Submission for the Special Issue: Advancing Simulation Theory and Practice with Distributed Computing

A novel parallelization technique for DEVS simulation of continuous and hybrid systems.

Federico Bergero†, Ernesto Kofman†, François Cellier‡

†Laboratorio de Sistemas Dinámicos. FCEIA - UNR. CIFASIS–CONICET.
Riobamba 245 bis - (2000) Rosario, Argentina
bergero@cifasis-conicet.gov.ar kofman@fceia.unr.edu.ar
‡Modeling and Simulation Research Group ETH Zürich, Switzerland
fcellier@inf.ethz.ch

Abstract

In this paper, we introduce a novel parallelization technique for DEVS simulation of continuous and hybrid systems. Here, like in most parallel discrete event simulation methodologies, the models are first split into several sub–models that are concurrently simulated on different processors.

In order to avoid the cost of the global synchronization of all processes, the simulation time of each sub–model is locally synchronized in a real–time fashion with a scaled version of physical time, which implicitly synchronizes all sub–models. The new methodology, coined Scaled Real–Time Synchronization (SRTS), does not ensure a perfect synchronization in its implementation. However, under certain conditions, the synchronization error introduced only provokes bounded numerical errors in the simulation results. SRTS uses the same physical time–scaling parameter during the entire simulation. We also developed an adaptive version of the methodology (Adaptive-SRTS) where this parameter automatically evolves during the simulation according to the workload.

We implemented the SRTS and Adaptive-SRTS techniques in *PowerDEVS* , a DEVS simulation tool, under the Real Time Operating System (RTOS) RTAI (RealTime Application Interface). We also tested its performance in the simulation of three large models, obtaining in all cases a considerable speedup.

1 Introduction

Computer simulation has become a crucial tool for studying, developing, and evaluating mathematical models of systems of different domains (physics, economics, biology, etc.).

Continuous systems are usually represented by sets of differential equations. Since most differential equations lack analytical solutions, they must be discretized to simulate the continuous systems they represent. To this end, most numerical integration methods approximate the differential equations by discretizing the time variable transforming them into sets of difference equations[9].

Some recently developed numerical methods, however, are based on state discretization. These methods, called QSS (after Quantized State Systems)[9] transform the originally continuous systems into discrete event systems that can be represented by the Discrete Event System Specification (DEVS) formalism[42].

The usage of QSS algorithms then allows the DEVS formalism to represent and to simulate continuous and hybrid systems.

DEVS models can be easily simulated, but when the models are large, the computational requirements become critical and parallelization techniques must be employed.

Parallel Discrete Event Simulation (PDES) is a technique that can be used for the simulation of large DEVS models in multi–core or cluster environments. In PDES, the model is first split into several subsystems called physical processes. Then, the processes are simulated concurrently on different logical processors (LP).

Compared to simulating a large model in a sequential manner, PDES reduces the computational cost but it introduces a new problem related to the need of synchronization between processes. Synchronization between different LPs is required since each process needs to know the results of other processes in order to correctly simulate its own subsystem.

If the LPs are not correctly synchronized, the simulation may receive events out of order (with time–stamps lower than the actual simulation time), which can lead to an incorrect result. This is called a *causality constraint*.

There are many approaches to overcome this problem. One of the first was the one proposed by Chandy, Misra, and Bryant (CMB algorithm)[7,10,27]. CMB implements a *conservative* synchronization where none of the LPs advances its simulation time until it is *safe*.

A different approach is that proposed by Jefferson[18], where an *optimistic* synchronization mechanism is introduced relaxing the *causality constraint*. The LPs are allowed to advance their simulation time as fast as they can, and proper actions are taken when inconsistencies are detected.

Finally, a technique called NOTIME[34] explores the effects of not synchronizing the processes at all.

In this article, we present a novel PDES technique, specialized for DEVS approximations of continuous and hybrid system, called Scaled Real–Time Synchronization (SRTS). The basic idea is to synchronize each LP's simulation time with a *scaled* version of the physical time or wall–clock time. As all LPs are synchronized with the physical time, they are indirectly synchronized against each other.

As our approach is intended for continuous and hybrid systems, events represent continuous trajectories. Thus, synchronization errors provoke numerical errors. Provided that the errors are bounded, they do not invalidate the final results, hence we can relax the *causality constraints*.

While in SRTS the user must provide the real–time scaling factor as a simulation parameter, we also developed an ASRTS algorithm where the scaling factor is adjusted dynamically depending on the system workload.

We have implemented SRTS and ASRTS in *PowerDEVS*[4], a general purpose open source DEVS simulator. *PowerDEVS* implements the complete family of QSS methods on which this work is based. As we need a precise synchronization with physical time in order to achieve a precise synchronization between the processes, the implementation runs under a real–time operating system called RTAI[13].

The article is organized as follows. Section 2 introduces the main concepts used in the rest of the paper, and Section 3 describes some related work in the field. Then Sections 4 and 5

introduce and describe the SRTS and ASRTS algorithms, the main contributions of this article. Finally Section 6 shows some application examples, and Section 7 concludes the article and discusses some lines of future work.

2 Background

In this section, we introduce a few basic concepts and ideas that will be used in the article.

We first describe the DEVS formalism and the QSS methods, a family of numerical integration methods for continuous and hybrid system simulation. Then, we describe the Real–Time Operating System RTAI and the DEVS simulation tool called *PowerDEVS* . Finally, we provide a brief introduction to parallel architectures.

2.1 DEVS

DEVS stands for Discrete EVent System specification, a formalism introduced first by Bernard P. Zeigler[41].

A DEVS model processes an input event trajectory and –according to that trajectory and its own initial conditions– provokes an output event trajectory.

An *atomic* DEVS model is defined by the following structure:

$$M = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta)$$

where:

- X is the set of input event values, i.e., the set of all possible values that an input event can adopt.
- Y is the set of output event values.
- S is the set of state values.
- δ_{int}, δ_{ext}, λ and ta are functions that define the system dynamics.

Each possible state s ($s \in S$) has an associated *Time Advance* computed by the *Time Advance Function* $ta(s)$ ($ta(s) : S \rightarrow \Re_0^+$). The *Time Advance* is a non–negative real number specifying how long the system remains in a given state in absence of input events.

Thus, if the state adopts the value s_1 at time t_1, after $ta(s_1)$ units of time (i.e., at time $t_1 + ta(s_1)$) the system undergoes an *internal transition*, changing its state from s_1 to s_2. The new state is calculated as $s_2 = \delta_{\text{int}}(s_1)$. Function δ_{int} ($\delta_{\text{int}} : S \rightarrow S$) is called *Internal Transition Function*.

When the state transitions from s_1 to s_2, an output event is produced with value $y_1 = \lambda(s_1)$. Function λ ($\lambda : S \rightarrow Y$) is called *Output Function*. In this way, the functions ta, δ_{int}, and λ define the autonomous behavior of a DEVS model.

When an input event arrives, the state changes instantaneously. The new state value depends not only on the input event value but also on the previous state value and the elapsed time since the last transition. If the system entered state s_2 at time t_2 and then an input event arrives at time $t_2 + e$ with value x_1, the new state is calculated as $s_3 = \delta_{\text{ext}}(s_2, e, x_1)$ (note that $ta(s_2) > e$). In this case, we say that the system performs an *external transition*. Function δ_{ext}

($\delta_{\text{ext}} : S \times \Re_0^+ \times X \to S$) is called *External Transition Function.* No output event is produced during an external transition.

The formalism presented is also called *Classic DEVS* to distinguish it from *Parallel DEVS*[42], which consists in an extension of the classic DEVS formalism designed to improve the treatment of simultaneous events.

Atomic DEVS models can be coupled. DEVS theory guarantees that the coupling of atomic DEVS models defines new DEVS models (i.e., DEVS is closed under coupling), and thus, complex systems can be represented by DEVS in a hierarchical way[42].

Coupling in DEVS is usually represented through the use of input and output ports. With these ports, the coupling of DEVS models becomes a simple block–diagram construction. Figure 1 shows a coupled DEVS model N that results from coupling models M_a and M_b.

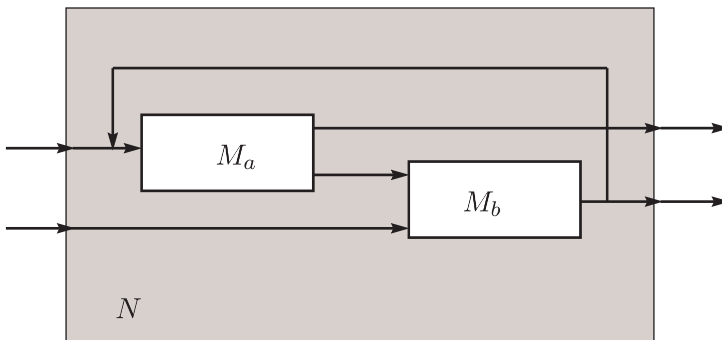

Figure 1: Coupled DEVS model

According to the closure property, the model N can itself be interpreted as an atomic DEVS and can be coupled with other atomic or coupled models.

2.2 Quantized State Systems

A continuous time system can be written as a set of ordinary differential equations (ODEs):

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \tag{1}$$

where $\mathbf{x} \in \Re^n$ is the state vector and $\mathbf{u} \in \Re^m$ is a vector of known input functions.

The mathematical model (1) can be simulated using a numerical integration method. While conventional integration algorithms are based on time discretization, a new family of numerical methods was developed based on state quantization[9]. The new algorithms, called Quantized State System methods (QSS methods), approximate ODEs like that of Eq.(1) by DEVS models.

Formally, the first–order accurate QSS method approximates Eq.(1) by

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{v}(t)) \tag{2}$$

where each pair of variables q_j and x_j are related by a *hysteretic quantization function.*

The presence of a hysteretic quantization function relating $q_j(t)$ and $x_j(t)$ implies that $q_j(t)$ follows a piecewise constant trajectory that only changes when the difference with $x_j(t)$ becomes equal to a parameter Δq_j, called *quantum.* The variables q_j are called *quantized variables.* They can be viewed as piecewise constant approximations of the corresponding state variables x_j.

Similarly, the components of $\mathbf{v}(t)$ are piecewise constant approximations of the corresponding components of $\mathbf{u}(t)$.

Since the components $q_j(t)$ and $v_j(t)$ follow piecewise constant trajectories, it results that the state derivatives $\dot{x}_j(t)$ also follow piecewise constant trajectories. Then, the state variables $x_j(t)$ have piecewise linear evolutions.

Each component of Eq.(2) can be thought of as the coupling of two elementary subsystems, a static one,

$$\dot{x}_j(t) = f_j(q_1, \cdots, q_n, v_1, \cdots, v_m) \tag{3}$$

and a dynamical one

$$q_j(t) = Q_j(x_j(\cdot)) = Q_j(\int \dot{x}_j(\tau) d\tau) \tag{4}$$

where Q_j is the hysteretic quantization function (it is not a function of the instantaneous value $x_j(t)$, but a functional of the trajectory $x_j(\cdot)$).

Since the components $v_j(t)$, $q_j(t)$, and $\dot{x}_j(t)$ are piecewise constant, both subsystems have piecewise constant input and output trajectories that can be represented by sequences of events.

Then, Subsystems (3) and (4) define a relation between their input and output sequences of events. Consequently, equivalent DEVS models can be found for these systems, called *static functions* and *quantized integrators*, respectively[9].

The piecewise constant input trajectories $v_j(t)$ can be also represented by sequences of events, and *source* DEVS models that generate them can be easily obtained.

Then, the QSS approximation Eq.(2) can be exactly simulated by a DEVS model consisting in the coupling of n quantized integrators, n static functions, and m signal sources. The resulting coupled DEVS model looks identical to the block diagram representation of the original system of Eq.(1), as Figure 2 shows.

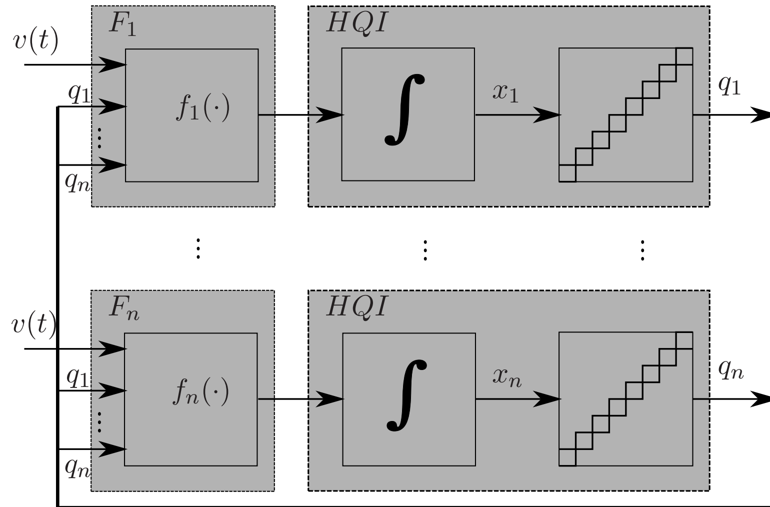

Figure 2: Coupled DEVS model of a QSS approximation

Based on the idea of QSS, a second–order accurate method was developed replacing the piecewise constant approximations by piecewise linear ones. The method, called QSS2, can be implemented using DEVS in the same way as QSS. However, the trajectories are now piecewise linear instead of piecewise constant. Thus, the events carry two numbers that indicate the initial

value and the slope of each segment. Also, the static functions and quantized integrators are modified with respect to those of QSS so that they can take into account the slopes.

Following the idea of QSS2, the third–order accurate QSS3 method[20] uses piecewise parabolic trajectories. The family of QSS methods has been augmented with methods for stiff and marginally stable systems (Backward QSS and Centered QSS of first order[26], and Linearly Implicit QSS methods of orders 1 and 2[25]).

2.3 RealTime Application Interface (RTAI)

RealTime Application Interface (RTAI) is a RTOS that supports several architectures (i386, PowerPC, ARM). RTAI is an extension of the Linux kernel to support real–time tasks to run concurrently with Linux processes. In order to enable real–time simulation, RTAI employs a method that was first used in RT-Linux[2].

The Linux kernel is not by itself a RTOS. It does not provide real–time services, and in some parts of the kernel, interrupts are disabled as a method of synchronization (to update internal structures in an atomic way). The periods of time when interrupts are disabled lead to a scenario where the response time of the system is unknown and time deadlines may therefore be missed.

To avoid this problem, RTAI inserts an abstraction layer underneath the Linux kernel. In this way, Linux never disables the *real* hardware interrupts. The Linux kernel is run on top of another micro–kernel (RTAI + Adeos [1]) as user processes. All hardware interrupts are captured by this micro-kernel and forwarded to Linux (if Linux has interrupts enabled).

Another problem running real–time tasks under Linux is that the Linux scheduler can take control of the processor from any running process without restrictions. This is unacceptable in a real–time system. Thus in RTAI, all real–time tasks are run in the micro–kernel without any supervision by Linux. Moreover, these processes are not seen by the Linux scheduler.

Real–time tasks are managed by the RTAI scheduler. There are two different kinds of processes, Linux processes and RTAI processes. RTAI processes cannot make use of Linux services (such as the File system) and vice versa. To avoid this problem, RTAI offers various IPC mechanisms (Pipes, Mailboxes, Shared Memory, etc.).

RTAI provides the user with basic services to implement real–time systems:

Deterministic interrupt response time: An interrupt is an external event to the system. The response time to this event varies from one computer to another, but RTAI guarantees an upper limit (on each computer), which is necessary to communicate with external hardware, for example a data acquisition board.

Inter Process Communication (IPC): RTAI supports a number of different methods of IPC, such as semaphores, pipes, mailboxes, and shared memory. These IPC mechanisms can be used for the communication between processes running in real time and normal Linux processes.

High accuracy timers: When developing real–time systems, the time handling accuracy is very important. RTAI offers clocks and timers with ns ($1.0e-9$ s) precision. On i386 architectures, these timers use the processor time stamp, therefore, they are very precise.

[1] Adeos[39,40] is the abstraction layer used in RTAI. Adeos implements the concept of *interrupt domain*. http://home.gna.org/adeos/

Interrupt Handling: RTAI allows the user to capture hardware interrupts and treat them with custom user handlers. Normal Operating Systems (such as Linux, Windows) hide hardware interrupts from the user, making the development of communication with external hardware more cumbersome (one needs to write a kernel driver).

Multiprocessor support RTAI supports also multiprocessor architectures enabling the user to run simultaneously different real–time tasks on separate processors.

2.4 *PowerDEVS* and Real–Time

PowerDEVS[4] is a general purpose tool for DEVS simulation. It consists of two main modules: The graphic interface that allows the user to describe DEVS models in a block diagram fashion, and the simulation engine that simulates the models.

In order to simulate a model, *PowerDEVS* first translates it into a C++ description and compiles it together with the simulation engine.

In spite of being a general purpose DEVS simulator, *PowerDEVS* was designed to simulate continuous and hybrid systems through the use of QSS methods. The standard distribution comes with a comprehensive library of atomic models for hybrid system simulation based on these methods. Users can add new atomic models by defining the dynamics of the DEVS model in C++.

PowerDEVS can be run under RTAI synchronizing the simulation with the physical (wall–clock) time obtaining an accuracy of the order of 1 μsec[4].

2.5 Parallelization platforms

Parallel computing is a form of computation, in which many calculations are simultaneously carried out on separate processors. Large problems can usually be divided into smaller ones that can then be solved concurrently.

There are different levels of parallelism: instruction level, data level, task level, etc. They differ in many aspects, such as shared or distributed memory, single or multiple processors, the type of communication mechanism they employ, such as message passing, or shared memory, etc.

We shall focus on task level parallelism, where each logical processor (LP) runs a different piece of code with different data, i.e., according to Flynn's taxonomy[14] we operate on Multiple Instructions, Multiple Data stream (MIMD).

In particular, we shall work with a multi–core architecture[5] as this offers many desirable features for our implementation, such as shared and unified memory, local communication (on chip), and a generic instruction set.

There exist also other architectures, such as the *Message Passing Interface (MPI)*[36] or *General–Purpose computing on Graphics Processing Units (GPGPU)*[33], but they both have their drawbacks.

MPI is a message passing protocol. This protocol is particularly well suited for clusters.

GPGPU does not fit our purposes well, since it lacks shared memory, the code executed on each processors must be the same, synchronization is very expensive, and the instruction set is not as generic as for a PC.

3 Related Work

In this section, we review previous work and similar developments.

3.1 Parallel Discrete Event Simulation

Since the beginning of the 1980's, much research has been performed on algorithms for synchronizing parallel simulation of discrete event systems.

One of the first solutions to the problem was proposed by Chandy, Misra, and Bryant[7,10,27]. They suggested a *conservative* distributed solution, whereby processes communicate only by message passing without shared memory and with a centralized coordinator.

The synchronization is achieved by enforcing the correct order of all messages, making all LPs wait until it is safe to produce the next event. By safe we mean that the LP producing the next event is certain that it will never receive a message with an earlier time–stamp than its own logical simulation time.

The authors prove formally the correctness of the algorithm and show that no deadlock can occur by the use of null messages. The memory consumption of the algorithm is constant as it does not require state saving at all. Chandy and Misra[11] also proposed a shceme that avoids the use of null messages and allows deadlock to happen, and then breaks it.

Its major drawback is that the amount of actual parallel computing done is diminished by the mechanism to enforce the causality constraint. By including *Lookahead* and *Structure Analysis*, the efficiency of the algorithm can be improved, but the achievable speedup is highly dependent on the model.

Another solution was proposed by Jefferson[18] where he introduces an *optimistic* synchronization method for parallel simulation, called *Time Warp*. Each LP carries its own logical time and advances as if there were no need for synchronization at all.

If an event with a smaller time–stamp than the simulation time is received (a *straggler* event), the LP rolls back to a previously saved state where it is safe to receive that event. It also un–sends (by using *anti–messages*) all events that were sent out in the rollback period. This involves saving state values in each LP for possible rollbacks, and thus, the memory requirements are larger than for other algorithms.

Both the state saving and the *anti–message* scheme (that could produce a cascade of rollbacks) are computationally expensive, and therefore, many extensions and additions have been made to the original algorithm to avoid rollbacks and reduce the number of state savings required. A complete review of these extensions can be found in the article by Liu[21].

A more recent algorithm called NOTIME[34] where the simulation is performed without any synchronization. Using this approach, the different LPs simulate their physical processes as fast as they can. *Straggler events* are allowed and accepted as correct. This introduces some error in the simulation since the logical time in the different LPs will vary from one processor to another.

While this exploits the parallelism optimally well, the technique can produce incorrect results, yet it may be an acceptable technique in situations where a user is willing to accept less accurate results for a lower simulation time.

The correctness of the result depends indirectly on how the work load is balanced across different LPs. If it is unevenly distributed, some LPs will run faster, while the ones with a heavier workload will run more slowly.

Tang et al.[38] studied an optimistic PDES using reverse computations applied to partial

differential equations problems obtaining linear speedups up to 4 processors. Similar works have been done applying PDES to simulate physical system such as the article by Bauer[3] (based on Nutaro's[29]) concluding that for certain kind of problem optimistic PDES is not suitable if not accompanied by a dynamic load balancing technique.

Outside DES, work has been done in distributed realtime simulation. Adelantado[1] presents a realtime capable HLA middleware and a formal model for validation. As we will see this is related to our work as it uses a local wall clock to synchronize different federates. On the other hand, in the formal model it is assumed that each federate has a periodic execution and a worst cases execution time, this assumptions do not hold in our work since the execution time is not know and varies during the simulation.

3.2 Parallel and Realtime DEVS Simulation

Many articles address the problem of simulating DEVS system in parallel using the before mentioned approches. Shafagh and Wainer[17] present CCD++ a conservative DEVS and Cell-DEVS simulator and they compare this approach with a optimistic one. Kim et al.[19] present a Time-Warp based methodology for distributed DEVS simulation. A mixed approach is investigated by Praehofer[32] that combines conservative and optimistic strategies that is able to optimally exploit the lookahead. Liu[22] explores a light version of the TimeWarp mechanism for a DEVS (and Cell-DEVS) simulator on the Cell processor. Here he introduces *LTW* and *MADS* as means to overcome TimeWarp bottlenecks and to hide the complexity of multicore programming from the general user.

Finally Hong et al.[16] proposed an extension to the DEVS formalism, to model Real Time systems called RealTime-DEVS (RTDEVS). The main idea is to fill time advances with executable *activities* and to specify time bounds required for each activity. They also implemented an DEVS executive (or DEVS simulator) that synchronizes the wall time with the simulation time. As we will see (Sec 5.3), this is related to the present work.

3.3 Parallel Simulation of QSS Approximations of Continuous Systems

Some previous results have also been reported concerning the parallelization of simulations of DEVS models resulting from the application of QSS algorithms to continuous systems.

Nutaro[28] investigates the first implementations of the QSS algorithms in a parallel DEVS simulator. The author studied the use of the Time Warp mechanism and concluded that it is unsuited for simulating large continuous systems.

By using a (first–order accurate) QSS, he simulated a sod shock tube problem achieving a speedup that increased to a maximum factor of 3 when using up to 6 processors, but quickly decreased when using more processors.

Another implementation of QSS on a multi–processor architecture has been investigated[23]. This implementation was performed using CUDA (a GPGPU language). Unfortunately as was already mentioned earlier in this article, graphical processing units (GPU) offer a limited instruction set, and synchronization tasks are very expensive. Moreover, GPUs follow a *Single Instruction, Multiple Data stream* (SIMD) under Flynn's taxonomy[14], where all processors execute the same code in parallel, and the performance is significantly reduced when the code has diverging branches.

These issues limit the classes of applications were GPUs can be efficiently used. In the

example analyzed in the cited reference, a speedup of up to a factor of 7.2 has been reported for a system with 64 states using 64 processors.

4 Scaled Real-Time Synchronization

In this section, we present the Scaled Real-Time Synchronization (SRTS) technique for PDES synchronization.

4.1 Basic idea

As mentioned already, several different PDES techniques have been proposed in the literature. They are based on either conservative, optimistic, or even unsynchronized algorithms.

DEVS models resulting of the application of QSS methods to large continuous systems have particular features:

1. Each event originates at a quantized integrator and is instantaneously transmitted to other quantized integrators through the static functions depending on the corresponding state variable.
2. The events received by quantized integrators change their future evolution, but they do not provoke instantaneous output events.
3. Different quantized integrators provoke output events at different times.
4. Each event represents a section of a piecewise polynomial function.

On the one hand, the second feature implies that synchronization is usually necessary, and techniques like NOTIME can rarely be used.

On the other hand, if the synchronization is strict so that it respects the ordering of all events, there will be very few calculations performed in parallel. The reason is simple. Each time one integrator provokes an event, the event is transmitted to a few static functions, and through them to a few quantized integrators.

A *conservative* algorithm will not allow any other LP to advance the simulation time until those calculations are complete. Consequently, almost nothing is gained by parallelization.

An *optimistic* algorithm will allow the simulation time of other LPs to advance, but as soon as the effect of an event is propagated to them, a rollback mechanism will need to take place. Rollback in a large continuous system is impractical, as this will consume a large amount of memory and time, and furthermore, it is difficult to assess in advance, how much time is going to be used up by the rollback.

To overcome these difficulties, we propose a technique, in which we perform a non–strict synchronization. Recalling that events represent sections of polynomials, errors in the synchronization imply a bounded numerical error in the trajectories transmitted from one process to another.

As with other PDES techniques, the overall model is divided in SRTS into sub–models representing physical processes, and each of these processes is simulated on a different LP.

In order to avoid the inter–process synchronization overhead, instead of synchronizing the simulation time between all LPs, we synchronize in SRTS the simulation time of each LP with the physical (wall–clock) time.

The only communication between different LPs takes place when events are transmitted from one LP to another. These events are transmitted using RTAI's mailbox IPC mechanism[13].

Each LP synchronizes with the wall–clock in the following way. If the next event in the LP is scheduled to occur after τ units of simulation time, the LP *waits* until the wall–clock advances by τ/r units of physical time. We call r the *real–time scaling factor*. It is a parameter that must be chosen according to the speed, at which the system can be simulated.

Once the waiting period is completed, the LP computes the output event and recomputes the state. When the output event needs to be propagated to sub–models belonging to different LPs, a *time–stamped* message containing the event is sent to the corresponding mailboxes.

During the waiting period, each LP checks for messages arriving at its mailboxes. Whenever a message arrives, the LP processes the event making use of the time–stamp, recomputes the state, and, if the recomputed next event is scheduled after the current time, it waits again.

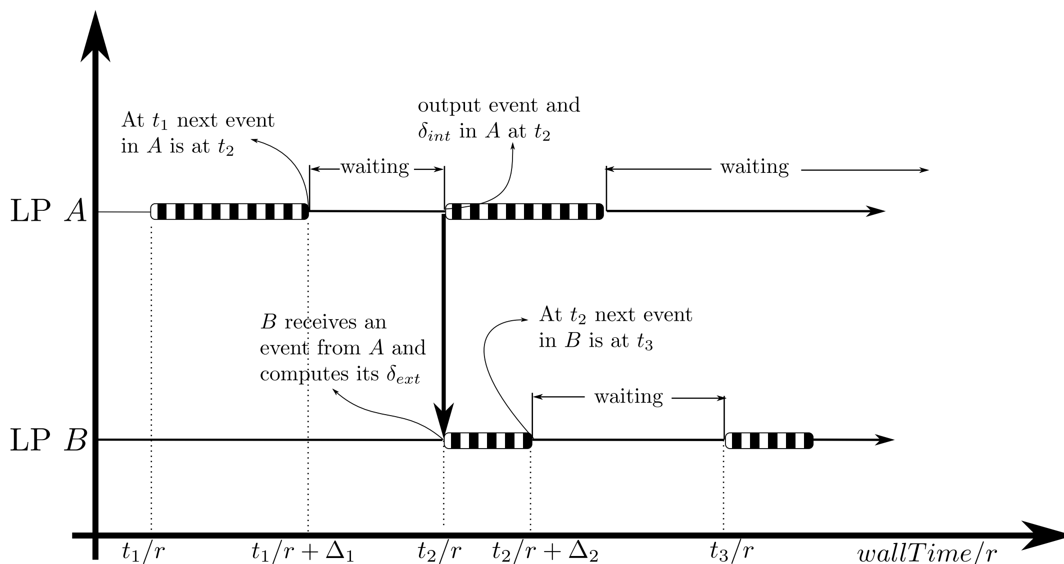

Figure 3: Time diagram of the SRTS technique

In Fig. 3 a time diagram is shown where two LPs synchronize using SRTS. Initially both LPs are waiting. At logical simulation time t_1 (scaled wall time t_1/r), LP A computes its next event. This computation takes Δ_1 units of wall time. Then LP A *waits* until t_2 is reached (at t_2/r of scaled wall time) when it emits its output event, makes an internal transition, and goes back to wait. This event is propagated to LP B who computes its external transition processing the received event (taking Δ_2 units of wall time), schedules its next event for t_3, and goes back to wait. This cycle is repeated until the final (logical) time is reached.

Implementing this scheme in an *abstract simulator*, there can be no *straggler* events; the only problem that could arise concerns simultaneous events in different LPs, but they have no consequences when simulating a QSS approximation.

In a real implementation, however, *straggler* events can appear, since neither the synchronization nor the communication methods are perfect, and also the computation of each event consumes some time.

Yet, as we mentioned before, bounded synchronization errors in the context of QSS simulation

provoke in most cases only bounded numerical errors in the trajectories transmitted between LPs.

4.2 SRTS Technique

As with other PDES techniques, SRTS requires that the model first be split into as many physical processes as there are different LPs to be used. In this work, we shall not discuss any methodology for efficiently partitioning a model but there are many works addressing this[12,37,15,6].

We consider that each LP holds a coupled DEVS sub–model, and those sub–models are coupled through input and output ports modeling the interactions between the subsystems.

Figure 4 shows the coupling structure of a model M divided into 4 sub–models simulated on 4 different LPs.

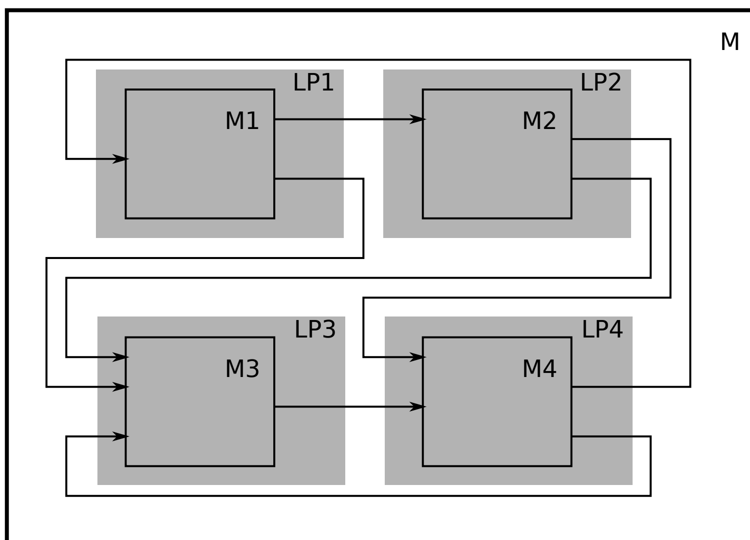

Figure 4: Coupling structure of SRTS

Simulation startup

A sequential DEVS simulation invokes first an initialization routine that sets the initial state of the model, the parameters, etc. Then, the simulation starts.

In SRTS, after each sub–model has been initialized, all simulation threads must start at the same wall–clock time in order to ensure synchronization between different LPs.

To accomplish a simultaneous startup, a barrier is used. All simulation threads are launched and the i–th thread starts like this:

1. Set the shared memory variable $flag[i] = true$.
2. Wait until $flag[j] = true$ for all $j \neq i$.
3. Measure the initial wall–clock time ($t_0[i]$) of the LP.
4. Start the simulation routine.

The initial wall–clock time is then used to compute the physical time from the beginning of the simulation, by subtracting it from the measured wall–clock time.

Simulation Algorithm

The SRTS simulation algorithm can be described as follows:

1. Compute the time of next event t_n of the sub–model .
2. If $t_n > t_f$ (final simulation time), then set $t_n := t_f$.
3. Wait until the scaled physical time reaches the simulation time (i.e., $t_p/r = t_n$) or until a message arrives.
4. If a message arrives, go to step 10.
5. If $t_n = t_f$ end of simulation.
6. Advance the simulation time to $t := t_n$.
7. Compute and propagate the output event inside the sub–model (external transition).
8. If the event propagates to other sub–models, send messages to the corresponding LPs.
9. Recompute the state of the model causing the event (internal transition) and return to step 1.
10. Advance the simulation time t to the value contained in the message time–stamp.
11. Propagate the event inside the sub–model (external transition) and return to step 1.

This algorithm works well provided that every LP sends and receives the messages at the correct time. To send a message at the correct time, a LP must finish the corresponding calculations before that time, i.e., it should not be in overrun. We shall discuss this issue next.

Overrun Policy

Overrun situations can be minimized by setting a sufficiently small value for the real–time factor r. However, we want r to be as big as possible in order to simulate fast.

Also, even choosing a very small value for r, real operating systems have latency, and overrun situations occur often.

When a LP is in overrun, it continues simulating without checking for input events. The reason is that input events arriving are in the future with respect to the logical time of the LP in overrun. Thus, it is better to process arriving messages only after the overrun situation has finished.

Message Handling

As mentioned earlier, events sent to sub–models belonging to different LPs are transmitted through time–stamped messages.

It can happen that a sub–model is in overrun or busy computing state transitions while a message is sent to it. If it does not finish its job before a new message arrives, we face two options. We can queue the messages until the LP is able to process them, or we can discard some of the messages.

While in the context of general DEVS models queueing the messages seems the most reasonable choice, we recall that SRTS is intended to simulate continuous systems where events represent changes in trajectories.

If at time t the LP finishes its job and realizes that there are two input messages waiting to be processed, one corresponding to time t_1 and the other to time t_2 with $t_1 < t_2 < t$, there is nothing to be done with the oldest message. It is already late, and the change in the trajectory it represented has been ignored. However, it makes sense to process the change in t_2 that contains more recent information about the trajectory.

For this reason, each sub–model in SRTS uses one mailbox for each input port (i.e., one mailbox per input trajectory), and each mailbox has capacity 1. No matter if the LP was able to read it or not, each message at the mailbox is overridden by the arrival of a new message.

4.3 SRTS and Numerical Errors in QSS Methods

The use of SRTS implies that messages sent between LPs may be received at incorrect times. Since LPs do not check for input messages when they are in overrun, they will not receive messages with time–stamps greater than the logical time. In other words, they never receive messages from the *future*. Synchronization errors always imply that messages arrive late.

Let us suppose that we use SRTS with two processors to simulate a QSS approximation of a high–order system:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{v}(t))$$

To this end, we split the system as follows:

$$\begin{aligned}\dot{\mathbf{x}}_a(t) &= \mathbf{f}_a(\mathbf{q_a}(t), \mathbf{q_b}(t), \mathbf{v}(t))\\ \dot{\mathbf{x}}_b(t) &= \mathbf{f}_b(\mathbf{q_a}(t), \mathbf{q_b}(t), \mathbf{v}(t))\end{aligned}$$

where:

$$\mathbf{x} = [\mathbf{x}_a\ \mathbf{x}_b]^T;\ \mathbf{q} = [\mathbf{q}_a\ \mathbf{q}_b]^T;\ \mathbf{f} = [\mathbf{f}_a\ \mathbf{f}_b]^T$$

The delay introduced by the SRTS mechanism implies that $\mathbf{q}_a$ is received with a delay by the LP that computes $\mathbf{x}_b$, and vice versa. Thus, the QSS approximation takes the form:

$$\begin{aligned}\dot{\mathbf{x}}_a(t) &= \mathbf{f}_a(\mathbf{q_a}(t), \mathbf{q_b}(t-\tau_b(t)), \mathbf{v}(t))\\ \dot{\mathbf{x}}_b(t) &= \mathbf{f}_b(\mathbf{q_a}(t-\tau_a(t)), \mathbf{q_b}(t), \mathbf{v}(t))\end{aligned}$$

where $\tau_a(t)$ and $\tau_b(t)$ are the communication delays between the LPs (including latency and overrun effects, and delays affected by the real–time scaling factor).

We shall discuss later some implementation issues that ensure that the delays are bounded. Depending on some features of the original ODE (input–to–state stability properties[8]), the

presence of bounded delays will only provoke a numerical error that will be superposed to the one introduced by the QSS approximation itself.

Although this analysis can be easily extended to systems with N LPs, we do not intend to perform a formal study of stability and numerical error bounds related to the usage of SRTS in this article.

It is, however, worth mentioning that the numerical error introduced by SRTS is minimized when latency and overrun are minimized.

4.4 Implementation of SRTS on *PowerDEVS* – RTAI

We implemented the SRTS algorithm in *PowerDEVS* on a multi–core architecture. We chose *PowerDEVS* because it implements the whole family of QSS methods and can run under a RTOS (RTAI).

The usage of a Real–Time OS is crucial for SRTS since, as explained before, the delay introduced by the latency must be bounded and minimized.

The PDES version of *PowerDEVS* generates a simulation executable that performs the following steps:

- It initializes the *PowerDEVS* model (see Sec 2.4).
- It starts the RTAI real-time system, initializing the real–time clocks and a main real–time task.
- For each LP, it creates and sets up as many mailboxes as there are input ports.
- It starts one real–time thread per physical processes. It then forks these threads with the simulation loop. An *index* parameter allows each thread to know, which physical processes it corresponds to.
- It waits for all real–time threads to finish and then exits while un–allocating all mailboxes and threads.

We have developed a live-cd from where one can try (or install) *PowerDEVS* –RTAI under any i386 processor.

Synchronization and CPU allocation

We implemented all synchronization tasks by a *busy–waiting* strategy, wasting CPU cycles without releasing the processor until the waiting time ends or a new message arrives. Although this strategy wastes computing power, it offers a more accurate synchronization.

A *busy–waiting* strategy under RTAI does not allow any other task to use the processor that is running the thread. In this way, the latency is minimized. However with this strategy, each processor can only run one thread, which limits the number of processes that can run in parallel.

RTAI also gives the user the ability to select the processor to run a given thread, i.e., we can allocate one thread per processor.

In consequence, we can divide the model into as many sub–models representing physical processes as there are processors available. On our hardware, we can allocate up to 12 LPs (as hyper–threading technology divides each core processor into two virtual processors).

Latency and overrun

In absence of latency and neglecting the processing time for messages, transitions, and other calculations, the algorithm described earlier in this section correctly simulates the overall model.

However in a real implementation, we cannot avoid some latency, and sometimes calculations take longer than the physical time allowed, and the simulation then enters an overrun situation.

For example, if the physical processor needs 1 wall–clock second to simulate 1 second of logical time, we cannot use a real–time scaling factor greater than 1. If we do that, the thread will never be able to catch up with the scaled wall–clock time.

While this type of overrun situation can be avoided by using a sufficiently small real–time scaling factor, one event generated by a quantized integrator in a QSS method usually triggers many simultaneous events through the static functions. Whereas the first event will be correctly synchronized, the remaining events will be invariably in overrun.

Latency also introduces some error in the synchronization, which affects the communication mechanism.

If an event scheduled for t_1 is sent out at time $t_1 + \Delta t$, the other threads will receive it with a time delay of Δt. If during that delay period the other threads computed an event, an error appears that affects the global result.

The hardware platform we use under RTAI exhibits an average latency that ranges from 150 to 500 *nsec*, with a maximum value around 10 μsec. These values limit the maximum real–time factor that we may use.

For instance, if we use a real–time factor of 1000, a latency of 10 μsec is translated to 10 *msec* of maximum simulation time error in events transmitted between different LPs. If that simulation time error provokes an unacceptable simulation error, then we would be enforced to use a smaller real–time factor.

5 Adaptive Real-Time Scaling

The main drawback of SRTS is that the real–time scaling factor r must be chosen by the user. Unless the user performs some previous trial and error experiments, he would be hard–pressed to know a priori, which value to assign to the real–time scaling factor.

Another problem is that r remains constant during the entire simulation. In many cases, the simulation workload changes with time, and it makes sense to adapt r according to the actual workload.

In this section, we introduce an adaptive version of SRTS, called ASRTS, where the real–time scaling factor is automatically adapted to optimize the efficiency of the simulation.

5.1 Basic idea

Adaptive SRTS attempts to automatically change the real–time scaling factor to minimize the time that processors spend waiting but without provoking overrun situations. In this way, ASRTS improves the overall efficiency by lowering the simulation time.

The idea behind ASRTS is rather simple. It is identical to SRTS, except that it periodically changes the real–time scaling factor.

ASRTS cuts the simulation time into sampling periods of equal duration. During each such period, each LP measures the total time that it spends waiting. At the end of the period, one

of the threads collects the accumulated waiting times of all LPs and determines their minimum (i.e., the waiting time of the process that had the heaviest workload). Using that information, it computes the ratio between the smallest accumulated waiting time and the length of the sampling period.

If the minimum waiting ratio w is greater than a prescribed *desired* waiting ratio w_0 (we use values around 10%), the algorithm is allowed to increase the real–time scaling factor and simulate faster. Otherwise, the algorithm decreases r.

In this way, ASRTS tries to keep the minimum waiting ratio around the *desired* waiting ratio w_0.

Changes in the real–time scaling factor are done synchronously, that is, *all* LPs change their value of r at the same time (logically and physically) and to the same value, since the scaling factor affects directly the implicit synchronization among the entire group of LPs.

Once r has been changed, the LPs continue with the simulation using the SRTS algorithm until the next sampling period has ended.

5.2 Adaptive-SRTS Algorithm

As mentioned above, ASRTS is identical to SRTS but includes periodic checkpoints (with a sampling period of ΔT), when the scaling factor r may be changed.

During each sampling, ASRTS gathers statistics about the workload and the time spent waiting by each LP. When they reach the next checkpoint, the LPs *stop* simulating and wait until the new real–time scaling factor has been computed.

This computation is done by a thread *coordinator* (thread 0), while the remaining threads wait in a synchronizing barrier that holds them until the new scaling factor has been computed.

At the end of each sampling period, after each LP computed its accumulated waiting time T_{w_i}, the coordinator computes the new real–time scaling factor as follows:

1. It finds the minimum waiting ratio

$$w = \frac{\min_i\{T_{w_i}, \sigma\}}{\Delta T}$$

   where σ is a value lower but close to 1 to avoid division by zero. In a real implementation not all the T_{w_i} can be 1 because of overheads.

2. It computes the *optimal* real–time scaling factor $\hat{r}$ for the *desired* waiting ratio w_0 as:

$$\hat{r} = r\frac{1 - w_0}{1 - w} \tag{5}$$

3. If $\hat{r} < r$, it computes the new real–time scaling factor as $r := \hat{r}$, slowing down the speed of the simulation.

4. Otherwise if $\hat{r} \geq r$, it computes the new real–time scaling factor as

$$r := \lambda r + (1 - \lambda)\hat{r} \tag{6}$$

   which smoothly increases the simulation speed according to the discrete *eigenvalue*[2] λ.

[2]Note that the Eq. 6 is a difference equation with λ as eingenvalue.

The ASRTS algorithm has 3 tuning parameters: the sampling period ΔT, the desired waiting ratio w_0, and the discrete eigenvalue λ. We discuss the most suitable values of these parameters later in this section.

ASRTS makes use of Eq.(5) that computes the real–time scaling factor $\hat{r}$, for which a waiting ratio w_0 is obtained. This expression is derived below.

Optimal real–time scaling factor

As mentioned above, ASRTS attempts to drive the minimum waiting ratio w to the desired waiting ratio w_0 by adjusting the real–time scaling factor r.

Suppose that during a sampling period ΔT the LP with the maximum workload had a waiting ratio w while the real–time scaling factor was r. Thus during that period, it spent computing a total of

$$T_c = \Delta T - w \cdot \Delta T = (1 - w) \cdot \Delta T$$

units of time, and the simulation time advanced by $r \cdot \Delta T$ units of time.

Thus, in order to advance $r \cdot \Delta T$ units of simulation time, the LP needed T_c units of physical time. We can define the workload as follows

$$W \triangleq \frac{\text{physical time}}{\text{simulation time}} = \frac{T_c}{r \cdot \Delta T} = \frac{1 - w}{r}$$

If we assume that the workload changes slowly, we can expect that, in the next sampling period, it will remain almost unchanged. If we use the *appropriate* scaling real time factor $\hat{r}$ in the new sampling period, we shall obtain the desired waiting ratio w_0 and then,

$$W = \frac{1 - w}{r} = \frac{1 - w_0}{\hat{r}}$$

from which we obtain Eq.(5).

Parameter selection

In order to properly set the values of the method parameters of ASRTS, we may consider the following points.

- The sampling period ΔT should be large compared to the physical time required by the re–synchronization routine and also compared with the time necessary to compute several events. In this way, the time spent in the synchronization routine is negligible, and every sampling occurs after several events were computed so that workload statistics make sense. However, ΔT must be small enough so that the real–time scaling factor is changed several times during the simulation to be able to react to workload changes in a timely fashion.

  In the *PowerDEVS* implementation, we used $\Delta T = 100msec$. In our platform, the synchronization routine takes from 10 to 100 μsec, and we are interested in simulations that take at least several seconds (otherwise parallelization might not be necessary). Thus, a period of $\Delta T = 100msec$ is a reasonable choice for most cases.

- The desired waiting ratio w_0 should be greater than 0 and less than 1. We want simulations to run fast, minimizing the waiting time, so a small value of w_0 should be chosen. However,

if w_0 is very close to 0, several types of overrun situations can occur that may cause large numerical errors.

In the *PowerDEVS* implementation, we normally use $w_0 = 0.15$, which means that the real–time scaling factor is adjusted such that the LP with the heaviest workload wastes approximately 15% of the time in a waiting routine.

- The discrete eigenvalue λ (with $0 < \lambda < 1$) determines how fast the real–time scaling factor r increases its value to reach the optimal value $\hat{r}$ according to Eq.(6). When λ is close to 0 the adaptation is fast. When it is close to 1, the adaptation is slow and smooth. In presence of fast changes in the workload, a slow adaptation prevents the real–time scaling factor from assuming very large values that might cause frequent overruns.

  For this reason, we normally choose $\lambda = 0.9$. With this value, the real–time scaling factor reaches 80% of its final value after 10 periods (i.e., 1 second in our implementation).

5.3 Relation with other approaches

The present work can be seen as a combination of two previous ideas of the PDES and DEVS world. First, is related to the RTDEVS executive[16] since simulations run synchronized with the wall clock, in our case a scaled version of the wall clock. However the RTDEVS formalism is not used for model description and regular DEVS is used instead. In our case the time consuming activities are in fact the computation of the output events and the busy waits to synchronize them.

SRTS and ASRTS are also related to the NOTIME technique[34] since the causality constraint can be violated eventually. This violation is minimized by the introduction of busy waits. These waits act as an artificial load that help balance the workload among LPs, thus implicitly solving the unbalancing problem of NOTIME.

6 Applications and Examples

In this section, we present some simulation results. We simulated three separate large systems using QSS methods first in a sequential way and then using SRTS and ASRTS techniques. All the examples were run using an Intel i7 970 – 6 cores with hyper–threading[24]. RTAI's testsuite reports 416ns average and 18000ns max latencies on this setup.

In all the cases, each simulation was run 10 times and the results did not experience any noticeable change.

6.1 Power control of an air conditioner population

This example, taken from Perfumo et al.[30], is a model to study the dynamics and control of the power consumption of a population of air conditioners (AC).

We consider here a large population of air conditioners used to control the temperature of different rooms. The temperature of the i–th room $\theta_i(t)$ follows the equation

$$\frac{d\theta_i(t)}{dt} = -\frac{1}{C_i \cdot R_i}[\theta_i(t) - \theta_a + R_i \cdot P_i \cdot m_i(t) + w_i(t)], \qquad (7)$$

where R_i and C_i are parameters representing the thermal resistance and capacity of the i–th room, respectively. P_i is the power of the i–th air conditioner when it is in its *on* state. θ_a is the ambient temperature, and $w_i(t)$ is a noise term representing thermal disturbances.

Variable $m_i(t)$ is the state of the i–th air conditioner, which takes a value of 1 when the unit is in its *on* state, and 0 otherwise. It follows the hysteretic on–off control law:

$$m_i(t^+) = \begin{cases} 0 & \text{if } \theta_i(t) \leq \theta_r(t) - 0.5 \text{ and } m_i(t) = 1 \\ 1 & \text{if } \theta_i(t) \geq \theta_r(t) + 0.5 \text{ and } m_i(t) = 0 \\ m_i(t) & \text{otherwise} \end{cases} \tag{8}$$

where $\theta_r(t)$ is the reference temperature that is calculated by a global control system.

The power consumption of the entire AC population is computed as:

$$P(t) = \sum_{i=1}^{N} m_i(t) \cdot P_i$$

and a global control system regulates it, so it follows a desired power profile $P_r(t)$. In order to accomplish this, a Proportional Integral (PI) control law is used to compute the reference temperature:

$$\theta_r(t) = K_P \cdot [P_r(t) - P(t)] + K_I \cdot \int_{\tau=0}^{t} [P_r(\tau) - P(\tau)]\mathrm{d}\tau$$

where K_P and K_I are the parameters of the PI controller.

In this example we chose $N = 2400$ air conditioner units, and the set of parameters given in the article[30]. Thus, we are dealing with a fairly large hybrid system consisting of a set of 2401 differential equations, with 2400 zero–crossing conditions.

We first simulated the system using the QSS3 method with a quantization of $\Delta Q_{rel} = 10^{-3}$ and $\Delta Q_{min} = 10^{-6}$ in a sequential fashion. Figure 5 plots the average power consumption $P(t)/P_{\max}$.

The final simulation time was $t_f = 3000$ sec, and the sequential simulation consumed 58 sec of physical time.

We then divided the system into 12 subsystems with 200 air conditioners in each sub–model. The first sub-model also included the PI controller.

We then used the SRTS technique with different real–time scaling factors. We started with a scaling factor of $r = 200$, which consumes $t_f z/r = 3000/200 = 50$ sec of physical time. We obtained decent results using scaling factors of up to $r = 950$, which takes $t_f/r = 3000/950 = 3.16$ sec.

For each scaling factor, we compared the simulation results with those of the sequential simulation and measured the relative RMS error.

Here and in the following examples the RMS error is computed as the:

$$RMS = \frac{\text{mean}(x - \hat{x})^2}{\text{mean}(x)}$$

where x is the sequential result and $\hat{x}$ is the parallal result.

Figure 6 plots the measured error in function of the real–time scaling factor.

In this example, using $r = 950$, SRTS was more than 18 times faster than the sequential simulation without introducing unacceptably large numerical errors.

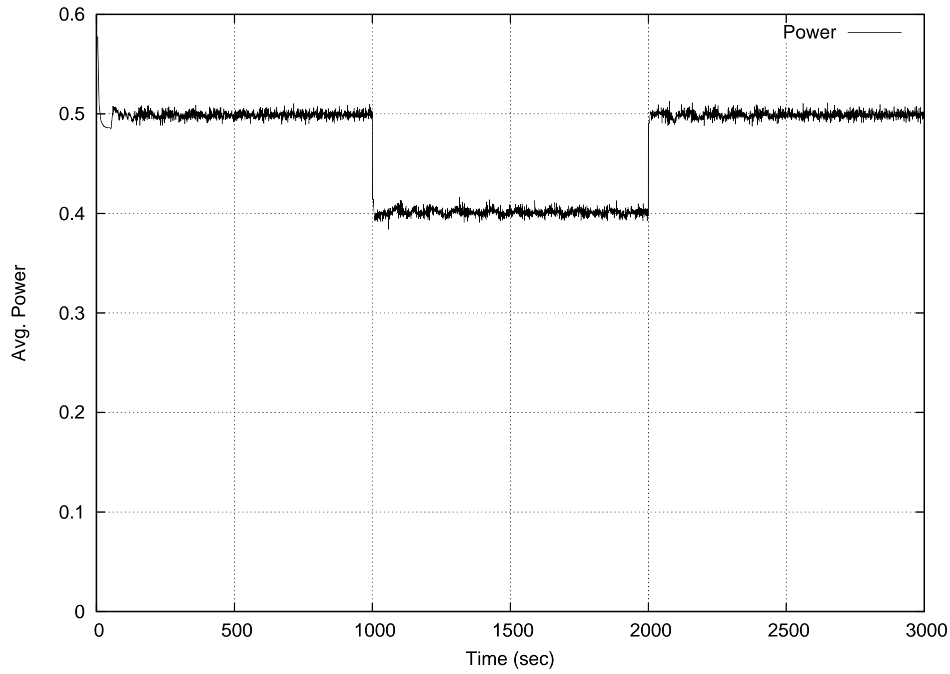

Figure 5: Power consumption in the AC system (sequential simulation result).

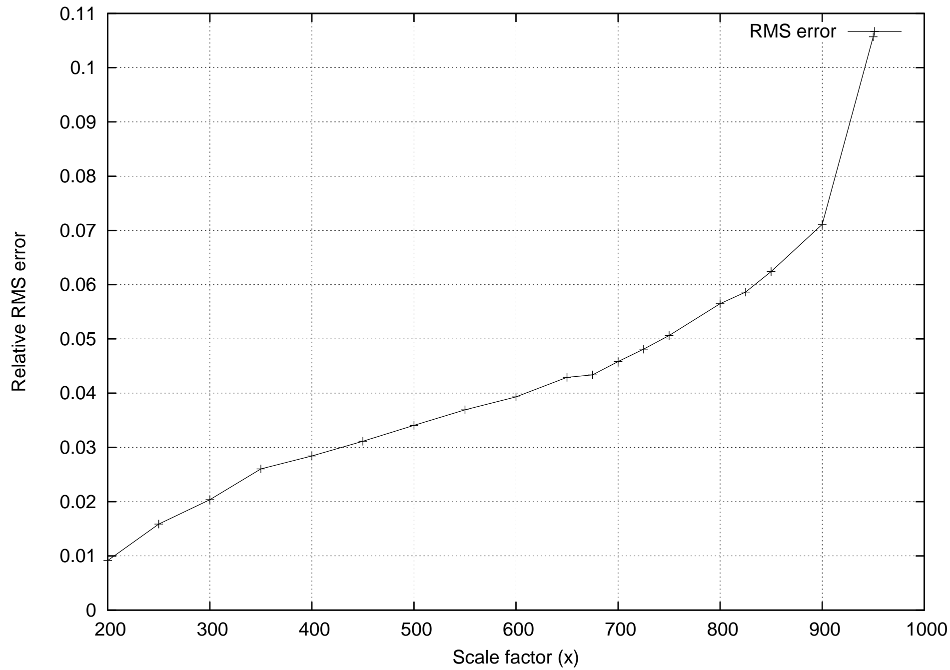

Figure 6: SRTS: Error *vs* real–time scaling factor – AC example

However for $r > 500$, we observed that the LPs were most of the time in overrun. Yet in this case, the overrun situation does not affect the numerical error in major ways.

Then we simulated the system using Adaptive SRTS with parameters $\Delta T = 100$ msec,

$w_0 = 0.15$, and $\lambda = 0.9$. The ASRTS simulation took 6.6 seconds, reaching a maximum real–time scaling factor of $r = 480$.

In consequence, ASRTS was almost 9 times faster than the sequential simulation using 12 processors. The real–time scaling factor reached was close to the limit, at which large overruns are observed.

We repeated the simulation with ASRTS for different values of λ. Figure 7 shows the way, in which ASRTS adapts the scaling real–time factor during each simulation run.

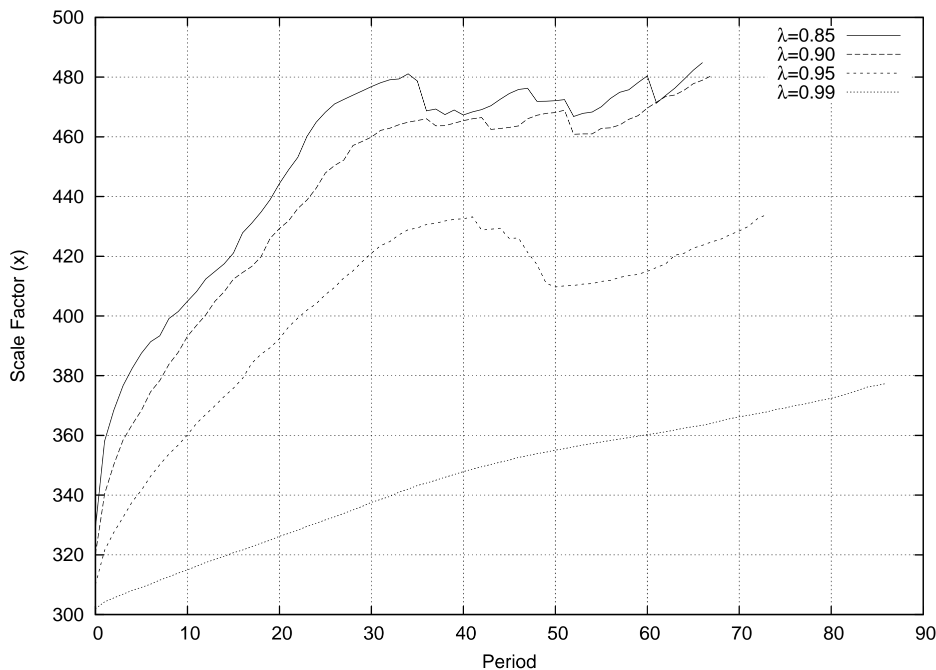

Figure 7: ASRTS: Real–time scaling factor adaptation for different values of λ – AC example

We notice that r converges very slowly to the optimal scaling factor for large values of λ (close to .99). Consequently the simulation is slow. On the other hand, r converges fast for small values of λ (around 0.8) but needs to reduce its value frequently. A reduction in the real–time scaling factor indicates that some LP waited less than the desired waiting ratio w_0, which is unsafe as this might indicate overrun.

A value of $\lambda \approx 0.9$ provides a good compromise between fast adaptation and overrun avoidance.

We then repeated the ASRTS simulation while varying the number of processors, splitting the model into 2, 4, 6, and 8 sub–models. Figure 8 shows the simulation speedup obtained for these cases.

We can observe a close to linear increase in the simulation speedup with the number of processors.

6.2 Logical inverter chain

A logical inverter performs a logical operation on a signal. When the signal assumes a high level, it outputs a low level, and vice-versa.

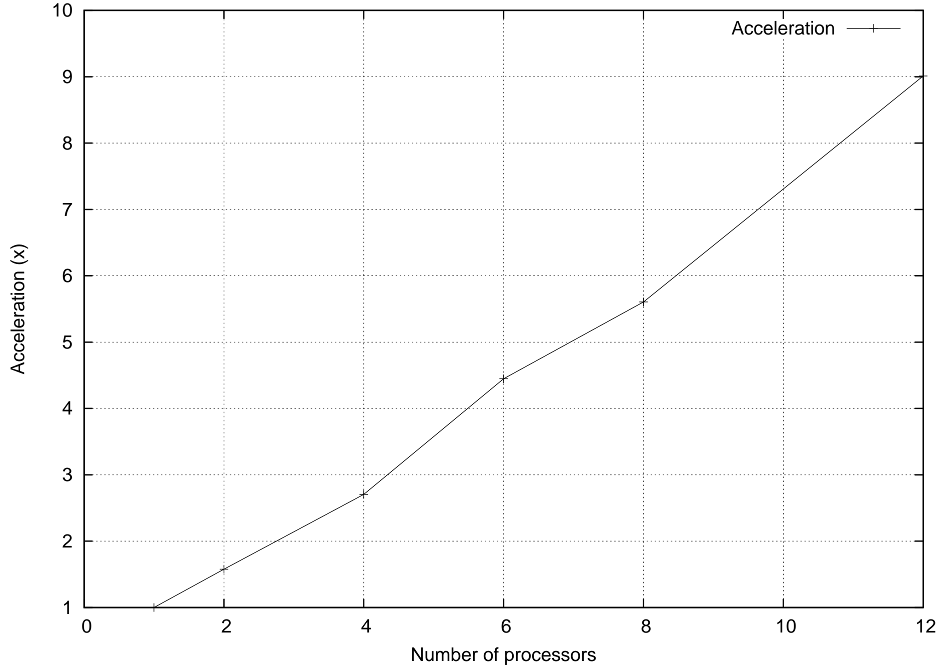

Figure 8: ASRTS: Speedup *vs* number of processors – AC example

Logical inverters are implemented by electrical circuits. They exhibit a non-ideal response, since the rise and fall time of the signal output is limited by physical characteristics, and the correct output level is not immediately obtained, i.e., it is delayed.

An inverter chain is a concatenation of several inverters, where the output of each inverter acts as the input to the next one. Making use of the aforementioned physical limitations, inverter chains can be used to obtain delayed signals.

We consider here a chain of m inverters according to the model given in Savcenco and Mattheij[35], which is characterized by the following equations:

$$\begin{cases} \dot{\omega}_1(t) & = U_{op} - \omega_1(t) - \Upsilon g\left(u_{in}(t), \omega_1(t)\right) \\ \dot{\omega}_j(t) & = U_{op} - \omega_j(t) - \Upsilon g\left(\omega_{j-1}(t), \omega_j(t)\right) \quad j = 2, 3, .., m \end{cases} \tag{9}$$

where:

$$g(u, v) = \left(\max(u - U_{thres}, 0)\right)^2 - \left(\max\left(u - v - U_{thres}, 0\right)\right)^2 \tag{10}$$

We used the set of parameters given in the article: $\Upsilon = 100$ (which results in a very stiff system), $U_{thres} = 1$, and $U_{op} = 5$.

The initial conditions are, as in the given reference, $\omega_j = 6.247 \cdot 10^{-3}$ for odd values of j and $\omega_j = 5$ for even values of j. The input is a periodic trapezoid signal, with parameters $V_{up} = 5V$, $V_{low} = 0V$, $T_{down} = 10$, $T_{up} = 5$, $T_{rise} = 5$, and $T_{fall} = 2$.

In this case, we considered a system of $m = 504$ inverters, so we have a set of 504 differential equations with 1008 discontinuity conditions due to the 'max' functions in Eq.(10).

As in the previous example, we first simulated the system in a sequential fashion using the stiff LIQSS3 solver. The resulting voltage at the last inverter is shown in Figure 9.

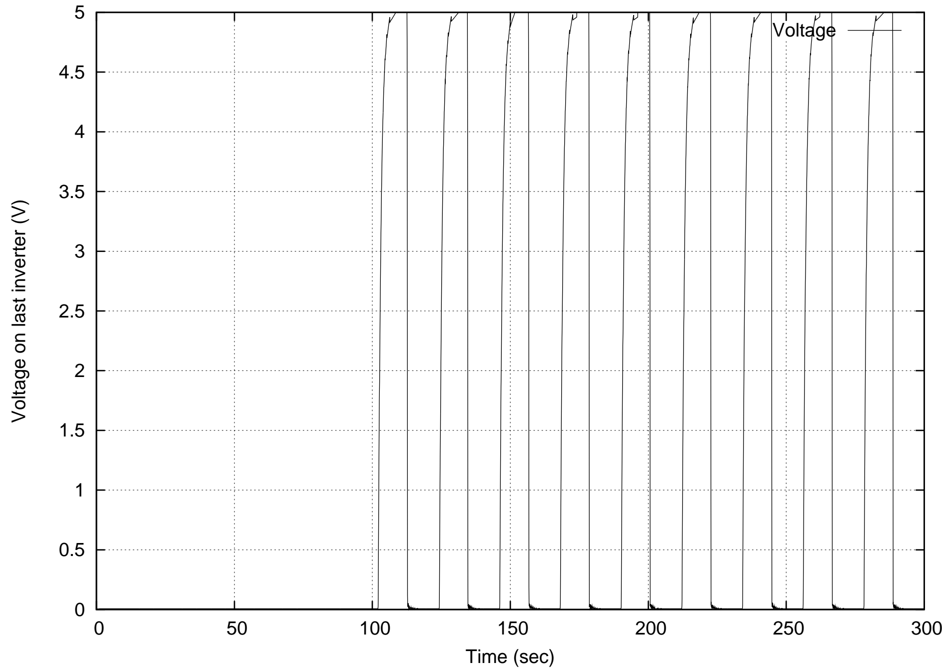

Figure 9: Voltage on last inverter (sequential simulation result)

We then divided the system into 12 sub-models with 42 inverters each and simulated with SRTS and different real–time scaling factors. This time we obtained decent results for up to $r \approx 35$.

Figure 10 shows the error in function of the real–time scaling factor used. The RMS error looks large because it is mainly due to a small time shift in the output voltage wave. As the wave shows abrupt changes, a small time shift can cause a large RMS error.

We then applied the ASRTS algorithm and, using 12 processors, the simulation was almost 4.5 times faster than the sequential one.

In this example, the workload is not evenly distributed on the system. As the wave travels, it provokes a high workload at the processor that computes the states that are switching. In contrast, processors that compute states that do not change show a very low workload.

Consequently in this case, neither SRTS nor ASRTS can exploit parallelization as efficiently as in the previous example. However, the results are still good.

We then repeated the simulations using ASRTS while varying the number of processors. Figure 11 shows the speedup obtained in each case.

Once again, the simulation speed increases almost linearly with the number of processors in use.

6.3 LC Transmission Line

The following system of equations represents a lumped model of a lossless transmission line, formed by 600 sections of LC circuits with parameters $L = C = 1$:

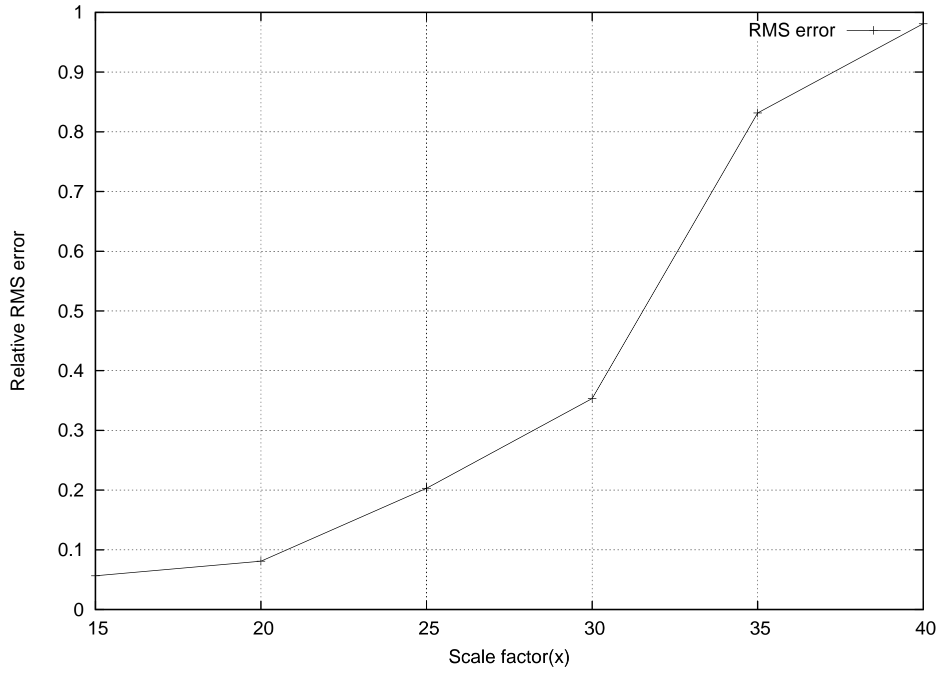

Figure 10: SRTS: Error *vs* real–time scaling factor – Inverter chain example

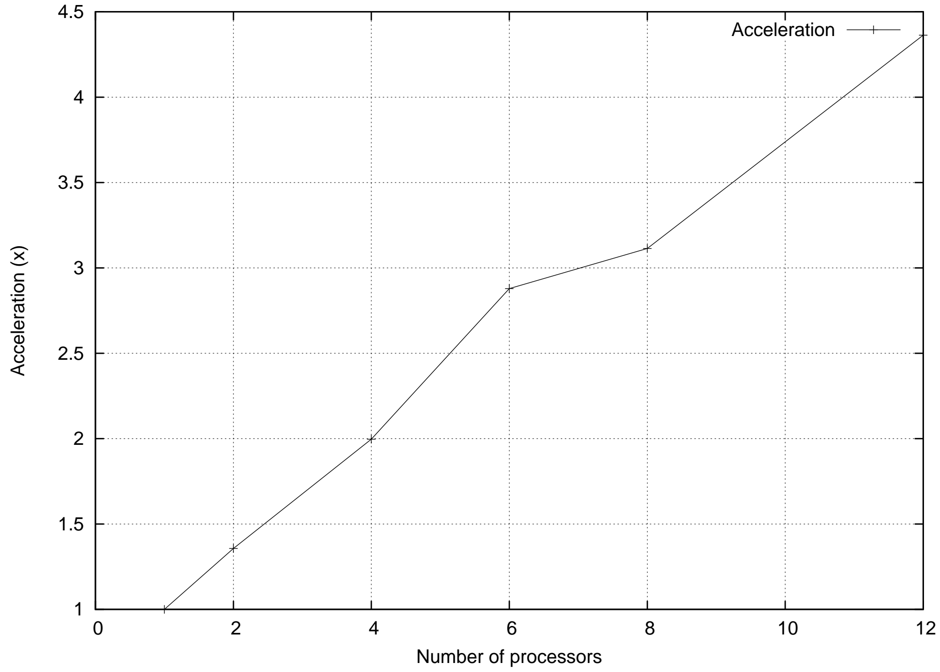

Figure 11: ASRTS: Speedup *vs* number of processors – Inverter chain example

$$
\begin{aligned}
\dot{\phi}_1(t) &= u_0(t) - u_1(t) \\
\dot{u}_1(t) &= \phi_1(t) - \phi_2(t) \\
&\vdots \\
\dot{\phi}_j(t) &= u_{j-1}(t) - u_j(t) \\
\dot{u}_j(t) &= \phi_j(t) - \phi_{j+1}(t) \\
&\vdots
\end{aligned}
\qquad (11)
$$

We consider a sinusoidal input signal:

$$u_0(t) = \sin(\omega t) \tag{12}$$

with $w_0 = 0.13$ Hz. We also set zero initial conditions $u_i = \phi_i = 0,\ i = 1, \ldots, n$.

This is a marginally stable system, of order 1200 as there are two state variables per segment.

As in the previous examples, we first simulated the system in a sequential fashion using the QSS3 solver. Figure 12 shows the resulting voltage at the end of the transmission line $u_{600}(t)$.

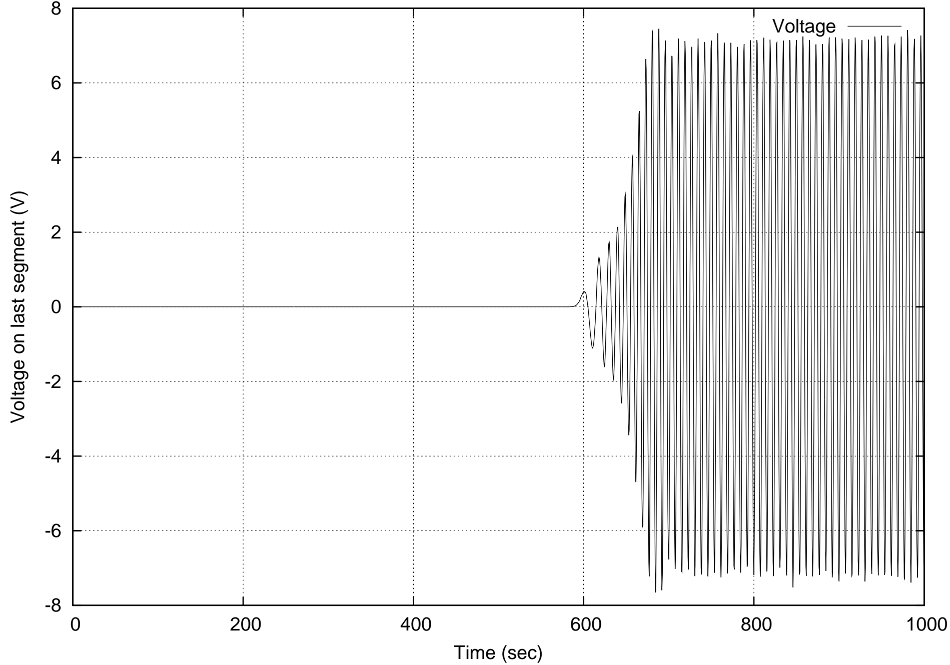

Figure 12: Voltage on last LC segment (sequential simulation result)

Then we divided the system into 12 sub–models with 50 segments each, and simulated it using SRTS with different values of the real–time scaling factor.

Figure 13 shows the RMS error as a function of the real–time scaling factor.

This time around, we obtained good results for $r < 400$. For larger values of the real–time scaling factor, the results soon become numerically unstable. This is not surprising as the addition of delays in marginally stable systems tends to cause instability.

We then simulated the system using ASRTS. With 12 processors, ASRTS managed to accelerate the simulation so it ran more than 5.5 times faster than the sequential simulation.

As in the previous example, the workload is not evenly distributed while the wave travels from the input to the output, which limits the advantages of parallelization.

We then used ASRTS with a varying number of processors, obtaining again an almost linear relationship between the number of processors in use and the speedup as shown in Figure 14.

7 Conclusions and Future Work

We introduced SRTS and ASRTS, two new techniques for Parallel Discrete Event Simulation of continuous and hybrid systems.

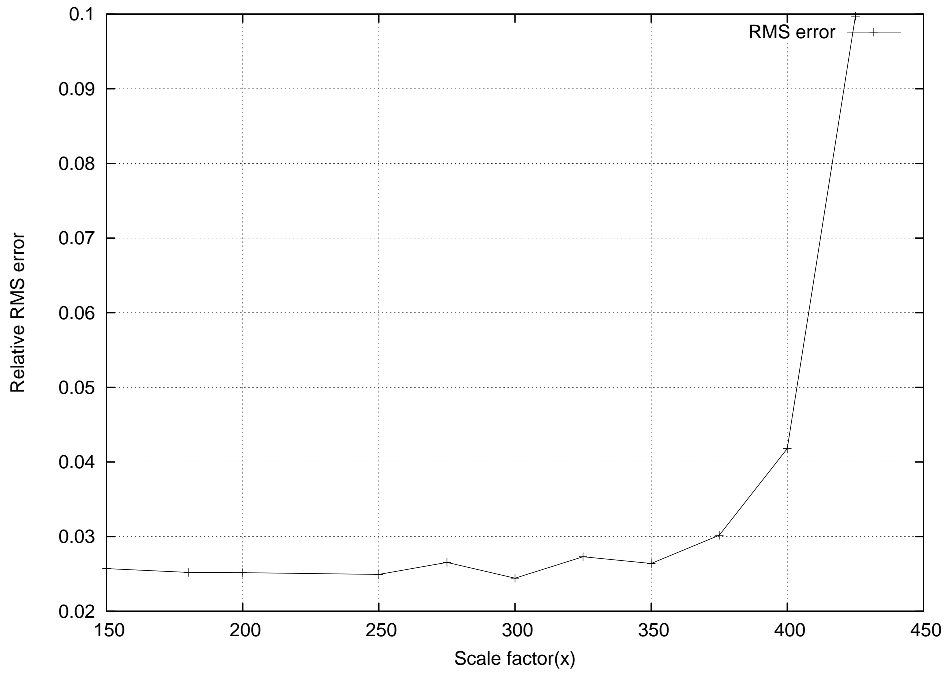

Figure 13: SRTS: RMS error *vs* real–time scaling factor – LC line example

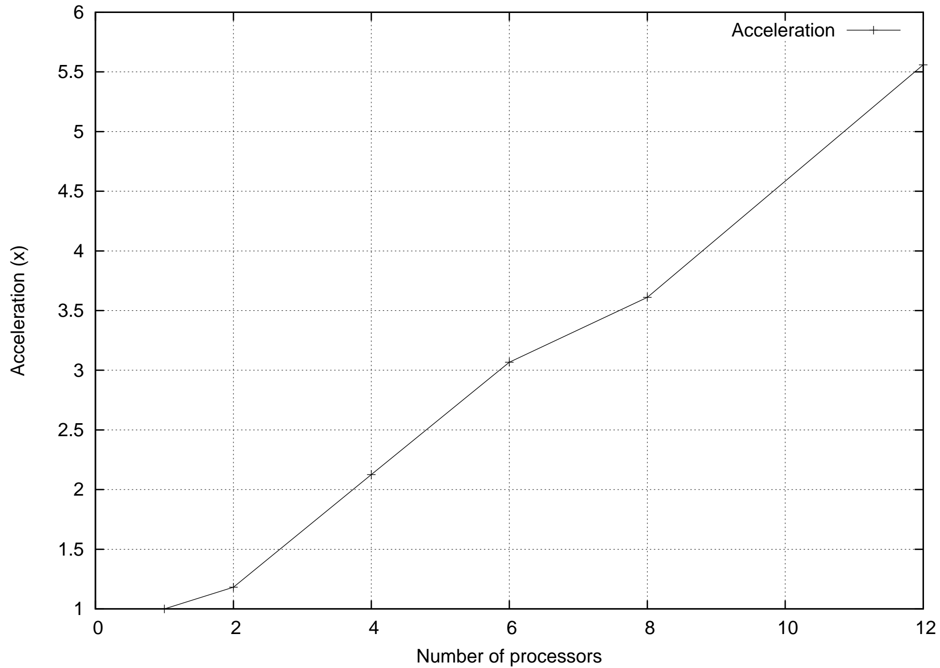

Figure 14: ASRTS: Speedup *vs* number of processors – LC line example

The techniques are based on the idea of synchronizing each LP simulation time with a *scaled* version of the physical time. As all LPs are synchronized with the physical time, they are indirectly synchronized also against each other.

This indirect synchronization permits running each process independently of the remaining processes, avoiding the overhead imposed by inter–process synchronization and coordination.

We implemented these new techniques in *PowerDEVS* using the real–time operating system RTAI. The RTOS was needed to ensure a precise synchronization, to properly perform the CPU allocation, and for avoiding that the OS (Linux) takes control of the processor while the thread is running.

We tested these new techniques with three application examples and studied their performance and limitations.

Using 12 processors, we found a major improvement on the simulation speed. In the examples analyzed, ASRTS simulated from 4.5 to 9 times faster than the sequential simulation. The non–adaptive $SRTS$ simulated even slightly faster but requires knowing the right real–time scaling factor to use.

We have also analyzed the additional numerical error introduced by the techniques. We found that, provided that the real–time scaling factor r remains small enough to avoid frequent overruns, the error remains bounded. Moreover, ASRTS adjusts r in order to avoid overruns so the error introduced is kept small.

This work opens up several lines of research to be considered in the future. A few of them are listed below:

- In the present work, we manually split the model trying to obtain a well–balanced division. It is interesting to investigate an automatic way to split the model and also to study techniques to dynamically re–balance the workload among the available processors.

  We are currently working on an extension to the DEVS formalism, called Vectorial DEVS, that will simplify this task.

- A fall–back mechanism for overrun situations is also a desirable feature. By implementing a rollback mechanism when a large overrun is found, we could bound the errors introduced.

  To this end, we could use the Adative-SRTS checkpoints as rollback stages. In this way, we would only need to save the state of the system at the last checkpoint consuming a bounded and constant amount of memory. Also, we can know exactly how much time we waste at each rollback: ΔT.

- We have developed the implementation on a multi–core system running a RTOS. A very important issue is to study how these algorithms scale with a greater number of processors or in a cluster environment. To this end, several problems have to be solved first, such as the clock synchronization among different nodes or CPUs, the communication between nodes (in the current multi–core approach it is implemented using shared memory), and the CPU allocation.

- We will work on a formal analysis of how the delays in the communication affect the numerical results, following the approach outlined in Section 4.3. The idea is to represent the effects of the delays as bounded perturbations and to analyze the perturbed systems to ensure numerical stability and error boundedness.

- We can also investigate new heuristics for the ASRTS. So far, we have only included the time spent waiting as a measurement of how the simulation is doing but including overrun statistics or delay data might result in an improved performance.

- Although we developed these new techniques for discrete event simulation of continuous systems, it may be interesting to study their usage also with other numerical integration methods for continuous and hybrid systems (particularly with multi–rate algorithms). Also, it is worth considering the study of SRTS and ASRTS in more general applications of discrete event simulation.

The complete code of the SRTS and ASRTS implementation in *PowerDEVS* can be found and downloaded from the *realtime* branch of the SVN[31].

REFERENCES

1. M. Adelantado, P. Siron, and J. Chaudron. Towards an HLA Run-time Infrastructure with Hard Real-time Capabilities. In *International Simulation Multi-Conference*, Ottawa, Canada, July 2010.
2. M. Barabanov. A Linux-based RealTime Operating System. Master's thesis, New Mexico Institute of Mining and Technology, New Mexico, 1997.
3. D. Bauer and E. Page. Optimistic parallel discrete event simulation of the event-based transmission line matrix method. In *Simulation Conference, 2007 Winter*, pages 676 –684, dec. 2007.
4. F. Bergero and E. Kofman. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation*, 87:113–132, January 2011.
5. G. Blake, R. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26 –37, november 2009.
6. A. Boukerche and C. Tropper. A static partitioning and mapping algorithm for conservative parallel simulations. *SIGSIM Simul. Dig.*, 24(1):164–172, July 1994.
7. R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Cambridge, MA, USA, 1977.
8. R. Castro, E. Kofman, and F. Cellier. Quantization Based Integration Methods for Delay Differential Equations. *Simulation Modelling Practice and Theory*, 19(1):314–336, 2011.
9. F. Cellier and E. Kofman. *Continuous System Simulation*. Springer, New York, 2006.
10. K. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *Software Engineering, IEEE Transactions on*, SE-5(5):440 – 452, sept. 1979.
11. K. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, Apr. 1981.
12. E. Deelman and B. Szymanski. Dynamic load balancing in parallel discrete event simulation for spatially explicit problems. In *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, pages 46 –53, may 1998.

13. DIAPM. Programming Guide 1.0. 2000. http://www.aero.polimi.it/ rtai/documentation/.

14. M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901 – 1909, dec. 1966.

15. D. Glazer and C. Tropper. On process migration and load balancing in Time Warp. *Parallel and Distributed Systems, IEEE Transactions on*, 4(3):318 –327, mar 1993.

16. J. S. Hong, H.-S. Song, T. G. Kim, and K. H. Park. A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development. *Discrete Event Dynamic Systems*, 7:355–375, 1997. 10.1023/A:1008262409521.

17. S. Jafer and G. Wainer. Conservative DEVS: a novel protocol for parallel conservative simulation of DEVS and cell-DEVS models. In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, pages 140:1–140:8, San Diego, CA, USA, 2010. Society for Computer Simulation International.

18. D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7:404–425, July 1985.

19. K. H. Kim, Y. R. Seong, T. G. Kim, and K. H. Park. Distributed simulation of hierarchical DEVS models: hierarchical scheduling locally and time warp globally. *Transactions of the Society for Computer Simulation International*, 13(3):135–154, 1996.

20. E. Kofman. A Third Order Discrete Event Simulation Method for Continuous System Simulation. *Latin American Applied Research*, 36(2):101–108, 2006.

21. J. Liu. *Parallel Discrete-Event Simulation*, volume Wiley Encyclopedia of Operations Research and Management Science. June 2010.

22. Q. Liu. *Algorithms for Parallel Simulation of Large-Scale DEVS and Cell-DEVS Models.* PhD thesis, Systems and Computer Engineering Dep. Carleton University, Sep 2010.

23. M. Maggio, K. Stavåker, F. Donida, F. Casella, and P. Fritzson. Parallel Simulation of Equation-based Object-Oriented Models with Quantized State Systems on a GPU. In *Proceedings of 7th Modelica Conference*, Como, Italy, 2009.

24. D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. Technical report, Intel Technology Journal, 2002.

25. G. Migoni and E. Kofman. Linearly Implicit Discrete Event Methods for Stiff ODEs. *Latin American Applied Research*, 2009. In press.

26. G. Migoni, E. Kofman, and F. Cellier. Quantization-Based New Integration Methods for Stiff ODEs. *Simulation: Transactions of the Society for Modeling and Simulation International*, 2011. in Press.

27. J. Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18:39–65, March 1986.

28. J. Nutaro. *Parallel discrete event simulation with application to continuous systems.* PhD thesis, 2003. AAI3119971.

29. J. Nutaro. A discrete event method for wave simulation. *ACM Trans. Model. Comput. Simul.*, 16(2):174–195, Apr. 2006.

30. C. Perfumo, E. Kofman, J. Braslavsky, and J. K. Ward. Load management: Model-based control of aggregate power for populations of thermostatically controlled loads. *Energy Conversion and Management*, 55:36–48, 2012.

31. PowerDEVS site at SourceForge - http://sourceforge.net/projects/powerdevs/.

32. H. Praehofer and G. Reisinger. Distributed simulation of DEVS-based multiformalism models. In *AI, Simulation, and Planning in High Autonomy Systems, 1994. Distributed Interactive Simulation Environments., Proceedings of the Fifth Annual Conference on*, pages 150 –156, dec. 1994.

33. F. Randima. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

34. D. Rao, N. Thondugulam, R. Radhakrishnan, and P. Wilsey. Unsynchronized parallel discrete event simulation. In *Simulation Conference Proceedings, 1998. Winter*, volume 2, pages 1563 –1570 vol.2, dec 1998.

35. V. Savcenco and R. Mattheij. A Multirate Time Stepping Strategy for Stiff Ordinary Differential Equations. *BIT Numerical Mathematics*, 47:137–155, 2007.

36. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

37. T. K. Som and R. G. Sargent. Model structure and load balancing in optimistic parallel discrete event simulation. In *Proceedings of the fourteenth workshop on Parallel and distributed simulation*, PADS '00, pages 147–154, Washington, DC, USA, 2000. IEEE Computer Society.

38. Y. Tang, K. Perumalla, R. Fujimoto, H. Karimabadi, J. Driscoll, and Y. Omelchenko. Optimistic parallel discrete event simulations of physical systems using reverse computation. In *Principles of Advanced and Distributed Simulation, 2005. PADS 2005. Workshop on*, pages 26 – 35, june 2005.

39. K. Yaghmour. Adaptive Domain Environment for Operating Systems. 2002. www.opersys.com/adeos/.

40. K. Yaghmour. Building a Real-Time Operating Systems on top of the Adaptive Domain Environment for Operating Systems. 2003. www.opersys.com/adeos/.

41. B. Zeigler. *Theory of Modeling and Simulation*. John Wiley & Sons, New York, 1976.

42. B. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation - Second Edition*. Academic Press, 2000.