Testing Real-time Properties of Embedded Systems

Pawel Gburzynski
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada

Bozena Kaminska
School of Engineering Science
Simon Fraser University
Burnaby, BC, Canada

Abstract

We introduce an executable model for verifying real-time properties of embedded systems programmed under PicOS, which is a tiny operating system for small-footprint wireless devices. One of its interesting features is a close relationship with a simulation/specification package, dubbed SMURPH/SIDE, offering a frugal, powerful, and friendly programming paradigm for multithreaded reactive applications. By inheriting that paradigm, PicOS threads become amenable to simulation and emulation techniques that facilitate testing and rapid prototyping of embedded systems. In this paper, we show how those techniques can be harnessed to diagnosing problems with real-time behavior of embedded applications.
Keywords: embedded systems, real-time systems, simulation, specification, reliability, testing.

1 Introduction

The most serious problem with implementing non-trivial, structured, multitasking software on microcontrollers with limited RAM is minimizing the amount of memory resources needed to describe a thread. While the basic record of a thread in the kernel of an embedded system can be contained in a handful of simple variables (status, code pointer, data pointer, one or two links), the most troublesome component of the thread footprint is its stack, which must be preallocated to every thread (in a safe amount sufficient for the thread's maximum possible need) upon its creation. In addition to providing room for the automatic variables used by thread functions, including the implicit ones (like return addresses), the stack is an important part of the thread's context. When the thread is preempted, the stack preserves the snapshot of its trace, which will make it possible to resume the thread later, in a consistent and transparent manner.

At first sight, it might seem that microcontrollers with very small amount of RAM, say 2KB, are condemned to running thread-less systems. For example, in TinyOS [1, 2], the issue of limited stack space has been addressed in a radical manner—by avoiding multithreading altogether. Essentially, TinyOS defines two types of activities: event handlers (corresponding to interrupt service routines and callbacks) and the so-called *tasks*, which are simply chunks of code that cannot be preempted by (and thus cannot dynamically coexist with) other tasks. Although programming within these confines need not be extremely difficult, a less drastic solution may be possible.

One way to strike a compromise between the complete lack of threads on the one hand, and overtaxing the tiny amount of RAM with partitioned and fragmented stack space on the other, may be to reduce the flexibility of threads regarding the circumstances under which they can be preempted (i.e., lose the CPU). The idea is to create an environment where the thread is forced to relinquish its stack before being preempted. That would restrict the preemption opportunities to a collection of *checkpoints* of which the thread would be aware. By stimulating a structured organization of those checkpoints, we could try to 1) avoid locking the CPU at a single thread for an extensive amount of time, 2) turn them into natural and useful elements of the thread's specification, e.g., enhancing its clarity and usefully reducing the complexity of its environment. Theses ideas lie at the heart of PicOS: a tiny operating system for microcontrollers, which has been successfully employed to programming multi-purpose low-cost wireless nodes, e.g., forming meshes of sensors and/or actuators [3]. PicOS brings about a special flavor of threads structured like finite state machines and exhibiting the dynamics of coroutines with multiple entry points and implicit control transfer.

One immediate problem resulting from the limited preemptibility of threads is its potentially detrimental impact on the real-time behavior of the embedded application [4, 5]. This is because the maximum rescheduling time for any thread (regardless of its priority [6]) will include the maximum non-preemptibility interval for any other thread in the system. We discuss this issue and demonstrate how the multiple threads of a PicOS application can be organized to provide hard and soft real-time deadlines. We also explain how the application can be rapidly tested in a virtual environment to obtain measures of its responsiveness to real-time demands. Finally, we demonstrate how the feedback from those (virtual)

tests can be used to modify the original design in a way that eliminates problems observed during real-life operation of the target device.

2 PicOS threads

PicOS has been described elsewhere in different contexts, e.g., its rationale [7], the evolution of its programming paradigm [8], and its application to building cheap wireless nodes [3]. In this section, we shall focus on the semantics of PicOS threads from the viewpoint of assessing their real-time properties.

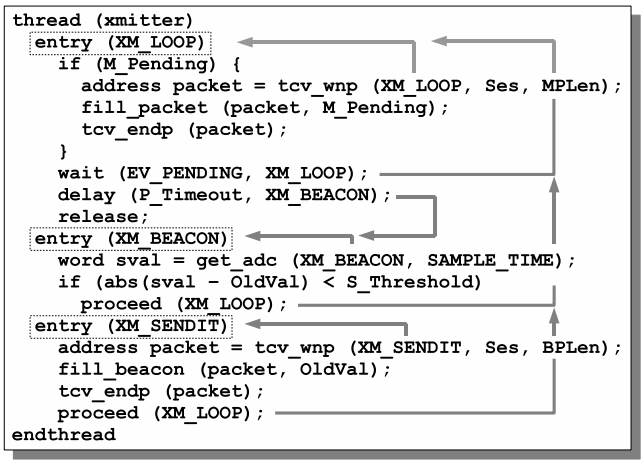

Figure 1: A PicOS thread.

Figure 1 shows a sample thread in PicOS. Such a thread is organized into a sequence of *states* giving it the appearance of a finite state machine (FSM). The possible transitions are illustrated with arrows in Figure 1. State boundaries represent the checkpoints at which a thread can be blocked (i.e., preempted) and resumed. A thread receiving the CPU is entered at a particular state and holds the CPU until it either decides to relinquish it directly (by executing `release`, as at the end of state `XM_LOOP` in Figure 1), or executes a function (system call) that cannot proceed until a certain condition is met (e.g., `tcv_wnp`). Such functions always accept a state argument: the invoking thread will be resumed (awakened) in the specified state when the awaited condition is fulfilled. That moment will be marked by the occurrence of some *event*.

It makes sense to say that the activities of a thread are driven by events that wake it up in specific states. The general view is that while executing instructions (in its current state) a thread may issue a number of *requests for events* to wake it up in the future. For example, before executing `release` in state `XM_LOOP`, the thread shown in Figure 1 issues two such requests: one with the `wait` operation specifying an explicit event identifier (representing an event to be delivered by another thread of the application), the other with `delay` setting up an alarm clock for the prescribed number of milliseconds. Whichever event happens earlier, it will wake up the thread in the respective state. Generally, a thread may be waiting for a number of independent events. Their collection determines the current options for the state transition function of the thread viewed as an FSM. When the thread wakes up, its list of anticipated events is cleared, i.e., in every state, the waking conditions are specified from scratch. This simple mechanism has proved extremely powerful, friendly, and useful for describing the kinds of applications typical of embedded systems, i.e., reactive ones [9, 10]. The self-documenting nature of PicOS threads transpires in the ease of expressing them in diagrams similar to StateCharts [11, 12].

3 Scheduling considerations

PicOS scheduler admits several options, which can be selected at the time the system is compiled into an application. The most naive (and also the most popular) of those options implements a fixed priority (non-preemptive) scheme, whereby the threads are sorted in the decreasing order of their importance. Whenever a thread releases the CPU, the scheduler assigns it to the first thread that is not waiting for any event. This way, when multiple threads are ready to run, the one closer to the front of the list will win the CPU.

In most cases, this trivial approach to scheduling is quite adequate to fulfill the real time requirements of the application, especially if those requirements are soft. This is because reactive applications tend to do little computations (are not CPU bound) and focus on processing events, which actions typically take a small amount of time. Notably, the truly critical actions (like extracting data from time-constrained peripheral equipment) are carried out in interrupts, which are not subjected to the kind of postponement experienced by threads. Consequently, the limited preemptibility of the latter does not impair the rate at which external events can be formally absorbed by the PicOS application.

3.1 A case study: part one

For illustration, consider an actual PicOS application in a BCG-based [13] heart-monitoring wireless datalogger device built around an MSP430 microcontroller[1] absorbing synchronous analog data from the cohort of eight sensors (accelerometers), digitizing them and storing locally in flash memory. As an option, the collected samples can be simultaneously transmitted over a simple

[1] From Texas Instruments, see http://www.ti.com/.

wireless channel[2] to the *access point* (AP). The AP can also request the transmission of a previously collected series of data stored in the datalogger's database.

Sample arrival events are strobed by the on-chip ADC converter at the rate of 500 Hz, with a single digitized sample consisting of 8 12-bit values packed into 12 bytes. The hard real-time constraint of the application is the need to avoid losing any samples at the stage of their collection. While the typical amount of time needed to store a sample in flash memory is below 100 μs (including software overhead), every once in a while, at a page boundary, it may jump to $t^W_{max} = 50$ ms, with the average of about 22 ms. With the page size of 512 bytes, and the sample size of 12 bytes, that hiccup happens roughly every 43 samples, which yields ca. 500 μs as the average write time per sample. Thus, with the inter-sample interval of 2 ms, the *sample collection thread* uses up about 25% of that interval for storing one sample, with an uneven, albeit reasonably periodic, distribution. As the delays caused by flash hiccups do not occupy the CPU (the collection thread is blocked and preempted while waiting for the flash to become writable), most of that time is available to other threads.

The physical acquisition of samples from the ADC is carried out within an interrupt service routine, which can never be delayed by more than a few microseconds. No sample can be lost at this stage. To compensate for the jitter in accommodating the samples in flash, the interrupt service routine stores them in a circular buffer in RAM, whose output end is serviced by the collection thread. To determine the minimum safe length of the circular buffer, one has to consider, in addition to t^W_{max}, the maximum rescheduling time [6] for the collection thread. Being the only thread with hard real-time constraints, the collection thread was assigned the highest priority in the application. Thus, its maximum rescheduling time can be expressed as $t^R_{max} = t_{sched} + t^S_{max} + t^I_{max}$, where t_{sched} is the (fixed) cost of running the scheduler, t^S_{max} is the longest execution time of a state among all the other threads, and t^I_{max} is the longest possible *interrupt interference* that can happen during $t_{sched}+t^S_{max}$. The latter is required because an interrupt will preempt any thread, which will be duly resumed upon the completion of the interrupt service routine. It is relatively easy to authoritatively estimate those delays by mechanically counting all machine instructions[3] representing the respective actions, especially that all the involved instruction sequences are loop-less. In particular, the maximum-time state code in the application was found to contain 213 CPU-level instructions with the total contribution of 802 CPU cycles. With the 4.5MHz CPU clock, this translates into $t^S_{max} < 180\mu$s. Three types of interrupts are allowed to occur during sample collection. Two of them: the system clock (running at 1024 ticks per second) and the ADC strober (500 times per second) are synchronous, while the third one (arriving from the RF module) has the minimum spacing interval of 2 ms. With t_{sched} bounded by 22 μs, each of the three interrupts can occur at most once within $t_{sched}+t^S_{max}$, and in the worst case, if all of them actually occur, their joint time contribution is limited by $t^I_{max} = 250\,\mu$s. Thus, we determine that $t^R_{max} = 452\,\mu$s, which is two orders of magnitude below t^W_{max}.

These considerations allow us to conclude that the minimum safe buffer size is determined solely by t^W_{max} and should conservatively account for up to 64 ms of delay in sample extraction, which translates into 32 samples (384 bytes).

3.2 Granularity of thread states

One moral from the above case study is that when organizing threads in PicOS, one should try to minimize (or at least contain) the maximum execution time of a single state, at least in those circumstances where the system has to conform to hard real-time requirements. Note that the granularity of thread states is under the programmer's control. There is virtually no penalty for introducing extra states with the purpose of bringing down the maximum duration of a CPU burst exhibited by the thread. In many circumstances, in addition to improving the real-time behavior of the entire application, this approach enhances the clarity, self-documentability, and re-usability of the thread code.

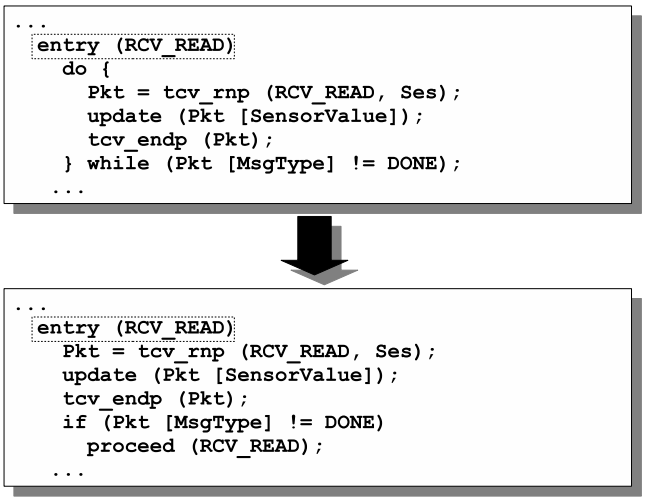

Figure 2: Avoiding non-trivial loops in PicOS threads.

It is also recommended to avoid non-trivial loops that do not cross state boundaries. Such loops can be always converted as shown in Figure 2, i.e., by starting the loop at a state boundary and closing it with `proceed`, which has the effect of enabling preemptibility at every turn.

[2]Implemented with the CC1100 RF module from TI.

[3]That was accomplished by a special program analyzing the assembly-language output of the compiler.

Formally, `proceed` is equivalent to a delay for 0 milliseconds followed by `release`. This way, the state transition involves the CPU scheduler, which will account for any higher priority threads before returning control to the present one.

All internal functions (system calls) of PicOS have been programmed with consistent adherence to the principle of simplicity and orthogonality. This also applies to the internals of VNETI,[4] which is a plugin-extensible layer-less networking module implementing buffer space management for accommodating incoming and outgoing packets. Every function implements a loop-less action, whose execution time is approximately constant. This way, the timing of code referencing such functions is easy to estimate within a very narrow uncertainty margin. With the avoidance of intrinsic loops in thread states, the calculation of t^S_{max} (see Section 3.1) mostly boils down to a purely mechanical tally of the CPU cycles in a sequence of machine instructions.

4 Real-time conformance testing

The case study in Section 3.1 demonstrates that simple RT requirements (involving a single high-priority thread) can be assessed by a straightforward (static) analysis of the application code. One option for testing a PicOS application before its deployment, including possible tests of its RT conformance, is a virtual execution in VUEE,[5] which is an emulated environment for executing wireless applications in PicOS at the level of their source code. VUEE is made possible by the close relationship of PicOS threads to SMURPH/SIDE [14], which allows us to mechanically adapt those threads to execution under SMURPH. Note that, if needed, the timing information regarding the amount of CPU time needed to execute a given sequence of states under PicOS can be obtained by the analysis of compiler output. At least in the case when those states are loop-less (which is the recommended practice), this analysis can be accomplished by a program. As SMURPH is equipped with detailed models of RF channels [15], the virtual execution under VUEE can be practically indistinguishable from real execution—to the point of appearing as a real application to its external agents.

4.1 Randomized state exploration

Recently, we have implemented a lighter-weigh tool, dubbed TERSE,[6] aimed at a quick automatic assessment of the application's RT behavior. It has already proved useful for obtaining authoritative performance bounds, e.g., upper limits on service delays, as well as detecting and diagnosing problems ("unexpected interferences"—see Section 4.2) in the series of state transitions involving different (competing) threads. TERSE utilizes a flavor of simulation in SMURPH based on a simplified model of the application. The simplification consists in replacing the external sources of events driving state transitions in threads with stochastic processes. Moreover, the actual contents of those states are represented by simple timed execution models that capture the three aspects relevant from the viewpoint of real-time behavior: 1) the time cost of executing the state, 2) its consequences in terms of enabling other states, possibly in other threads, and 3) the scheduling mechanism that selects the next state among the enabled ones. A model like this can be executed fast (taking advantage of the discrete-time event-driven semantics), thus exploring a large number of states, i.e., experiencing a large interval of virtual time, within an incomparably shorter amount of the real execution time. While the problem specification is simplified, it can be adequate to obtain meaningful bounds. Depending on the required quality of those bounds, one may even be able to get away with a grossly oversimplified specification, as long as all the simplifications have been applied "in the safe direction." This is to say, the bounds may be crude (overly pessimistic), but they will be valid.

Let S^T_i denote a state of some thread T. Such a state can be equivalenced with a part of the global transition function of the application and described by the set of so-called *effectuations*: $S^T_i = \{C_j\}$, where each effectuation C_j is a triplet, $C_j =< T_j, E_j, W_j >$ representing one possible execution path of the state. Its attributes denote the amount of time (T_j), the set of events triggered by the thread and possibly awaited by other threads (E_j), and the set of *wait requests* describing possible transitions to other states (W_j). While T_i is a single number, both E_j and W_j are sets. The elements of E_j are event identifiers, and each item in W_j consists of a pair: a wait condition and the state to be assumed when the condition is met. A wait condition can be of one of three types: 1) a fixed time delay, 2) an explicit event triggered by another thread of the application, 3) an external event. Note that the `proceed` operation is viewed as a special case of fixed-time delay.

For illustration, consider the thread shown in Figure 1. The set of effectuations pertaining to its first state may look like this:

```
<15, NULL, {FREEMEM -> XM_LOOP}>
<27, NULL, {EV_PENDING->XM_LOOP, P_Timeout->XM_BEACON}>
<39, {XQUEUE}, {EV_PENDING->XM_LOOP, P_Timeout->XM_BEACON}>
```

The first effectuation corresponds to the situation when `tcv_wnp` blocks: the thread loses the CPU awaiting an enabling event to re-try the operation, which will be triggered whenever any thread releases some memory. The second effectuation represents the case when the `if` statement fails, and the threads issues two wait requests

[4] Versatile NETwork INterface.
[5] Virtual Underlay Execution Engine.
[6] Timing Estimation by Random State Exploration.

at the end of the state sequence. Finally, the third effectuation includes the second one plus the (implicit) event triggered by `tcv_endp` (the function queues an outgoing packet and notifies the RF transmission thread). The number starting each triplet represents the upper bound on the path execution time, including the requisite activity of the CPU scheduler and all the necessary statements within the state.

The intention of the TERSE model sketched above is to simulate the behavior of the complete cohort of application threads by mimicking their enabling conditions and the amount of time spent by them in the different states. For that the model needs a way to select among the possibly multiple paths of a state, as well as a source of external events. The first part can be implemented in several ways, depending on the required accuracy of the model. For example, the condition in the first state of the `xmitter` thread from Figure 1, can be modeled directly by simply maintaining the requisite variable (`M_Pending`) and updating it consistently in the same way as in the original application. Generally, it makes sense to associate one more item with an effectuation (turning it into a 4-tuple), namely the *action* specifying a sequence of operations affecting some auxiliary objects in the model. One standard role for such actions is to collect performance measures related to the application's RT behavior and/or validate assertions identifying abnormal or special conditions. One example of such a measure is the longest time interval separating two consecutive executions of the same state.

In those situations when a deterministic resolution among the multiple execution paths is impossible or infeasible, one can resort to randomization. In many cases, when we are interested in crude (but valid) upper bounds on delays, random decisions can be used even in those cases when deterministic ones wouldn't be prohibitively difficult to implement. Statistically, such simulations will tend to explore more global paths than are ever reachable in the real application; however, they will not exclude any feasible path. To this end, the model provides for an easy description of stochastic processes driving effectuation selection as well as the timing of external events, i.e., those that do not originate in application threads. For the latter, in addition to the timing of the arrival process, the description also covers the duration of the interrupt service routine needed to present the event to the collection of threads. That duration adds to the formal execution time of the current thread state preempted by it.

4.2 A case study: part two

A TERSE model, built according to the previous section, was applied to a performance study of the device described in Section 3.1. For the exercise discussed here, we focused on the issue of selecting the best priority assignment to the threads of our application (tentatively assuming that the collection thread occupies the top location). The remaining threads include: the heart-rate monitor (whose role is to calculate the running value of the heart beat rate), the RF receiver (responsible for extracting packets from the output FIFO of the CC1100 RF module and storing them in buffers), the RF transmitter (feeding the input FIFO of CC1100 with outgoing packets), the LCD driver responsible for keeping the LCD display contents up to date, and the application monitor (handling commands arriving from the AP as well as sending periodic beacons/status messages over the RF channel).

One interesting aspect of the application's performance was the operation of simultaneous transmission of the collected samples together with their acquisition. Owing to the fact that the channel capacity was stretched to its limits, the transmission protocol was extremely simple: the samples were packaged into 48-byte packets (4 samples per packet) and sent blindly without any feedback from the recipient (the AP). At the end, the AP would determine which samples were missing and request their retransmission, possibly in a series of rounds, until all of them were eventually received. A problem that immediately surfaced in our tests with the real device was a slight lag of the transmitter, which were not quite able to keep pace with the collection thread. The buffer accommodating outgoing packets would occasionally overflow, and some packets would have to be dropped. While the problem was not critical (the missing samples were requested by the AP at the end of the collection phase and retransmitted), it was annoying because our numerical estimations suggested that the RF module should be able to catch up to the sample arrival process.

Note that the rather idiosyncratic conditioning of our system precluded a solution with two pointers into the flash storage, whereby the collection thread would write the consecutively acquired blocks at one pointer, while a separate sending thread would read them at a different pointer (at its own preferred rate). That was because the constant switching between reading and writing would drastically increase the average write access time to the flash (which would compromise the RT performance of the collection thread) as well as considerably reduce its life time. Consequently, dropping the overrunning packets and retransmitting them later was the only viable solution.

The problem was studied by modeling the application and simulating the RT behavior of its threads in TERSE (Section 4.1). Owing to its synchronous character, the sample collection component of the application was easy to model accurately. The delays and timing characteristics of the RF module were measured experimentally and plugged into the model. Our primary concern was the dynamics of the duty cycles of the RF

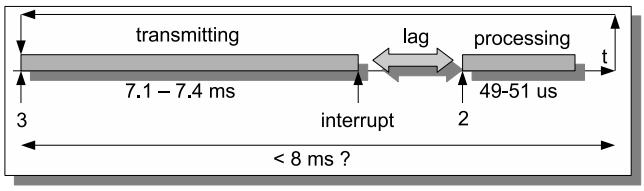

Figure 3: The transmitter cycle.

module while transmitting the packets queued by the collection thread. The output end of the packet queue was processed by the transmitter thread of the RF driver according to this scheme:

1. If the queue is empty, wait for an `XMQUEUE` event (waking up at 1) and block.

2. If the RF module is busy (emptying the FIFO), wait for an `XREADY` event (waking at 2) and block. The `XREADY` event will be triggered by the RF module as an interrupt.

3. Fill the FIFO of the RF module with a packet and proceed at 1.

The timing diagram is shown in Figure 3, with the *lag* arrow representing the only variable delay component, i.e., the amount of time separating the physical occurrence of the `XREADY` interrupt from the moment the transmitter thread receives the CPU in state 2 of its action sequence.

The dominant and mostly fixed component of the timing diagram is the actual transmission time, which demonstrates that the RF channel operates at the limit of its ability to cope with the data arrival rate. The wide 8 ms interval denotes the formal synchronous interarrival time of the amount of data contributing to a single packet. Owing to the fact that a single packet completely occupies the module's FIFO, there is no way to compensate for the transmitter's rescheduling lag: the operation of filling the FIFO for the next transmission must strictly follow the `XREADY` event. Consequently, the distribution of this lag may impact the transmitter's ability to expedite all outgoing packets at their arrival rate. This is because a single increase (fluctuation) in that lag directly translates in a decrease of the effective transmission rate: the lag time must be subtracted from the useful component of the module's duty cycle.

The model immediately allowed us to see the distribution of the lag time, which turned out to be between 20 and 1500 μs, with the mean around 150 μs and a rather large variance. Note that the occasional occurrences of low lag can only help if the queue of outgoing packets is not empty. Another finding from the model was that the lag tended to grow precisely in those circumstances when the queue was filled up, thus provoking packet loss.

The culprit turned out to be the collection thread, whose importance, in terms of its real-time requirements, was overestimated by our first design. During a flash hiccup, the collection thread would block for up to t^R_{max} (Section 3.1) while the ADC converter would fill up the circular buffer with samples waiting to be stored and transmitted. Then, when the flash became available, the collection thread would loop through the queued samples, processing them, i.e., writing them to the flash and queuing them for transmission, while the transmission thread was unable to receive the CPU. That activity tended to use about 80 μs of CPU time per sample. As it wasn't uncommon for the number of samples accumulated during the hiccup to reach 20, the total duration of that burst could reach over 1.6 ms. Even though the collection thread was programmed in a loop-less manner (see Section 3.2), that didn't help, because it was the highest priority thread. Consequently, all the other threads were forced to wait until all the samples accumulated during the hiccup had been processed. That would result in a large momentary lag of the transmitter.

The immediate suggestion from the above study was to give the highest priority to the transmitter thread. Note that that would be counterintuitive without the requisite insight, because formally the collection thread is the only thread in our application with hard RT requirements. The transmitter thread, on the other hand, was not initially considered critical: its priority was assigned below even that of the receiver thread. The argument was that reception is more important than transmission, because responsiveness to the commands sent by the AP could be considered a soft real time requirement. Having moved the transmitter thread to the top of the scheduler's list, we immediately saw a tremendous improvement. Even during heavy hiccups, the transmitter was allowed to run in between the multiple backlogged samples, thus shortening the lag to an acceptable value. Notably, extensive tests in the model, as well as numerous field experiments, exhibited no losses in the collection thread with the original size of the circular buffer. That was because 1) the longest effectuation of a transmitter state was relatively short (of the order 70 μs), 2) the transmitter was always forced to block (for at least 7 ms—see Figure 3) following a non-trivial CPU burst; thus, it would never hog the CPU in a manner similar to the collection thread.

5 Conclusions

We have shown how a special flavor of simulation can help us make meaningful statements concerning the real-time behavior of threads in an embedded system. The tool discussed in this paper is currently at a relatively early stage of development, but our plans include integrating it into a comprehensive platform encompassing

PicOS and VUEE in way that would automate most operations that are currently carried out by hand. For example, the construction of the set of state effectuations described in Section 4.1, specifically tagging them with execution time, can be done entirely by a program. The right way to this end seems to be a transformation of the VUEE model, which already includes a simulation kernel as its component. The user would be responsible for identifying and describing the external elements of the node's environment, which may be viewed as an optional add-on to the VUEE part of the application's description.

The most beneficial feature of the kind of testing described in the paper is its enormous speed-up (not to mention sheer convenience) compared to field tests of the real hardware. For example, the complete insight into the misbehavior of the transmitter thread (Section 4.2), including the way to remedy the problem, was acquired after about ten minutes of experimenting and looking at the timing data. The extent of a five-minute run, in terms of the statistical coverage of the application's state space, corresponds to weeks and months of field testing. This is not only because of the actual speed up in (virtual) execution time of the model (three or four orders of magnitude) but also due to the possibility of simulating (emulating) conditions that are difficult to come upon in a real-life test.

Finally, we would like to mention that the voluminous insight from running the threads in a virtual world and looking at the timing of their actions brings about specific suggestions for scheduling policies to be incorporated into the operating system. We are experimenting with ideas like partial round-robin, hinted state transition delays, priority drops, and so on, which have been inspired by TERSE models.

References

[1] J. Hui, "TinyOS network programming (version 1.0)," 2004, TinyOS 1.1.8 Documentation.

[2] P. Levis *et al.*, "TinyOS: An operating system for sensor networks," in *Ambient Intelligence*, W. Weber, J. Rabaey, and E. Aarts, Eds. Springer, 2005, pp. 115–148.

[3] P. Gburzyński, B. Kaminska, and W. Olesinski, "A tiny and efficient wireless ad-hoc protocol for low-cost sensor networks," in *Proceedings of Design Automation and Test in Europe (DATE'07)*, Nice, France, Apr. 2007, pp. 1562–1567.

[4] F. Balarin *et al.*, "Scheduling for embedded real-time systems," *IEEE Design and Test of Computers*, vol. 15, no. 1, pp. 71–82, 1998.

[5] J. Liu and E. Lee, "Timed multitasking for real-time embedded software," *IEEE Control Systems Magazine*, vol. 23, no. 1, pp. 65–75, 2003.

[6] K. Schwan and H. Zhou, "Dynamic scheduling of hard real-time tasks and real-time threads," *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 736–748, 1992.

[7] E. Akhmetshina, P. Gburzyński, and F. Vizeacoumar, "PicOS: A tiny operating system for extremely small embedded platforms," in *Proceedings of ESA'03*, Las Vegas, June 2003, pp. 116–122.

[8] W. Dobosiewicz and P. Gburzyński, "From simulation to execution: on a certain programming paradigm for reactive systems," in *Proceedings of the First International Multiconference on Computer Science and Information Technology (FIMCSIT'06)*, Wisla, Poland, Nov. 2006, pp. 561–568.

[9] K. Altisen, F. Maraninchi, and D. Stauch, "Aspect-oriented programming for reactive systems: a proposal in the synchronous framework," Verimag CNRS," Research Report #TR-2005-18, Nov. 2005.

[10] B. Yartsev, G. Korneev, A. Shalyto, and V. Ktov, "Automata-based programming of the reactive multi-agent control systems," in *International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, Waltham, MA, Apr. 2005, pp. 449–453.

[11] D. Harel, "On visual formalisms," *Communications of the ACM*, vol. 31, no. 5, pp. 514–530, May 1988.

[12] D. Drusinsky, M. Shing, and K. Demir, "Creation and validation of embedded assertions statecharts," in *Proceedings of 17th IEEE International Workshop Rapid Systems Prototyping*. IEEE CS Press, 2006, pp. 17–23.

[13] O. Such *et al.*, "On-body sensors for personal healthcare," in *Advances in Health Care Technology: Shaping the Future of Medical Care*, B. Spekowius and T. Wendler, Eds. Springer, 2006, vol. 6, pp. 436–488.

[14] W. Dobosiewicz and P. Gburzyński, "Protocol design in SMURPH," in *State of the art in Performance Modeling and Simulation*, J. Walrand and K. Bagchi, Eds. Gordon and Breach, 1997, pp. 255–274.

[15] P. Gburzyński and I. Nikolaidis, "Wireless network simulation extensions in SMURPH/SIDE," in *Proceedings of the 2006 Winter Simulation Conference (WSC'06)*, Monetery, California, Dec. 2006.