresponding logging overhead.

By the above optimizations, the overhead for determining S_P samples when the non-incremental mode is operative is expected to be negligible, thus again not altering the validity of the non-incremental overhead model in equation (1).

3) Per-Byte Log/Restore Costs: The last parameters involved in the sampling process are the per-byte log/restore costs, namely δ_{LB} and δ_{RB}. However, δ_{RB} does not appear in the final formula and we concentrate on δ_{LB}. To sample δ_{LB}, we have again exploited the `gettimeofday()` service, in combination with the sampling process of S_F and S_P depicted in the previous section. In particular, the i-th log operation latency, say Δ_{log}^i, is sampled via `gettimeofday()` and is normalized to either the corresponding S_F sample, or the corresponding S_P sample, just depending on the currently active log mode. Given that Δ_{log}^i also accounts for the cost of manipulating and logging meta-data associated with the logged chunks, the normalization allows taking samples for δ_{LB} actually expressing how the meta-data management cost is charged to the log operation of each single byte.

IV. Experimental Results

In this section we report experimental results for an evaluation of the presented autonomic log/restore architecture. The test-bed simulation model refers to GSM coverage along the ring high-way 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 of coverage along the highway. As in the actual system organization supported by the in charge Telecommunication Company, each cell hosts 1000 radio channels [2]. 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). GSM cells along the ring high-way primarily serve communication requests for mobiles on board of vehicles running on the high-way. 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 rushes, 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 on 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 [10]. 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 via dynamically allocated data structures, which gets linked to a list of already active records. Each record gets 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 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 10 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. Statistics about the vehicle-traffic variations have been derived from [1]. Simulated night-time periods are characterized by near-zero utilization factors (correspondingly, by the data from [1], 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 [1], 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 [1], weekend days have a different workload, which exhibits a behavior in between normal and night ones. Exact traces for calls involving mobiles along the high-way could not be directly used due to privacy issues.

As pointed out, our autonomic layer has been integrated

[3] Memory compare operations are in fact similar in cost to memory copies, since they both involve similar memory/register data moves. Also, the early stop for chunk compare operations should additionally favor the latency of comparing the chunks across the stable portion of the snapshot.

within the optimistic ROOT-Sim platform, which ultimately relies on MPI for data exchange across different instances of the simulation kernel layer. Beyond the already depicted log/restore facilities, ROOT-Sim also offers completely transparent mapping of LPs onto the concurrent kernel instances, as well as CPU scheduling and event queues management. ROOT-Sim supports a very simple programming model based on a callback event handler, a service allowing injection of events for whichever LP, and an additional callback passing to the application layer a committed object state image, which can be used to implement, e.g., termination detection based on stable predicates [5]. The above simulation model has been developed for integration with the ROOT-Sim environment in a way that each LP instance models a single GSM cell. Callbacks involve therefore the update of the state of individual cells, and cross-LP events are essentially related to hand-offs between different cells.

The hardware platform used in this experimental study is a QuadCore machine, equipped with an Intel Core 2 Quad Q6600 (64bit execution support, 2.4GHz, 4MB L2 Cache per couple of cores, 32KB L1 Cache per core, 1GHz Front Side Bus speed) and 4GB of RAM memory. As for the software environment, the running Operating System is GNU/Linux (kernel 2.6.22-31 64bit, distribution OpenSUSE 9.2), the used `gcc` version is 4.2.1, the used `binutils` version (`ld` and `gas`) is 2.17.50 and the used MPI version is OpenMPI 1.2.4. Four instances of the ROOT-Sim kernel have been run on this platform (one per CPU-core), each hosting two adjacent GSM cells along the ring high-way.

We report in Figure 2 (left) the variation 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. Actually, this parameter indicates the speed according to which a given virtual time period is simulated. The higher the event rate, the faster the execution while simulating a given virtual time period. We report three plots referring to (i) the case in which the autonomic layer is active (ii) the case in which the autonomic layer is active, but we always force the incremental log/restore mode, with the corresponding optimized value for χ_I and (iii) the case in which the layer is active but the non-incremental (full) log/restore mode is forced, with the corresponding optimized value for χ_F. The plots for cases (ii) and (iii) express performance levels that could be achieved via an optimized log/resotre mode (adaptive in the selection of the log interval) 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.

By the results, we see that, depending on the simulated period (night-time vs day-time), forced-incremental and forced-full modes alternately exhibit better execution speed. In particular, the forced-full mode is faster while simulating night-time periods, while the forced-incremental mode is faster while simulating day-time periods. This is a reflection of the fact that, during night-times and in the weekend, each GSM cell, and hence each LP, exhibits a reduced state size due to the minimal number of records allocated for ongoing calls. This is not the case for day-time periods, where the state size of the LPs can grow significantly (especially for rush hours), up to the limit of slightly less than 70 KB, and the update pattern of the state upon the occurrence of the events allows the incremental-log mode to outperform the full one, once the correspnding log period get optimized. Anyway, the most important outcome by the event rate plots is that the autonomic configuration always switches to the best performing mode (incremental vs non-incremental) depending on the currently simulated period (e.g. night vs day), and hence depending of the actual dynamics (e.g. in terms of state size, event granularity, memory update pattern and so on).

The final effect on performance by the above optimized behavior is expressed by the plots in Figure 2 (right), where we draw the cumulated amount of committed events vs the 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. By the results, the ability of the autonomic configuration to always switch to the best suited mode is reflected in the fact that its cumulated event rate curve always exhibits the best pendency vs wall-clock-time. In other words, it allows the model execution to be carried out in a significantly faster manner, compared to what done by the other two schemes. In particular, the wall-clock-time by the autonomic scheme for reaching the required amount of events to be committed for the whole simulation is reduced of about 13% compared to the forced-full mode, and of about 9% compared to the forced-incremental mode. Given that these modes run according to an optimized configuration, thanks to dynamic (re-)selection of well suited log intervals, this is a significant result.

V. Assessments and Conclusion

In this paper we have presented the design and implementation of an autonomic log/restore layer for optimistic simulation systems. It is a fully featured state management subsystem transparently allowing the use of standard dynamic memory services at the application programming level, and transparently supporting both incremental and non-incremental log/restore modes (in time interleaved fashion) depending on current execution dynamics. The autonomic layer relies on an instrumentation tool allowing dual-version code generation, which ultimately provides an executable image entailing two versions of the application modules, embedding or not memory-write tracing facilities. Each log mode is therefore supported via the run-time exploitation of the best suited code version for that mode.

The autonomic selection of the best suited log mode is based on an innovative modeling/optimization approach relying on the ability to capture fluctuations in the run-time dynamics. The effectiveness of the approach has also

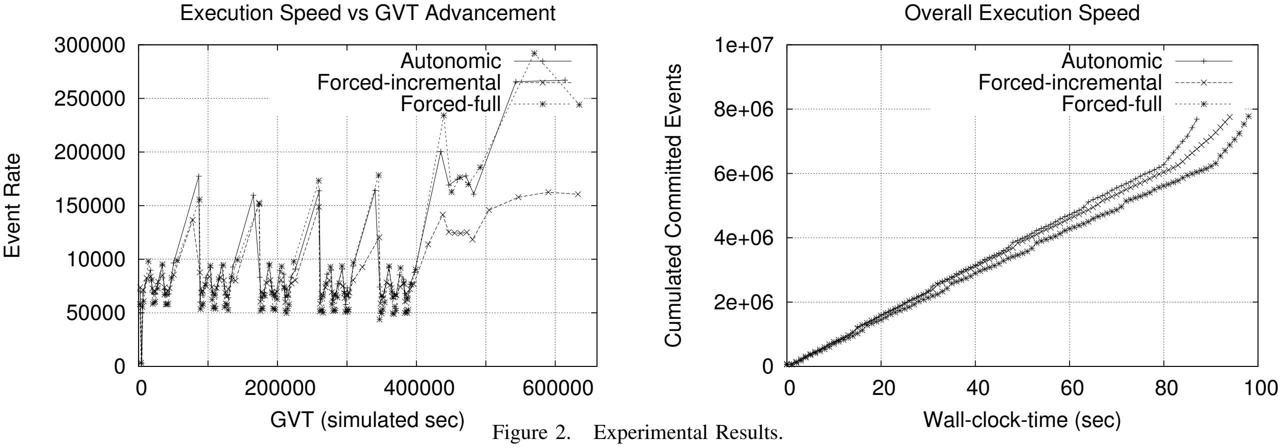

Figure 2. Experimental Results.

been tested with a real-work case study related to wireless connectivity along a ring high-way.

References

[1] http://traffico.octotelematics.it/.

[2] Private communication.

[3] SPEEDES. *http://www.speedes.com*, 2005.

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

[5] 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 IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 227–234, 2007.

[6] S. Das, R. 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, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[7] J. Fleischmann and P. Wilsey. Comparative analysis of periodic state saving techniques in Time Warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 50–58. IEEE Computer Society, June 1995.

[8] S. Franks, F. Gomes, B. Unger, and J. Cleary;. State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 72–79. IEEE Computer Society, June 1997.

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

[10] 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.

[11] 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 23nd Workshop on Principles of Advanced and Distributed Simulation*, pages 45–53. IEEE Computer Society, 2009.

[12] 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*, 4(3):223–253, July 1994.

[13] F. Quaglia. A cost model for selecting checkpoint positions in Time Warp parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):346–362, Feb. 2001.

[14] F. Quaglia and A. Santoro. Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):593–610, 2003.

[15] R. Ronngren and R. Ayani. Adaptive checkpointing in Time Warp. In *Proc. of the 8th Workshop on Parallel and Distributed Simulation*, pages 110–117. Society for Computer Simulation, July 1994.

[16] 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*, pages 70–77. IEEE Computer Society, May 1996.

[17] A. Santoro and F. Quaglia. Transparent state management for optimistic synchronization in the High Level Architecture. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 171–180. IEEE Computer Society, 2005.

[18] A. Santoro and F. Quaglia. A version of MASM portable across different UNIX systems and different hardware architectures. In *Proceedings of the 9th International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society, 2005.

[19] H. Soliman and A. Elmaghraby. An analytical model for hybrid checkpointing in Time Warp distributed simulation. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):947–951, 1998.

[20] J. Steinman. Incremental state saving in SPEEDES using C++. In *Proceedings of the Winter Simulation Conference*, pages 687–696. Society for Computer Simulation, 1993.

[21] 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. IEEE Computer Society, 2008.

[22] R. Vitali, A. Pellegrini, and F. Quaglia. Benchmarking memory management capabilities within ROOT-Sim. In *13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society.

[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.