Abstract—Parallel Discrete Event Simulation (PDES) is a widespread technique used to perform real-world models' simulations efficiently. In particular, the optimistic Time Warp synchronization protocol, based on rollback recovery to maintain causality, has been shown to likely favor speedup in general application/architectural contexts. According to the PDES paradigm, with Time Warp the simulation model is partitioned into several distinct Logical Processes (LPs) each one representing a portion of the global model. Traditionally, LPs store their execution information into disjoint simulations states, forcing events exchange to communicate data between each other. In this work we propose the design and implementation of an extension to the traditional Time Warp protocol, targeted at shared-memory/multicore machines, allowing LPs to share parts of their simulation states by using global variables. In order to preserve optimism's intrinsic properties, global variables are transparently mapped to multi-version ones, in order to avoid any form of safety predicate verification upon updates. Execution's consistency is ensured via the introduction of a new rollback scheme which is triggered upon the detection of an incorrect global variable's read. At the same time, efficiency in the execution is guaranteed by the exploitation of non-blocking algorithms in order to manage the multi-version variables' lists. Furthermore, our proposal is integrated with the simulation model's code through software instrumentation, in order to allow the application-level programmer to avoid using any specific API to mark or to inform the simulation kernel of updates to global variables. Thus we support full transparency. An assessment of our proposal, comparing it with a traditional message-passing approach, namely the Time Warp protocol, has shown not to suffer (in terms of amount of rollback) from non-minimal message delivery latency. The latter feature makes it suited for a wide variety of computing platforms, including large scale traditional GRID systems and even desktop GRID environments.

On the other hand, supporting the Time Warp protocol, while still guaranteeing a simple and flexible programming model for the simulation application layer, is not trivial. In particular, being Time Warp based on concepts related to state recoverability, the level of transparency towards the application programmers depends on the extent and the mode according to which state recoverability functionalities operate within the Time Warp platform. Recent achievements along this direction (see, e.g., [4]) have enabled fully transparent and performance optimized recoverability via state logs for LPs making use of dynamic memory for the representation of their states, and possibly relying on third party libraries (thus software external to the application layer) for performing state updates during event processing. At the same time, alternative attempts to the semi-automated generation of reverse code for backward computation based state reconstruction have been also presented (see, e.g., [5]). Such kind of approaches extremely simplify the job of the application programmers since no state-management task in relation to synchronization (e.g. state-log tasks) requires to be implemented at the application level.

Actually, most of the solutions tackling transparency have been oriented to the original definition of the Time Warp protocol [2], where the LPs' states are assumed to be disjoint. Hence, according to this definition, each LP is only allowed to modify its private state variables upon processing new events, and the interactions (namely inter-dependencies) across LPs are only allowed to be instantiated via cross-LP scheduling of simulation events. On the other hand, having different LPs sharing (at least a portion of) the state of the simulation model may result in a more flexible paradigm, whose relevance has...
been fully recognized as a crucial issue in the development of parallel simulation applications [6], [7].

In this article we tackle the issue of transparently and efficiently supporting shared-state in optimistic simulation systems run on top of shared-memory/multi-core machines, by enabling the application programmer to access within the event processing routine both the private state of the LP and a global portion of the state, whose instance is represented by the value of global variables admitted within the application level code. Overall, with our solution the programmer is allowed to rely on the heap for allocating/deallocating memory chunks belonging to the private state of each LP, as already supported via the approach in, e.g., [4], while also being able to rely on global variables for the shared portion of the state thanks to the innovative solution we provide in this paper.

We implemented a fully featured shared-state management system targeted at IA-32/x86-64 architectures and ELF executables. Also, we have integrated such system within the open source ROOT-Sim (ROme OpTimistic Simulator) package [8] which implements an optimistic run-time environment supporting ANSI-C compliant application level software implementing the LPs’ logic in the form of event-handlers (as typical of most of the existing PDES platforms, such as [9], [10]). Thanks to the integration of our shared-state management system, these event handlers have now the possibility to transparently manage global variables, with related synchronization operations (e.g. rollback of the values of the variables and related propagation of causality errors) also carried out in a totally transparent way to the application programmer.

In order to provide efficient supports for the management of shared-state variables, in terms of both forward and backward computation, our proposal relies on an application transparent multi-version scheme based on non-blocking access/update operations. This allows improving the level of parallelism when the shared-state is accessed by multiple LPs concurrently scheduled on different CPU-cores, namely by different simulation kernel instances.

The results of an experimental assessment of the shared-state management architecture are also reported for the case of a wireless system simulation application run on top of ROOT-Sim on an HP proliant server equipped with 32 CPU-cores and 32 GB of RAM memory.

The remainder of this paper is structured as follows. In Section II we discuss related work. The shared-state management architecture is presented in Section III. A proof of correctness of our approach is provided in Section IV. Section V presents experimental data aimed at assessing the pragmatical viability of our proposal.

II. RELATED WORK

The work in [11] discusses how, in the context of optimistic simulation, state sharing is an orthogonal property to state saving, and shows that since any simulation system already supports interaction, sharing might be emulated by using a separate LP hosting the shared data and acting as a centralized server. To tackle performance issues, the work proposes to modify the rollback behavior of this special LP by introducing the notion of version records. This is an approach similar to the one proposed in [7], where a theoretical presentation of algorithms to implement a Distributed Shared Memory mechanism is presented in terms of protocols to keep replicated instances of a variable coherent. In particular, one of the provided algorithms proposes to realize variables as multi-version lists where write operations install new version nodes and read operations find the most suitable version. Although this approach shows similarities to ours, read and write operations are mapped to message passing primitives, which is instead not the case for our proposal. This places a hard burden on the centralized node(s), which in the case of a simulation model performing frequent read/write operations on shared variables can produce a non-sustainable overhead. Additionally, these approaches are strongly oriented to distributed simulation environments, while we target the trend of shared-memory/multi-core machines.

A further enhancement has been presented in [12], where the notion of state query is introduced. An LP needing a portion of the state which belongs to a different LP can issue a query message to it, and wait for a reply containing the suitable value. In case this value is later detected to be no longer valid, an anti-message (1) is sent so to invalidate the query. Again, this approach relies on message passing, and is not transparent to the application programmer.

The work in [13] proposes to integrate supports for shared-state in terms of global variables, by basing the architecture on [14]. Although this proposal supports in-place read/write operations as we do (i.e., LPs access directly the only copy of the data, avoiding a commit phase at the end of the execution of an event), they provide no transparency, as the application-level code must explicitly register an LP as a reader/writer on the shared variables, and furthermore the synchronization between LPs while accessing shared variables is based on locks, while we provide a non-blocking implementation.

In the context of the High-Level-Architecture (HLA), proposals for supporting shared-state can be found in [15], [16]. These proposals are again targeted at a distributed environment, since they are based on a middleware component which relies on a timestamp-ordering approach for implementing a request-reply protocol. Additionally, these approaches are targeted at the conservative synchronization protocol, where there is no need to detect and handle causality violations, while we target optimistic synchronization.

The Software Transactional Memory (STM) paradigm [17] allows multiple threads to access global information while ensuring consistency wrt concurrent accesses. The main differences between multi-version-based STMs [18] and our proposal lie in that (i) STM does not enforce transparency wrt the application level programmer, since transactions must be explicitly marked; (ii) writes operations do not work in-place, i.e., data updating is performed on separate buffers (i.e.,

1In Time Warp, an anti-message s a negative copy used to annihilate a previously sent message, namely an already scheduled event. Anti-messages are used to propagate the effects of causality errors across the LPs by retracting events scheduled during the causal inconsistent portion of the simulation.
CAS any number of processes. In our implementation we rely on Read and Write, is both necessary and sufficient. A universal synchronization primitive that is universal, in conjunction with most interesting data types (linked lists among them), a Herlihy [22] proved that for non-blocking implementations [20]. Lamport [21] gave the first non-blocking algorithm for this paradigm, so that our proposal is set aside this one.

As for non-blocking algorithms, avoiding mutual exclusion has been considered a benefit since the early 1970’s [20]. Lamport [21] gave the first non-blocking algorithm for the problem of a single-writer/multi-reader shared variable. Herlihy [22] proved that for non-blocking implementations of most interesting data types (linked lists among them), a synchronization primitive that is universal, in conjunction with Read and Write, is both necessary and sufficient. A universal primitive is one that can solve the consensus problem [23] for any number of processes. In our implementation we rely on Compare&Swap (CAS), which is a universal primitive. The work in [24] presents the implementation of a non-blocking linked list, which we have readapted for our own purposes.

A subtle problem associated with most lock-free algorithms is the ABA problem. It was first reported in association with the introduction of the CAS instruction on the IBM System 370 [25]. It occurs when a thread T1 reads a value A from a shared object and then an interrupting thread T2 modifies the value of the shared object from A to B and then back to A. When T1 resumes, it erroneously assumes that the object has not been modified. Given such behavior, there is a serious risk that T2’s execution is going to violate the correctness of the object’s semantic. Practical solutions to the ABA problem include the use of hazard pointers [26] or the association of a version counter to each element in platforms supporting a double-word compare-and-swap primitive (CAS2) such as IA-32 [27]. We explicitly rely on the latter solution to avoid the ABA problem in our non-blocking implementation.

III. SHARED-STATE MANAGEMENT ARCHITECTURE

Being our approach targeted at multi-core machines, in our Shared-State Management Subsystem (SSMS) we have explicitly decided to rely on shared memory for keeping the current state of global variables. This allows a fast access to the data structures, i.e., no message passing is needed across distinct simulation kernel instances, although requiring some sort of synchronization between instances in order to ensure correctness. To leverage the synchronization burden, we have decided to implement data structures’ accesses as non-blocking algorithms [20], which are expected to ensure better performance than locking ones when accesses are statistically spread across the various portions of the data.

To ease the application-level programmer, we have addressed transparency via software instrumentation, so that no additional API or code construct should be used to notify SSMS of accesses to global variables.

In this section, we provide a detailed description of the architectural choices and the motivations behind each key component of SSMS. Additionally, we will discuss the reduced set of APIs provided by SSMS, which allow a fast integration into any optimistic simulation platform adhering to the optimistic synchronization protocol.

A. Read/Write Detection

In order to provide complete transparency to the application-level programmer, accesses in read/write mode to global variables must be explicitly intercepted. To this end, we rely on instrumentation techniques aimed at modifying the actual instructions executed by software executables, without altering their actual semantics. In particular, in the work in [29] we presented a versatile Instrumentation Tool (IT) targeted at IA-32/x86-64 instruction sets [27], [30] and ELF executables [31], on GNU/Linux Operating Systems. By relying on IT, at compile time the application-level instruction code (i.e., the assembly bytestream) is modified in order to replace operations loading data to and from memory with actual function calls which are the entry points of our SSMS. These entry points are associated with the following APIs provided by SSMS: write_global_variable(void *orig_addr, time_type lvt, ...) and void *read_global_variable(void *orig_addr, time_type my_lvt). They allow accessing the versions within the version lists for a given variable at a certain Logical-Virtual-Time (LVT).

We have identified two main groups of instructions/code blocks which have to be handled within the application-level assembly code. First, in IA-32 simple load and store operations are identified by mov instructions. Whenever IT’s parser identifies a mov instruction, it is analyzed in order to determine whether it is targeting memory as a source or destination operand, and a call to write_global_variable or read_global_variable is replaced accordingly. When the mov instruction involves a load operation from memory, an additional postamble to the function call is placed, in order to have the actual value returned by read_global_variable placed into the correct CPU register where the application-level software is expecting the value to be found.

Second, the IA-32 instruction set provides more complex instructions which allow an executable to efficiently modify memory areas in-place. As a relevant example, we propose instructions like ADD m32, r32 or INC m32. In this case, IT replaces the instructions with a block of instructions, entailing a couple of calls to the SSMS’s read and write APIs, and re-implementing the same logic with several CPU
instructions. This implementation of course adds some overhead, nevertheless it allows to integrate our SSMS completely transparently wrt the application-level programmer.

High-level programming languages allow to access memory objects in a non-direct way, namely through the use of pointers. Since IT works at compile time, it is not possible to statically determine whether a pointer will target a global variable or not. To cope with this issue, we use IT to instrument any mov instruction which can handle pointers through a call to a monitor function which fastly determines if a pointer targets a global variable. In particular, at compile time, via the usage of a custom ld-based linker script we insert symbols called _bss_start, _bss_end, _data_start, _data_end, within the application-level ELF executable, which mark off the area containing global variables. Upon a call to the monitor routine, a fast check on these boundaries is performed. If a pointer falls within this area, the operation is redirected to SSMS, on the other hand the original mov instruction is executed.

As a last note, Intel’s instruction set provides string instructions which allow to perform operations on memory buffers instead of single memory locations. In particular, movs and stos instructions allow the program to copy or modify large buffers at once. In order to cope with the presence of these complex instructions, SSMS provides two additional APIs, namely copy_buffer() and set_buffer() which simulate the execution of these operations on version lists if they are found to target global variables (e.g., global arrays). Otherwise, they just execute the original mov or stos operations. Therefore, at compile time, IT replaces every string operation involving memory update with a function call to these APIs, accordingly.

The last operation we perform at compile time is the inspection of the application-level ELF object file in order to extract information concerning global variables. In particular, by exploring the application object file we extract from the symbol table .symtab all the STT_OBJECT / STT_COMMON symbols and store their name, address and size in a text file which will be later used at startup time for setting up the version lists. In this way, by exploiting the (name, address, size) tuple, we are able to transparently identify any access to global variables which will be likely used by the application-level code during the execution of the simulation model, allowing the programmer to rely on the complete set of constructs provided by ANSI-C. We note that, although there will be more instances of the simulation kernel running the application-level code, a global variables’ address is a common information shared among the instances, as long as its virtual address will be the same and is cabled into the executable.

B. Accounting for Third-Party Libraries

The possibility to rely on third-party libraries depends on whether they will be invoked on global variables or not. We have explicitly addressed the case of read/write operations performed by third-party software, just focusing on stdlib. Specifically, SSMS provides a set of function wrappers for all those functions which produce in-memory accesses by

the application-level software through pointers passing. The wrappers simply check whether global variables are involved in the operation. In this case, operations are redirected to SSMS APIs for accessing version lists. Otherwise, the original stdlib functions are called.

At the time this paper has been written, we are currently working on techniques for allowing the user to automatically rely on any third-party library. This task entails evaluating which can be the most effective trade-off between user-level transparency and versatility wrt the set of libraries which can be supported.

C. Memory Map and Version Lists

As hinted before, SSMS explicitly targets shared-memory/multi-core machines. In order to significantly enhance performance, we have decided to avoid requesting to the underlying operating system shared memory segments on-demand, whenever SSMS needs to install some data structure. On the other hand, at simulation startup the master kernel installs a large shared memory segment, and broadcasts to other kernel instances its id. The shared segment is partitioned according to the definition of the following structure:

typedef struct _globvar_shmem {
    int num_vars;
    globvar_info variables[MAX_GLOBVARS];
    volatile int first_node_free;
    globvar_node versions[MAX_VERSIONS];
    atomic_t correlation_m[MAX_GLOBVARS][MAX_GLOBVARS];
    atomic_t aggregation_m[MAX_GLOBVARS][MAX_GLOBVARS];
    time_type read_list[];
} globvar_shmem;

In particular, the shared memory segment is divided into several fixed-sized portions. One portion, namely variables, is an array which is used to manage global variables. Upon initialization of SSMS, the configuration text file described in Section III-A is loaded and parsed. The field num_vars is used to keep track of how many variables are actually handled, and for each of them an entry in the variables array is populated. To allow a fast retrieval of the global variables, we use a fast hash function to determine which entry in the variables array will store the information associated with a specific variable. In particular, the position in the array is determined with a fast bitwise operation — namely, address & (~(-MAX_GLOBVARS)) — since MAX_GLOBVARS is set to be a power of two. In case collisions are found, separate chaining is used as a means for finding a free place.

Each entry in the variables array is structured as:

typedef struct _globvar_info {
    void *orig_addr;
    unsigned short int size;
    long long head;
    long long tail;
} globvar_info;

orig_address stores the global variable’s original address, which is used as hash table’s key; size describes which is the size (in bytes) of the global variable.

Since we are preallocating shared memory, version lists must be implemented using nodes scattered around the pre-
allocated segment. In particular, versions is an array of fixed-sized nodes which can be used for any list, and head and tail are indices within this array, which is composed of entries structured as follows:

typedef struct _globvar_node {
  volatile int alloc;
  time_type lvt;
  unsigned char value[MAX_BUFF];
  spinlock_t read_list_spinlock;
  long long next;
} globvar_node;

where lvt is the logical time associated with the version, value is the global variable’s value, and next is used to identify which is the following node in the list. A node can therefore be seen as a snapshot of the state of a single global variable at a certain LVT. In Figure 1 we provide a complete picture of the preallocated memory map.

Node versions' entries can belong to any list, and given that lists are accessed without the use of locks, a special allocation function must be used, ensuring that no two simulation kernel instances running concurrently are given the same entry for handling two different versions.

Algorithm 1 Shared Memory Allocation

1: procedure ALLOCATE
2:   \(m \leftarrow \text{generate\_mark}()\)
3:   \(slot \leftarrow \text{first\_node\_free}\)
4:   while true do
5:     \(\text{all} \leftarrow \text{vers}[slot].\text{alloc}\);
6:     if all \(\lor \neg\) CAS(vers[slot], alloc, all, m) then
7:       \(slot \leftarrow \text{next slot in circular policy}\)
8:   else
9:     break
10: end if
11: end while
12: update first_node_free
13: return slot;
14: end procedure

The ALLOCATE pseudocode is given in Algorithm 1. In order to allow concurrent accesses, it relies on CAS (\(^3\)), which allows to update involved data only if no other process has updated the data in the meanwhile. The globvar_shmem data structure holds in the first_node_free the value of the first element of the versions array to start trying to allocate from. Its manipulation is based on the classical algorithm used by the LINUX kernel for managing the bitmap of file descriptors associated with a process. Specifically, it is always increased upon allocation, and gets decreased in case an entry is released having index less than the first chunk currently available within that block. Starting from that slot, a kernel instance tries to allocate a node by storing via a CAS operation a non-zero value within the alloc field of globvar_node, which tells whether a node is currently in use. In case the CAS fails, the next node in the array is selected and the procedure is repeated, until it eventually succeeds (\(^3\)). The companion function RELEASE is much simpler, as it only entails resetting the alloc and updating first_node_free, via an atomic_set call.

In order to cope with the ABA problem [25], we have explicitly decided to consider a node allocated if the alloc field is non-zero. In particular, we store into it a unique value every time a node is allocated, so that two allocations can be identified as different. The macro generate_mark produces an integer value which is composed of two short integers, one holding the unique id of a kernel instance and the other holding the value of a per-kernel counter which is incremented any time the macro is invoked (\(^4\)).

Once a node is allocated, it gets organized into a non-blocking linked list, which is implemented according to a modified version of the one proposed in [24]. Concurrent insertions are handled via the use of a single CAS operation, which is used to introduce the newly allocated node into the list by acting on the next field of the predecessor node. As for deletion, two CAS are used, one to mark the next field of the deleted node as logically deleted, and another to physically delete the node. We have slightly modified the algorithm in order to take into account our specific needs. In particular, the FIND-NODE procedure has been augmented in order to return the alloc field, to explicitly cope with the ABA problem, and the INSERT procedure does not fail if a node with the same key (i.e., LVT) already exists. In addition, we note that LPs are more likely to access versions associated with higher LVTs, since well partitioned/balanced optimistic simulations usually proceed relatively evenly. Therefore, we sort the versions in the lists in descending order, to avoid a complete scan of the list every time we want to find a node in it.

To avoid the ABA problem in linked lists, pointers (i.e. indices) to nodes are composed (every time they are updated) by a unique mark generated via the aforementioned macro generate_mark and the real index, allowing to capture the situation where two nodes are still adjacent but one was deallocated and then reallocated during the execution of the non-blocking algorithm by different kernel instances.

The operations performed on the versions lists are depicted

\(^2\)In particular, we rely on the IA-32’s cmpxchg. Throughout this paper we mention atomic operations which are implemented directly in assembly using native atomic instructions.

\(^3\)To check if the space is up, a counter of available free nodes is kept as well in shared memory, which is managed via an atomic_decrement operation.

\(^4\)generate_mark can of course return two equal values when the counter overflows, but this situation can happen after a significant simulation time, so we consider it to be statistically non-significant for the ABA problem.
in Figure 2.

D. Accessing Version Lists

The APIs offered by SSMS provide two main functions to access global variables, namely read_global_variable and write_global_variable, which we will refer to as READ and WRITE from now on.

**Algorithm 2 Global Variable Read**

1: procedure READ(addr, lvt)
2: slot ← hash table’s entry associated with addr
3: if slot ∈ AccessSet then
4: version ← AccessSet[slot]
5: else
6: while hasRead do
7: ⟨version, alloc⟩ ← FIND-NODE(slot, lvt)
8: AccessSet[slot] ← version
9: spin_lock(read_list_lock)
10: if alloc has been changed then
11: spin_unlock(read_list_lock)
12: continue
13: end if
14: add ⟨lp, lvt⟩ into ReadList
15: spin_unlock(read_list_lock)
16: hasRead ← true
17: end while
18: end if
19: return vers[version].value;
20: end procedure

READ operation’s pseudocode is provided in Algorithm 2. For efficiency reasons, before letting an LP execute a simulation event, SSMS sets up an AccessSet, i.e., a mapping between version nodes and variables. Whenever a variable is accessed for the first time, FIND-NODE(7) determines which is the most suitable version for the given LVT, and a couple ⟨slot, version⟩ is placed into AccessSet in order to speedup the retrieval of the version, avoiding the scan of the list upon subsequent accesses.

As for the WRITE operation, whose pseudocode is presented in Algorithm 3, its behavior is twofold depending on whether

in Figure 2.

**Algorithm 3 Global Variable Write**

1: procedure WRITE(addr, lvt, val)
2: slot ← hash table’s entry associated with addr
3: if slot ∈ AccessSet then
4: version ← AccessSet[slot]
5: vers[version].value ← val
6: else
7: version ← INSERT-VERSION(slot, lvt, val)
8: AccessSet[slot] ← version
9: end if
10: for all ⟨lp, lvt′⟩ ∈ ReadList s.t. lvt′ ≥ lvt do
11: send antimessage to lp
12: end for
13: end procedure

**Fig. 2: Non-Blocking Linked List Operations**

**Fig. 3: Occurrence of the Rollback Operation**

It’s invoked for the first time since the beginning of the current event’s execution. In particular, upon the first access on a variable, the AccessSet for that particular event is populated. Otherwise, a call to INSERT-VERSION is performed which, as stated in Section III-C, creates a new version. The second part of the READ operation entails checking the ReadList for ensuring consistency, as it will be clearly depicted in Section III-E.

E. Synchronization and Rollback Operations

In order to strengthen the optimism of our implementation, we allow interleaved reads and writes on a version list, and we explicitly avoid a version k installed at LVT t_k to invalidate every version j such that t_k < t_j. In fact, we note that consistency is violated only if, at LVT t_x an LP reads the version associated with LVT t_y such that t_y ≤ t_x, and at a certain point during the execution a new version node associated with LVT t_y ≤ t_x < t_v is installed.

This means that every process which reads a certain version node must leave a mark of that operation, i.e., visible reads [32] are enforced. In fact, as shown in Figure 3, we are interested in undoing only the events which read a version older than the new one which has just been inserted.

To this end, we augment the classical notion of rollback as presented by the Time Warp synchronization protocol, by sending a special anti-message to all the LPs which have read a so-defined causally inconsistent event after any write operation. This is reflected into Algorithms 2 and 3. In fact, in the READ operation, before returning the variable’s value, the

---

3We remind that FIND-NODE is a modified version of the one presented in [24]. For a detailed description of the procedure, we throw back to that work. In addition, we note that a version node is always available, even before any WRITE operation, since at startup the initial value of the global variable is placed into the version list.
tuple $\langle lp, lvt \rangle$ is inserted into the $ReadList$ for that particular version. This operation is included within a specially designed critical section to ensure consistency. In fact, a spinlock for that particular $ReadList$ is taken, ensuring that no other process will start the rollback operation while the $ReadList$ is being updated, as this would produce an invisible read. In addition, after the spinlock has been taken, a check on the variation of the $alloc$ field for that particular version is performed, so to avoid the ABA problem due to a critical race between the deallocation/allocation procedure and the $ReadList$ update. At the same time, at the end of the Write operation, the $ReadList$ of the left node is checked in order to find all the LPs which read the previous node’s value, while they were requesting a version at an LVT such that they should have read the one in the version which was just installed. Although the list is linked in only one direction, given the implementation of Find-Node, locating the previous node is immediate.

For performance reasons the $ReadList$ is implemented as a heap of given size. In particular, when allocating the shared memory segment at simulation startup, a portion of space is reserved for any $ReadList$ entries. Every node keeps an array of integers identifying an entry in this heap, which is handled for allocation/deallocation in a way similar to the version nodes’ one.

We note that another step must be undertaken in order to ensure correctness. In particular, whenever a special antimes-sage is received because of an inconsistent read, any version node installed due to that particular event must be removed. To this end, we augmented the concept of message queue and modified the Write function so that whenever a node is installed during the execution of an event, the message queue keeps track of this operation via a pointer to the node created during the event’s execution. In case a rollback operation entails the undoing of that event, the node is removed from the version list, and the $ReadList$ is scanned for sending antimesages to every LP which read that particular node.

F. Memory Recovery

In Time Warp, the notion of fossil collection is defined, i.e., the process of recovering memory by deleting simulation state snapshots which are no longer needed. In particular, at a periodic rate, the Global Virtual Time (GVT) is computed as the minimum timestamp of not yet processed events or in transit messages/antimesages in the whole simulation system. Since during the execution of an event an LP can schedule a new event at an LVT which is equal to, or greater than, the one associated with the event being executed, there cannot be a rollback operation involving a simulation state snapshot associated with a timestamp less than the GVT. Therefore, any snapshot belonging to a logical time window before the GVT can be discarded.

In our proposal, we extend the notion of fossil collection by defining the version list pruning. In particular, upon GVT computation, the version lists associated with global variables are scanned in order to find which is the first node $i$ stamped with $t_i \leq \text{GVT}$ and that node is selected as the barrier node. Any node marked with a timestamp $t_k < t_i$ is marked as free and removed from the list.

Since during the GVT operation there is no actual event processing, we note that the version list pruning is thread safe, and can therefore be executed efficiently, with no need to synchronize the access. In particular, the various lists are divided evenly across the various kernel instances, and each kernel performs the memory recovery executing in isolation. This choice provides a more efficient execution and still ensures correctness.

IV. Correctness of the Approach

The SSMS algorithm allows dispatched LPs to concurrently access global shared variables in an optimistic way and postpones synchronization among concurrent read/write operations executed on the shared-state only whenever a conflict materializes. Therefore the implemented concurrency control scheme maintains a high degree of parallelism by ensuring that: (i) the read/write operations executed by a committed event $e$ on the shared-state $S$ appear as they happened at same indivisible point in time associated with the logical virtual time $lvt_e$ in which $e$ has been processed; (ii) all the committed events execute the same operations and produce the same outcome as they were processed sequentially without violating logical virtual time advancement. For this reason, if we model an event $e$’s execution as an atomic transaction $T_e$ [33] to be considered committed whenever $e$ is committed according to the Time Warp algorithm (i.e., it can be established a GVT value $gvt$ such that each event $e'$ executed at a time $lvt_{e'} < gvt$ cannot be revoked anymore and $lvt_e < gvt$), we can adopt the serializability consistency criteria [33], [34] over the histories of the committed events as the target correctness criteria of the proposed solution.

Even if in practice SSMS behaves as an STM system, we have not designed it having in mind the typical correctness criteria guaranteed by STMs, namely opacity [35]. In fact, guaranteeing that every read operation always returns a value from a consistent state of the shared memory would not prevent an LP to see an inconsistent state of the simulation due to the operating rules proper of the Time Warp algorithm: an user model code may be executed using data arguments that are inconsistent with the logical state of the code [36].

Before showing the proof we formalize the concepts of history on committed events and operation. A history $H_{\text{gvt}}$ over a set $E$ of committed events $e$ at the GVT value $gvt$ consists of (i) a partial order of operations that reflect the write/read operations performed within $e$ on the simulation shared-state together with the begin (i.e., the invocation of $e$) and the complete (i.e., the commit of $e$), and (ii) the version order $\ll$ that specifies a total order on the object’s versions created by committed events. A write operation on an object $x$ issued by an event $e$ is denoted by $w_e(x_c)$ while a read operation on a version $x_{c'}$ of object $x$ is denoted by $r_e(x_{c'})$.

We can build a Direct Serialization Graph $DSG(H_{\text{gvt}}, \ll)$ over a history $H_{\text{gvt}}$ as stated in [34] in order to define serializability in terms of topological properties on that graph. In particular a graph $DSG(H_{\text{gvt}}, \ll)$ contains a node $N_e$ for
each committed event $e$ in $H_{gvt}$ and a directed edge $N_e \to N_{e'}$ for each pair of committed events $e, e'$ in $H_{gvt}$ such that one of the following dependencies occurs: (i) $e'$ directly read-depends on $e$ if there exists an object $x$ such that $e'$ executes a read $r(x_e)$; (ii) $e'$ directly write-depends on $e$ if there exists an object $x$ such that $e$ executes a write $w(x_e)$, $e'$ executes a write $w(x_{e'})$ and $x_{e'}$ immediately follows $x_e$ in the total order defined by $\ll$ on $x$; (iii) $e'$ directly anti-depends on $e$ if there exists an object $x$ and a committed event $e''$ such that $e$ executes a read $r(x_{e''})$, $e'$ executes a write $w(x_{e'})$ and $x_{e'}$ immediately follows $x_{e''}$ in the total order defined by $\ll$ on $x$. Then a history $H_{gvt}$ is serializable if the associated $DSG(H_{gvt}, \ll)$ does not contain oriented cycles as defined in [33].

Therefore the correctness proof of SSMS is formalized in the following Theorem:

**Theorem 1.** For each GVT value $gvt$ and for each history $H_{gvt}$ of committed events admitted by the SSMS algorithm then the $DSG(H_{gvt}, \ll)$ graph does not contain any oriented cycle.

**Proof:** We prove that the $DSG(H_{gvt}, \ll)$ does not contain any oriented cycle by showing that for each edge $N_e \to N_{e'}$, $lvt_e < lvt_{e'}$ always holds.

If an edge $N_e \to N_{e'}$ is in $DSG(H_{gvt}, \ll)$ we have to distinguish three cases:

1) $e'$ directly read-depends on $e$. In this case SSMS has performed a read operation on an object $x$ by returning the version $x_e$ having the greatest logical virtual time $lvt_e$ less than $lvt_{e'}$. Therefore $lvt_e < lvt_{e'}$.

2) $e'$ directly write-depends on $e$. $e'$ overwrites a value (by adding a new version $x_{e'}$) of an object $x$ already written by $e$. This is admitted only if $lvt_e < lvt_{e'}$.

3) $e'$ directly anti-depends on $e$. $e'$ adds a new version of an object $x$ after the version read by $e$. If $lvt_e \geq lvt_{e'}$ holds then SSMS forces a rollback for $e$. Since both $e$ and $e'$ are committed then $lvt_e < lvt_{e'}$.

By Theorem 1 follows that every committed history generated by SSMS does not violate serializability.

V. EXPERIMENTAL DATA

A. Test-Bed Application

The hardware architecture used for testing our proposal is a 64-bit NUMA machine, namely an HP Proliant server, equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 cores (for a total of 32 cores) that share a 12MB L3 cache (6 MB per each 4-cores set), and each core has a 512KB private L2 cache. The operating system is 64-bit Debian 6, with Linux Kernel version 2.6.32.5. The compiling and linking tools used are gcc 4.3.4 and binutils (as and ld) 2.20.0.

We have run our model on top of the ROme OpTimistic Simulator (ROOT-Sim) [8], which is an open-source, general-purpose simulation platform developed using C/POSIX technology, based on a simulation kernel layer that ultimately relies on MPI for data exchange across different kernel instances, and which adheres to the optimistic synchronization paradigm. Interaction with the application-level software is handled via a simple and reduced API, while LPs’ state management and recoverability is offered by DyMeLoR [29], [37], a memory manager which allows dynamic memory allocation and release by the application, performed via hooked standard malloc library calls, offering full transparency wrt memory management.

As a test-bed, we have used Personal Communications Service (PCS), a suite of differently parameterized simulation models of wireless communication systems adhering to GSM technology. The different parameterization entails variations of the transmission capabilities offered by each cell, as well as variations of the call arrival rate. In the employed simulation models, wireless communication channels are modeled in a high fidelity fashion via explicit simulation of power regulation/usage and interference/fading phenomena (implemented according to the results in [38]) on the basis of the current state of the corresponding cell (also expressed as a function of current meteorological conditions). Specifically, each modeled wireless 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 wireless cell, a call-setup record is instantiated via dynamically-allocated data structures, which gets linked to a list of already active records within that same cell. Each record gets released when the corresponding call ends or is handed-off towards an adjacent cell. In the latter case, a similar call-setup procedure is executed at the destination 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 ROOT-Sim in a way that each LP models a single wireless cell. Hence, the event-handler callback involves the update of individual cells’ states, and cross-LP events are essentially related to hand-offs between different cells.

Calls inter-arrival time is exponentially distributed, and average duration is set to 2 minutes. The expected rate for call inter-arrival has been set to achieve channel utilization factor on the order of 15%, while the residence time of an active device within a cell has a mean value of 5 min and follows the exponential distribution.

To evaluate the efficiency of our proposal, we have extended the simulation model having a set of global variables handling global statistics. In particular, upon each events execution the total number of calls, the total number of handoffs, and the global cumulated power is updated in the shared state. In
addition, we have re-implemented the model in order to have a centralized LP keeping in its disjoint simulation state the global attributes. Every LP willing to update a shared attribute issues a message request to the centralized LP, which in turn sends back the current value. Any update on the current value is then sent as another message to the centralized LP. In this scenario, every message exchanged with the centralized LP is marked with the same timestamp as the event’s which generated the read/write flow. Therefore, this baseline version of the benchmark is a simple zero-lookahead request-reply approach, which we do not expect to scale well with respect the number simulation kernel instances being run.

For the above scenario, we have run experiments with 64 wireless cells, modeled as hexagons covering a square region, each one managing 1000 wireless channels. We have measured the cumulated event rate (expressed as the amount of cumulated committed events per Wall-Clock-Time unit), which is a classical indicator of the speed of the optimistic simulation run.

B. Results

In Figure 4 we present the throughput associated with our proposed test-bed model run on top of 32 simulation kernel instances, each one running on a private CPU-core of our test machine. By the results, we can see that the execution of the simulation model relying on our SSMS provides a speedup in the order of 70%. In addition, we note that there is a tangible difference between the two curves’ trends. In fact, the throughput associated with the SSMS execution has a constant growth, which suggests a constant event commitment rate. On the other hand, the centralized-LP implementation’s slope shows fluctuations, which are related to the large amount of events associated with variables’ reads/updates which must be processed. Therefore, the number of committed events per GVT interval is not constant, due to the fact that the amount of workload processed by differentiated LPs is totally different and that the LVT of the LP keeping the shared state diverges from the other LPs’ one (this can entail a higher rollback probability), a scenario which is not present at all when relying on the multiversion lists in the shared memory version case.

At the same time, Figure 5 shows the total execution time of the simulation wrt the number of parallel simulation kernel instances on which the model is run. In addition to the set of experiments described before, we present also the curve associated with another implementation of the benchmark, where the shared attributes are kept in the disjoint LPs’ simulation states and are reduced at the end of the simulation. By the results, we can see that both the SSMS and the centralized-LP implementation suffer from some form of thrashing. In fact, the centralized-LP version provides a speed-down in the order of 100% when the model is parallelized on top of 4 parallel kernel instances, while SSMS shows the same behaviour starting from 8 parallel kernel instances. The version with no shared state shows a trend which is the one expected by a parallel simulator.

We note that in this configuration, the SSMS’s speedup wrt the centralized-LP is very large. Of course, the overhead in the centralized-LP case could be leveraged by having different LPs handle different variables, but this solution would not scale well wrt the size of the shared state in the simulation model.

Finally, we note that the simulation model used to assess the validity of our proposal is a worst case for our architecture, since at every event’s execution some updates on the global variables are performed, producing a large contention on the linked lists. A simulation model which relies on shared-state for synchronization rather than for global statistics would benefit much more from the proposed architecture.

VI. CONCLUSIONS

In this work we have presented the design/implementation of an efficient support to shared-state for optimistic simulation platforms, targeted at multi-core/shared-memory architectures. We have explicitly relaxed the classic simulation models’ state constraints, allowing the usage of portion of shared states among simulation objects, relying on global variables. We have provided the application-level programmer with full transparency, and we have exploited intrinsic characteristics of
our target architecture to enhance synchronization performance by relying on a non-blocking implementation.

REFERENCES


