Statistical Model Checking of Processor Systems in Various Interrupt Scenarios

Josef Strnadel(✉)

Faculty of Information Technology, Centre of Excellence IT4Innovations, Brno University of Technology, Bozetechova 2, 612 66 Brno, Czech Republic
strnadel@fit.vutbr.cz
http://www.fit.vutbr.cz/~strnadel

Abstract. Many practical, especially real-time, systems are expected to be predictable under various sources of unpredictability. To cope with the expectation, a system must be modeled and analyzed precisely for various operating conditions. This represents a problem that grows with the dynamics of the system and that must be, typically, solved before the system starts to operate. Due to the general complexity of the problem, this paper focuses just to processor based systems with interruptible executions. Their predictability analysis becomes more difficult especially when interrupts may occur at arbitrary times, suffer from arrival and servicing jitters, are subject to priorities, or may be nested and un/-masked at run-time. Such a behavior of interrupts and executions has stochastic aspects and leads to the explosion of the number of situations to be considered. To cope with such a behavior, we propose a simulation model that relies on a network of stochastic timed automata and involves the above-mentioned behavioral aspects related to interrupts and executions. For a system, modeled by means of the automata, we show that the problem of analyzing its predictability may be efficiently solved by means of the statistical model checking.

Keywords: Cpu · System · Interrupt · Arrival · Servicing Execution · Priority · Jitter · Nesting · Masking · Late arrival Tail chaining · Modeling · Stochastic timed automaton · Predictability Analysis · Statistical model checking

1 Introduction

Predictability plays an important role in terms of applicability of many systems in practice. Especially, this holds for real-time (RT) systems [1]. They must operate both in a functionally correct way and on time, mostly because of their,

This work was supported by The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science – LQ1602. Next, it was supported by the project Advanced parallel and embedded computer systems – FIT-S-17-3994.

typically, cyber-physical nature. The problem with analyzing predictability lies in the following facts. Firstly, predictability of a system must be analyzed for various operating conditions. Typically, such an analysis is expected to be performed at the very beginning of the system's development cycle. Practically, it must be done "much earlier" before the system starts to operate or even before its prototype exists (e.g., for specified conditions, it must be analyzed well in advance if some property may never be violated or, if some property always holds). Secondly, a designer of a predictable system must face many sources of unpredictability such as environmental changes, disturbances and anomalies, effects of aging and degradation, defects and damages, operator errors, lack of energy, aperiodicity of events, digitization effects, or drift of a digital clock.

Scope of This Paper. As the problem is too complex to be fully resolved (let alone in this article), we have decided to limit the scope of this paper just to digital, processor (CPU) based systems detecting events through interrupts. The main advantage of such a detection is that no CPU time is consumed regarding an event until the corresponding interrupt is triggered. At a glance, interrupts may look like random variables, adverse effects of which can be neither simply analyzed nor mitigated. However, more careful investigation of interrupt-related aspects reveals new solutions to both the analysis and mitigation. In particular, the predictability analysis of RT systems typically builds on the values of parameters such as the best-case execution time (BCET), worst-case execution time (WCET) or worst-case response time (WCRT). Their values are utilized later, e.g., to facilitate the process of analyzing schedulability of a set of RT tasks constrained, by their deadlines etc., in the time domain [2].

Basically, two approaches to analyzing the parameters, such as WCET, of a system exist [3]: the static timing analysis (e.g., analysis of source codes of the system's software) and dynamic timing analysis (being typically performed using a real platform, its credible simulator or emulator [4]). Some papers, such as [5,6] present a credible simulation model of a CPU system (including architectural elements such as pipelines or caches) to analyze WCET competitively to a real platform or its emulator.

In this paper, we have decided to use the latter approach, but to abstract from architectural details and substitute them by a stochastic model. To meet our expectation, the model must be expressive enough to allow a credible predictability analysis of CPU based systems for various sources of unpredictability. At the input of our approach, we suppose that both the computational platform and the software it executes are known and analyzed precisely, along with parameters such as BCET and WCET. This allows us to focus on further sources of unpredictability (particularly, on those related to interrupts) and on quantification of further important parameters (especially, from the schedulability analysis viewpoint), such as CPU load, WCRT, stack utilization, interrupt service and latency times or throughput of produced/serviced interrupt requests.

Structure of This Paper. The rest of this paper is organized as follows. Section 2 introduces phenomena being modeled and analyzed in this paper.

Section 3 presents our model, basic interrupt scenarios, queries for checking predictability parameters and, finally, representative results. Section 4 concludes the paper.

2 Preliminary

In this section, we present principles playing an important role in understanding the problem solved in this paper. First of all (Sect. 2.1), we remind concepts behind managing the control flow of a program and the CPU context. Secondly (Sect. 2.2), we summarize basic mechanisms and overheads associated with detecting events through interrupts in CPU systems. Finally (Sect. 2.3), we discuss effects of processing interrupt requests (IRQs) to the execution of a program.

2.1 CPU Context and Control Flow of a Program

At this point, we assume that a reader is familiar with technical aspects behind CPUs and their programming, memory and exception/interrupt models, mechanisms they use to execute a program etc. Such an assumption allows us to skip the phase of repeating well-known facts and emphasize further aspects, important from the viewpoint of this paper. For the sake of simplicity, the following emphasizing text focuses just to single-CPU systems, processing instructions in a single operating mode, often denoted as the "run" mode. Aspects of multi-CPU systems and of executing a program in further modes such as test, debug/tracing, wait or (deep) sleep are not discussed – we leave it for further work.

First of all, we would like to emphasize that a CPU is a highly sequential circuit, the inner state of which, denoted as the "context" as well, is a function of many events (a change of the value of an external signal, the start of an instruction etc.) and partial/sub states, e.g., contents of a program/data memory, pipeline and CPU registers such as the stack pointer (SP), program counter (PC) or condition code register (CCR). Both the events in a CPU and its inner state have a significant impact to the control flow of a program executed by the CPU. Events may be either synchronous (e.g., a function call) or asynchronous (e.g., an exception) to the control flow.

To guarantee the correctness of the control flow at run-time (i.e., its consistency with a programmer's intention), a program must be written so that events, temporarily allowed to change the intended control flow, are managed in a way allowing to return the control flow back. Practically, such a management is typically divided into two phases. The goal of the first phase is to store the CPU context (at least, the content of PC) before the control flow changes due to an event (such as a function call). The purpose of the second phase is to restore the CPU context back, i.e., to resume the control flow changed by the event (e.g., by placing the return address of the called function into PC). Let it be noted there that the CPU context may differ for various events; for example, if an exception occurs, further registers, such as CCR, must be stored with PC.

The CPU context is typically stored onto the stack; some CPUs, however, store their context (fully or partially) into special registers as well. Independently on a particular technical solution and if needed, we often denote such a storing as "stacking" and restoring as "unstacking" in this paper.

2.2 IRQ Processing Aspects

Now, let us focus on the execution viewpoint. Ideally, a CPU is continuously busy by processing instructions of a program it executes (we denote the program as "`main()`" or "`main`", too). Asynchronously to the program control flow, an interrupt request (IRQ) may occur; for an illustration, see Fig. 1a. At the time of its occurrence, however, the IRQ can be masked (if it is maskable), an instruction may subject to processing etc. Until such a situation is over, servicing of the IRQ cannot start, which delays a reaction to the IRQ. After such an obstacle disappears, stacking of the CPU context starts. Typically, it ends by disabling all (maskable) interrupts followed by the arbitration of IRQs that are pending at that moment. The arbitration fetches the vector of the highest-priority pending IRQ and loads it into the CPU's program counter (PC). By the loading, the associated interrupt service routine (ISR), denoted as "(IRQ) handler" too, starts. Typically, an ISR consists of an application specific prologue, the service itself and an application specific epilogue. An ISR completes by unstacking the CPU context, etc.

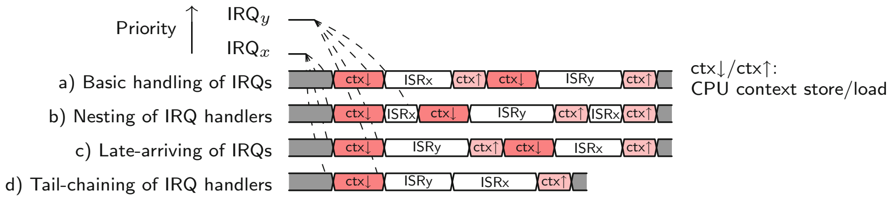

Fig. 1. An illustration to an exception entry, handling and return for ARM® Cortex® and four different interleaving patterns (a, b, c, d) when two IRQs (IRQx, IRQy) are raised consecutively in the order IRQx, IRQy.

If IRQs are unmasked in IRQ handlers, the handlers may nest (embed) in an recurrent way (Fig. 1b). This happens if an IRQ (y), of a sufficiently high priority, pends while an IRQ (x), of lower priority, is being serviced. Then, a handler (ISRx) of the lower-priority IRQ is preempted by a handler (ISRy) of the higher-priority IRQ, where "preempted" means that the execution of the ISRx stops, its CPU context is stacked and then, the execution of the ISRy starts. After the ISRy completes, the CPU context switches back to resume the ISRx, etc. If a higher-priority IRQ (y) occurs after a lower-priority IRQ (x), but during the stacking initiated by the IRQx, the ISRy may start prior to ISRx (Fig. 1c). Such a behavior is possible if the so-called "late-arriving" mechanism is enabled to minimize interrupt response times of higher-priority IRQs.

Further mechanism, denoted as "tail-chaining", may be enabled to minimize the un/stacking overhead regarding IRQ handlers. The mechanism is applicable if a lower-priority IRQ (x) is pending just after a handler of a higher-priority IRQ (y) completes. If it is so, the unstacking after finishing the ISRy and (consequent) stacking before starting the ISRx are not performed. Practically, this mechanism saves the CPU time and minimizes both the interrupt response time and interrupt recovery time of IRQs (Fig. 1d).

2.3 Effects of IRQ Processing to Program Execution

Below, we discuss effects of processing an IRQ to the execution of a program ("`main`"). For simplicity, we suppose just two unmasked IRQs, the first arriving at t_{IRQ_i} and the second arriving at $t_{IRQ_{i+1}}$, handled by ISR$_i$ and ISR$_{i+1}$, resp. (see Fig. 2). Because the un/stacking and handling times are typically fixed for an IRQ$_i$ and the given platform (in total, the overhead is $t_{IRQ_i over}$), the CPU spends at most $t_{main_i} = \Delta_{t_{IRQ_i}} - t_{IRQ_i over}$ units of time by executing the program within $\Delta_{t_{IRQ_i}}$. In other words, t_{main_i} is proportional to IRQ inter-arrival times ($\Delta_{t_{IRQ_i}}$). Ideally, from the program execution viewpoint, it holds $\Delta_{t_{IRQ_i}} \gg t_{IRQ_i over}$ (Fig. 2a). With decreasing $\Delta_{t_{IRQ_i}}$, t_{main_i} decreases as well (see Fig. 2b, c), but is nonzero if $\Delta_{t_{IRQ_i}} > t_{IRQ_i over}$. In the worst-case ($\Delta_{t_{IRQ_i}} \leq t_{IRQ_i over}$), no CPU time remains to execute the program, i.e., $t_{main_i} \leq 0$. A system may stop working correctly or collapse suddenly if t_{main_i} drops below an application-specific level. This is typically denoted as the interrupt overload (IOV) effect [7], the seriousness of which grows with the criticality of a program.

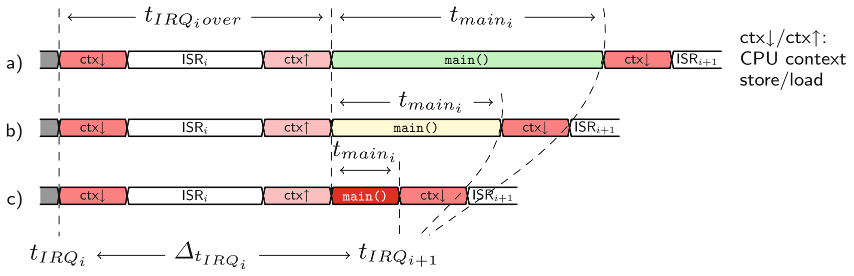

Fig. 2. An illustration to the effect of processing an IRQ to the execution of a program.

2.4 Mitigating Adverse Effects of IRQ-Based Event Detection

Adverse effects, such as IOV, of detecting events through interrupts cannot be always avoided. But, they can be, often efficiently, mitigated to maximize the predictability of a system. Such a mitigation can be done at various levels, depending on the nature of an effect. For example, so-called timing disturbance effects can be efficiently solved in a hardware (e.g., by reconfigurability of IRQ priorities) [7,8] or in a software (e.g., using the common (joint) ISR/task priority space

[9,10] or, resource access protocols [2,12] able to avoid effects such as priority inversion and deadlock). Further effects, implying from the inability of a system to predict IRQ arrival times, can be efficiently mitigated by the so-called event limiters (for an illustration, see Fig. 3). An event limiter [7] is a mechanism constructed to bound the number of event services, i.e. to limit the CPU time consumed for that purpose, within a predefined interval (t_{limit}). Basically, this can be done in four ways (Fig. 3a–d), each characterized by a different approach to detecting an event and to the limiting.

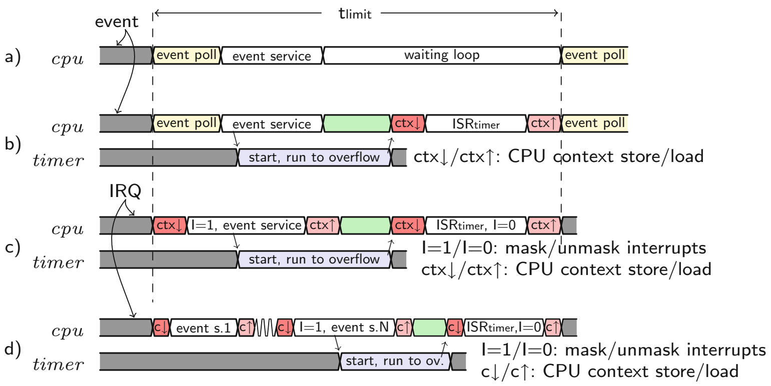

Fig. 3. An illustration to polling based event limiters that measure time by an active, software waiting loop (a) or an on-chip timer (b), resp., and interrupt based event limiters – strict (c), bursty (d); the green slots allow the CPU to execute a useful code. (Color figure online)

The first approach (Fig. 3a) represents a purely software implementation of an event limiter. At the beginning, the limiter detects an event by checking the event flag in a polling loop. If the flag is set, the event service starts. Then, an (well-tuned) active waiting loop starts to disallow checking an event flag before t_{limit} expires. In the second approach (Fig. 3b), the process of measuring time in a loop that wastes the CPU time is replaced by using a timer for that purpose. The timer measures time independently of the CPU and is able to signalize the end of the measurement by issuing an IRQ. This allows the CPU to execute a useful code of a program while the measurement is in a progress. However, using a timer leads to further overheads such as time needed to configure, start or stop the timer, to un/stack the CPU context and to execute the IRQ handler. The approaches from Fig. 3a, b are often denoted as the polling based event limiters. Alternatively, one may use so-called interrupt based limiters (Fig. 3c, d). Such limiters expect that an event is detected by an IRQ. If the IRQ is unmasked etc. (see Sect. 2.2), its handler starts (Fig. 3c, strict limiter). In the handler, further IRQs (except of timer's) are masked, the event is serviced and a timer is configured to generate an IRQ after a predetermined time. In the timer's IRQ handler, IRQs are unmasked again to resume the IRQ based event detection not before t_{limit} expires. The approach from Fig. 3d (bursty limiter, the recent industrial practice [13]) differs in a way it masks IRQs. Rather that (strictly) mask IRQs in each event handler, it masks IRQs in the last from a burst of (N) IRQ handlers. For N = 1, the bursty limiter reduces to the strict one.

3 Our Approach

This section introduces our approach, based on the means of Uppaal SMC [21], and demonstrates its applicability for the purposes of analyzing predictability of CPU based systems. Firstly, we present (Sect. 3.1) key aspects of our approach (Sect. 3.2). Secondly (Sect. 3.2), we clarify what we mean by the so-called interrupt scenarios. Finally, we present (Sect. 3.3) representative queries and results regarding predictability analysis of CPU systems in various interrupt scenarios.

To the best of our knowledge, we offer the most complex solution to analyzing predictability of interruptible CPU systems. Existing approaches, such as [15], analyze predictability for masked IRQs, [11,16] are limited to a simplified model and an analytical solution – they only support periodic IRQs, but do not support nesting of IRQ handlers, execution jitters, un/masking, priorities and arbitration of IRQs at runtime. Authors of [17] expect that an IRQ occurs not before the so-called hyper-period while in [18], it is supposed that an IRQ occurs each time an instruction completes. In [19], a method supporting nesting and priorities of IRQs is proposed, however, disregarding, unpredictability of IRQ arrival times, un/masking of IRQs at runtime and variability in executing IRQ handlers.

3.1 Proposed Model

Hardware and main(). Our model of hardware consists of three key parts: (Fig. 4a–c): the model of a system (a), of a CPU within the system (b) and of an IRQ controller within the CPU (c) whereas parts b, c have been introduced in [14][1]. In this paper, they are extended (see Fig. 4) by further aspects, such as late arriving, tail chaining, synchronization and measurements by means of stop-watches. To avoid re-publishment of the same ideas, the following text only summarizes main extensions we have made to [14]. For more details about the original models, please consult [14] and the footnote at the bottom of this page.

Simply said, the model of a system (Fig. 4a) generates the reset signal for all CPUs in the system and performs the system-level initialization. Next, the model of a CPU (Fig. 4b) waits until it receives the reset signal. Before it happens all CPU-level stopwatches are forced not to progress (`ttotal` measures the total time consumed by the corresponding CPU (`cpu`), `tmain` measures the time `cpu` spends by executing `main()`, `tover` measures the time `cpu` spends by managing interrupt limiters and `t_e` measures the time to dispatch events in a system). Then, it either fetches and processes an instruction of `main()` or, if an IRQ

[1] models are available at http://www.fit.vutbr.cz/~strnadel/publ/2018/dandt/.

pends at the moment, it skips that and moves to `irqPend`. Here it stays while a pending IRQ exists. The control flow of `main()` is given by the `fetch()` function that may reflect a stochastic behavior of `main()` and/or particular aspects of pipelines, caches etc. As a presentation of the aspects is beyond the scope of this paper, please find concepts of their modeling, e.g., in [5]. Consequently, we limit this paper just to the former way of controlling the flow of `main()` in our model. Simply said, we understand instructions of `main()` just as a factor affecting the interrupt latency. Last, the model of an IRQ controller (Fig. 4c) waits until the corresponding CPU is ready to process an IRQ. Then, it either moves to `isrFetch` (to perform the tail-chaining of IRQ handlers, when enabled) or, it captures IRQ flags by calling `captif()` and moves to `exeStop` (to interrupt either `main()` or an IRQ handler, whichever is being executed at the moment). After stacking the CPU context, the model checks (when enabled) if a late arriving IRQ is pending at the moment, masks IRQs for the given CPU and arbitrates pending IRQs. Finally, it fetches the vector of an IRQ handler that has won the arbitration and then, it starts the handler.

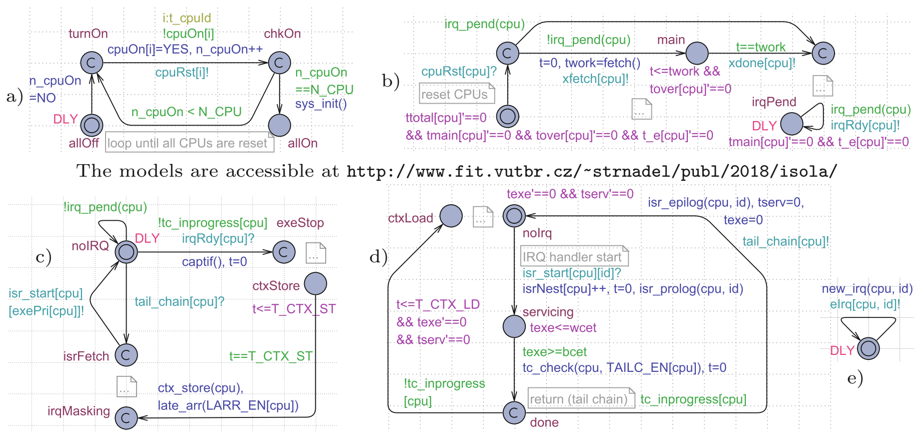
The models are accessible at `http://www.fit.vutbr.cz/~strnadel/publ/2018/isola/`

Fig. 4. Skeletons of revised models: (a) system (the model `Sys`), (b) CPU (`Cpu`), (c) IRQ controller (`irqCtrl`), (d) IRQ handler (`isrSWI`) and (e) IRQ source (`irqRst`).

IRQ Handler and Sources. The basic model of an IRQ handler has been introduced in [14] as well. In this paper, it is extended to cover the tail chaining functionality (see Fig. 4d). Alike in the case of Fig. 4b–c, the following text just summarizes main extensions we have made to [14]. For more details about the original model, please consult [14]. After the servicing completes in the extended model, the function `tc_check()` tests the preconditions for tail-chaining and the model moves to `done`. If the tail-chaining has been activated, the CPU context remains stacked and the corresponding IRQ controller is signaled to start a new (tail-chained) IRQ handler without prior stacking of the CPU context etc. If the tail-chaining has not been activated, the handler completes by unstacking the CPU context and resuming the unstacked execution.

Figure 4e illustrates a simplified skeleton used to model an IRQ source. Basically, the main role of an IRQ source lies in setting the corresponding IRQ flag. In our model, this is done by calling `new_irq()`. Dynamics of such a setting may be expressed in many ways depending on the nature (one-shot, periodic, aperiodic, sporadic [14] etc.) of a particular source. In our illustration, an IRQ source sets its IRQ flag in random moments, given by the exponential distribution of probability (with its parameter set to `DLY`).

Events. To facilitate the modeling and analysis of "non-IRQ" events (e.g., of un/masking of IRQs) being produced synchronously to instructions of `main()`, we have decided to create a separate automaton per a system/CPU (for a simple example, see Fig. 5). Let us note here that each CPU uses a separate clock (`t_e[cpu]`) to manage the occurrence times of events. For such a clock, it holds that it progresses during the execution of `main()`. If needed, it may be reset to express relativity (of the occurrence times) with respect to the time being spent in `main()`. Moreover, an event is synchronized, by `xdone[cpu]?`, with the completion of an instruction of the corresponding CPU. Such a synchronization is needed to avoid the modeling of unrealistic/contradictory behaviors such as a change of an IRQ mask during the execution of `main()`, but asynchronously to `main()`. The function `event(e)` encapsulates more complex actions related to an event `e`. At its start, the model from Fig. 5 waits until `main()` consumes 1000 units of the CPU time. Then, it starts to produce the predefined events in a cyclic, `main()` dependent, manner.

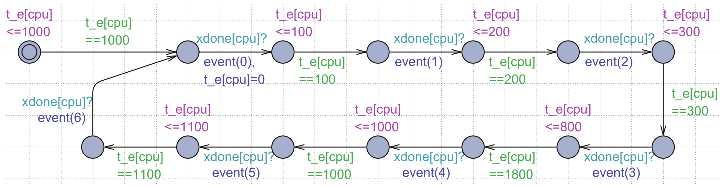

Fig. 5. Illustration to modelling of events in a system/CPU.

Event Limiters. In Fig. 3, we have presented concepts of the so-called event limiters. Below, we present an approach we have utilized to model the limiters. First, let us focus to modeling of a polling based event limiter designed to measure *tlimit* by an active waiting loop (Fig. 6a). It is supposed that the limiter starts at the beginning of `main()`, so it first waits for entering `main()`. Then, it stays in `sync` until an instruction completes. This model expects that the event flag is cyclically checked (in `poll`) by the first instruction of `main()`. If the flag is unset, the model moves (via `done`) to initiate further checking. Otherwise, the model clears both the IRQ request (given by `id`) and the clock `tlim` to measure *tlimit*, then moves to `service`. Here, it loops until the servicing takes BCET to WCET units of the CPU time. Then, the model moves to `wait_entry`, where it loops actively until *tlimit* is over. Finally, it moves toward starting a new polling instance. Let us note that `serv0` and `wait0` represent a potential preemption of the servicing/waiting process by an IRQ handler. During such a preemption, neither of the associated clocks progresses because the CPU time is not consumed by the limiter. However, some of the clocks progress in `serv1`, `wait1` to measure the consumption of the CPU time by the limiter.

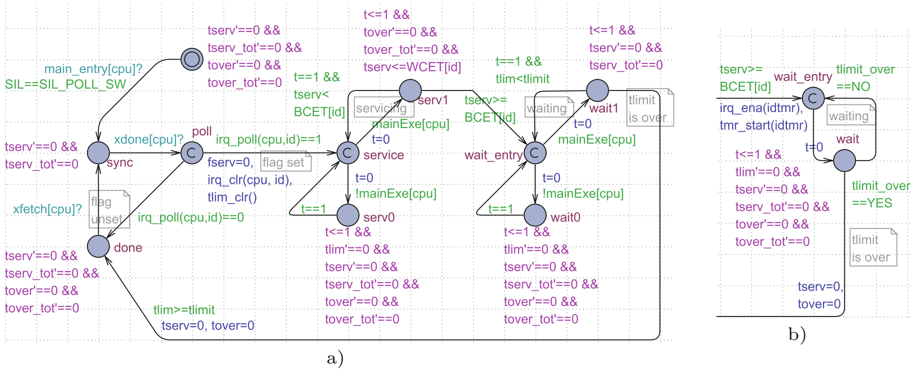

Fig. 6. Models of polling based event limiters that measure *tlimit* time using (a) an active waiting loop (full model), (b) a timer (model cutout).

Figure 6b illustrates a cutout of the model of a polling based event limiter designed to measure *tlimit* by a timer. As this limiter differs from the previous one just in the right part of the figure, nothing but details to that part are presented. Before entering `wait_entry`, IRQs for the timer are enabled and then, the timer stars to expire after (*tlimit*-`tlim`) units of time, if greater than zero. Comparing `wait1` in Fig. 6a, the CPU time is never consumed while staying in `wait` in Fig. 6b. The models of the strict and bursty limiters from Fig. 3b, c are absent in Fig. 6 simply because their modeling is trivial. Particularly, it is just necessary to modify the body of `isr_prologue(cpu, id)` of an IRQ handler to be limited. For the strict limiter, we must modify the body to mask all IRQs (except of timer's) and then, start a timer to unmask the IRQs after it overflows (the unmasking is done in the timer's IRQ handler). For the bursty limiter, we must add further modification, i.e., to increment a counter of events that started within the *tlimit* window and to check if the counter value has reached the burst size (N). If so, we initiate the strict functionality.

Further Aspects. To quantify parameters of predictability, such as WCRT, our models have utilized various instruments that have not been explained yet. The most important ones are discussed in the following text. Firstly, the CPU utilization factor of a *cpu* (U_{cpu}) is quantified as the ratio of `tmain[cpu]` to `ttotal[cpu]` (see Fig. 4b). An accurate estimation of that parameter is needed, e.g., to predict the schedulability of (a set of) RT tasks using a schedulability analysis based on the CPU utilization. Secondly, we have utilized the clock `tserv` to measure the CPU time needed to service an event, either in an IRQ (Fig. 4d) or by an event limiter (Fig. 6). Regarding an event limiter, we are interested in further times as well – particularly, in `tserv_tot` (total time needed to service all events detected by the limiter), `tover` (overhead of the limiter per an event) and `tover_tot` (total overhead of the limiter across all detected events). Moreover, we utilize a couple of counters to gather the numbers of events, started handlers, serviced events etc.; we increment the counters in the bodies of `new_irq()`, `isr_prologue()` and `isr_epilogue()`, respectively. The measurements like that allows us to compare effects of various event handling approaches from the predictability viewpoint. Thirdly, we are able to analyze the evolution of the stack pointer (SP) during runtime. For that purpose, we manipulate SP in functions such as `ctx_store(cpu)` from Fig. 4c, `ctx_load(cpu)` from Fig. 4d or a function call within the execution of `main()`. Such an analysis simplifies our efforts of adjusting safe stack sizes for RT tasks etc. in the given interrupt scenario. Fourthly, we measure the interrupt latency time, i.e., the time between i^{th} occurrence of an IRQ and starting the corresponding handler. Probably, this has been one of the most challenging problems to solve in the area of measurement. We have decided to solve the problem by means of a spawnable timed automaton (see Fig. 7). Such an automaton is created dynamically, after an IRQ occurs (the occurrence is detected via the channel `eIrq[cpu][id]`, see Fig. 4d). Then, the automaton measures time (`tilat`) until the corresponding IRQ handler starts (it is signalized via the channel `isr_start[cpu][id]`).

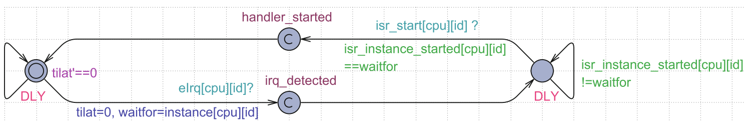

Fig. 7. An illustration to the spawnable automata for measuring the interrupt latency.

3.2 Interrupt Scenarios

We have decided to test and demonstrate applicability of our model from Sect. 3.1 in various interrupt scenarios, details to which follow. The identifier of a scenario is prefixed by "SC". To refer unambiguously to a particular scenario, we have encoded its characteristics into a binary string being situated in the right subscript of "SC". For parts of the string (i.e., base and suffix), see Fig. 8.

The value of a bit within the base part of the string (Fig. 8a) represents a boolean flag that indicates presence (1) or absence (0) of the corresponding feature such as nesting of IRQs or enabling a particular IRQ source. If an event limiter is utilized, then the string is completed with a 2-bit suffix "xy" (Fig. 8b, where xy = 00 for the polling limiter w. an active waiting loop, xy = 01 for the polling limiter w. timer, xy = 10 for the strict limiter and finally, xy = 11 for

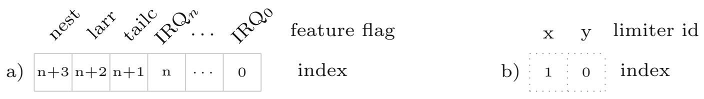

Fig. 8. An illustration to encoding an interrupt scenario: (a) base, (b) suffix.

the bursty limiter. The suffix is separated from the base by a dash ("—"); if needed, the feature flags indexed by n, n+1 may be separated by a dot ("."). If it is necessary to observe impacts of modifying an attribute of a scenario to a system, we prepare a set of modifications for that purpose and identify each of them by Roman numerals in the right superscript of "SC". For an illustration, let us present some representatives of encoding a scenario (for n = 2): SC_{00001}, $SC_{000.10}$, $SC_{11101-10}$, $SC_{110.11-11}$, $SC^{I}_{110.11-11}$, $SC^{V}_{000.10}$.

During the testing/demonstration, we have utilized four IRQ sources (i.e., n = 3 in Fig. 8a), referred to as IRQ_0 (non-maskable, highest priority IRQ with arrivals given by the normal distribution of probability), IRQ_1 (maskable, higher-middle priority IRQ with periodical arrivals), IRQ_2 (maskable, lower-middle priority IRQ with periodical arrivals) and IRQ_3 (maskable, lowest priority IRQ with arrivals given by the uniform distribution of probability). For their characteristics, see Fig. 9 please.

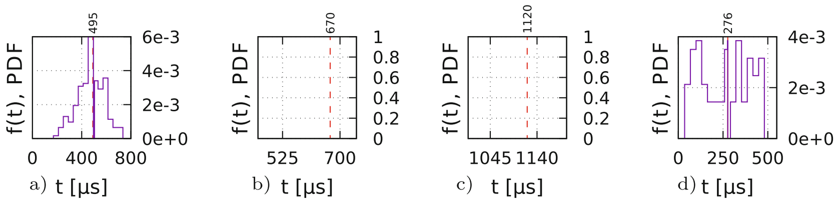

Fig. 9. Characteristics (i.e., probability distribution functions, PDFs) of IRQ sources: (a) IRQ_0, (b) IRQ_1, (c) IRQ_2, (d) IRQ_3. Vertical, dashed red lines do mark mean values. (Color figure online)

3.3 Queries and Results

Due to the limited space, we have restricted this section to just selected representatives of queries and results. For the same reason, we skip herein an introduction to the query language and refer to [20,21] instead.

Our representative results are summarized in Figs. 10, 11 and 12. They were produced by the toolset UPPAAL SMC [21] based on queries, details to which follow. Figure 10 shows the results of a query in the form `simulate 1 [<= 2500] {exePri, isrNest, ..., cpu(0).main}` being applied to four distinct interrupt scenarios.

Figure 11 presents results of four queries. Firstly, Fig. 11a shows an impact of the IRQ arrival rate to the CPU utilization for various event limiters. The subfigures (a)–(c) result from a query in the form `E[<= 25000] (max : ...)`. Secondly, Fig. 11b relies on measuring `tilat` (see Fig. 7) for IRQ_0 and IRQ_3. Thirdly, Fig. 11c results from estimating the maximum of `texe` (see Fig. 4d). Finally, Fig. 11d results from a query in the form `Pr`[<= 2500] {<> `isrNest` >= 3}. It shows the probability distribution function (PDF) and the mean of time instants when the ISR nesting level exceeds 2.

The last figure (Fig. 12) shows how our approach scales with the number of IRQ sources. We can conclude that it scales about linearly in the time domain and sub-linearly in the space (memory) domain.

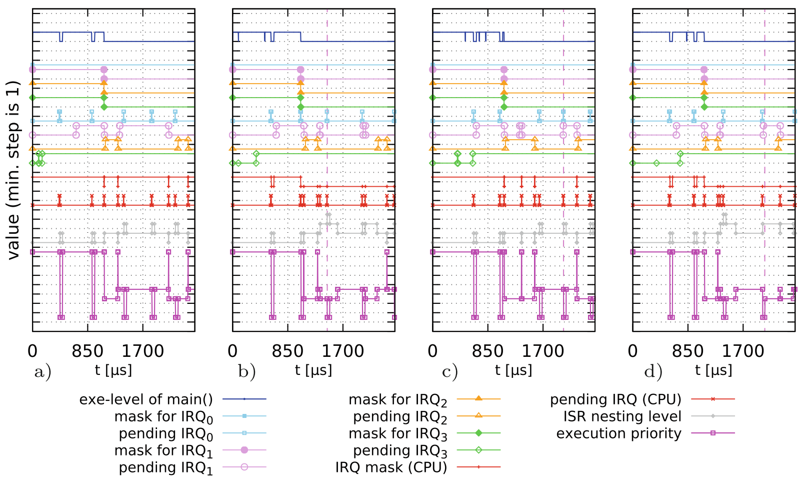

Fig. 10. An illustration to effects of (a) no nesting of IRQ handlers, no late arriving, no tail chaining, (b) nesting of IRQ handlers (c) late arriving (d) nesting of IRQ handlers, late arriving and tail chaining under $\text{SC}_{xxx.1111}$. Vertical, dashed violet lines in (b)–(d) do mark the start times of the nesting, late arriving and tail chaining, respectively. (Color figure online)

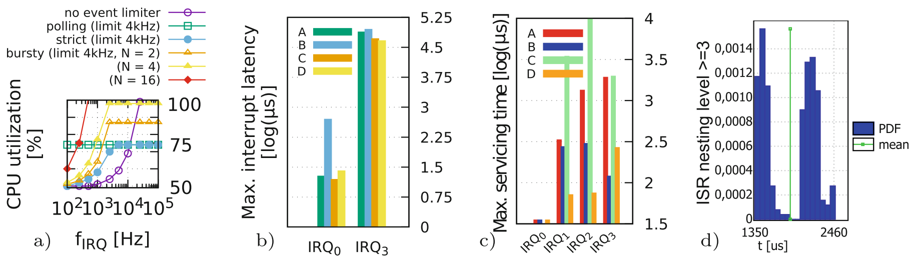

Fig. 11. Representative results of our SMC based predictability analysis. A, B, C and D represent scenarios $\text{SC}_{100.1111}$, $\text{SC}_{000.1111}$, $\text{SC}^{I}_{100.1111}$ and $\text{SC}^{II}_{100.1111}$, respectively. The modification I decreases both the mean and deviation of IRQ_0 ten times. Alike, II decreases BCET and WCET of IRQ_1, IRQ_2 ten times. All results, except (d), hold for the 25 ms window.

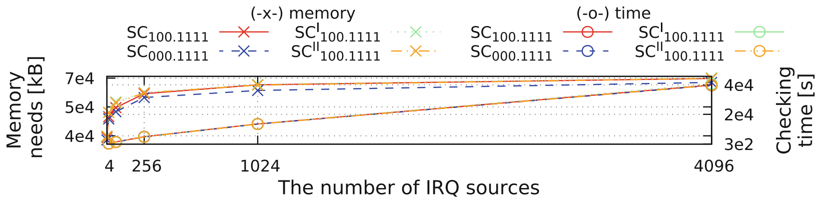

Fig. 12. The scalability of the SMC process as a function of the number of IRQ sources. The modification I decreases both the mean and deviation of IRQ_0 ten times. Alike, II decreases BCET and WCET of IRQ_1, IRQ_2 ten times.

4 Conclusion

This paper presents a simulation model of a processor based system with interruptible executions. To analyze the predictability of such a system, our model copes with adverse phenomena such as processing instructions of a program, arbitrary occurrence times of interrupts, jitters of interrupt occurrence/servicing times, priorities, or nesting and un/masking of interrupts at run-time. Such phenomena have stochastic aspects and lead to the explosion of the number of situations to be considered. In the paper, we show that such an explosion is effectively solvable by means of the statistical model checking. We show that such a checking is able to facilitate the analysis of parameters such as interrupt latency, interrupt servicing time, CPU utilization and to minimize the over/underestimation of their values. In the near future, we plan to move from the single-CPU environment to multi/many-CPU environment and to extend our model to involve the operating system, application and power consumption levels as well.

References

1. Kopetz, H.: Real-Time Systems - Design Principles for Distributed Embedded Applications. Real-Time Systems Series, 376 p. Springer, New York (2011). https://doi.org/10.1007/978-1-4419-8237-7. ISBN 978-1-4419-8236-0
2. Buttazzo, G.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 376 p. Springer, New York (2011). https://doi.org/10.1007/978-1-4614-0676-1. ISBN 978-1-4614-0675-4
3. Wilhelm, R., et al.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3), 36:1–36:53 (2008). https://doi.org/10.1145/1347375.1347389
4. Strnadel, J., Rajnoha, P.: Reflecting RTOS model during WCET timing analysis: MSP430/FreeRTOS case study. Acta Electrotechnica et Informatica **12**(4), 17–29 (2012). https://doi.org/10.2478/v10198-012-0041-3

5. Dalsgaard, A.E., Olesen, M.C., Toft, M., Hansen, R.R., Larsen, K.G.: METAMOC: modular execution time analysis using model checking. In: Lisper, B. (ed.) 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010). OASIcs, vol. 15, pp. 113–123. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2010). https://doi.org/10.4230/OASIcs.WCET.2010.113
6. Cassez, F., de Aledo, P.G., Jensen, P.G.: WUPPAAL: computation of worst-case execution-time for binary programs with UPPAAL. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 560–577. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_28
7. Regehr, J., Duongsaa, U.: Preventing interrupt overload. In: Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools For Embedded Systems, New York, United States, pp. 50–58. ACM (2005). https://doi.org/10.1145/1070891.1065918
8. Pellizzoni, R.: Predictable and monitored execution for cots-based real-time embedded systems, Ph.D. thesis, Bonn, Germany. University of Illinois at Urbana-Champaign (2010)
9. Amiri, J.E., Kargahi, M.: A predictable interrupt management policy for real-time operating systems. In: Proceedings of CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST), pp. 1–8. IEEE (2015). https://doi.org/10.1109/RTEST.2015.7369843
10. Lynx. Lynx Software Technologies Patented Technology Speeds Handling of Hardware Events (2018). http://www.lynx.com/whitepaper/lynx-software-technologies-patented-technology-speeds-handling-of-hardware-events/
11. Leyva-del Foyo, L.E., Mejia-Alvarez, P., de Niz, D.: Integrated task and interrupt management for real-time systems. ACM Trans. Embed. Comput. Syst. **11**(2), 32:1–32:31 (2012). https://doi.org/10.1145/2220336.2220344
12. Cottet, F., Delacroix, J., Kaiser, C., Mammeri, Z.: Scheduling in Real-Time Systems. Wiley, New York (2001). ISBN 978-0-470-84766-4
13. Automotive Open System Architecture GbR (AUTOSAR). Specification of Operating System. Technical report (2018). http://www.autosar.org
14. Strnadel, J.: Predictability analysis of interruptible systems by statistical model checking. IEEE Des. Test **35**(2), 57–63 (2018). https://doi.org/10.1109/MDAT.2017.2766568
15. Chattopadhyay, S., Tresina, M., Narayan, S.: Worst case execution time analysis of automotive software. Procedia Eng. **30**, 983–988 (2012). https://doi.org/10.1016/j.proeng.2012.01.954
16. Kotker, J., Sadigh, D., Seshia, S.A.: Timing analysis of interrupt-driven programs under context bounds. In: Proceedings of Formal Methods in Computer-Aided Design (FMCAD), pp. 81–90 (2012)
17. Kidd, N., Jagannathan, S., Vitek, J.: One stack to run them all. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 245–261. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16164-3_18
18. Wu, X., Wen, Y., Chen, L., Dong, W., Wang, J.: Data race detection for interrupt-driven programs via bounded model checking. In: Proceedings of the 2013 IEEE Seventh International Conference on Software Security and Reliability Companion, SERE-C 2013, Washington, DC, USA pp. 204–210. IEEE CS (2013). https://doi.org/10.1109/SERE-C.2013.33

19. Kroening, D., Liang, L., Melham, T., Schrammel, P., Tautschnig, M.: Effective verification of low-level software with nested interrupts. In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, ser. DATE 2015, Jose, CA, USA, pp. 229–234. EDA Consortium (2015). http://dl.acm.org/citation.cfm?id=2755753.2755803
20. Baier, C., Katoen, J.-P.: Principles of Model Checking, ser. Representation and Mind. MIT Press, London (2008). https://mitpress.mit.edu/books/principles-model-checking
21. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.: UPPAAL SMC tutorial. Int. J. Softw. Tools Technol. Transf. **17**(4), 397–415 (2015). https://doi.org/10.1007/s10009-014-0361-y