Noname manuscript No.
(will be inserted by the editor)

Transparent Speculative Parallelization of Discrete Event Simulation Applications Using Global Variables

Alessandro Pellegrini · Sebastiano Peluso · Francesco Quaglia · Roberto Vitali

the date of receipt and acceptance should be inserted later

Abstract Parallelizing (compute intensive) Discrete Event Simulation (DES) applications is a classical approach for speeding up their execution and for making very large/complex simulation models tractable. This has been historically achieved via Parallel DES (PDES) techniques, which are based on partitioning the simulation model into distinct simulation objects (somehow resembling objects in classical object-oriented programming), whose states are disjoint, which are executed concurrently and rely on explicit event-exchange (or event-scheduling) primitives as the means to support mutual dependencies and notification of their state updates. With this approach, the application developer is necessarily forced to reason about state separation across the objects, thus being not allowed to rely on shared information, such as global variables, within the application code. This implicitly leads to the shift of the user-exposed programming model to one where sequential-style global variable accesses within the application code are not allowed. In this article we remove this limitation by providing support for managing global variables in the context of DES code developed in ANSI-C, which gets automatically parallelized. Particularly, we focus on speculative (also termed optimistic) PDES systems that run on top of multi-core machines, where simulation objects can concurrently process their events with no guarantee of causal consistency and actual violations of causality rules are recovered through rollback/recovery schemes. In compliance with the nature of speculative processing, in our proposal global variables are transparently mapped to multi-versions, so as to avoid any form of safety predicate verification upon their updates. Consistency is ensured via the introduction of a new rollback/recovery scheme based on detecting global variables' reads on non-correct versions. At the same time, efficiency in the execution is guaranteed by managing multi-version variables' lists via non-

A. Pellegrini · F. Quaglia · R. Vitali
Sapienza University of Rome, DIAG, Via Ariosto 25, 00185, Rome, Italy

S. Peluso
Virginia Tech, ECE Department, Blacksburg VA 24061, USA

blocking algorithms. Furthermore, the whole approach is fully transparent, being it based on automatized instrumentation of the application software (particularly ELF objects). Hence the programmer is exposed to the classical (and easy to code) sequential-style programming scheme while accessing any global variable. An experimental assessment of our proposal, based on a suite of case study applications, run on top of an off-the-shelf Linux machine equipped with 32 CPU-cores and 64GB of RAM, is also presented.

1 Introduction

Timeliness in the delivery of simulation outputs is an increasingly relevant issue to cope with, especially in contexts where simulation is exploited as a tool for decision making. For the case of Discrete Event Simulation (DES) models, which are particular instances of the class of event-based applications, performance issues have been traditionally targeted via the Parallel DES (PDES) paradigm [29], which is based on the partitioning of the simulation model into distinct simulation objects (SOBJs), to be executed concurrently. Each SOBJ models a portion of the simulated system, namely the modeled (virtual) world, and the interactions between the different portions of the model are captured by the exchange of *timestamped* events across the SOBJs. Thanks to concurrent SOBJs' execution, PDES allows exploiting the computing power offered by (high-end) parallel/distributed platforms in order to speedup model execution and to make very large and/or accurate models tractable.

A SOBJ is usually implemented as a set of data structures managed (e.g. updated) via proper callback functions, which resemble classical methods in object oriented programming. Also, the execution of any callback function is dispatched by an underlying simulation-platform (see, e.g., [13, 51]), namely the PDES run-time environment, and represents the processing of a simulation event at the target SOBJ. It may give rise to updates of the SOBJ's state and to the injection of additional timestamped events destined to whichever concurrent SOBJ.

Correctness of PDES runs is ensured via synchronization mechanisms, which are used to maintain causality patterns across simulation events and associated SOBJ state transitions. Although differentiated definitions of causal consistency have been devised in literature [12, 32, 64], the most widely exploited correctness criterion states that each SOBJ must process its input events (scheduled either by itself or by other SOBJs) in non-decreasing timestamp order. To support local timestamp ordering at the SOBJs, two synchronization approaches have been proposed: *conservative* and *optimistic*.

The conservative approach (see, e.g., [19]) avoids at all the possibility for any event to be executed out of timestamp order. This is achieved via block-until-safe policies that suspend processing activities at the SOBJ until the underlying run-time environment determines that the execution of its next pending event is coherent with logical-time ordering. On the other hand, the optimistic approach (see, e.g., [47]) allows the SOBJ to speculatively process its available input events under the assumption that timestamp-ordering will not be violated. If any violation is eventually detected, rollback recovery mechanisms bring the involved SOBJs (i.e. the ones affected by the causality error) back to a correct snapshot of their states, starting from which execution is resumed. Literature results show that the optimistic approach is prone to higher parallelism exploitation. This advantage is reflected also on the side of scalability, as shown in [16], where very large platforms (with thousands of CPU-cores) are employed for a comparative analysis of conservative vs optimistic approaches.

Motivated by such a great potential, a lot of effort has been spent in investigating how to support optimistic PDES runs by hiding state recoverability (namely synchronization) and other parallelism-related aspects (such as load balancing) to the programmer. Along the path of transparency of data structures (namely SOBJs' states) recoverability, we find various works, which are either based on classical log/restore facilities [62] or on reverse computing techniques [17]. Some of them also cope with dynamic memory based layouts of the SOBJs' state [58,60], and with the interaction of the application level software with third-party libraries (thus being able to recover side effects caused by common third party libraries on the SOBJ's state). These approaches offer support for the ease of programming, thanks to their ability to fully mask state recoverability aspects to the application code developer, who is therefore requested to only design/develop the code for forward (normal) mode execution of the SOBJs. More in detail, the ultimate target of the proposals coping with parallelism transparency, in combination with transparent speculative processing, is to provide the programmer with the illusion that SOBJs' callback procedures are executed sequentially, one a time on the basis of the order of events' timestamps. Hence, the programmer is exposed to a programming model based on the notion of objects and on sequentially-executed methods operating on the state of individual objects, which are allowed to always observe causally-consistent state information.

However, all these literature solutions have been tailored to the conventional (and classical) optimistic PDES scenario where no sharing of information is allowed across the different SOBJs within the same application. According to this programming model, each SOBJ is only allowed to modify its private state upon processing new events. This could be a limitation given that having different SOBJs sharing (at least a portion of) the state of the whole application state may result in a more flexible paradigm, whose relevance has been recognized as a crucial issue for the (ease of) development of DES code [31, 54]. Such a relevance is further motivated by the advent (and the large diffusion) of shared-memory parallel machines, like multi/many-core machines, which (as opposed to historically used distributed memory clusters) offer the technical possibility to directly (and efficiently) share memory slices across parallel threads, concurrently running the different SOBJs, by relying on a unique address space and/or operating system supported shared-memory.

In this article we tackle the issue of transparently and efficiently supporting shared-state in optimistic PDES applications run on top of shared-memory/multi-core machines, by enabling the application programmer to access within the event processing routines both the private state of the SOBJ (as already allowed by literature proposals) and a global portion of the state (as allowed by our innovative proposal), whose instance is represented by the value of global variables admitted within the application-level code.

We implemented a fully-featured shared-state management system targeted at x86_64 architectures and ELF executables. Also, we have integrated it within the open-source ROOT-Sim (ROme OpTimistic Simulator) package [43], an optimistic run-time environment supporting ANSI-C compliant application-level software implementing the SOBJs' logic in the form of event handlers.

In order to provide efficient support for the management of shared-state variables within the speculative processing scheme, in terms of both forward and backward (namely recovery) computation, our proposal relies on application transparent multi-versioning of global variables, which is based on non-blocking access/update operations of version lists. Thanks to this scheme, each SOBJ's callback procedure is allowed to observe the correct snapshot of the shared-state upon reading any global variable. In fact the global variable version that matches the timestamp of the event currently being processed at the SOBJ is transparently accessed (in non-blocking mode) via our software facilities. This allows maximizing the level of parallelism when the shared-state is actually accessed by multiple SOBJs callback procedures that are concurrently dispatched on different CPU-cores. Also, global variables' read operations that are eventually revealed to be non-consistent are automatically undone, which is achieved by rolling back any global variable update that is detected to be dependent on the incorrect read operation. The innovative rollback scheme for managing causally inconsistent dependencies due to accesses to (transparently multi-versioned) global variables has been also integrated with classical rollback schemes aimed at recovering the private state of each SOBJ, thus allowing to always maintain causal consistency of both SOBJ-private and shared-state information. Further, the whole process of accessing and manipulating global variables, in their multi-version form, is achieved via application software automatic instrumentation techniques (of ELF object files), which are the fulcrum of parallelization (and speculation) transparency vs the application code developer.

The results of an experimental assessment of the shared-state management architecture are also reported for the case of a suite of DES applications including a classical synthetic benchmark and various real-world applications. All the case study applications have been run on top of an HP ProLiant server equipped with 32 CPU-cores and 64GB of RAM memory, a machine that is representative of current off-the-shelf commodity hardware exploitable for scientific computing like DES.

The remainder of this paper is structured as follows. For readers who are less familiar with PDES concepts, we provide in Section 2 a brief overview of the optimistic (speculative) approach. Related work is discussed in Section 3. Our innovative shared-state management architecture is presented in Section 4. A proof of correctness of the devised approach based on the notion of serializability is provided in Section 5. In Section 6 we present experimental data aimed at assessing the pragmatical viability of our proposal.

2 Recap of Optimistic PDES

Each SOBJ included in a PDES application is associated with its own view of logical time, known as *Local Virtual Time* (LVT), which is used to locally track the advancement of the computation along the logical-time axis. Any event execution at the SOBJ (possibly) updates its state, and moves the LVT to the timestamp of the processed event. Clearly, it is the responsibility of the underlying run-time environment [29] to keep track of changes of the LVT of some SOBJ upon dispatching the event to be processed by the SOBJ's callback procedure. While executing the callback procedure in charge of processing some event, new events (still marked with timestamps) destined to whichever concurrent SOBJ can be produced, and injected into the system by relying on some API provided by the run-time environment. We note that this kind of structuring of PDES applications leads them to fall into the broader class of event-based ones.

The core aspect of PDES is that causality rules among the state transitions concurrently occurring while executing SOBJs' callback procedures are based on enforcing non-decreasing timestamp order for the processing of events at any SOBJ. This is recognized as a sufficient condition for the correctness of the computation [29]. Under this enforcement, each SOBJ in the system has a coherent view of the flow of logical time, given that its LVT never decreases.

In the optimistic (speculative) approach to synchronization, also known as Time Warp [47], events are stored by the run-time environment into per-SOBJ event-lists, each of which is logically partitioned into a *future-event-list* and a *past-event-list*. The future-event-list stores events not yet processed, while the past-event-list records already processed events. Each SOBJ's callback procedure is eligible for CPU-dispatching unless the SOBJ's future-event-list is empty. Once dispatched, the callback procedure is allowed to process the event kept by the future-event-list having the minimum timestamp. Such an event is moved to the past-event-list once dispatched for processing it.

Timestamp order violations might arise since any SOBJ may receive an event with timestamp lower than its LVT (given that SOBJ dispatching is not subject to safety verification of causal consistency of its next to-be-processed event). If a timestamp order violation is detected, all the events that were executed out of timestamp order are rolled back by the run-time environment (they are moved back from the past-event-list to the future-event-list). Also, the LVT of the SOBJ is pushed back to the timestamp of the last event executed in correct order, and the SOBJ's state is recovered to its value prior to the timestamp order violation, which is achieved by either relying on traditional checkpointing methods (see, e.g, [62, 63, 65]) or by the means of reverse computing approaches (see, e.g., [17]), where reverse versions of the SOBJs' callback procedures are executed with the events to be rolled back as input, or where dynamically generated reverse versions of code blocks leading to actual memory updates are exploited (see, e.g., [22]).

For the classical scenario where SOBJs' states are disjoint data structures, with no cross read/write operations performed by the application-level callbacks, recovering the system to a correct state actually entails recovering individual SOBJs' state images separately. In particular, in case dependencies have been materialized due to the scheduling of some event between a rolling back SOBJ, say $SOBJ_a$, and another one, say $SOBJ_b$, these dependencies are undone via so called *anti-events*[1]. More in detail, an anti-event is materialized for each event injected in the system by $SOBJ_a$ during the rolled back portion of the computation, and is used to retract the originally injected event. Upon the delivery of an anti-event associated with an already executed event by $SOBJ_b$, the recipient rolls back as well. Instead, if the event has not yet been executed, the anti-event has the only effect to "annihilate" the originally injected event, which occurs within the run-time environment. After rolling back, any SOBJ resumes the execution of the events from its future-event-list, still dispatched by the run-time environment.

A concept that is relevant to optimistic synchronization is Global Virtual Time (GVT), which is defined as the smallest timestamp among those of (i) unexecuted events already inserted into the SOBJs' event lists, (ii) events being executed, (iii) events/anti-events in transit from source to destination. Since no SOBJ can ever rollback to logical time preceding GVT [47], the GVT value indicates the commitment horizon of the speculative computation. It is used both to execute actions that cannot be subject to rollback, such as displaying of intermediate results [3, 24], and for recovering memory. Specifically, event-buffers with timestamps lower than the GVT value will never need to be re-executed after a rollback, therefore the run-time environment can discard them from the past-event-lists of the SOBJs. The same happens to obsolete state information, if any, maintained to support state recoverability. The action of recovering memory after GVT calculation is typically referred to as *fossil collection*.

By the above description, it is clear that the inclusion of support for shared-state information, directly accessible in read/write mode by the application-level callback procedures concurrently dispatched along multiple threads, leads to enriching the traditional PDES programming model, which (as hinted) has been historically based on disjointness of SOBJs' states. In this paper, we provide such a support for the case of global variables included within the application code. Also, our support operates fully transparently to the programmer, who is therefore allowed to design SOBJs' callback procedures entailing the capability of accessing whichever global variable in read/write mode, under the illusion that the accesses are performed as if the application were executed serially, with SOBJs' callback procedures dispatched in non-decreasing timestamp ordering of the events they are processing. As already hinted, this

[1] An anti-event is an exact copy of the corresponding event, or of its digest, except for a single-bit value.

means exposing to the application developer a programming model based on the notion of objects and sequentially executed methods, possibly accessing both object-private and shared data.

3 Related Work

The work in [9] discusses how state sharing might be emulated by using a separate SOBJ hosting the shared data and acting as a centralized server. To tackle performance issues, a modification of the rollback behavior of this special SOBJ is presented via the notion of *version records*. This is an approach similar to the one proposed in [54], where a theoretical presentation of algorithms to implement a Distributed Shared Memory mechanism is provided in terms of protocols to keep replicated instances of a variable coherent. In particular, one of these 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. As for performance, this can pose a hard burden on the centralized node(s), which in the case of computations performing very frequent read/write operations on shared variables can give rise to a non-sustainable overhead, even in case message-passing based interactions operate on top of tightly coupled shared-memory systems. Overall, to mimic the access to a memory location, a whole stack of layers, including message passing ones, needs to be involved in the operation, which does not favor performance. Also, these specific solutions do not provide methods to automatically map a direct access to the shared-state onto the message-passing based one. Hence the programmer is himself exposed to the usage of remote interaction APIs (e.g. remote procedure calls) for coding the access in the application software. Overall, these approaches are strongly oriented to distributed environments, while we target the trend of shared-memory/multi-core machines. Further, in the above solutions, the centralized server processes the read/write requests sequentially, while we allow non-blocking concurrent read/write operations to be carried out by the threads running the application.

In [27] the notion of *state query* is introduced. Specifically, a SOBJ's callback procedure that needs to access a portion of the state whose owner is a different SOBJ can issue a query event to it, and wait for a reply-event, piggy-backing the target information. In case this value is later detected to be no longer valid (i.e., the access to the value was served violating the timestamp ordering of the events, including query-events), a classical anti-event is injected by the run-time environment in order to invalidate the query. Again, this approach relies on explicit query-events to be injected by the application code, hence access to the shared-state is not transparent to the application programmer, i.e., it is not mapped to direct read/write statements within the application code. Overall, this solution is still bound to the classical programming model based on disjoint state accesses by SOBJs' callback procedures and pure event-based interactions.

The work in [35] proposes to integrate the support for shared-state in terms of global variables, by basing the architecture on [20]. Although this proposal supports in-place read/write operations as we do, they provide no transparency, as the application-level code must explicitly register the SOBJ as a reader/writer on the shared variables. Furthermore, synchronization of the accesses by the concurrently executed callback procedures along different threads is based on locks, while we provide non-blocking shared-state management support.

In the context of the High-Level-Architecture (HLA) standard [44,45], proposals for supporting shared-state can be found in [34, 50]. These solutions are again targeted at distributed environments, since they are based on a middleware component that 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 associated with out of timestamp order data accesses, while we target speculative processing and optimistic synchronization.

The Software Transactional Memory (STM) paradigm, originally introduced in [67], allows multiple threads to access shared data while ensuring consistency wrt concurrent accesses. The main differences between multi-version-based STMs [11] and our proposal lie in that (i) STM does not enforce transparency wrt the application-level programmer, since transactions must be explicitly marked; (ii) when an update is externalized, it cannot be undone, i.e., there is no need for supporting rollback operations on externalized values, as instead it may occur in the optimistic synchronization protocol we target.

The work in [21] proposes a framework targeting multi-core machines and optimistic synchronization where Extended SOBJs (Ex-SOBJs), defined as a collection of SOBJs, can access state variables of each other directly. However, only one of them can be active at any time since an Ex-SOBJ can be run by a single thread in the system, hence no concurrent accesses can ever occur (and need to be handled) on these state variables. Every Ex-SOBJ should anyhow manage an event-list to perform rollback operations due to causally inconsistent shared-data accesses (along time) by the SOBJs it is hosting. In addition, public attributes are referred to variables which can be accessed by SOBJs in other Ex-SOBJs. These can actually be accessed concurrently along different threads, and the work proposes to handle the accesses to shared-attributes by relying on a specifically targeted STM implementation, where events are mapped to transactions and the actual implementation of the STM is based on [35]. This proposal inherits most of the features of the general STM paradigm, so that our proposal is set aside this one. Specifically, in this solution the accesses to shared attributes need to be carried out via a specific STM API, while in our proposal the access to shared-data is performed by simply referencing some shared (global) variable using its name, or a classical pointer to it.

As for pointer-based concurrent cross-state accesses by the SOBJs, the recent proposal in [58] provides an application-transparent support, integrated within a speculative processing environment. This solution relies on an ad-hoc Linux memory management subsystem that allows to detect, at run-time, any pointer de-reference (in read/write mode) to the state of some SOBJ. All the SOBJs' states touched by processing a specific application-level callback procedure are temporarily locked for access by the thread executing the procedure. A core difference between this proposal and the work we present here is that we provide application transparent non-blocking support for the access to shared data plus transparent multi-versioning, whose combination favors concurrency, while the approach in [58] is based on a blocking scheme and does not provide multi-versioning. As a consequence, upon trapping into an access to the state of some SOBJ, the executing thread needs to temporarily block the current processing activities till the unique active copy of the target SOBJ's state gets aligned (in terms of current logical time) to the one of the event being processed by the thread. Further, our proposal and the one in [58] can be seen as targeting orthogonal problems, since we deal with transparency and speculative access to global variables while the solution in [58] deals with accesses into the heap. Also, we use lightweight (and transparent) software instrumentation for tracking memory accesses, while the proposal in [58] is based on operating system level memory faults (dealt with by the ad-hoc memory management architecture). These orthogonality aspects lead these two approaches not to be competing ones, rather they could be ideally integrated in order to achieve a fully transparent support for speculative parallelization when considering the possibility to access both the heap and the data sections along any running thread.

As for non-blocking algorithms, avoiding mutual exclusion has been considered a benefit since the early 1970's [26]. Lamport [49] gave the first non-blocking algorithm for the problem of a single-writer/multiple-reader shared variable. Herlihy [40] proved that for non-blocking implementations of most interesting data types (linked lists among them), a synchronization primitive that is universal, in conjunction with reads and writes, is both necessary and sufficient. A universal primitive is one that can solve the consensus problem [28] for any number of processes. In our implementation we rely on Compare&Swap (`CAS`), which is a universal primitive. The work in [38] 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 [23]. 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 [55] 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 [46]. We explicitly rely on the latter solution to avoid the ABA problem in our non-blocking implementation of the support for managing shared-state information.

4 The Shared-State Management Architecture

4.1 Overview of the Methodological Approach

As hinted, our approach is targeted at PDES platforms to be run on top of shared-memory computing systems. Furthermore, we tailor our solution to run-time environments based on the multi-threading paradigm. These have been shown to provide a set of benefits, such as optimized usage of the computing resources (see, e.g., [21,71,72]) when compared to the traditional counterpart where parallelization is achieved by running a set of single-threaded processes within the PDES platform. Overall, we designed a shared-state management architecture allowing not to loose the benefits from multi-threading. As a consequence, the shared-state managed by our architecture represents the `data` and `bss` sections of a single executable, and all the multi-versioned variables (plus any metadata kept to correctly manage them) are allocated within the same address space that is accessible by the concurrent threads running within the executable, although the actual access/manipulation to/of multi-version lists occurs in a fully transparent manner to the application executable modules ultimately included in the `text` sections of the finally built application.

Keeping metadata and version lists into the same address space where the application threads operate is the basis for supporting fast access to the data structures. Further, speed in managing the concurrent accesses to the data structures by the different threads is (as already pointed out) further favored by the exploitation of non-blocking coordination algorithms [41].

A schematization of the overall approach we rely on is provided in Figure 1. The scheme shows how the actual organization and run-time behavior of our system is related to what is exposed to the application programmer. Particularly, the left side of the picture shows how the management of global variables within the application software can be actually coded by the software developer by relying on classical in-place accesses (via either variable names or pointers). When relying on classical compilation tool-chains (such as `gcc` plus `ld`), the final executable is guaranteed to perform causally consistent (timestamp ordered) accesses to global variables (within either the `data` or the `bss` sections) only under the constraint that the program is executed according to a single thread mode (provided that the events are passed as input to the application code according to timestamp ordering from an underlying scheduler library, such as an implementation of, e.g., a calendar-queue sequential scheduler [8]).

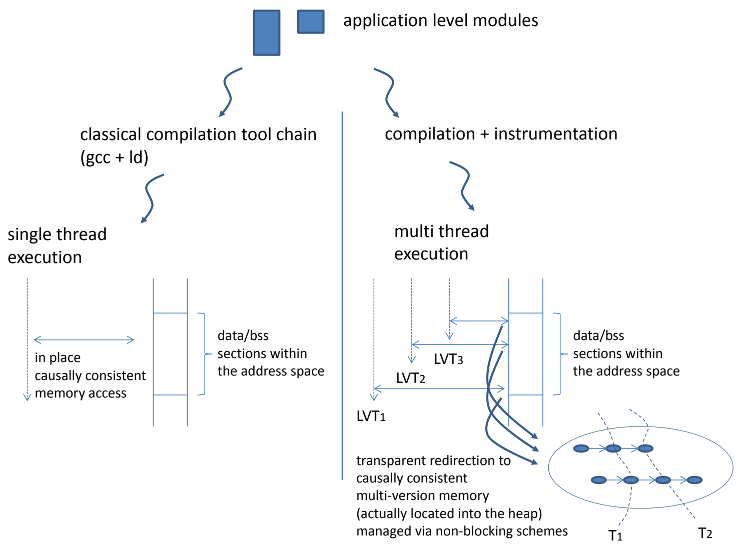
application level modules
classical compilation tool chain (gcc + ld)
single thread execution
in place causally consistent memory access
data/bss sections within the address space
compilation + instrumentation
multi thread execution
LVT3
LVT2
LVT1
data/bss sections within the address space
transparent redirection to causally consistent multi-version memory (actually located into the heap) managed via non-blocking schemes
T1
T2

Fig. 1: A Schematization of the Approach

With our proposal we still support in-place access to global variables by the application code, avoiding the need for inclusion of causal consistency management modules within the application code. Rather, as shown in the right side of the picture, we rely on application transparent instrumentation in order to generate an actual executable where memory accesses (to either `data` or `bss` sections) are intercepted at run-time, and are served by transparent redirection to a multi-version implementation of the global variables. Multi-versioning associates with each global variable value a timestamp, so that each thread living in the transparently enabled multi-threaded run can access the version that complies with the per-thread view of logical time (namely its LVT), just depending on what SOBJ has been scheduler for event execution along that thread. On the other hand, when some thread produces a new version of a global variable (say it writes on a global variable), the version will be assigned a timestamp that is equal to the LVT associated with the SOBJ that has been scheduled along that thread. Also, miss-speculation (hence the miss of the correct version) upon read access to some global variable, which may occur in case some concurrent threads posts an update on the variable in the past of (i.e. with a timestamp lower than) the LVT of the reading thread is automatically handled with no programmer intervention. This is done via an ad-hoc rollback protocol guaranteeing mutual consistency of SOBJs' states kept within the heap and shared data.

In our proposal, the identification of the actual address for a memory access is carried out at run-time, so as to cope with, e.g., non-determinism and CPU state dependent memory referencing (e.g. register-displacement based accesses). Hence, in the memory management system that we propose, `data` and `bss` sections represent kind of virtual resources, given that the actual memory access to global variables takes place as an access to multi-version lists, whose storage is transparently mapped to the heap. These virtual resources are exclusively exploited in terms of their boundary. In fact, any access to a global variable is discriminated by the transparently instrumented code depending on whether the referenced memory address falls within the global variable area boundaries. We also note that the reliance on the heap for the transparent allocation of multi-version lists allows for maximal flexibility in terms of the exploitation of virtual memory for speculation (namely for the maintenance of multiple speculatively generated versions of the global variables). Also, in principle this might even be integrated with secondary storage management of speculative data (by moving speculative data that have likely exited the locality of the application to secondary storage before their commitment and final removal–which is determined periodically upon GVT advancement–thus early freeing virtual memory buffers) according to suggestions and guidelines provided in, e.g., [73]. As we already hinted, scalability in the access to multi-version lists (hence to actual versions of global variables) is supported via the reliance on non-blocking dynamic lists' management. These avoid mutual blocking of the threads, at the expense of the possibility for some thread to retry its own operation in case a conflicting access by some other thread materializes. However, we might expect such a conflicting access to occur especially in kind of corner case scenarios entailing (severe) hot-spot accesses (especially in write mode) to some global variable.

Concerning atomicity of the access to multi-version lists, as we already hinted, our solution has relations with the STM paradigms, although they typically do not support in-place access to shared data, which is instead supported by our proposal. Also, an additional variation with respect to STM lies in that speculatively produced versions of a global variable are exposed (hence made accessible to concurrent threads) prior to the actual commit of the events that led to the production of those versions. This complies with the classical optimistic PDES philosophy, which is based on (ideally unbounded) chains of speculatively processed events possibly exhibiting data dependencies, whose causal consistency and committability is verified a-posteriori on a periodic basis—say upon periodic GVT computation[2]. As for this aspect, our methodology has relations with recent proposals in the context of transactional systems speculative replication [66], where individual processes can speculate along some highly likely transaction serialization order (say the one associated with request arrival order from the clients), whose commitability is then established a-posteriori (in case all the replicas have agreed upon the speculated order), which avoids a-priori synchronization across the replicas. We achieve a similar objective given that we avoid threads' synchronization for causal consistent (timestamp ordered) accesses to be carried out a-priori

[2] The only limitation to unbounded speculation is represented by storage constraints for keeping the speculatively processed/produced data records [25].

of the actual access to the timestamped versions of global variables. In fact, we tackle miss-speculation via rollback schemes operating a-posteriori of the access to global variables.

As an additional note, our proposal is fully based on user-space software solutions, thus neither requiring special hardware nor ad-hoc operating system support.

In the remainder of this section we initially provide hints on the instrumentation mechanisms we used to transparently track the accesses to global variables by the application code. Successively we present the data structures used to manage multi-versioning, and the non-blocking algorithms for the actual management operations. Then we present the schemes that we devised for integrating classical rollback operations of the SOBJs' states with the ones related to non-causally consistent access/manupulation of global variables.

4.2 Detecting Read/Write Memory Operations

In order to provide complete transparency to the application-level programmer, accesses in read/write mode to global variables must be explicitly intercepted, which has been supported by relying on instrumentation techniques aimed at modifying the actual instructions executed by software executables, without altering their semantics. To this purpose, we exploited the free software instrumentation tool called Hijacker [57], which we previously worked on, by further extending its capabilities in order to match our application transparent instrumentation objectives.

The instrumentation process we rely on works at compile-time, and is able to handle relocatable objects. It does not allow to alter finally linked executables (this has been a specific design choice, preventing possible security issues), but it can be seen as an additional compilation stage within the whole compilation tool-chain.

We have specifically targeted x86/x86_64 compliant assembly code [46], and have focused on the ELF executable formats for x86 and x86_64 architectures [53, 68]. This combination currently represents the vast majority of modern computing architectures targeted at parallelism and high-performance computing, as it is shown[3] in Figure 2 and Figure 3, for which the provisioning of transparent support for parallelization of the application code is a core topic to address.

In our approach, the application-level instruction code (i.e., the assembly byte-stream) is modified at compile-time in order to replace operations loading data to and from memory with actual function calls, where the invoked functions represent the entry points of our shared-state management architecture.

The following two API functions are provided as the entry points to our innovative shared-state management facilities:

- `write_global_variable(void *orig_addr, time_type lvt, ...)`

[3] Figures are drawn with data taken from `http://www.top500.org` as of March 2014.

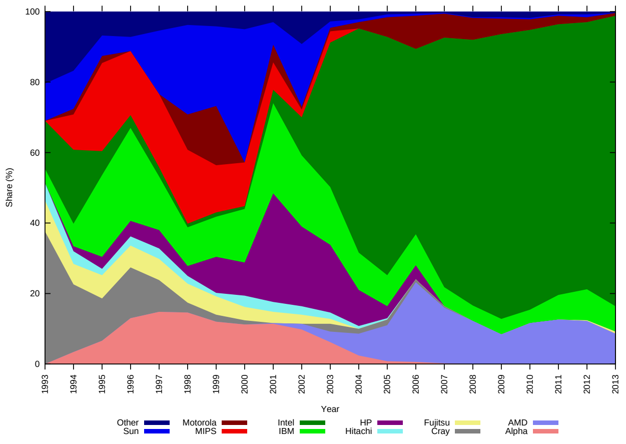

Fig. 2: Top500: CPU Vendors Share Over time

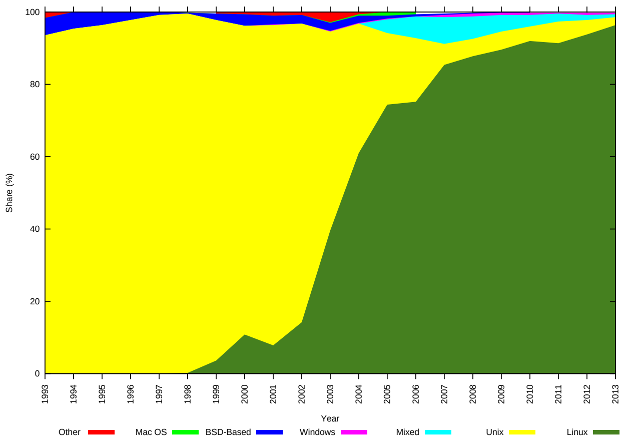

Fig. 3: Top500: OS Share Over Time

- `long long read_global_variable(void *orig_addr, time_type my_lvt)`

These functions allow the application code to access (in either read or write mode) any global variable according to the transparently managed multi-version scheme, and on the basis of the current LVT value experienced by the SOBJ for which the application level callback procedure accessing the global variable has been dispatched. We note that the LVT value of any SOBJ is at anytime known by the run-time environment that manages and schedules the SOBJs for event processing.

We have identified two main groups of instructions/code-blocks which have to be handled within the application-level assembly code, and transparently instrumented via calls to the above API functions. First, in x86/x86_64 simple load and store operations are identified by `mov` instructions. Whenever our instrumentation process 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 transparently 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. Of course, the register used by the `read_global_variable` function is pushed/popped on stack, which is done in order not to alter the actual view on the processor state by the application.

Second, the x86/x86_64 instruction set provides more complex instructions which allow an executable to efficiently modify memory areas in-place. As a relevant example, we mention instructions like `ADD m32, r32` or `INC m32`. In this case, our instrumentation scheme replaces the memory read/write instruction to be instrumented with a block of instructions, entailing a couple of calls to the shared-state manager read and write API, and re-implementing the same logic with several CPU instructions. This implementation, although adding some overhead, allows fully transparent integration of our shared-state management system with the application code, independently of the actual assembly instructions the compiler (e.g. a classical `gcc` compiler) selects for the translation of programmer-specified application level statements into machine-level code blocks.

High-level programming languages also allow to access memory locations in a non-direct way, namely through the use of pointers. Since our instrumentation process works at compile-time (a choice motivated by the need for avoiding the instrumentation overhead of, e.g., dynamic disassembly schemes, to become predominant and to hamper performance of the transparently achieved parallel run [60]), it is not possible to statically determine whether a pointer will target a global variable or not. To cope with this issue, we instrument any `mov` instruction which can handle pointers through a call to a `globvar_monitor` function, namely an assembly level routine, which efficiently determines if a pointer targets a global variable. To this purpose, we cache some disassembly information, allowing to fast determine the address targeted by the `mov` intruction into a record which is accessed by `globvar_monitor`, thus paying the disassembly overhead only once at compile-time.

In particular, x86/x86_64 architectures identify a memory address as the linear combination of (up to) five parameters, namely `segment`, `base`, `index`, `scale` and `displacement`. However, in common operating system technology (such as for Linux), global data segments are mapped to the same linear address, therefore the corresponding parameter becomes irrelevant in the determination of the actual target address in the linear address space[4]. Hence, cached data from the disassembling of one single `mov` instruction (namely, the

[4] Thread local storage data rely on segment registers to identify data positioning in memory. However, thread local storage is out of the scope of our proposal, due to its non-global nature.

record to be accessed by `globvar_monitor` to determine wether the `mov` is targeting a global variable), are organized as follows:

```
struct globvbar_monitor_entry {
    unsigned int size;
    char flags, base, index, scale;
    long displacement;
};
```

The `flags` field is used to identify which of the aforementioned four parameters are actually relevant and should be considered by `globvar_monitor` for computing the exact address (and size) of the memory-write operation. If this complies with the positioning of global variables within the ELF layout, then the internal mechanism of global variables' multi-versioned access is triggered via the proper API functions, as we already illustrated for the case of global variable access by the variable name within the application code (namely the case of `mov` instructions directly targeting global variable locations/addresses).

Given that the execution of the `globvar_monitor` module is an operation to be executed along the critical path of the application execution, hence directly impacting the event-execution cost experienced at the application level, we have adopted the following strategy for minimizing the performance overhead while accessing cached disassembling information (namely, the instance of the above record associated with any instrumented `mov` instruction). For each `mov` instruction involving a memory read/write, a set of `push` instructions are injected before the actual `call` to the `globvar_monitor` module, to let `globvar_monitor` directly find on the stack (rather than searching for it into some data structure) a memory area structured as `struct globavar_monitor_entry`, where the values of the fields describe the original `mov` instruction which caused the actual invocation of the module. In other words, the monitor pushed the record in question (as defined at compile-time) onto the stack right before accessing it. We recall that this approach has also the advantage that the pushed data (namely, the cached dissembling data) will be immediately accessible into the higher level hardware-cache, having it just been wrote to memory.

As a last note, the x86/x86_64 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, our shared-state management architecture provides two additional functions within its API, 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 `movs` or `stos` operations. Therefore, at compile-time, the instrumentation process replaces every string operation involving memory update with a function call to these APIs, accordingly. Also, `cmov` instructions are handled by replacing them with an assembly code snippet which mimics their semantic, and in turn relies on `mov` instructions (adopted to perform the memory update in case the condition is met) which are subject to the same instrumenting procedure as the one depicted above.

The last operation we perform at compile-time while transparently instrumenting the application code is the inspection of the application-level ELF object file in order to extract information concerning individual global variables. In particular, by exploring the application object 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 of the shred-state management architecture for setting up the version lists. In this way, by exploiting the $\langle name, address, size \rangle$ 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, allowing the programmer to rely on the complete set of constructs provided by ANSI-C. We note that, due to the multi-threaded nature of our reference run-time environment, a global variable's address is a common information shared among all the active threads within the same executable. This leads to cross-thread validity of the above information extracted by the instrumentation process, and used for setting up the multi-versioned memory model ultimately accessible by the concurrent threads during their execution.

Since we address (and rely on) assembly `mov` instructions in the instrumentation, we note that their opcode immediately provides information about the size of the memory operation. Therefore, we can easily rely on the `long long read_global_variable()` function, as its return type actually represents the largest type which can be accessed by any x86/x86_64 assembly instruction. Therefore, our injected code will simply copy the return value from this function into the proper used register (in case of an original `mov` operation from memory) using only the necessary bits for the associated `mov`.

4.3 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. As for this aspect, we have explicitly addressed the case of read/write operations performed by third-party software, just focusing on stdlib. Specifically, we have implemented a set of function wrappers which produce in-memory accesses via pointer passing. 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. As already mentioned in the methodological overview, these are used to detect at run-time whether a memory access falls within the global variables area. We exploited these same symbols while handling third party libraries. Specifically, the wrappers simply check whether global variables are involved in the operation, by comparing the variable's address with that of the injected symbols, before applying the actual access. In case a passed pointer targets a global variable, operations are redirected to the shared-state management architecture API, which actually access version lists.

4.4 Memory Map and Version Lists

In order to significantly enhance performance, we have decided to avoid requesting to the underlying memory manager (namely `malloc`) memory segments on-demand, whenever the shared-state management architecture needs to install some data structure. On the other hand, we install and manage large pre-allocated segments, by relying on the `mmap` service, and by selecting high (close to the stack) virtual addresses, so as to avoiding interfering with the `malloc` library (e.g., in the management of the program `brk`). Each pre-allocated memory segment is partitioned according to the definition of the following structure:

```
typedef struct _globval_mem {
    int num_vars;
    globvar_info  *variables;
    volatile int  first_node_free;
    globvar_node  *versions;
} globvar_mem;
```

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. The choice of having only one memory segment, rather than per-thread ones, is because global variables can be accessed by all worker threads. Hence, in case of per-thread segments, a read operation by some worker thread would need to find the correct version by scanning all the per-thread data structures, which would entail a non-negligible overhead, especially for large number of threads to be managed.

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` (which is used to store the size of the hash table) is set to be a power of two. Given that at startup the total number of global variables is known, `MAX_GLOBVARS` is increased (always keeping it a power of 2) until the collision is less than 20%. This choice uses more memory than needed, but can generate a significant speedup when accessing variables.

Anyway, in case collisions are found even after the increment, separate chaining is used as a means for finding a free place. Although this might seem sub-optimized, we note that global variables' virtual addresses are clustered in a contiguous portion of the address space, therefore the least significant bits are more likely to define a different key for each of them in the hash table. 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;
```

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

Since we rely on memory pre-reserving, version lists must be implemented using nodes scattered around the pre-reserved 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;
  time_type *read_list;
} globvar_node;
```

where `lvt` is the logical time (namely, the LVT value) associated with the version (i.e., the timestamp T_e associated with the event e processed by some SOBJ callback procedure during the execution of which the version was generated), `value` is the global variable's value, and `next` is used to identify 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 logical time. In Figure 4 we provide a complete picture of the memory map installed onto the pre-reserved memory segment.

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 threads running concurrently are given the same entry for handling two different versions.

The ALLOCATE pseudo-code is reported in Algorithm 1 in the Appendix. In order to allow non-blocking concurrent accesses, it relies on `CAS`. The `globvar_mem` data structure holds in `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

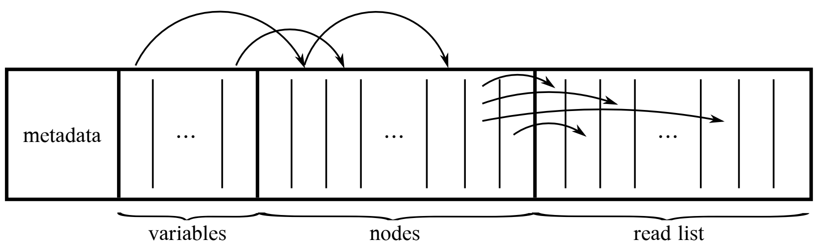

Fig. 4: Memory Map Organization

bitmap of file descriptors associated with a process. Specifically, it is always atomically increased upon allocation, and gets atomically decreased in case an entry is released having index less than the first chunk currently available within that block (the decrease leads the final value to correspond to the index of the freed node). Starting from that slot, a kernel instance tries to allocate a node by storing via a `CAS` operation a non-zero value into 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[5]. The companion function RELEASE is much simpler, as it only entails resetting the `alloc` and updating `first_node_free` via an atomic set operation, implemented again relying on `CAS`.

In order to cope with the ABA problem [18], 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 based on the *Cantor pairing function*:

$$\frac{(n_1 + n_2)(n_1 + n_2 + 1) + n_2}{2} \tag{1}$$

where we set n_1 to the thread logical identifier (as defined by the run-time environment in the internal $[0, maxThreads - 1]$), and n_2 to the value of a monotonic per-thread counter (initially set to 1) which is incremented upon each call to `generate_mark`. This function is very fast, as it is mostly based on integer operations, and allows to generate system-wide unique marks[6].

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 [39]. 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 deletions, two `CAS` operations are used, one to mark the `next` field of the deleted node as *logically* deleted, and

[5] 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.

[6] `generate_mark` can of course return two equal values when the counter overflows, but this situation can happen after a significant wall-clock-time is elapsed, so we consider it to be statistically non-significant for the ABA problem.

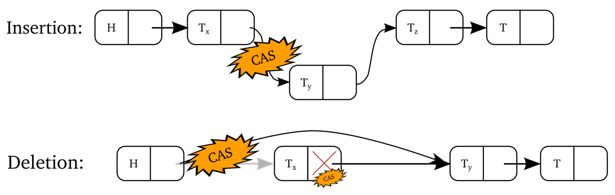

Fig. 5: Non-Blocking Linked List Operations

another to actually delete the node. We have slightly modified the algorithm in order to take into account our specific needs. In particular, the Find-Node procedure from [39] 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., the logical time value associated with the timestamp T_e of the generating event e) already exists. Specifically, the new node is simply linked after the originally existing one. This allows tracking causality among versions generated at the same logical time instant.

In addition, we note that SOBJs' callback procedures are more likely to access versions associated with higher logical time values, since well partitioned/balanced PDES computations usually proceed relatively evenly (along logical time) across all the SOBJs [15, 36, 61, 71], hence the locality of the accesses by the threads processing different events moves in a balanced way towards the latest snapshots of the global variables. 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 concurrent thread instances. The operations performed on version-lists are depicted in Figure 5.

4.5 Accessing Version Lists

As hinted, the API offered by our shared-state management architecture provides two 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.

The pseudo-code for Read operations is reported in Algorithm 2 in the Appendix. For efficiency reasons, before letting any SOBJ callback procedure execute an event, the shared-state manager sets up an *AccessSet*, i.e., a mapping between version nodes and variables. Whenever a variable is accessed for the first time, Find-Node determines which is the most suitable version for

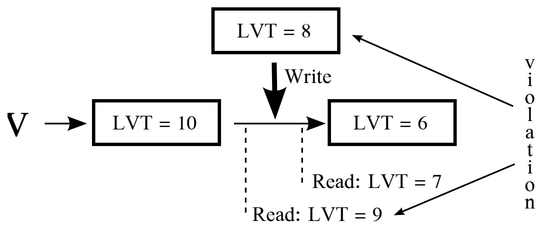

Fig. 6: Occurrence of the Rollback Operation

the current LVT associated with the SOBJ, and the tuple $\langle slot, version\rangle$ is placed into *AccessSet* in order to speedup the retrieval of the version, avoiding the scan of the list upon subsequent accesses while processing the same instance of callback.

As for WRITE operations, the associated pseudo-code is reported in Algorithm 3 in the Appendix. Its behavior is twofold, depending on whether it is 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. In any case, a call to INSERT-VERSION is performed, which, as stated in Section 4.4, creates a new version. The second part of the WRITE operation entails checking the *ReadList* for ensuring consistency, as it will be clearly depicted in Section 4.6.

4.6 Synchronization and Rollback Operations

In order to strengthen the optimism-oriented (namely, speculative) nature of our implementation, we allow interleaved reads and writes on any version list, and we explicitly avoid a version k installed at logical time T_k to invalidate every version j such that $T_k < T_j$. In fact, we note that consistency is violated only if, at logical time T_x, some SOBJ callback procedure reads the version associated with logical time T_y such that $T_y \leq T_x$, and at a certain point during the execution a new version node associated with logical time T_z such that $T_y \leq T_z < T_x$ is installed.

This means that every SOBJ callback procedure which reads a certain version node must leave a mark of that operation, i.e., visible reads [10] are enforced. In fact, as shown in Figure 6, we are interested in undoing only the events' processing instances that have read a version older than the just inserted one.

To this end, we augment the classical notion of rollback as presented by the Time Warp synchronization protocol for PDES [47] (based on state separation across the SOBJs), by injecting (after any write operation) a special anti-event towards all the SOBJ which have read a so-defined causally inconsistent version. This is reflected into Algorithms 2 and 3. In fact, in the READ operation, before returning the variable's value, the couple $\langle sobj, lvt\rangle$ 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 thread will start the rollback operation while the *ReadList* is being updated. Otherwise, this scenario would produce a non-trackable read operation.

In addition, after the spinlock has been taken, a check on the variation of the `alloc` field for that particular version is performed, so as 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 SOBJs which have read the previous node's value, while they were requesting a version at a logical time such that they should have read the one in the version list 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, as it is in the current left node.

We note that another step must be undertaken in order to ensure correctness. In particular, whenever a special anti-event 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 event queue (managed a the level of the run-time system) and modified the Write function so that whenever a node is installed during the execution of an event, the event queue keeps track of this operation via pointers to the nodes created during the event's execution. In case a rollback operation undoes that event, the node is removed from the version list, and the *ReadList* is scanned for injecting anti-events towards every SOBJ which has read that particular node.

4.7 Memory Recovery and Management

In order to correctly manage memory recovery and prune obsolete data within the multi-version scheme, we had to extend the classical *fossil collection* approach used in the context of SOBJs' state disjointness (i.e. no global variables). Before entering the details, let us recall a few aspects related to recoverability of private SOBJ states.

Commonly, not all state snapshots of the private SOBJ can be directly restored via checkpoint recoverability, due to the need for reducing the cost (and memory footprint) of logging state information, e.g., by taking checkpoints infrequently [62, 63]. State snapshots that are not directly available in the log are reconstructed by reloading some older state image and by reprocessing intermediate events in a fictitious way, namely with no actual externalization of the processing outcome (such as the injection of events in the system). All these activities can still be supported transparently to the application code by the run-time environment, if designed/implemented according to proper schemes [14, 60]. This allows to optimize the tradeoff between logging and recovery costs, under the assumption that the replay leads to the same identical trajectory of the state updates, which is achieved by making the SOBJs' callback procedures live within piece-wise-deterministic environments (e.g. by relying on piece-wise-deterministic versions of the used libraries—for example, random number generators). The state reconstruction phase based on fictitious events reprocessing is termed *coasting forward*.

When coasting forward is admitted, once a new GVT value is computed (which is typically done by the run-time environment on a periodic basis, and according to various reduction-algorithm variants, see, e.g., [6, 33]) and is available at time t, say $GVT(t)$, event buffers and SOBJ-state recoverability data can be pruned according to the following scheme:

– for each SOBJ, say $SOBJ_i$, it is determined which is the latest directly recoverable state snapshot (based on recoverability data) with logical time less than or equal to $GVT(t)$; we denote as T_i the corresponding logical time value. This state (namely the corresponding log) cannot be discarded given that it would be needed to be recovered just in case a rollback pushes the SOBJ logical time back to $GVT(t)$, namely to the commitment horizon. Recall that no rollback to logical time earlier than the commitment horizon can ever occur.
– An event buffer keeping any event e that has been delivered to $SOBJ_i$ can be pruned in case its logical time T_e satisfies the condition $T_e < T_i$. Any other event buffer needs to be retained just for SOBJ's state recoverability purposes, given that it may be requested to be reprocessed in a coasting forward phase.

Under the above scenario, we can consider the retained event buffers as (potential) inputs for the SOBJs' callback procedures. However, when we admit global variables within the application program, then the global variables' versions in our multi-version scheme also represent potential input data to the SOBJs' callback procedures. As a consequence, we need to retain all the global variable versions whose logical time might be requested to be accessed in read mode during any coasting forward phase by some dispatched SOBJ's callback procedure.

As a consequence, we integrated the above depicted fossil collection scheme with the following additional steps:

(i) the absolute minimum value across all the T_i values related to directly recoverable snapshots of the SOBJ private states is identified. We denote this value as $\hat{T}$, and its identification can be easily carried out by an additional reduction step after GVT has been already computed.
(ii) those versions of a global variable such that their logical time is less than or equal to $\hat{T}$ are pruned, except for the latest one, which can be used to serve read requests falling in the past (in logical time) up to $\hat{T}$.

By point (ii) above, on any multi-version list, at least one version is retained (the latest one with logical time less than or equal to $\hat{T}$). Hence the list of versions associated with any global variable can be seen as logically partitioned into two disjoint sublists, namely the one including all the elements that are subsequent to the one to be retained (these are the elements to be kept alive), and another one including all the elements to be discarded. This allows pruning the list while still concurrently admitting the insertion of additional nodes in the part devoted to keeping the alive entries, and both these activities can be carried out via the non-blocking (allocation/deallocatoin) algorithms we have already devised. Hence we can allow some concurrent thread to perform list pruning while some other thread can dispatch SOBJs' callback procedures in forward (normal) execution mode, leading to the creation of new versions in some global variable version-list. In other words, our scheme can be integrated with non-blocking GVT and fossil collection algorithms, such as the one in [59], which do not impose kind of barrier synchronization across the threads prior to admitting them to reprocess events after a GVT/fossil-collection phase has been run, hence enhancing scalability and resilience to thread reschedule delays at the level of the operating system.

Other two points need to be noted. One is related to how to distribute the pruning of version lists across the active threads running the application (and hence running within the run-time PDES environment). This aspect is non-trivial given that different global variables may show different patterns of usage (e.g. update patterns) by the application, hence some multi-version list may include a lot of versions to be pruned, while other version-lists might include a reduced amount of versions to be pruned. Statically binding the pruning of partitions of the global variable set across the threads may therefore lead to suboptimal distribution of the pruning load, which in turn does not favor balanced advancement in logical time of the SOBJs managed by the threads, given that the threads sustaining less pruning load can promptly resume (according to the aforementioned non-blocking GVT/fossil-collection scheme) the forward execution mode of the SOBJs they are managing.

To overcome this problem, we have devised a scheme where a bit into a prune-bitmap is associated with each global variable, hence with its multi-version list. Once, $\hat{T}$ is computed, the threads running the application scan the prune-bitmap by performing a `CAS` to update one single bit within a word of the bitmap. In case the `CAS` instruction succeeds, then the thread is charged with the responsibility to prune the corresponding multi-version lists. Hence, threads that are fast in pruning their currently associated multi-version list, promptly try to take on the job of pruning other multi-version lists, since they fast try to win the race on the subsequent bits in the prune-bitmap. We note that in this scheme the bit-map does not even need to be reset after pruning, given that at alternate GVT computation and prune phases the `CAS` is used in order to support either the bit-transition from 0 to 1, or the opposite one from 1 to 0. With this scheme, the load of pruning the multi-version lists is more fairly (dynamically) distributed, given that if a thread prunes lists that are found to be short, it will more likely be in charge of pruning a greater amount of them.

Another relevant observation is related to the management of write operations of global variables in case of coasting forward processing activities (due to whichever rollback instance execution). As hinted above, in case a SOBJ's callback procedure is executed in a coasting forward phase, it needs to carry out side effects onto the private SOBJ state (for realignment purposes of the private state snapshot to the correct logical time), but no externalization of this execution via side effects on global variables needs to be actuated. To cope with this problem, the shared-state management architecture has been augmented with a dual mode execution capability of the `write_global_variable` internal API. More in detail, in case the write operation of the global variable is performed in coasting forward mode of the dispatched SOBJ, then the `write_global_variable` execution path boils down to a null one, given that the version to be inserted has been already created in the original (still correct) part of the computation of the dispatched SOBJ's callback procedure, along which we are simply coasting forward.

As the very last note, in case the memory buffer pre-allocated for keeping the version nodes gets filled, we reallocate it by doubling its size. This is actually done by copying separately the nodes and read list entries on a larger memory area—this is the reason why we have relied on indices to identify slots rather than standard pointers, as this frees from the need to correct internal pointers during this resize operation. Due to its intrinsic non-atomicity, resizing the version list requires some sort of synchronization. In particular, whenever a worker thread finds the buffer full, it relies on a `CAS` operation to atomically set the `first_node_free` field to the value -1, which tells all the other worker threads that someone is already resizing the structure. To avoid the scenario where the resize is executed while some other thread is already operating on the data structure, the variable list's metadata is augmented with a *presence counter*, namely a counter of the threads which are currently operating on the variable list. This counter is incremented/decremented atomically, by relying on assembly instructions (namely `lock inc` and `lock dec`). After that `first_node_free` has been set to -1, the worker thread spins waiting for the presence counter to become zero. In this way, the thread waits for other ones (which accessed the data structure *before* setting `first_node_free` to -1) to complete their operations. After the resize is executed, the current worker thread relies on a second `CAS` to set `first_node_free` to the first position available in the new portion of allocated memory, thus giving other threads access to the variable list.

5 Correctness of the Approach

In this section we provide a proof of correctness of the presented scheme for shared-state management. Particularly, we show that the application level callback functions processing the events ultimately lead to a linearizable history [42] of operations on the shared-state, whose serialization order is compliant with the timestamp order of all the events that are concurrently executed by the threads (whenever dispatching any SOBJ), and that are eventually committed due to the advancement of the GVT along wall-clock-time.

Our shared-state management architecture allows dispatched SOBJs' callback procedures to concurrently access global shared variables in an optimistic way and postpones synchronization among concurrent read/write operations executed on the shared-state to the time of conflict materialization. Therefore the implemented concurrency control scheme maintains a high degree of parallelism by ensuring that:

1. the read/write operations executed by a committed event e on the shared-state appear as they happened at the same indivisible point in time, associated with the logical time T_e in which e has been processed;
2. all the committed events execute the same operations and produce the same outcome as they were processed sequentially without violating the advancement of logical time.

For this reason, if we model an event e's execution as an atomic transaction τ_e [7] to be considered committed whenever e is committed according to the classical Time Warp algorithm (i.e., at a wall-clock-time instant t it can be established a GVT value $GVT(t)$ such that each event e', executed at a logical time $T_{e'} < GVT(t)$, cannot be revoked anymore), we can adopt the *serializability* consistency criterion [2, 7] over the histories of the committed events as the target correctness criterion of the proposed solution.

Even if in practice our shared-state management system behaves as an STM system, we have not designed it having in mind the typical STM-oriented correctness criterion, namely *opacity* [37]. In fact, guaranteeing that every read operation always returns a value consistent with the LVT of the reading thread would not prevent some SOBJ's callback procedure to see an inconsistent state due to the intrinsic speculative nature of the underlying Time Warp algorithm. Hence, independently of the presence or not of shared-state, as discussed in [56], a user defined code may be executed using data arguments that are inconsistent with the logical state of the application. These particular scenarios require complementary techniques to avoid anomalies, which are out of the scope of this paper.

Before showing the proof of correctness of our approach, we formalize the concepts of *history* on committed events and *operation*. A history $H_{GVT(t)}$ over a set E of committed events e at the GVT value $GVT(t)$ consists of:

1. a partial order of operations that reflect the *write/read* operations performed within e on the shared-state together with the *begin* (i.e., the invocation of e) and the *complete* (i.e., the commit of e);
2. the version order $\ll$ that specifies a total order on the variable's versions created by committed events. A *write* operation on a variable x issued by an event e is denoted by $w(x_e)$ while a *read* operation on a version $x_{e'}$ of variable x is denoted by $r(x_{e'})$.

We can build a Direct Serialization Graph $DSG(H_{GVT(t)}, \ll)$ over a history $H_{GVT(t)}$ as stated in [2] in order to define serializability in terms of topological properties on that graph. In particular a graph $DSG(H_{GVT(t)}, \ll)$ contains a node N_e for each committed event e in $H_{GVT(t)}$ and a directed edge $N_e \rightarrow N_{e'}$ for each pair of committed events e, e' in $H_{GVT(t)}$ such that one of the following dependencies occurs:

(i) e' directly *read-depends* on e if there exists a variable x such that e' executes a read $r(x_e)$;
(ii) e' directly *write-depends* on e if there exists a variable 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 a variable 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 $H_{GVT(t)}$ is serializable if the associated $DSG(H_{GVT(t)}, \ll)$ does not contain oriented cycles as defined in [7].

Therefore the correctness proof of our shared-state manager is formalized in the following Theorem:

Theorem 1 *At any wall-clock-time instant t, for the associated value $GVT(t)$ and for each history $H_{GVT(t)}$ of committed events admitted by the shared-state management algorithms then the $DSG(H_{GVT(t)}, \ll)$ graph does not contain any oriented cycle.*

Proof We prove that the $DSG(H_{GVT(t)}, \ll)$ does not contain any oriented cycle by showing that for each edge $N_e \rightarrow N_{e'}$, $T_e < T_{e'}$ always holds.

If an edge $N_e \rightarrow N_{e'}$ is in $DSG(H_{GVT(t)}, \ll)$ we have to distinguish three cases:

1. e' directly *read-depends* on e. In this case the shared-state manager has performed a read operation on a variable x by returning the version x_e having the greatest logical time T_e less than $T_{e'}$. Therefore $T_e < T_{e'}$.
2. e' directly *write-depends* on e. e' overwrites a value (by adding a new version $x_{e'}$) of a variable x already written by e. This is admitted only if $T_e < T_{e'}$.
3. e' directly *anti-depends* on e. e' adds a new version of a variable x after the version read by e. If $T_e \geq T_{e'}$ holds then the shared-state manager forces a rollback for e. Since both e and e' are committed then $T_e < T_{e'}$.

By Theorem 1, it follows that every committed history generated by our shared-state manager does not violate serializability.

6 Experimental Data

In this section we provide experimental data related to the run-time behavior of our shared-state management architecture. We note that, beyond the objective of simplifying the programmer job via automatic parallelization (with speculative processing) of the access to global variables, another objective is clearly the one of avoiding explicit event-based interactions across the SOBJs in case some of them encapsulates in its state data that need to be read/written as a result of the system wide execution of SOBJs' callback procedures. The avoidance of these interactions may result in reduced overhead while managing state information, as actually confirmed by the data we provide in this experimental study.

We initially provide information on the run-time environment where we have integrated our shared-state management facilities. Then we present the case study applications and the data related to their run-time behavior.

6.1 Run-time Environment and Experimental Computing Platform

All the software facilities forming the presented shared-state management architecture have been integrated and made available for free download within the ROOT-Sim platform [43], which is an open source C/POSIX environment for the development and the execution of discrete event applications. It implements a general-purpose environment relying on the optimistic synchronization paradigm, which offers a very simple programming model based on the classical notion of simulation-event handlers (as typical of several well known PDES systems such as [14,52]) to be implemented according to the ANSI-C standard, which represent application entry points for providing control to the SOBJs involved in the application.

More in detail, a unique callback needs to be specified by the programmer for application coding, whose signature is the following one:

```
int ProcessEvent(int me, ltime_t now, int event_type, void *content,
int size, void *state)
```

where the parameters have the following meaning:

- `me` identifies the dispatched SOBJ;
- `now` is the timestamp (the logical time) of the event being dispatched for processing, which represent the logical time currently seen by the dispatched SOBJ;
- `event_type` is event numerical code;
- `content` is the buffer keeping `size` bytes of event payload (if any);
- `state` is the pointer allowing the SOBJ to access its state (namely the top level data structure representing the state) into the heap.

Upon running the application, the user can specify the number N of SOBJs to be managed, whose identifiers are automatically mapped by the run-time system on the interval of values $[0, N-1]$. Also, at startup time the above callback is called at least once for each SOBJ, with the special event `INIT` filled as input, so as to allow them to perform startup operations, such as initial allocation of their states into the heap and initial injection of application specific events into the system. The latter task is achieved by exploiting a proper API, still exposed by the run-time environment, having signature:

```
int ScheduleNewEvent(int where, ltime_t timestamp, int event_type,
void *content, int size)
```

where some of the parameters have the same meaning as above, while the parameter `where` is the identifier of the destination SOBJ, and the parameter `timestamp` is the logical time at which the event will need to be processed.

All the aspects related to how to concurrently call the above callback function for the different SOBJs, how to (dynamically) bind the SOBJs to the threads that are activated at startup, how to provide support for recoverability of the SOBJs' states within the heap (with run-time selected optimal state log frequency) and how to flush output results (e.g. produced by `printf` calls within the event handlers) only when they refer to the actually committed portion of the computation are fully transparently managed by the ROOT-Sim run-time environment (details can be found in [3, 60, 61, 71]).

By the above API, the application programmer is requested to reason on no aspect related to parallelism. However, prior to the introduction of the currently presented shared-state management architecture, no `ProcessEvent` callback was allowed to manage global variables. More in detail, these variables, if present within the application code, were not managed according to correct causality rules of read/write dependencies by the concurrently processed application callbacks. The integration of the presented shared-state management proposal in the run-time environment leads to offer to the programmer the possibility to develop truly ANSI-C applications, including global variables, according to a sequential style programming model, which are then automatically parallelized according to speculative processing schemes.

This run time environment, and the hosted case study applications, have been run on top of 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 CPU-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.

6.2 The PHOLD Case Study

We initially exploit a well known benchmark for PDES systems, originally proposed in [30], which is called PHOLD. The objective of the study in this section is primarily to assess the overhead imposed by our share-state management approach and how it impacts synchronization dynamics.

PHOLD is a synthetic application where each SOBJ executes fictitious events only involving the advancement of the local simulation clock to the event timestamp. Each time an event is executed, a new fictitious event is scheduled, destined to some SOBJ inside the system, with a timestamp increment following some exponential distribution. The execution of an event is typically implemented as a busy loop (which emulates a specific CPU delay for event processing, hence a reference event granularity) plus some specific memory access in read/write mode into the LP state (e.g. to mimic specific locality patterns [70]).

We have considered a configuration of PHOLD with 512 concurrent SOBJs and event granularity of the order of 50 μs, which is representative of medium grain DES applications. The SOBJs schedule events for each other with probability 0.2, which expresses moderate (but non-negligible) interactions across the concurrent SOBJs.

This configuration, entailing no-global variables at all (which we will refer to as *baseline* in the plots), has been then modified so as to implement a variation of the PHOLD benchmark where a global array of `long long` variables is kept, with one entry associated with each SOBJ. When a SOBJ is scheduled for event execution, beyond executing the classical actions specified by the benchmark, it also reads and then writes a new value on its associated variable within the global array. In this new configuration, the global variables are accessed in data separation by the concurrent SOBJs. Hence this settings is useful for assessing the overhead imposed by the management of multi-version lists (including the overhead for pruning the lists upon GVT computation and fossil collection) without altering the interactions across the SOBJs. In fact, they will never generate causal dependencies in relation to the usage of global variables (given the data separation access-profile). For this configuration we considered two different probability values for an event executed by a SOBJ to actually access the per-SOBJ entry with the global variables' array, say 10% (representing a scenario with moderate usage of global variables) and 30% (representing a scenario characterized by more intensive access, in both read and write modes, to the global variables).

We have also developed a second variant of the PHOLD benchmark where the global variables within the array are no more accesses in data separation. More in detail, we have replaced the generation of cross-SOBJ events with the update of the global variable associated with the destination SOBJ for the interaction. This way, the concurrent SOBJs never develop dependencies due to cross-scheduling of events. Rather they only develop mutual dependencies due to updates performed onto the global variables, given that each SOBJ accesses in read mode to its own global variable instance upon executing the events. This occurs according to a causality pattern that complies to the one characterizing the original benchmark configuration (the baseline). This additional configuration allows us to assess in a comparative manner the scalability and actual performance provided by the two alternative interaction methods (the classical one based on events' cross-scheduling and the one supported by our innovative proposal, which is based on read/write operations on global variables implementing the shared-state).

For all the considered settings, we report in Figure 7 the variation of the benchmark execution time while varying the number of used worker threads within the underlying PDES platform (up to 32, which is the number of available physical CPU-cores in the underlying machine)[7]. By the data we can observe that the configuration with global variables accessed in data separation and 10% access frequency gives rise to negligible overhead with respect to the baseline benchmark configuration, indicating how a moderate usage of

[7] All the samples reported in the experimental study result as the average over 5 observations collected in different runs and with different seeds for pseudo-random generation.

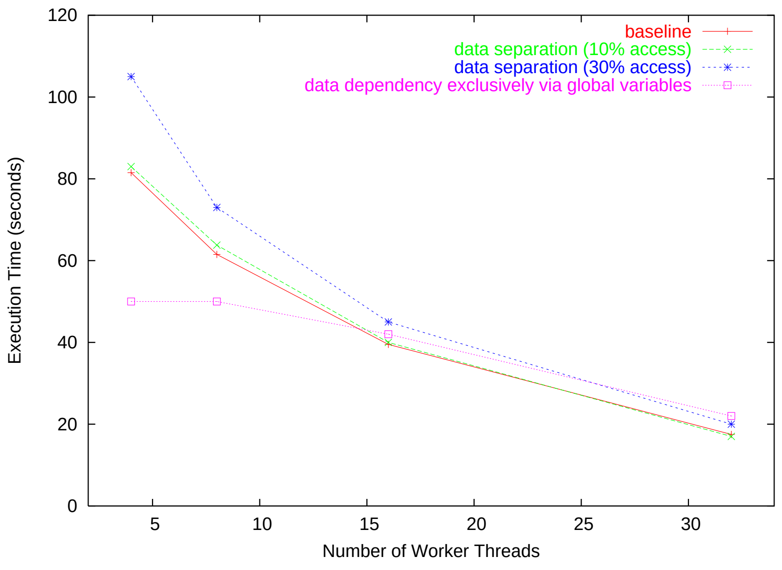

Fig. 7: Execution Time Results with the PHOLD benchmark

global variables is fully affordable in term of run-time overhead induced by our share-data management system. On the other hand, when the access frequency is increased to 30%, the overhead tends to increase. However, an interesting point is that the overhead tends to scale down when increasing the level of parallelism (say when employing more worker threads), which is imputable to the reduction of the negative impact by the management of version-lists on the execution locality (e.g. thanks to the increase of cache storage when increasing the number of used physical CPU-cores). In fact, while the overhead with respect to the baseline is of the order of 25% when employing 4 worker threads, it decreases to the order of 10% when running with 32 worker threads.

As for the configuration with global variables based interactions (replacing SOBJs' cross-scheduling of events), it provides significant performance improvements when the degree of parallelism is relatively limited (say up to 16 worker threads). However, when the level of execution parallelism is set to 32, it induces a small slow down of the execution (compared to the baseline). Clearly, this is due to the fact that with more worker threads, and accesses no more carried out in data separation, the likelihood of experiencing a conflicting access to multi-version lists (with consequent need for retry of the operation) increases. Recall that with no data separation, the multi-version lists need to be concurrently accessed also upon rollback operations, given the need for undoing speculative SOBJ-interactions that eventually reveal non-consistent. This does never occur in case of data separation accesses, since a rollback operation leads a unique worker thread to operate on any multi-version list, say the thread running the rolling back SOBJ associated with a specific global variable instance. On the other hand, we note that the slow down with 32 worker threads is limited to the order of 10%, a price that looks justifiable in any context where the reliance on global variables' in-place read/write accesses can lead to reducing the complexity of code development (via the avoidance of cross-scheduling of events).

6.3 The PCS Case Study

In this section we move to a real-world case study by focusing on a DES application that models the evolution of a Personal Communication System (PCS), namely a mobile network adhering to GSM technology. In this application, each SOBJ is used to model the state's evolution of an individual wireless cell (modeled as a hexagon), and the whole set of cells provides wireless coverage on a square region of variable size. Each cell handles a number N of wireless channels, which are modeled in a high fidelity fashion via explicit simulation of power regulation and interference/fading phenomena, according to the results in [48]. The event types which can occur at any SOBJ are: *Start Call*, which simulates a new call installation on a target cell; *End Call* which simulates a call termination; *Handoff Leave* which simulates the leave of an on-going call from the current residence cell; *Handoff Receive* which simulates the installation of a call handed-off from an adjacent cell.

Upon the start of a call (or upon an hand-off between adjacent cells), power regulation is performed by the target SOBJ, namely the one modeling the target cell. This involves scanning a dynamic list of records included in the SOBJ state, whose elements keep track of current power allocations for standing calls, for computing the minimum transmission power allowing the current call setup to achieve the threshold-level Signal-to-Interference (SIR) value.

This application is highly parameterizable. Beyond the already mentioned number N of wireless channels per cell, the set of configurable parameters entails: i) τ_A, which expresses the inter-arrival time of subsequent calls to any target cell; ii) $\tau_{duration}$, which expresses the expected call duration; iii) τ_{change}, which expresses the residual residence time of a mobile device into the current cell. These parameters affect the *utilization factor* of available channels, expressed as $\tau_{duration}/(\tau_A * N)$. This impacts the granularity of the events processed by the SOBJ callback procedure since the more the busy channels, the more power-management records are allocated and consequently scanned/updated during the processing of different events. At the same time, higher values of the channel utilization factor lead to higher memory requirements for the state image of individual SOBJs.

In our study, we considered a scenario with 1024 cells (SOBJs), each one managing $N = 1000$ wireless channels (resembling macro-cell technology). Also, $\tau_{duration}$ has been assumed as exponentially distributed, with average value 120 seconds, τ_{change} has been still assumed as exponentially distributed, with average value of 300 seconds, while the inter-arrival time τ_A (still assumed as exponentially distributed) has been varied in terms of its mean value in order to get 3 different values for the average wireless channels' utilization factor, namely 25%, 50% and 75%, so as to significantly diversify the actual execution pattern (CPU/memory demand) by the application. These three scenarios will be referred to as Low/Medium/High load in the plots.

In this case study, global variables are not used to represent part of the modeled system. Rather, they have been used to implement global (aggregated) statistics (such as the average transmission power selected upon setting up the call, or the likelihood of SIR dropping under the predetermined threshold due to dynamic fading phenomena). This is an explicit choice, which gives rise to a scenario where the size of the global variables' area is relatively small, when compared to the heap storage used for keeping the private state of the 1024 SOBJs running within the application. Also, the access to the global variables' area leads the SOBJ's callback procedure to perform both read and write operations on the global variables, given that the update of the global statistics is performed incrementally. This leads to a scenario where any SOBJ can develop read/write dependencies with any other upon accessing the global variable area along the execution's logical time axis. In order to achieve different settings in which such dependency materializes with different frequency along wall-clock-time, we have parameterized the period according to which any SOBJ's callback procedure accessed the global variables' area for updating global statistics, in compliance with the updates of local statistics occurred on the private state of the SOBJ. In particular, the global statistics update occurs either after 100 or 5000 events processed at any SOBJ. With high update frequency (each 100 events), referred to HAF in the plots, we get higher likelihood of actual concurrent accesses to the global variable area by different threads, which may lead to increased likelihood of failure/retry of the operations that depend on the success of `CAS` updates (as compared to the low update frequency case, referred to as LAF in the plots), which are required to managed the various algorithms that underlie the shared-state management architecture.

For this case study, we report in Figures 8-10 the variation of the application execution time vs the number of used CPU-cores, in the configuration where we activate in the run-time environment one thread per each used CPU-core. In particular, we compare the execution time of our shared-state proposal vs the one achieved by encapsulating the global statistics into the private state of a proper SOBJ, and coding in the application additional events for interacting with this SOBJ (particularly for pushing to this SOBJ the incremental variation of the statistics observed by any other individual SOBJ on the basis of the information recorded into its private state). This is the only alternative for allowing on-the-fly update (and possible inspection) of global statistics in case no support for directly accessing a shared piece of information (like the one we offer) is provided. We also consider the scenario where global statistics are not updated on-the fly, hence excluding any shared-state at all. In this scenario, the SOBJ only keeps its own view of the statistics, which is flushed to a file for off-line analysis and aggregation. This is a baseline configuration

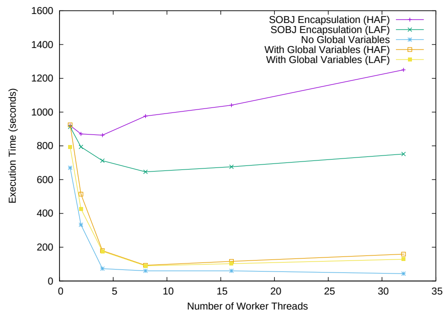

Fig. 8: PCS—High Load

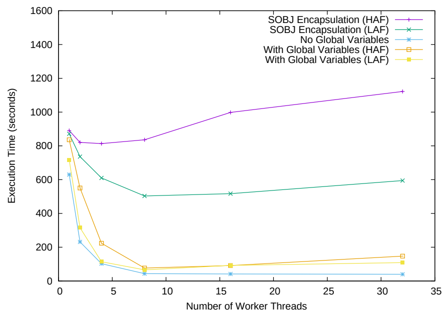

Fig. 9: PCS—Medium Load

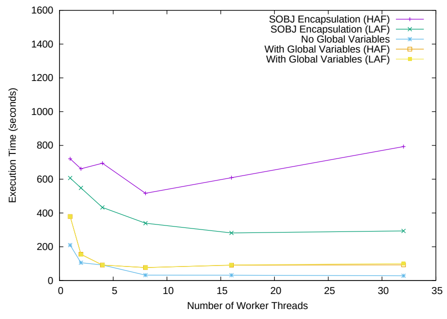

Fig. 10: PCS— Low Load

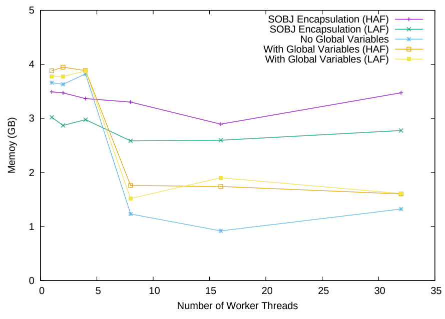

Fig. 11: PCS—High Load

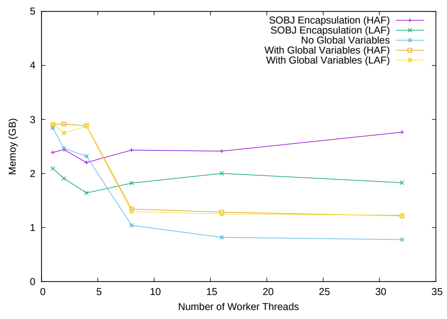

Fig. 12: PCS—Medium Load

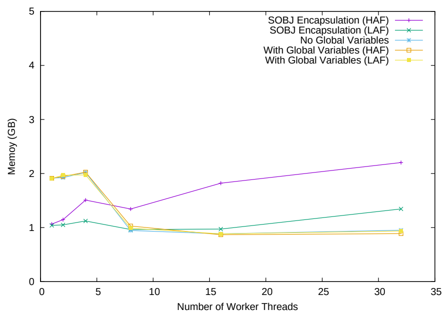

Fig. 13: PCS—Low Load

mostly used to assess the validity (e.g. in terms of reduced overhead) of the others.

By the data we can observe that the optimum number of threads for all the settings of the PCS application, when running with our support for transparent parallelization in the access to global variables, is on the order of 8/16. In fact we observe a great reduction of the execution time when increasing the number of used CPU-cores up to the value 8/16. On the other hand, after this value, the performance stabilizes. However this phenomenon is not primarily due to scalability issues in our proposal, since the same (or very similar) performance trend is noted for the baseline configuration entailing no shared-state at all. In fact, this configuration tends to significantly favor performance while increasing the number of used CPU-cores up to 32 limited to the case of High workload, as expected due to the higher resource demand of this settings and to the fact that running with no global variables (no on-the-fly computation of global statistics) leads to a scenario with very reduced coupling of the SOBJs' state transitions along the logical time axis (in fact in this scenario the coupling is caused exclusively by hand-off events, which occur relatively infrequently). In other words, 8/16 CPU-cores looks the best suited parallelism degree for the execution of this case study application in most of its different configurations. Hence, further increasing the number of CPU-cores beyond the value 8/16 leads (in most of the settings) to scenarios of over parallelism, which do not pay off independently of the reliance on global variables or not. Overall, when running with over parallelism, our shared-state management architecture does not induce significantly increased performance penalty compared to the scenario when global variables are not used at all. This is an indication that all the optimizations included in the architecture (e.g. in terms on non-blocking management of shared-state operations) provide an effective transparent support. This also allows the overhead by our solution, as compared to the baseline, to stay (almost) flat vs the increase of the number of used CPU-cores.

On the other hand, the execution scenario based on the proper SOBJ, whose private state is used to encapsulate the global statistics, manifests scalability issues as soon as the number of used CPU-cores increased beyond the value 4/8. As expected, the scalability issue is more evident for the case of frequent updated of the global statistics, given the increased workloads of events to be processed by this SOBJ. Overall, beyond simplifying the programming model, our shared-state management architecture also leads to improved concurrency of useful work carried out on the shared-state portion, as compared to the case of encapsulation of the shared-state within a single application object.

Given the real world nature of PCS (as opposed to the synthetic nature of PHOLD), memory usage represents another interesting aspect to study, for which we provide run-time data. Specifically, we report in Figures 11-13 the variation of the total amount of memory requested by the application (as periodically sampled via the `top` command and averaged at the end of the run) when running with our shared-state management support and with the other considered configurations. An interesting observation is related to the fact that, increasing the number of threads leads to a reduction the overall memory demand, up to a point where the memory usage stabilizes. This phenomenon is due to the fact that increasing the number of threads leads to the increase in the amount of work committed per GVT/memory-recovery cycle (executed each 1 wall-clock-time unit in our runs), which leads to more efficient/prompt pruning of the data structures. On the other hand, the interesting point in this

Table 1: PCS—Maximum Speedup vs the Calendar-Queue based Serial Run

Load	**Global Variables**		**No Global Variables**	**SOBJ Encapsulation**	
	HAF	LAF		HAF	LAF
Low	2.82	2.80	9.18	0.50	0.91
Medium	10.54	10.42	23.98	1.18	1.90
High	21.80	22.37	46.65	2.33	3.10

plot is related to the fact that the configuration with no shared-state at all has very similar memory demand as the one employing our support for shared-state, which indicates that the additional memory requirement for managing the multi-lists of global variables' versions (even in case of frequent updates and nodes insertions into the multi-version lists) is very limited. Although we may have expected this outcome, given the reduced size of the global variables section, as compared to the overall size of the collection of SOBJs' private states, the experiments fully confirm it. We also note that the reduction in memory usage vs the increase of the number of used CPU-cores does not arise in case of the configuration based on encapsulation of global statistics into the state of the SOBJ. Again, this is mostly due to the fact that, with this settings, the run commits events much slower that with the other settings, hence leading to less prompt recovery of memory buffers keeping obsoleted data (since data become obsolete after longer time). The only exception to this behavior is noted for the case of Low load, with LAF, a configuration where the memory demand by the application layer is anyhow reduced, hence leading the memory-reclaiming procedure of obsolete buffered data (e.g. committed events' buffers) to exhibit a reduced impact on the actual memory usage of the whole system (run-time environment included).

The very last part of this section is devoted to reporting data related to the speedup achieved by the parallel runs vs an optimized sequential run where the same application code is executed on top of a calendar-queue event scheduler [8] (in fact the data in Figures 8-10 show, for the case of single CPU-core, the execution latency that has been achieved with a run-time environment, namely ROOT-Sim, which is optimized for parallel executions, rather than with one optimized for sequential execution). By the data in Table 1 we see that except for the case of encapsulation of shared information in the SOBJ state, the optimal speedup provided by the parallel runs, over the optimized sequential one, is significative, which points out how the whole experimental study has been conducted with competitive parallel executions. This strengthens the relevance of the achieved outcomes. Also, the shared-state architecture allows for achieving (especially for higher workload contexts) on the order of 50% of the maximum speedup achieved in case of no global variables at all, which is a significant results when considering that this speedup percentage is achieved vs the one characterizing an execution scenario that is much more biased towards an embarrassingly parallel one (given the reduced coupling of SOBJs' state trajectories in case of no usage of global variables).

6.4 The TCAR Case Study

The Terrain Covering Ant-Robots (TCAR) application models the evolution of a scenario where specialized mobile agents (the robots) are used to explore a bi-dimensional space-region, e.g., for rescue purposes. Our implementation of this application conforms to the specification in [69]. More in details, a group of robots is set out into an unknown space, with the goal of fully exploring it, while acquiring data from sensors which are used to map the environment.

Robots are equipped with enough processing power to elaborate the sensors data online (thus, the map is constructed during the exploration), so as to allow them to rely on the acquired knowledge to drive the exploration in a more efficient way. Specifically, whenever a robot has to make a decision about which direction should be taken to carry on the exploration, it is done by relying on the notion of *exploration frontier*. By keeping a representation of the explored world, the robot is able to detect which is the closest unexplored area it can reach, computes the fastest way to reach it and continues the exploration.

Robots explore independently of each other until one coincidentally detects another robot. Whenever two robots enter a proximity region, they perform three different actions: i) they use their sensors to estimate their mutual physical position—recall that they are just in *proximity*; ii) they verify the goodness of their position hypothesis by creating a rendez-vous point in the explored part of the region, and trying to meet again there; iii) if the hypothesis is verified, they exchange the data acquired during the exploration, thus reducing the exploration time and allowing for a more accurate decision of the actions to be taken. Additionally, in case step ii) succeeds (i.e., the robots actually meet in the rendez-vous point), it means that the estimation of their respective position is correct. Therefore, they can form a *cluster*, i.e. they can start exploring the environment in a collaborative way. Specifically, this collaborative exploration can take place in two different ways. On the one hand, they jointly define (by relying on *cost* and *utility* functions, as defined in [69]) their next exploration targets, so that they can minimize the time required for a complete environment exploration. On the other hand, they might decide to make a *guess* about the position of other robots (the total number of which is known) which are not part of the cluster yet. In the latter case, one of the robots (the one for which the utility/cost ratio is convenient) targets the hypothesized position. If a robot is found there, the aforementioned steps are carried out, so as to increase the knowledge of the environment.

When implementing the above model according to PDES style rules, e.g. by relying on disjoint SOBJs' states to model the evolution of the entities within the system, three main hindrances are found. First, discovering the presence of a nearby robot can be difficult. In fact, either the associated SOBJs must communicate to each other their current position by injecting events, or they have to notify it to proper SOBJs, used to model the state of the regions (portions) of the whole area to be explored, again via the injection of events. Third, exchanging map information by notifying it as events' payloads could entail a data transfer non-negligible in size across the SOBJs. Additionally, all these programmatic steps are not straightforward, as they force the application developer to reason according to the classical SOBJs' state-separation paradigm.

Thanks to our shared-state management architecture, this application has been developed by having SOBJs representing only the active entities in the system, namely the robots, while representing the state of the regions belonging to the area to be explored into the global variables' section of the application code. In particular, a *presence bitmap* is embedded within the structured global variable representing the state of the region, each bit of which is associated with a specific robot, and its value is associated with the robot currently being registered as located in the region (or not). By relying on a fast bitmap scan, each robot (thus each dispatched SOBJ's callback procedure) is able to discover which ones are present in the region. Upon entering one cell, the SOBJ (beyond updating the bitmap) simply transfers its (so far acquired) local knowledge into a prober buffer in the global variables' section in order to make it available for knowledge merging with any other robot currently present in the same region.

This case study significantly differs from PCS in two main aspects. First, global variables are explicitly used to model part of the whole target scenario, namely the current state of the regions (e.g. in terms of robots' presence in the region). Second, the global variable area, a portion of which is touched by any SOBJ's callback procedure, represents a good percentage of the whole main memory storage used to represent the model. On the other hand, any SOBJ's callback procedure will access in read/write mode a reduced portion of the global variables' section, given that only the data structure representing the state of the region being entered (or exited) by the robot will be accessed. In other words, locality of the accesses to global variables changes depending on the level of (temporary) clustering of the robots into specific regions (instead, in the PCS case study, accesses to global variables were always localized on the same structured record, namely the one keeping the global statistics). In our study we consider a case where the environment to be explored is formed by 4096 regions (still modeled as hexagons) that globally represent a square bi-dimensional area. Also, 100 robots are used to explore the area, giving rise to an average density of robots per-region equal to 0.02, realistically representing cases where, e.g., a reduced number of highly specialized agents (robots) is employed for the exploration of a non-minimally sized region.

In Figure 14 we show the execution latency of the application for the case of employment of global variables, as well as for the case of the settings where regions are modeled by encapsulating the corresponding data structures into SOBJs' private states, hence requiring event-based interactions across regions and robots (as discussed above). We also report the execution latency for the case of an optimized serial run (based on the calendar-queue event scheduler) of the same (parallelized) application code exploiting global variables. For the parallel runs we report data while varying the number of worker threads (WT) used to sustain the application execution. By the data we see that the scheme

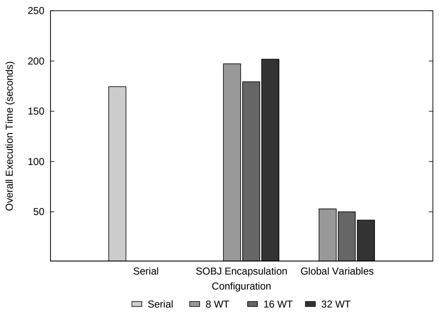

Fig. 14: TCAR—Execution Time

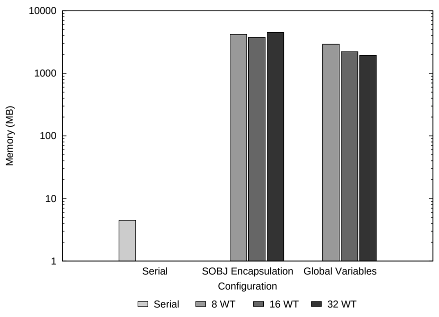

Fig. 15: TCAR—Memory Usage

relying on encapsulation is not able to outperform the optimized sequential run, while the configuration based on global variables it able to (providing a maximum speedup on the order of slightly less than 4). The core reason lies in that, for the particular settings of the TCAR application we have used, events are very fine grain (in fact an event takes, on the average, about 10 microseconds for being processed by the corresponding application callback), which leads to unsuitability (due to excessive overhead for cross-SOBJ data exchange) of the pure event-based interaction model for accessing some shared information. We also note that such a fine grain profile of individual tasks to be executed is generally adverse to parallelization, especially in contexts where the tasks need to be actually synchronized to ensure causal dependency, as for the case of PDES applications. Hence, achieving speedup of the order of 4 for this scenario via our shared-state management architecture looks a promising result, even if it is achieved with a non-minimal amount of threads.

As for memory usage, the run-time data are reported in Figure 15. By the data, we see similar memory usage by both the parallel configurations. Also, as expected, the memory usage of the parallel runs is much greater than the one by the sequential run. This is due to the need for keeping either logs of the region-SOBJ states (in the encapsulated configuration) or multi-versions of region information (in the global variable scheme). However, such an increased memory demand is not a major impairment to performance improvements by the parallel runs based on global variables, at least for the adopted test settings. Further, the increase in memory demand in speculative computations is a well known issue that can be tackled (if requested, given potential, or actually materialized, impact on performance due to reduced locality) with techniques that stand as orthogonal to the share-state management architecture we have presented, such as those based on artificially limiting speculation in order to avoid excessive growth of data to be kept in memory for recoverability purposed [25].

6.5 The Traffic Case Study

As final case study, we have used *traffic*, a real world application that we have developed in cooperation with colleagues from logistic engineering so as to provide support to decision making processes (such as scheduling of delivery services across the country). This application simulates a complex highway system (at a single car granularity), where the topology is a generic graph, in which nodes represent cities or junctions and edges represent the actual highways.

Every node is described in terms of car inter-arrival time and car leaving probability, while edges are described in terms of their length. At startup phase, the simulation model is asked to distribute the highway's topology on a given number of SOBJ. Every SOBJ therefore handles the simulation of a node or a portion of a segment, the length of which depends on the total highway's length and the number of available SOBJs.

Cars enter the system according to an Erlang probability distribution, with a mean inter-arrival time specified (for each node) in the topology configuration file. They can join the highway starting from cities/junctions only, and are later directed towards highway segments with a uniform probability. If after having traversed part of the highway a car enters a new junction, according to a certain probability (again specified in the configuration file) it decides whether to leave the highway. Whenever a car is received from any SOBJ, it is enqueued into a list of traversing cars, and its speed (for the particular SOBJ it is entering in) is determined according to a Gaussian probability distribution, the mean and the variance of which are specified at startup time. Then, the model computes the time the car will need to traverse the node, adding traffic slowdowns which are again computed according to a Gaussian distribution. In particular, the probability of finding a traffic jam is a function of the number of cars which are currently passing through the node. A `LEAVE` event is scheduled towards the same SOBJ at the computed time. Additionally, when a car is enqueued, the whole list of records associated with cars is scanned, in order to update their position in the queue, which reflects updates on the relative positions of the cars along the path they are traversing.

Accidents are derived according to a probability function as well. In particular, they are more likely to occur when the amount of cars traversing the highway portion modeled by a SOBJ is about half of the cars which can be hosted altogether. In fact, if few cars are in, accidents are less frequent. Similarly, if there are many, the traffic factor produces a speed slowdown, thus reducing the likelihood of an accident to occur. Therefore, the model discretizes a Normal distribution, computing the Cumulative Density Function in a contour defined as $cars\ in\ the\ node \pm \frac{1}{2}$, having as the mean half of the total number of cars which are at the current moment in the system, and as the variance a factor which can be specified at startup. The total number of cars which can be hosted by a SOBJ is computed according to the actual length of the simulated road, which is determined when the model is initialized. When an accident occurs, the cars are not allowed to leave the path portion modeled by the corresponding SOBJ, until the road is freed. In fact, if a `LEAVE` event is received, but its execution is associated with a car involved in an accident, the record associated with the car is not removed from the queue. Rather, its leave time is updated according to the accident's durations, and a new `LEAVE` event is scheduled. The duration of an accident phase is determined according to a Gaussian distribution, the mean and the variance of which are again specified at startup.

During the scan of the queue entries, with a certain small probability (specified at startup), a car decides to stop for a certain amount of time (e.g. for fuel recharge). This is reflected by setting a special flag in the record, and a duration for the stop is drawn from a Gaussian distribution. In this case, if a `LEAVE` event is received associated with a stopped car, the behavior of the model is the same as in the case of an accident. During a queue scan, if a stopped car expires its stop time, the relevant flag is reset, so that the next `LEAVE` event will allow it to exit from the path portion modeled by the current simulation object.

We have simulated the whole Italian highway network (particularly one hour of its evolution), by distributing the model over 1024 SOBJs. We have discarded the highways segments in the islands in order to simulate an undirected connected graph, which allows to have the actual workload migrating overall the highway. The topology has been derived from [4], and the traffic parameters have been tuned according to the measurements provided in [5]. The average speed has been set to 110 Km/h, with a variance of 20 Km/h, and accident durations have been set to 1 hour, with 30 minutes variance. This model has provided results which are statistically close to the real measurements provided in [1].

Also, for the purpose of the present study, we have further expanded the modeled scenario by having the highway model integrated with a model of a traffic/congestion monitoring system, which is aimed at building a simulated picture of the global state of the highway (e.g. for possible spreading via aside diffusion services to the customers). In particular, the 5 most important nodes (say SOBJs) in the highway (associated with five main cities within the Italian country) have been augmented with the capability of modeling data-aggregation centers. These periodically receive notifications by other SOBJs in relation to the state of the highway nodes/segments and on possible jams. This information is used to build the view of the global state of the traffic in the highway. We have implemented this model extension by either relying on cross-scheduling of events for traffic condition notifications (as in the classical approach not entailing shared-data among the SOBJs), or by relying on global variables so as the update of the traffic state for a given node/junction occurs by posting the corresponding information on some globally accessible record within an array. The array is periodically accessed by the SOBJs modeling the data-aggregation centers, so as to actually build the global view of the state of the highway.

Compared to the other two real world case studies we provided (say PCS and TCAR), *traffic* offers a still different execution pattern. Specifically, in the implementation based on global variables, most of the SOBJs access them in write mode, while only a few SOBJs (those modeling the data-aggregation centers) access the global variables in read/write mode. This leads to a scenario where synchronization across the concurrent SOBJs along the simulation time axis (due to the management of data-aggregation within the simulation) takes place via causal dependencies materialized by a few of them upon reading the global variables. In case of a miss of some version (due to speculative read operations that are eventually revealed as inconsistent), these SOBJs are forced to rollback, with consequent rollbacks possibly induced on the others as a secondary phenomenon. Clearly the same causal dependencies (and synchronization dynamics) are expected to appear for the version not relying on global variables. In fact, the scheduling of a data update event by whichever SOBJ towards one modeling some data-aggregation center may lead to a rollback operation in case the destination SOBJ already (say speculatively) stepped a head of the corresponding logical time of data delivery[8].

We report in Figure 16 the plots for the execution time of the two different configurations of *traffic* (the one with global variables and the one compliant with traditional style PDES coding, say with no shared-state) while varying the number of used CPU-cores. By the data we can draw the following main conclusions. The implementation based on global variables provides a noticeable gain in the execution speed, at any scale of the underlying computing platform. In fact, for this particular application, beyond simplifying the programming job, the reliance on global variables allows each SOBJ to notify all the SOBJs modeling the data-aggregation centers about its current state with a single write of a record in globally-shared memory, clearly managed by our shared-state architecture via multi-versioning. Instead, with traditional style PDES coding, each SOBJ needs to explicitly schedule multiple data-update events, each one destined to one of the 5 SOBJs modeling the data-aggregation centers, which induces a non-minimal amount of additional work on the data-exchange subsystem within the run-time PDES environment. Also, the corresponding overhead tends to increase with more worker threads given that the data structures supporting cross-scheduling of events within the ROOT-Sim run-time environment are managed via critical sections, whose conflicts for their access tend to increase with increased levels of execution parallelism.

Another interesting point is related to the data we report in Figure 17, where we plot the memory usage by the different runs. Here we can note that the configuration relying on global variables shows a moderate increase of the used memory when increasing the number of worker threads, which is somehow physiologic given that the speculative run progresses more fast, hence needing more memory for keeping speculatively produced data (event-buffers and multi-versions of global variables) prior they are pruned. However, this phenomenon is exacerbated for the settings based on tradition style PDES coding just because of the need for using more memory buffers for scheduling

[8] Clearly, these synchronization dynamics are mixed with–hence additional to–those already induced via the simulation of the passage of vehicles across different SOBJs.

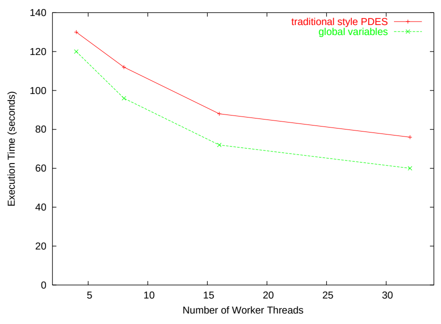

Fig. 16: *traffic*—Execution Time

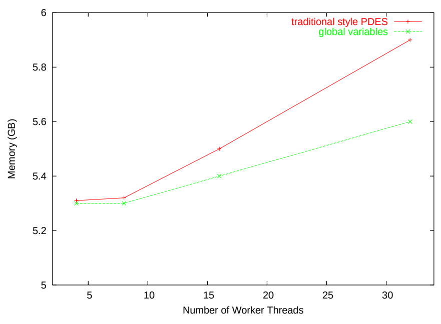

Fig. 17: *traffic*—Memory Usage

a same data-update event with multiple instances towards the different SOBJs modeling the data-aggregation centers. Clearly, the more evident increase of the memory requirements by the traditional style PDES coding configuration leads, as secondary effect, to reduced locality in the access, which additionally contributes to the performance loss of this configuration compared to the one based on global variables. As a final note, although not reported in the plots, also for this application the parallel runs provided remarkable speedups (say up to the order of 10) with respect to the corresponding sequential execution.

7 Conclusions

In this article we have addressed the issue of transparently parallelizing the access to global variables in discrete event applications based on the C programming language, particularly those relying on timestamped events as the input to application entry points (callbacks) that can operate on the memory image of the application, and where causal consistency is ensured by forcing timestamp-ordered accesses to slices of data. Our proposal has direct applications in the context of Discrete Event Simulation (DES), where the application code is typically based on multiple simulation-objects, each one having its own private state (generally allocated in the heap) and its own callbacks, and (possibly) shared-state information (such as global variables), accessible while executing any dispatched object-callback. Our proposal is based on speculative processing schemes and rollback/recovery techniques that can cope with global variables accesses that are eventually revealed as causally inconsistent (along logical time). Also, we use application transparent multi-versioning of global variables, plus non-blocking algorithms for managing the multi-version lists, in order to fully fit the nature of speculative processing schemes. A rollback scheme that integrates the classical one where the private states of the objects are restored to mutually consistent snapshots (due to causally inconsistent event-based interactions across the objects) has been devised so to account for non-consistent interactions due to read/wite operations on global variables by any concurrently executed application level callback. Application transparency is guaranteed in our approach via automatic software instrumentation, tailored to the Executable-and-Linkable-Format (EFL), leading the application code to transparently interact with the multi-version scheme. An experimental assessment of our proposal has been also presented, based on running a suite of case study DES applications on top of an off-the-shelf 32-core Linux machine. By the outcome of the experimentation, our approach has both the capability of simplifying the application code development and the one of improving performance compared to sharing information across concurrent objects by relying on event based interactions, as in the classical style to parallelization of DES applications.

References

1. ACI: Dati e statistiche. http://www.aci.it/?id=54
2. Adya, A., Liskov, B.: Lazy consistency using loosely synchronized clocks. In: PODC, pp. 73–82 (1997)
3. Antonacci, F., Pellegrini, A., Quaglia, F.: Consistent and efficient output-stream management in optimistic simulation platform. In: Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, pp. 315–326. ACM (2013)
4. AUTOMAP: Atlante stradale italia. http://www.automap.it/
5. Autostrade per L'Italia S.p.A.: Reportistica sul traffico. `http://www.autostrade.it/studi/studi\_traffico.html`
6. Bauer, D.W., Yaun, G., Carothers, C.D., Yuksel, M., Kalyanaraman, S.: Seven-o'clock: A new distributed gvt algorithm using network atomic operations. In: Proceedings of the 19th Workshop on Parallel and Distributed Simulation, pp. 39–48. IEEE Comp. Soc. (2005)
7. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley Longman Publishing Co., Inc. (1986)
8. Brown, R.: Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. Communications of the ACM **31**, 1220–1227 (1988)
9. Bruce, D.: The treatment of state in optimistic systems. In: Proceedings of the 9th Workshop on Parallel and Distributed Simulation, pp. 40–49. IEEE Comp. Soc. (1995)
10. Burns, J., Lynch, N.A.: Mutual exclusion using invisible reads and writes. In: Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing, pp. 833–842 (1980)
11. Cachopo, J.a., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. **63**(2), 172–185 (2006)
12. Cai, W., Turner, S.J., Lee, B.S., Zhou, J.: An alternative time management mechanism for distributed simulations. ACM Transactions on Modeling and Computer Simulation **15**(2), 109–137 (2005)
13. Carothers, C.D., Bauer, D.W., Pearce, S.: ROSS: a high performance modular Time Warp system. In: Proceedings of the 14th Workshop on Parallel and Distributed Simulation, pp. 53–60. IEEE Comp. Soc. (2000)
14. Carothers, C.D., Bauer, D.W., Pearce, S.: ROSS: A high-performance, low-memory, modular time warp system. J. Parallel Distrib. Comput. **62**(11), 1648–1669 (2002)
15. Carothers, C.D., Fujimoto, R.M.: Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms. IEEE Transactions on Parallel and Distributed Systems **11**(3), 299–317 (2000)
16. Carothers, C.D., Perumalla, K.S.: On deciding between conservative and optimistic approaches on massively parallel platforms. In: Winter Simulation Conference, pp. 678–687 (2010)
17. Carothers, C.D., Perumalla, K.S., Fujimoto, R.M.: Efficient optimistic parallel simulations using reverse computation. ACM Transactions on Modeling and Computer Simulation **9**(3), 224–253 (1999)

18. Case, R.P., Padegs, A.: Architecture of the IBM system/370. Communications of the ACM **21**, 73–96 (1978)
19. Chandy, K.M., Misra, J.: Distributed simulation: A case study in design and verification of distributed programs. IEEE Transactions on Software Engineering **SE–S5**(5), 440–452 (1979)
20. Chandy, K.M., Sherman, R.: Space-time and simulation. Proceedings of the SCS Multiconference on Distributed Simulation pp. 53–57 (1989)
21. Chen, L.l., Lu, Y.s., Yao, Y.P., Peng, S.l., Wu, L.d.: A well-balanced time warp system on multi-core environments. In: Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation, PADS, pp. 1–9. IEEE Comp. Soc. (2011)
22. Cingolani, D., Pellegrini, A., Quaglia, F.: Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES. In: Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation, London, United Kingdom, June 10 - 12, 2015, pp. 211–222 (2015)
23. Corporation, I.B.M.: IBM System/370 Extended Architecture, Principles of Operation. IBM Publication No. SA22-7085 (1983)
24. Cucuzzo, D., D'Alessio, S., Quaglia, F., Romano, P.: A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation. In: IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, vol. 0, pp. 227–234. IEEE Comp. Soc., Los Alamitos, CA, USA (2007)
25. Das, S.R., Fujimoto, R.M.: Adaptive memory management and optimism control in Time Warp. ACM Transactions on Modeling and Computer Simulation **7**(2), 239–271 (1997)
26. Easton, W.B.: Process synchronization without long-term interlock. SIGOPS Oper. Syst. Rev. **6**(1/2), 95–100 (1972)
27. Fabbri, A., Donatiello, L.: Sqtw: a mechanism for state-dependent parallel simulation. description and experimental study. In: Parallel and Distributed Simulation, 1997. Proceedings. 11th Workshop on, pp. 82 –89 (1997). DOI 10.1109/PADS.1997.594590
28. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
29. Fujimoto, R.M.: Parallel discrete event simulation. Communications of the ACM **33**(10), 30–53 (1990)
30. Fujimoto, R.M.: Performance of Time Warp under synthetic workloads. In: Proceedings of the Multiconference on Distributed Simulation, pp. 23–28. Society for Computer Simulation (1990)
31. Fujimoto, R.M.: Feature article - parallel discrete event simulation: Will the field survive? INFORMS Journal on Computing **5**(3), 213–230 (1993)
32. Fujimoto, R.M.: Exploiting temporal uncertainty in parallel and distributed simulation. In: Proceedings of the 13th Workshop on Parallel and Distributed Simulation, pp. 46–53. IEEE Comp. Soc. (1999)
33. Fujimoto, R.M., Hybinette, M.: Computing global virtual time in shared-memory multiprocessors. ACM Transactions on Modeling and Computer Simulation **7**(4), 425–446 (1997)
34. Gan, B.P., Low, M.Y.H., Wei, J., Wang, X., Turner, S.J., Cai, W.: Synchronization and management of shared state in HLA-based distributed simulation. In: Proceedings of the 2003 Winter Simulation Conference, vol. 1, pp. 847–854. IEEE Computer Society (2003)
35. Ghosh, K., Fujimoto, R.M.: Parallel discrete event simulation using space-time memory. In: Proceedings of the 1991 International Conference on Parallel Processing, pp. 201–208 (1991)
36. Glazer, D.W., Tropper, C.: On process migration and load balancing in time warp. IEEE Transactions Parallel Distrib. Syst. **4**(3), 318–327 (1993)
37. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, PPoPP, pp. 175–184. ACM (2008)
38. Harris, T.: A pragmatic implementation of non-blocking linked-lists. In: J. Welch (ed.) Distributed Computing, *Lecture Notes in Computer Science*, vol. 2180, pp. 300–314. Springer Berlin / Heidelberg (2001)

39. Harris, T.: A pragmatic implementation of non-blocking linked-lists. In: J. Welch (ed.) Distributed Computing, vol. 2180, pp. 300–314. Springer Berlin / Heidelberg (2001)
40. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
41. Herlihy, M., Shavit, N.: On the nature of progress. In: OPODIS, pp. 313–328 (2011)
42. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12**(3), 463–492 (1990)
43. HPDCS Research Group: ROOT-Sim: The ROme OpTimistic Simulator - v 1.0. `http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/` (2012). URL `https://github.com/HPDCS/ROOT-Sim`
44. IEEE Std 1516-2000 (2000): IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules. Tech. rep., Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA (2000)
45. IEEE Std 1516.1-2000 (2000): IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface (FI) Specification. Tech. rep., Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA (2000)
46. Intel Corporation: IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2: Instruction Set Reference
47. Jefferson, D.R.: Virtual Time. ACM Transactions on Programming Languages and System **7**(3), 404–425 (1985)
48. Kandukuri, S., Boyd, S.: Optimal power control in interference-limited fading wireless channels with outage-probability specifications. IEEE Transactions on Wireless Communications **1**(1), 46–55 (2002)
49. Lamport, L.: Concurrent reading and writing. Commun. ACM **20**(11), 806–811 (1977)
50. Low, M.Y.H., Gan, B.P., Wei, J., Wang, X., Turner, S.J., Cai, W.: Shared state synchronization for hla-based distributed simulation. Simulation **82**(8), 511–521 (2006)
51. Martin, D.E., McBrayer, T.J., Wilsey, P.A.: WARPED: A Time Warp simulation kernel for analysis and application development. In: HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture, p. 383. IEEE Comp. Soc. (1996)
52. Martin, D.E., McBrayer, T.J., Wilsey, P.A.: WARPED: A time warp simulation kernel for analysis and application development. In: Proceedings of the 29th Hawaii International Conference on System Sciences - Volume 1: Software Technology and Architecture, p. 383. IEEE Computer Society, Washington, DC, USA (1996)
53. Matz, M., Hubicka, J., Jaeger, A., Mitchell, M.: System V Application Binary Interface AMD64 Architecture Processor Supplement (2007). URL `http://www.x86-64.org/documentation.html`
54. Mehl, H., Hammes, S.: How to integrate shared variables in distributed simulation. SIGSIM Simul. Dig. **25**(2), 14–41 (1995)
55. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
56. Nicol, D.M., Liu, X.: The dark side of risk (what your mother never told you about time warp). In: Proceedings of the 11^{th} Workshop on Parallel and Distributed Simulation, PADS, pp. 188–195. IEEE Computer Society (1997)
57. Pellegrini, A.: Hijacker: Efficient static software instrumentation with applications in high performance computing (poster paper). In: Proceedings of the 2013 International Conference on High Performance Computing & Simulation, HPCS, pp. 650–655. IEEE Computer Society (2013)
58. Pellegrini, A., Quaglia, F.: Transparent multi-core speculative parallelization of DES models with event and cross-state dependencies. In: SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS '14, Denver, CO, USA, May 18-21, 2014, pp. 105–116 (2014)
59. Pellegrini, A., Quaglia, F.: Wait-free global virtual time computation in shared memory time warp systems. In: Proc. of the 29th IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Paris, France, October 2014 (2014)
60. Pellegrini, A., Vitali, R., Quaglia, F.: Autonomic state management for optimistic simulation platforms. IEEE Trans. Parallel Distrib. Syst. **26**(6), 1560–1569 (2015)

61. Peluso, S., Didona, D., Quaglia, F.: Supports for transparent object-migration in PDES systems. J. Simulation **6**(4), 279–293 (2012)
62. Preiss, B.R., Loucks, W.M., MacIntyre, D.: Effects of the checkpoint interval on time and space in Time Warp. ACM Transactions on Modeling and Computer Simulation **4**(3), 223–253 (1994)
63. Quaglia, F.: A cost model for selecting checkpoint positions in Time Warp parallel simulation. IEEE Transactions on Parallel and Distributed Systems **12**(4), 346–362 (2001)
64. Quaglia, F., Baldoni, R.: Exploiting intra-object dependencies in parallel simulation. Inf. Process. Lett. **70**(3), 119–125 (1999)
65. Quaglia, F., Santoro, A.: Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation. IEEE Transactions on Parallel and Distributed Systems **14**(6), 593–610 (2003)
66. Romano, P., Palmieri, R., Quaglia, F., Carvalho, N., Rodrigues, L.E.T.: On speculative replication of transactional systems. J. Comput. Syst. Sci. **80**(1), 257–276 (2014)
67. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, PODC '95, pp. 204–213. ACM, New York, NY, USA (1995)
68. The SCO Group, Inc.: System V Application Binary Interface, Intel386 Architecture Processor Supplement, fourth edn. (1997). URL `http://www.sco.com/developers/devspecs/`
69. Vincent, R., Fox, D., Ko, J., Konolige, K., Limketkai, B., Morisset, B., Ortiz, C., Schulz, D., Stewart, B.: Distributed multirobot exploration, mapping, and task allocation. Ann. Math. Artif. Intell. **52**(2-4), 229–255 (2008)
70. Vitali, R., Pellegrini, A., Quaglia, F.: Benchmarking memory management capabilities within root-sim. In: Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications. IEEE Comp. Soc. (2009)
71. Vitali, R., Pellegrini, A., Quaglia, F.: Load sharing for optimistic parallel simulations on multi core machines. SIGMETRICS Performance Evaluation Review **40**(3), 2–11 (2012)
72. Wang, J., Jagtap, D., Abu-Ghazaleh, N.B., Ponomarev, D.: Parallel discrete event simulation for multi-core systems: Analysis and optimization. IEEE Trans. Parallel Distrib. Syst. **25**(6), 1574–1584 (2014)
73. Young, C.H., Radhakrishnan, R., Wilsey, P.A.: Optimism: Not just for event execution anymore. In: Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation, PADS '99, Atlanta, GA, USA, May 1-4, 1999, pp. 136–143 (1999)

Appendix

Algorithm 1 Memory Allocation

```
1: procedure ALLOCATE
2:     $m \leftarrow$ GENERATE_MARK( )
3:     $slot \leftarrow$ first_node_free
4:     while $true$ do
5:         $alloc \leftarrow vers[slot].alloc$;
6:         if $alloc \lor \neg$ CAS($vers[slot].alloc$, $alloc$, $m$) then
7:             $slot \leftarrow$ next slot in circular policy
8:         else
9:             break
10:        end if
11:    end while
12:    atomically update first_node_free
13:    return $slot$;
14: end procedure
```

Algorithm 2 Global Variable Read

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

Algorithm 3 Global Variable Write

```
 1: procedure WRITE($addr$, $lvt$, $val$)
 2:     $slot \leftarrow$ hash table's entry associated with $addr$
 3:     if $slot \in AccessSet$ then
 4:         $version \leftarrow AccessSet[slot]$
 5:         $vers[version].value \leftarrow val$
 6:     else
 7:         $version \leftarrow$ INSERT-VERSION($slot$, $lvt$, $val$)
 8:         $AccessSet[slot] \leftarrow version$
 9:     end if
10:     for all $\langle sobj, lvt' \rangle \in ReadList$ s.t. $lvt' \geq lvt$ do
11:         inject anti-event towards $sobj$
12:     end for
13: end procedure
```