An Efficient and Portable Scheduler for RTOS Simulation and its Certified Integration to SystemC

Hiroaki Nakamura
Tokyo Research Laboratory
IBM Japan, Ltd.

Naoto Sato
Tokyo Research Laboratory
IBM Japan, Ltd.

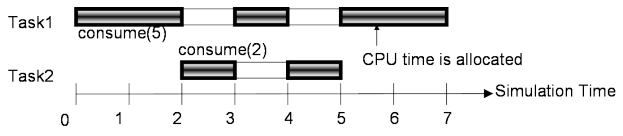
Naoshi Tabuchi
Tokyo Research Laboratory
IBM Japan, Ltd.

Abstract

We propose a new task scheduling algorithm for timed-functional simulation of concurrent software tasks. It attains efficiency by reducing the frequency of context-switching between concurrent tasks. It also provides a high-degree of portability in the sense that it only needs the underlying system to support a very small number of primitives. We provide a concrete implementation built on top of the SystemC scheduler and show some results of preliminary evaluation.

1. Introduction

In the system-level design process, designers first describe functional aspects without specifying implementation details. By attaching *delay annotations* (wait(delay) in SystemC, and waitfor(delay) in SpecC) to the functional model, they can analyze performance related properties. For simulating multi-tasking software running on a single CPU, the delay annotation should be interpreted so that the enclosing task shares the CPU with other tasks. For this purpose, we introduce a new delay annotation consume(*delay*) to specify a *delay*-length of CPU time. For example, the round-robin scheduler interleaves consume(5) at the initial time 0 and consume(2) at 2 in the preemptive manner as follows.



One possible realization of consume is to serialize the delay requests so that only one task is running at any time [1], but it does not support interleaved scheduling as exemplified in the above figure. Another solution is to build an elaborate RTOS scheduler model [2], but it causes too many context switches between the RTOS and software tasks, which would deteriorate the simulation performance. The performance drawback could be mitigated if we can modify the underlying simulation kernel, though this has a negative impact on its portability.

To resolve this challenge, we propose an efficient and portable task scheduling algorithm for simulating software tasks running on an RTOS.

2. A New Task Scheduling Algorithm

```
struct TASK {int remaining_time, checkpoint; bool active;};
const int ntask;
TASK tasks[ntask];
int curr_time, last_time;

void consume(int dt, int taskid) {
  tasks[taskid].remaining_time = dt;
  tasks[taskid].checkpoint = curr_time + dt;
  tasks[taskid].active = false;
  SUSPEND_AND_RESUME(scheduler);
}

void update(int t1, int t2 );
//  requires t1 < t2  and ∀ i. tasks[i].remaining_time ≥ 0
//  ensures  ∀ i.tasks[i].remaining_time ≤ tasks[i].remaining_time@pre
//    and  ∑_i tasks[i].remaining_time@pre - tasks[i].remaining_time ≤ t2 - t1
//  (@pre indicates the value at the start of the execution)

void scheduler_run() {
  update(last_time, curr_time);
  last_time = curr_time;
  for(i=0;i<ntask;i++)
    if( tasks[i].checkpoint == curr_time )
      if( tasks[i].remaining_time == 0 ) tasks[i].active = true;
      else tasks[i].checkpoint = curr_time + tasks[i].remaining_time;
  for(i=0;i<ntask;i++)
    if( tasks[i].active ) SUSPEND_AND_RESUME(tasks[i]);
}

void scheduler_main() {
  curr_time = 0; last_time = 0;
  while(1) {
    scheduler_run();
    curr_time = min_task_checkpoint(); // min{tasks[i].checkpoint | i}
  }
}
```

Figure 1. Scheduling Algorithm

Figure 1 shows our scheduling algorithm that coordinates the execution of software tasks running on a CPU.

3-9810801-0-6/DATE06 © 2006 EDAA

consume(dt, taskid) This procedure sets the initial values for remaining_time and checkpoint of tasks[taskid], and then calls SUSPEND_AND_RESUME(scheduler), which suspends the task and resumes the scheduler. When the scheduler executes SUSPEND_AND_RESUME(task), the software task is resumed and the procedure consume terminates.

update(t1, t2) The procedure update, which implements a specific scheduling policy, allocates a certain period of time between t1 and t2 to each task, which results in reducing the remaining time of the task.

scheduler_main(), scheduler_run() The procedure scheduler_main, which runs in its own thread, repeatedly calls scheduler_run and updates the value of curr_time. The procedure scheduler_run first calls update(last_time, curr_time) and then resumes the task whose remaining time has become '0', by calling SUSPEND_AND_RESUME.

This algorithm accepts preemption requests with no delay, because the procedure consume promptly switches to the scheduler. This algorithm is also efficient in that the procedure scheduler_run does not switch contexts at any points between last_time and curr_time, whose interval is larger than the granularity of time allocated to tasks.

On the other hand, the portability of this algorithm may be limited, because the execution of threads is controlled explicitly using SUSPEND_AND_RESUME, a low-level and platform-dependent synchronization primitive. We modified our algorithm so that it works on top of existing simulation platforms such as SystemC and SpecC. Then, we provided formal proofs of the correctness of the two versions and their equivalence. See [3] for further detail.

3. Implementation and Preliminary Evaluation

3.1. Integration into SystemC

Figure 2 shows the portable version of our algorithm that works on top of SystemC. The common super class Scheduler implements consume, and subclasses of Scheduler implement update according to particular scheduling policies. Scheduler::consume uses two dynamic structures: table maps a task to its remaining time, and tasks maintains the priorities and execution order of the scheduled tasks.

A scheduling policy is encapsulated in the method update of a subclass of Scheduler. An implementation of update for the preemptive priority scheduler assigns the whole duration of time between the two checkpoints to the highest priority task, while that for the round-robin scheduler assigns TIME_SLICE to the scheduled tasks in rotation. While we showed only two variations of scheduling policies, one may model other policies by implementing update for each.

3.2. Preliminary Evaluation

We evaluated the efficiency of our algorithm by comparing the number of context switches with that of a full

```
class Scheduler {
protected:
  map  table;
  list tasks;
  sc_time last_time;
public:
  virtual void update(void) = 0;
  void consume(sc_time& dt, int task) {
    update();
    tasks.put(task);
    table.put(task, dt);
    while (dt > SC_ZERO_TIME) {
      wait(dt);
      update();
      dt = table.get(task);
    }
    table.remove(task);
  }
};
```

```
class PreemptivePriority :
  public Scheduler {
public:
  void update(void) {
    time_to_consume = sc_time_stamp()
                      - last_time;
    if (tasks.notEmpty()) {
      task = tasks.popHighestPriority();
      remaining_time = table.get(task);
      remaining_time = remaining_time
                       - time_to_consume;
      table.put(task,remaining_time);
    }
    last_time = sc_time_stamp();
  }
};
```

```
class RoundRobin : public Scheduler {
public:
  void update(void) {
    time_to_consume = sc_time_stamp() - last_time;
    while (time_to_consume > SC_ZERO_TIME && tasks.notEmpty()) {
      task = tasks.removeFirst();
      remaining_time = table.get(task);
      step_time = min(TIME_SLICE, time_to_consume, remaining_time);
      new_remaining_time = remaining_time - step_time;
      table.put(task,new_remaining_time);
      if (new_remaining_time != SC_TIME_ZERO)
        tasks.putLast(task);
      time_to_consume -= step_time;
    }
    last_time = sc_time_stamp();
  }
};
```

Figure 2. Scheduler Integrated to SystemC

RTOS model that interrupts software tasks for every timing interval. We used a timed-functional simulation program `simple_perf`, which comes with the SystemC reference implementation as the basis of our experiment. This program runs two tasks, the producer and the consumer. The producer sends characters to the consumer via a fifo. We modified the program so that it calls consume instead of wait to specify delays, where consume is processed by a round-robin scheduler whose time slice is 1 time unit.

	Number of Context Switches
Full RTOS Model	20,068,800
Our Algorithm	839,191

The result indicates that our algorithm successfully eliminates 96% of the context switches required by the full RTOS model.

References

[1] A. Gerstlauer, et al. : RTOS Modeling for System Level Design, *Proc. DATE*, 2003.

[2] M. A. Hassan, et al. : RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC, *Proc. DATE*, 2005.

[3] H. Nakamura, et al. : An Efficient and Portable Scheduler for RTOS Simulation and its Certified Integration to SystemC, IBM Research Report RT0627, 2005.