An Evolutionary Algorithm to Optimize Log/Restore Operations within Optimistic Simulation Platforms

Alessandro Pellegrini
pellegrini@dis.uniroma1.it

Roberto Vitali
vitali@dis.uniroma1.it

Francesco Quaglia
quaglia@dis.uniroma1.it

DIS, Sapienza, Università di Roma

ABSTRACT

In this work we address state recoverability in advanced optimistic simulation systems by proposing an evolutionary algorithm to optimize at run-time the parameters associated with state log/restore activities. Optimization takes place by adaptively selecting for each simulation object both (i) the best suited log mode (incremental vs non-incremental) and (ii) the corresponding optimal value of the log interval. Our performance optimization approach allows to indirectly cope with hidden effects (e.g., locality) as well as cross-object effects due to the variation of log/restore parameters for different simulation objects (e.g., rollback thrashing). Both of them are not captured by literature solutions based on analytical models of the overhead associated with log/restore tasks. More in detail, our evolutionary algorithm dynamically adjusts the log/restore parameters of distinct simulation objects as a whole, towards a well suited configuration. In such a way, we prevent negative effects on performance due to the biasing of the optimization towards individual simulation objects, which may cause reduced gains (or even decrease) in performance just due to the aforementioned hidden and/or cross-object phenomena. We also present an application-transparent implementation of the evolutionary algorithm within the ROme OpTimistic Simulator (ROOT-Sim), namely an open source, general purpose simulation environment designed according to the optimistic synchronization paradigm. Further, we provide the results of an experimental study testing our proposal on a suite of simulation models for wireless communication systems.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Distributed Applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming, Parallel Programming*; I.6.8 [**Simulation And Modeling**]: Types of Simulation—*Discrete Event, Parallel*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIMUTools 2011 March 21–25, Barcelona, Spain.
Copyright 2011 ACM ...$10.00.

General Terms

Algorithms, Performance, Measurement

Keywords

Parallel discrete event simulation, evolutionary algorithms, state recoverability, code instrumentation

1. INTRODUCTION

Parallel and distributed computing techniques are known to be a classical means to build high-performance simulation systems. This is done via the partitioning of the simulation model into distinct objects which concurrently execute simulation events on traditional clusters, on SMP/multi-core machines and/or even on desktop grids [20]. The central problem in the design/development of this type of simulation systems is related to synchronization, whose aim is to allow causally consistent (e.g., timestamp-ordered) execution of simulation events at each simulation object [13].

Among the synchronization approaches proposed in literature, optimistic synchronization [15] is one of the most promising. It avoids block-until-safe policies for event processing and guarantees causal consistency via rollback recovery techniques, which restore the system to a correct state upon the a-posteriori detection of consistency violations. This type of synchronization has been shown to exhibit performance relatively independent of both the lookahead of the simulation model and the communication latency between the concurrently running simulation objects. It results therefore viable and effective for a wide spectrum of application-specific and infrastructure-related settings. 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 also pursued.

In this work we tackle state recoverability via log/restore techniques, and present an evolutionary approach for the adaptive tuning of the parameters determining the operating mode of the log/restore layer. These parameters correspond to the current log mode (incremental vs non-incremental) and the current log interval (number of events between subsequent log operations). They are expressed as a *gene* associated with the simulation object, thus log modes and intervals associated with the whole set of simulation objects express the genotype. Whenever the algorithm performs a mutation on the genotype, the offspring generation is evaluated through a *liveness* function based on the throughput of committed events in order to assess the goodness of the

changes. Hence, the evolution is followed through, making the genotype configuration mutate according to a reward.

Since our approach is only based on the a-posteriori evaluation of the liveness function, the evolutionary algorithm has the ability to cope with run-time dynamics that are not (and would be difficult to be) catched via analytical models of the log/restore overhead, which are typically used as a means for performance optimization (see, e.g., [25, 27]). Among the most relevant dynamics we can mention locality variations (potentially hampering the performance of optimistic systems due to the large use of buffers for, e.g., state logs) and the variation of the rollback pattern (e.g., the frequency of rollback) due to cross-object effects associated with changes of the log interval (and thus of the expected recovery latency) of different simulation objects (the so called thrashing effect [24]). Furthermore, the used liveness function does not mandatorily require (very) fine grain (e.g., microsec) timers for its evaluation, since event throughput can be (re-)evaluated according to a coarser grain period (e.g., millisec/sec). Hence, our solution can be deployed on both dedicated and shared computing platforms, where coarser information on actual CPU usage can be gathered via standard services offered by, e.g., the operating system.

We also present an application-transparent implementation of the evolutionary algorithm within ROOT-Sim (ROme OpTimistic Simulator), a `C`/POSIX-based open source simulation environment relying on the optimistic synchronization paradigm [14]. This implementation exploits an application-transparent dual-coding mechanism that provides the supports for the optimized coexistence of incremental and non-incremental log/restore modes according to the "pay for what you get" paradigm. Further, we report experimental results for an assessment of the viability and efficiency of our proposal when tested on a suite of simulation models for mobile communication systems.

The remainder of this work is structured as follows. In Section 2 we discuss related work. In Section 3 the evolutionary algorithm is presented, together with implementation details. Section 4 is devoted to experimental results.

2. RELATED WORK

Log/restore mechanisms constitute the traditional means for supporting state recoverability in optimistic simulations, and have been deeply investigated in literature. The outcoming proposals deal with the design/implementation of log/restore architectures (either incremental or not) [26, 29, 34, 37], the definition of heuristics and/or the definition of analytical models to be used as the base for, e.g., run-time optimization of the log interval [10, 24, 26, 27]. Compared to the present proposal, none of the above works cope with coexistence and dynamic switch between incremental and non-incremental log modes. Also, the proposed analytical models of the log/restore overhead explicitly neglect hidden run-time dynamics (e.g., locality effects), which are indirectly captured by the presented evolutionary approach.

Mixtures of incremental and non-incremental log modes have been studied in [6, 12, 33], where multiplexed or hybrid log protocols are provided. Differently from our proposal, these studies do not provide application-transparent design/implementation of the protocols. Also, these proposals dynamically adjust the log/restore layer parameters on the basis of analytical models of the overhead which still explicitly neglect run-time dynamics possibly exhibiting a relevant impact on performance.

In terms of architectural design/implementation, a work close to the present proposal is the autonomic log/restore architecture we have presented in [36]. This is based on an application-transparent coexistence of incremental and non-incremental log modes. However, like the aforementioned works, this solution optimizes log/restore parameters by relying on an analytical model not targeted at capturing the above discussed locality and thrashing effects. A direct experimental comparison between this solution and the one provided in this paper (see Section 4) will show how such a lack can lead to suboptimal performance.

As for application transparency, the results in [30, 31] provide a log/restore architecture for federated simulations based on the High-Level-Architecture (HLA). 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. Contrarily, the architecture in which we embed our evolutionary algorithm supports the incremental log/restore mode via a lightly instrumented version of the application modules allowing the tracking of memory updates with arbitrary granularity. Consequently, the log/restore overhead associated with the approaches in [30, 31] is likely higher (e.g., since logging exhibits page size granularity) and affordable only when comparable with the cost of interoperability services supported by HLA middleware. The present work targets more traditional optimistic simulation engines, typically relying on a restricted set of services (see, e.g., [23]), where the relative cost of log/restore operations can represent a dominating performance factor.

3. ALGORITHM DESIGN AND IMPLEMENTATION

3.1 Background on Evolutionary Algorithms

Evolutionary Algorithms (EA) are a class of stochastic optimization methods that simulate the process of natural evolution to build adaptive systems [3, 5, 8]. This class of methods operates on a set (named *population*) of candidate solutions (each one named *individual*), which is iteratively modified. To provide more precise indications on EAs, let us consider an arbitrary optimization problem with k objectives, which are all to be maximized and equally important. A solution to this problem can be described in terms of a *decision vector* $(x_1,\ x_2,\ \ldots,\ x_n)$ in the *decision space* X. A function $f : X \rightarrow Y$ evaluates the quality of a specific solution by assigning it an *objective vector* $(y_1,\ y_2,\ \ldots,\ y_m)$ in the *objective space* Y.

Now, let us suppose that the objective space is a subset of the real numbers, i.e., $Y \subseteq \mathbb{R}$, and that the goal of the optimization is to maximize one single objective. By the fact that there exists a total order on $\mathbb{R}$, a solution $x^1 \in X$ can be considered better than another solution $x^2 \in X$ if $y^1 > y^2$, where $y^1 = f(x^1)$ and $y^2 = f(x^2)$. On the other hand, in the case of a vector-valued evaluation function f with $Y \subseteq \mathbb{R}^k$ and $k > 1$, the comparison of two candidates x^1 and x^2 may not lead to a single optimal solution, i.e., there (might) not exists only a single optimum in the objective space. Following the concept of *Pareto dominance*, an objective vector y^1 is said to *dominate* another objective vector y^2 $(y^1 \succ y^2)$ if at least one component of y^1 is greater than the corresponding component of y^2 and there is no one component which is smaller. Accordingly, we can say that a solution x^1 *dominates* x^2 $(x^1 \succ x^2)$ if $f(x^1)$ dominates $f(x^2)$. Therefore, we

can see that several optimal objective vectors (representing different trade-offs between the objectives) can be found.

The set of optimal solutions in the decision space X is denoted as the *Pareto set* $X^* \subseteq X$, and its image in the objective space is the *Pareto front* $Y^* = f(X^*) \subseteq Y$. Since generating the Pareto set can be computationally expensive (and often unfeasible), the main goal of an EA is to find an approximation of the Pareto set X^* [11,18,32].

EAs are based on two building blocks referred to as *selection* and *variation* As for the former, we can distinguish between *mating* and *environmental* selection, aimed, respectively, at (i) picking promising solutions and (ii) determining which of the newly generated individuals are kept in memory. Concerning variation, it takes a set of solutions and systematically or randomly modifies them in order to actually generate a potentially better population.

The selection process usually consists of two stages: *fitness assignment* and *sampling*. In the first stage, the individuals in the current population are evaluated in the objective space and then assigned a scalar value, the *fitness*, reflecting their quality. Afterwards, a so-called mating pool is created by random sampling from the population, according to the fitness values. Then, the variation operators are applied to the mating pool, which are usually the *recombination* and *mutation* operators.

The recombination operator (also known as *crossover*) takes a certain number of parents and creates a predefined number of children by combining parts of the parents, with a certain probability chosen in order to mimic the stochastic nature of evolution. By contrast, the mutation operator modifies individuals by changing genes (i.e., small parts in the associated vectors) with randomly selected values within a predefined range, according to a given mutation rate. Finally, an *environmental selection* determines which individuals from the population and the modified matin pool form the new population, deterministically choosing the best individuals for survival.

Natural evolution is simulated by an iterative computation process, starting from an initial population which is created at random, or according to a predefined scheme. A loop consisting of the evaluation, selection, recombination and/or mutation steps is executed a certain number of times. Each loop iteration is called a *generation*, and often a predefined maximum number of generations serves as the termination criterion of the loop. At the end, the best individuals in the final population represent the outcome of the EA. Evolutionary programming tends to emphasize a mutation-driven search, where mutation acts on strings. Special effort is put into *elitism*, i.e., the issue of how to prevent non-dominated solutions from being lost, as shown in [9,17,21,38–40].

3.2 Problem Formulation

As hinted, the goal of our proposal is to develop an evolutionary log/restore subsystem capable of simultaneously optimizing log/restore parameters for all the simulation objects involved within the run. These parameters reflect both (i) the mode according to which log operations occur for a simulation object, namely incremental vs non-incremental, and also (ii) the interval (classically evaluated in terms of executed events) in between subsequent log operations. In case the log mode is incremental, the cost of each log operation can be reduced (compared to non-incremental logging), since only dirty portions within the object memory layout are logged. However, we need to pay some cost for tracking write operations occurring within that layout, which depends on application-proper execution patterns (read vs write intensive) as well as on the specific architecture used to support memory-write tracking. On the other hand, for both incremental and non-incremental logging, a state recovery operation upon rolling back may require the reprocessing of intermediate events in between the latest logged state image and the point corresponding to the causality error. This reprocessing phase is referred to as coasting-forward, and typically exhibits an expected cost proportional to the length of the interval in between subsequent logs [24].

As pointed out in the Introduction, beyond the above cited direct costs associated with the adopted log/restore parameters, we may also have hidden effects, such as locality effects (see, e.g., [19]) and the so-called thrashing phenomena. The latter is related to the variation of the latency of the state restore procedure (e.g., due to variations of the coasting-forward latency after a re-tune of the log interval), which can produce cross-object interactions negatively affecting performance. In particular, the work in [24] has experimentally shown the potential for an increase in the rollback amount (exactly referred to as thrashing) in case of large increase in the log interval. This phenomenon has been associated with the potentially significant increase of the expected coasting-forward latency experienced for larger log interval values, which may induce (further) drift of the logical time across different simulation objects while the run goes on.

The above scenario gets even more complicated when considering that log/restore parameters have an impact on the execution dynamics of other core protocols employed to support optimistic synchronization. In particular, they have an impact on the so called fossil collection protocol, which is used to reclaim memory buffers associated with obsolete logs and events. Specifically, upon the calculation of the new value for the commit horizon of the optimistic run, referred to as Global Virtual Time (GVT) ([1]), the latest logged state image with time lower than GVT is searched so that all the logged state information associated with older snapshots is discarded. On the other hand keeping in memory the latest snapshot preceding the GVT does not suffice for guaranteeing state recoverability, since all the simulation events with timestamp in between that snapshot and the GVT also need to be retained. In fact, they could be needed in case of a coasting-forward phase starting exactly from that snapshot. Given this constraint, the larger the interval between subsequent state logs, the larger the amount of simulation events that cannot be collected, and the lower the actual locality for the optimistic run. Such an interaction between log/restore and fossil collection has been addressed in literature by heuristically bounding the log interval of each simulation object to values on the order of 40/50 events [10,27]. However, no approach attempts the determination of log interval values optimizing combined effects in relation to fossil collection.

By the above discussion, the selection of an optimal configuration for log/restore parameters, keeping into account both direct and indirect effects on the final perceivable performance is far from being a trivial task. This is also the reason why, as discussed in Section 2, most of the optimization approaches provided in literature tune the configuration of log/restore parameters on the basis of analytical models

[1]GVT calculation is a form of global predicate which is based on information associated with in-transit and/or unprocessed messages. It is used to determine the current commitment horizon along the simulation time axis, prior to which no causality violation is guaranteed to ever occur.

Table 1: Log Interval Values Expressed by the Gene

Log Interval Range	[1 - 8]	(8 - 12]	(12 - 15]	(15 - 40]
Represented Values	all	each 2	each 3	each 5

that keep into account exclusively direct effects (e.g., the expected overhead for taking each single log).

On the other hand, executing the reproductive cycle of even a simple EA on long individuals and/or large populations requires high computational resources. In fact, evaluating a fitness function for every individual in order to compute its quality as a solution is usually a very costly operation, affordable only in case EAs are used as off-line algorithms. This situation is not perfectly suitable for an advanced optimistic simulation environment, where optimizations must be performed at run-time.

Furthermore, EAs usually try to explore the search space seeking the solution to one single problem. In the context of this work, the environment is constantly changing, i.e., there can be continuous variations in the execution pattern or in the workload associated with the overlaying simulation application. So the evolutionary algorithm optimizing log/restore parameters must promptly adapt the individual with respect to those fluctuations.

We bridge the above aspects via an innovative proposal that works on a very compact population and provides a fitness function, based on the throughput of committed simulation events (expressing the amount of productive simulation work), which is sensitive to environmental changes. These features allow to (i) compute the fitness of the current selected candidate in a fast way, and (ii) make the platform reactive to changes in the application execution pattern, which is obtained at runtime for the whole execution.

In our proposal, the configuration of the log/restore system is represented as a bit string, whose sub-portions (*genes*) represent the current values of the log/restore parameters associated with each simulation object. In such a scenario, compactness deals with keeping the size of the genotype (namely the size of each gene) small. This is directly related to pragmatical effectiveness of the evolutionary approach since reduced size would provide faster convergence towards an optimized configuration via genes evolution. On the other hand, we do not want to loose expressive power while representing log/restore parameters via a reduced size gene. To cope with this aspect we decided to exploit literature results, in particular all those results (e.g., [10, 27]) that have shown how the overhead directly imputable to log/restore can significantly vary for variations of the log interval around small values (e.g., a few units). On the other hand, such an overhead varies in a much less significant manner in case of variations of the log interval around larger values (e.g., on the order of tens). Overall, we decided to express the log interval within the gene associated with a simulation object by bounding it to 40. However, not all the possibilities within the admissible interval [1,40] are really expressed by the gene. In particular, the representable log interval values are non-uniformly scattered across that interval, with higher density towards the lower extreme value 1, and reduced density towards the upper extreme value 40. More precisely, our gene expresses each possible log interval in case of corresponding small values, while it skips expressing some log intervals for corresponding larger values. In more details, the log/restore gene associated with each simulation object is expressed in our proposal by five bits (which we suggest to be embedded within a byte for optimizing machine instruction patterns for accessing and manipulating the genes within any implementation one could devise from our proposal). The most significant bit indicates the log mode (1 for incremental and 0 for non-incremental, also referred to as full log mode), and the least significant four bits are used to access an array that maps 16 different log intervals in the range from 1 to 40 according to the above mentioned non-uniform approach as expressed by Table 1. An example of genes associated with four simulation objects is reported below:

10001010	00001110	00001000	10000000
I |15	F |35	F |10	I |1

Additionally, the evolution of the genotype is carried out by evolving sets of genes in an independent manner. Specifically, the aforementioned bit string is split across all the involved simulation kernel instances, so that each instance has a complete view of the log/restore configuration for the simulation objects it is currently handling. According to this architectural view, the evolution, driven by modifications of the genotype, occurs according to a distributed scheme that directly maps onto a multi-objective optimization where the objective vector $(y_1, \ldots, y_m)$ is such that:

- m corresponds to the number of simulation kernel instances involved within the optimistic run, and
- y_i expresses the throughput of committed simulation events by the i-th kernel instance, across all the hosted simulation objects, within a GVT cycle (i.e., in between two subsequent GVT calculations).

In this way, we avoid each kernel instance to have a complete panorama on the current log/restore configuration of every simulation object and of the fitness of such a configuration (as a whole), which would require some sort of coordination, possibly producing performance degradations. In other words, the above approach allows each simulation kernel to locally evolve parts of the genotype as soon as the new value of the GVT (and hence of the fitness value y_i) is available, without the need for waiting for a coordination phase in which locally evaluated event rates are exchanged across the kernels in order to define a unique event rate value to be used for the assessment of the fitness of the current genotype configuration as a whole.

On the other hand, the evolution of the portions of the genotype hosted by different kernel instances gets correlated by the a-posteriori evaluation via the fitness function. In this way, in case the i-th kernel instance observes a negative variation of y_i, possibly caused by a suboptimal configuration across portions of the genotype hosted by different kernel instances, it will trigger a mutation of its own portion that will adjust the local configuration in order to provide actual synergy with the configuration portions hosted by the other kernels. Alternatively, it will trigger a similar gene rearrangement on those kernels. In both cases, the genotype evolves so to move away from configurations that are distant from the Pareto set, thus actuating configurations that likely approximate one element of this set.

3.3 Gene Mutation

Our evolutionary algorithm can be considered a *Genetic Modifying Algorithm* (GMA). Specifically, whenever we receive a reward value from the system (i.e., we are able to

assess the current system performance), we select those portions of the configuration which can be considered to have provided an improvement or a worsening in the computation efficiency. In particular, whenever we have an enhancement in performance, we can claim that changes in the genotype that have occurred at the last evolution step, have produced a better fitting individual, so we impose a mutation on those parts of the genotype that were not touched in that evolution step. On the other hand, whenever we detect a deterioration in performance, we can argue that the fenotype's poor quality comes directly from the current genotype. In this case, we restore the snapshot of the genotype prior to the last mutation, and resume performing mutations on it.

The genotype is started up in a random configuration. Then the first transformation that occurs over this configuration is associated with a random mutation. In particular, a new string of bits is randomly generated and installed as the current genotype. The above mutation, as well as any subsequent mutation, is triggered upon the computation of a new GVT value.

In every phase the i-th kernel knows the reward associated with the current and the previous genotype snapshots, namely the corresponding values of y_i expressing the throughput of committed events by this kernel instance. Depending on the result of the comparison of these reward values, four possible actions can be taken by the evolutionary algorithm in order to evolve the portions of the genotype hosted by the i-th kernel instance:

- *The last genotype configuration shows a significant improvement (over a certain threshold) with respect to the previous configuration.* In this case the genotype (representation) undergoes a driven mutation step. The parts of the current configuration that differ from the previous configuration are kept, since they are likely responsible for the improvement, while the rest of the genotype is randomly modified to continue the evolutionary process.

- *The last genotype configuration shows a significant worsening (over a certain threshold) with respect to the previous configuration.* In this case, the previous genotype snapshot is used to resume the evolution according to a driven mutation step. Specifically, the parts of the previous gene configuration that differ from the last configuration are kept since their modifications are likely responsible for the worsening. The remaining parts of the genotype get randomly modified to continue the evolutionary process.

- *The last genotype configuration shows a non-significant improvement (under a certain threshold) with respect to the previous configuration.* In this case a new genotype is created randomly starting from the last adopted configuration.

- *The last genotype configuration shows a non-significant worsening (under a certain threshold) with respect to the previous configuration.* In this case, a new genotype is created randomly. However, the older genotype configuration (and its associated throughput) is kept, instead of the last used one, for the comparison in the next evolution phase.

The corresponding evolutionary algorithm pseudo-code is provided in Figure 1.

```
Upon event: new GVT
REWARD_CURR ← NEWLY_COMMITTED/TIME_SINCE_LAST_GVT;
if REWARD_CURR ≫ REWARD_OLD then
    DIFF ← (GENE_OLD ⊕ GENE_CURR);
    GENE_OLD ← GENE_CURR;
    GENE_NEW ← (rand & ¬DIFF) | (GENE_CURR & DIFF);
else if REWARD_CURR ≪ REWARD_OLD then
    switch(GENE_OLD, GENE_CURR);
    DIFF ← (GENE_OLD ⊕ GENE_CURR);
    GENE_OLD ← GENE_CURR;
    GENE_NEW ← (rand & ¬DIFF) | (GENE_CURR & DIFF);
    REWARD_CURR ← REWARD_OLD;
else if REWARD_CURR > REWARD_OLD then
    GENE_NEW ← rand;
    GENE_OLD ← GENE_CURR;
else
    GENE_NEW ← rand;
    REWARD_CURR ← REWARD_OLD;
end if
REWARD_OLD ← REWARD_CURR;
```

Figure 1: Evolutionary algorithm pseudo-code (used by each kernel for updating the local portion of the genotype).

3.3.1 Discussion

Amongst the above four evolutionary steps, the third and fourth steps can be considered as the ones more tightly linked to a very casual generation of new individual configurations. On the other hand, the strength of an EA relies on its ability to provide mutations that are actually oriented to improvements of the individual vs the external environment. This would entail a form of avoidance of purely casual mutations, at least for stable environmental conditions, in order to mimic a kind of evolution process where the history of genotype configurations of better fitting individuals is kept (at least partially) over the subsequent evolution steps. As for this aspect, we note that the purely casual determination of a new configuration according to the third/fourth step above likely can not repeatedly happen many times in case the environment is stable. This is because, for a same environmental configuration (e.g., application execution pattern) the likelihood of keeping a very similar performance level while repeatedly selecting a random genotype likely decreases from generation to generation. On the other hand, in case of continuous environmental variations (where repeated selections of random genotypes might be induced by the above third/fourth step due to casual interactions with environmental changes), such a purely casual selection process represents the strength of the evolutionary process, since the history of the genotype improvements is no more reflected in genotype adequacy vs the changed environmental conditions.

As a last note, the driven evolutionary process embedded within the first and second steps listed above, have been conceived according to the classical hypothesis that the recent past behavior is expected to be representative of the immediate future. In particular, these evolution steps assume that a significant improvement or decrease of the reward function is mostly related to good or bad past choices while modifying the genotype, and not to relatively fast variations of the environmental conditions. This assumption can be subverted since reward variations might be directly induced by environmental variations having a strong, direct impact on the reward metric. As an example, when the (average) event granularity associated with the running simulation model changes over time, this is reflected into changes of the y_i value observed by the i-th kernel, which might be negative

even though the genotype modification by that kernel would have fit the original event granularity configuration. In such a case, by structure, the evolutionary algorithm will likely bias its execution (in subsequent adaptations) towards step three and step four. Thus the above depicted purely-casual evolutionary process will be triggered which will lead, over time, to a different genotype configuration, optimized for the newly materialized environmental conditions, once (and if) they become stable (at least for a while).

3.4 Implementation within ROOT-Sim

We have implemented the evolutionary log/restore algorithm within the ROme OpTimistic Simulator (ROOT-Sim). This is an open source, general purpose platform developed using C/POSIX technology, which is based on a simulation kernel layer that ultimately relies on MPI for data exchange across different kernel instances. The platform transparently supports all the mechanisms associated with parallelization (e.g., mapping of simulation objects on different kernel instances) and optimistic processing, which is carried out in compliance with the Time Warp protocol as specified in [15]. The platform API exposed to the application programmer is quite simple, and consists of one service, namely `ScheduleNewEvent()`, and two callbacks, namely `ProcessEvent()` and `OnGVT()`. The corresponding execution semantics are provided below:

- `ScheduleNewEvent()` allows injecting a new simulation event within the system, to be destined to whichever simulation object, either locally hosted by the same kernel instance, or by a different kernel instance. The input parameters specify the destination object, simply identified via a numerical code, the timestamp for the event to be scheduled, and the event payload (as a flat sequence of bytes).

- `ProcessEvent()` supports the actual processing of simulation events. By this callback the kernel gives control to the application layer, in particular to a specific simulation object, in order to execute a single simulation event. The identity of the dispatched object is specified by the numerical code univocally identifying the object within the system, which is passed as input parameter to the callback together with the payload of the event, and the event timestamp.

- `OnGVT()` allows passing control to the application layer by also providing a reference to a committed snapshot of the simulation object. This facility can be used to support (periodic) audit of the object state trajectory, or to support distributed termination detection via the verification of application level stable predicates as discussed in [7].

ROOT-Sim embeds advanced mechanisms for transparently supporting non-incremental log/restore of simulation object states scattered across non-contiguous, dynamically allocated memory chunks [35], 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 [22]. The dynamic memory map associated with whichever simulation object is controlled by ROOT-Sim via a simulation kernel level hook of standard services offered by the malloc library. In particular, each time a memory allocation/deallocation occurs, the hook intercepts the call and, depending on whether we were executing within application or kernel level contexts, different actions are taken. In case the execution was in kernel mode, then the actual service offered by the malloc library gets executed. Otherwise, the service call is redirected to the memory map manager [35], which serves dynamic memory allocation/release operations via a second level pre-reserving scheme (on top of the real malloc services). In this way, dynamic memory buffers destined to the use by whichever simulation object are mapped to blocks of contiguous virtual addresses so to maximize locality (which likely favors log/restore operations, implemented as pack/unpack to/from log buffers) and to minimize the size of second level meta-data used to track chunks currently allocated within the state layout.

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 [22]. 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. According to this design, the tracking mechanism for write operations operates with arbitrary granularity, thus the architecture is capable of identifying memory updates at the level of each single chunk dynamically allocated within the simulation object memory map. Actually, incremental logging has been enabled in ROOT-Sim even in cases where the memory map of any object gets altered via a third party library. In fact, beyond capturing any memory allocation/release, even occurring inside a library (via the aforementioned hook), standard libraries are wrapped in order to determine touched memory areas within the object state layout via the corresponding parameters ([2]).

In a very recent advance [36] we have further extended the instrumentation tool so to provide supports for coexistence of incremental and non-incremental logging. In particular, automatic ELF rewriting schemes have been used to transparently 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 using standard `ld` facilities. Further, 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. In this way, once the executable is finally built and run, a kernel level run-time switch between the two different log modes simply involves reassigning the callbacks' pointers within the ROOT-Sim API to the entry point symbol associated with the corre-

[2] In this way third party libraries are not instrumented, thus remaining available for conventional use to the simulation kernel.

sponding version of the dual-coded application level modules. Also, each log mode is supported according to a highly optimized run-time scheme, where memory update tracking gets completely excluded (thus avoiding at all the associated costs) whenever the kernel decides to switch to the non-incremental log mode. Such a dual-coding mechanism was complemented via analytical performance models used to built an autonomic manager able to dynamically switch to the well suited log mode (and correspondingly tune the log interval) [36]. The autonomic manager mandatorily requires monitoring fine-grain latency parameters (e.g., the latency for processing each single event, or for taking each single incremental/full log) appearing within the performance models. This has been done by relying on the `gettimeofday` service, offering access to elapsed-time timers (not real CPU usage timer). Such a solution results therefore tailored for scenarios where the computing platform is reserved for the optimistic simulation run.

3.4.1 Implementation Details

We have added data structures and control logic exactly implementing the evolutionary log/restore algorithm on top of such an optimized dual-coding architecture, thus inheriting that, whichever log mode is evolutionary selected for a given execution phase, it is run in a highly optimized manner. On the other hand, a change in the design direction has been adopted for what concerns the supports for the evaluation of the reward function (namely the throughput of committed events y_i on each kernel). In particular, we decided to rely on real CPU usage, instead of elapsed-time timers, since this approach allows employing the algorithm also in scenarios where the computing platform is shared among multiple applications, thus widening its applicability compared to the autonomic approach provided in [36]. Mainstream examples of those kind of shared environments are related to Cloud Computing technology, where virtualized infrastructures possibly devoted to the optimistic simulation run are in their turn hosted (possibly according to sharing policies) by a real computing environment. To capture real CPU usage information we have exploited the `getrusage` service, which belongs to the standard API provided by POSIX compliant operating systems. This service is not able to capture CPU usage with granularity at the level of microseconds, since it relies on software level times maintained by POSIX compliant kernels. On the other hand, GVT recalculation in optimistic simulation platforms typically occurs according to periods on the order of tens of milliseconds, or even seconds, in order to keep the overhead of the GVT protocol low. Hence, real CPU usage for intervals with granularity on the order of at least tens of milliseconds can be safely catched via such a standard operating systems service. This provides accuracy to how in our implementation the fitness function (i.e., the reward y_i) gets evaluated for a give genotype configuration.

As a last note, the extension of the memory map manager presented in [22], tailored to support incremental logging, is based on an approach that allows state recovery operations by accessing log chains that can contain both full and incremental logs according to an interleaved scheme. Also, even when operating according to the incremental mode, some full logs can be sparsely forced to optimize recovery operations and, more important, to enable effective fossil collection operations upon GVT calculations. By this design approach, any switch between full and incremental logging by our evolutionary algorithm only entails setting the (per simulation object) log-mode flag, and updating the aforementioned callback pointers to be used as application access points within the dual-coding scheme. Overall, the underlying state recovery logic transparently supports correct state reconstruction even in case a mix of incremental and full logs needs to be accessed within the log queue, without the need for any additional synchronization aimed at explicitly avoiding the possibility to rollback to a simulation time when a different logging approach was being used for the corresponding forward computation phase.

4. EXPERIMENTAL RESULTS

As test-beds for the experimental study we have used two different wireless-system simulation models adhering to GSM technology. The first model refers to GSM coverage along a ring highway while the second model refers to wireless coverage of a square urban area. 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 power regulation model has been implemented according to the results in [16]. 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 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 adjacent cell. 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. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a meteorological model defining climatic conditions (and related variations). The climatic model accounts for variations of the climatic conditions (e.g., the current wind speed) with a minimum time granularity of ten seconds. The employed simulation models have been developed for execution on top of the ROOT-Sim environment in a way that each simulation object models a single GSM cell. The `ProcessEvent` callback issued by the ROOT-Sim kernel involves therefore the update of individual cells' states, and cross-object events are essentially related to hand-offs between different cells.

The platform 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 has 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 the experimental study we have measured the following two key performance indicators: (A) The variation of the amount of committed events per wall-clock-time second (which we refer to as 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. (B) The cumulated amount of committed events vs the wall-clock-time for the simulation run. This parameter expresses the ability of each log/restore configuration to commit events (and hence to carry out useful simulation work) while the 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. Actually, the reported event rate is sampled over a single run (for easiness of showing) while the number of cumulated events (throughput) is the average over 10 runs, for which we also report the standard deviation. Also, we have set to the fixed value of 5% the threshold determining the changes in the event throughput triggering the different evolutionary step in the proposed algorithm. A study on the sensibility vs variations of this threshold, and on the possibility to adapt the threshold value dynamically is demanded to future work.

We report plots for a comparison of the evolutionary algorithm provided in this article vs the autonomic solution in [36] (based on analytical modeling of the direct effects of log/restore tasks). This solution has been already shown to outperform approaches alternatively based on the use of incremental or non-incremental log/restore, but not supporting optimized coexistence of the two different modes. In other words, we compare the evolutionary approach against an highly optimized log/restore solution. However, to show that the experimental study has been carried out in the context of a competitive parallel execution scenario, we also report performance data related to serial execution of the simulation models based on the calendar queue scheduler [28].

Figures 2(a) and 2(b) show the results for the GSM coverage network along the ring highway around the city of Rome, namely the Grande Raccordo Anulare (GRA). 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. Hence, for this model, we have 8 simulation objects, each hosted by a different kernel instance running on a dedicated core on the underlying machine. As in the actual system organization supported by in the charge Telecommunication Company, each cell hosts 1000 radio channels [2].

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 [1]. Simulated night-time periods are characterized by near-zero utilization factors (correspondingly less than 800 vehicles run along the highway 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 highway 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 55% in rush periods considering an average call duration of 120 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 highway could not be directly used due to privacy issues.

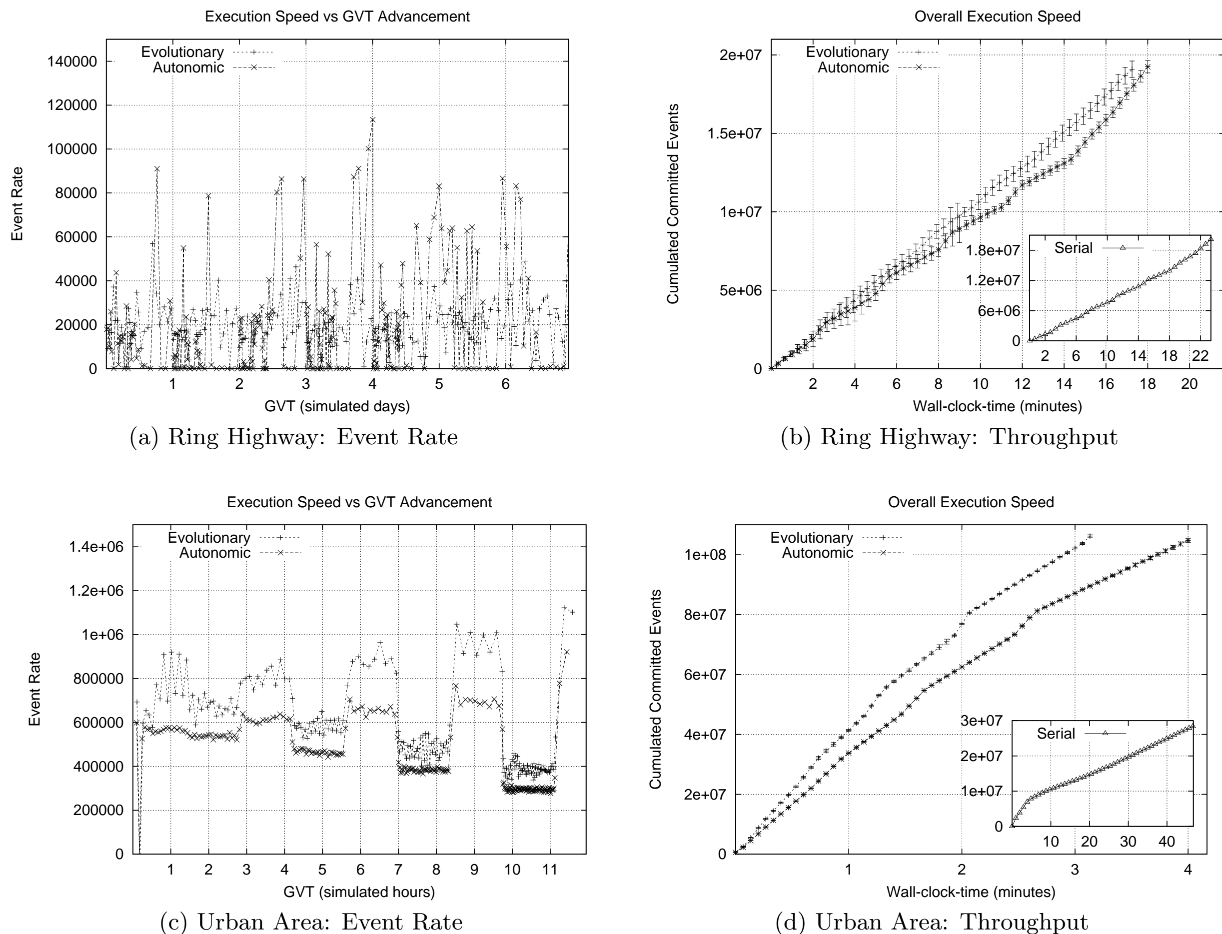
By the results, we see that, the autonomic system exhibits a slightly better perfomanche during nigh-time periods and weekends. This is due to the fact that when the system is lightly loaded, there is reduced competition for the cache against the underlying simulation kernel (we recall that for this test-bed we have a single object mapped to a kernel instance), since the object modeling the cell exhibits a reduced size state due to the minimal number of records allocated for ongoing calls. In this scenario, the autonomic system is capable of depicting with a higher precision application-level dynamics, and at the same time the effects of hidden dynamics (such as locality variations due to interactions between application and kernel layers) are not so relevant.

This is not the case for day-time periods, where the size of the simulation objects' states can grow significantly (especially for rush hours), and the access/update pattern of the state upon the occurrence of the events, together with housekeeping operations by the kernel (e.g., log operations) may produce secondary locality-variation effects which are indirectly captured by the evolutionary algorithm.

Furthermore, in the cumulated-committed-events plot we can see that, although the autonomic configuration produces a curve with a sheer slope during the night and weekend periods, the evolutionary algorithm produces a better throughput during the overall execution, since the workload associated with day-time periods exhibits runtime dynamics which, as said above, cannot be completely captured by the autonomic system.

In Figures 2(c) and 2(d) we present the event rate and the number of cumulated events (throughput) for the second test-bed application, which simulates 2048 GSM micro-cells (showing classical hexagonal shape), still managing 1000 channels, serving a square urban area. The corresponding 2048 simulation objects are equally distributed across 24 instances of the simulation kernel, each one running on one of the 24 cores available within the computing platform. The application layer simulates a variable traffic workload, which is split into several phases of a same duration, to each of whom a particular system's average workload is associated. The workload, expressed in terms of arrival rate of calls to each cell, varies in between a minimum and maximum value which lead, respectively, to about 5% and 25% of the highest workload sustainable by the telecommunication infrastructure (in terms of channel occupancy). Phases are distinguished between even and odd ones, the former having a workload which is always increased towards the maximum value, the latter having a workload which is decreased towards the minimum. Call duration and hand-off intervals are selected according to an exponential distribution [4], whose mean value is, respectively, 120 seconds and 300 seconds.

The results show that, although both the autonomic and the evolutionary approaches present an event rate which, as natural, depend on the execution dynamics characterizing specific phases, the evolutionary algorithm provides a better throughput. In fact, in Figure 2(d) we see that the curve associated with the evolutionary algorithm presents a slope steeper than the autonomic, producing an enhancement in the performance of about 20%. We argue that this gain is related exactly to the fact that optimization of the log/restore parameters by relying on performance models



(a) Ring Highway: Event Rate

(b) Ring Highway: Throughput

(c) Urban Area: Event Rate

(d) Urban Area: Throughput

Figure 2: Experimental results.

only expressing direct effects of the associated operations (as the autonomic scheme does) does not capture significant aspects, which acquire further relevance for this test-bed application (compared to the previous test-bed). In fact, the definitely larger amount of simulation objects hosted by each kernel instance likely induces, e.g., higher contention within the caching hierarchy. This occurs not only in relation to the larger amount of virtual addresses accessed (in phase interleaved mode) while the hosted simulation objects are alternatively dispatched for event execution on a same kernel instance (compared to the case in which a single object is hosted), but also because housekeeping operations by the kernel need to span a larger amount of virtual addresses due to, e.g., event lists and log queues to be maintained.

Further, we can see that the fluctuations in the evolutionary algorithm event rate, due to the algorithm intrinsic of searching for a better solution (we recall that the algorithm always tries to explore the search space to determine whether the environmental settings underwent some changes), never produce an extreme worsening in the event rate itself (even when a currently optimum value has been reached), thus keeping its mean value above the one by the autonomic scheme. In addition, we want to emphasize how this behavior allows the algorithm to take into account environmental changes in a fast way, causing its learning curve to be really sharp, and allowing the system to cope with environmental changes in a way which is definitely comparable to the autonomic's, which is on the contrary based on a closed-formula and on time measurements, thus providing a mode switch as fast as possible.

As for the comparison with serial execution, we have reported a zoomed view over the amount of committed events achieved by the serial run in the proximity of the wall-clock-time interval where the parallel runs get completed. By the results we see that the speedup by the parallel run is on the order of 1.5 for the smaller ring-highway model (with 8 simulation objects running on 8 cores), and is on the order of 30 (i.e., super-linear) for the larger urban area model (with 2048 simulation objects running on 24 cores).

The above discussed results provide a view of the overall performance achievable by our proposal. Given that the autonomic model was strongly optimized, since it was based on analytical and dynamic (re-)selection of well suited log intervals, and given that it had been proven to outperform classical checkpointing schemes, this is a significant result.

5. CONCLUSIONS

In this paper we have presented the design and implementation of a log/restore layer for optimistic simulation systems based on an innovative evolutionary algorithm. The algorithm constantly explores the search space of possible configurations of log intervals and log modes (incremental

vs non-incremental), looking for an effective approximation of one element of the Pareto set, which is expected to provide the optimized performance. Further, the effectiveness of the approach has been tested with real-world case studies related to wireless connectivity along a ring highway, and on a square urban area.

6. REFERENCES

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

[2] Private communication.

[3] T. Bäck, U. Hammel, and H.-P. Schwefel. Evolutionary computation: Comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1:3–17, 1997.

[4] A. Boukerche, S. K. Das, A. Fabbri, and O. Yildz. Exploiting model independence for parallel PCS network simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 166–173. IEEE Computer Society, May 1999.

[5] C. A. C. Coello, D. A. Van Veldhuizen, and G. B. Lamon. *Evolutionary Algorithms for Solving Multi-Objective Problems.* Kluwer, 2002.

[6] V. Cortellessa and F. Quaglia. A checkpointing-recovery scheme for time warp parallel simulation. *Parallel Computing*, 27(9):1227–1252, 2001.

[7] 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.

[8] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms.* Wiley, 2001.

[9] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: Nsga-ii. In *PPSN VI: Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, pages 849–858, London, UK, 2000. Springer-Verlag.

[10] 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*, pages 50–58. IEEE Computer Society, June 1995.

[11] M. P. Fourman. Compaction of symbolic layout using genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 141–153, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc.

[12] 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, 1997.

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

[14] http://www.dis.uniroma1.it/∼quaglia/software/ROOT-Sim.

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

[16] 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.

[17] J. Knowles and D. Corne. The Pareto archived evolution strategy: a new baseline algorithm for Pareto multiobjective optimisation. volume 1, page 105 Vol. 1, 1999.

[18] F. Kursawe. A variant of evolution strategies for vector optimization. In *PPSN I: Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, pages 193–197, London, UK, 1991. Springer-Verlag.

[19] R. A. Meyer, J. Martin, and R. Bagrodia. Slow memory: the rising cost of optimism. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS 2000*, 2000.

[20] A. Park and R. Fujimoto. Optimistic parallel simulation over public resource-computing infrastructures and desktop grids. In *Proc. of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications*, 2008.

[21] G. T. Parks and I. Miller. Selective breeding in a multiobjective genetic algorithm. In *PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, pages 250–259, London, UK, 1998. Springer-Verlag.

[22] 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 2009 ACM/IEEE/SCS 23rd Workshop on Parallel and Distributed Simulation (PADS 2009.*

[23] K. S. Perumalla. μ-sik: A micro-kernel for parallel/distributed simulation systems. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 59–68, Washington, DC, USA, 2005. IEEE Computer Society.

[24] B. R. Preiss, W. M. Loucks, and I. 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.

[25] 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.

[26] 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, June 2003.

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

[28] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation*, 7(2):157–209, 1997.

[29] R. Rönngren, 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.

[30] 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, June 2005.

[31] 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, Oct. 2005.

[32] J. D. Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 93–100, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc.

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

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

[35] R. Toccaceli and F. Quaglia. DyMeLoR: Dynamic memory logger and restorer library for optimistic simulation objects with generic memory layout. In *PADS '08: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 163–172, Rome, Italy, 2008. IEEE Computer Society.

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

[37] 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.

[38] E. Zitzler, K. Deb, and L. Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 8:173–195, 2000.

[39] E. Zitzler, M. Laumanns, and L. Thiele. Spea2: Improving the strength Pareto evolutionary algorithm. Technical report, 2001.

[40] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach, 1999.