Slovak University of Technology in Bratislava

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Hardware Architectures of Real-Time Kernels

Hardvérové architektúry jadier operačných systémov reálneho času

Dissertation Thesis Abstract

to obtain the academic title *Philosophiae doctor(PhD.)*

Author:	Ing. Lukáš Kohútka
Study programme:	Electronics and Photonics
Study field:	5.2.13 Electronics
Registration number:	FEI-104404-5813

Place and Date: Bratislava, 31. 7. 2018

Slovak University of Technology in Bratislava

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Hardware Architectures of Real-Time Kernels

Hardvérové architektúry jadier operačných systémov reálneho času

Dissertation Thesis Abstract

to obtain the academic title *Philosophiae doctor(PhD.)*

Author:	Ing. Lukáš Kohútka
Study programme:	Electronics and Photonics
Study field:	5.2.13 Electronics
Registration number:	FEI-104404-5813

Place and Date: Bratislava, 31. 7. 2018

Dissertation Thesis has been carried out at the Institute of Electronics and Photonics, Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava.

Author: Ing. Lukáš Kohútka
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology in Bratislava
Ilkovičova 3, 812 19 Bratislava, Slovak Republic

Supervisor: prof. Ing. Viera Stopjaková, PhD.
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology in Bratislava
Ilkovičova 3, 812 19 Bratislava, Slovak Republic

Reviewers: prof. Ing. Lukáš Sekanina, Ph.D.
Faculty of Information Technology
Brno University of Technology
Božetěchova 1/2, 612 66 Brno, Czech Republic

doc. Ing. Miloš Drutarovský, CSc.
Faculty of Electrical Engineering and Information Technology
Technical University of Košice
Letná 9, 042 00 Košice, Slovak Republic

Dissertation Thesis Abstract was sent: ....
(Date of sending)

Dissertation Thesis Defence will be held on August 22, 2018 **at** 11 a.m.

At: Faculty of Electrical Engineering and Information Technology
Slovak University of Technology in Bratislava
Ilkovičova 3, 812 19 Bratislava, Slovak Republic
in room ....

prof. Dr. Ing. Miloš Oravec
Dean of the Faculty of Electrical Engineering and Information Technology,
Slovak University of Technology in Bratislava

Table of Contents

1 Introduction

The research analysed and presented in this thesis is focused on hardware acceleration of algorithms used in real-time systems, especially algorithms that are used in real-time operating systems. Hardware architectures of such algorithms have higher impact on real-time systems than the acceleration of just specific applications belonging to the domain of real-time systems. Performance of whole real-time systems can be improved by improving the performance of real-time operating systems they are using. Nevertheless, the worst-case timing, predictable behaviour and low response time are much more important than the average performance in real-time systems [1]. Hardware acceleration is even more beneficial approach for real-time systems because by applying hardware acceleration, asymptotic time complexity of the accelerated algorithm can be reduced. In some cases, the complexity can be reduced even down to a constant time, thus the latency of the chosen algorithm does not change with respect to its parameters and with respect to the amount of data the algorithm is processing. The constant time complexity is one of the most important aspects of algorithms used in real-time systems because it improves the determinism and predictability of the whole system [2].

There are already several existing solutions focused on hardware acceleration of algorithms in real-time operating systems. However, they have not been very successful in being adopted by companies in real world because of few reasons. The problems of existing solutions include mainly lack of flexibility, reusability and efficiency. The main purpose of the research conducted and presented in this work is to discover new solutions that will solve these problems by designing more robust, flexible and efficient solutions with constant response time. This could make hardware acceleration more successful and popular, especially for real-time systems. Thanks to this, new real-time systems based on such new solutions could be more deterministic and provide higher performance with lower response time for a reasonable cost. Additionally, such systems could be more intelligent and robust, either in terms of the number of tasks performed by the systems or in terms of how complex these tasks are.

2 Background

A real-time system is a special class of embedded systems. Real-time systems are based on real-time paradigm, which defines the systems timing. The real-time paradigm is saying that the quality of real-time outputs depends not only on the correctness of this output but also on the time when is the output is produced. This means that producing an output of the real-time systems too late can cause the same issues as producing incorrect output [3].

Recently, there was highlighted a connection between real-time embedded systems and the physical environment by creating a new term called *cyber-physical systems*. These systems are specified as an integration of computations and physical processes [4]. This new term is highlighting a connection to physical parameters like time, energy and space. For such systems, one can expect new models representing physical environment. Cyber-physical systems can contain an embedded system (the part responsible for information processing) and physical environment [5, 6].

According to the consequences of failing to meet a deadline, real-time systems can be divided to the following two categories [3, 7]:

- Soft real-time systems – occasional missing of deadlines is allowed, however with increasing the amount of occasions when a deadline is not met, the overall quality of the real-time system can be degraded.
- Hard real-time systems – these systems are also called *safety-critical systems*, thus missing a single deadline can result in a failure of the whole system, which can cause serious problems.

There are misconceptions for engineers when it comes to real-time computing paradigm. Some engineers believe that in order to meet the timing requirements of real-time systems, it is sufficient to just make the system fast enough. However, the system speed refers to the average performance of the system. In order to guarantee that the real-time system meets all deadlines in any cases, one must deal with the worst-case timing scenario. Thus, while the average execution time is important for conventional systems, the worst-case execution time is important for real-time systems. In order to be able to guarantee that all deadlines are met even with the worst-case timing, the whole real-time system must be adjusted to be more deterministic and predictable – application software, system software (operating system, firmware, drivers) and the last but not least, the hardware too [3, 8]. The most deterministic and predictable algorithms are such algorithms that have constant time complexity O(1). Constant time complexity means that the computation of the algorithm takes the same amount of time regardless of any parameter of the algorithm [9].

The current trend in real-time embedded systems is that these systems are becoming more complex, require higher performance and determinism. A development time is increasing significantly as well. Therefore, it is necessary to contribute to creation of a new hardware platform of real-time systems that will be efficient, universal and deterministic.

To guarantee determinism and completion of real-time tasks on time, and efficient and robust task scheduling is necessary. The biggest issue of the analysed scheduling algorithms is that either they are fast to execute but too simple for generation of efficient schedules, or they are complex enough for generation of optimum schedules but consume too much CPU time to schedule the tasks, which reduces the remaining time that can be used for execution of the scheduled tasks. Hardware acceleration is a possible solution that allows operating systems to use more complex, intelligent and effective scheduling algorithms, while the time complexity of their execution remains constant and reasonably low. It was also discovered that vast majority of the analysed task scheduling algorithms are focusing on one or few types of tasks only. However, systems in real world usually consist of various combinations of these task types and there is a trend of increasing complexity of real-time systems, including increasing the number of task types that are combined in the same system. Thus, more complex scheduling algorithms are becoming more beneficial and needed. Current real-time operating systems are using simple scheduling algorithms because since those can be implemented in software with constant time complexity. Due to an increasing need for adoption of more complex scheduling algorithms, an efficient and flexible hardware architecture of these algorithms is needed [3, 10, 11, 12, 13].

Hardware-accelerated task schedulers need to use an architecture of data sorting for realization of ready queues. Each architecture has some advantages and disadvantages that also significantly affect the attributes of algorithms that are using such architecture. The major attributes are: resource cost (chip area for ASIC or LUTs consumption for FPGA), power consumption, determinism, critical path length / clock frequency, latency and throughput (in clock cycles). In order to design efficient hardware architecture of real-time task scheduler, an efficient and scalable min/max queue in terms of resource costs with constant critical path length, constant latency and constant throughput is needed. Therefore, if new hardware architecture of data sorting with better attributes is developed, it would increase the efficiency of the whole system that is based on such data sorting architecture as well. Among the analysed existing architectures, the Systolic Array architecture provides the best results so far. To design an efficient hardware architecture of real-time task scheduler, a new min/max queue architecture with constant critical path length, constant latency, constant throughput and minimum resource costs is needed. By reducing the resource costs and keeping the constant timing of the min/max queue, the maximum number of tasks present in real-time systems can be increased as long as a proper scheduler is adopted. The higher number of tasks allows to develop more complex real-time systems with additional functionality [10, 14, 15, 16].

3 Research Objectives

Considering the background of real-time task scheduling and their hardware acceleration including the hardware architectures of data sorting that can be used for task scheduling, the following research objectives were defined.

- Analysis, implementation and evaluation of existing hardware sorting architectures and their efficiency through comparison of their synthesis results.
- Design of new hardware architecture for data sorting in real-time systems in order to minimize their logic/area cost and power consumption.
- Design of new hardware architecture for task scheduling in multiple-core real-time systems in order to increase the number of tasks that meet their deadlines.
- Extension of the task scheduling hardware for mixed-critical real-time systems in order to increase their flexibility and robustness.
- Verification of designed architectures, comparison with existing solutions and evaluation of their benefits.

4 Proposed Efficient Min/Max Queue Architectures

Since most of the scheduling algorithms suitable for real-time embedded systems need to use sorting in a form of min/max queues to find the most suitable task to be executed, this chapter is focused on design of new min/max queue architectures that are supposed to be more efficient in terms of resource costs and energy consumption than the existing architectures. Thus, this chapter is focused on fulfilling the research objective [RO-2].

4.1 Rocket Queue

The Rocket Queue is a new min/max queue that contains items to be sorted. These items consist of two values only [A8, A10]:

- ID – unique identification number (unique number for each item present). The only exception is 0 that is used for representation of empty items (i.e. no data).
- DATA – the items should be sorted according to this value. If the queue is a min queue, then the output of the queue is the item with the lowest value of DATA. If the queue is a max queue, then the output of the queue is the item with the highest value of DATA.

The Rocket Queue is a min/max queue in a form of an IP core that can be on-chip implemented in a form of a coprocessor. The coprocessor extends an instruction set of existing CPUs with its custom instructions of the coprocessor. The coprocessor implements these two instructions [A8, A10]:

- INSERT – one new item is provided by the CPU, and this item is inserted into the min/max queue. The insertion is performed in such a manner that the items in the queue remain sorted so that the item with lowest/highest DATA remains to be the output of the min/max queue.
- REMOVE – one existing item is removed from the queue depending on the ID that is provided alongside with this instruction. This is performed regardless of the position of the item that is being removed. The item is removed from the queue in such a manner that the items in the queue remain sorted so that the item with lowest/highest DATA remains to be the output of the min/max queue.

One of the most resource consuming parts of the queues is a comparator. Comparison is performed in each cell of the Systolic Array architecture as well as in Shift Registers. By reducing the number of comparators used within the queue, significant portion of resources can be saved. Therefore, a new architecture called Rocket Queue was developed [A8, A10].

The Rocket Queue architecture is using less than one comparator per cell in average. This is accomplished by sharing one comparator within several cells. The sharing of comparators is achieved by using multiplexers. The architecture is organized in levels, where each level consists of several cells that share the same comparator. The first few levels are called duplicating levels because each level below the duplicating level contains two times more cells. The duplication of cells per level stops at some point, which is defined by the number of duplicating levels. The levels below the group of duplicating levels are called merged levels and they keep the same number of cells per level for the rest of the architecture. Figure 1 depicts an example of the Rocket Queue architecture, where 3 duplicating levels and 11 merged levels are used. Each node represents memory storage for one item, while the connections between the nodes represent possible movements of items between the nodes that can occur due to adding or removing of an item [A8, A10].

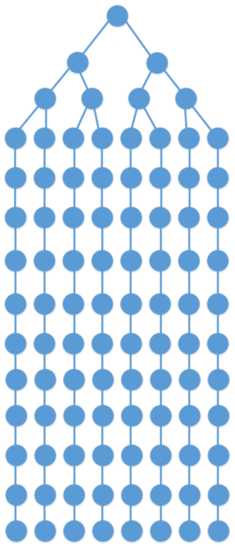
Figure 1: Example of Rocket Queue architecture [A8, A10]

There are two disadvantages caused by merging comparators to one common comparator within each level. The first one is that the multiplexers mentioned above must be inserted so that the comparator can be shared by more cells. This increases resource costs and critical path length; therefore, the number of duplicating levels is limited. The second drawback is that the Rocket Queue architecture also needs to add a counter to each cell for decisions, in which direction to propagate the instructions that are adding new items to the queue (i.e. insert instructions). All instructions start at the first level that is located on the top of the queue and are propagated down at a speed of one level per clock cycle. Only one cell is actively used by the insert instruction in each line, which is controlled by the counters. The remove instructions are checking ID values of all cells within the current level [A8, A10].

The INSERT instructions are working in the following way. The instruction starts at the first level from the top and goes one level down per each clock cycle. Each time the INSERT instruction is executed in the corresponding level, the provided sort data is compared to the internal sort data of one selected cell in the level. Only one cell is selected for the instruction within a level. The selection of a cell within the level is realized with respect to the total amount of items below. Each cell situated in duplicating levels has two successors, the left one and the right one. If there are fewer items in the left successor than in the right successor, then the INSERT instruction continues to the left successor, otherwise it continues to the right one. Each cell contains one counter that contains total number of items in the subtree represented by this cell. The subtree of a cell is set of all successors of the selected cell, including successors of the successors, and including the selected cell itself. Counters are immediately updated whenever an INSERT instruction is executed in particular level, even if the provided item is not yet inserted in this level [A8, A10].

An example of the INSERT instruction is shown in Figure 2. The instruction is inserting an item with DATA = 17. In the first step, the instruction is executed in level 1, thus the number 17 is compared to the number 15. Since this example is a min queue and the new item has a higher sort value, the new item continues to the below level. The new item with the instruction continues to the right successor because it has fewer items in total (2 items, while the left successor has 3 items). The step 3 is again a comparison of the input sort value with the internal value in the cell. This time, the internal value is higher, so a replacement of items occurs. The replaced old item is removed from the cell and continues with the INSERT instruction further below. Step 4 is again a decision whether the instruction goes to the left successor or to the right successor. There are zero items to the left and one item to the right, thus the instruction continues to the left successor. Since this cell is free, the item is inserted there. Steps 1 and 2 are executed in the same clock cycle as well as steps 3 and 4 [A8, A10].

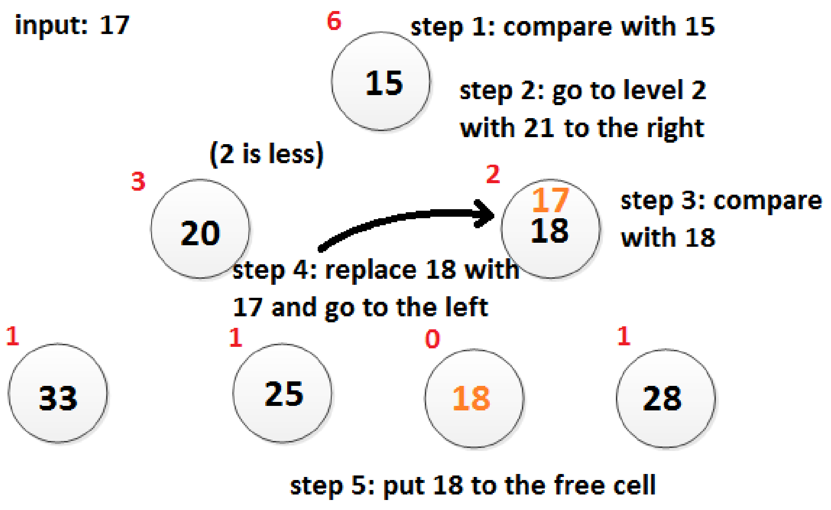

Figure 2: Example of Rocket Queue architecture [A8, A10]

The REMOVE instructions are propagated to all cells within the same level simultaneously, thus IDs of all items located at the same level are compared to the provided ID in one clock cycle. Whenever any of these items has the same ID as the ID provided by the instruction (i.e. the item to be removed is identified), the item is removed from the corresponding cell by overwriting the contents of the cell registers with the item of a selected successor and the same operation is propagated to all items under the removed item (i.e. the items located in the same column are moved up) [A8, A10].

4.2 Heap Queue

The proposed Heap Queue architecture is layered into levels similar way as the Rocket Queue architecture. The difference is that while the Rocket Queue architecture consists of duplicating levels and merged levels, the levels used in the Heap Queue architecture are all duplicating ones. This means that each level is duplicating the number of items that can be stored within such level. For example, the first level has capacity of one item, the second level two items, the third level four items, the fourth level eight items, and the next level contains sixteen items [A14, A17].

Another difference between Rocket Queue and Heap Queue is that the Rocket Queue architecture is storing all data into registers, which is not true for the Heap Queue architecture. Items that are inserted into Heap Queue are stored in dual-port random access memories. Similarly, the numbers used for tree balancing are stored in RAM instead of registers too. The tree balancing is a feature that was developed for Rocket Queue in order to ensure that whenever a new item is inserted into the queue, the items are reorganized in such a manner that the tree of filled cells (i.e. the cells filled with an item) is balanced. The tree balancing feature is very common for binary trees in theoretical informatics too [A14, A17].

Figure 3 depicts the Heap Queue architecture layered into levels. Each level consists of one Control Unit (CU) and various number of Item Storage units (IS). One IS unit can be used for preserving of one item and one number that is used for the tree balancing feature. Each subsequent level contains two times more IS units than the previous one. The Control Units communicate with other Control Units from neighbouring levels. This communication is used for propagation of instruction from upper levels below and for items exchanges between levels. Since one Control Unit manages several IS units, the selection of IS unit is performed according to address provided by the Control Unit. The first three levels use registers for implementation of the IS units due to too small memory sizes. All the other levels are using dual-port RAM memories for implementation of IS units. The Control Unit of the first level serves as an interface of the whole queue to the external environment. It provides the first item (stored in the IS of level 1) as an output, which represents the item with the minimum/maximum sorting value among all items inserted into the queue [A14, A17].

One can also notice that the Control Unit performs a combination of several Sorting Nodes employed in heapsort algorithm. The reason is that each Sorting Unit would require to instantiate its own comparator, and since comparators are relatively resource expensive, the merging of several Sorting Units into one Control Unit saves significant portion of combinational logic within the queue [A14, A17].

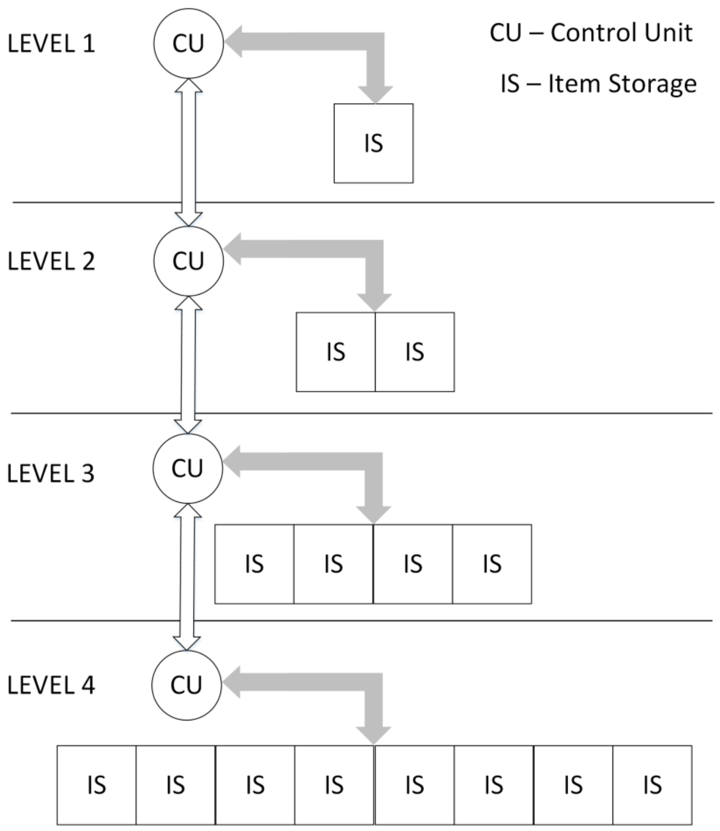

Figure 3: Heap Queue architecture example with 4 levels [A14, A17]

Since the Heap Queue architecture is using RAM memory instead of registers, it is no longer possible to find and remove any item within the queue in a reasonable (and constant) time. The reason is that with registers, the Rocket Queue architecture is able to read all registers within the same level simultaneously in one clock cycle [A8, A10, A13]. However, with RAM memory it would be needed to sequentially read the memory one item per clock cycle. Therefore, the Heap Queue architecture allows removing only the first item (at the top of the queue) from the queue. Depending on the application of the min/max queue, this limitation may be acceptable or not. Thus, the Heap Queue architecture is proposed only for those applications, where only the item with the min/max sorting values are needed to be removed (e.g. scheduling of hard real-time tasks and Dijkstra's algorithm). For other applications, the Rocket Queue architecture remains to be the optimum solution for hardware acceleration (e.g. Worst-fit memory allocation algorithm) [A14, A17].

The Heap Queue architecture is very similar to the DP RAM Heapsort architecture too [4]. However, the major difference between these two architectures is that the DP RAM Heapsort architecture lacks any tree balancing techniques, which can cause data overflows if items are inserted without simultaneous item removals. The Heap Queue has reused the tree balancing technique from the Rocket Queue architecture [A8, A10, A13]. Thus, the Heap Queue architecture represents a combination of Rocket Queue and DP RAM Heapsort architectures into a novel architecture that uses the best advantages from both former architectures [A14, A17].

Since the DP RAM Heapsort architecture is performing both instructions POP and INSERT simultaneously [4], the Heap Queue architecture has adopted this feature too. It is provided in a form of a fourth instruction called POP_INSERT, which is identified by opcode 11. This instruction can be used to further improve the performance of the application that using this min/max queue by reducing the number of instruction calls [A14, A17].

4.3 Usage of MIN/MAX Queue for Task Scheduling

Task scheduling as the main part of operating systems is responsible for deciding which task (i.e. process or thread) is running and executed by CPU in what time. These decisions highly depend on the algorithm that is used for the scheduling. While ordinary operating systems usually schedule tasks according to their priorities, real-time systems should create the schedule according to the deadlines of the tasks, because it is critically important to meet the deadlines of all hard-RT tasks. One of the most common and popular algorithm used for scheduling of hard-RT tasks is the Earliest-Deadline First (EDF) algorithm, which simply sorts all tasks according to their deadlines so that the task with the earliest deadline (i.e. with the lowest deadline value) is selected for execution. Therefore, the min/max queues are essential for implementation of the EDF algorithm. By improving the parameters (e.g. resource cost, power consumption or latency) of min/max queue, an EDF scheduler implemented by the selected min/max queue is improved as well [A13].

Although the proposed Rocket Queue and Heap Queue architectures are suitable for implementation of EDF scheduler for hard real-time systems, these min/max queue architectures can be also used for any other real-time application that needs to use data sorting or min/max queues as a part of its algorithm.

5 Proposed Real-Time Task Schedulers

Existing research papers are assuming that real-time systems consist of hard real-time tasks only and there is no mixing of task criticality needed (e.g. real-time and non-real-time tasks within the same scheduler). Thus, the existing solutions are forced to use hypervisors and split computing power to separate subsystems – operating systems managed by the hypervisor. However, such solutions are limiting, less efficient and harder to adopt. Therefore, new scheduler architectures that can handle various tasks of different criticality are proposed. This chapter is focused on fulfilling the research objective [RO-4].

5.1 EDF with Priority based FCFS

The first proposed solution is based on a combination of EDF algorithm that is suitable for hard real-time tasks and priority based FCFS scheduling algorithm that is used for scheduling of non-real-time tasks. There is also a task killing feature available that is important for making the algorithm extensible by any software extensions for the scheduling (e.g. EDF can be extended to GED by software). The combination of EDF and priority based FCFS scheduling algorithms is performed in the following way. The implementation of EDF algorithm is sorting the tasks according to their deadline values. Non real-time tasks can be added to the EDF algorithm only when the priorities of these tasks are represented as deadlines of the tasks used in the EDF implementation. Thus, the same sorting (min/max) queue is used for and shared by both algorithms. In addition to that, in order to ensure that non-real-time tasks can never cause any deadline miss of any hard real-time task, the hard real-time tasks shall always have lower deadline value than the priority value of non real-time tasks. In order to fulfil such a requirement, the top 1024 deadline values are reserved for representation of priorities for the non-real-time tasks, while the rest of the possible deadline values remain used for hard real-time tasks [A2, A5, A6].

The proposed scheduler implements these two instructions [A5, A6]:

- Task ADD – adding of a new task to the scheduler. The scheduler automatically updates the schedule and selects the new task to be executed.
- Task KILL – removing of an existing task from the scheduler according to task ID.

For sorting of the tasks Shift Registers was used. The amount of pipeline stages as well as many other parameters can be easily changed by setting a new value for corresponding constant in the code. Thus, the developed accelerators are designed to be as much parameterized as possible [A5, A6].

Depending on whether the task killing feature is required by the system, this scheduler can be directly implemented by using either Rocket Queue (with killing) or Heap Queue (without killing) architecture for sorting of the tasks [A13, A17].

5.2 GED based Scheduler

While EDF algorithm accepts all requests for adding a new task, the GED algorithm is using the Guarantee Routine for deciding whether to accept the request and add the new task to the Ready queue or to reject the request and discard the new task. By doing this, it is guaranteed that all deadlines of accepted tasks will be met in all cases. So far, there is no known hardware implementation of the GED algorithm, while the software implementation of GED is not practical due to the high amount of execution time that is required every time a new task is scheduled [A16].

The implementation of Guarantee Routine checks whether the new task is going to cause system overload. System overload is a state when not all scheduled tasks (i.e. tasks stored in the Ready queue) are guaranteed to meet their deadline. If the new task is going to get the system into "system overload" state, then the Guarantee Routine rejects the task. Otherwise the new task is accepted and added to the Ready Queue. This feature is especially useful for periodic soft real-time tasks, where occasional rejection of the task leads to higher average amount of successfully completed tasks on time. The problem of accepting all tasks all the time without checking whether the system is in "system overload" state is that a domino effect can occur [A16].

The Guarantee Routine feature is the only difference between the EDF and GED algorithms. Thus, by implementing the Guarantee Routine, one can extend an existing EDF based scheduler to GED based scheduler [A16].

The implementation of GED algorithm can be extended by support of non real-time tasks by adding the priority-based FCFS algorithm into the scheduler the same way as in case of the EDF algorithm, which was presented in section 6.1. Thanks to this, the proposed GED based scheduler can handle soft real-time tasks and non-real-time tasks, and any combinations of these tasks within the same scheduler [A16].

5.3 RED based Scheduler

The RED algorithm is an extension of the GED algorithm. This algorithm is predicting whether any task is going to miss its deadline and selects the most appropriate task among the ready tasks to be temporarily placed into the Reject queue (i.e. rejected), which is depicted in Figure 4. In this way, it is guaranteed that all hard real-time tasks will meet their deadlines and also as many soft real-time tasks will meet their deadlines as possible. In the case when one of the scheduled tasks is completed sooner than expected, and there is enough time to reclaim and complete one of the rejected tasks, the algorithm uses reclaiming policy to return the rejected task back to the Ready queue containing the scheduled tasks [A15].

Prior to our research and as far as we know, no existing hardware implementation of task schedulers based either on GED or RED algorithm has been found and the main problem of the software implementations is that the scheduling consumes too much CPU time. Furthermore, the software implemented RED schedulers have variable duration of scheduling execution that depends on many various factors. Therefore, the software based RED schedulers are not deterministic and thus practically applicable to complex real-time systems [A15].

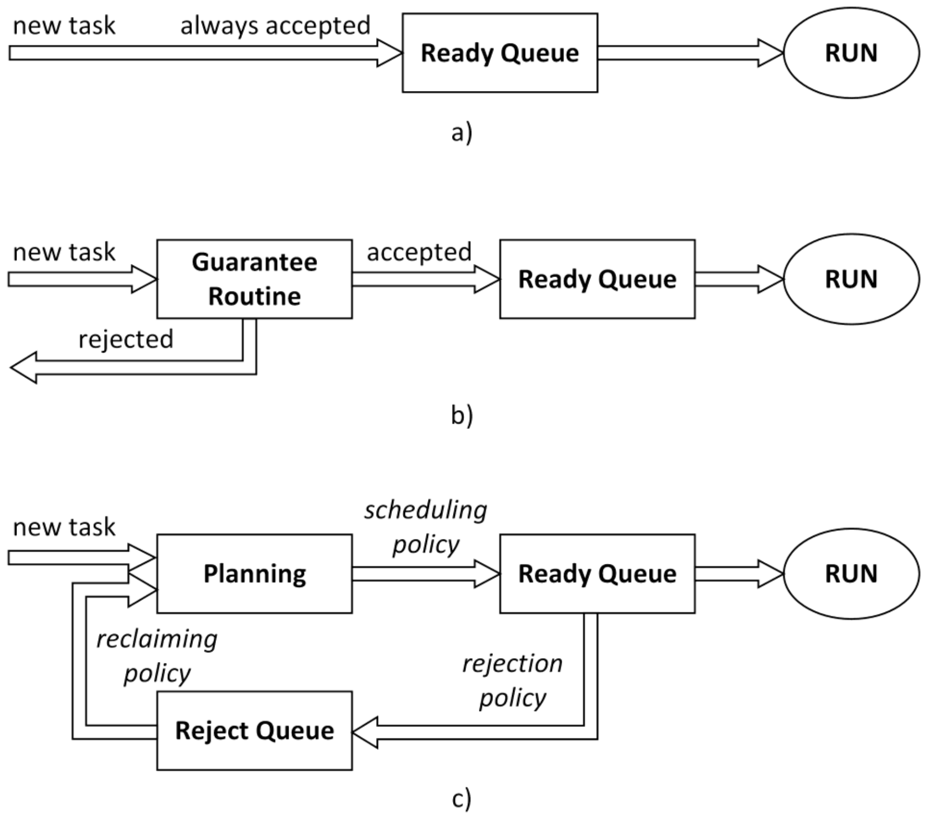

Figure 4: EDF (a), GED (b) and RED (c) algorithms. [A15]

The proposed RED based scheduler implements a special case of the RED algorithm, where one new constraint is added [A1]:

$$primary\ deadline\ =\ secondary\ deadline \tag{1}$$

While the RED algorithm itself does not define any policy for the task sacrificing to reduce the overload of the whole system, we also propose a method for selection of such tasks. This method selects the tasks according to 2 attributes, primary and secondary. The primary attribute is a type and priority of the task, where the task with the highest priority number is the best candidate to be sacrificed. The secondary attribute is considered in the case that there are multiple tasks with the highest priority. This attribute represents a contribution of the task to overload of the system. Task with the highest primary attribute and then, the highest secondary attribute should be chosen to be sacrificed. The term “sacrifice” represents a removing of the task from the queue of the scheduled tasks and adding the task to the list of rejected tasks, called rejection list. The list of rejected tasks exists in a hope that some of the tasks being executed will be completed sooner than expected, and the processor will be able to complete some of the rejected tasks before the deadline [A1].

The top module of the proposed scheduler represents a coprocessor unit and it is displayed in Fig. 5. This module is composed of three submodules: Ready Queue, Control Unit and Reject Queue. The input of the top module is INSTR, which represents the coprocessor instruction provided from the CPU that is connected to the coprocessor. The output TASK_TO_RUN represents the task that is chosen by the scheduler to be executed by the CPU at the moment [A15].

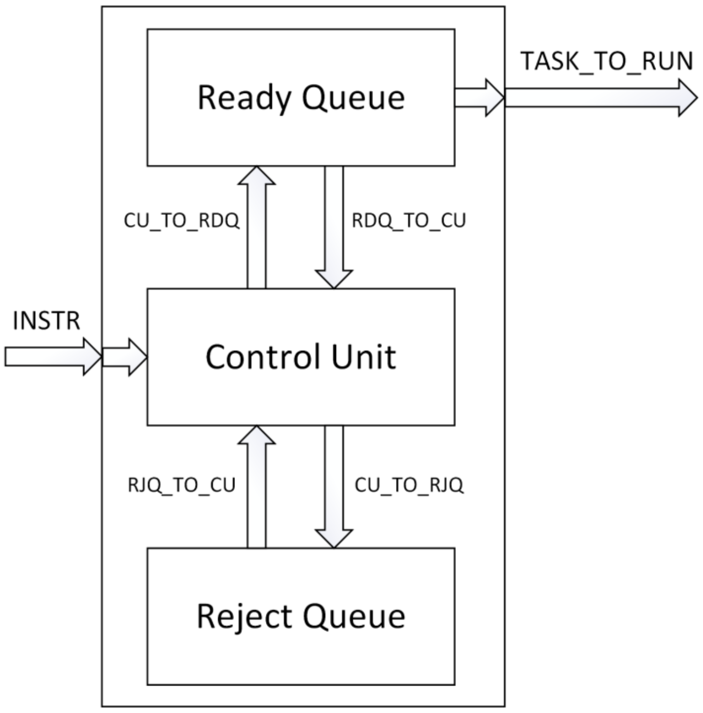

Figure 5: Top module of RED based scheduler [A15]

5.4 EDF for Dual-Core CPUs

The EDF based scheduler was consequently improved by adding a support for dual-core CPUs. With this extension, more powerful systems can be used as real-time systems. This allows finishing more tasks in the system before their deadlines are met. Thus, more time consuming real-time systems can be developed. Two novel architectures of task scheduling coprocessor unit for dual-core real-time embedded systems are proposed. In dual-core systems, a conflict can occur whenever both CPU cores attempt to use the coprocessor at the same clock period. Therefore, our research was focused on conflict-free solutions [A3, A4, A5, A6].

The first architecture is based on semaphore approach that solves conflicts whenever they occur by selecting one core as a winner and locking the scheduler for the selected core, while the other core is a loser that is stalled in meantime. Each core is stalled once per 2 conflicts because the selection of the winner and loser is always switched. This ensures fairness of the approach in any cases [A4, A5].

The second architecture is based on simultaneous processing approach that eliminates the possibility of a conflict occurrence. This is achieved by implementation of more complex task cells and interfaces between them. As a result, both CPU cores can use the coprocessor unit simultaneously without interfering each other [A3, A4, A5, A6].

This new architecture is shown in Figure 6, where an example of killing Task 1 is presented. The CPU core 1 stops executing Task 1 and starts executing Task 3 (task with the earliest deadline among the tasks in the Ready tasks queue), while the CPU core 2 continues executing Task 2 without any change [A3, A5, A6].

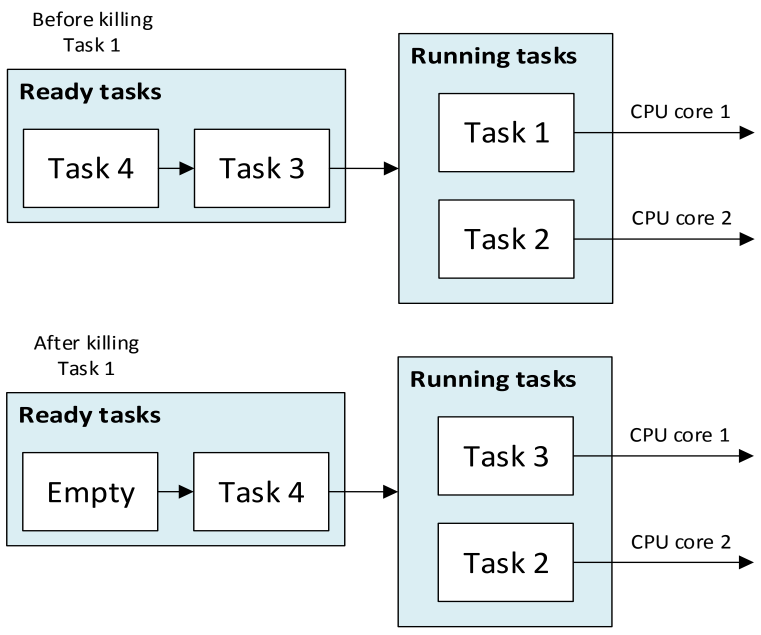

Figure 6: Running tasks out of the task queue (proper approach) – task killng [A3, A5, A6]

The Running tasks module contains control logic that is responsible for decision whether to keep actual tasks in the cells or to perform any change. Task with the earliest deadline among the ready tasks is inserted to the Running tasks module whenever one of the running tasks is killed. Whenever a new task is being added to the system, a situation called preemption can occur depending on the deadline of the new task and deadlines of the running tasks. If the new task has lower deadline than at least one of the running tasks, then this task should replace the running task with the highest deadline. The replaced task is called preempted task and execution of the preempted task should be paused. If preemption occurs then the Ready tasks queue receives the preempted task, otherwise the new task is inserted to the Ready tasks queue [A3, A5, A6].

The second example (illustrated in Figure 7) is a case when a new task (Task 4) is added to the system. The new task has lower deadline than the deadlines of the running tasks, and Task 1 has higher deadline than Task 2 and therefore preemption occurs. In the Running tasks queue, Task 1 is replaced by the new task - Task 4 and inserted into the beginning of the Ready tasks queue. In this way, we keep both CPU cores executing tasks with the earliest two deadlines while eliminating all unnecessary occurrences of task switching [A3, A5, A6].

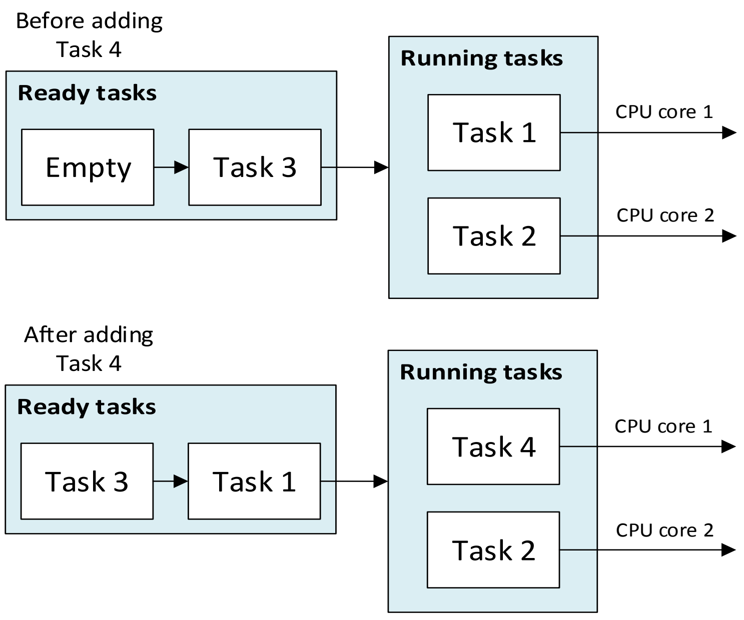

Figure 7: Running tasks out of the task queue (proper approach) – task adding [A3, A5, A6]

Simultaneity Conflict Solving

Nevertheless, another problem can also occur. What if both CPU cores decide to add a new task or kill an existing task at the exact same time (clock cycle)? Let us call such a situation a conflict. All previous approaches assumed that there is always one request at a time. This assumption is totally correct in single-core systems. However, the opposite is true in multi-core systems. There can be very low probability of such a situation. For example, if each CPU core uses custom instructions of the coprocessor 1% of the time (one instruction of the scheduling per 100 instructions) then the probability of a conflict occurrence is 0.01%. So, the conflict would occur very rarely but still it has to be considered. We propose two approaches to solve this problem and discuss their advantages and disadvantages [A3-A6].

The first proposed approach is simpler, and it is called the semaphore approach. If a conflict occurs (a situation when both CPU cores want to use the coprocessor at the same time), it is resolved dynamically by semaphore module. This module is responsible for choosing, which CPU core will be granted an immediate access to use the coprocessor. The second core will use the coprocessor after the instruction of the first CPU core is finished and thus, this second core needs to be stalled. The architecture proposed to implement the semaphore approach for handling of 2 CPU cores is depicted in Figure 8 [A3, A5, A6].

This approach can be also extended to support more than just 2 CPU cores, which is described in 5.5 EDF for Quad-Core CPUS.

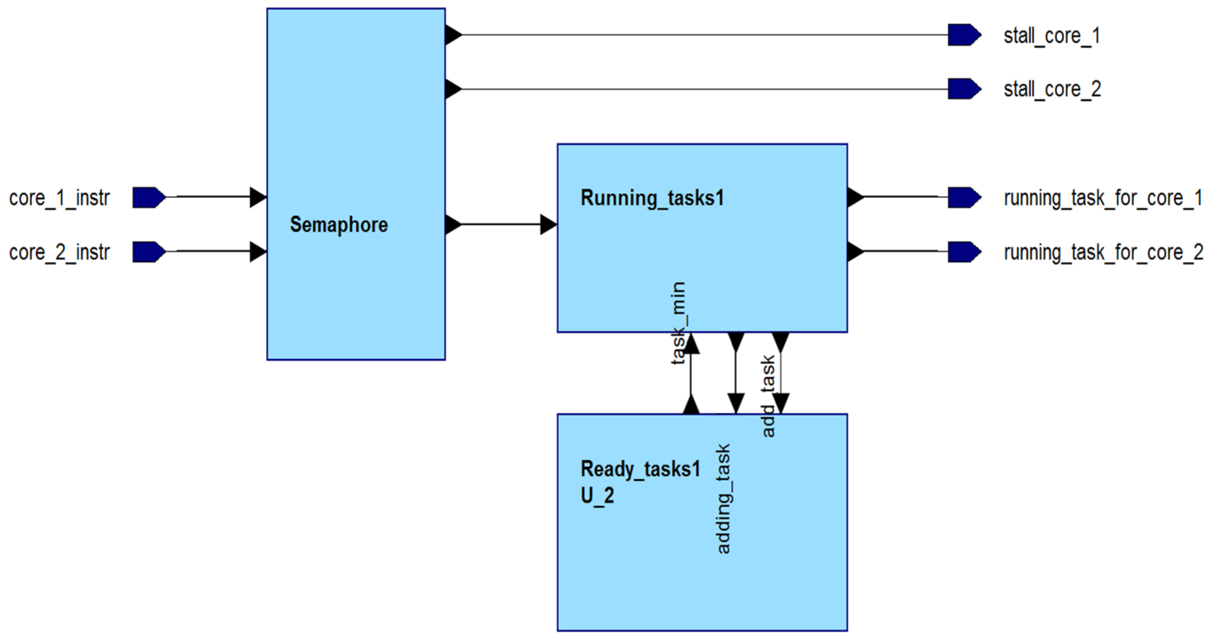

Figure 8: Semaphore approach for dual-core systems [A3, A5, A6]

The semaphore module consists mainly of multiplexers for selecting the instruction. In addition to that, there is only one D Flip-Flop added for remembering whether the last conflict-winning core was CPU core 1 or CPU core 2. In the case of a conflict occurrence, the multiplexer is also controlled by the output of this DFF. The best-case execution time is one clock cycle. However, we have to keep on mind that the main target is hard real-time systems, where the worst-case execution time should be taken in consideration instead of the best case or average execution time. The worst-case execution time of the semaphore approach is two clock cycles because in the worst-case scenario, a conflict occurs every clock cycle. The biggest disadvantage of the semaphore approach is the fact that the best-case execution time and the worst-case execution time differ, which means that such a coprocessor and also the whole real-time system would be less deterministic and predictable [A3, A5, A6].

The second proposed approach is no doubt much more complicated solution, and it is called the simultaneous processing approach. This approach works with modified Running_tasks and Ready_tasks modules that are able to accept and execute two instructions of the scheduling at the same clock cycle, without any conflicts. In order to achieve such functionality, we need to extend the interface of the modules, internal logic of the task cells in the Ready_tasks and internal logic of the Running_tasks component. The global architecture of the simultaneous processing approach is shown in Figure 9 [A3, A5, A6].

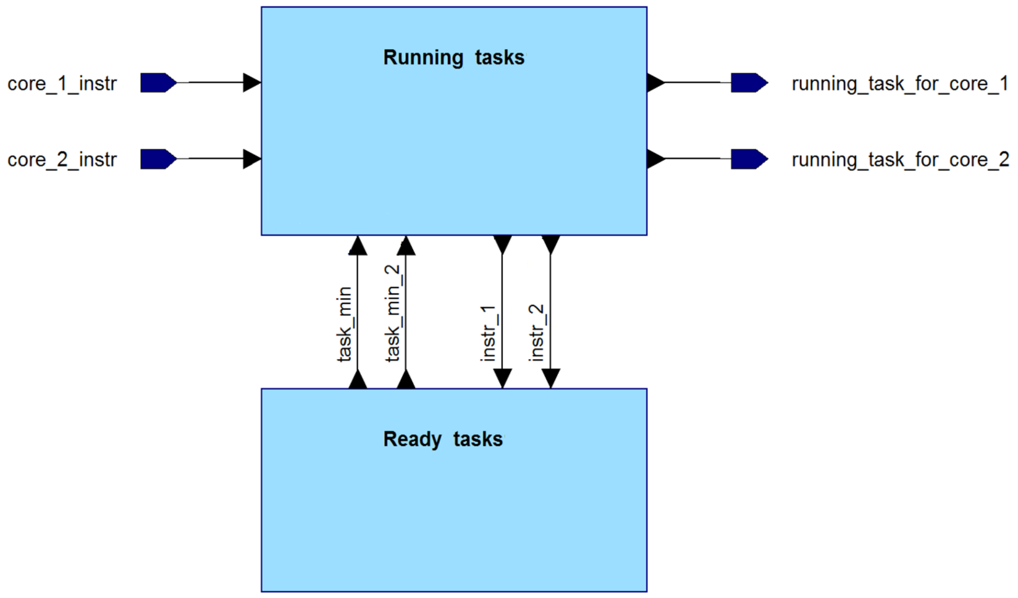

Figure 9: Simultaneous processing approach for dual-core systems [A3, A5, A6]

Both Ready_tasks and Running_tasks modules are receiving two instructions simultaneously. For one instruction input, there are two possible types of instructions: task_add or task_kill. When having two instruction inputs, there are four possible combinations of instruction types: task_add & task_add, task_add & task_kill, task_kill & task_add or task_kill & task_kill [A3, A5, A6].

5.5 EDF for Quad-Core CPUs

The top module of the quad-core version is consisting of the same four components as the dual-core version, which are [A9, A12]:

- Ready Tasks
- Running Tasks
- Semaphore
- Control Unit

The organization of these four blocks within the top module for dual-core version of the scheduler is described by a block diagram displayed in Figure 10. The instructions coming from CPU cores are at first being processed by the Semaphore unit that is responsible for deciding, which CPU core has granted access to add or remove a task at the moment, and which CPU core has to wait for two clock cycles. The instruction of the winning CPU core is being provided to the next component - Control Unit. Control unit generates specific control signals for another component (Running Tasks) according to the received instruction. The Running Tasks component is handling the tasks that are supposed to be executed at the moment. Communication with the Ready Tasks component is performed only through the Running Tasks component, which sends to the Ready Tasks component those tasks that are pre-empted, and which is reading the task with the lowest deadline value among the ready tasks (from the Ready Tasks). The control signals from Control Unit are also driven to the Ready Tasks only indirectly through the Running Tasks component [A9, A12].

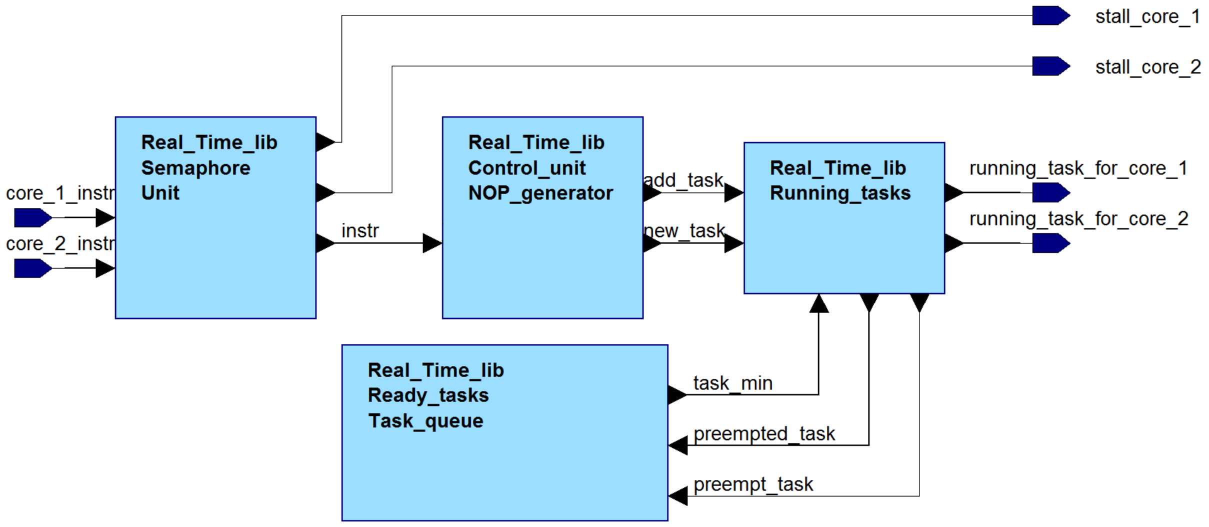

Figure 10: Block diagram of top level module for dual-core version of scheduler [A12]

The top module of the quad-core version of the scheduler is very similar to the previous dual-core version. The only difference is that the interfaces of the top module, Semaphore component and Running Tasks component are extended for communication with four CPU cores. The block diagram for the quad-core version of the scheduler is displayed in Figure 11 [A9, A12].

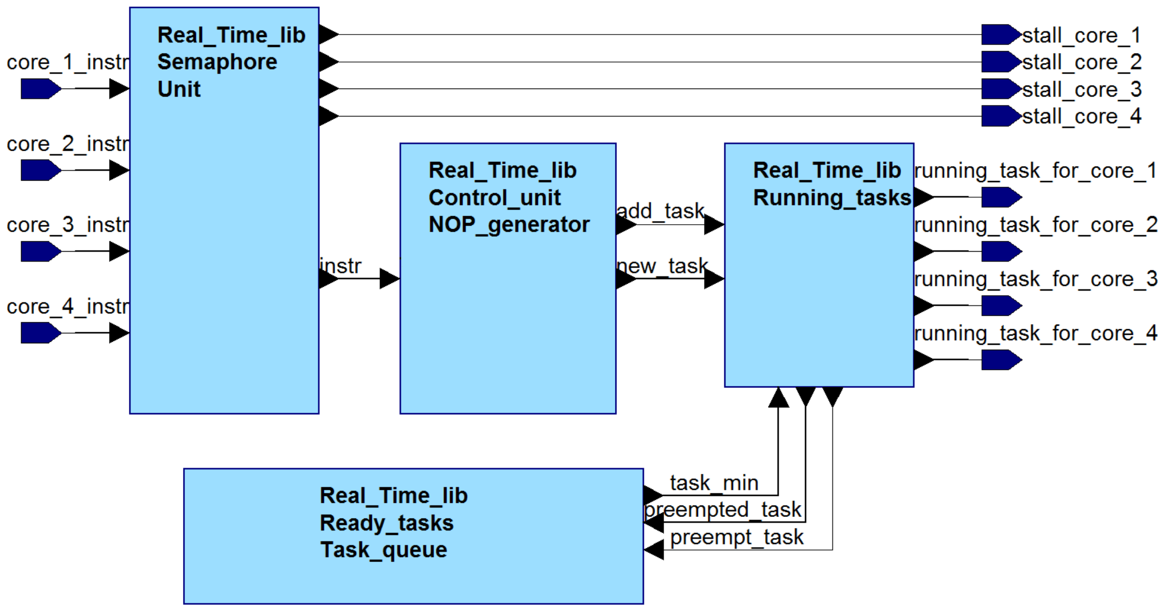

Figure 11: Block diagram of top level module for quad-core version of scheduler [A9, A12]

6 Achieved Synthesis Results

6.1 FPGA Synthesis Results for Min/Max Queues

Fig. 12 shows the scalability of Systolic Array, Rocket Queue and Heap Queue architectures with increasing queue capacity that represents the maximum number of items that can be stored in the queue.

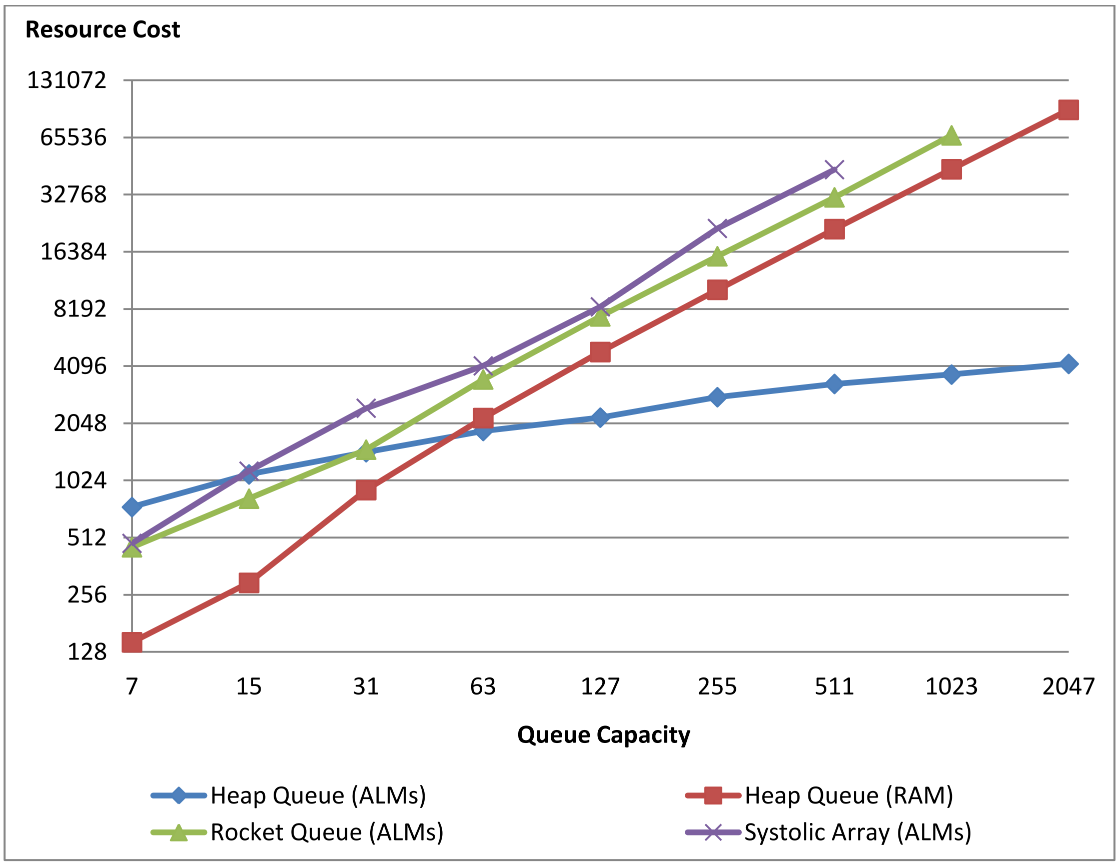

Figure 12: FPGA synthesis results of Heap Queue, Rocket Queue and Systolic Array [A14]

Fig. 13 shows the scalability of Systolic Array, Rocket Queue and Heap Queue architectures with increasing bit width of sorted data values. One can notice that the new proposed architectures are scaling better than Systolic Array. One can also notice that the Heap Queue architecture is mentioned in results two times: for ALMs consumption and for static RAM consumption. The RAM consumption is valid only for the Heap Queue architecture since the other architectures do not use any RAM bits at all [A14].

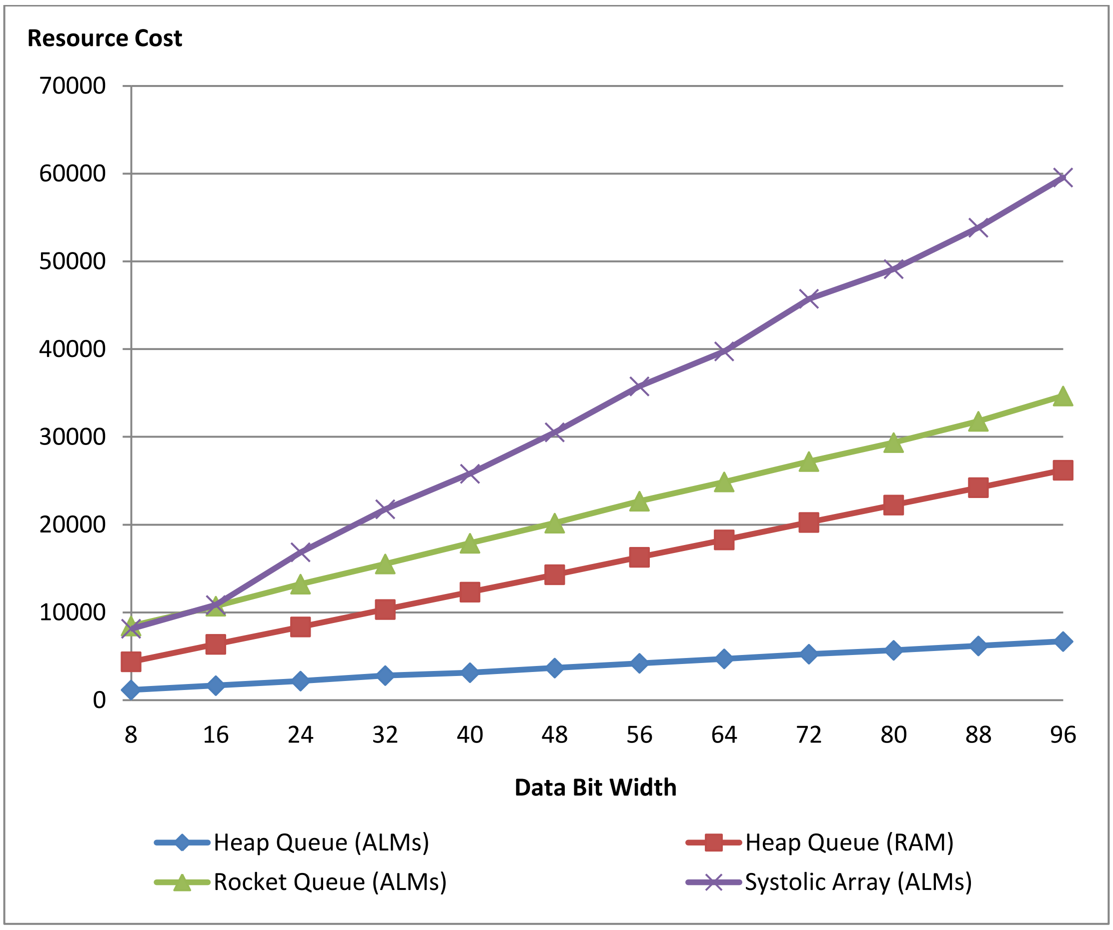

Figure 13: FPGA synthesis results of Heap Queue, Rocket Queue and Systolic Array [A14]

6.2 ASIC Synthesis Results for Min/Max Queues

An ASIC synthesis of the Heap Queue, Rocket Queue and Systolic Array architectures were performed in order to compare the resource costs of these three architectures. The ASIC technology used for synthesis is 28 nm high-performance mobile version of TSMC. A clock frequency of 1 GHz and power supply of 0.9 V was used. Several tables were created, each table comparing the efficiency with respect to a different parameter that is being changed. Figure 14 compares chip area costs and Fig. 15 shows a power consumption comparison [A17].

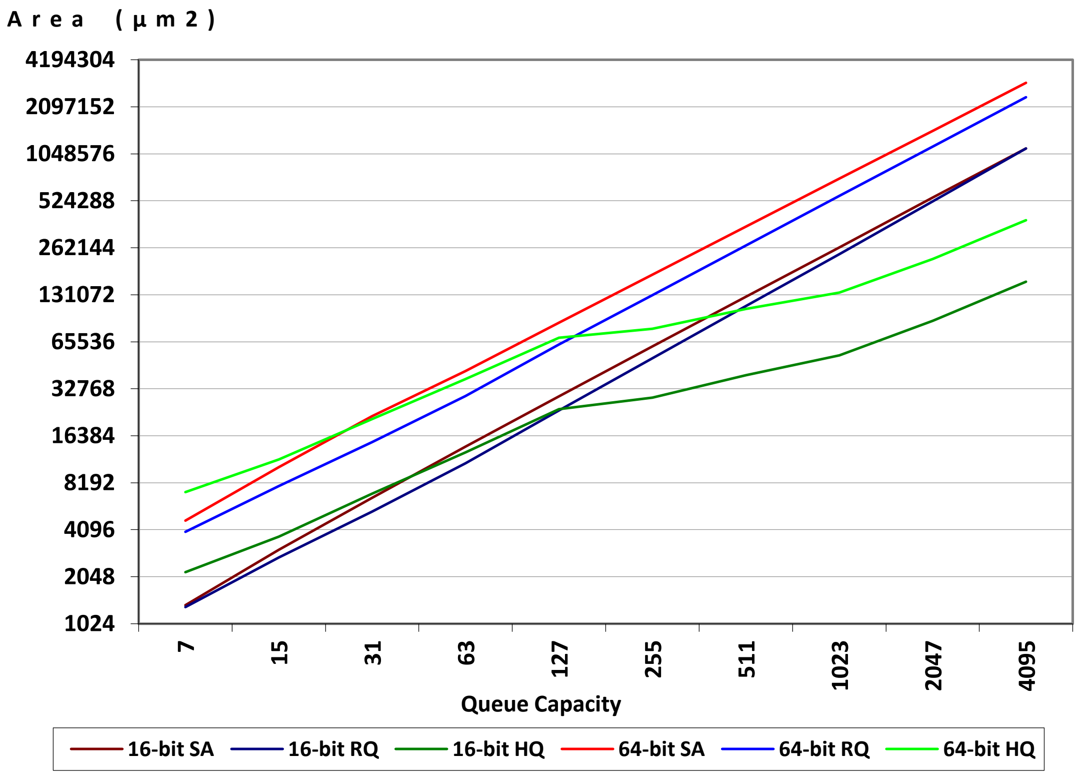

Figure 14: Chip area costs of Rocket Queue (RQ), Heap Queue (HQ) and Systolic Array (SA) for various queue capacities [A17]

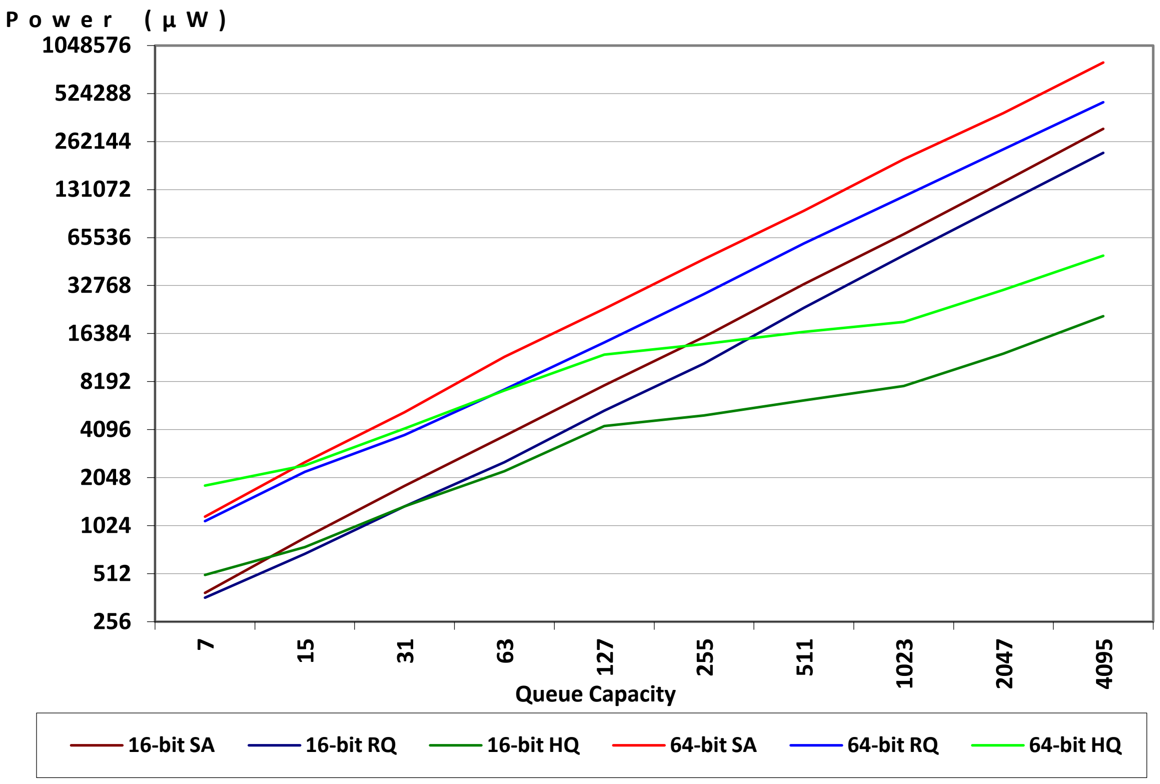

Figure 15: Power consumptions of Rocket Queue (RQ), Heap Queue (HQ) and Systolic Array (SA) for various queue capacities [A17]

6.3 Comparison of 1-Core, 2-Core and 4-Core EDF Schedulers

A synthesis of HW-accelerated EDF-based task scheduling for single-core, dual-core and quad-core real-time systems was performed in order to compare the area chip costs of all three versions of the scheduler. Shift registers architecture was used for implementation of the ready queue within the EDF scheduler. Figure 16 shows a relative increase of costs for the dual-core and quad-core versions of the scheduler in comparison to the single-core version, depending on the maximum number of tasks in the scheduler [A12].

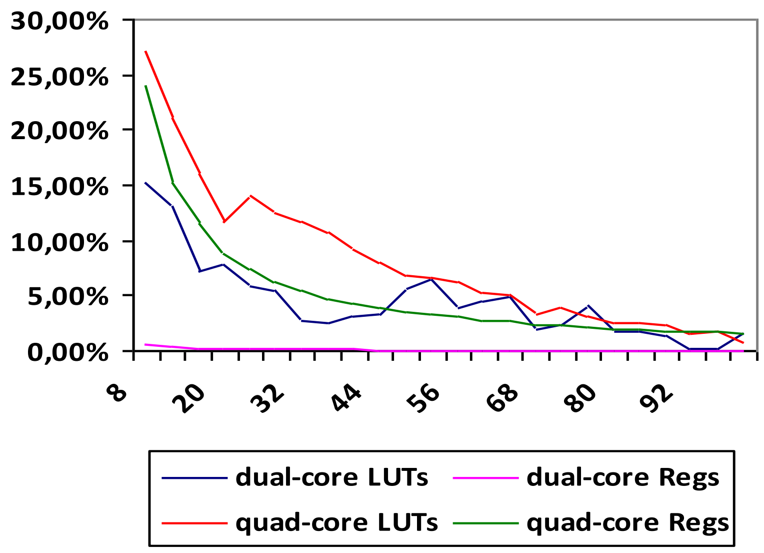

Figure 16: Relative resource cost increase when changing single-core EDF scheduler to dual-core or quad-core version [A9, A12]

The overall comparison of the proposed schedulers, based on the achieved synthesis results, is presented in Table I. One can conclude that it is definitely worth of using 4 CPU cores and the scheduler designed for quad-core CPUs. The resource cost of supporting 4 CPU cores is relatively low in comparison to the performance improvement caused by increased throughput of the system [A12].

TABLE I. OVERALL COMPARISON OF THE SCHEDULERS [A12]

Criterion	Scheduler Version		
	single-core	*dual-core*	*quad-core*
Chip Area Costs	100 %	100 – 115 %	101 – 127 %
Best Case Execution Time	2 clock cycles	2 clock cycles	2 clock cycles
Worst Case Execution Time	2 clock cycles	4 clock cycles	8 clock cycles
CPU Stalls	0	0 to 1	0 to 3
CPU Cores	1	2	4
CPU Throughput	100%	200%	400%

6.4 Comparison of EDF, GED and RED Schedulers

Figure 17 shows a comparison of how fast ALM consumption grows with increasing number of task cells in all three schedulers, EDF-based, GED-based and RED-based.

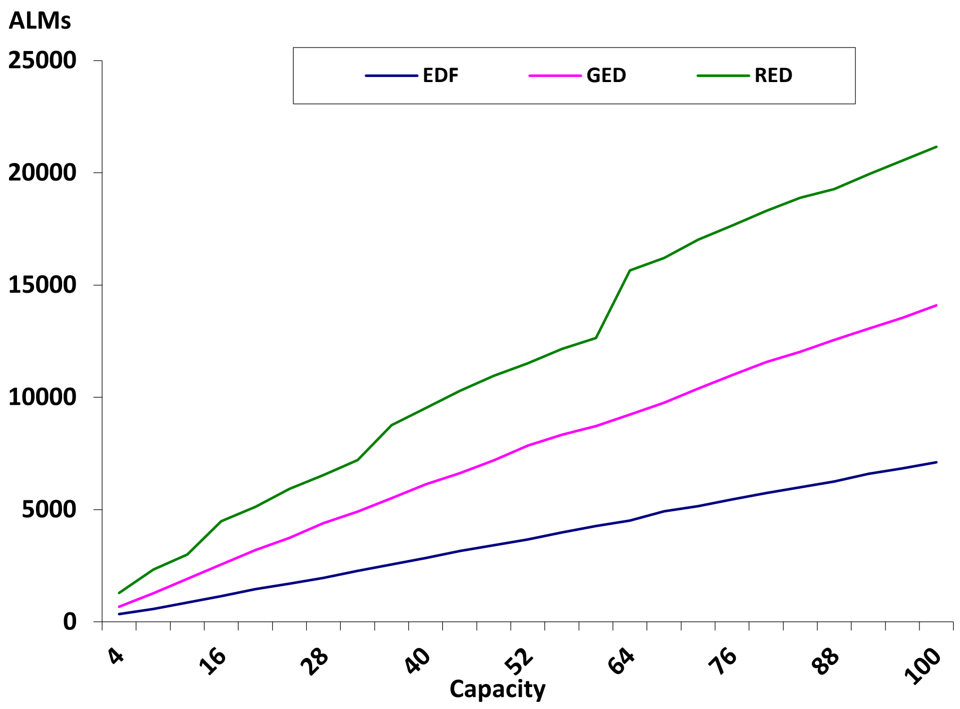

Figure 17: Comparison of ALM consumptions of EDF, GED and RED schedulers for various queue capacities [A15]

Fig. 18 shows a comparison of how fast ALM consumption grows with the increasing bit width of both time values (i.e. deadline values and execution time values) all three schedulers - EDF-based, GED-based and RED-based [A15].

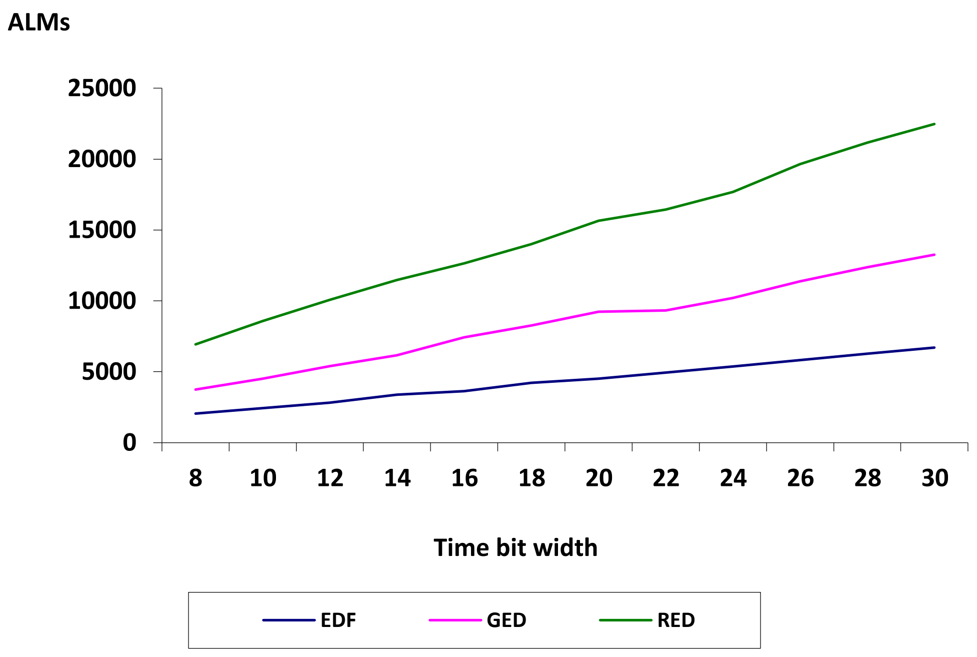

Figure 18: Comparison of ALM consumptions of EDF, GED and RED schedulers for various bit widths of time values [A15]

The overall comparison of the EDF-based, GED-based and the novel RED-based schedulers is presented in Table II. The support of all three major types of tasks in one scheduler is provided at a cost of increased resource consumption (i.e. LUTs for FPGAs or chip area for ASICs) [A15].

TABLE II. COMPARISON OF EDF, GED AND RED SCHEDULERS [A15]

Criterion	**Selected Scheduler**		
	EDF-based	***GED-based***	***RED-based***
ALMs	E	1.94 - 2.24 x E	2.98 - 4.07 x E
Registers	R	1.75 – 2.42 x R	3.02 – 5.75 x R
Best Case Execution Time	2 clock cycles	2 clock cycles	2 clock cycles
Worst Case Execution Time	2 clock cycles	2 clock cycles	2 clock cycles
Hard RT tasks	yes	no	yes
Soft RT tasks	no	yes	yes
Non-RT tasks	yes	yes	yes

7 Summary of Contributions

This section summarizes the direct and indirect contributions of the architectures that were proposed in this PhD thesis. The summary of contributions is shown in Figure 19. All the new hardware architectures proposed in this thesis are listed in the first column. *Proposed Solutions*. Direct contributions and *Impacts on RTOS* of the proposed hardware solutions on the improvement of real-time operating systems are presented within the second column. These improvements have subsequent impact on the overall real-time system that would use such improved RTOS. Thus, the indirect contributions of the proposed architectures are listed in column *3. System impacts*.

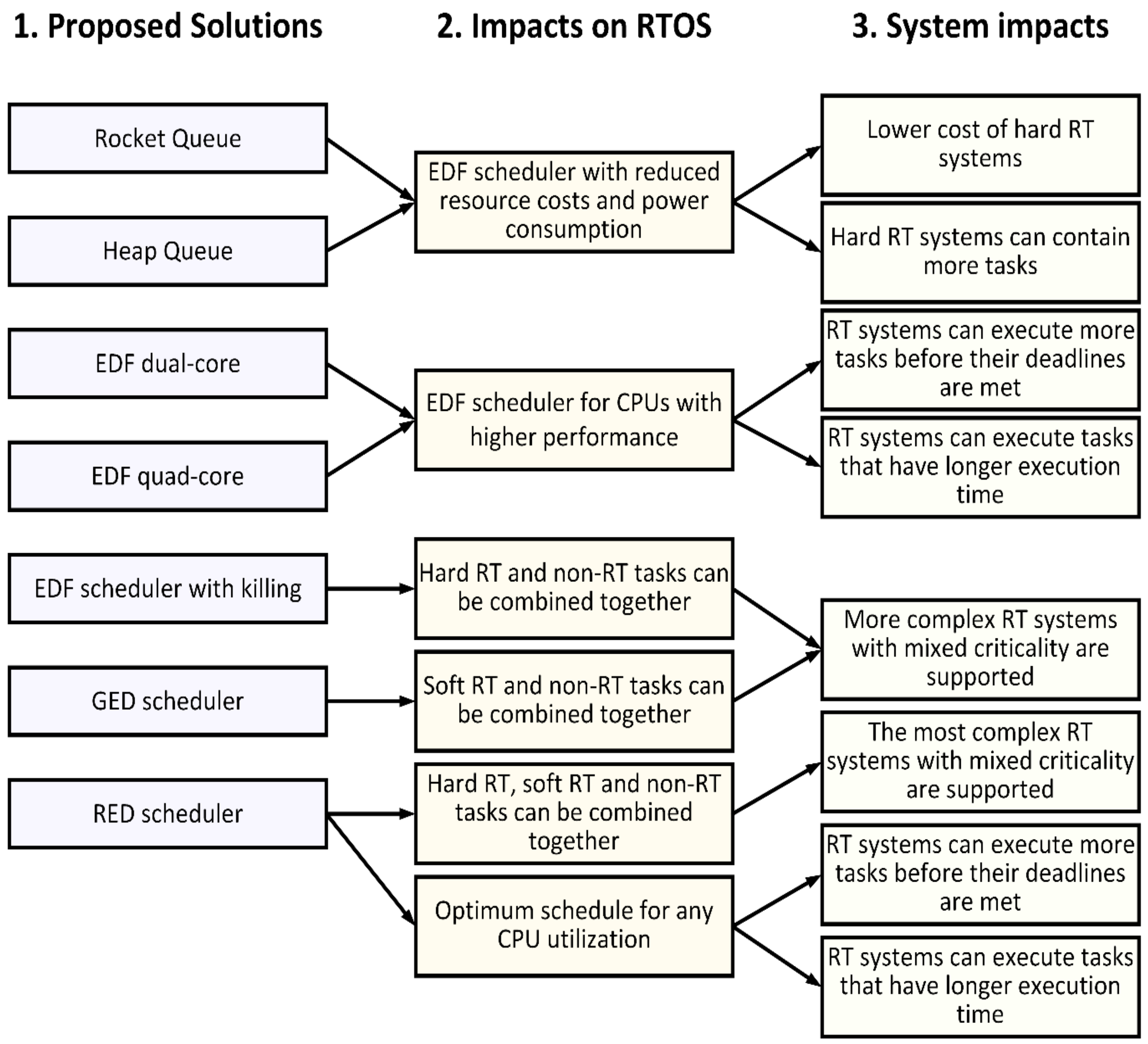
1. Proposed Solutions
2. Impacts on RTOS
3. System impacts
Rocket Queue
Heap Queue
EDF scheduler with reduced resource costs and power consumption
Lower cost of hard RT systems
Hard RT systems can contain more tasks
EDF dual-core
EDF quad-core
EDF scheduler for CPUs with higher performance
RT systems can execute more tasks before their deadlines are met
RT systems can execute tasks that have longer execution time
EDF scheduler with killing
GED scheduler
RED scheduler
Hard RT and non-RT tasks can be combined together
Soft RT and non-RT tasks can be combined together
Hard RT, soft RT and non-RT tasks can be combined together
Optimum schedule for any CPU utilization
More complex RT systems with mixed criticality are supported
The most complex RT systems with mixed criticality are supported
RT systems can execute more tasks before their deadlines are met
RT systems can execute tasks that have longer execution time

Figure 19: Summary of contributions of research carried out within this PhD thesis

Main contributions and features of the proposed solutions are:

- The new proposed min/max queue architectures, Rocket Queue and Heap Queue, consume less resources (i.e. chip area for ASIC and LUT count for FPGA) than the existing architecture called Systolic Array. Therefore, a task scheduler that realizes EDF algorithm and uses either Rocket Queue or Heap Queue architecture has reduced resource costs and power consumption as well. Subsequently, the reduced chip area or LUTs consumption can have impact on the total costs of the real-time system that uses one of the proposed min/max queue architectures instead of Systolic Array. Alternatively, one can decide to rather increase the queue capacity in order to support more tasks in the system and keep the resource costs approximately the same as before.

- The proposed extensions of EDF schedulers that support an execution of two tasks (i.e. dual-core version) or four tasks (i.e. quad-core version) in parallel, have direct impact on the performance of the CPU executing the real-time tasks. The enhanced performance of the CPU increases the number of tasks that finish their execution before the deadline is met. Alternatively, tasks with longer execution time can be used instead.

- The proposed EDF-based scheduler that supports task killing can schedule non-RT tasks in addition to the hard RT. Therefore, more complex RT systems containing both types of tasks can be developed relatively easily because there is no need to assign priorities to the hard RT tasks, only to non-RT tasks. This scheduler allows to create new and more complex real-time systems.

- The GED-based scheduler developed within our research is the first hardware accelerated scheduler that realizes GED algorithm (according to our best knowledge). Thanks to the features of the GED algorithm, soft RT tasks are scheduled optimally. The GED scheduler is extended to support non-RT tasks as well. Thus, real-time systems consisting of soft RT tasks and non-RT tasks are suitable for usage of the GED scheduler.

- The last proposed solution, the RED scheduler, is realizing the most complex scheduling algorithm called RED. This solution provides the combination of hard RT tasks, soft RT tasks and non-RT tasks within one operating system easily and effectively. Such a combination allows to create the most complex real-time systems with mixed task criticality and task types. The proposed RED-based scheduler also creates the optimum schedule regardless of CPU utilization or CPU load, which minimizes the occurrence of deadline misses. Thus, more tasks and tasks with longer execution time can be used.

8 Conclusion

In this work, analysis of real-time systems and their attributes was performed at first. Real-time operating systems and their most frequently used algorithms were analysed too. Existing hardware architectures for data sorting and existing hardware accelerated task schedulers for real-time systems were analysed as well. According to the knowledge obtained from the performed state-of-the-art analysis, a motivation for development of improved architectures for real-time task scheduling was identified. Based on this motivation, research objectives for this thesis were specified. The research objectives were focused on design, verification and evaluation of new architectures that shall improve real-time task scheduling.

Two new sorting architectures for min/max queues were proposed in order to minimize resource costs and power consumption of hardware accelerated EDF-based schedulers that create the optimum schedule for systems consisting of hard real-time tasks only. To create a new hardware accelerated task scheduler that supports mixed criticality systems, EDF scheduler with task killing, GED scheduler and RED scheduler were developed. Each of these scheduler architectures is suitable for a different system (application), depending on types of tasks that are present in the system. Additionally, two approaches for extension of the task schedulers to support dual-core and quad-core CPUs were proposed and designed too. When using more CPU cores, significantly more tasks or more computationally intensive tasks can be executed in the real-time system without deadline misses.

The correctness of all the proposed solutions was intensively verified in UVM-based simulations, which is considered to be the current state of the art for verification of digital circuits designed for real-time systems, such as systems within aerospace industry. In this way, it was proven that the designed architectures meet their high-level algorithmic specifications. Since only existing algorithms were implemented within the architectures, it is secured that these specifications are correct too. Additionally, all the proposed solutions were also proved to be synthesizable, since synthesis into ASIC and/or FPGA technologies was performed as well. Last but not least, the synthesized designs were also tested in FPGA using functional BIST technique and a static timing analysis was performed as well.

The synthesis results of the proposed solutions were evaluated and compared to other comparable results, within the same implementation technology and same parameterization of the synthesized circuit. Based on the achieved results and the comparison to the results of alternative solutions, a summary of contributions achieved within this PhD thesis was described, underlining the main advantages and benefits of the proposed solutions and their impact on the real-time system potentially using such solutions.

9 Záver

V rámci tejto práce bola najprv vykonaná analýza systémov reálneho času a ich atribútov. Taktiež boli analyzované operačné systémy reálneho času a ich najčastejšie používané algoritmy. Následne boli analyzované existujúce hardvérové architektúry pre zoraďovanie dát a existujúce hardvérovo-akcelerované plánovače úloh pre systémy reálneho času. Podľa vedomostí získaných z vykonanej analýzy súčasného stavu bola identifikovaná motivácia pre vývoj vylepšených architektúr pre plánovanie úloh reálneho času. Na základe tejto motivácie boli špecifikované ciele dizertačnej práce. Tieto výskumné ciele boli zamerané na návrh, verifikáciu a vyhodnotenie nových architektúr, ktoré majú vylepšiť plánovanie úloh reálneho času.

Boli navrhované dve nové zoraďovacie architektúry pre min/max rady za účelom minimalizovania nákladov na výpočtové zdroje a spotrebu energie pre realizáciu hardvérovo akcelerovaných plánovačov, ktoré sú založené na EDF algoritme a vytvárajú tak optimálny plán pre systémy pozostávajúce z hard RT úloh. Na vytvorenie nového hardvérovo-akcelerovaného plánovača, ktorý podporuje systémy zmiešanej kritickosti, boli vyvinuté plánovače typu EDF so zabíjaním úloh, GED plánovač a RED plánovač. Každý z týchto plánovačov je vhodný pre iný systém (aplikáciu) v závislosti na type úloh, ktoré sa nachádzajú v takomto systéme. K tomu boli navrhnuté dva prístupy pre rozšírenie plánovačov úloh, aby podporovali dvojjadrové a štvorjadrové procesory. Použitím viac CPU jadier je možné vykonať výrazne viac úloh alebo výpočtovo náročnejšie úlohy v stanovených časových ohraničeniach.

Korektnosť všetkých navrhovaných riešení bola intenzívne overená v UVM simuláciách, čo je považované za súčasný štandard pre verifikáciu číslicových obvodov navrhnutých pre systémy reálneho času, ako napríklad systémy v rámci leteckého a vesmírneho priemyslu. Týmto spôsobom bolo dokázané, že navrhnuté architektúry spĺňajú svoje vysokoúrovňové algoritmické špecifikácie. Keďže iba existujúce algoritmy boli implementované v rámci týchto architektúr, je zabezpečené, že tieto špecifikácie sú taktiež korektné. Navyše, všetky navrhované riešenia boli tiež syntetizované do ASIC a/alebo FPGA technológie. Bola vykonaná statická časová analýza a návrhy syntetizované do FPGA boli tiež otestované v FPGA použitím techniky funkcionálneho BIST-u.

Výsledky syntéz navrhovaných riešení boli vyhodnotené a porovnané s ostatnými porovnateľnými výsledkami v rámci rovnakej implementačnej technológie a rovnakej parametrizácie syntetizovaného obvodu. Na základe dosiahnutých výsledkov a ich porovnania s alternatívnymi riešeniami bolo opísané zhrnutie dosiahnutých prínosov v rámci tejto dizertačnej práce. Boli podčiarknuté hlavné výhody a benefity navrhovaných riešení a ich vplyv na systémy reálneho času, ktoré potenciálne použijú takéto riešenia.

Author’s Publications

[A1] L. Kohútka, "Hardware task scheduling in real-time systems," in IIT.SRC 2015, Student Research Conference, Bratislava, 2015.

[A2] L. Kohútka, M. Vojtko and T. Krajčovič, "Hardware accelerated task scheduling in real-time systems," in ECBS-EERC 2015, Student Research Conference, 2015 IEEE Fourth Eastern European Regional Conference on the Engineering of Computer Based Systems, Brno, 2015.

[A3] L. Kohútka and V. Stopjaková, "Hardware Accelerated Task Scheduling in Real-Time Systems: Deadline Based Coprocessor for Dual-Core CPUs," 2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Košice, 2016.

[A4] L. Kohútka and V. Stopjaková, "Hardware Accelerated Task Scheduling in Real-Time Systems," 4th International Conference on Advances in Electronic and Photonic Technologies (ADEPT), 2016.

[A5] L. Kohútka and V. Stopjaková, "Task Scheduler for Dual-Core Real-Time Systems," 2016 MIXDES - 23rd International Conference Mixed Design of Integrated Circuits and Systems, Lodz, 2016, pp. 474-479.

[A6] L. Kohútka and V. Stopjaková, "Improved Task Scheduler for Dual-Core Real-Time Systems," 2016 Euromicro Conference on Digital System Design (DSD), Limassol, 2016, pp. 471-478.

[A7] L. Kohútka and V. Stopjaková, "Hardvérová platforma pre systémy reálneho času," Počítačové Architektury a Diagnostika (PAD), 2016.

[A8] L. Kohútka and V. Stopjaková, "Rocket Queue: New data sorting architecture for real-time systems," 2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Dresden, 2017, pp. 207-212.

[A9] L. Kohútka and V. Stopjaková, "Real-Time Task Scheduler for Quad-Core CPUs," 5th International Conference on Advances in Electronic and Photonic Technologies (ADEPT), 2017.

[A10] L. Kohútka and V. Stopjaková, "A new efficient sorting architecture for real-time systems," 2017 6th Mediterranean Conference on Embedded Computing (MECO), Bar, 2017

[A11] L. Kohútka and V. Stopjaková, "Hardware Kernel for Real-Time Systems," Počítačové Architektury a Diagnostika (PAD), 2017.

[A12] L. Kohútka and V. Stopjaková, "Extension of hardware-accelerated real-time task schedulers for support of quad-core processors," 2017 5th IEEE Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE), Riga, 2017.

[A13] L. Kohútka and V. Stopjaková, "Reliable real-time task scheduler based on Rocket Queue architecture," Microelectronics Reliability, Volume 84, 2018, pp. 7-19, ISSN 0026-2714, https://doi.org/10.1016/j.microrel.2017.12.007.

[A14] L. Kohútka and V. Stopjaková, "A Novel Efficient Hardware Architecture of MIN/MAX Queues for Real-Time Systems," 2018 IEEE 21th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Budapest, 2018.

[A15] L. Kohútka and V. Stopjaková, "A novel hardware-accelerated real-time task scheduler based on robust earliest deadline algorithm," 2018 IEEE 13th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS), Taormina, 2018.

[A16] L. Kohútka and V. Stopjaková, "A Novel Hardware-Accelerated Scheduler for Soft Real-Time Tasks," 2018 IEEE 19th International Conference on Industrial Technology (ICIT), Lyon, 2018. (accepted but not yet published)

[A17] L. Kohútka and V. Stopjaková, "A Novel Hardware-Accelerated Priority Queue for Real-Time Systems," 2018 Euromicro Conference on Digital System Design (DSD), Prague, 2016. (accepted but not yet published)

Publications Unrelated to the Topic of PhD Thesis

[U1] L. Kohútka, "Faster Synthesis of Combinational Logic Based on Multiplexer Trees and Binary Decision Diagrams," in IIT.SRC 2014, Student Research Conference, Bratislava, 2014.

[U2] K. Burda, R. Grežo, M. Hasin and L. Kohútka, "Therapeutic Systen for Children with Movement Disorders," in IIT.SRC 2014, Student Research Conference, Bratislava, 2014.

[U3] L. Kohútka, "Faster synthesis of combinational logic based on multiplexer trees and binary decision diagrams, " 2014 IEEE 12th IEEE International Conference on Emerging eLearning Technologies and Applications (ICETA), Stary Smokovec, 2014, pp. 239-244.

[U4] L. Kohútka, "Design of digital systems — A challenge towards new study programs," 2016 International Conference on Emerging eLearning Technologies and Applications (ICETA), Vysoke Tatry, 2016, pp. 171-176.

Author’s Honors and Awards

Commemorative Sheet 2014

Nov 2014 – Dean of Faculty of Informatics and Information Technologies

Award for achievements in academic year 2013/2014

IET Prize 2015

Jul 2015 – The Institution of Engineering and Technology

http://conferences.theiet.org/achievement/-documents/prize-winners2015.cfm?type=pdf

Dean's Laudatory Appreciation 2015

Jul 2015 – Dean of Faculty of Informatics and Information Technologies

Award for excellent Master Thesis

Professor Jan Hlavicka Prize 2016

Sep 2016 – Processor Architectures and Diagnostics - PAD 2016

Award for excellent results in doctoral study

http://www.fit.vutbr.cz/events/PAD/oceneni.html

The Best Paper at MECO 2017

Jun 2017 – IEEE MECO 2017

Award for the best research paper among all papers published in conference MECO 2017.

Professor Jan Hlavicka Prize 2017

Sep 2017 – Processor Architectures and Diagnostics - PAD 2017

Award for excellent results in doctoral study

Student of Year 2017

Nov 2017 – Rector of Slovak University of Technology in Bratislava

Rector's Award 2017

Nomination for Student celebrity of Slovakia 2017

Nov 2017 – Rector of Slovak University of Technology in Bratislava

Main references used in the Abstract

[1] O'Reilly, C.A., and Cromarty, A.S., "Fast” is not “Real-time” in designing effective real-time AI systems, SPIE Vol. 5~8 Application of Artificial Intelligence II, pp. 249-257, 1985.

[2] J. A. Stankovic, K. Ramamritham, Tutorial hard real-time systems, Computer Society Press, 1988.

[3] G. C. Buttazzo: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 2011, ISBN 9781461406754, 9781461406761.

[4] Lee, I.; Leung, J. Y.-T. & Son, S. H., Handbook of Real-Time and Embedded Systems, Chapman & Hall/CRC, 2007.

[5] P. Marwedel: Embedded System Design: Embedded Systems Foundations of Cyber-physical Systems, 2010, ISBN 9400702566.

[6] J. A. Stankovic: Real-time and embedded systems http://doi.acm.org/10.1145/234313.234400

[7] G. C. Buttazzo, M. Bertogna, Y. Gang, "Limited Preemptive Scheduling for Real-Time Systems. A Survey," in *Industrial Informatics, IEEE Transactions on* , vol.9, no.1, pp.3-15, Feb. 2013

[8] C. A. Neil, “Deadline Monotonic Scheduling,“ Department of Computer Science, University of York, Heslington York, 1990.

[9] H. Kopetz, “Real-Time Systems Design Principles for Distributed Embedded Applications,” 2011.

[10] Cottet F, Delacroix J, Kaiser C and Mammeri Z, "Scheduling in realtime systems," Wiley, New York, 2002, ISBN 978-0470847664.

[11] D. de Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in RTSS, 2009.

[12] A. S. Tanenbaum: Modern Operating Systems (4th Edition) (GOAL Series). Prentice Hall, 2014, ISBN 978-0133591620.

[13] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," in Proceedings of the IEEE, vol. 82, no. 1, pp. 55-67, Jan 1994.

[14] C. Ferreira, A. S. R. Oliveira: RTOS Hardware Coprocessor Implementation in VHDL, 2009.

[15] J. Swart “Real Time Operating Systems Implemented in Hardware”, 2010.

[16] S. W. Moon, K. G. Shin, J. Rexford: Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches, 2000, ISSN 00189340.