Autonomic State Management for Optimistic Simulation Platforms

Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia
HPDCS Research Group – DIAG, Sapienza, Università di Roma

Abstract—We present the design and implementation of an Autonomic State Manager (ASM) tailored for integration within optimistic Parallel Discrete Event Simulation (PDES) environments based on the C programming language and the Executable and Linkable Format (ELF), and developed for execution on x86_64 architectures. With ASM, the state of any Logical Process (LP), namely the individual (concurrent) simulation unit being part of the simulation model, is allowed to be scattered on dynamically allocated memory chunks managed via standard API (e.g. `malloc`/`free`). Also, the application programmer is not required to provide any serialization/deseralization module in order to take a checkpoint of the LP state, or to restore it in case a causality error occurs during the optimistic run, or to provide indications on which portions of the state are updated by event processing, so to allow incremental checkpointing. All these tasks are handled by ASM in a fully transparent manner via (A) run-time identification (with chunk level granularity) of the memory map associated with the LP state, and (B) run-time tracking of the memory updates occurring within chunks belonging to the dynamic memory map. The co-existence of the incremental and non-incremental log/restore modes is achieved via dual versions of the same application code, transparently generated by ASM via compile/link time facilities. Also, the dynamic selection of the best suited log/restore mode is actuated by ASM on the basis of an innovative modeling/optimization approach which takes into account stability of each operating mode with respect to variations of the model/environmental execution parameters.

Index Terms—Parallel Discrete Event Simulation, Optimistic Simulation, Autonomic Computing, Checkpointing, Time Warp

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, performance issues have been traditionally targeted via the Parallel-DES (PDES) paradigm [1], which is based on the partitioning of the simulation model into distinct Logical Processes (LPs) to be executed concurrently. Each LP models a portion of the simulated system, and interactions between different portions are captured by the exchange of *timestamped* event messages across the LPs. Thanks to partitioning and concurrent LPs' execution, PDES allows exploiting the computing power offered by (high-end) parallel/distributed platforms in order to both speedup model execution and make (very) large and/or accurate models tractable. An LP is usually implemented as a set of data structures updated via a callback, whose execution, representing the processing of a simulation event, is dispatched by an underlying simulation-platform (see, e.g., [2, 3]).

On the other hand, PDES relies on synchronization mechanisms, which are required to ensure that causality patterns across simulation events are maintained. Although differentiated definitions of causal consistency have been devised in literature [4–6], the most widely exploited correctness criterion states that each LP must process its input events (scheduled either by itself or by other LPs) in non-decreasing timestamp order. To support local timestamp ordering at the LP, two synchronization approaches have been proposed: *conservative* and *optimistic*. The conservative approach (see, e.g., [7]) avoids at all the possibility for any event to be executed out of timestamp order. This is achieved via block-until-safe policies suspending processing activities at the LP until it is determined that the execution of its next pending event is coherent with logical-time ordering. On the other hand, the optimistic approach (see, e.g., [8]) allows the LP 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 LPs 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, and to deliver performance which is less influenced by the simulation model lookahead[1]. These advantages are reflected also on the side of scalability, as recently shown in [9], where very large platforms (with thousands of CPU-cores) are employed for a comparative analysis of conservative vs optimistic approaches.

On the other hand, recoverability of the LPs' states, which is the building block for optimistic synchronization and which has been traditionally supported via log/restore techniques, poses problems on the side of both resource usage and application transparency. As for the former aspect, we need to consider both CPU usage for tasks enabling state recoverability (such as

- *Alessandro Pellegrini:* `pellegrini@dis.uniroma1.it`
- *Roberto Vitali:* `vitali@dis.uniroma1.it`
- *Francesco Quaglia:* `quaglia@dis.uniroma1.it`

1. The lookahead expresses the ability to predict the non-occurrence of events within an interval of simulated time in the future.

state logging) and recovery actions, as well as memory usage for keeping recoverability-related data/metadata.

The issue of transparency deals with avoiding the need for recoverability modules to be demanded from the application programmer, hence masking to her the actual synchronization paradigm. This is a non-trivial aspect that relates to the flexibility according to which the programmer is allowed to organize the data structures representing the LP state image, whose log/restore is then demanded from the underlying PDES platform. For incremental logging, this also requires transparent and efficient identification of the updates occurring within the LP-state layout.

In this article we present a fully innovative Autonomic State Manager (ASM) architecture, targeting C based platforms, ELF and x86_64 architectures, which jointly addresses transparency and performance issues in state recoverability by exhibiting the following features:

- It allows the application programmer to use standard constructs for dynamic memory allocation/deallocation, hence allowing the LP state to be scattered across non-contiguous memory chunks.
- It transparently enables phase-interleaved adoption of incremental and non-incremental log/restore modes.
- It runs each log/restore mode in a highly optimized fashion, via the adoption of dual-coding approaches and of classical schemes for the optimization of parameters determining the actual overhead for each mode (such as the frequency of the log operation).
- It dynamically (and transparently) switches to the best suited operating mode (incremental vs non-incremental) depending on proper execution dynamics of the optimistic simulation run. This is done on the basis of an innovative approach that takes into account performance stability of each operating mode vs variations of application and/or environmental parameters.

While some of the above points are dealt with by log/restore proposals in literature (as we shall discuss in the related work section), none of them fully covers the whole set of listed features.

ASM has been integrated (hence being available for download) into the ROOT-Sim open source optimistic simulation platform [10, 11] based on the C language and MPI. We also report a performance study for an assessment of ASM, which has been based on running a suite of Personal Communication System (PCS) simulation models on top of a 32-core HP ProLiant machine equipped with 64 GB of RAM memory, which is representative of current off-the-shelf commodity hardware exploitable for scientific computing.

The remainder of this article is structured as follows. In Section 2 related work is discussed. Section 3 presents the complete ASM architecture, including memory management aspects and the performance models it relies on. Experimental results are presented in Section 4.

2 Related Work

Logging is the classical means to support recoverability of the LP state in optimistic PDES systems. It relies on (infrequently) saving the LP state image in order to generate restoration points along the simulation time axis. Several studies have provided analytical models describing the expected log/restore overhead when experiencing a given rollback pattern (e.g. in terms of frequency of rollback occurrence at the LP) and when taking state logs, namely checkpoints, at specific points of the execution (for instance every χ events) [12–14]. By monitoring the independent parameters appearing in the analytical expressions, the models can be used to (dynamically) determine the settings that keep the log/resore overhead at minimum values. More specifically, taking checkpoints less frequently reduces the log cost. However, the state to be recovered might not be immediately available within the log, and would need to be reconstructed by restoring an older snapshot and by fictitiously reprocessing the intermediate events up to the restoration point (this re-processing phase is also known as *coasting forward*). Analytical models help determining the well-suited balance among these two opposite overhead tendencies.

The provided models target either non-incremental logging or incremental logging, or the case where the two approaches are used in combination (e.g. by taking incremental logs between subsequent non-incremental logs) [15] or are considered comparatively [16]. However, these proposals have been mainly tailored for the evaluation of log/restore policies (once known the costs for basic operations, such as the copy of the whole, or part of, the LP state image into the log buffer), not to provide log/restore architectures explicitly tackling transparency of log/restore tasks to the application-level code.

The issue of transparency has been dealt with by other studies. For incremental log/restore schemes this has been done by either instrumenting application level code (in order to transparently insert code portions aimed at identifying state update operations) [17] or by employing operator overloading schemes, as for the case of the object-oriented proposal in [18]. These approaches require compile-time identification of the memory portions forming the actual state image of the LP, hence they do not cope with dynamic memory. On the other hand, the solution in [19] provides supports for transparency in the context of dynamic memory based state layouts, but limited to non-incremental logging.

Other proposals have provided log architectures based on specialized hardware [20, 21]. They achieve some level of transparency, while also offloading the CPU, at the price of limiting the programming model, e.g., by imposing contiguousness or static determination of the memory area maintaining the state image of the LP.

The case of dynamic-memory based LP states has been addressed by the proposals in [22, 23]. However, the level of transparency is not maximized since ad-hoc dynamic memory allocation/deallocation APIs are

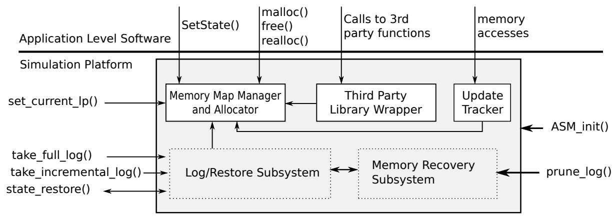


Fig. 1: Memory management architectural organization

used to notify the underlying simulation platform that the corresponding operation needs to be rollbackable.

Full transparency, in combination with incremental logging, has been provided by the High-Level-Architecture (HLA) oriented proposal in [24]. This approach exploits page-based memory update tracking mechanisms relying on facilities offered by the underlying operating system. Hence the granularity according to which incremental logs are taken cannot be set arbitrarily, and cannot be optimized depending on the actual needs. This proposal is suited for federations of simulation components where a middleware layer (namely the HLA Run-Time Infrastructure) is used to operate distributed coordination and data exchange, whose overhead tends to mask the one imposed by page-based logging. It results less suited for traditional PDES platforms, relying on highly optimized low-overhead engine-level coordination and data-exchange facilities.

An approach to state recoverability which is orthogonal to the aforementioned solutions is based on reverse computing [25–27], where the forward execution code (namely the native implementation of the application level simulation code) is coupled with a reverse code version which is in charge of backward compensating (hence undoing) the updates performed on the LP state in case a rollback occurs. The issue of automation of the generation of the reverse code, which targets transparency to the application programmer, is also faced in [25]. This approach reduces the memory demand for state-log buffers, while also nullifying the log overhead (since logs are not taken at all). On the other hand, the tradeoff is towards a potential increase of the restore latency in case very long rollbacks occur, which would require long reverse computing paths to reach the requested state recovery point. This approach demands the combination with periodic state logging in order (a) to avoid excessively long backward computation phases, and (b) to deal with non-reversible operations (such as plain assignments within the state image).

3 The Autonomic State Manager

3.1 Memory Management Architecture

3.1.1 Memory Mapper and Allocator

ASM's memory allocator relies on `malloc`, `realloc`, and `free` wrappers which are transparently interposed via simple compile time directives between the application-level code and the standard malloc library (see Figure 1). In order to allow rollbackable memory-management operations, the wrapper must know the identity of the calling LP. This is done by internally assigning a unique identifier in the range $[0, N-1]$ to each LP, where N is the total number of LPs hosted by the local instance of the simulation platform. This can be a single-threaded process within a classical multi-process PDES platform, or a thread within a multi-threaded platform organization [28][2]. Using `ASM_init(int num_LPs)` the platform can notify the number of locally hosted LPs to ASM, which in turn allocates an array of `num_LPs` entries structured as:

- `base_state_address`, identifying the address that should be passed to the event processing callback upon dispatching the LP so to allow it to correctly access its state in memory;
- `state_layout_info`, identifying the address and the current size of a metadata table keeping information on the memory layout for the LP state.

The API `void* set_current_LP(int LP_id, time_t sim_time)` exposed by ASM allows the simulation platform to notify ASM what is the identity of the local LP that is currently about to execute its next simulation event. In this way, the wrapper can identify the LP metadata it must refer to upon subsequent `malloc/free` invocations by the application software, and can be informed about the current logical time of the dispatched LP. This is required to make memory deallocations correctly rollbackable, based on the relation between the advancement of the Global Virtual Time (GVT) of the simulation and the simulation time associated with `free` calls (see Section 3.1.2)[3].

LP metadata, accessible via `state_layout_info`, are organized into table entries structured as:

```
struct malloc_area {
  int my_index, dirty_area;
  size_t chunk_size;
  void* where;
  int total_chunks, in_use_chunks, dirty_chunks;
  int next_chunk;
  time_t last_access;
  struct malloc_area *prev, *next;
}
```

Each entry is used to manage a block of given-size contiguous memory chunks, and different blocks host chunks with size corresponding to different powers of 2 (as supported by standard configurations of the malloc library). The `chunk_size` field indicates the size associated with the `malloc_area` entry. The `where` field is initially set to `NULL`, meaning that the chunks' block associated with that specific size has not yet been reserved, i.e. `malloc_area` entry is currently not valid.

When a `malloc` call is issued by the application-level code, the chunk size that best fits the request is

2. For platforms not adopting this strategy to identify LPs, a mapping between ASM's strategy and any other one can be created.

3. Optimistic synchronization relies on both event-messages and anti-messages, used to annihilate previously sent event-messages and inform the original receiver of the occurred rollback (this may in turn trigger a rollback in case the annihilated event-message was already processed at the destination). The GVT value represents the commit horizon of the simulation, namely the time barrier currently separating the set of committed events from the ones which can still be subject to rollback. This barrier corresponds to the minimum timestamp of not yet processed or in-transit event-messages/anti-messages.

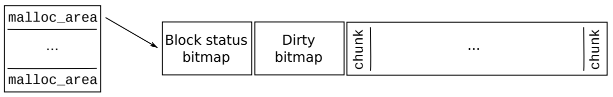


Fig. 2: Memory mapper/allocator data structures

identified, a block of contiguous chunks is allocated by the memory map manager via the underlying standard malloc library, and its address is registered within the `where` field, thus validating that `malloc_area` entry. This approach implements a pre-allocation strategy, where a block of pre-allocated chunks is reserved for a specific LP, thus improving memory locality for its state.

By default, the virtual address returned by the ASM memory allocator upon the very first `malloc` call for a specific LP is registered as the `base_state_address` for that LP. However, the `SetState(void* addr)` API is offered to the programmer to allow changes in the `base_state_address`. Hence, the LP can notify the underlying platform about any change in the memory positioning of the data structure representing the current root for the whole memory layout of its state.

For both time and space efficiency, each chunk within a pre-allocated block is associated with a single bit that indicates its current status, in term of whether it is in use or not[4]. The resulting status bitmap is placed at the head of the pre-allocated block of chunks, along with a dirty bitmap. These support data structures are created and managed only in case the corresponding block of chucks is actually allocated. Figure 2 shows the exact memory layout for the aforementioned data structures.

Upon a memory allocation, the `in_use_chunks` counter is updated, and `next_chunk` in the involved `malloc_area` is used to identify the most convenient position for starting the bitmap search in order to identify a free chunk. The manipulation of `next_chunk` follows a first-fit policy aimed at reducing both free chunks and bitmap fragmentation by aggregating in-use chunks in the initial part of the block.

When a block of chunks of a given size gets exhausted, the metadata table is expanded via a standard realloc operation, leaving available at least one new `malloc_area` entry, which gets linked via the `prev` and `next` fields, creating a list of entries used to manage chunks of a given size. Also, a new block of contiguous chunks of that size gets allocated. In this scenario, we can deduce that chunks of that size are highly useful for serving memory allocation requests for the LP. Consequently, whenever we expand the metadata table and reserve a new block of chunks, we double the block's size. This strategy further exalts memory contiguity for the LP state, minimizing costs associated with malloc library accesses, due to memory blocks pre-allocation.

Upon a `free` call, the associated chunk (and the corresponding block) is not released. Instead it is marked in the status bitmap as available for future allocations. In this way, memory deallocations are correctly rollbackable until they get eventually committed due to GVT advancement. Operatively, this is achieved by also exploiting the `last_access` field within the `malloc_area` entry, which is used to record the logical time associated with the last memory allocation/deallocation operation within the corresponding block, and to determine whether a block formed by chunks that have all been released can be really deallocated.

The explicit design choice to avoid per-chunk metadata would require the scan of all the `malloc_area` entries to check whether the entry is active, and (in the positive case) whether the chunk being released via the `free` call belongs to it. To avoid such a scan, ASM has been equipped with a software-level direct-mapped caching subsystem, with cache lines formed by the tuple $\langle chunk\_address, chunk\_end\_address, malloc\_area\_index \rangle$. Upon chunk allocation, the cache line is filled so that, in case of a subsequent `free` operation associated with that same chunk address, the wrapper retrieves the corresponding `malloc_area` in $O(1)$ time (unless for cases where the same cache line is overwritten during the run). A cache line is reset only when the corresponding chunk gets really deallocated.

3.1.2 Non-Incremental Log/Restore Support

The support for non-incremental logging operations (also termed full logs) linearizes and packs the LPs' currently-allocated chunks in a properly-sized contiguous log buffer (allocated via the malloc library), along with metadata describing the current memory layout. Further, in order to make the invocation of `SetState()` rollbackable, the `base_state_address` value is also logged. The trigger of the full log operation is a call to the function `take_full_log(void)`, which takes the checkpoint for the current LP, namely the one the identity and current logical time of which was notified to ASM via the aforementioned `set_current_LP` service. The log buffer is then linked to the head of a list of logs ordered by logical time values passed through by the LP.

To minimize checkpoint size, `malloc_area` entries that are not currently valid (i.e. those areas with no chunks allocated) are not logged at all, while the logged ones explicitly keep track (via the `my_index` field) of their original location within the metadata table. The chunks to be checkpointed are identified via a memory block's bitmap scan operation. Chunks with status bit set are the only ones packed into the checkpoint buffer, while the use bitmap is entirely logged, in order to allow correct chunks restoration by providing information on the correct positions they need to be copied back in their memory block. As the dirty bitmap does not contain any useful information for full logs, it is ignored.

Bitmap scanning is early stopped if the number of bits that have already been found set is equal to `in_use_chunks`. If only a few chunks are currently

4. This is a main difference from the original malloc library, where a complex header is associated with each managed chunk in order to maximize flexibility in memory usage (e.g. by dynamically partitioning or aggregating chunks according to the so-called "boundary tagging" scheme [29]). Nevertheless, ASM exploits such a flexibility by ultimately relying on the malloc library for actual virtual memory allocation.

in use within a block, by the aforementioned chunk-selection-for-allocation algorithm (i.e. the one which updates the value of next_chunk), they are likely to be placed in the initial portion of the memory block, which helps making the early stop approach effective[5].

On the other hand, when most of the chunks in a memory block are currently in use, an orthogonal optimization has been introduced. Specifically, no bitmap scanning is performed. Rather, the whole memory block (including unused chunks as well) is copied with a single memcpy call. The considerable benefit comes from the fact that modern processors offer optimized instructions to copy contiguous memory buffers of generic size (e.g. the movs instruction on IA-32 compliant processors). This optimization exploits a threshold-based mechanism that switches between the two approaches (i.e. selective vs non-selective) when the percentage of in-use chunks within a memory block oversteps a given value, providing a hysteresis region for stability reasons.

The general API offered by ASM for triggering a restore operation of the state of the current LP is state_restore(time_t requested_time, time_t *restored_time). When operating in non-incremental log mode, this function traverses the log-chain searching for the most recent full log with logical time less than or equal to requested_time. The restoration procedure unpacks the state, by de-linearizing memory chunks stored into the contiguous log buffer and placing them back into the original memory blocks' positions. malloc_area entries (maintained by the same log buffers) are used to identify correct original blocks, and are restored as well. The exact chunk locations within the blocks are identified using the logged bitmaps, which are also restored at the head of the corresponding memory blocks. The only exception is when the chunks within a given block were saved non-selectively due to large block occupancy, which is signaled via a special per-block flag inserted in the log buffer. In this case the bitmap is restored but is not used to identify the position of the chunks within the block since it is directly determined by the layout of the contiguous log buffer. Finally, ASM notifies the restored logical time via the restored_time parameter.

Metadata table's entries which were not valid at checkpointing time are not present in the log buffer, yet they must be reinitialized to make them compliant with the restored state layout and its logical time. Specifically, we reset the in_use_chunks field to zero and we set the last_access field to the currently restored logical time. However, memory blocks pointed by the where field are not really released, although the associated bitmap is reset. In fact, memory areas reserved for the LP are never deallocated due to the effects of a rollback operation. Also, when restoring valid malloc_area entries, the current linking of the metadata table entries (including the non-valid ones) is maintained. This allows the previously described algorithms for memory allocation to be Piece-Wise-Deterministic (PWD), meaning that the address of the allocated chunk is deterministically selected, unless a new block allocation is really required. In the latter case the address depends on the block allocation address selected by the underlying malloc library. Hence, ASM can serve a replayed sequence of LP allocation requests (which has been already served before the state restoration procedure) with the same identical memory addresses. This allows supporting coasting-forward operations correctly in case the overlying application complies with the PWD assumption (even when the application logic strongly relies on the addresses of allocated buffers). Hence, any optimized strategy for selecting checkpointing intervals and for balancing checkpoint-overhead reduction with coasting-forward latency can be also employed for performance optimization reasons, as we shall discuss later on.

5. The only exception is when a very adverse sequence of operations occurs, formed by several allocations and then a few deallocations, leaving memory holes scattered across the whole block.

3.1.3 Incremental Log/Restore Support

Incremental log operations for the current LP are similar to full ones, although they are triggered via the take_incremental_log(void) API offered by ASM. Also, the incremental-log buffer is still linked to the log chain according to increasing values of the logical time of the current LP. However, actual packing operations depend on extra information explicitly used to track updates in memory chunks (see Section 3.1.5), as follows:

A: dirty_area is set and dirty_chunks is zero. In this case the malloc_area is logged together with the status bitmap. Yet the dirty bitmap and the currently in-use chunks are not logged.

B: dirty_area is set and dirty_chunks is greater than zero. In this case the malloc_area is packed into the log buffer together with the status bitmap, the dirty bitmap and the chunks that are currently in use and which have been dirtied.

C: dirty_area is not set. In this case, no information associated with the area is logged at all.

Further, independently of the actual case among the aforementioned ones, data structures tracking dirty data/metadata are reset.

We finally emphasize that incremental state log operations do not require to be forced at each simulation event, but can be taken periodically. In fact they are based on the recognition of memory portions that have been dirtied since the last log, independently of the amount of events actually performing the dirtying operations.

When a restore operation must be executed, still invoked for the current LP via the generic API state_restore(time_t requested_time, time_t *restored_time), the latest incremental log with time less than or equal to requested_time is selected (its timestamp will then be returned via the restored_time parameter). After, the following steps are iterated by backward traversing the chain of logs:

1) A (not-yet-restored) malloc_area found inside the log buffer is put back in place inside the metadata table. The associated status bitmap is also

copied back from the log buffer (recall that independently of the type of log, a logged `malloc_area` is always associated with the corresponding status bitmap to guarantee recoverability of chunk allocation/deallocation operations).

2) Each dirty chunk found inside the log and associated with the `malloc_area`, which has not yet been restored in a previous iteration while backward traversing the log, is copied back in its correct position inside the corresponding memory block.

The iterative restore procedure stops when all the active `malloc_area` entries have been restored and all the in-use chunks that have been dirtied are also restored. Although in principles this could entail an indefinite number of iterative backward steps along the log chain, in practice the restore operation can be immediately finalized once we find a full log while backward retraversing the log chain. In fact, all the in-use chunks that have not yet been restored are immediately available inside the full log for copy-back operations. Full logs can be explicitly interleaved to incremental ones just for performance optimization purposes, as we shall discuss.

3.1.4 Memory Recovery

As explained, incremental and non-incremental logs are linked all together within per-LP lists, sorted by simulation time. Obsolete logs can be discarded, thus allowing virtual memory recovery, via the `void prune_logs(time_t new_GVT)`. This function scans the log queue for each managed LP, finds the oldest full log with time less than or equal to value of `new_GVT`, and prunes all the logs with a lower simulation time. Given the organization of the aforementioned recovery procedures, maintaining at least one full-log with time less than or equal to the newly computed GVT value allows correct recoverability of the LP state (independently of whether state restoration passes through the management of incremental logs).

3.1.5 Memory Update Tracker

The incremental logging support is based on the ability to know what memory regions have been modified since the last log operation (incremental or non-incremental). This task is carried out via an application-transparent software injection technique, based on a Software Instrumenting Tool (SIT) that we have specifically designed for analyzing and rewriting relocatable ELF (Executable and Linkable Format) objects generated by standard `gcc` compilers for x86_64 architectures. SIT parses the object generated after linking together all the application level modules (except third party libraries), and identifies every memory-write instruction, namely `mov` instructions with a memory location as the destination. The instrumentation process is then supported via the insertion of a `call` instruction to an assembly `update_tracker` module, which performs the identification of the exact memory address and the size (amount of bytes) involved in the memory update operation. We note that this is a typical approach to tracking memory-update references (e.g. for program debugging [30]). However, its adoption in optimistic PDES systems poses (more) stringent performance issues, requiring the design of a monitor performing memory-update tracking via very few machine instructions, in order to not significantly impact event execution latency.

In x86_64 architectures, the address of each memory-write operation depends on a set of up to four parameters, namely `base`, `index`, `scale` and `displacement`. The former two parameters identify registers containing actual values, while the latter two correspond to specific values of fields inside the memory-writing instruction. The instruction opcode tells which of those parameters are relevant. Also, the opcode, together with its prefixes, establish the real size of the memory area touched by the write operation. To cope with performance, instead of disassembling instructions at runtime (which could be onerous especially due to the complexity and variable format/length of the x86_64 instruction set) compile-time disassembling information is cached within the application-level code. In particular, the write size and the aforementioned four parameters needed to identify the actual write address are packed into a structure which is pushed onto the stack by additional instructions which are injected before the actual call to the `update_tracker` module. The data structure also maintains a `flags` field which is used to identify which parameters are actually needed for recomputing the target address. Hence it synthetically expresses the outcome of the compile-time disassembly process for any individual memory-write instruction. Overall, the data structure pushed onto the stack has the form:

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

Upon its activation, `update_tracker` retrieves the `write_instruction` record from the stack, so that the memory address for the write operation and the size of the memory being dirtied can be easily computed by the monitor via a few machine instructions[6]. To enforce transparency, `update_tracker` saves the whole CPU context (i.e. general purpose registers and the EFLAGS register), and puts them back before returning control to the actual memory-write instruction.

After the correct memory-write operation's destination address is computed, `update_tracker` invokes an internal routine which flags the dirty bit corresponding to the involved memory chunk(s). For this task, the software cache described in Section 3.1.1 is exploited again in order to perform a reverse query which translates a generic memory address into the chunk(s) actu-

6. The only exception occurs when the `write_instruction` record refers to `movs` or `stos` instructions, used for moving arbitrary size memory blocks. These instructions keep the information for identifying the destination address and the current size of the memory block being written into predefined registers, namely EDI and ECX, which are anyhow directly accessible by `update_tracker`, thus still allowing it to operate via a very reduced amount of machine instructions.

ally containing the memory buffers and the associated `malloc_area` entry. This allows fast identification of the bitmap to be involved in the update operation.

As a last note, for space constraints we cannot report additional details related to code transformations by SIT, such as those required for correctly handling indirect jumps (namely register-jumps) within the instrumented code. For these details we remaind the reader to [31].

3.1.6 Third Party Library Wrapper

When running in incremental mode, we would need to capture memory changes caused by the execution of third-party libraries. These are however not instrumented, which does not allow us to rely on `update_tracker`. We have explicitly addressed the case of update operations performed by third-party software, just focusing on stdlib. Specifically, ASM provides a set of function wrappers for all those functions which produce in-memory changes by the application-level software through pointers passing. The wrappers simply throw back the call to the underlying standard-library function, and then pass control to the memory-map manager with explicit indication of the address of the updated buffer, and the size of the updated memory block. In case the size cannot be retrieved by the library function signature (as for pointers to buffers used for strings), the memory-map manager updates the dirty bits for all the currently allocated contiguous chunks starting from the pointed address. This is obviously a conservative way of managing the memory map since some chunks that have not been really dirtied by the library are actually considered as dirty ones, thus being subject to log/restore operations. However, correctness is in no way touched, given that the wrapped stdlib library functions are all stateless, thus posing no issue on the side of memory log/restore.

As for stdlib functions which allocate new memory buffers (i.e. `strdup`), ASM provides a set of wrappers as well, which in turn re-implement the library functionalities by relying on its memory allocator, thus allowing the new memory buffers to be located within the actual LPs' state layout. Allocation/deallocation and update operations of these buffers are therefore made recoverable. We are currently working on techniques for allowing the application code to automatically rely on any (stateful) third-party library.

3.1.7 Dual Coding Mechanism

Memory-update tracking facilities should be enabled or disabled depending on the log-mode selected for the current phase of execution, in order to minimize the actual execution overhead. The most immediate way to achieve this goal is to insert a check on a particular predicate at the very beginning of the tracking routine, so that the monitor would simply return control to the application-level code if actual updates are not required to be identified. This is the case when running in full-log mode. Nevertheless, such a solution would impose overhead for non-useful housekeeping tasks, given that we already know that the current log-mode is the full one. Specifically, the creation of the stack frame with the `memory_write` record and the actual `call` to the `update_tracker` routine are de-facto useless tasks.

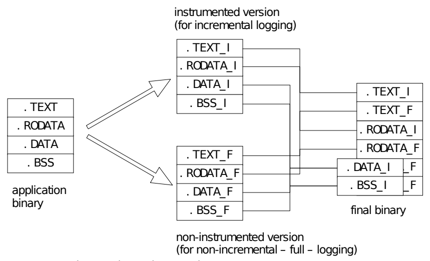


Fig. 3: The dual-coding generation process

To overcome this issue, the SIT module has been augmented in order to provide optimized coexistence of two versions of the same application modules, according to a dual-coding approach. Particularly, SIT allows generating two .text sections in the final executable, one containing a non-instrumented version of the application code, the other containing the instrumented counterpart. These sections are transparently placed into the executable layout in different virtual memory sections, using some standard facilities provided by the GNU linker `ld`. At the same time, SIT allows accessing the functions entry points (via function pointers) in such a way that the simulation platform is able to call the LP dispatching-callback via its standard API, and it is ASM's burden to make the pointers actually point the right code version. Also, the replicated .rodata/.data/.BSS sections associated with the two versions of the application object code have been collapsed on the same virtual addressing range in order to provide a single actual copy of initialized and non-initialized data, accessible by both the generated code versions. The whole dual-code generation process by SIT is schematized in Figure 3.

The above scheme would only entail additional virtual addresses consumption due to the presence of two versions of the application executable modules. However, this should not represent a real problem given that modern 64-bit processors enable extremely wide span of virtual memory addressing and text sections usually fill a reduced percentage of the virtual addressing range.

3.2 Log/Restore Overhead Modeling

After having enabled the optimized co-existence of incremental and non-incremental log/restore modes, as explained in the previous section, we provide the models assessing the corresponding overhead per event (due to both log and restore operations). These models borrow from the one presented in [12] for periodic non-incremental logging, for which we provide both (i) a specialization to capture internal mechanisms proper of our advanced memory-map manager (i.e. the cost of managing metadata identifying scattered memory layouts), and (ii) an extension to accommodate the case of incremental logging as supported by our architecture. The model in [12] describes the log/restore overhead on a per-LP basis. We inherit this feature in our modeling approach, thus providing a scheme allowing dynamic optimization of the log/restore mode for any individual LP. Consequently, from now on, overhead modeling and optimization of the log/restore mode are implicitly referred to what experienced for each single LP.

For the non-incremental case, borrowing from [12] and recalling the aforementioned specialization, the log/restore overhead per event can be expressed as

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

where

δ_e	is the average event execution cost.
S_F	is the average size of a full log.
δ_{LB}	is the average cost for logging a single byte belonging to the state image, which we consider to include the per-byte cost for logging the metadata kept by the memory-map manager.
δ_{RB}	is the average cost for restoring one byte from the log, again assumed to include the per-byte cost for the restoring state layout metadata.
P_r	is the rollback probability (frequency of rollback occurrences over event executions).
χ_F	is the selected log interval when operating according to the non-incremental mode.

By the result in [12], the above overhead is minimized for $\chi_F = \left\lceil\sqrt{\frac{2}{P_r}\frac{\delta_{LB}S_F}{\delta_e}}\right\rceil$, and we denote as χ_F^{opt} the optimal non-incremental log-interval according to this equation. Also, as suggested in [12], an upper bound of 40 is operatively selected for χ_F^{opt} in order to enable efficient fossil collection of obsolete event-buffers[7].

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

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

where

S_P	is the average size of a partial (incremental) log.
X_I	is the selected log-interval when operating according to the incremental mode.
$X_{I,F}$	is the interleave step between full and incremental logs (number of incremental log operations after which a full-log is taken).
δ_m	is the per-event cost for running the memory-update tracking module.

7. Events with timestamp between the newly computed GVT and the timestamp of latest available log preceding GVT cannot be garbage collected to allow correct coasting forward.

In equation (2), the term $S_F\delta_{RB}$ accounting for the cost of state reload from the log is comparable to the one in equation (1), due to the aforementioned mechanism, according to which all the in-use chunks belonging to the state image are restored (by retrieving them either from the incremental logs along the log chain, or the first full log found during the log chain backward traversing procedure). Further, each event is charged with the memory-update monitoring overhead δ_m, which also appears during costing forward. By exploiting the same arguments used in [12] for the minimization of the overhead vs the log interval, we get that the optimum value for the interval of incremental logs can be computed as $\chi_I = \left\lceil\sqrt{\frac{2}{P_r}\frac{\delta_{LB}S_P}{\delta_e+\delta_m}}\right\rceil$, and we denote as χ_I^{opt} the optimal interval according to this equation (which we still upper bound by the value 40). This value is related to the ideal case where $\chi_{I,F}$ tends to infinite. However, such an unbounded value could prevent fossil collecting obsolete logs. Hence we have opted for setting $\chi_{I,F}$ on the order of 10, which, by literature results [32], is a well suited setting providing no significant additional overhead due to full logs, when interleaving full and incremental logs.

3.3 Autonomic Optimization

In ASM, the log/restore overhead models derived in the previous section are not used to simply select as the best operating log mode the one for which the corresponding expected overhead is minimal (once identified the best log-interval value). Instead, the selection step keeps into account fluctuations that can affect the set of parameters appearing within the overhead models (e.g. the expected event execution cost δ_e), which cannot be directly controlled since they depend on proper run-time dynamics related to the simulation model execution. This set includes all the parameters appearing within the models, except the log-intervals χ_F and χ_I (or $\chi_{I,F}$), that can be controlled at run-time by ASM.

Such an approach, aimed at proactively providing stability of the optimal performance, well fits performance optimization when the set of possible operating modes is differentiated, each of them providing different overhead sensibility vs parameter fluctuations and/or variations. Literature approaches for log/restore optimization do not cope with such a multiple operating-mode scenario, which is the reason why sensibility of the a-priori uniquely selected operating mode vs parameter variations did not require to be addressed. Further, as shown in literature [13], when dynamically optimizing the parameters driving the log/restore subsystem in optimistic PDES environments, the same optimization process may give rise to secondary effects (particularly, throttling or thrashing effects) which slightly modify the actual dynamics in terms of, e.g, the rollback probability experienced by the LPs. Having an approach which selects and configures the log/restore operating mode on the basis of stability criteria vs variations of run-time parameters can also cope with such indirect effects.

Overall, the best suited operating mode is selected on the basis of on a cost function $CF(\chi_F^{opt}, \chi_I^{opt})$ defined as:

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

and on the result of the integration of this cost function over a multi-dimensional domain defined by the values of the parameters $(\delta_e, \delta_m, \delta_{LB}, \delta_{RB}, P_r, S_F, S_P)$.

For each parameter x defining a dimension of the integration domain, we integrate the cost function over the interval $\bar{x} \pm \alpha\bar{x}$, where we suggest $\alpha = 0.1$ to capture statistically relevant fluctuations of the parameters that can be envisaged at the time the dynamic selection is carried out. If the integration result is negative, then the selected operating mode is non-incremental (with the log-interval set to χ_F^{opt}), otherwise the incremental mode is selected (with log-interval set to χ_I^{opt}). Assuming the independence of the parameters defining the integration domain (which is reasonable in our approach since the mean values are operatively determined by direct sampling of the corresponding stochastic processes – see Section 3.4), the integral function for $CF(\chi_F^{opt}, \chi_I^{opt})$ is a polynomial that, after the substitution of the integral domain variables, has the following expression:

$$\left(\frac{2\alpha\bar{S}_F}{\chi_F^{opt}} - \frac{(\chi_{I,F}-1)2\alpha\bar{S}_P + 2\alpha\bar{S}_F}{\chi_I^{opt}\chi_{I,F}}\right)2\alpha\bar{\delta}_{LB} + 2\alpha\bar{P}_r\left(\frac{\chi_F^{opt}-\chi_I^{opt}}{2}2\alpha\bar{\delta}_e - \frac{\chi_I^{opt}-1}{2}2\alpha\bar{\delta}_m\right) - 2\alpha\bar{\delta}_m \tag{4}$$

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

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

3.4 Run-time Parameter Sampling

ASM relies on a run-time sampling process for computing the mean of each parameter used to define the integration boundaries within the multi-dimensional integration domain. One relevant difficulty is related to the fact that the mean value of every parameter x appearing in the performance models needs to be tracked over time independently of the current operating mode of the log layer (incremental vs non-incremental). This is because the mean is used both to trigger the re-selection process of the best suited log mode, and to determine the outcome of the selection. Accordingly, the parameters δ_m and S_P, used to capture run-time costs proper of the incremental log mode, require to be sampled (or estimated) even when the non-incremental mode is currently operating. Ad-hoc schemes to address this issue will be provided and discussed in this section. We do not explicitly discuss how to sample P_r since we rely on typical approaches (see [12]) based on counting the number of rollbacks over an interval of executed events.

3.4.1 Event and Memory-Update Tracking Costs

To determine event and memory-update tracking costs, ASM relies on a sampling mechanism based on the hardware tick counter. This approach is not intrusive since the current number of ticks can be retrieved via a single machine instruction, namely `rdtsc`.

A per-LP counter $Count$ is used to determine the number of invocations of the memory-write tracking routine occurring during the processing of each event. In case the current log mode is incremental, the application level modules whose execution is currently triggered with invocation of the proper callback entry point (according to the dual-version-code scheme presented in Section 3.1.7) embeds the memory-update tracking routine, which increments $Count$ upon its execution.

In order to be able to determine such a counter value also when running in non-incremental log mode (where memory updates' tracking is not active), we have slightly modified the dual-version code generation procedure in such a way that the code version running when non-incremental log/restore is active embeds a very light instrumentation scheme where each memory-write instruction is preceded by a single "`INC m32`" assembly instruction allowing the update of $Count$. In this way, we can infer the value of δ_m^i (namely the cost for memory updates tracking for the i-th event processed by the LP) by simply multiplying the i-th sample of the counter value by an estimated value of the memory updates' tracking routine $\delta_{tracking}$. This approach requires instrumenting memory-writes with negligible overhead (via a single machine-instruction), hence not altering the validity of the overhead model in expression (1), describing the case of non-incremental logging, which excludes costs associated with instrumenting instructions.

Of course, to estimate an accurate value of monitor's execution time when running in non-incremental mode, some samples coming from real execution of the monitoring routine should be used. To cope with this issue, we can startup the simulation with incremental logging as the default initial mode and exploit the estimation of $\delta_{tracking}$ performed during the initial phase.

We note that the above mechanisms based on real-time clocks directly fit cases where the computing platform is dedicated to the parallel simulation run, as typical of scenarios where performance is a critical factor. In case of time-sharing with other applications, such an approach needs to be complemented with solutions based on code pre-analysis and lightweight run-time profiling [14].

3.4.2 Size of Full and Partial Logs

Samples S_F^i of the full-log size can be taken independently of the currently active log mode since the memory-map manager keeps an accumulator recording at any time the real memory occupancy of the LP state image (in terms of the amount of bytes associated with currently allocated chunks). Hence, S_F^i samples are taken by querying the memory-map manager.

A different approach is instead required for taking S_P^i samples of partial (incremental) logs. Specifically, when the currently active log mode is incremental, the memory-map manager updates a second accumulator accounting for the amount of bytes associated with chunks that have been dirtied since the last log. The accumulator is updated on the basis of actual memory-write operations that are tracked at run-time. The value of this accumulator is directly used as a valid S_P^i sample when the incremental log mode is active.

In case the current log mode is non-incremental, the above accumulator does not get updated. Hence, we have decided to infer the value of S_P^i according to the following different approach. Each $K \times \chi_F^{opt}$ non-incremental log operations, we flag the corresponding LP so that, after the subsequent event is executed by the LP, we compare chunk by chunk the current memory image content after the event with the last one packed within the log buffer. The comparison is carried out only over chunks that belong both to the memory image packed within the log buffer, and to the current memory image, hence taking into account the portion of the state layout that is stable across the two subsequent snapshots. Obviously, the cost of this operation depends on the value of K and on specific optimizations for the comparison of each couple of chunks. As for the second aspect, we have developed efficient, ad-hoc assembly modules that iteratively compare memory areas by fully exploiting the size of CPU registers at each compare step, and that exactly implement the early stop procedure upon the detection of the first different byte between the two chunks (which is not guaranteed by all implementations of the classical `memcmp()` function). This matches the chunk-based granularity offered by ASM's log/restore approach. Also, these modules are optimized in order to maximize the likelihood of actual early stop in case of different chunks between the two snapshots according to the following scheme. Small size chunks are checked within the comparison process by starting from the top byte, and then going towards the bottom. Instead, for large chunks, we have implemented a procedure that checks the bytes in an interleaved mode starting from the top and from 3/4 of the chunk size. The above approach well fits typical programming practices, which tend to structure records in such a way that the most frequently touched data are at the top of the record and/or at the bottom (see, e.g., pointers for linking between memory scattered dynamically allocated records). Hence, for large chunks, it is better to check top/bottom portions with higher priority. Also, starting from 3/4 of the chunk size accounts for internal chunk fragmentation, due to the typical un-correlation between the size of the record to be placed by the application software within the allocated chunk, and the actual size of the chunk that best fists the allocation request. Once identified the dirty chunks over the aforementioned stable portion of the snapshot, the corresponding percentage p of dirty bytes is applied to the total current state size S_F^i to generate the j-th S_P sample as $S_P^j = p \times S_F^i$. Concerning the value of K, we have used a static approach where K is set to the value 20. Given that the cost associated with the estimation procedure for a single S_P sample is, at worst, comparable with the one for a full-log operation[8], this would simply increase the real overhead experienced when the non-incremental mode is active by, at worst, 5% of the corresponding logging overhead. By these optimizations, the overhead for determining S_P samples when the non-incremental mode is operative is likely negligible, thus again not altering the validity of the non-incremental overhead model in equation (1).

3.4.3 Per-Byte Log/Restore Costs

The last parameters involved in the sampling process are the per-byte log/restore costs, namely δ_{LB} and δ_{RB}. However, δ_{RB} does not appear in the final formula and we concentrate on δ_{LB}. To sample δ_{LB}, we have again exploited `rdts`, in combination with the sampling

8. Memory compare operations are similar in cost to memory copies, since they both involve similar memory/register data moves. Also, the early stop approach should favor the latency of comparing the chunks across the stable portion of the snapshot.

process of S_F and S_P depicted in the previous section. In particular, the latency of the i-th log operation, say Δ^i_{log}, is sampled via `rdts` and is normalized to either the corresponding S_F sample, or the corresponding S_P sample, just depending on the currently active log mode. Given that Δ^i_{log} also accounts for the cost of manipulating and logging metadata associated with the logged chunks, the normalization allows taking samples for δ_{LB} actually expressing how the metadata management cost is charged on the log operation of each single byte.

4 Experimental Results

4.1 The ROOT-Sim Platform

We have integrated ASM within the ROme OpTimistic Simulator (ROOT-Sim) [10, 11], an ANSI-C/MPI-based open-source optimistic simulation platform based on the Time Warp protocol [8] and tailored for UNIX-like systems. ROOT-Sim is designed as a general-purpose solution, supporting differentiated simulation models adhering to a very simple and intuitive programming model. The platform transparently handles all the mechanisms associated with parallelization (e.g., mapping of LPs on different kernel instances, namely processes).

The programming model supported by ROOT-Sim is based on the following API: i) `int ProcessEvent(int me, time_type now, int event_type, void *content, int size, void *state)` – a callback that gives control to the application for event processing. `me` identifies the dispatched LP, `now` is the event timestamp, `event_type` is event numerical code, `content` is the buffer keeping `size` bytes of event payload, and `state` is the pointer (to be set to the current value of `state_base_address`) allowing the LP to access its state in memory. ii) `int ScheduleNewEvent(int where, time_type timestamp, int event_type, void *content, int size)` – this function injects a new simulation event within the system, to be destined to whichever LP identified by `where` (the other parameters have the meaning as before). iii) `int OnGVT(int me, void *state)` – a callback that passes control to the application, providing the LP snapshot belonging to the committed-computation part.

By the API, the application programmer is requested to reason on no aspect related to parallelism. She is only requested to understand that what is coded within the `ProcessEvent` callback will be executed speculatively. Hence, any audit on the simulation model state-trajectory (when considering committed state updates) will need to be carried out via inspection on the LPs' states through the `OnGVT` callback.

4.2 Benchmark Application

We have run experiments to assess the overall performance and behavior of ASM by relying on the Personal Communication System (PCS) benchmark, which models a mobile network adhering to GSM technology. Each LP models the state's evolution of an individual hexagonal cell, and the whole set of cells provides wireless coverage on a square region of variable size. Each cell handles a parameterizable 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 result in [33].

The event types which can occur at any LP 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; *Recompute Fading*, which simulates the effects of climatic variations onto the fading (and consequently interference) phenomena for ongoing calls.

Upon the start of a call, a call-setup record is instantiated via dynamically-allocated data structures, which is linked to a list of already active records within that same cell. Each record is released when the corresponding call ends or is handed off towards an adjacent cell. In the latter case, a similar call-setup procedure is executed at the destination cell. Upon call setup, power regulation is performed, which involves scanning the aforementioned list of records for computing the minimum transmission power allowing the current call setup to achieve the threshold-level SIR value. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a meteorological model defining climatic conditions (and related variations).

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 since the more the busy channels, the more power-management records are allocated and consequently scanned/updated during the processing of different events. On the other hand, higher values of the channel utilization factor lead to higher memory requirements for the state image of individual LPs. Both the above dependencies (namely, CPU demand and memory) are anyhow bounded depending on the total number N of per-cell managed channels.

To study the effects of ASM when considering differentiated execution and memory access patterns for the application layer, we use two different configurations of the PCS application. In one configuration we simulate 1024 cells, each one managing up to 1000 wireless channels, where the expected duration of a call $\tau_{duration}$ has been set to 120 sec, the residual residence time for an active call in the current cell τ_{change} has been set to the value 300 sec, while the inter-arrival time τ_A has been varied during the simulation so to generate a configuration where the actual load on the cells depends on the period of the day. Specifically, 17

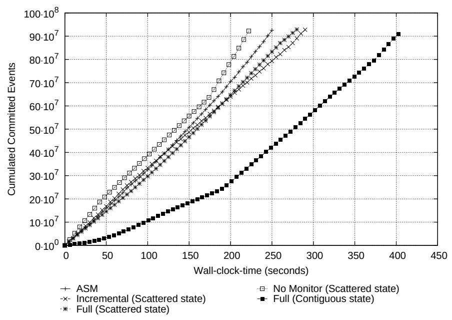


Fig. 4: Variable τ_A

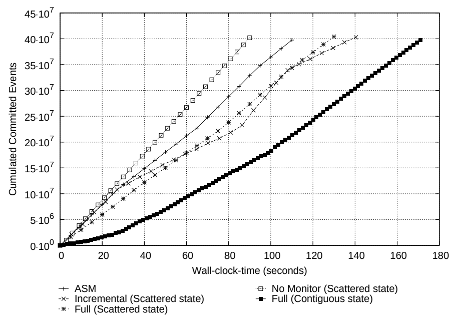


Fig. 5: Frequent fading recalculation

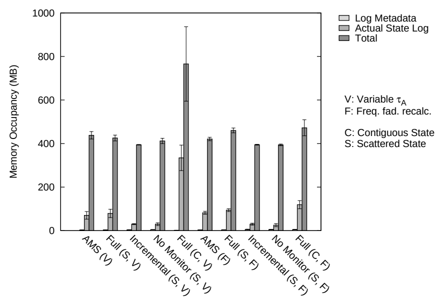


Fig. 6: Memory usage

hours of operativity of the cellular system have been simulated (from 00:00 AM to 17:00 PM) with variations of τ_A in the interval [0.64, 3.20], with peak intensity of the workload during the morning until lunch time, and minimum load very early in the morning (around breakfast). Consequently, the utilization factor has been varied in the interval [0.31, 0.06]. For this configuration of the PCS model, climatic conditions have been set as good and steady, thus not causing the need for frequent recalculation of fading coefficients. We will refer this configuration to as "Variable τ_A". On the other hand, the second configuration of PCS has been parameterized by having the expected inter-arrival time τ_A fixed to the value 0.8 (giving rise to channel utilization values on the order of 25%), which leads to focusing the simulation on a morning operativity scenario, but where the climatic conditions exhibit variations that lead to periods where frequent recalculation of fading coefficients needs to be operated. We will refer this configuration to as "Frequent fading recalculation". Both the above configurations lead to run time dynamics that vary, e.g., in terms of event granularity and portion of the LP state that needs to be updated by the events, however this is achieved in different manners in the different scenarios.

4.3 Results

In all the configurations of the PCS benchmark, we have evenly distributed 1024 LPs on top of 32 simulation-kernel processes, each of them being mapped onto one CPU-core. We have run experiments on a 32-core HP ProLiant server equipped with 32GB of RAM and running Debian 6 on top of the 2.6.32-5-amd64 Linux kernel.

We report in Figure 4 and in Figure 5 the cumulated committed events achieved by the parallel run vs wall-clock-time. These values have been computed as the average over ten runs (done with different pseudo-random seeds), with a minimal variance observed across different runs. This parameter (and the slope of the associated curve) indicates the speed according to which a given platform configuration commits simulation events, and hence how fast the configuration supports model execution. We report five plots referring to (i) the case in which ASM is active (ii) the case in which ASM is active, but we always force the incremental log/restore mode, with the corresponding optimized value for χ_I and (iii) the case in which ASM is active but the full log/restore mode is forced, with the corresponding optimized value for χ_F, (iv) the case where we force the incremental log mode by having the application layer directly calling the memory map manager for notifying which portions of the state have been updated by event processing, and (v) the case in which the application code was modified so to avoid using dynamic memory, hence leading to the situation where the state buffer for each LP is pre-allocated at startup in the form of an array of entries. The plots for cases (ii) and (iii) express performance levels that could be achieved via an optimized log/restore mode (adaptive in the selection of the log interval) based on either the incremental or the non-incremental log mode, as supported by ASM, but not allowing autonomic switch between the two modes. On the other hand, the plots for case (iv) represent scenarios that benefit by optimal checkpoint interval calculation and incremental state log/restore, but require the intervention of the programmer in relation to some of the tasks enabling incremental logging, thus offering a transparency level which is strictly lower than the one offered by ASM. Hence, this case allows quantifying the performance penalty associated with full state-management transparency as provided by ASM. Finally, case (v) is representative of scenarios where no facilities other than the bare minimal log and restore operations are supported, and without any infrastructure allowing for dynamic memory handling, thus requiring the state to be contiguous and statically sized to the maximum value admitted by the model parameterization. This is a baseline for the evaluation of the advantages by ASM.

By the results, we see that, depending on the specific phase within the simulation run, (e.g. early morning vs lunch time for the variable τ_A configuration) forced-incremental and forced-full modes alternately exhibit better execution speed (which is indicated by the different slope of the cumulated committed events curve while the run is in progress). Anyway, the most important outcome by the cumulated event rate plots is that ASM always switches to the best performing mode (incremental vs non-incremental) depending on the currently simulated period, and hence depending on the actual dynamics (e.g. in terms of state size, event granularity, memory update pattern, etc.). The overall effect is that ASM allows faster execution, on the order of 10% to 14% over the other modes for the case of the variable τ_A configuration, and on the order of 11% to 27% for the case of frequent fading recalculation. Given that the other modes represent anyway optimized configurations, the achieved improvements show high effectiveness by the autonomic approach. Compared to configuration (iv), ASM shows a slowdown varying between 10% and 20%. However, configuration (iv) removes all the transparent facilities provided by ASM in terms of identification and notification to the memory map manager of the portions of the LP state that have been dirtied. Finally, compared to the baseline configuration (v), ASM has a throughput increase ranging from 35% to 40%, which indicates how an enhancement in programmability (via the transparent support for dynamic memory allocation) is strictly coupled with a non-negligible performance increase.

In Figure 6 we report average per-process memory usage for all the considered configurations. In particular, we show average memory occupancy for the whole simulation process (i.e., simulation-platform layer and application-level model), for state logs, and for log meta-data. In all the runs we have set the GVT (and memory recovery) period to 1 sec, which gives rise to negligible coordination overhead (given the tight coupling of the underlying architecture) while allowing prompt release of memory buffers. Also, the memory usage samples refer to the state of the processes as observed right before performing memory recovery. As we can see, memory requirements for metadata is very reduced (on the order of 1%) in any configuration, highlighting memory efficiency by the data structures keeping track of memory allocation. The overall average memory occupancy shows a greater variance when dealing with phase-interleaved configuration of the PCS benchmarks, due to the fact that some phases execute more coarse-grained events and therefore require less logs per time unit. In both the frequent fading recalculation and the variable τ_A configurations, the forced full snapshot execution mode has a higher memory requirement, which is a predictable result due to the higher amount of information which is stored into a snapshot. However, such a memory consumption remains significantly lower than the one for the baseline case (v), especially for the variable τ_A configuration, which gives rise to better locality still favoring performance. At the same time, the configurations relying on the forced incremental snapshot mode and the one where the application layer calls the memory map manager to explicitly update the dirty portion of the memory map show a memory usage for logs which is very comparable, indicating similar dynamics in terms of logging frequency, which confirms how the 10% to 20% performance loss by ASM vs configuration (iv) is essentially related to the overhead for transparently handling memory updates via instrumentation. Finally, the memory usage for logs by AMS always stands between the forced incremental and the forced full ones, which reflects AMS' behavior switching from one configuration to the other.

We also report in Table 1 the execution time for running the PCS applications (very same code used for the parallel runs) in serial mode on top of a calendar queue scheduler. By the data, the parallel runs with ASM allow significant speedups, especially for the frequent fading recalculation setting. Overall, the experimental study has been carried out with competitive parallel executions.

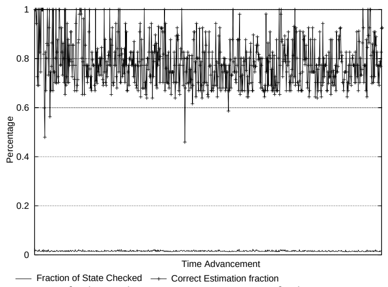


Fig. 7: Estimation of the dirty portion of the LP state and effects of the early stopping scheme

PCS configuration	execution time	speedup by the parallel run with ASM
variable τ_A	6400	25.6
frequent fading recalc.	4442	40

Tab. 1: Speedup values

To complement these results, we report in Figure 7 two additional plots related to the goodness of internal tasks/dynamics within ASM. One plot expresses the precision in the estimation of the dirty portion of the LP state, when executing according to the non-incremental mode. These data refer to the frequent fading recalculation setting, since with this configuration we have a relatively stable size of the whole LP state but with a very variable read/write access pattern within the state image, which represents a good test case for the target objective. We recall that the estimation is based on chunk comparison between successive state images only over the stable portion of the state snapshot (see Section 3.4.2). In particular, we report the ratio between the estimated size of the dirty portion of the LP and the actual size (as observed by actually tracking memory updates while executing according to the incremental mode). The plotted curve refers to a fraction of the whole simulated time interval, however, the data are representative of the overall simulation model execution dynamics. By the data we see that the average error in the estimation process is on the order of no more than 20%. The second plot expresses the effects of the early stop approach to chunk comparison (see again Section 3.4.2). In particular, we report the ratio between the actual number of compared bytes (across two subsequent state images) and the total amount of bytes forming the dirty portion of the reached state image. Given that our implementation of the simulation model collocates frequently accessed fields associated with on-going call records at the top of the records, the early stop approach provides actual advantages by allowing chunk comparison to be executed only over 5% of the stable portion of the snapshot. We expect such an optimization to even provide scale-up advantages for generic simulation models entailing scaled-up LP state size, provided that the above common field collocation approach in the used memory chunks is adopted.

5 Conclusion

In this paper we have presented ASM, an innovative autonomic state management subsystem targeted at optimistic Parallel Discrete Event Simulation (PDES) engines. ASM provides full transparency of state log/restore to the application layer, and at-runtime autonomic re-selection of the best suited log mode (incremental vs non-incremental) depending on the actual runtime dynamics of the optimistic simulation run, accounting as well for stability of the selected mode vs fluctuations in such dynamics. Mode switching is supported by ASM via an application transparent dual-coding mechanism, allowing to run the application code that best fits the requirements of the currently active log mode. Experimental results for an assessment of the proposal have been shown as well for a case study on a PCS simulation application run on top of a 32-core NUMA machine.

References

[1] R. M. Fujimoto, “Parallel discrete event simulation,” *Communications of the ACM*, vol. 33, pp. 30–53, Oct. 1990.

[2] D. E. Martin, T. J. McBrayer, and P. A. Wilsey, “WARPED: A time warp simulation kernel for analysis and application development,” in *Proceedings of the 29th Hawaii International Conference on System Sciences*, p. 383, IEEE Computer Society, 1996.

[3] C. D. Carothers, D. W. Bauer, and S. Pearce, “ROSS: a high performance modular Time Warp system,” in *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pp. 53–60, IEEE Computer Society, May 2000.

[4] F. Quaglia and R. Baldoni, “Exploiting intra-object dependencies in parallel simulation,” *Information Processing Letters*, vol. 70, no. 3, pp. 119–125, 1999.

[5] R. M. Fujimoto, “Exploiting temporal uncertainty in parallel and distributed simulation,” in *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 46–53, IEEE Computer Society, May 1999.

[6] W. Cai, S. J. Turner, B.-S. Lee, and J. Zhou, “An alternative time management mechanism for distributed simulations,” *ACM Transactions on Modeling and Computer Simulation*, vol. 15, pp. 109–137, Apr. 2005.

[7] K. M. Chandy and J. Misra, “Distributed simulation: A case study in design and verification of distributed programs,” *IEEE Transactions on Software Engineering*, vol. SE–S5, pp. 440–452, Sept. 1979.

[8] D. R. Jefferson, “Virtual Time,” *ACM Transactions on Programming Languages and System*, vol. 7, pp. 404–425, July 1985.

[9] C. D. Carothers and K. S. Perumalla, “On deciding between conservative and optimistic approaches on massively parallel platforms,” in *Winter Simulation Conference*, pp. 678–687, 2010.

[10] A. Pellegrini, R. Vitali, and F. Quaglia, “The ROme OpTimistic Simulator: Core internals and programming model,” in *Proceedings of the 4th International Conference on Simulation Tools and Techniques*, ICST, 2011.

[11] HPDCS Research Group, “ROOT-Sim: The ROme OpTimistic Simulator - v 1.0.” http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/, Oct. 2012.

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

[13] B. R. Preiss, W. M. Loucks, and I. D. MacIntyre, “Effects of the checkpoint interval on time and space in Time Warp,” *ACM Trans. Model. Comput. Simul.*, vol. 4, no. 3, pp. 223–253, 1994.

[14] F. Quaglia, “A cost model for selecting checkpoint positions in Time Warp parallel simulation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, pp. 346–362, Feb. 2001.

[15] H. Soliman and A. Elmaghraby, “An analytical model for hybrid checkpointing in Time Warp distributed simulation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 947–951, october 1998.

[16] A. C. Palaniswamy and P. A. Wilsey, “An analytical comparison of periodic checkpointing and incremental state saving,” in *Proceedings of the 7th Workshop on Parallel and distributed simulation*, pp. 127–134, ACM, 1993.

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

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

[19] R. Toccaceli and F. Quaglia, “DyMeLoR: Dynamic Memory Logger and Restorer library for optimistic simulation objects with generic memory layout,” in *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pp. 163–172, IEEE Computer Society, 2008.

[20] F. Quaglia and A. Santoro, “Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 593–610, june 2003.

[21] R. M. Fujimoto, J. Tsai, and G. Gopalakrishnan, “Design and evaluation of the rollback chip: Special purpose hardware for Time Warp,” *IEEE Transactions on Computers*, vol. 41, no. 1, pp. 68–82, 1992.

[22] “SPEEDES.” http://www.speedes.com, 2005.

[23] S. R. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, “GTW: a time warp system for shared memory multiprocessors,” in *Proceedings of the 26th Winter Simulation Conference*, pp. 1332–1339, Society for Computer Simulation Intl, 1994.

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

[25] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, “Efficient optimistic parallel simulations using reverse computation,” *ACM Transactions on Modeling and Computer Simulation*, vol. 9, pp. 224–253, july 1999.

[26] D. W. Bauer and E. H. Page, “An approach for incorporating rollback through perfectly reversible computation in a stream simulator,” in *21st International Workshop on Principles of Advanced and Distributed Simulation*, pp. 171–178, IEEE Computer Society, 2007.

[27] S. K. Seal and K. S. Perumalla, “Reversible parallel discrete event formulation of a tlm-based radio signal propagation model,” *ACM Trans. Model. Comput. Simul.*, vol. 22, pp. 4:1–4:23, Dec. 2011.

[28] R. Vitali, A. Pellegrini, and F. Quaglia, “Towards symmetric multi-threaded optimistic simulation kernels,” in *Proceedings of the 26th International Workshop on Principles of Advanced and Distributed Simulation*, pp. 211–220, IEEE Computer Society, Aug. 2012.

[29] D. Lea, “A memory allocator.” http://g.oswego.edu/dl/html/malloc.html, 1996.

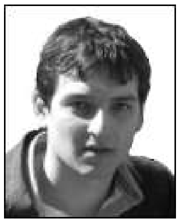
[30] R. Wahbe, S. Lucco, and S. L. Graham, “Practical data breakpoints: Design and implementation,” in *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–12, 1993.

[31] R. Vitali, *Design of Software Support Structures for High Performance Optimistic Simulations with Special Focus on Multi-Core Hosting Environment*. PhD thesis, Computer Science Departement, DIS, Sapienza Univerità di Roma, Italy, 2013.

[32] R. Vitali, A. Pellegrini, and F. Quaglia, “Benchmarking memory management capabilities within ROOT-Sim,” in *Proceedings of the 13th International Symposium on Distributed Simulation and Real Time Applications*, IEEE Computer Society, 2009.

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

Alessandro Pellegrini is a PhD student at DIAG at *Sapienza, University of Rome*, collaborating with the *High Performance and Dependable Computing Systems*, where he is working in the area of Distributed Systems and Parallel Simulation. He achieved the Bachelor's degree in Computer Engineering in 2008 and the Master's degree in Distributed Systems and Computer Architectures in 2010. His other research interests include Autonomic Computing and Compilers.



Roberto Vitali is a PhD student at *DIAG* at *Sapienza, University of Rome*. He is a member of *High Performance and Dependable Computing Systems* research group. He achieved the Bachelor's degree in Computer Engineering in 2007 and the Master's degree in Distributed Systems and Computer Architectures in 2009. His research interests are in Distributed Simulation, Computer Architectures and Operating Systems.

Francesco Quaglia received the Laurea degree (MS level) in Electronic Engineering in 1995 and the PhD degree in Computer Engineering in 1999 from the University of Rome “La Sapienza”. From summer 1999 to summer 2000 he held an appointment as a Researcher at the Italian National Research Council (CNR). Since January 2005 he works as an Associate Professor at the School of Engineering of the University of Rome “La Sapienza”, where he has previously worked as an Assistant Professor since September 2000 to December 2004. His research interests are in distributed systems and protocols, middleware platforms, parallel discrete event simulation and federated simulation systems. He has served as Program Co-Chair of PADS 2002, PADS 2010 NCA 2007 and SIMUTools 2011, as General Chair of PADS 2008 and as General Co-Chair of SIMUTools 2012.