Transparent State Management for Optimistic Synchronization in the High Level Architecture

Andrea Santoro and Francesco Quaglia

SIMULATION 2006; 82; 5
DOI: 10.1177/0037549706065350

Transparent State Management for Optimistic Synchronization in the High Level Architecture

Andrea Santoro
Francesco Quaglia
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
via Salaria 113, 00198 Roma, Italy
santoro@dis.uniroma1.it
quaglia@dis.uniroma1.it

In this article, the authors present the design and implementation of a software architecture—namely, MAgic State Manager (MASM)—to be employed within a runtime infrastructure (RTI) in support of High Level Architecture (HLA) federations. MASM allows performing checkpointing/recovery of the federate state in a way completely transparent to the federate itself, thus providing the possibility of demanding to the RTI any task related to state management in optimistic synchronization. Different from existing proposals, through this approach, the federate programmer is required neither to supply modules for state management within the federate code nor to explicitly interface the federate code with existing, third-party checkpointing/recovery libraries. Hence, the federate programmer is completely relieved from the burden of facing state management issues. One major application of this proposal is the possibility to employ optimistic synchronization, even in case of federates originally designed for the conservative approach. This can provide a way of improving the simulation system performance in specific scenarios (e.g., in case of poor or zero lookahead within the federation). The authors elaborate on this issue by discussing on how to integrate MASM within the RTI to achieve such a synchronization objective. Some experimental results demonstrating limited runtime overhead introduced by MASM are also reported for two case studies—namely, an interconnection network simulation and a personal communication system simulation.

Keywords: HLA, federated simulation systems, transparent checkpointing/recovery, middleware

1. Introduction

The High Level Architecture (HLA) is a standard for the integration of autonomous simulators into a single, distributed simulation system [1]. The autonomous simulators are usually known as *federates*, while the resulting distributed simulation system is known as the *federation*. Each federate interacts with the rest of the federation, employing a middleware called the runtime infrastructure (RTI), which provides a general set of services [2].

A major problem to address in HLA is how to ensure correct order (i.e., timestamp ordering) for the execution of simulation events at each federate. To cope with this problem, the HLA specification defines a suite of services, called Time Management, to be offered by the RTI in support of synchronized execution among federates. Although Time Management offers an interface to accommodate both conservative and optimistic synchronization, most work in the HLA context is oriented toward conservative simulation. The main reason is that optimistic synchronization typically requires the federate code to embed a set of mechanisms, among which are checkpointing and state recovery, to correctly support such a synchronization scheme.

Recent works attempted to tackle issues related to optimistic synchronization in HLA federations via the introduction of so-called rollback managers (or controllers) [3, 4]. The objective is to move the handling of most part of rollback procedures (e.g., the retraction, or cancellation, of events) to the middleware in support of the federation. However, these solutions still make use of *GetState* and *SetState* callbacks to the federate code in order to let the federate collect a snapshot of its state and pass it to the middleware or reload its state to a value passed by the middleware via the callback. In other words, the federate software is still required to implement modules for handling state-related actions in support of optimistic synchronization. Moreover, such callbacks are not prescribed by HLA standards.

SIMULATION, Vol. 82, Issue 1, January 2006 5-20

DOI: 10.1177/0037549706065350

In this work, we propose a software architecture allowing completely transparent checkpointing/recovery of the federate state. Once integrated within the RTI, our architecture allows optimistic synchronization to be carried out without the need for having modules for state management within the federate software. Actually, the activation of state management functionalities within the architecture does not require passing through any explicit interfacing between the federate and RTI, thus not even requiring the federate to offer state management callbacks to RTI. Hence, our proposal also has the advantage of maintaining the interface of RTI compliant to the HLA standard.

As we shall describe in detail, the software architecture, which we refer to as MAgic State Manager (MASM), has been developed for LINUX systems and is composed of a set of user-level and kernel-level modules. These modules allow the following:

- Identification of the portion of the memory image of the federate (i.e., the federate state) at any time, within the memory image of the whole application—namely, federate plus RTI
- Checkpointing/recovery of the portion of the memory image related to the federate at a given simulation time

The capabilities of MASM rely on low-level management of the application address space, which allows MASM to treat the federate state independently of the semantic of the specific simulation model. In addition, low-level management of the page table within the operating system, also performed by MASM, allows supporting checkpointing of the federate state efficiently by performing snapshots based on the incremental copy of the modified (dirty) pages only.

As hinted above, our target is to allow the implementation of optimistic synchronization without the need for having state management modules within the federate code. Therefore, the objective of MASM is to achieve transparency for state management at the level of the federate programmer. A direct consequence is that MASM can be used to design an RTI capable of supporting optimistic synchronization, even in the case of federates only interfacing to the conservative portion of Time Management services (which typically occurs for most COTS simulation packages, whether they be based on parallel or sequential simulation approaches). For example, this could be achieved by having the RTI deliver unsafe events to the federate, even if the federate is originally designed for conservative synchronization (or simple sequential execution); in the case of timestamp order violations, rollback actions can be executed by the RTI code transparently to the federate, exploiting checkpointing and recovery functionalities offered by MASM. This approach would allow simplification of simulation software production since federate programmers could rely on timestamp-ordered delivery of events independently of the fact that applications will be federated according to the optimistic synchronization scheme.

In other words, beyond transparency, our proposal is in the direction of widening the spectrum of possibilities for synchronization within HLA federations independently of the specific nature (conservative or optimistic) of the involved federates. This has the strong advantage of coping with performance problems related to (1) poor or zero lookahead within the specific federation and (2) large costs for the preventive computation of event safety in case the simulation system is hosted by an infrastructure with, for example, nonminimal delivery delay among the different instances of the RTI.

The remainder of this article is structured as follows. In section 2, we describe MASM, including implementation details. In section 3, we focus on design issues related to the integration of MASM within the RTI to transparently support optimistic synchronization for conservative federates. In section 4, we discuss relations between MASM and prior works. An experimental evaluation of the proposed architecture is presented in section 5.

2. MAgic State Manager (MASM)

In this section, we present the architecture of MASM. At first, we provide a high-level description of the main approaches used by MASM to support checkpointing and recovery of the state of a federate. Then we enter details of the implementation for LINUX systems.

2.1 Main Approaches for Checkpointing and State Recovery

In typical implementations (see, e.g., Georgia Tech Research Corporation [5]), both the federate and the RTI are parts of a same process within the operating system. However, as we shall discuss later while presenting implementation details, exploiting appropriate techniques (such as assembler and linker options) and ad hoc management of dynamic memory allocation within RTI, it is possible to identify all the virtual memory regions reserved for either RTI data or federate data. We refer to those regions as **MD** (Middleware Data) and **FD** (Federate Data). Furthermore, those regions can be obtained as aligned to the page within virtual addressing so that each virtual memory page either belongs to **MD** or **FD**. The only exception is the application stack since its pages might contain data related to both RTI and the federate due to the fact that RTI software can be activated by the federate via standard function calls; similarly, software modules within the federate can be activated by RTI via callbacks.

By the previous arguments, we are able to identify the entries of the page table associated with virtual pages within each of the regions **MD** and **FD**. Hence, except for the stack that needs a special treatment presented later, the page table can be seen as logically divided into two differ-

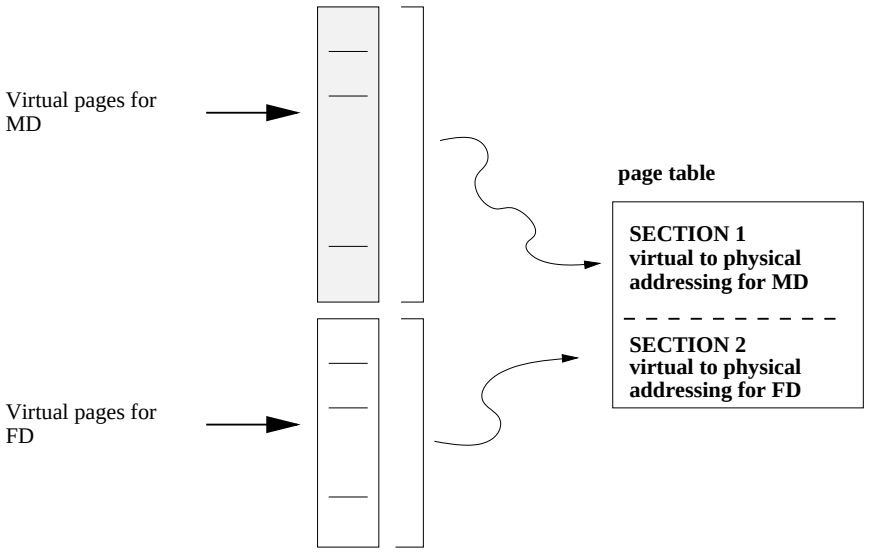


Figure 1. Virtual memory addressing and page table sections

ent data sections referring to mapping of virtual to physical addresses for the two regions **MD** and **FD**. A schematization of this is shown in Figure 1. We note that MASM is designed to be used as a software component for RTI; hence, in our schematization, the **MD** region is also used to denote virtual memory areas accessed by MASM (e.g., for keeping the values of proper data structures it requires).

Based on this schematization, the MASM architecture allows copying the content of virtual memory pages belonging to **FD** within memory buffers belonging to **MD**—that is, memory buffers managed by MASM (hence by RTI) to perform checkpointing. At the same time, MASM allows restoring the content of virtual memory pages belonging to **FD** by reloading data previously saved into memory buffers belonging to **MD**.

We have designed MASM in order to let it deal with checkpointing of the federate state efficiently for what concerns both CPU time and memory usage. Specifically, on the top of the LINUX virtual memory system, we have developed software modules allowing MASM to take a complete snapshot of the federate state (i.e., of the content of **FD**) by copying only those virtual pages that have been modified since the last snapshot. This is achieved by exploiting standard protection mechanisms within the LINUX virtual memory system together with the previously hinted logical partitioning of the page table into separate sections for **MD** and **FD**. Specifically, we have embedded within MASM software modules that allow marking as write-protected into the page table all the virtual memory pages belonging to **FD**. Also, we have augmented the standard page fault handler for LINUX with the capability to detect a page fault occurring due to MASM memory protection on any page within **FD**. Hence, we are able to identify all those virtual memory pages within **FD** that have been updated (one or more times) since the last mark operation. This allows the identification of all the federate data that have been modified due to the execution of one or more simulation events since the last mark operation. Copying those modified pages within MASM buffers allows taking a checkpoint of the whole federate state by only copying the incrementally modified virtual memory pages within **FD**. This is shown in Figure 2, where the gray pages denote the initial snapshot of the content of **FD** at simulation time T', and the dark gray pages denote a copy of those virtual pages within **FD** that have been modified due to event executions up to time T''. As shown by the dotted line, the union of the dark gray pages C and D, plus the additional gray pages A and B within the snapshot at time T', constitutes the snapshot of the federate state at simulation time T''. Hence, in case of future rollback occurrence, we can reload the federate snapshot at either T' or T'' depending on the simulation time for the causality violation. An important observation we would like to bring to the reader's attention is that, even though the snapshot of the federate state is built incrementally, we do not need to backward apply all the incremental logs to perform state recovery. We simply need to identify, for each virtual page to be recovered within **FD**, the corresponding page to reload from the log. As we shall show, this can be performed by adequately designing data structures allowing, for each logged virtual page, the identification of the simulation time interval for which its content is valid. As an example, the logged gray page A is valid (i.e., can be reloaded into the corresponding page within **FD** upon state recovery) in case of rollback to either time T' or T''; hence, it is valid within such a whole interval of simulation time. Instead, the logged dark gray

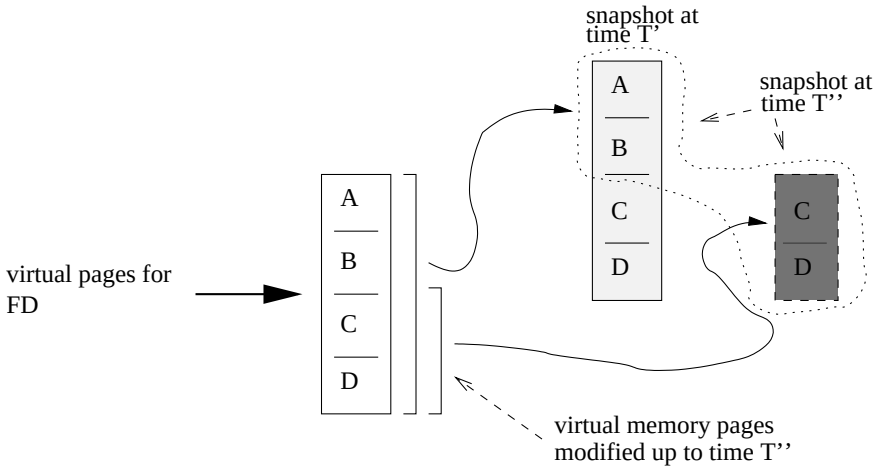


Figure 2. Snapshots of the federate state at simulation time T' and T''

page C cannot be reloaded in case of rollback to time less than T'', hence exhibiting a different interval of validity within simulation time.

2.2 Stack Management

In the previous section, we outlined the basic mechanism used to cope with checkpointing and recovery of the memory image of the federate for what concerns all the federate data that are outside the application stack. This section is instead specifically devoted to the stack management and to techniques we have adopted to support checkpointing and recovery of this region of the address space of the application.

In an HLA federation, the execution always starts within federate software, and RTI software is activated on demand through calls to proper services provided by RTI. As a consequence, the application stack is always organized in a way that the initial portion contains data (including flow control information) related to the state of the federate. Also, in the case of a federate that needs synchronization supports (e.g., a time-constrained federate in the HLA terminology), RTI provides grants to the federate in order to let its simulation clock advance. As a consequence, each time we enter RTI software, we know that the initial stack portion related to the federate represents part of the federate snapshot at a given simulation time. An example is shown in Figure 3, where we have three different configurations for the stack at different simulation times T, T', and T''.

By the previous consideration, the same basic scheme used for the **FD** portion of the address space, which has been described in section 2.1, can be adopted to support checkpointing of the stack portion related to the federate. Specifically, once RTI has control, we can perform the incremental copy of modified virtual pages belonging to the federate stack portion into proper buffers maintained by MASM. This is shown in Figure 3(b), where the snapshot of the stack portion related to the federate at time T' can be built incrementally by combining part of the snapshot at time T and the portion of the stack that has been modified while the federate simulation clock has moved from T to T'.

However, the problem with the application stack is state recovery for the federate since the stack portion related to the federate might have different sizes at different simulation times. As a consequence, simple reload of virtual memory pages related to the federate stack content at an earlier simulation time might compromise the correct execution flow within RTI software. This problem can be more clearly explained through the following example, still referring to Figure 3. Consider the case of a federate having simulation clock T'', and suppose RTI has now control; hence, the application stack contains at least one record for RTI data. This case corresponds to Figure 3(c). Also, suppose the RTI decides that we need to perform rollback of the federate state at simulation time T'. If we performed state recovery by simply copying back the stack snapshot for the federate taken at simulation time T', we would overwrite RTI data within the application stack itself. This is because the size of the stack referring to the federate at time T' is larger than the one at current simulation time T''.

To address this problem, we have decided to design MASM in order to support a switch of the stack when we enter the RTI software. As we shall describe in detail later, this solution is based on the preventive allocation of a virtual memory region to be used as the stack when entering RTI, as well as on the use of a proper assembly layer manipulating stack pointer registers. With this solution, we can avoid overwriting RTI records within the stack, even in case the federate portion of the stack is recovered to a larger stack size compared to the current one.

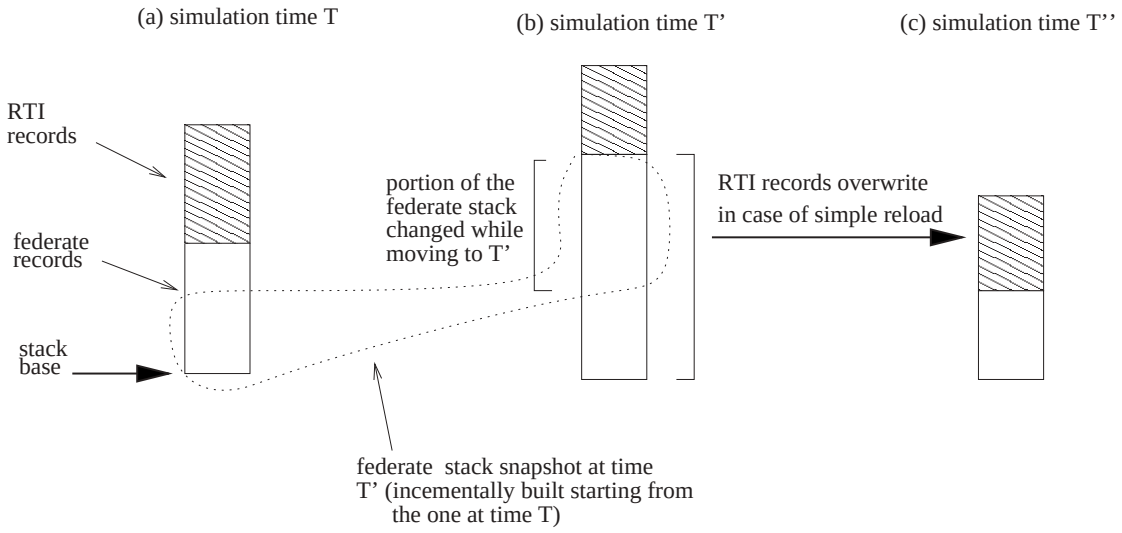


Figure 3. Application stack at simulation time T, T', and T''

As a final observation, similar to the case of **FD** described in the previous section, the use of an adequate data structure within MASM, which we shall illustrate, allows performing recovery of the federate stack without the need for backward applying all the incremental logs of virtual memory pages. Specifically, the data structure permits, for each logged virtual page belonging to the federate stack, the identification of the simulation time interval for which its content is valid.

2.3 Implementation Details

In this section, we present details related to the implementation of MASM. Actually, the implementation has been developed on top of LINUX kernel 2.6.6 with machine-dependent portions of the software developed for IA-32 compliant hardware. We recall, however, that machine-dependent modules, required for the manipulation of the page table and CPU registers, represent a minimal portion of the whole MASM architecture. Hence, MASM might be adapted to different hardware systems with minimal programming effort. Anyway, we are currently investigating the possibility to increase the level of portability at the expense of limited additional overhead by avoiding machine-dependent modules and kernel patches. Also, the implementation is tailored to classical monothread and asynchronous multithread RTI process models, supported by many commercial and noncommercial implementations (e.g., [5-7]).[1]

1. To our knowledge, there is a single runtime infrastructure (RTI) commercial implementation employing a pure multithread process model [8], according to which the RTI performs callbacks to the federate code while the federate concurrently proceeds with computation in a separate thread. The current implementation of MAgic State Manager (MASM) cannot cope with this type of process model since it assumes the federate does not touch its state while checkpointing is in progress (i.e., it assumes that the log of the federate data is taken as an atomic action). We plan to cope with this type of process model in future implementations.

We proceed in this section by first describing solutions dealing with the separation of **FD** and **MD** into two different sections within the page table. Then we provide details related to the stack management. Finally, we overview modules for the management of the page table (including those modules augmenting the LINUX page fault handler) and then provide a description of the main data structures used to maintain logs of virtual memory pages for the federate.

2.3.1 Separation and Identification of **FD** *and* **MD**

MASM must be aware of which virtual addressing memory ranges are part of **FD**. Also, the **FD** region must not share virtual pages with **MD**. To achieve this, our solution relies on the following:

- The observation that **FD** can simply be viewed as "all the allocated virtual memory not belonging to **MD**." Hence, knowing the start and the end points of **MD** automatically allows the determination of the **FD** region.
- The use of proper techniques to force adequate placement of **MD** (i.e., page aligned) within virtual addressing.

We base our solution on the so-called *incremental linking*. This linking technique allows the building of a single relocatable object, starting from all the RTI object files (including MASM files), with the advantage that each of its sections (text, data, and BSS) is located within a contiguous virtual addressing range. In other words, with incremental linking, we are able to ensure that **MD** is composed of a few virtual memory blocks instead of being nondeterministically scattered throughout the whole virtual addressing range for the application.

To make MASM aware of start and end addresses for those virtual memory blocks and to ensure that those blocks are aligned to the virtual page, we have exploited again incremental linking plus two simple assembler modules we have developed. Specifically, a START module is used to mark the start of the data and BSS sections. An END module is used to mark the end of those sections. To perform the mark operation, each of the two modules contains a symbol for the data section and a symbol for the BSS section. This is shown in Figure 4 for what concerns the START module, where the symbols `__data_RTI_start` and `__bss_RTI_start` are used. The modules also contain alignment assembler directives, so that the start and end points of **MD**, identified by the corresponding symbols, are aligned to the page.

Incremental linking with START and END modules linked at the head and at the tail, respectively, allows us to place those symbols at the head and the tail of their related sections within virtual addressing. Thus, MASM does only need to reference those symbols to identify virtual addressing ranges destined for data maintained by RTI—namely, those forming the **MD** region. Also, it knows that those ranges are automatically aligned to the virtual page.

Now, to determine the virtual page ranges forming the **FD** region, we only need to consider the whole virtual addressing for the application and exclude the regions associated with the previous symbols. This is achieved in our implementation by considering that the whole virtual addressing for the application can be completely identified since the linker automatically inserts symbols to mark the start and end of each section in the memory address space of the executable file (e.g., the `__end` symbol for the end of the whole virtual addressing area).

Actually, the solution proposed above is used by MASM to identify and keep track of the static (i.e., compile-time defined) portion of **FD**. However, it is common for applications to dynamically allocate memory (e.g., through the `malloc` library). This is accomplished by extending the area destined to the heap by using the proper system call (i.e., `brk()` on LINUX) and organizing this additional memory space by maintaining a directory of free and allocated memory segments. MASM is confronted with the following two different issues concerning dynamic memory allocation.

The first issue is that, during its computation, the federate might decide to extend the heap area. However, a state recovery might need to recover a snapshot of **FD** whose heap area had a limit different from the current one. We have addressed this problem by considering the value of the heap area limit as part of a snapshot of the federate at a given simulation time. Upon state recovery, we set the limit back to the value it had at the time the snapshot was taken. `sbrk()` and `brk()` were used by MASM to read the limit and set it back (if necessary).[2]

```
#define PAGE_SIZE 4096

.data
.align PAGE_SIZE
.global __data_RTI_start
__data_RTI_start:

.bss
.align PAGE_SIZE
.global __bss_RTI_start
__bss_RTI_start:
```

Figure 4. The START module to mark data and BSS portions for the runtime infrastructure (RTI)

The second issue is connected to the fact that both the federate and RTI might require dynamic memory allocation, but the data maintained by the `malloc` library must only be located within **FD** to allow correct state recovery for the federate (i.e., to allow correct undoing of dynamic memory allocation performed by the federate). This also means that the RTI cannot use directly the `malloc` services since any memory allocation by the RTI might be reverted in case of a state recovery for the federate. To allow the RTI to still use dynamic memory, any call to the `malloc` library from the RTI object file is intercepted at linking time and redirected toward a memory allocator internal to MASM. Actually, the memory allocator we currently provide within MASM has a simple implementation, possibly subject to future improvements, based on static memory reserved by MASM at compile time and managed according to the first fit policy for what concerns "free holes" of virtual memory to be assigned to RTI.

It is important to note that the previous issues related to the `malloc` library can be framed by the more general problem related to the handling of third-party libraries employed by both the federate and RTI. If we place data managed by those libraries within **FD**, then it means those data belong to the recoverable portion of the application. This, however, does not mean we can really recover any

2. Critical situations in which the `malloc` library might use the `mmap()` system call to reserve additional memory outside the heap when `brk()` fails have not yet been addressed.

action performed by a call to a function within the library since it might have an interaction with the operating system, whose actions cannot be assumed to be recoverable. Considering, for example, the `stdio` library, state recovery on data managed by this library does not mean we are able to recover an input/output (I/O) operation on the terminal (this is the well-known problem of the output commit in rollback recovery [9]). With respect to this point, except for the `malloc` library, we suggest to the RTI developer who decides to use MASM to link all the libraries having an interaction with the operating system to the RTI portion of the code through the incremental technique described above. This allows imposing that data maintained by those libraries are not recoverable.

As a final observation, the approach we have taken of allowing recovery of a third-party library by placing its data within **FD** has straightforward application in case of static linking of libraries. Otherwise, modification of the loader of dynamic libraries is required to properly place data related to a dynamic library within **FD**.

2.3.2 Implementation of Stack Switch

As already pointed out in section 2.2, the stack cannot be recovered by simply reloading its logged virtual memory pages at a given simulation time. This is because the stack portion to be recovered cannot be assumed to have the same size of the stack portion being replaced, which might cause overwriting of RTI records currently within the stack itself when RTI has control and attempts to perform a state recovery for the federate. As also pointed out, to solve such a problem, anytime we enter an RTI function that can ultimately yield to a checkpoint (or, equivalently, a rollback), we perform a stack switch, moving the RTI stack portion in a different memory area, statically allocated by MASM. Note that, once the stack switch is performed, we can proceed deeper into nested calls within RTI without the need for switching again.

Our implementation of stack switch relies on a proper option in the linker (e.g., the `–wrap` option if the linker is `ld`), which allows the redirection of the RTI function (or functions) into proper wrappers we have developed, which are available within MASM. Actually, we need a different wrapper for each RTI function whose activation needs stack switch. This is because the wrapper itself needs to keep the name of such RTI function, which is defined at compile time (we only need the function prototype to generate the wrapper). We are currently investigating the possibility to support wrapping of multiple RTI functions with a single parameterized wrapper.

Each wrapper performs the following steps:

- It saves the stack and frame pointers of the current stack (i.e., the federate stack) into proper locations—namely, `federate_stack_pointer` and `federate_base_pointer` managed by MASM. This is achieved through assembly instructions nested within the wrapper. The same occurs for general-purpose registers.
- It copies into the new stack region a small portion at the top of the original stack, containing the parameters passed by the federate to the currently invoked RTI function, and the local variables. Actually, to ensure that parameters are really passed through the stack, we can use the `regparm(0)` compiler directive (we note that this is actually the default choice for standard `gcc` compilers).
- It performs the real stack switch operation by loading adequate values into the stack and frame pointers within the architecture. This is also achieved through assembly instructions nested within the wrapper.
- It invokes the RTI function originally invoked by the federate through a `call` instruction. The same parameters originally specified by the federate are passed to the function since they have already been copied (see point 2) into the new stack area.
- It puts the original stack back in place together with general-purpose registers (as though nothing happened) when the invoked RTI function returns.

Every time a checkpoint of the federate takes place, MASM takes as part of the checkpoint also the stack and frame pointers, as well as the general-purpose registers, saved in step 1. When a recovery takes place, the stack and frame pointers are copied back in place, together with the virtual memory pages to be recovered within the federate stack. Therefore, when the RTI releases control after a recovery, the federate will resume with the same stack content and the same stack/frame pointers it had before the checkpoint.

2.3.3 Tracking Modified Pages

The checkpointing operation is demanded to the application level of MASM. However, to perform the checkpoint by incrementally logging only the modified pages of the federate portion of the application, MASM relies on the following kernel-level mechanism.

MASM maintains in its memory space a bitmap representing the mapping of all user-memory pages of the application. This bitmap is initialized to zero, and its address is notified to the kernel by a system call `notify_bitmap(void *addr)` we have added to the kernel API. We have also developed an additional system call—namely, `mark()`—which allows MASM to mark all the entries in the page table corresponding to pages of **FD** and of the federate stack as write-protected. Actually, to keep the kernel modifications to a minimum, we do not mean for these pages to be copy-on-write (CoW) pages, and thus we need a mechanism to distinguish them from simple CoW pages. In the IA-32 architecture (as in

other architectures), there are unused bits inside the page table entries, which are available for kernel designers. We have reserved one of these bits to distinguish between a real CoW and MASM write protection.[3] When the federate attempts to write in one of its write-protected pages, it will generate a page fault. Thus, the page fault handler will be run, and by checking both the write protection and the bit reserved for MASM within the page table, it will be able to decide whether the fault is a real CoW or a MASM-related fault. In the latter case, it will run the MASM fault handler, which performs the following two steps:

- Sets to 1 the bit corresponding to the faulting page into the bitmap
- Sets the page as unprotected, so that no other fault will occur on that page until a subsequent mark operation is executed

At this point, the program resumes as if nothing happened, except that when a federate page is write-accessed at least once after a mark operation, the corresponding bit in the bitmap is set to 1. By checking the bitmap, MASM detects which pages changed since the last checkpoint and saves only the modified pages (incremental checkpointing). After taking a checkpoint, the bitmap is reset, and a new mark operation is executed so to allow the MASM fault handler to be able to notify future changes of the federate state occurring after the current checkpoint operation.

We note that, in the context of checkpointing in support of fault tolerance, there has been a proposal to address the tracking of the modified pages by only relying on application-level modules exploiting UNIX standard system calls for protecting memory and detecting write-access through UNIX signals [10]. Although the solutions underlying this proposal would help portability, they do not directly cope with incremental checkpointing of the stack area. Also, they do not cope with the management of two stack portions, among which only one is subject to checkpointing and recovery actions. Both these tasks need to be performed by MASM, which is the main reason why we have decided not to rely on these solutions while starting with MASM design.[4] Anyway, as future work, we surely plan to analyze whether and how extensions/modifications of these solutions can cope with all the needs of MASM. We note, however, that, compared to such an application-level-based scheme, our approach is expected to provide higher performance effectiveness thanks to the execution of operations related to the ad hoc memory management scheme directly within the kernel.

3. For generality, we note that in architectures that do not reserve bits for designer usage, one could simply allocate another bitmap into the kernel, telling, for each page, whether the protection is used by MASM or by the kernel.

4. The need for handling two stack portions has been already discussed in section 2.2. Compared to solutions in support of fault tolerance, incremental checkpointing of the federate stack needs to be addressed due to the more pressing performance requirements of checkpointing in support of optimistic simulation. These derive from the fact that checkpointing typically needs to be performed more frequently than what happens in the fault tolerance field due to the fact that rollback in optimistic simulation is an endemic phenomenon.

2.3.4 Data Structures Keeping Track of Checkpoints

As pointed out, MASM performs checkpointing of the whole federate state by incrementally logging only the modified pages. However, upon state recovery, we need to identify all the logged pages that need to be reloaded to reconstruct the federate state at a given simulation time. As hinted in section 2.1, this is achieved by associating with each logged copy of the xth page within virtual addressing a simulation time interval for its validity. Specifically, MASM maintains for each federate page within virtual addressing a double-linked list of entries structured as in Figure 5, where the fields `start_time` and `end_time` identify the simulation time interval for the logged page validity, and the `page_address` field points to the logged page (recall this page is maintained within MASM managed buffers). When we need to perform state recovery at simulation time T, MASM scans the list associated with any page to be recovered and determines the logged version of the page that is valid at time T. Such a page is copied back into its original virtual addressing location (this location is uniquely identified by the index of the page we are recovering). Actually, to determine which pages need to be recovered, MASM exploits both (1) the logged value of the stack pointer valid at time T, which determines how many pages need to be recovered starting from the stack base, and also (2) the logged value of the limit of the **FD** area at time T (see the discussion on dynamic memory allocation and virtual addressing boundaries in section 2.3.1), which determines how many pages need to be recovered for data/BSS and heap portions related to the federate. Those logged values are also maintained by MASM through adequate lists.

```
struct log_entry{
    double start_time;
    double end_time;
    void * page_address;
    struct log_entry * next;
    struct log_entry * prev;
}
```

Figure 5. Data structure for the double-linked list of logged versions of each virtual page

3. On the Integration within the RTI

As outlined in the Introduction, a major application of MASM is the possibility to support optimistic synchronization in the case of federates originally designed for conservative synchronization. This requires integration of MASM within the RTI to provide a middleware layer capable of performing optimistic synchronization tasks transparently for what concerns both state management and messaging (e.g., retraction of events scheduled for remote federates). With respect to this issue, we provide in this section a description of how MASM access points can be mapped over the RTI conservative interface for Time Management. In addition, we outline the capabilities that must be built inside the RTI software to correctly support all the previous tasks when relying on MASM. Since we want to keep all the following considerations applicable to the widest possible spectrum of RTIs, our discussion intentionally remains at the level of design indications, hence not entering details on how MASM can be interfaced with RTI internals, for which there is even no standardization.

According to HLA specifications [2], the typical service used for conservative synchronization of event-driven federates is *next message request* (NMR).[5] This service allows the federate to ask for an advancement of its local simulation clock to a given value t, specified as an input parameter to the service invocation (typically this is the time of the next event in the federate local event queue). The effect of an NMR invocation is the delivery of the safe TSO (timestamp ordered) message with the minimum timestamp destined to the federate and having time up to t, if any. In case the RTI knows that no message will be ever delivered to the federate with simulation time less than or equal to t, the RTI grants to the federate the advancement to time t. On the other hand, in case there is a chance that a future message will arrive for the federate with time less than or equal to t, the RTI will eventually deliver the incoming safe TSO message with the minimum simulation time to the federate and then will grant the federate with the simulation time of that message. The grant is delivered to the federate through the *time advance grant* (TAG) callback invoked by the RTI.

The optimistic interface employs a mechanism similar to NMR/TAG but with two key differences. (1) The NMR access point is substituted with the *flush queue request* (FQR) access point, which allows a federate to request the delivery of all the incoming TSO messages, either safe or unsafe, up to time t. (2) A *retract* (R) service is used by the federate in a rollback phase to inform the RTI that a certain message should have never been sent, and all associated computation should be undone. Invocation of R might result in the *request retraction* (RR) callback on the remote federate(s) in case the delivery of the message to be unsent has been already executed by the destination RTI(s) (if the delivery has not yet been executed, then R will simply result in an annihilation of the information to be delivered, which is performed internally by the RTI).

On the other hand, MASM offers to the RTI the following access points (no callbacks are needed):

`SaveState()`. When the RTI invokes this service, MASM performs an incremental checkpoint of the federate state. Among the parameters passed to this service, we have the current simulation time for the federate, which is associated by MASM with the currently taken checkpoint.

`RecoverState()`. When the RTI invokes this service, MASM restores a previously checkpointed state. Among the parameters passed to this service, we have the simulation time determining which checkpointed state is selected by MASM from the log for the recovery procedure.

`PruneStateLog()`. When the RTI invokes this service, MASM frees the memory that keeps the oldest checkpoints. Among the parameters passed to this service, we have the simulation time determining which checkpoints can be removed from the log.

A general organization for an RTI embedding MASM, which can transparently support optimistic synchronization of conservative federates, could be as follows. The federate normally performs its computation without interferences from the RTI. Only, when a page is modified, that page is marked as "dirty" by the MASM memory fault handler. When the federate needs to advance its simulation clock, it will be using the NMR (or TAR) service, according to the conservative interface. However, when the RTI receives the NMR (or TAR) request, it can first call the `SaveState()` service offered by MASM to take a checkpoint of the federate state, by incrementally saving the dirty pages only, and then can deliver to the federate the requested message(s), regardless of their actual safety. After the delivery of the requested message(s), the RTI should also deliver the TAG to the federate, with either the time of the NMR (or TAR) request or the time of the delivered message(s), in order to let it go ahead with the computation.

Naturally, it might happen that some messages (or requests for RR callbacks) might arrive to the RTI after an unsafe message with a later simulation time has already been delivered to the federate. Since a federate designed for a conservative synchronization has no way to cope with this situation, the RTI must be equipped with a mechanism to detect out-of-sequence messages. When this happens, the RTI must also trigger state recovery for the federate by using the `RecoverState()` service offered by MASM. Moreover, after the state is recovered and the computation is resumed, the RTI will be asked by the federate to deliver again the messages whose simulation time is higher

5. Time-stepped federates will rely on the similar service *time advance request* (TAR).

than the state recovery time. (Recall that, with the previous scheme, checkpoints are taken upon the NMR call. Hence, a federate always resumes computation from the point of an NMR call, which just asks for the delivery of message(s).) This means that the RTI must have kept those messages buffered. Moreover, HLA specifications prescribe the possibility to handle so-called receive ordered (RO) messages [2]. These messages must be delivered to the federate in the same order they have been received by the RTI. Given that a state recovery procedure for the federate executed by the RTI through `RecoverState()` also undoes the delivery of RO messages (starting from those delivered after the simulation time for the recovered state was reached), the RTI must also keep RO messages buffered for redelivery after a rollback. In addition, a rollback might entail the need to cancel some messages previously sent to other RTI instances. Although the way to actually implement the cancellation of sent messages is dependent on the specific RTI software, it seems likely that in most RTIs, the software used to implement the R access point of the optimistic interface might be used also for canceling messages sent by the conservative interface.

We also note that, in case of a federate exhibiting piecewise deterministic (PWD) behavior [9], checkpointing might even not occur at each NMR (or TAR) invocation since intermediate states between different invocations can be reconstructed through a local replay phase with filtered messaging toward remote federates performed by the RTI (i.e., like a coasting forward phase in classical optimistic synchronization [11]). This might help optimize the runtime trade-off between checkpointing and state recovery costs.

Finally, when the RTI advances the Lower Bound on TimeStamps (LBTS) for the federate, it can recover storage used for both saved state information (this can be executed by using the `PruneStateLog()` access point offered by MASM) and for "obsolete" messages it keeps currently buffered.

4. Related Work

Apart from the already discussed results on the rollback controller approach (i.e., [3, 4]) recently proposed just in the context of HLA, our work is also related to a number of other results in the field of checkpointing in the context of both traditional parallel discrete event simulation (PDES) and fault tolerance. However, it exhibits significant differences from any of those results, as we shall discuss below.

For PDES systems, several checkpointing solutions have been introduced that are based on logging the whole state of a simulation object (at each event execution or after an interval of executed events) [12-16], on incremental logging of changes of state variables occurring at each event execution [17-20], or on a mix of the two approaches [21, 22]. All these solutions require anyway the application programmer (1) to supply the necessary code to collect snapshots of the objects state, (2) to employ calls to functions within the API of proper checkpointing libraries, or (3) to explicitly declare which portions of the address space need to be considered part of the state. In all cases, transparency is not supplied since the programmer must necessarily be faced with issues related to the state snapshot collection. Also, explicit declaration of which portions of the address space need to be part of a snapshot makes some of these solutions unfeasible in case of dynamic memory allocation for the object state. Our solution tackles previous issues by supporting checkpointing without the need for specific log/recovery modules within the application code, or for explicit interfacing with checkpointing libraries, and by allowing checkpoints of an object state even in case it is scattered on dynamically allocated memory chunks. Different from those approaches, our solution is therefore well suited for usage within a middleware component (e.g., RTI) that aims at managing simulation object states with generic structure in a way completely transparent to the application-level programmers.

Another approach to support optimistic synchronization in PDES is the so-called reverse computation [23], where state recovery is achieved by performing the inverse of the operations that were performed during forward event computation. However, with this solution, the application programmer is required to provide both the reversible and the reversing code, which again limits transparency compared to our approach. With respect to the latter assertion, even in the case of employing compiler supports for the automatic production of the reversing code as suggested in Carothers, Perumalla, and Fujimoto [23], the level of transparency achievable is still bound by the one related to classical checkpointing techniques for PDES. This is just because reverse computation still requires checkpointing the portions of the object state that are modified through nonreversible operations (e.g., plain assignments).

In the context of fault tolerance, several solutions have been proposed to support checkpointing of a process state by exploiting memory protection and operating system facilities to detect dirty pages and incrementally log them [10, 24, 25]. However, different from our approach, those solutions cannot be employed in the context of a single process logically partitioned into a recoverable portion (i.e., the federate in the HLA context) and a portion that is not subject to rollback operations (i.e., the RTI). The novelty of our proposal resides therefore in the ability to manage separate data/BSS/stack areas for the two different portions and to recover one of these portions selectively whenever required.

5. Experimental Evaluation

We base the evaluation of MASM on the observation that this software architecture has not been designed to improve performance compared to optimized checkpointing/recovery mechanisms built in the federate code or

available to the federate programmer through proper libraries. Instead, its target is transparency, which, as also discussed in section 3, would even allow optimistic synchronization in the case of federates equipped with no state management module and interfacing with no state management library. Recall this is the case of both federates programmed to be employed with conservative synchronization and federates resulting from adaptation of sequential (legacy) simulation software (i.e., federates designed with the assumption of delivery of safe events only by the RTI). In this context, advantages from MASM should be assessed when considering that it can avoid the preventive detection of event safety in the case of HLA federations including those types of federates. With respect to the latter point, we note that, as also recently discussed in McLean and Fujimoto [26], detection of event safety (i.e., of the LBTS for a federate) is a distributed reduction to be performed by the RTI, whose latency depends on the timely participation of all other processes in the system and on the timely delivery of messages. Both these dependencies make such a computation a performance-dominating factor, especially in the case of federations with poor lookahead and/or deployed on an infrastructure where the speed of different processes can diverge while performing the reduction and/or there are periods during which the latency of message delivery can get unexpectedly long. For those situations, optimistic synchronization might offer better performance if we are able to provide an underlying checkpointing mechanism with limited overhead. Hence, one focus of this experimental study is the evaluation of the checkpointing overhead imposed by MASM while the computation proceeds. This evaluation can anyway provide indications on the relation between the efficiency of MASM and that of any optimized checkpointing mechanism built in the federate code. Specifically, if the overhead by MASM is limited, then the system performance during forward computation should not be significantly different from the one achievable with the optimized built-in approach.

Beyond the overhead in forward computation, it is also important to observe the effects of MASM when state recovery operations are executed. This might help in assessing whether the management of data structures maintained by MASM (e.g., for identifying the correct snapshot to be recovered across all the incrementally logged pages) provides effective runtime behavior. Hence, we will also report measures related to the cost of state recovery while varying the amount of undone computation.

Before showing the results of the experiments, we provide details on the used test settings and test cases.

5.1 Test Settings and Test Cases

We have not yet integrated MASM with an existing RTI, which is actually the objective of future work; therefore, the experimental study has been conducted using a stub-RTI simulating the presence of HLA middleware. We note, however, that these settings provide a highly controlled and parameterizable environment, allowing us to evaluate MASM with no interference due to factors related to tasks performed by a real RTI (e.g., interrupts associated with communication).

The stub-RTI simply collects NMR invocations by the application-level code and delivers TAG callbacks granting an advance at simulation time equal to the one requested by the application via NMR. To maintain the stub-RTI interface compliant with that of typical RTI layers (see, e.g., [5]), the TAG callback takes place only after the RTI-tick service is invoked by the federate once NMR has returned. Hence, the timing of interactions between federate and stub-RTI results in the one shown in Figure 6. Once the stub-RTI has taken control, it can activate MASM checkpointing/recovery routines (i.e., it can map MASM access points on the Time Management interface for conservative synchronization, just as suggested by the guidelines in section 3), before passing control back to the federate upon the return of the RTI-tick service.

We have used two different case studies for what concerns the application-level code (i.e., the federate code). In the first one, the application-level code simulates an interconnection network with packets transmitted according to wormhole switching and deadlock-free planar adaptive routing (PAR) [27]. In the second case study, the application-level code simulates a personal communication system with fixed base stations offering communication services to mobile devices and performing power regulation based on the signal-to-interference ratio (SIR) evaluated, considering cross-channel interference within a same cell [28].

Both simulators have been integrated with the stub-RTI with few modifications. Specifically, an NMR call to the

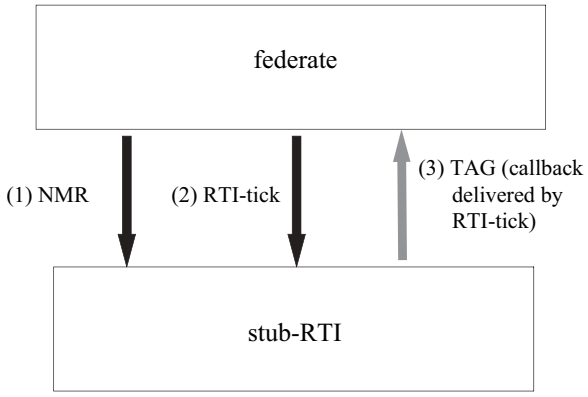


Figure 6. Timing of interactions between federate and stub-RTI

stub-RTI has been inserted before the execution of any new event. The argument for this call is the next event time in the local event queue of the federate. The next event is then executed after the receipt of the TAG callback originated by the stub-RTI. In other words, control is passed to the stub-RTI anytime the federate wants to advance its simulation time to the timestamp of the next event in its local queue.

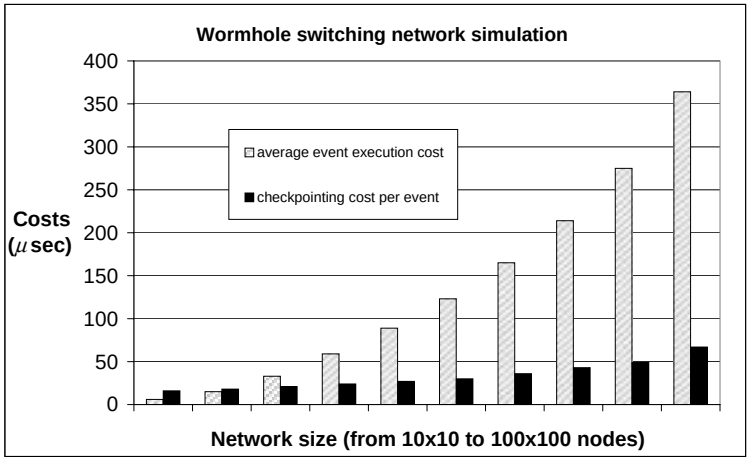
For what concerns the wormhole-switching simulation model, the network has a mesh topology with wrap-around connections, and PAR is supported through two virtual channels for each routing direction. Transmitted packets have an average size of 10 data flits (each 1 byte in size), plus head and tail flits. The interarrival time of packets to be transmitted follows an exponential distribution with a mean value of 500 units per node, while the flit transmission delay has been set to 1 unit. Also, packets are equally likely destined to whichever node in the network. Finally, we have varied the size of the simulation model from 10×10 to 100×100 interconnected nodes. For what concerns the personal communication system simulation model, we have used hexagonal cells, each with 100 channels, and have set the threshold SIR to be achieved by power regulation at about 10 DB, similar to what happens in standard GSM transmission. The movement of mobile devices follows a classical random walk model [29], with a cell switch time exponentially distributed with a mean of 10 minutes. The average call-holding time has been set to 2 minutes, and the interarrival time of calls per cell follows an exponential distribution [30-32], with a mean value of 2 seconds (this yields to a channel utilization factor of about 50% for this specific configuration). Also, we have varied the size of the simulation model from 100 to $10,000$ cells within the coverage area. For both these case studies, the local event queue of the federate relies on preallocated memory buffers for storing the events, linked as a linear list. Finally, the experiments have been carried out on a Pentium 4 2.8-GHz CPU (512-KB cache—1 GB RAM) running LINUX (kernel version 2.6.6).

5.2 Results

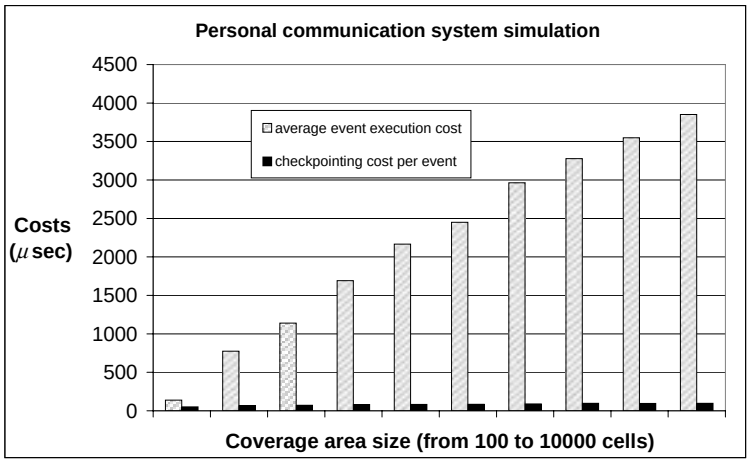
To determine the checkpointing overhead due to MASM (i.e., the overhead imposed by MASM in forward computation), we have measured both the average execution cost of the events, evaluated by considering all the execution time spent within the application-level code, and the average execution cost per event of any checkpointing-related action performed by MASM. The latter cost includes the time for incrementally logging dirty pages, the time to manage data structures maintained by MASM, and also the time for the page fault caused by the MASM protection mechanism described in section 2.3.3. Each reported value, expressed in microseconds, is the average over a number of samples ensuring a confidence interval of 10% around the mean at the 95% confidence level. Also, all the samples have been taken by letting the stub-RTI take a checkpoint at each NMR call issued by the overlying federate. By the obtained results, shown in Figure 7, we get that, for the personal communication system simulation application, the checkpointing overhead per event is low or negligible independent of the simulation model size. On the other hand, it is kept low also for the case of the finer-grain wormhole-switching simulation application, except for a very small network size.

In Table 1, we report additional data related to the variation of both the size of the federate (in terms of portion of the address space destined for the federate) and the size of the incremental log associated with each checkpoint operation. As mentioned, for both the test cases, the local event queue uses preallocated memory buffers, with an occupancy within the portion of the address space destined to the federate of about 2.5 MB for the wormhole-switching network simulator and about 2 MB for the personal communication system simulator. These data show that the memory consumption per checkpoint is kept very limited. Specifically, for the personal communication system simulator, we get an increase of 1 MB of storage used for state logs each 25.6 checkpoints for a minimal model size and each 15 checkpoints for a very large model size. This occupancy is further reduced for the wormhole-switching network simulator, where an increase of 1 MB of storage used for state logs is observed each 42.7 checkpoints for a minimal model size and each 32 checkpoints for a very large model size. By these data, we also observe that the size of the checkpoint is almost flat while varying the model size. At the same time, the checkpoint cost increases with a rate lower than the one related to both the event execution cost and the federate size. This points out how the effectiveness of MASM while managing its internal data structures (e.g., the bitmap keeping track of dirty pages) does not suffer from an increase in the data structures themselves while the federate size is largely increased. In other words, MASM runtime behavior scales well with the size of the data structures.

The final part of this experimental study is dedicated to the observation of MASM runtime behavior in the case of state recovery. Specifically, we report data related to the latency for a state recovery operation while varying the length of rollback, in terms of the number of processed events to be undone. Given that we are using the stub-RTI approach, rollbacks are injected artificially by the stub-RTI. More precisely, when the stub-RTI gains control, it periodically activates the recovery routine supported by MASM in order to recover the federate state to a past simulation time (i.e., the simulation time related to n events back from the current point). In the study, we have varied n from 1 to 100. The obtained results, shown in Figure 8 for three different model sizes (ranging from the minimum to the maximum), outline two important points. First, independent of the model size, the cost for state recovery in the case of rollback of a single event is only slightly larger than the checkpointing cost per event. Hence, such a cost is low or negligible for almost all the considered configu-



(a)

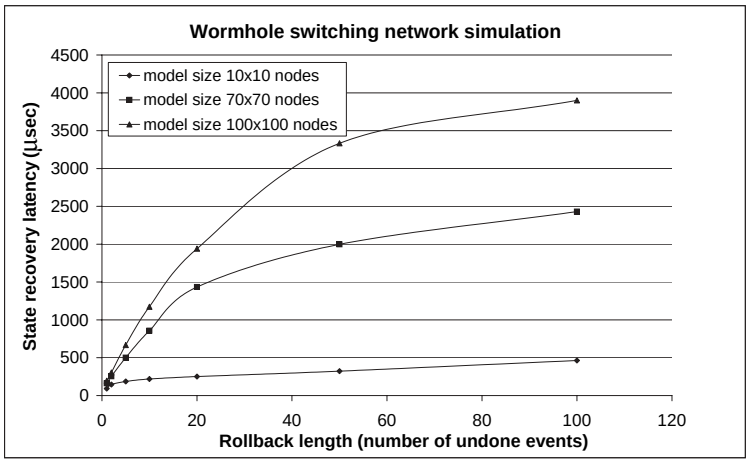


(b)

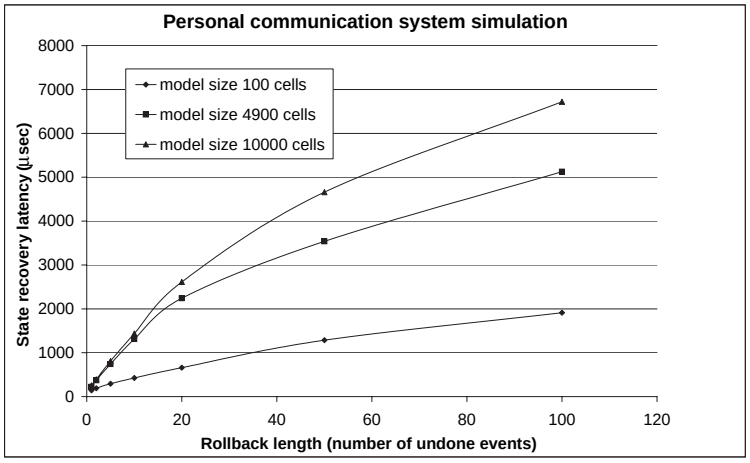
Figure 7. Event and checkpointing costs for the two case studies

Table 1. Federate size versus average checkpoint size for the two test cases

Wormhole Switching Network Simulation (from 10 × 10 to 100 × 100 Nodes)										
Federate size (Mbytes)	2.9	4.3	6.7	9.8	14.5	19.3	25.6	32.3	40.2	47.8
Checkpoint size (4-Kbyte pages)	6	7	7	7	7	7	7	7	7	8
Personal Communication System Simulation (from 100 to 10,000 Cells)										
Federate size (Mbytes)	3.2	4.9	7.7	11.1	16.2	22.1	29.5	37.3	46.2	56.7
Checkpoint size (4-Kbyte pages)	10	13	14	15	16	17	17	17	17	17



(a)



(b)

Figure 8. State recovery latency versus the rollback length

rations of both the test cases. Also, in the case of a large amount of events undone by each single rollback, the state recovery latency tends to a flat behavior, clearly indicating that the state recovery overhead per event even decreases compared to the cost observed in the case of rollback of a single event. Actually, one reason for this behavior is that, even if MASM adopts an incremental approach to checkpointing, the incremental logs do not need to be wholly reapplied to reconstruct the state to be recovered. Hence, the number of pages to be reloaded during a state recovery phase typically tends to the working set of pages updated by simulation events executed in forward computation. In our experiments, this was particularly true for the case of reduced model size, as shown by the plots related to the variation of the average number of recovered pages while varying the rollback length in Figure 9.

We note that the reduction of the rollback cost per event while the rollback length increases is an indication that speculative computation of even a nonminimal amount of simulation events can be supported with very reduced state recovery overhead, even when all those events are eventually undone. Specifically, for the personal communication

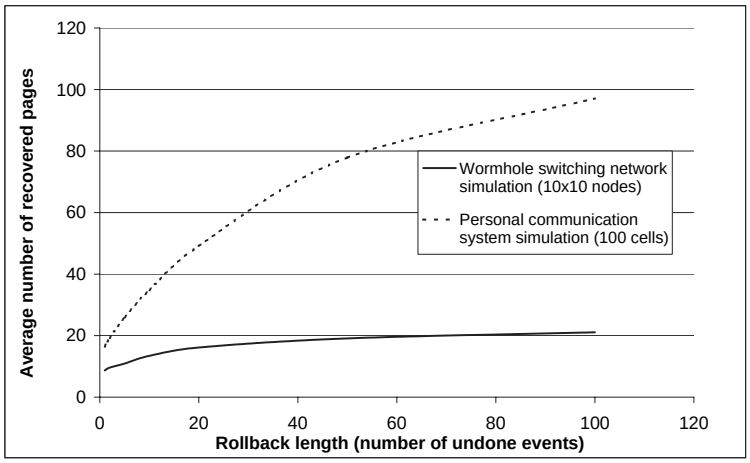


Figure 9. Average number of recovered pages versus the rollback length

system simulation application, the state recovery latency in the case of undoing 100 events in a rollback phase is about 2 msec for a minimal model size and lower than 7 msec for a maximal model size. These latencies actually represent 16% and about 2% of the corresponding costs for those same 100 events in forward computation. Slightly less favorable results are observed for the finer-grain wormhole-switching network simulation application, where undoing 100 events has a cost less than 0.5 msec for a minimal model size and 4 msec for a maximal model size. However, these costs are kept almost equal to (or even less than) the average cost for processing those same 100 events in forward computation.

Overall, the additional costs imposed for checkpointing and (in the case of rollback) state recovery by MASM when performing speculative computation result in a reduced percentage of the execution cost of the events speculatively processed. In addition, we note that the relative overhead by MASM would be even lower if it were evaluated with a real RTI implementation providing supports for a (complete) set of HLA services, which further contributes to the performance effectiveness of our proposal.

6. Conclusions

In this article, we have addressed the issue of state management in optimistic synchronization in HLA. Specifically, we have proposed the design and the implementation of a software architecture capable of transparently supporting checkpointing and recovery of the state of a federate. This software architecture can be integrated within a conventional HLA middleware component (i.e., the so-called RTI), in order to demand to the RTI all the tasks associated with optimistic synchronization. We have also performed an experimental evaluation showing low runtime overhead achieved by our architecture in support of state management transparency. As discussed, one major application of our proposal is the possibility of supporting optimistic synchronization even in the case of federates originally designed for classical conservative synchronization, which can provide improvements in the runtime effectiveness of the simulation system each time we are in the presence of, for example, federates with limited lookahead. This can be achieved with even no effort from the application programmer for what concerns the development of application-level modules supporting optimistic synchronization tasks or the interfacing of application-level software with third-party libraries supporting those same tasks.

7. References

[1] IEEE Std 1516-2000. 2000. *IEEE standard for modeling and simulation (M&S) High Level Architecture (HLA): Framework and rules*. New York: Institute of Electrical and Electronics Engineers.

[2] IEEE Std 15161-2000. 2000. *IEEE standard for modeling and simulation (M&S) High Level Architecture (HLA): Federate interface (FI) specification*. New York: Institute of Electrical and Electronics Engineers.

[3] Vardanega, F., and C. Maziero. 2000. A generic rollback manager for optimistic HLA simulations. In *Proceedings of the 4th Workshop on Distributed Simulation and Real-Time Applications*, pp. 79-85.

[4] Wang, X., S. J. Turner, Y. H. Low, and B. P. Gan. 2004. Optimistic synchronization in HLA based distributed simulation. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pp. 123-30.

[5] Georgia Tech Research Corporation. n.d. FDK: Federated Simulations Development Kit. http://www.cc.gatech.edu/computing/pads/fdk/
[6] DMSO. n.d. Runtime infrastructure (RTI). http://www.dmso.mil/public/transition/hla/rti
[7] Virtual Technology Corporation. n.d. RTI NG Pro^{TM} version 2.0.2. http://www.virtc.com/Products/prdFulltextjsp?ID=1z_RTI
[8] PITCH. n.d. pRTI 1516^{TM}. http://www.pitch.se/prti1516/default.asp
[9] Elnozahy, E. N., L. Alvisi, Y.-M. Wang, and D. B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys* 34 (3): 375-408.
[10] Plank, J. S., M. Beck, and G. Kingsley. 1995. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of USENIX Winter Technical Conference*, pp. 213-23.
[11] Jefferson, D. R. 1985. Virtual time. *ACM Transactions on Programming Languages and System* 7 (3): 404-25.
[12] Fleischmann, J., and P. A. Wilsey. 1995. Comparative analysis of periodic state saving techniques in Time Warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 50-8.
[13] Preiss, B. R., W. M. Loucks, and D. MacIntyre. 1994. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation* 4 (3): 223-53.
[14] Quaglia, F. 2001. A cost model for selecting checkpoint positions in Time Warp parallel simulation. *IEEE Transactions on Parallel and Distributed Systems* 12 (4): 346-62.
[15] Quaglia, F., and A. Santoro. 2003. Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Transactions on Parallel and Distributed Systems* 14 (6): 593-610.
[16] Ronngren, R., and R. Ayani. 1994. Adaptive checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pp. 110-17.
[17] Bruce, D. 1995. The treatment of state in optimistic systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 40-9.
[18] Ronngren, R., M. Liljenstam, R. Ayani, and J. Montagnat. 1996. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 70-7.
[19] Steinman, J. 1993. Incremental state saving in SPEEDES using C Plus Plus. In *Proceedings of the Winter Simulation Conference*, pp. 687-96.
[20] West, D., and K. Panesar. 1996. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 78-85.
[21] Franks, S., F. Gomes, B. Unger, and J. Cleary. 1997. State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pp. 72-9.
[22] Soliman, H. M., and A. S. Elmaghraby. 1998. An analytical model for hybrid checkpointing in Time Warp distributed simulation. *IEEE Transactions on Parallel and Distributed Systems* 9 (10): 947-51.
[23] Carothers, C. D., K. S. Perumalla, and R. Fujimoto. 1999. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation* 9 (3): 224-53.
[24] Li, K., J. F. Naughton, and J. S. Plank. 1994. Low latency concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems* 5 (8): 474-9.
[25] Plank, J. S., J. Xu, and R. H. Netzer. 1995. Compressed differences: An algorithm for fast incremental checkpointing. Tech Rep CS-95-302, University of Tennessee at Knoxville.
[26] McLean, T., and R. M. Fujimoto. 2003. Predictable time management for real-time distributed simulation. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, pp. 89-96.
[27] Chien, A. A., and J. H. Kim. 1995. Planar-adaptive routing: Low-cost adaptive networks for multiprocessors. *Journal of the ACM* 42 (1): 91-123.
[28] Kandukuri, S., and S. Boyd. 2002. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications* 1 (1): 46-55.
[29] Akyildiz, I. F., Y. B. Lin, W. R. Lai, and R. J. Chen. 2000. A new random walk model for PCS networks. *IEEE Journal on Selected Areas in Communications* 18 (7): 1254-60.
[30] Boukerche, A., S. K. Das, A. Fabbri, and O. Yildz. 1999. Exploiting model independence for parallel PCS network simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 166-73.
[31] Carothers, C. D., R. M. Fujimoto, P. England, and Y. B. Lin. 1994. Distributed simulation of large-scale PCS networks. In *Proceedings of the 2nd IEEE International Workshop on Modeling, Analysis, and Simulation of Compute and Telecommunication Systems*, pp. 2-6.
[32] Carothers, C. D., R. M. Fujimoto, and Y. B. Lin. 1995. A case study in simulating PCS networks using Time Warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 87-94.

Andrea Santoro *is a Research Assistant in the Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," Roma, Italy.*

Francesco Quaglia *is an Associate Professor in the Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," Roma, Italy.*