SysRT: A Modular Multiprocessor RTOS Simulator for Early Design Space Exploration

Jun Xiao*, Andy Pimentel* and Giuseppe Lipari†
*University of Amsterdam, Amsterdam, The Netherlands
†University of Lille, Lille, France
Email: J.Xiao@uva.nl, A.D.Pimentel@uva.nl, Giuseppe.Lipari@univ-lille1.fr

***Abstract*—Modern embedded systems increasingly accommodate several applications running concurrently on a multiprocessor platform managed by a real-time operating system (RTOS). The increasing design complexity of such systems calls for good design tools to evaluate real-time performance during the very early stages of design. To this end, fast system-level simulators that allow for efficient hardware/software co-simulation are essential. In this paper, we present SysRT, a generic and high-level RTOS simulator that is highly suited for early design space exploration (DSE). The simulator contains different types of application models and a modular RTOS kernel model, all developed in SystemC. Efficient and precise modeling of preemptive scheduling is achieved via an event-driven simulation approach, allowing simulations to be performed much faster than cycle-accurate simulations. At the same time, the kernel model is developed to be generic and modular to support for easy plug-in of new schedulers as well as new resource sharing protocols. Comparing SysRT with state-of-art simulators, it achieves faster simulation speeds with an identically small simulation error. We demonstrate the flexibility of SysRT and its benefits for early DSE using experiments with a mixed workload executing on multiprocessor platforms with different numbers of cores.**

I. Introduction

In the past years, the design of systems-on-chip (SoCs) has become increasingly complex. Hardware architectures are migrating from simple single-core based systems to more complex multi-core architectures. In the embedded systems domain, together with the increasing hardware complexity, the software complexity has also been growing dramatically. Modern embedded systems increasingly execute several applications of different types concurrently on the underlying computing platform. These applications can have different execution requirements. For example, control applications typically are hard real-time applications and thus have stringent timing constraints, while best-effort applications prefer a short task response time. These systems are usually managed by a Real-Time Operating System (RTOS).

Raising the level of abstraction is generally considered as a solution to address the design complexity, thus reducing time-to-market. To provide a simulation environment and to help in the design space exploration (DSE) at the early stages of design, various system-level design languages (SLDL) and methodologies have been proposed, such as SystemC [1] and SpecC [2]. Originally, SLDLs primarily focused on hardware modeling and did not properly address the modeling of software aspects. Later efforts introduced methods to model timing behavior of software in SLDLs. But most solutions still lack direct support for simulating the real-time behavior of concurrent applications, such as preemption or scheduling within the RTOS. To verify that the timing requirements posed by applications are met during the early stages of design, a fast system-level simulator, capturing both the modeling of software and hardware, is needed.

In this paper, we present SysRT [3], a generic and modular high-level RTOS simulator that is highly suited for early DSE to study RTOS design alternatives. The SystemC-based SysRT simulator improves on current state-of-art RTOS simulators by providing the unique combination of being, at the same time, highly accurate, efficient and easy to extend to facilitate early DSE. SysRT contains different types of application models, an RTOS kernel model and an abstract architecture model. Efficient and precise modeling of preemptive scheduling is achieved via an event-driven simulation approach, which utilizes scheduling events associated with task states and interrupts. At the same time, the kernel model is developed to be generic and modular to support for easy plug-in of new schedulers as well as new resource sharing protocols. We have compared the accuracy and simulation performance of SysRT with state-of-art RTOS simulators, of which the results show that our simulator is faster while still producing the most accurate results.

The rest of the paper is organized as follows. Section II gives an overview of the related work. The overall RTOS simulation framework is described in Section III. Section IV describes the application models. In Section V, the kernel model is detailed, and Section VI presents a range of experimental results. Section VII concludes the paper.

II. Related Work

The modeling and simulation of RTOS with SLDL have received widespread attention from many researchers, [4], [5], [6]. In [7], the modeling capability of SystemC has been extended by RTOS services to provide more realistic software modeling features. However, to realize features such as preemption and scheduling, a scheduler model is invoked every simulation quantum, similar to the way a real OS scheduler behaves. This quantum-granularity based simulation approach therefore introduces large overheads, resulting in low simulation speeds. Later efforts such as [8], [9] focused on improving the accuracy of high-level simulation via while

maintaining high performance. However, these works still trade-off speed for accuracy.

In [10], a host-compiled multi-core system simulator is presented for early real-time performance evaluation. They present an integrated approach for automatic timing granularity adjustment to optimally navigate simulation speed versus accuracy. This approach switches between prediction mode and fallback mode. In prediction mode, a prediction of the next scheduling points is performed based on the simulation parameters and states of periodic tasks. Schirner et al. [11] introduce preemptive scheduling in abstract RTOS models using Result Oriented Modeling (ROM). To speed up simulation, ROM optimistically predicts the finish time of a process already at the start time by a "run to finish" assumption. ROM records any possible preemption that may alter the predicted outcome. While time passes, it validates the prediction and takes corrective measures to ensure accuracy. However, predictions of preemption points are difficult if the simulation uses more complex task models like Directed Acyclic Graphs (DAGs) and resource sharing models as supported by SysRT.

SysRT provides a framework of RTOS services in SystemC that allows developers and researchers to easily explore and validate embedded RTOS design alternatives. Compared with quantum-granularity based simulators and prediction-based simulators, SysRT has two main advantages: (i) it has been developed to be generic and modular to support for easy plug-in of new schedulers as well as new resource sharing protocols. Thus, it is more flexible to simulate various real-time scheduling algorithms; (ii) it typically achieves higher simulation speeds via an event-driven simulation approach while obtaining identical accuracy results.

III. Modeling Framework

SysRT consists of three layers, as shown in Fig. 1: the *application layer*, the *kernel layer*, and the *architecture layer*. In the application layer, the user can model a set of *processes*. A process can be a single job instance (named ST in Fig. 1), a *Periodic Task* (PT) of which job instances are invoked periodically, or a process with execution precedences modeled by a DAG, as will be explained in Section IV.

The application layer interacts with the RTOS kernel layer. The application informs the kernel of its execution states, while the kernel model returns task scheduling decisions. We model four functionalities of the OS kernel, namely process management, resource management, interrupt handling and real-time scheduling. A queue in the OS kernel is used to order the tasks that become ready for execution. The OS kernel further has a resource manager sub-module that controls access to resources shared between tasks. The resource block queues store tasks waiting to get access to a particular resource due to mutual exclusion. Moreover, interrupt service routines are defined in the OS kernel model. When an interrupt is generated, either from software or hardware, the OS kernel schedules the corresponding interrupt handler depending on the handler priority. Different real-time (preemptive) schedulers are implemented in the scheduling module of the OS kernel model. The architecture layer models the hardware computing platform. It specifies the number of cores in the SoC platform, the interconnection between the cores, and the hardware interrupt interfaces. The current architecture model mainly accounts for the scheduling overhead including migration and context switching overhead after a scheduling decision is made by the OS kernel. The implementation details of the architecture model are beyond the scope of this paper.

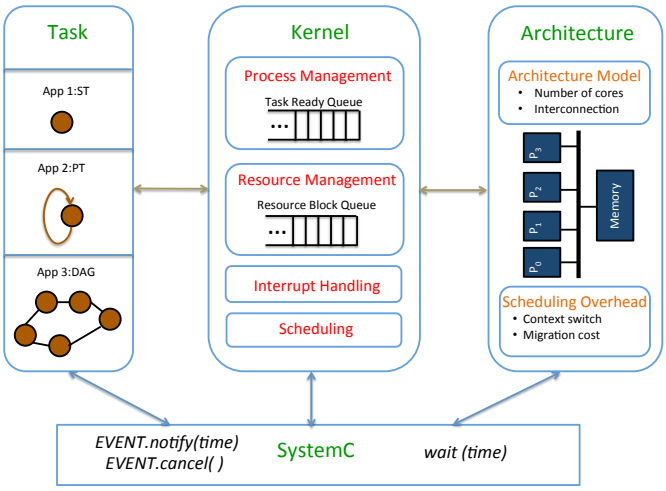


Fig. 1: Simulation framework of SysRT.

Application layer, OS kernel layer and architecture layer are implemented on top of the basic classes and primitives provided by SystemC. We use event-driven simulation, where events are modeled by the *sc_event* class. This class allows explicit triggering of events by means of a notification method. The *Event.notify(sc_time t)* method notifies or posts an event after time *t*. If a simulation process is set to be sensitive to an event, then this process acts as the corresponding event handler. When an event occurs, the corresponding event handler is invoked and scheduled by the SystemC simulation kernel. Scheduled events may be canceled with the *event.cancel()* method.

Modelling preemption is always a challenging topic for a RTOS simulator. Most RTOS simulators that are built on top of SystemC use *wait(sc_time time)* to model task execution latency. If a task is preempted for some time, then the preemption time is counted as extra task execution latency, resulting in another execution of *wait(sc_time time)* for that task. However, this approach comes with a speed penalty due to the frequent computations of the preemption time and the frequent executions of *wait(sc_time time)*. Unlike this approach, SysRT adopts an event-driven approach that uses only *sc_event* to model preemption. Events are extracted from the task execution states, which will be discussion soon. Once a task is preempted, the only work to do is to cancel the task finishing event. When this task is scheduled again, a new task finishing event is posted after the remaining execution time. compared with the *wait(sc_time time)* method, this event-driven approach introduces less simulation overhead.

IV. Application Model

The *Application* is a program that contains a set of coordinated tasks modeled by the user through the *Task* module. In this work, the actual task functionality is abstracted away, and only the timing of task execution is simulated. Here, we assume that timing information of task execution latencies are estimated or known a priori.

A. Task Model

In the task model, three kinds of constraints specified on real-time tasks are considered: timing constraints, precedence relations, and access control on shared resources. Timing constraints, such as execution times and job deadlines, are specified at the creation of a real-time task object. Precedence constraints are realized by a DAG task model [12]. Contention on shared resources is simulated by adding *wait/signal* instructions in the task execution routine, as will be explained below.

A task module contains a list of high-level instructions that are executed in sequence. *Instruction* sub-modules are added to a *task* module by the *InsertCode* method. For example, consider a task T_1 that computes for 500 milliseconds, then tries to get access to a shared variable R_1 after which it occupies the resource for 50 milliseconds once the access is granted, and after releasing the shared resource the task finishes its current job by computing for another 300 milliseconds. This can be modeled by: T_1.*InsertCode*("*execute(500)*; *wait*(R_1)*; execute(50)*; *signal*(R_1)*; execute(300)*"). Details about the instruction module will be described in Section IV-B.

The simulation is driven by events generated by the first job of each task. The typical events generated for a task are illustrated in Fig. 2. A *job_arrival* event is posted at the activation offset (start time) ϕ_i by the *start_of_simulation()* method in the *Task* module which is called at the beginning of the simulation. A *job_arrival* event is notified every time when the task becomes ready to execute. Between the job arrival time and finish time, a job may miss its relative deadline. For such cases, a *deadline_miss* event is posted at time ϕ_i+D_i, where D_i is the relative deadline of task i. The action of the *deadline_miss* event handler is specified by the user. Possible actions are to kill the job instance, to ignore the deadline miss or even to stop the simulation. Once a job starts its execution, a *job_end* event is posted at time ϕ_i+C_i, where C_i is the execution latency of task i. The responsibility of the *job_end* event handler is to cancel the pending *deadline_miss* event and to call the kernel interface to inform it to schedule another task. A schedule event is posted by the OS kernel to a specific task if it was selected to be scheduled. The schedule event handler *schedule()* then schedules the instructions of the task. A *deschedule* event is generated if a task is preempted by another task with a higher priority. The *deschedule* event handler *deschedule*() cancels the pending *job_end* event, records the current time stamp and computes the executed job length. When the task is re-scheduled, a new *job_end* event is posted for the job's remaining execution time.

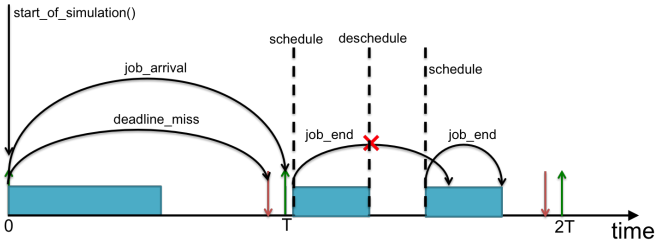


Fig. 2: Task events.

The UML class diagram of task modules is shown in Fig. 3(a). *AbsTask* defines the interface that must be implemented by a general task. It includes an *activate()* method, which activates the task, as well as *schedule()/deschedule()* methods, which modify the task state and related variables when a task is scheduled/descheduled. *AbsRTTask* defines the interface that should be provided by a real-time task and contains methods for getting the absolute and relative deadline of a task.

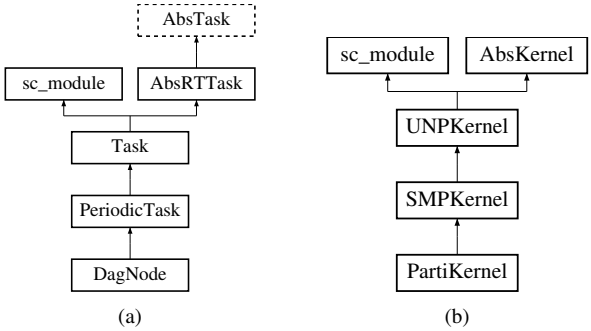


Fig. 3: (a) Task module and (b) Kernel module.

Periodic Task Model: Periodic tasks consist of a number of instances or jobs that are regularly activated at each period. Periodic tasks are reactivated by the *job_arrival* event handler, which posts a new *job_arrival* event at the next period.

DAG Task Model: A DAG is a graph of real-time subtasks (also called nodes) that captures their execution precedences. The subtasks share the same deadline and period but differ in their WCET. The *DagNode* module is used to construct a DAG application model in SysRT.

B. Instruction Model

Instructions inside tasks are modeled using the *Instruction* class. There are two kinds of instructions. First, *execute(sc_time time)* is used to model the execution time required to execute a real code segment in an application. It can be described by a random variable, making it is possible to model a portion of code with an arbitrarily distributed random execution time. The other instruction type is *wait(Resource res)/signal(Resource res)*, which models the request or release of a shared resource. A task executes all the instructions in sequence. A job instance is completed only after its last instruction was executed. If a task is activated again (i.e. firing

a new job), then the instruction pointer is reset to the first instruction.

The *schedule/deschedule* event propagates from a task to its instructions. If a task is selected to execute at time *t*, the task calls its instruction interface and notifies a *schedule* event in the *Instruction* module. Suppose that the execution duration of the instruction is *instr_time*, the *schedule* event handler in the *Instruction* module will post an *end_instr* event at time $t+instr\_time$. The *end_instr* event handler increments the instruction pointer to the next instruction in the task and posts a new *end_instr* event for the next instruction. If there are no more instructions to execute, the interface of the task module is invoked and a *job_end* event is posted. During instruction execution, a task may be preempted and rescheduled. A similar event propagation mechanism between a task and its instructions applies to the *deschedule* event.

Based on the assumption that the actual requesting and releasing of a resource takes zero time, the *end_instr* event is notified immediately if the current scheduled instruction is *wait* or *signal*. The *end_instr* event handler for the *wait* instruction communicates with operating system kernel by calling the interface *request_resource(Kernel, Resource, resource_quantity)*. As a result, the task gets the resource if a sufficient quantity of that resource is available. Otherwise, the task is blocked by the operating system kernel. For the *signal* instruction, the *end_instr* event handler invokes the interface *release_resource(Kernel, Resource, resource_quantity)* in the operating system kernel module. The task releases the resource quantity used.

V. RTOS Kernel Model

Fig. 3(b) shows the UML class diagram of the OS kernel module. The AbsKernel class is an abstract class that defines the minimal functionality of a kernel. The *UNPKernel* and *SMPKernel* classes are implemented to model an OS kernel running on a uniprocessor system (UNP) or a symmetric multiprocessor system (SMP), respectively. Traditional real-time multiprocessor schedulers can be classified in two categories: global and partitioned schedulers. Global Earliest-Deadline-First (G-EDF) and Partitioned-EDF (P-EDF) are examples of each category. The *SMPKernel* class models a general OS kernel with a global scheduler, whereas the *PartiKernel* class models an OS kernel with partitioned schedulers.

In this work, we mainly consider services of process management, resource management, interrupt handling and real-time scheduling provided by the OS kernel. We have developed the modules of the OS kernel model with the aim to provide a flexible and extendable framework to facilitate implementation, testing and evaluation of different real-time schedulers with various resource sharing protocols.

A. UNPKernel Model

The *UNPKernel* module is developed to model a real-time OS kernel running on a uniprocessor. It contains sub-components such as the *Scheduler* module and the *ResManager* module that is responsible for performing resource access related operations. These sub-components are set through methods *set_sched (Scheduler* s)* and *set_resmanager (ResManager* rm)*.

At initialization, a *CPU* pointer, which points to the modeled architecture, is created in the *UNPKernel* module to get information of the architecture platform. Since at most one task is allowed to execute at a time in a uniprocessor system, one pointer *cur_exe* is enough to track the current executing task.

For the communication with tasks, the *UNPKernel* module provides several functions. These include the functions *Arrival(AbsRTTask* t)* and *End(AbsRTTask* t)*. The function *Arrival(AbsRTTask* t)* is called by the task arrival event handler. This method inserts the task in the ready queue, followed by a function call to make a schedule decision. *End(AbsRTTask* t)* is invoked by a task when the task completes its execution. This function removes the task from the ready queue and sets the *cur_exe* pointer to null. To suspend a task, the UNPKernel class implements a *Suspend(AbsRTTask* t)* function. This function removes the task from the ready queue. If the task was executing, then it will first be descheduled. When a task is resumed (from suspension by the OS or from being blocked on a resource), the kernel reactivates the task by calling *Activate(AbsRTTask* t)* which simply inserts the task in the ready queue and changes the task's state to ready.

The operation of allocating the CPU for task execution is referred to as dispatching. The dispatching activity is simulated by the *dispatch()* function. Any circumstance that may change the current executing task should invoke *dispatch()* to make a scheduling decision:

- when a new task becomes ready;
- when a task finishes its current job;
- when a task is blocked;
- when an interrupt arrives, activating its corresponding interrupt handler.

On uniprocessor systems, just one execution flow can progress at a time. Therefore, *dispatch()* is simple in *UNPKernel* as compared with its implementation in other kernel modules. It simply compares the executing task with the first task in the ready queue. If they are different, it forces a context switch, which involves the participation of architecture model to simulate the context switch overhead. When the context switch has finished, the kernel schedules the newly dispatched task. Important to realize is that the *dispatch()* function *has been decoupled from the scheduler* that actually determines the order of the tasks in the ready queue, according to the implemented scheduling algorithm.

B. SMPKernel Model

The *SMPKernel* is a module modeling a real-time kernel with a global scheduler for (SMP) multiprocessor systems. On multiprocessor systems, multiple tasks are allowed to run concurrently. The *SMPKernel* module keeps track of the status of each individual processor, storing information about which task is executing on which processor, which tasks are about

to be dispatched to which processor, and whether or not processors are in the process of performing a context switch.

The functions provided to the *Task* module and methods related to process management in the *SMPKernel* module are similar to those in the *UNPKernel* module. However, the function to make a scheduling decision, *dispatch()*, is more complicated. Pseudocode 1 shows the procedure of the *dispatch()* method in *SMPKernel*. In this code, the variable *newtasks* denotes the number of tasks that are not executing but need to be scheduled. Assuming a simulated architecture with *m* processors, *newtasks* therefore equals to the number of tasks that are among the first *m* tasks in the task ready queue that are not yet executing or being dispatched. Newly scheduled tasks are dispatched to free processors if there are any available. If all processors are busy, then task preemption will take place.

Pseudocode 1: The procedure of $dispatch()$ with a system architecture with m processors

```
1:  while newtasks > 0 do
2:      t^new ← first non-executing task in ready queue that
        needs to be scheduled (i.e., among the first m entries)
3:      c ← find next free core {return NULL if no more free
        cores}
4:      if c == NULL then
5:          t_remove ← first executing task in ready queue not
            part of the first m entries ;
6:          c ← get the index of core executing task t_remove
7:      end if
8:      dispatch_to_proc(t^new, c)
9:      newtasks ← newtasks - 1
10: end while
```

The *dispatch()* method decides on the index of the selected cores for task dispatch. By calling *dispatch_to_proc(Task * newtask, CPU *c)*, the OS kernel also deschedules any task currently executing on processor c and computes the scheduling overhead including the context switch and task migration costs. The computed scheduling overhead is passed from the kernel layer to the architecture layer, which subsequently simulates this overhead. Hereafter, a newly dispatched task is selected to start execution on processor c. The procedure of *dispatch_to_proc(Task * newtask, CPU *c)* is shown in Pseudocode 2.

C. PartiKernel Model

In a partitioned scheduler, ready tasks are first inserted in a global ready queue. Through this global scheduler, ready tasks are then dispatched to a specific local task queue according to the task's affinity. Each processor has its own local queue in which the order depends on the local scheduler. Each processor may use a different scheduler. Since the structure of such a partitioned scheduler is different from the global scheduler, a different kernel module, *PartiKernel*, has been implemented to facilitate the development of partitioned schedulers. The interface provided to the *Task* module and

Pseudocode 2: The procedure of $dispatch\_to\_proc(Task * newtask, CPU * c)$

```
1:  AbsRTTask current_task ← the task currently executing
    on core c
2:  if current_task ≠ NULL then
3:      deschedule current_task
4:  end if
5:  if newtask == NULL then
6:      RETURN
7:  else
8:      prepare newtask to execute on core c
9:  end if
10: Compute the scheduling overhead
11: Send the overhead to architecture model
```

functions related to process management in the *PartiKernel* module are slightly different than those in *SMPKernel* due to task affinity. However, the *dispatch()* method has been completely re-implemented. If a task is inserted to or is removed from a local queue, instead of calling *dispatch()*, *PartiKernel* invokes a *dispatch(CPU *cpu)* function that passes the task affinity as a parameter to make a local rescheduling decision for the processor in question. Changes on a local queue have no effect on the ordering of other local queues. In this sense, the *dispatch(CPU *cpu)* function is similar to *dispatch()* in *UNPKernel*.

D. Scheduler Model

When a task becomes ready to execute, it is inserted to the ready queue managed by the scheduler, which is a sub-component of a kernel module. The ready queue is ordered by task priority assigned by the scheduling algorithm. At a scheduling point, the scheduler (i.e. dispatcher) is responsible for selecting the task(s) at the front of the ready queue to execute. In SysRT, the following schedulers have currently been implemented:

- Global Earliest Deadline First [13] (G-EDF)
- First Come First Out (FIFO)
- Fixed Priority Scheduler (FPS)
- Rate Monotonic Scheduler (RMS)
- Round Robin (RR).
- Proportional Fairness [14] (P-FAIR)
- Partitioned-based Scheduler (PS) including P-EDF
- Non-Preemptive EDF (NP-EDF)

E. Resource Management Model

The *Resource* module models a resource shared by two or more tasks. It provides an interface to the OS kernel module to, for example, perform locking operations for providing access to these shared resources. The resource availability is checked by the method *IsAvailable(int amount)*. It returns false if the quantity of a certain resource is not sufficient. Every task uses resources through a critical section surrounded by *wait* and *signal* instructions. If the executing task requests/releases a certain resource quantity, the resource

manager in the OS kernel invokes the interface of the resource, *lock(int amount)/unlock(int amount)*, to decrease/increase resource availability for that particular resource.

The *ResManager* module models a resource manager that implements the resource accessing protocol. It contains multiple block queues, each associated with a particular resource to store tasks blocked on that resource. These block queues are ordered by task priority. Different resource sharing protocols can be implemented by the *ResManager* module. Taking the Priority Inheritance Protocol [15] as an example, requesting a resource is implemented by first checking the availability of the requested resource. If there are not enough available resources, the resource manager calls the kernel interface to suspend the task that is requesting the resource. Furthermore, the priority of the resource owner is changed to the maximum priority of those tasks that are blocked for the resource. If the requested resources are available, the resource manager invokes the unlock interface of the resource and grants the resources to the task. Releasing a resource unlocks the resources and changes the priority of the releasing task back to its original priority, after which it checks if the resource block queue is empty. If the queue is not empty, the resource manager removes the first task from the block queue, and activates the task through the kernel interface and locks the resource for the new owner.

VI. Experimental Results

In this section, we evaluate the accuracy and simulation performance of SysRT, and demonstrate its flexibility and benefit in DSE. All experiments were conducted on a 3.4GHZ Intel Core I5. The default time unit of the task parameters in the following experiments is the simulation resolution set by SystemC.

A. Simulation performance and accuracy

The first experiment is to evaluate the accuracy and simulation performance of SysRT by comparing it with four other simulators: the state-of-art (prediction-based) HCSim simulator [10] and three conventional quantum-granularity based simulators (also described in [10]) with a simulation quantum of 1ms, 10ms and 100ms, respectively. All simulators model a Partitioned-Fixed Priority scheduler, where tasks have been uniformly partitioned over the simulated processors. Task execution costs and periods, priorities are randomly distributed over the intervals [50ms, 150ms], [100ms,10s] and [1, 100], respectively. The simulated time is 10 minutes. Note that all these tasks are not necessarily real-time tasks.

Figures 4 (a), (b) and (c) show the simulation times taken by each simulator simulating a different number of processors, ranging from 1 to 16, where the number of tasks is 16, 100 and 1000. Figure 4 clearly shows that SysRT achieves the fastest simulation speed in these experiments. Both SysRT and HCSim are scalable with respect to the number of processors and the number of tasks. The simulation speed of the conventional simulator with largest simulation quantum is similar to that of HCSim and SysRT. However, it suffers from a lower accuracy, as will be discussed later on. Conventional simulators get much slower if the simulation quantum size decreases.

To derive a reference for the task response times, we have also performed the experiment with the same task sets on a real Linux-based RTOS, i.e. Litmus [16], varying the number of active processors from 1 to 4. For each task, we calculate the relative errors between the response times obtained from simulators and the actual response times from Litmus. The accuracy is measured by the average error of all tasks in the testing task set.

Table I is the average simulation error of those tests. The number of active processors and the number of tasks in different testing sets is not reported since it turns out that these factors have little effect on the relative error of each individual task. SysRT, HCSim and conventional simulation with the smallest simulation quantum yield high accuracy, whereas conventional simulators with a larger simulation quantum suffer from degraded accuracy.

TABLE I: Average Simulation Error of Five Simulators

HCSim	SysRT	Quantum:1ms	Quantum:10ms	Quantum:100ms
0.166%	0.166%	0.166%	4.182%	>100%

Note that, although SysRT and HCSim are supposed to be theoretically accurate, several factors in Litmus such as context switches and kernel tasks with high priorities could lead to small simulation errors. Fortunately, both SysRT and HCSim provide support to model the scheduling overhead to improve accuracy.

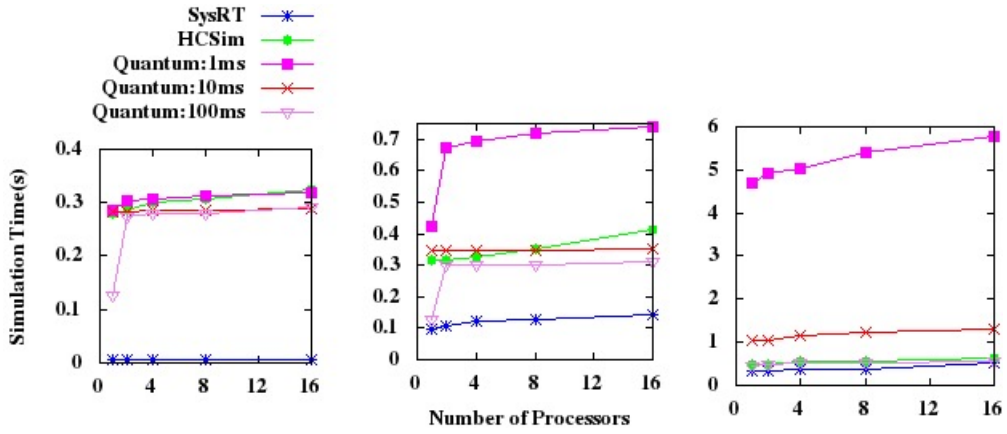
B. Flexibility of SysRT

As most prediction-based RTOS simulators do not support simulating real-time resource access protocols due to difficulties in predicting preemption points, we show the flexibility of SysRT by simulating a set of four periodic tasks T_1, ..., T_4 that exclusively access two shared resources R_1 and R_2. Task parameters are listed in Table II. P_i is the task activation period and C_i the execution time. Variable $\xi_{j,i}$ denotes the duration of the critical section that T_i occupies R_j. The value 0 for $\xi_{j,i}$ means that T_i does not use R_j. Tasks are scheduled on an uniprocessor by a RM scheduler with priority inheritance as resource sharing protocol.

TABLE II: Task Parameters and Theoretical WCRT.

Tasks	P_i	C_i	$\xi_{1,i}$	$\xi_{2,i}$	$WCRT_i$
T_1	100	5	0	0	5
T_2	110	16	3	3	71
T_3	200	70	20	0	142
T_4	350	102	0	30	310

The analytically calculated Worst Case Response Time (WCRT) for each task is given in the last column of Table II. We have run the simulation for 80000 time units. The simulated response time of the first 200 jobs of each task are shown in Fig. 5. As can be seen from Fig. 5, the response times obtained from simulation are consistently lower than



(a) 16 tasks (b) 100 tasks (c) 1000 tasks

Fig. 4: Simulation time of five simulators.

the theoretical WCRTs. Thanks to the modular and flexible implementation of SysRT, the resource sharing protocol is correctly simulated.

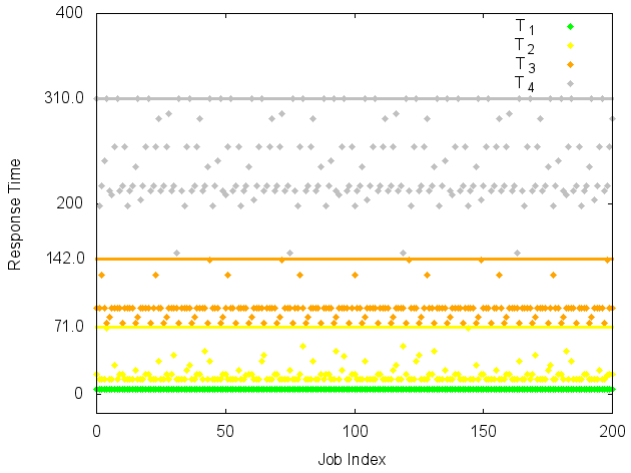


Fig. 5: Response time of jobs in tasks.

C. Benefit of SysRT in DSE

The second experiment demonstrates the flexibility of SysRT and its benefits for early DSE. An embedded system with a mixed application workload is simulated. The task set is composed of three Hard Real-Time (HRT) tasks, five Soft Real-Time (SRT) tasks and three Best-Effort (BE) tasks. Task types, parameters and utilization (P_i divided by C_i) are listed in Table III. If an interval [a, b] is assigned to P_i (or C_i), then P_i (or C_i) is a random variable uniformly distributed in that interval. This models workload variations.

The application requirement for hard real-time tasks is to guarantee that deadlines are always met. SRT tasks are allowed to miss deadlines, thus their performance is measured by the deadline miss ratio. For best-effort tasks, the performance is calculated by their average response time. We have run

TABLE III: Task Type and Parameters.

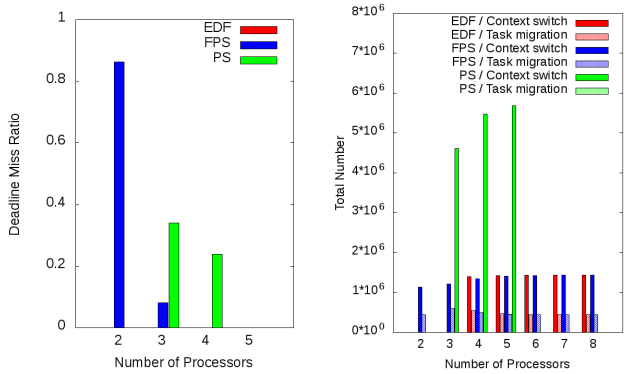
Tasks	Type	P_i	C_i	U_i
T_1	HRT	50	20	0.4
T_2	HRT	90	30	0.333
T_3	HRT	140	50	0.357
T_4	SRT	190	30	0.157
T_5	SRT	350	80	0.228
T_6	SRT	500	170	0.34
T_7	SRT	1000	[200, 700]	[0.2, 0.7]
T_8	SRT	1300	[500, 900]	[0.385, 0.692]
T_9	BE	[1000, 5000]	200	[0.04, 0.2]
T_{10}	BE	[3000, 9000]	500	[0.056, 0.167]
T_{11}	BE	[5000, 15000]	1500	[0.1, 0.3]

simulations with three kinds of schedulers on different architecture models. EDF and FPS schedulers are tested with systems containing 2 to 8 processors, and a partitioned-based scheduler (PS) has been tested for systems with 3 to 5 cores. For the latter, Table IV lists the local scheduling policies and scheduled task(s) on each processor. The simulation is aborted if a HRT task misses a deadline.

TABLE IV: Patitioned-based Scheduler Configuration.

# Processors	Processor	Local Scheduler	Tasks
3	1	FPS	T_1,T_2,T_9,T_{11}
	2	EDF	T_3,T_4,T_6
	3	RR	T_5,T_7,T_8,T_{10}
4	1	P-FAIR	T_1,T_2,
	2	FPS	T_3,T_4,T_6
	3	EDF	T_5,T_7,T_8
	4	RR	T_9,T_{10},T_{11}
5	1	P-FAIR	T_1,T_2,T_5
	2	FPS	T_3
	3	NP-EDF	T_4,T_7
	4	EDF	T_6,T_8
	5	RR	T_9,T_{10},T_{11}

The average deadline miss ratio of the five SRT tasks is shown in Fig. 6(a). The deadline miss ratio decreases as the number of processors increases and becomes 0 for five processors. HRT tasks are not schedulable under EDF if the number of processors is less than four, thus no results are



(a) (b)

Fig. 6: (a) Average deadline miss ratio (b) Scheduling overhead.

plotted for EDF for 2 and 3 processors.

Fig. 6(b) shows the scheduling overhead including the total number of context switches and task migrations. It is interesting to observe that partitioned schedulers have no task migration but suffer from a large number of context switches incurred by P-FAIR, which serves as a local scheduler.

Fig. 7 illustrates the average response times of the BE tasks. As the number of processors increases, the average response time becomes smaller. The response times are very large if the number of processors is less than 4, thus they are not plotted. Evidently, such system performance estimates as obtained by SysRT are helpful to make design decisions at the very early system design stages.

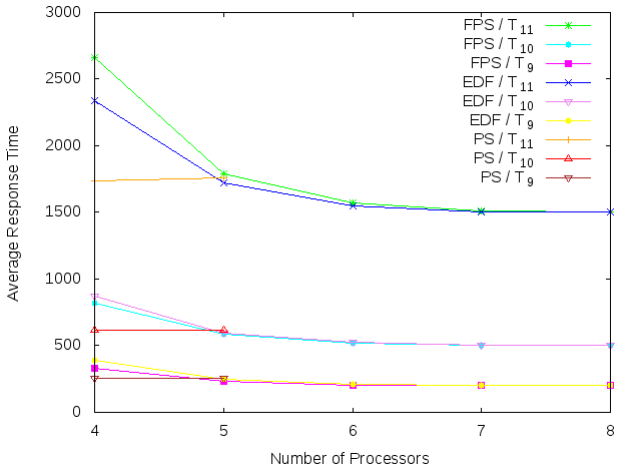


Fig. 7: Response time of BE tasks.

VII. Conclusion

In this paper, we presented SysRT, a generic and high-level SystemC-based multiprocessor RTOS simulator. It provides the unique and novel combination of being highly accurate, efficient and easy to extend to facilitate early DSE. To this end, it contains different types of application models and a modular RTOS kernel model. Efficient and precise modeling of preemptive scheduling is achieved via an event-driven simulation approach. Its modular design allows for easy plug-in of new schedulers as well as new resource sharing protocols. Comparing SysRT with state-of-art simulators, it achieves faster simulation speeds with the same small simulation error. We demonstrated the flexibility of SysRT by experiments with a mixed workload executing on multiprocessor platforms with different numbers of cores.

For future work, we plan to establish co-simulation with more advanced hardware simulators developed in SystemC. Moreover, we are interested in designing and studying new schedulers for mixed application workloads executing on embedded platforms.

References

[1] SystemC, http://www.accellera.org.

[2] SpecC, http://www.cecs.uci.edu/ specc/.

[3] SysRT, https://github.com/jxiao90/SysRT.

[4] Y. Yi, D. Kim, and S. Ha, "Fast and time-accurate cosimulation with os scheduler modeling," *Des. Autom. Embedded Syst.*, vol. 8, no. 2-3, pp. 211–228, Jun. 2003. [Online]. Available: http://dx.doi.org/10.1023/B:DAEM.0000003963.20442.29

[5] H. Zabel, W. Müller, and A. Gerstlauer, *Accurate RTOS modeling and analysis with SystemC*. Netherlands: Springer Netherlands, 2009, pp. 233–260.

[6] R. L. Moigne, O. Pasquier, and J. P. Calvez, "A generic rtos model for real-time systems simulation with systemc," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 3, Feb 2004, pp. 82–87 Vol.3.

[7] P. Hastono et al., "Real-time operating system services for realistic systemc simulation models of embedded systems," in *Proc. of FDL'04*, 2004, pp. 380–391.

[8] R. S. Khaligh and M. Radetzki, "Modeling constructs and kernel for parallel simulation of accuracy adaptive tlms," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 1183–1188.

[9] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and accurate resource conflict simulation for performance analysis of multi-core systems," in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–6.

[10] P. Razaghi and A. Gerstlauer, "Host-compiled multicore system simulation for early real-time performance evaluation," *ACM Trans. Embed. Comput. Syst.*, no. 5s, pp. 166:1–166:26, Dec. 2014.

[11] G. Schirner and R. Dömer, "Introducing preemptive scheduling in abstract rtos models using result oriented modeling," in *Proc. of DATE'08*, New York, NY, USA, 2008, pp. 122–127.

[12] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*, ser. RTSS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 217–226. [Online]. Available: http://dx.doi.org/10.1109/RTSS.2011.27

[13] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973. [Online]. Available: http://doi.acm.org/10.1145/321738.321743

[14] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *ALGORITHMICA*, vol. 15, pp. 600–625, 1996.

[15] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, 1990.

[16] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "Litmusrt: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proc. of the 27th IEEE Real-Time Systems Symposium*, 2006, pp. 111–123.