Bucket of Ignorance: A Hybrid Data Structure for Timing Mechanism in Real-Time Operating Systems

Marcel Ebbrecht, Kuan-Hsun Chen and Jian-Jia Chen
Design Automation and Embedded Systems Group
Technische Universität Dortmund, Germany
{marcel.ebbrecht, kuan-hsun.chen, jian-jia.chen}@tu-dortmund.de

***Abstract*—To maintain deterministic timing behaviors, Real-Time Operating Systems (RTOSes) require not only a task scheduler but also a timing mechanism for the periodicity of recurrent tasks. Most of existing open-source RTOSes implement either a tree-based or a list-based mechanism to track which task is ready to release on-the-fly. Although tree-based mechanisms are known to be efficient in time complexity for searching operations, the additional effort processing removals and insertions are also not negligible, which may countervail the advantage, compared to list-based timer-managers, even with small task sets. In this work, we provide a simulation framework, which is ready to be released, to investigate existing timing mechanisms and analyze how do they perform under certain conditions. Throughout extensive simulations, we show that our proposed solution indeed requires less computation effort than conventional timing mechanisms when the size of task set is in the range of 16 to 208.**

***Index Terms*—Timing Mechanisms, Hybrid Data Structure, Bucket of Ignorance, Real-Time Operating System**

I. Introduction

To guarantee the real-time properties on safety-critical systems, real-time operating systems (RTOSes) have been widely adopted. For uniprocessor systems, two types of scheduling algorithms are usually considered: fixed-priority (FP) scheduling and dynamic priority scheduling. Particularly Earliest-Deadline-First (EDF) policy as a dynamic priority scheduling is proved to be an optimal scheduling policy for uniprocessor systems [5], whereas it requires a large runtime overhead for updating absolute deadlines from a job to the other. In contrast, people commonly believe that the FP scheduling provides a more practical basis with a lighter operating overhead [8], by using a bit-map representation for the ready queue resulting in $\mathcal{O}(1)$ queuing and de-queuing operations [4], which is supported in many RTOSes, e.g., Amazon FreeRTOS [1] and RTEMS [9]. However, an RTOS requires not only a task scheduler but also a timing mechanism for maintaining the periodicity of recurrent tasks to keep deterministic timing behaviors [3].

In an RTOS that runs a strictly periodic taskset where a new job for each task τ_i is released every period, the timer-manager that implements one specific timing mechanism completes the following operations at every system tick t_k:

- search the underlying structure for a timer-item with an expiry time equal or lower to the current system tick,
- remove this item, release the corresponding task, and
- insert a new timer-item into the data-structure to ensure strict periodicity.

As these steps introduce runtime overhead at each t_k and raise the time required for the job activation primitive, the required computation effort is directly related to the overhead for processing the original tasks. It is worth noting that actually the above steps are designed in *an EDF manner*, by which the next ready-to-be-released task can be tracked according to its assigned period [2]. If the time complexity of the underlying timing mechanism is large, the time complexity of adopted scheduling algorithm might be dominated. We pose this study as RILO-Queuing (random in lowest out) problem: Random values get stored into a data structure, but only the lowest value needs to be accessed and removed.

In order to study the behavior of different existing timing mechanisms, at first we develop a simulation framework to analyze the list-based management in Amazon FreeRTOS [1] and the red-black tree-based management in RTEMS [9]. Additionally, we propose a new timing mechanism that overcomes one major weakness of the existing ones. The conducted simulations with randomly generated task sets with log-uniform distributed periods suggested by Emberson et al. [7] show that, our proposed timing mechanism can save up to 53% of computation effort in terms of value or null comparisons (defined in Section V-A), which is more efficient than both existing mechanisms (see Section V-B).

Our Contributions:

- We investigate existing timing mechanisms and show by our simulation results how they perform under certain conditions (see Section III).
- We introduce our new approach for a new timing mechanism based on a hybrid data structure that is more efficient than lists and trees for this purpose under the same conditions (see Section IV).
- We present a self-developed simulation framework we used for evaluation (see Section V) and the related results and configurations are all available on GitHub [6].

Since our current solution does not always outperform the other timing mechanisms, we plan to further explore the design space of various data structures to find a better solution while considering the related properties of targeted tasksets. Furthermore, we plan to develop a benchmark with crucial tests for the RILO-Queuing problem formally, by which the design of timing mechanisms might have a common standard to judge the performance of implementations in the future.

II. System Model and Problem Definition

In this section we give the definition of computation effort, our task model and the corresponding timer-items. Next we explain how the timer-manager handles these timer items and at last we point out problems of current implementation.

As this paper is about real implementations, the well-known asymptotic bounds for different operations are only somewhat helpful. For a sophisticated view on a real system, we define the computation effort E that is needed in the code to fulfill a certain operation as the number of comparisons that occur.

A. Task Model

A taskset $\mathcal{T}$ consists of n (strictly) periodic tasks $\tau_i = (\Phi_i, C_i, T_i, D_i, j_i)$ with $i \in \{1...n\}$, periods $T_i \geq T_{i+1}$ and a job counter j_i initialized as $j_i = 0$. A job J_i of τ_i is the j-th released instance of this task. As the focus of the timer-manager is on the strictly periodic releases of these tasks' jobs, we ignore the workload C_i, the phase Φ_i and the deadline D_i and reduce a task to $\tau_i = (T_i, j_i)$.

B. Timer-Items and Timer-Manager

In this paper, we focus on handling the timer-items for tasks in RTOS and exclude other events like interrupts as they do not affect the way the timer-manager works in principle. A timer-item $I_i(e_i, \tau_i)$ consists of an expiry-time e_i for each job J_i of a task $\tau_i \in \mathcal{T}$ with period T_i.

The timer-manager M implements one specific timing mechanism. It consists of a data-structure L that keeps the timer-items and functions to manage this data-structure i.e. *insert(I_i)*, *remove(I_i)* and $minTimer()$: this function returns $I_i \in L$, such that $\forall I_k \in L \setminus I_i : e_i \leq e_k$. The timer-manager processes each tick interval t_k, as shown in Figure 1. There are three main events:

1) **Scheduler starts:** At $t_k = 0$ M creates I_i for all $\tau_i \in \mathcal{T}$ and starts the processing for the first time.
2) **Stop processing:** When there is no more eligible I_i in L (minimal found $e_i > t$), then M stops processing for that t_k.
3) **New Tick:** When a new system-tick occurs, M restarts processing L.

As stated earlier in the introduction, there are three main operations: 1) Search for I_i with the minimal e_i, 2) remove the I_i with the minimal e_i and 3) insert a new I_i with $e_i = T_i \cdot (j + 1)$. Therefore, on task sets with periods greater than 1, the search operation is the most executed one and should need effort as little as possible. On the other hand the effort for removing and inserting timer-items should be kept low as well.

C. Problem of Current Timing Mechanisms

Currently, the timing mechanisms of most existing open-source RTOSes are implemented based on a tree structure, i.e., in RTEMS the timer searches for the next first watchdog in a Red-Black tree, whereas some use list-based approaches, i.e. FreeRTOS. At the first glance, a tree-based approach might be the best choice, as it performs very well when it comes to searching, especially when data is large, i.e., searching performs at $\mathcal{O}(\log n)$, whereas list-based approaches usually need $\mathcal{O}(n)$. Nevertheless, inserting and removal operations on tree-based approaches lead to additional operations, e.g., rotation and re-balancing, and this effort may mitigate the advantage on searching.

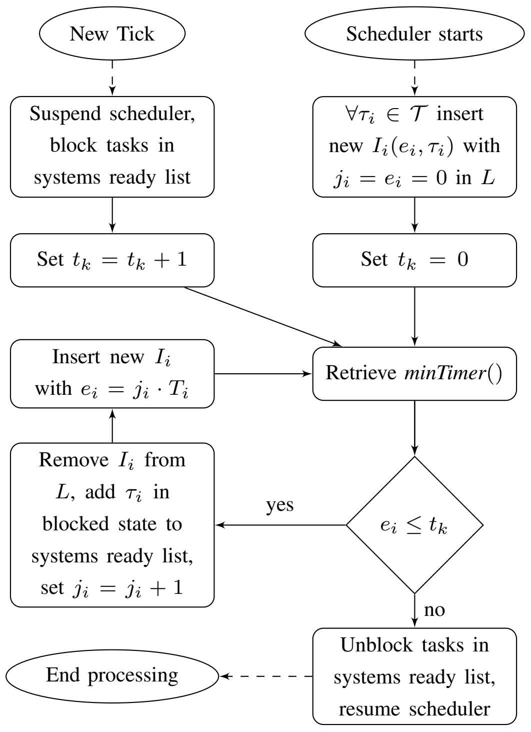
New Tick
Scheduler starts
Suspend scheduler, block tasks in systems ready list
$\forall \tau_i \in \mathcal{T}$ insert new $I_i(e_i, \tau_i)$ with $j_i = e_i = 0$ in L
Set $t_k = t_k + 1$
Set $t_k = 0$
Insert new I_i with $e_i = j_i \cdot T_i$
Retrieve *minTimer*()
Remove I_i from L, add τ_i in blocked state to systems ready list, set $j_i = j_i + 1$
yes
$e_i \leq t_k$
no
End processing
Unblock tasks in systems ready list, resume scheduler

Fig. 1. Timer-Manager processing

III. Existing Timing Mechanisms

In this section we present the respective advantages and disadvantages of two commonly used timing mechanisms.

A. List-based Management

Because M searches I_i with the lowest e_i each t_k, the I_i are sorted by ascending e_i in a linked list, such that *minTimer*() returns the head with $\mathcal{O}(1)$. To remove a I_i, only the head pointer of the list must be changed. The downside occurs on assorted insertion: L must be searched for the next correct position to insert the new I_i. This needs additional effort for each processed position. The higher the period, the more effort this process needs, i.e., $\mathcal{O}(n)$ in the worst case. The advantage of list-based management is clearly on receiving the next expiring I_i, but it suffers from the search effort when inserting a new I_i.

B. Tree-based Management

The e_i of each I_i is used as key for the nodes N_j and $minTimer()$ points to the head item from the leftmost node's list. Every node N_j holds a list $C(N_j)$ of timer-items with equal e_i. Insertion of new I_i is shown in Figure 2. On removal,

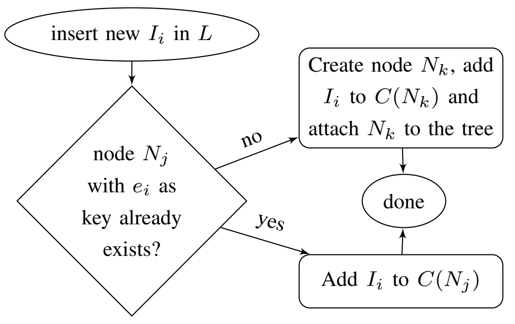
insert new I_i in L
node N_j with e_i as key already exists?
no
Create node N_k, add I_i to $C(N_k)$ and attach N_k to the tree
yes
done
Add I_i to $C(N_j)$

Fig. 2. Tree: Insert new I_i in L

I_i is removed from a node N_j by removing it from $C(N_j)$. If $C(N_j)$ becomes empty, N_j is removed from the tree.

While searching the correct position on inserting a new I_i runs in $\mathcal{O}(\log n)$ in the worst case, $minTimer()$ needs this effort as well. With the growing periods, the high effort for checking for eligible timers each tick might countervail the advantage from fast insertion obviously. Additionally, inserting and removing nodes both cause additional efforts, i.e., rotation and re-balancing, on balanced trees.

IV. New Timing Mechanism: Bucket of Ignorance

In this section, we present our approach to reduce the effort for the management of L. Both aforementioned mechanisms share one weakness: They insert the new I_i directly at the correct position even if e_i is significantly higher than current t_k. This is potentially a waste of computation time. To eliminate this potential waste, we propose a hybrid structure, that actually combines an ordered list L_O, as used in Section III-A, with an unordered list L_B. We term our strategy as *Bucket of Ignorance* (BoI). Additionally we define for this first static version of the mechanism $s = n/2$ as the splitting point. The necessary operations work as follows:

- **Initialization:** During initialization all I_i are added to L_B. Then we start the *refill* process as shown in Algorithm 1.
- **Retrieve and Remove**: $minTimer()$ and *remove*(I_i) work as on the list-based mechanism (see Section III-A) and return resp. remove the head of L_O.
- **Insert:** Whenever a new I_i should be inserted into L, I_i is added either to L_B or L_O depending on its e_i as shown in Algorithm 1 where `ListInsert` is the inserting method from the list-based mechanism. As we see there, I_i is only inserted into a non empty L_O, and therefore the insertion only generates effort, if its e_i is below e_k of the head element I_k of L_B (caused by sorting on refilling L_O this is the minimum of L_B), or when L_B stays empty after insertion caused by the refill process.

There are two corner-cases handled as follows:

- **Overflow (Line 8):** It is crucial for this mechanism, that $\forall I_i \in L_O, \forall I_k \in L_B : e_i < e_k$. Therefore we have to move more than s timer-items I_i from L_B to L_O as long as e_i of the last moved I_i equals e_k of the current head item I_k of L_B.

Algorithm 1 Bucket of Ignorance

```
1: Refill $L_O$:
2: MergeSort($L_B$)
3: for x=0, x<s, x++ do
4:     $e_{min} \leftarrow L_B$.head.getExpireTime()
5:     Append $L_B$.head to $L_O$
6:     Remove $L_B$.head
7: end for
8: for y=0, y<s, y++ do
9:     if $e_{min} == L_B$.head.getExpireTime() then
10:        Append $L_B$.head to $L_O$
11:        Remove $L_B$.head
12:    else
13:        break
14:    end if
15: end for

16: Insert $I_i$ in $L$:
17: if $L_B$.isEmpty() then
18:     Append $I_i$ to $L_B$
19: else
20:     if $L_O$.isEmpty() then
21:         if $e_i < L_B$.head.getExpireTime() then
22:             Append $I_i$ to $L_O$
23:         else
24:             Append $I_i$ to $L_B$
25:             Refill($L_O$)
26:         end if
27:     else
28:         if $e_i < L_B$.head.getExpireTime() then
29:             ListInsert($I_i$, $L_O$)
30:         else
31:             Append $I_i$ to $L_B$
32:         end if
33:     end if
34: end if
```

- **Hyperperiod (Line 17):** If $t_k = lcm(T_i : \forall \tau_i \in \mathcal{T})$ (hyperperiod of $\mathcal{T}$) L_B becomes empty after refilling L_O due overflow. Therefore we put the next incoming I_j directly to L_B since $\forall I_k \in L_O : e_j > e_k$.

The saved effort by just adding new I_i to L_B accordingly surpasses the additional effort for the decision where to add new timer-items and sorting L_B according to task sets with periods. Please note that the *refill* procedure only runs during the *insert* procedure, while all tasks are blocked already (see Figure 1). Since a new timer is always higher than the current tick, so no context switch can happen during the procedure.

V. Simulation Framework and Setup

In this section we first describe the in-house analysis framework and the setup we use for evaluation. Afterwards, we present the results in terms of computation effort over different type of timer-managers.

A. Analysis Framework

To compare and test different timer-managers with the scope of the computation effort needed by a real system, we develop an in-house analysis framework in Java. The framework allows to simply implement new timer-managers through an interface class, manually define that the effort a real system needs and test it with different kinds of periodic task sets, e.g., incremental, harmonic, random, etc. As implementation on a real system differs from simulation, we manually define the effort a certain operation needs E. At the moment the framework allows to count comparisons, as these are the most time consuming operations. As the framework is not a crucial part of this paper, please refer to README.md and the commented code on our repository [6].

B. Simulation Setup and Preliminary Evaluation

To analyze the scalability, the cardinality of the task sets ranged in $\{8, 16, ..., 248, 256\}$. For each cardinality, we evaluated 10^5 randomly generated task sets for a duration of 200 ticks. The periods of those tasks were generated according to a log-uniform distribution with two orders of magnitude over the interval $[1, 100]$, as suggested by Emberson et al. [7]. This configuration ensures, that all tasks must be processed at least twice. Due to the space constraints, we focus on the most important part of the analysis. All the configurations and results are available on the repository together with the analysis framework [6]. We compare BoI with the list-based mechanism and the red-black tree-based mechanism.

TABLE I
SIMULATION RESULTS

Size n	**List**(n)	**Tree**(n)	**BoI**(n)	$D_{BoI}(n)$
8	537	$3,260$	705	-31.28%
16	$2,292$	$7,949$	$2,099$	8.42%
32	$9,450$	$17,608$	$6,026$	36.23%
64	$38,044$	$37,331$	$17,198$	53.93%
128	$151,918$	$76,681$	$52,633$	31.36%
208	$400,565$	$125,004$	$121,775$	2.58%
212	$431,899$	$129,863$	$129,950$	-0.07%
256	$606,541$	$153,759$	$174,687$	-13.61%

Table I shows the average amount of comparisons for each size n that was done during timer-simulations. $D_{BoI}(n) = 1-(\frac{BoI(n)}{\min(List(n),Tree(n))})$ is the percentage of amount BoI needs less than the minimum of both existing list or tree-based mechanisms. Figure 3 shows clearly, that the tree-based and BoI mechanisms are more efficient than list-based the larger n gets. Additionally the results show that BoI performs faster than the tree-based mechanism up to $n = 208$ in the average case and up to $n = 72$, when comparing best result of tree-based with worst of BoI. With growing n BoI suffers from two problems: 1) As we define s as half of n, L_O gets longer and therefore the effort for inserting new I_i into L_O grows as using a list-based mechanism (see Section III-A). 2) As we generate the task sets with periods from the interval $[1, 100]$, we get more tasks with identical periods that causes a higher occurrence of overflow corner-case (see Section IV). We might face these problems at high n as follows: 1) For task sets with a high amount of tasks with identical periods we insert sub-lists of I_i into L_0 that holds timers with equal e_i and 2) for task sets with a low amount of tasks with identical periods we use a tree instead of a list on L_O.

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose a new hybrid solution that requires less computation effort than conventional list-based and tree-based method under certain conditions. Our approach may

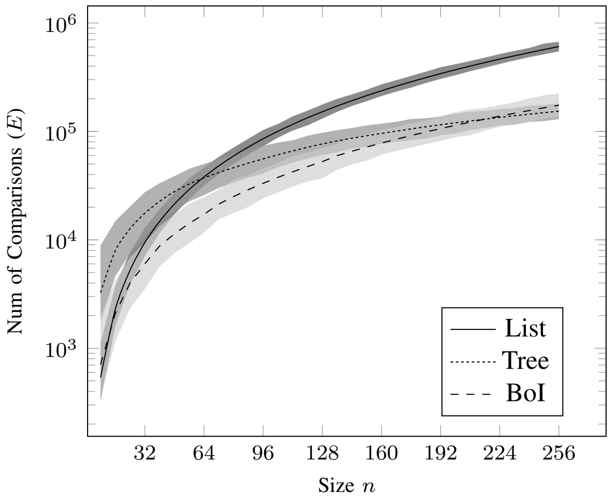

Fig. 3. Number of comparisons over different size n of task sets as arithmetic mean, where the areas around the plots covers all results for each n. Note that the y-axis is in logarithmic scale.

result in significant improvement on handling RILO-Queues. With the preliminary results presented in this work, we plan to continue working on this topic with the following ideas:

- Processing the refilling L_O from L_B by a normal task instead of during insertion could be an interesting idea. We plan to analyze the worst case formally to calculate the maximum possible period of this task such that L_O never runs empty.
- To understand the behaviour of the proposed mechanism in the average case, we are going to improve and optimize the framework to allow the simulation of larger sized task sets and measure the effort at a finer granularity.

ACKNOWLEDGEMENT

This paper is supported by Deutsche Forschungsgemeinshaft (DFG), as part of the Collaborative Research Center SFB 876 with Subproject A1 (http://sfb876.tu-dortmund.de).

REFERENCES

[1] Amazon Web Services. Freertos user guide. https://docs.aws.amazon.com/freertos/latest/userguide/freertos-ug.pdf. Accessed: March, 2020.

[2] G. C. Buttazzo. Rate monotonic vs. edf: Judgment day. In R. Alur and I. Lee, editors, *Embedded Software*, pages 67–83, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[3] K.-H. Chen, G. Von Der Brüggen, and J.-J. Chen. Overrun Handling for Mixed-Criticality Support in RTEMS. In *WMC 2016*, Proceedings of WMC 2016, Porto, Portugal, Nov. 2016.

[4] R. I. Davis. A review of fixed priority and edf scheduling for hard real-time uniprocessor systems. *SIGBED Rev.*, 11(1), 2014.

[5] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, 1974.

[6] M. Ebbrecht and K.-H. Chen. RTMCT - RTOS Timer Manager Comparison Toolkit. https://github.com/marcelebbrecht/timerComparision. branch rtas20, Accessed: March, 2020.

[7] P. Emberson, R. Stafford, and R. Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS workshop at the Euromicro Conference on Real-Time Systems*, pages 6–11, 7 2010.

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):4661, Jan. 1973.

[9] RTEMS Documentation Project. Rtems classic api guide. https://docs.rtems.org/branches/master/c-user/scheduling_concepts.html. Accessed: March, 2020.