Synthesis of Safe, QoS Extendible, Application Specific Schedulers for Heterogeneous Real-Time Systems

Christos KLOUKINAS [*] Sergio YOVINE [†‡]

VERIMAG,Centre Équation,2 avenue de Vignate,38610 GIÈRES,France
Christos.Kloukinas@imag.fr Sergio.Yovine@imag.fr
February 27, 2003

Abstract

We present a new scheduler architecture, which permits adding QoS policies to the scheduling decisions. We also present a new scheduling synthesis method which allows a designer to obtain a safe scheduler for a particular application and at the same time helps him in analysing the task interactions and the overall system behaviour. Our scheduler architecture and scheduler synthesis method have not been developed for a particular application model and, therefore, can be used for heterogeneous applications, where there are periodic tasks, event-driven ones and tasks which are always enabled and where the tasks communicate through various synchronisation primitives. Finally, we present a prototype implementation of this scheduler architecture and related mechanisms on top of an open-source *OS* for embedded systems.

Keywords: QoS Scheduling Policies, Scheduler Architecture, Scheduler Synthesis, Application Analysis, Hard Real-Time, Heterogeneous Applications

1 Introduction

Safety & mission-critical systems need to be of extremely high quality, due to the great dangers and the high cost of their potential failure. For this reason, when they are multi-threaded they must be guaranteed to be free of deadlocks and all threads must be guaranteed to meet their deadlines under all circumstances.

The current practice for avoiding deadlocks is to use a *priority inheritance protocol (*PIP*)* [13] for the sharing of non-preemptable resources. This is done in order to solve the *priority inversion* problem, which arises when a high priority task is blocked from a lower priority one due to a shared resource [1]. This procedure has a certain number of disadvantages though. First of all, Sha *et al.* [13] have shown that the basic PIP does not guarantee deadlock freedom. For this reason they proposed the *priority ceiling protocol (*PCP*)*, which is not as widely supported in the currently available Real-Time (*R-T*) operating systems (*OS*). Second, all the priority inheritance protocols (the PCP included) are pessimistic in nature and, therefore,

[*] `http://www-verimag.imag.fr/PEOPLE/Christos.Kloukinas`

[†] `http://www-verimag.imag.fr/PEOPLE/Sergio.Yovine`

[‡] This article has been submitted to the 15th Euromicro Conference on Real-Time Systems (ECRTS'03) `http://www.hurray.isep.ipp.pt/ecrts03/`

[1] For an account of the priority inversion problem in the Mars Pathfinder spacecraft, read the RISKS-19.49 and RISKS-19.54 digests at the Risks site (http://www.risks.org).

can refuse access to a shared resource even when there is no real danger of a deadlock at the current situation. Additionally, in order to apply the PCP, the designer of the system must choose a set of priorities for all the tasks in the system. Furthermore, for each shared resource the designer must identify the threads which use it, in order to assign a priority to that resource (*i.e.*, the *ceiling* priority of the resource). More importantly, however, the PCP cannot on its own support tasks which synchronise using monitors and communicate using condition variables [14], as for example is done in JAVA [4]. Sha *et al.* had mentioned that in such cases one should split tasks, which means that the complexity of what the designers have to do in order to use PCP is quite high. Worse yet, not all cases of communication through condition variables can be translated according to these rules, since there can be tasks which wait on a condition variable while still holding some resource locked. These tasks cannot be split since the underlying assumptions of the PCP clearly demand that tasks finish executing without holding any resources. Therefore, it is not straightforward how one can use the PCP with a language such as *R-T* JAVA [11].

Finally, the methods currently used do not allow designers to easily extend them for incorporating QoS to the scheduler decisions. Nevertheless, QoS is becoming more and more important for the systems we are designing. Being able to extend a scheduler with QoS characteristics could allow us to experiment with ways to minimise energy consumption, or further increase the speed of the system, by minimising, for example, the number of context switches. An example of this can be found in [3] where the authors present a dynamic scheduling method which also treats the QoS aspect of the system. However, in this work the authors consider a fixed task model where all tasks are periodic and do not consider deadlock situations or the communication aspect of the system.

In the following, we present a method for synthesising *QoS extendible* and *safe* schedulers following the controller synthesis [18] paradigm and continuing previous work at Verimag [1, 2]. We start in section 2 by presenting the overall architecture of the scheduler we synthesise and how each part of it participates in the control of the system. Then, in section 3 we present the model of the systems we consider and in section 4 the particular method we use for the synthesis of the scheduler. In section 5 we present a prototype implementation of our scheduler on top of a real operating system and we conclude in section 6 with a discussion of our method.

2 Scheduler Architecture

The applications we consider consist of a set of threads synchronising through monitors and communicating through condition variables. These applications also have the following characteristic. All threads and shared objects (*i.e.*, mutexes, condition variables, *etc.*) are created at the initialisation phase of the application and, therefore, are known during the execution phase. Additionally, we only con-

sider applications executing on a single processor for the moment. In this section we present the architecture of the schedulers we synthesise in order to control these applications, which consists of two three-layered stacks, as shown in Figure 1. The left stack is responsible for selecting an application thread for execution. The right stack is responsible for selecting an application thread for the reception of a notification. Being able to control which thread will be notified for a particular event is something that other scheduling policies like the PCP do not offer, since they concentrate only on the selection of threads for execution.

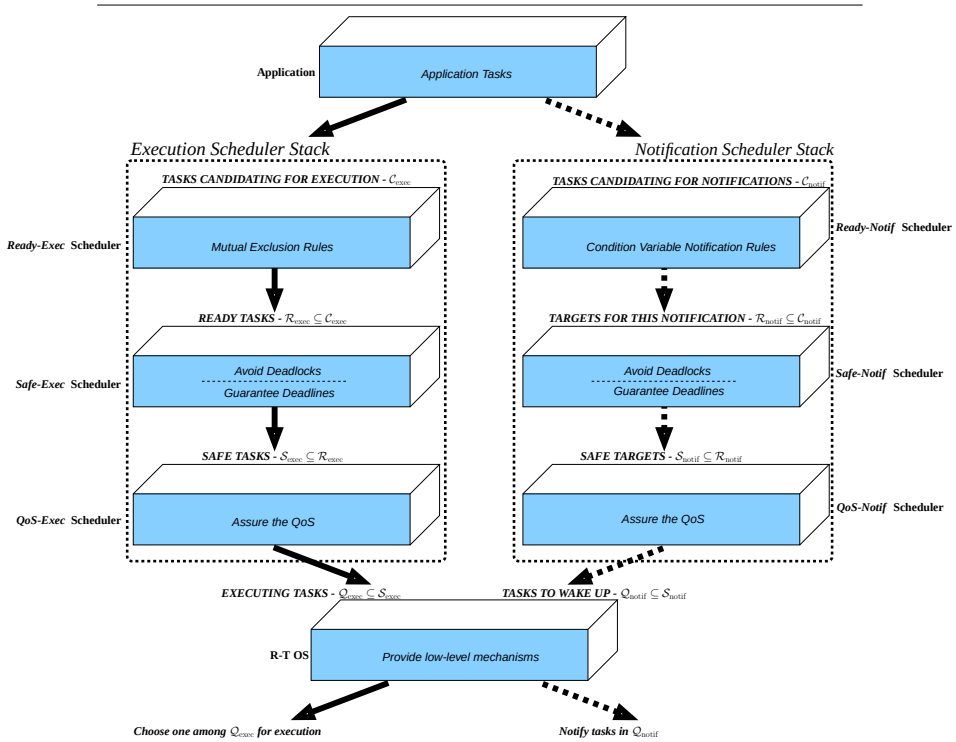


Figure 1: A three-layered scheduler architecture

The left stack of the scheduler is given control by the application when the latter tries to call one of **monitorEnter**, **monitorExit**, **waitForPeriod**, **wait** and **waitTimed** or when an interrupt arrives, as is the case where an alarm fires. The right stack is passed control when the application calls **notify** or **notifyAll**. After one of the scheduler stacks is finished, it passes control to an underlying *R-T OS* which provides low-level kernel mechanisms. Such mechanisms include the ability to create, suspend and resume an application thread, as well as the ability to create, set and disable alarms for future events (*e.g.*, arrival of next period or the timeout of a **waitTimed**). In the following sub-sections we will examine each of the scheduler stacks in more detail.

2.1 Controlling the Currently Executing Thread

As aforementioned, the left stack takes control of the system when the application calls one of **monitorEnter**, **monitorExit**, **waitForPeriod**, **wait** and **waitTimed**, or when an alarm expires. In these cases, it must choose one of the available threads as the thread which should next be run on the processor. It does this in three steps, each one performed at a different layer. In the first layer, referred to as the *Ready-Exec* scheduler, it calculates the set of threads $\mathcal{R}_{\text{exec}}$ which are ready to execute without directly blocking due to mutual exclusion. That is, it examines whether an application thread will try to enter a monitor which is already occupied by another thread, if it is chosen as the next thread to execute. Having calculated the set $\mathcal{R}_{\text{exec}}$, this layer passes it to the next layer.

The *Safe-Exec* scheduler layer is responsible for calculating the subset $\mathcal{S}_{\text{exec}}$ of $\mathcal{R}_{\text{exec}}$, which consists of those threads that can *safely* execute. Safety here refers both to deadlock freedom (*i.e.*, entering a monitor would not cause a deadlock later on), as well as, to meeting the timing constraints of the different threads (*i.e.*, choosing a thread for execution will not delay another thread enough to make it miss its deadline).

The *Safe-Exec* layer passes the set $\mathcal{S}_{\text{exec}}$ to the third layer *QoS-Exec*, which calculates the subset $\mathcal{Q}_{\text{exec}}$ of $\mathcal{S}_{\text{exec}}$. The $\mathcal{Q}_{\text{exec}}$ set effectively consists of the *safe* threads (since it is a subset of $\mathcal{S}_{\text{exec}}$), which, in addition, respect the QoS requirements.

Normal scheduling policies usually stop at this point. In our case, through the right stack of the scheduler, we are also able to control the communication aspect of the system.

2.2 Controlling the Notified Thread

The reader will have noticed that the left stack of the scheduler, deciding which thread will execute next, does not get called when the application does a notification (either a **notify** or a **notifyAll**). The reason for this is that the threads which will be notified (if any) cannot in reality ever be selected for execution. This is because they will immediately try to re-enter the monitor after being notified and thus get blocked by the notifier (which is already in the monitor). In other words, notified threads will be excluded by the first layer *Ready-Exec*. Nevertheless, when a thread notifies a condition variable, then we can control which among the threads waiting for the notification should receive the event. Indeed, languages (like JAVA [4]) which provide the monitor construct, or thread libraries (like POSIX threads [9]) which offer it to languages which do not provide it, leave this point *unspecified*, allowing each implementation to choose the thread to be notified as it convenes it the best. The apparent cases where we cannot effect any control on the system are three. Firstly, the case where no thread waits on the condition variable being notified. Secondly, the case where only one thread waits on the condition variable. Finally, the case where the notifying thread does a **notifyAll**, in which

case we are obliged to notify all the threads waiting on the condition variable. In these cases, the right stack of the scheduler does not make any control decision, but simply *changes the PC* of all threads waiting on the current notification, that is, the threads belonging to the set $\mathcal{R}_{\mathrm{notif}}$, to mark them as notified.

When, however, we have a simple **notify** and there are more than one threads waiting on the condition variable, then the top layer *Ready-Notif* calculates the set $\mathcal{R}_{\mathrm{notif}}$ of threads waiting on the condition variable being notified. Then, it passes this set to the middle layer *Safe-Notif*, which calculates the subset $\mathcal{S}_{\mathrm{notif}}$ of $\mathcal{R}_{\mathrm{notif}}$. This subset consists of those threads which, if notified, will not cause the system to enter into a deadlock state or cause some other thread to miss its deadline (*i.e.*, they are safe). Finally, the *Safe-Notif* layer passes the $\mathcal{S}_{\mathrm{notif}}$ set to the bottom *QoS-Notif* layer, which calculates the subset $\mathcal{Q}_{\mathrm{notif}}$ of $\mathcal{S}_{\mathrm{notif}}$, consisting of the threads which we can safely notify and also respect the QoS properties of the application. The *QoS-Notif* layer is also responsible for choosing one of the threads in $\mathcal{Q}_{\mathrm{notif}}$ as the recipient of the current notification and marking it as notified by changing its *PC*. This is a way to *simulate* the behaviour of communication by means of condition variables through the suspension (at a **wait**, **waitTimed**) and the eventual resumption of threads marked as notified (once the respective **monitorExit** has occurred), without really using the condition variable and mutex mechanisms of the underlying *R-T OS*.

2.3 QoS Policies

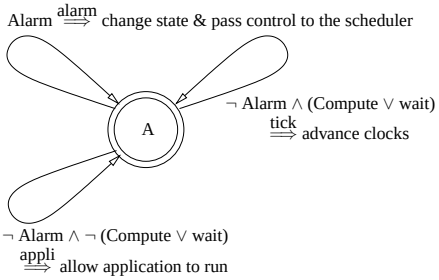
By incorporating a layer for QoS in the scheduler, we offer an *extendible* mechanism for providing additional properties to the system. Additionally, we allow the QoS policies to use the same information which is available to the scheduler to control the application. That is, the QoS layer has access to the *program counters (PC)* of the threads, the *currently executing thread (*T_{Exec}*)*, as well as the value of the *system clock (*C_{System}*)*. In addition to this information, it can also keep statistics of the use of the various system resources, if so desired by the system designer.

Therefore, the complexity of the QoS layer is controlled by the application designer. In choosing a QoS policy (or policies, since these are composable) the designer can balance between the execution time and extra memory space needed by the policy and the gains to the overall system quality the particular policy can offer. A QoS policy is, for example, the *local minimisation of context switches* in order to speed-up the execution by eliminating unneeded switches. This policy can be implemented quite easily, since all one needs to examine is whether the currently executing thread T_{Exec} is in the set $\mathcal{S}_{\mathrm{exec}}$ of threads which are safe to execute next. If this is the case, then we can let it continue its execution, by setting the set $\mathcal{Q}_{\mathrm{exec}}$ equal to the singleton $\{T_{\mathrm{Exec}}\}$. This particular policy has another advantage: by decreasing the number of context switches, we also decrease the cache misses of the application, since now there are fewer points in the execution where the threads compete for the cache, potentially flushing each other's data out of it. This can help *decrease the energy consumption* of the system, since a cache miss can lead to two

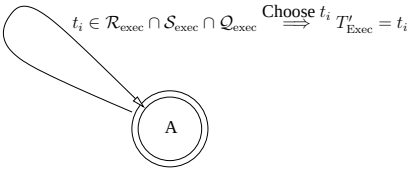
main memory accesses, which are known to be quite demanding with respect to energy [10]. In fact, since a cache reads and flushes one *cache line* at a time (*i.e.*, multiple consecutive memory addresses) the benefits can be even greater, both with respect to energy consumption and execution speed.

3 System Model

The model of the system we construct is the parallel composition of an automaton which is responsible for advancing time and firing the alarms, one automaton for each of the application threads and two more automata, for the *QoS-Exec* and the *QoS-Notif* scheduler layers respectively. The automaton of time and the automata of the application threads perform a finite number of actions and then block, letting the scheduler automata respond. The actions of the time and application automata being *uncontrollable*, the only *controllable* actions are those of the two scheduler automata. Thus, our model can be seen as a two player game with the scheduler automata on one side (*i.e.*, the controller) and the time and application thread automata on the other (*i.e.*, the plant, see [18]). In this game, the automata related to the application simulate the locking and unlocking of resources, as well as, the waiting and notification on condition variables. The computations performed by the application threads is simulated just by their minimum and maximum execution times, *i.e.*, the automata block on the transition simulating the particular computation until enough time has passed to perform the computation in reality. Each statement s of an application thread (where s is one of **monitorEnter**,**monitorExit**,**wait**,**waitTimed**,**waitForPeriod**,a conditional or a computation) is modelled by a separate automaton state and a transition from it to the next statement position ($@s'$) which is taken when the statement s can be executed.The only exception to this rule is the case of the **wait** and **waitTimed** statements. These statements are effectively modelled by two states; the first one models the release of the mutex associated with the condition variable on which we wait and the second one models the attempt of the thread to re-acquire the mutex, once it has been notified.The advancement of time is the responsibility of a single automaton (see Figure 2-a) which, in addition, enables transitions in the application automata which correspond to timeouts, such as in the case of a **waitTimed** or a **waitForPeriod**. This automaton is also responsible for advancing the local thread clocks, that is, the clocks which model the time spent by threads in computations (and in waiting when doing a **waitTimed**). These clocks are set to zero at the beginning of a computation by a thread and are incremented alongside with the global time, until the duration of the computation is over (or the timeout of the **waitTimed** has expired). Finally, the two automata for the *QoS-Exec* and *QoS-Notif* scheduler layers are passed control as described in sub-sections 2.1 and 2.2 and decide which of the application automata should be allowed to execute next or be notified of an event. These automata are comprised of a single state and $n + 1$ transitions, where n is the number of application threads (the additional transition



2-a: Time automaton



2-b: *QoS-Exec* Scheduler automaton

Figure 2: Time & Scheduler automata

is for the *idle* thread). A transition of these automata selects one of the threads for execution (resp. for notification). It is guarded by a predicate which asserts that the corresponding thread belongs to the $\mathcal{Q}_{\text{exec}}$ (resp. $\mathcal{Q}_{\text{notif}}$) set (see Figure 2-b). The transition choosing the idle thread asserts that the rest of the threads are not safe to execute (resp. no thread waits to be notified on the current event).

To summarise, the state in our model comprises of:*(i)* a program counter (PC_i) for each of the application threads, *(ii)* a local clock (C_i) for each thread which is used for their computations and the timeouts if they execute a **waitTimed**, *(iii)* a global clock (C_{System_i}) for modelling the periods of each periodic thread, *(iv)* a variable (T_{Exec}) holding the currently executing thread, *(v)* two boolean variables (*Exec_Sched_Enabled* & *Notif_Sched_Enabled*) for controlling whether it is one of the scheduler automata (and which of them), or the time (when they are both true) or the time and application automata (when they are both false) which should execute, and *(vi)* the *boolean* variables of the application threads used in conditionals which are *significant* [2] with respect to the use of resources and communication of events. Our model goes through three different modes of execution, as shown in Figure 3. In the "Time Only" mode (where *Exec_Sched_Enabled* $=$

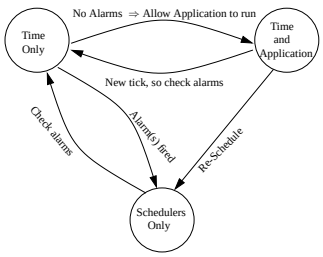


Figure 3: Model execution modes

[2] We perform *slicing* of the original code to identify these variables.

Notif_Sched_Enabled = **true**) the time automaton is the sole automaton enabled in the system and it can fire one or more alarms, if any is enabled. If an alarm is fired then the execution mode changes to "Schedulers Only" (where *Exec_Sched_Enabled* = ¬*Notif_Sched_Enabled*), so that our scheduler can treat the alarm. If there is no alarm to be fired then the execution mode changes to "Time and Application" (where *Exec_Sched_Enabled* = *Notif_Sched_Enabled* = **false**). At this mode, both the time automaton and the automata of the application are enabled. If the time automaton gets to execute first, then a tick (*i.e.*, a time step) is performed and we pass back to the "Time Only" mode, so as to check if an alarm is now enabled. If it is one of the application automata which gets to execute first, then it executes until it needs to perform an action which causes re-scheduling, in which case it passes control to the schedulers (*i.e.*, the mode now becomes "Schedulers Only"). If the application automaton needs to execute a time guarded action (*i.e.*, a computation), then it blocks, allowing time to advance.

As an example, let us consider the model shown in Figure 4. Here, the application consists of three threads, one of which is a *periodic* one (the `User`) and two *aperiodic* ones (the `Writer` and the `Refresher`). One should note that the `Writer` and the `Refresher` are continually enabled aperiodic threads and do not have any deadlines directly associated with them.

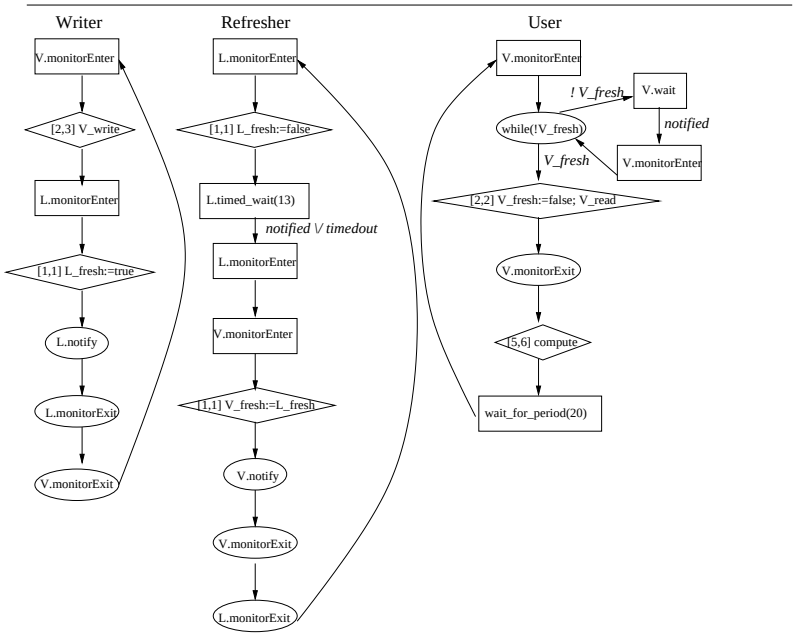


Figure 4: Application automata
(Ovals depict Atomic states, diamonds depict Preemptable states and boxes depict Blocking states. To increase readability we have omitted the clock variables, showing only the interval of the duration for each computation)

4 Scheduler Synthesis

In order to synthesise the *Safe-Exec* and *Safe-Notif* scheduler layers, we first construct the whole space of the states the system can reach and, thus, identify the deadlock states. These are the states where the application threads are deadlocked, or the states where some thread has missed its deadline (since in that case we block the system explicitly). The existence of these states indicates that the predicate we are currently using to describe the set *Safe-Exec* (resp. *Safe-Notif*) needs to be constrained even further. This predicate starts with the value of **true**, thus accepting initially all threads in the set *Ready-Exec* (resp. *Ready-Notif*) as safe. Having obtained the deadlocked states, we do a backwards traversal of the whole state space starting from the deadlocked states, until we reach a state which corresponds to a choice of one of the scheduler automata. There, we identify the choice $T_{\mathrm{Exec}} = t_i$ which allowed the path leading to a deadlock state(s) and create a new constraint for the layer *Safe-Exec* (resp. *Safe-Notif*). This constraint is constructed by changing the set $\mathcal{S}_{\mathrm{exec}}$ (resp. $\mathcal{S}_{\mathrm{notif}}$) to be:

$$\mathcal{S}'_{\mathrm{exec}}(\overrightarrow{state}) = \mathcal{S}_{\mathrm{exec}}(\overrightarrow{state}) \setminus \{t_i\} \tag{1}$$

If at some point we find that $\mathcal{S}'_{\mathrm{exec}}(\overrightarrow{state})$ is equal to the empty set, then we add the current state to the set of deadlocked states and continue the synthesis procedure.

Constructing the whole state space and then performing a backwards traversal is straightforward to implement, since it keeps the state space construction and the synthesis algorithm separated. Its disadvantage is that we must always construct the entire state space. Instead of this, we could construct the state space and perform a forwards traversal at the same time, thus mixing the state-space construction and synthesis algorithms together. This is less obvious to implement correctly but offers the opportunity to perform synthesis *on the fly*, thus avoiding to construct the whole state space most of the times.

4.1 State Space Reduction & Application Analysis

Even though the basic idea of synthesising the *Safe-Exec* and *Safe-Notif* scheduler layers is simple, it is evident that in practice it suffers from the state explosion problem. In addition, obtaining a long list of constraints that the scheduler should always impose, does not help a designer to better understand the thread interdependencies and the overall behaviour of the system under design. Therefore, it is imperative that we use techniques to minimise the size of the state space and at the same time provide the designer with easy to use information concerning the behaviour of the system. Altisen *et al.* [1] proposed to constrain the system with a high level policy (*i.e.*, FIFO scheduling, EDF scheduling, *etc.*) the idea being that this will constrain enough the system to allow us to construct the entire (constrained) state space of it. However, this method can sometimes over-constrain the system and remove all possible paths which would allow us to avoid the unsafe

states. Another result of applying these policies early on is that we now have a smaller degree of freedom to apply the QoS policies. Finally, it is not always clear how one can apply policies such as EDF, RMA, *etc.* when the application consists of heterogeneous threads. Our method consists of synthesising schedulers for successively more detailed models, adding thus complexity to a model only when we have already calculated how we can constrain the more abstract one. This method also helps in aiding the designer better understand the system, since the constraints we construct at each step can be directly linked to a particular system property. The scheduler synthesis is performed in two major steps under our method. In the first step, we examine the *untimed* model of the system and search for constraints which can guarantee the *absence of deadlocks*. In the second step, we re-introduce time into the model and after constraining the system with the constraints needed for avoiding deadlocks, we search for those constraints which can guarantee that *all threads meet their deadlines*.

4.2 Deadlock Avoidance

Examining the untimed model of the system first, has the disadvantage that some of the deadlocks we identify are not possible in reality, due to the existing timing relations. This means that the set of behaviours we are searching is in fact larger than the "real" untimed one. However, there are significant advantages in treating the untimed model first. First, adding time to a model significantly increases its size and thus renders the analysis and synthesis a lot more difficult. On the other hand, searching for deadlocks in the untimed model allows us to examine a much smaller search space and thus allows us to attack larger systems. For the example application shown in Figure 4, treating the untimed version of the model means that we have to examine a model reduced by 97% to discover the 8 constraints which can help us avoid all the 10 deadlocks caused by the use of shared resources. More importantly, once the deadlocks have been found and removed from the untimed model, we no longer have to worry about deadlocks, if we decide to change the underlying platform or optimise the algorithms used in the computations. In other words, finding and removing *all* deadlocks in the untimed model means that the application is *logically* correct and allows a designer to experiment with different underlying platforms and algorithms for implementing the application computations. Both these actions can change the timing properties of the computations performed by the threads and thus unmask deadlocks which were previously impossible. For this reason we refer to these deadlocks as *dormant* deadlocks. Whether the system meets or not its *R-T* requirements depends heavily on the particular platform and computation algorithms used. Thus, we establish a strong invariant of the application, which is independent of the underlying platform and specific computational algorithms used. In addition, the dormant deadlocks produce constraints which do not reduce our choices when trying to resolve the problem of meeting the different thread deadlines. This is because these constraints are active only in those states of the system which are currently impossible to reach due to the particular tim-

ing relations of the application. Finally and most importantly, being able to guard the application against all deadlocks, possible and dormant ones, means that the application can survive the bad case where a deadlock has been *wrongly characterised* as dormant due to a *wrong estimation* of the various timing relations of the threads. Since it is difficult to obtain exact *upper* and *lower* timing measurements of threads, this is a very important point that should not be taken lightly.

4.3 Guaranteeing Deadlines

Having found all the potential deadlocks in the system, we add the synthesised $\mathcal{S}_{\text{exec}}$ and $\mathcal{S}_{\text{notif}}$ scheduler sets obtained so far to the timed version of the initial model, in order to search for the *timeliness* constraints, which can guarantee that all threads will meet their deadlines. In order to make the problem more tractable, we apply a safety-preserving abstraction to the model, which reduces the number of overall states.

4.3.1 Speeding Up Time

Not all time instances are visible when the scheduler automata take control during the execution of the system. That is, there are certain states of a system where the only event allowed is the advancement of time. For example, it may well be the case that some threads are blocked and another thread is waiting for its new period. In such cases, we do not really gain anything by explicitly constructing the complete sequence of all the different time steps. Instead, we can jump directly to the last state of this sequence, where the time has advanced enough to allow some other event to occur. In other words, what we wish to achieve is to speed up the advancement of time, when the time automaton is the only one which can execute, all the way up to the time instance where another automaton can execute. In this way, when we consider the timed model we only have to pay the cost of coding the exact time instances at which our system can indeed perform an interesting action. Indeed, for the application shown in Figure 4 we obtain a 74% reduction of the state-space.

This state space reduction can be effectively obtained through the *branching bisimulation equivalence (bbe) reduction* [16], which eliminates "unobservable" actions (in our case the Tick action) but only when doing so preserves the branching structure of processes. The preservation of the branching structure of the application is crucial for us, since the synthesis of the scheduler depends on it for calculating the states where a controllable action can help avoid taking a path which leads to an undesired state. Given a set of equivalent states under the *bbe* reduction, we elect as a representative of this set the state which has the maximum global clock value. In other words, in the *bbe* reduced system, our scheduler takes its decisions at the latest moment possible.

Having reduced the state space of the timed model, we continue the synthesis of the scheduler for the timing constraints, breaking again the process into two

steps.

4.3.2 Non Preemptable Threads

In the first step, we consider that the application threads cannot be *preempted* while they are computing. The non-preemption hypothesis reduces the state space we have to consider, since it removes all the cases where the execution of a thread is suspended by an interrupt (*e.g.*, for starting a new period of some other thread). In the application of Figure 4 the reduction obtained by not allowing preemption is 40% when applied on the initial timed model and 80% when applied to the *bbe* reduced timed model. Once we can indeed safely schedule the system under the hypothesis that threads are never preempted, then we can use the constraints obtained during this step to *reduce even further* the state space that we have to construct and analyse when we do allow threads to be preempted. Indeed, for the application of Figure 4, we can reduce the state space by an additional 10% with the 30 constraints we construct during this step.

The non-preemption of threads is easily added to our models *through the use of a QoS policy*, which is another indication of the modularity of our architecture and of the usefulness of the layer dealing with the QoS requirements. This policy effectively forbids the schedulers from choosing a thread for execution, when another thread is already in a state where it is performing a computation. That is, the non-preemption QoS policy is expressed as:

$$\mathcal{Q}_{\text{exec}}(\overrightarrow{state}) = \{t \,.\, t \in \mathcal{S}_{\text{exec}}(\overrightarrow{state}) \wedge \neg\exists t' \neq t \,.\, \text{computes}(t')\} \qquad (2)$$

However, we cannot safely schedule all systems when we do not allow threads to be preempted. Indeed, it is easy to see from the non-preemption QoS policy, that sometimes the thread t' which is executing may not be in the $\mathcal{S}_{\text{exec}}(\overrightarrow{state})$ safe set. This means that for these systems we will not obtain any scheduling constraints and, therefore, will be obliged to examine the large, unconstrained state space of the timed model. Nevertheless, the fact that we have identified our inability to safely schedule the system when threads cannot be preempted is of great importance to the designer. It shows that the system may be *overloaded* and thus *under difficulty to meet the deadlines* of the threads if preemption is not allowed and it also helps the designer *identify the exact set of problematic threads*. In this case, designers will probably want to consider other computation algorithms and/or underlying hardware platforms, in order to have a system which is not overloaded and, therefore, not so *sensitive* to the execution times of the threads.

4.3.3 Allowing Preemption

Having performed the scheduler synthesis for the case where threads are not preemptable, we add the additional constraints we synthesised (if any) to the model and perform the scheduler synthesis once more, this time allowing threads to be preempted. This is the final step of the scheduler synthesis, which provides us

with the whole set of constraints that we must impose on the application in order to guarantee that it will be deadlock free and that it will meet all the deadlines of the threads. For the application of Figure 4, this last synthesis step produces an additional 18 constraints and thus we can safely schedule this application with a total of 56 constraints, avoiding both deadlocks and missed deadlines. These 56 constraints are all part of the *Safe-Exec* layer, since in this application there is always at most one thread waiting to be notified on a particular condition variable and thus we cannot control the communication aspect of the application. This safe model has a state space which is 96% smaller than the original, unsafe one.

4.4 Removing Clocks

Having synthesised a safe scheduler for an application does not necessarily mean that we can implement it easily on a usual *R-T OS* though. The difficulty of implementing it as is, arises from the fact that the constraints we produce during the synthesis use the state of the system to decide what are the safe choices at each point during the execution and, therefore, also make reference to the values of the local clocks of the threads. However, these clocks do not really exist but were only introduced as a way to model the computations of the threads. Introducing them in a real *R-T OS* means that we will have to add for each thread an additional timer object and reset and activate (resp. reset and deactivate) the timer before (resp. after) each computation and read its value when making a scheduling decision. Even though we do allow preemptable threads in general, we do not expect the additional degree of control that these clocks would offer us to be great enough to justify the complexity of using the additional timers. Therefore, we *remove* the clock evaluations from the constraints to obtain a clock-free scheduler, which only examines the *PC*s of the threads. This will make the scheduler itself faster to execute, since in order to make a scheduling decision it now only needs to examine the n values of the different *PC*s and not the $2n + 1$ values of the *PC*s, the local thread clocks and the global clock. On the other hand, removing the clocks from the constraints can introduce states where the scheduler will take the wrong decision and cause a thread to miss its deadline. These states are those where a scheduler gets called at the same configuration of thread *PC*s but at different time instances. Since the time instance (and therefore the clock values) are different, the safe set $\mathcal{S}_{\text{exec}}$ of these states can be different themselves as well. When we decide to not observe the clock values while scheduling, we are effectively unable to differentiate among these different sets and all these states become *equivalent*, as far as our scheduler is concerned. Therefore, if we wish our scheduler to always make a decision which is *safe*, then the $\mathcal{S}_{\text{exec}}$ set of this *equivalence class* of states should be the *intersection* of the $\mathcal{S}_{\text{exec}}$ sets of the states which belong to the same equivalence class.

$$\mathcal{S}_{\text{exec}}(\overrightarrow{class_j}) = \bigcap_{state_i \in class_j} \mathcal{S}_{\text{exec}}(\overrightarrow{state_i}) \tag{3}$$

Sometimes, the $\mathcal{S}_{\text{exec}}(\overrightarrow{class_j})$ set will be empty, if the scheduler decisions at

the members of this equivalence class were conflicting. When encountering such an equivalence class whose $\mathcal{S}_{\text{exec}}$ set is the empty set, we need to add its members to the set of deadlocked states and continue the synthesis algorithm, until we find a set of constraints which helps us to avoid the whole class.

In some cases it may be impossible to safely schedule the application without taking into account the values of the clocks. However, we believe that this will not be the case for the majority of the real applications. The example application of Figure 4 is indeed an application which can be scheduled without observing the clock values and so the scheduler we synthesise for it observes only the values of the *PC*s.

4.5 Other QoS Policies

Once we have synthesised a safe scheduler, we can compose it with other QoS policies, as aforementioned in section 2. These policies are used to choose among the safe threads those which better realise the QoS requirements of the system. However, it can always be the case that some of these policies can cause the application to miss a deadline, by choosing no thread for execution (*i.e.*, setting the $\mathcal{Q}_{\text{exec}}$ set to be the empty set). For this reason we must verify them and change them if they can indeed cause the application to miss a deadline. Since the verification is performed on the *safely schedulable* application, the size of the state space we must explore is quite small. For the example application of Figure 4, we verified two additional QoS policies. The first is a fixed priority QoS policy, where `User` $>$ `Refresher` $>$ `Writer` when they are *safe* and it is effectively verified in a model 98.4% smaller than the original timed one. The second is a QoS which (locally) minimises the number of context switches and it is verified on a model which is slightly even smaller (a 98.8% reduction). In Appendix A we present all the data we collected for the application of Figure 4.

5 Scheduler Implementation

In this section we present the implementation of our scheduler over an *OS* for embedded systems. We first present the approach we have taken for implementing the election of the next thread to execute and then for electing the thread that should receive a particular notification. Our current implementation does not make use of priorities, mutexes or condition variables of the underlying system. It rather uses *thread suspension and resumption* to simulate these mechanisms, which allows us to provide an implementation of our approach even on operating systems which have a very limited number of priorities or have no priorities whatsoever.

5.1 Actions Causing Rescheduling

First, we consider the actions which activate our scheduler in order to elect a new thread to execute. As we have aforementioned, these are the actions **monitorEnter**,

monitorExit, **waitForPeriod**, **wait** and **waitTimed**, as well as the expiration of a timeout. All these actions are implemented similarly. First, the *OS* scheduler is locked so as to avoid interrupt handlers from changing the system state. Then our scheduler examines the current state of the system, *i.e.*, the *PC*s of all the threads, and decides which should be the next thread to execute, by using the synthesised sets $\mathcal{R}_{\text{exec}}$ & $\mathcal{S}_{\text{exec}}$ and the user-provided set $\mathcal{Q}_{\text{exec}}$. Finally, our scheduler suspends all threads, resumes the one it has chosen for execution and returns after having unlocked the *OS* scheduler, thus re-allowing interrupts to occur.

The only case which is treated differently, is the case where an interrupt arrives to signal a timeout (either for a **waitTimed** or a **waitForPeriod**). In this case, the associated interrupt handler changes the *PC* of the respective thread to mark it as no longer waiting and, then, suspends the currently executing thread (if any) and resumes the thread that was waiting. Once this thread starts to execute, it calls our scheduler in its turn, to assure that it can safely continue. If this is the case, our scheduler will allow it to execute, otherwise it will suspend it and resume another thread.

It is easy to see that if more than one interrupts arrive at the same time, our scheduler will be called consecutively more than once. However, there can be no more than 3 consecutive calls to our scheduler ever (the proof of this is provided in Appendix B).

5.2 Actions Not Causing Rescheduling

Finally, we examine the actions which do not cause our scheduler to elect a new thread for execution, that is, the actions **notify** and **notifyAll**. As we have already seen in section 2, unlike the previously presented actions which made use of the left stack of our scheduler, these actions make use of its right stack, which deals with the communication aspects of the system. Indeed, when a **notify** occurs, the scheduler is called and it checks whether there are any threads waiting on the event notified. If so, it selects one of them *using the* $\mathcal{Q}_{\text{notif}}$ *set*, marks it as notified and gives back the execution to the thread which did the **notify**.

5.3 `eCos` Implementation

Currently, we have implemented our architecture on top of `eCos` [12], an open-source *OS* for embedded systems. Lacking a real embedded platform, we used *synthetic-Linux* as the execution platform of `eCos`, which means that `eCos` and its application are running as a single Linux process. The advantage of this is that one has an easy to use and cheap development platform, with the disadvantage that measurements obtained are not the same that would have been obtained if `eCos` was running on its own. Nevertheless, it allows us to experiment with different methods and arrive at some initial conclusions. For example, our experiments showed that despite executing a *R-T* application on such a non-*R-T* environment, it did indeed honour its deadlines. Another point which interested us was the cost

of executing the scheduler stack. The tests we have performed showed that, even in such a rather hostile environment, the cost of evaluating the different scheduler sets when running `eCos` over Linux is of the order of 0.66 micro-seconds per state on the average, when executed on a 330 MHz Pentium II machine [3]. Given the fact that our prototype implementation is not particularly optimised for speed, this is a rather small execution cost.

6 Conclusions

We have presented a new methodology for building application-driven schedulers based on the controller synthesis paradigm [18] and a prototype implementation of our scheduler using an open-source *OS* for embedded systems.

Our work is based on [1, 2]. Controller synthesis for timed automata has also been considered in [5], where the problem is reduced to the untimed framework of [18] using the *region graph* construction which results in state space explosion. [17] treats the problem in a more general setting of linear hybrid automata and presents a semi-decision procedure (the problem is in general undecidable for this class of systems). The approach proposed in [7] is also similar to ours, in the sense that it uses an automata-based formalism (after translation from ACSR), but it relies on a different algorithm based on the notion of weak bisimulation and does not propose a particular scheduler architecture or implementation. A scheduler synthesis tool has also been described in [8]. It differs from ours in two major aspects: *(i)* it computes static cyclic schedules by sequencing events in a fixed time frame, whereas our algorithm produces dynamic (and not necessarily cyclic) schedules for an unbounded time frame, and *(ii)* it is restricted to deterministic execution times, while we can handle nondeterministic ones.

In [6], the authors consider applications comprising of heterogeneous thread types, just as we do herein, but do not consider the problem of thread interdependencies due to the sharing of non-preemptable resources.

The advantages of our synthesis method is that we can handle larger models than if we would have tried to attack the original timed version of the model at once. In addition, following our method a designer is better able to understand the behaviour of a system, since we successively drive him through: *(i)* the states which cause a deadlock later on, *(ii)* the states where a system is overloaded (and, therefore, he needs to allow preemption of threads), and finally, *(iii)* the states where the scheduler also needs to observe the values of the local clocks measuring the duration of each computation of the threads. Finally, an advantage of our method is that it does not consider any particular model for threads and therefore can be applied to applications comprising of any mix of periodic, aperiodic, *etc.*

[3] Minimum/maximum overhead was 0.00μs/ 5.00μs, average deviance was 0.45μs, population variance was 0.23 and 65% of the measurements belonged in the interval $\mathrm{average} \pm \mathrm{average\ deviance}$ (*i.e.*, 0.66μs $\pm$ 0.45μs) while 33% of them belonged in the interval $\mathrm{minimum} \pm \mathrm{average\ deviance}$ (*i.e.*, 0.00μs $\pm$ 0.45μs).

threads, which share non-preemptable system resources and communicate through condition variables.

A disadvantage of our method is that we must build the entire state space before we can synthesise a scheduler for an application. We plan to address this problem in future versions of our tools, which will perform the synthesis in an on-the-fly manner while constructing the state-space, as for example was done in [15]. We also plan to study ways to perform the synthesis symbolically, without explicitly constructing the state space graph.

In this article we focused on models instead of a particular programming language. Such models can be extracted from programs using static analysis techniques. We indeed plan to develop such a model extraction for Java, so as to be able to schedule real-time Java programs.

Acknowledgements

This work has been partially funded by the French RNTL project Expresso.

References

[1] K. Altisen, G. Gößler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems*, 23(1):55–84, July 2002.

[2] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*, pages 1–20, Berlin, 1995. Springer-Verlag.

[3] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Trans. Computers*, 51(3):289–302, Mar. 2002.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, USA, 1996.

[5] G. Hoffmann and H. Wong Toi. The input-output control of real-time discrete event systems. In *30th IEEE CDC*, 1991.

[6] D. Isović and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *21st IEEE Real-Time Systems Symposium*, Walt Disney World, Orlando, Florida, USA, Nov. 2000.

[7] H. Kwak, I. Lee, A. Philippou, J. Choi, and O. Sokolsky. Symbolic schedulability analysis of real-time systems. In *IEEE RTSS'98*, Madrid, Spain, Dec. 1998.

[8] A. K. Mok, D. C. Tsou, and R. C. M. Rooij. The MSP.RTL real-time scheduler synthesis tool. In *RTSS'96*, Washington, D.C., USA, Dec. 1996. IEEE Computer Society Press.

[9] The Open Group. *The Single UNIX Specification, Version 2: Threads*, 1997. Available online at `http://www.unix-systems.org/single_unix_specification_v2/xsh/threads.html`.

[10] F. Parain, M. Banâtre, G. Cabillic, T. Higuera, V. Issarny, and J.-P. Lesot. Techniques de réduction de la consommation dans les systèmes embarqués temps-réel. Technical Report 1332, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, May 2000. (In French).

[11] Real-Time Java Working Group. Real-time core extensions, revision 1.0.14. Technical report, J Consortium, Sept. 2000. Available at `http://www.j-consortium.org/rtjwg/rtce.1.0.14.pdf`.

[12] RedHat Corporation. eCos reference manual. Available at `http://sources.redhat.com/ecos/docs-latest/pdf/ecos-ref.pdf`, Sept. 2000.

[13] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, C-39(9):1175–1185, Sept. 1990.

[14] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, N.J., 1st edition, 1992.

[15] S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99, World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes in Computer Science*, pages 233–252, Toulouse, France, Sept. 1999. Springer-Verlag.

[16] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM (JACM)*, 43(3):555–600, May 1996.

[17] H. Wong Toi. The synthesis of controllers for linear hybrid automata. In *36th IEEE CDC*, pages 4607–4612, 1997.

[18] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal of Control and Optimization*, 25(3):637–659, May 1987.

A Experimental Results

In Table 1 we present the sizes of the different models we used, the reductions we achieved with our proposed optimisations, the different synthesis steps for this case study and finally, some QoS policies with which we experimented. In the first section of the table, titled *"Model Abstractions & Optimisations"*, we present how the different optimisations we have introduced decrease the state-space of the problem. The attribute "No Clocks" used therein refers to the fact that the synthesised scheduler at the particular step does not observe the values of the clocks, either the local clocks or the global ones. The second section of Table 1, titled *"Synthesis Steps"* shows how the model sizes change during the scheduler synthesis process. Finally, the third section of the Table 1 shows the size of the *safe* model of the application, when we apply to it different QoS policies. The first QoS policy is one where *safe* threads have been prioritised, in the order `User` $>$ `Refresher` $>$ `Writer` and the second QoS policy locally minimises the number of thread context switches, by selecting the previously executing thread if it is still safe to do so.

The state-space graph construction and the *bbe* reduction of the graph in these experiments have been carried out using the CADP tools [4].

Table 1: Comparing the size of the different models

Model kind†	**States**	**Red.%**	**Transitions**	**Red.%**	**Deadlocks**‡	**Red.%**	**Constraints**††
Model Abstractions & Optimisations							
T ***original* (i.e., *Preemption*)**	45470	0.00%	48786	0.00%	367	0.00%	—
U	1352	97.03%	1645	96.63%	10	97.28%	—
T ***No Preemption***	27266	40.04%	29118	40.31%	134	63.49%	—
T ***Preemption, bbe reduction***	11437	74.85%	13648	72.02%	1	99.73%	—
T ***No Preemption, bbe reduction***	8648	80.98%	10038	79.42%	1	99.73%	—
Synthesis Steps							
U	1352	97.03%	1645	96.63%	10	97.28%	0 (8)
U, No Deadlocks	1200	97.36%	1451	97.03%	0	100.00%	8 (0)
T ***No Preemption, bbe reduction***, No Deadlocks	8642	80.99%	10027	79.45%	1	99.73%	8 (30)
T ***No Preemption, bbe reduction***, Safe	1542	96.61%	1668	96.58%	0	100.00%	30+8=38 (0)
T ***Preemption, bbe reduction***, No Clocks	4640	89.80%	5532	88.66%	1	99.73%	30+8=38 (18)
T ***Preemption, bbe reduction***, No Clocks, Safe	1593	96.50%	1740	96.43%	0	100.00%	18+30+8=56 (0)
QoS Policies *(reduced with bbe)*							
T Safe, Fixed Priorities‡‡	728	98.40%	750	98.46%	0	100.00%	56 (—)
T Safe, Locally Min. Context Switches	549	98.79%	573	98.83%	0	100.00%	56 (—)

† *T* indicates a *Timed* and *U* an *Untimed* model.

‡ Missed deadlines manifest themselves as deadlocks in our models.

†† This column reports the number of constraints *applied* to the model, that is the number of constraints synthesised in the *previous* step(s), as well as, the number of *new* constraints synthesised at each step (inside parenthesis).

‡‡ In this case, User has a higher priority than Refresher and Refresher has a higher priority than Writer.

B Consecutive Calls to the Scheduler

The following theorem presents an upper bound on the number of consecutive calls to the scheduler which can happen.

[4] `http://www.inrialpes.fr/vasy/cadp/`

Theorem B.1 *At each point of execution, we can do at most 3 directly consecutive reschedulings.*

Proof:

The worst case is when n interrupts arrive, where $n > 1$, while we are executing an action which will pass control to the (left stack of the) scheduler, especially one of **monitorEnter** and **monitorExit**.

Then, we might do one rescheduling for the action. After exiting the action and unlocking the scheduler, we will execute the interrupt handlers in some random order, since the interrupt handlers always have a higher priority than the application threads. After having executed all the interrupt handlers, we will execute the last thread among those awaken, and perform a second rescheduling. If this last thread we chose was not the one that should be running at this point, it will give priority to another one. If that one is one of those recently awaken, then this will perform a third rescheduling. However, since the state of the threads has not changed since the second rescheduling, the third rescheduling will have no effect. Thus, this last thread will continue executing.

Therefore, we may execute *at most 3 directly consecutive* reschedulings.