Novel efficient on-chip task scheduler for multi-core hard real-time systems

L. Kohútka, V. Stopjaková*

Institute of Electronics and Photonics, Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava, Slovakia

ABSTRACT

This article presents an efficient hardware architecture of EDF-based task scheduler, which is suitable for hard real-time systems due to the constant response time of the scheduler. The proposed scheduler contains a queue of ready tasks that is based on a new MIN/MAX queue architecture called Heap Queue, which is inspired by Shift Registers, Systolic Arrays, heapsort algorithm, the Rocket Queue architecture and dual-port RAMs. The instructions of the proposed scheduler have throughput of one instruction per two clock cycles regardless of the actual number of tasks managed by the scheduler, and regardless of the scheduler capacity. The developed task scheduler is optimized for low chip area costs, which leads to lower energy consumption. The Heap Queue-based architecture has constant time complexity due to two clock-cycle response time of the instructions and therefore, the architecture is highly deterministic. The scheduler supports CPUs that can execute 1, 2 or 4 tasks simultaneously, and contains an implementation of clever and efficient logic that can handle conflicts caused by the fact that the scheduler is used by all CPU cores at the same time. The proposed scheduler was verified through SystemVerilog UVM-like simulations that applied billions of randomly generated test instructions. Achieved ASIC (28 nm) and FPGA synthesis results are presented and compared. More than 86% of the chip area and 93% of the total power consumption can be saved if Heap Queue architecture is used in hardware implementations of EDF algorithm. Advantages and disadvantages of the proposed task scheduler are discussed through the comparison to the existing solutions.

1. Introduction

Real-time systems represent a category of embedded systems that are processing real-time tasks. Success of real-time tasks depends not only on the computation result itself but also on time when these tasks are completed. Too late completion of hard real-time tasks may represent the same failure as an incorrect result of the task. Thus, reliability of real-time systems means that the tasks are completed in correct time [1, 2].

Task scheduling algorithms are usually containing data sorting in a form of MIN/MAX queues, which are typically implemented in software. Software implementations are often sufficient for small and simple real-time systems with small amount of tasks. Nevertheless, increased complexity of systems and the number of real-time tasks require higher average performance and less varying latency caused by data sorting (i.e. constant response time). Constant response time is especially important for safety-critical systems. Hard real-time systems are usually also safety-critical systems, and vice versa, safety-critical systems usually include hard real-time systems. Due to this, the requirements for meeting deadlines of tasks can be also seen as reliability requirements because missing a deadline is also considered as a failure of the systems. Even if a micro-controller with the highest possible performance was used, there would still be no guarantee that all tasks will meet their deadlines. Therefore, a dedicated task scheduler that provides scheduling of real-time tasks should be used in real-time / safety-critical systems [3-7].

The constant latency of all operations within the system, including data sorting, is very important for more reliable scheduling in hard real-time systems. In such cases, software implementations do not fulfill all the requirements because software algorithms for data sorting do not operate in constant time. Alternative solutions are based on hardware acceleration, thus the data sorting and MIN/MAX queues can be implemented in a digital integrated circuit (e.g. ASIC or FPGA) [8-31]. Several hardware architectures designed for data sorting or implementation of MIN/MAX queues have been developed so far. However, they all suffer from consuming too many LUT (Look-up Tables) resources and they have relatively high chip area costs in ASIC (in addition to FPGA) technologies as well [10-28].

The research presented in this article is focused on designing an improved version of a coprocessor that implements task scheduling based on the Earliest-Deadline First (EDF) algorithm that is described in [4]. The EDF algorithm can be also seen as a dynamic version of deadline-driven scheduling algorithms [32]. The improvements are focused on increased performance together with system determinism on one side, and the reduced hardware costs (i.e. chip area or FPGA utilization) together with power consumption on the other side. This is achieved by implementation of an efficient hardware-based data sorting in a form of MIN/MAX queues, which requires reduced amount of logic resources needed. For this purpose, a new architecture called Heap Queue was developed in order to increase the scalability of data sorting implemented in hardware so that more items can be efficiently sorted [33, 34]. This architecture is suitable for usage in task scheduling for real-time systems, where the overall resource costs of the scheduling were significantly reduced too.

This article also deals with the performance issue of modern CPUs that usually run multiple tasks in parallel due to multi-core paradigm adopted for CPU design. This brings new obstacles and

challenges for hardware-accelerated task scheduling and especially, for task scheduling suited for hard real-time systems.

The structure of the paper is as follows. Section 2 describes related work on task schedulers for hard real-time systems. Section 3 contains related work on sorting MIN/MAX queue architectures that can be used for implementation of real-time task schedulers. In Section 4, two new task schedulers are proposed. Verification of the described solutions is described in Section 5. Section 6 contains synthesis results in a form of tables. These results are discussed and a conclusion of the paper is presented in Section 7.

2. Related work on task schedulers for real-time systems

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 classic 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. The most common and popular algorithm used for scheduling of hard-RT tasks is called Earliest-Deadline First. This algorithm 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 ideal for implementation of the EDF algorithm [4, 35, 36].

An ideal real-time task scheduler always schedules the optimum sequence of tasks so that all tasks will be computed and completed before their deadlines are met. In addition to this, the ideal real-time task scheduler has no overhead on CPU that executes the tasks. The more the CPU is used for the scheduling algorithm, the less effective it is for computation and completion of the scheduled tasks. Of course, a real scheduler will always need to consume some amount of CPU time because it is needed to spend at least one CPU clock cycle to write input data to the scheduler or read the output data from the scheduler. Nevertheless, for the performance reasons, we would like to consume as little CPU time as possible. In order to keep the whole embedded system deterministic and well predictable, constant amount of spent CPU time is targeted regardless of any parameters, e.g. the number of tasks that are currently scheduled or the maximum possible number of tasks of the system (i.e. task queue size).

In our previous work [24], a novel real-time task scheduler based on EDF algorithm and implemented in a form of a coprocessor unit has been developed. A comparison of hardware and software implementations in terms of performance and efficiency of the scheduler was done and achieved results were presented. The coprocessor uses instructions consuming two clock cycles of the CPU regardless of the current and the maximum number of tasks in the system.

Then, we designed and proposed an extended version of the scheduler that is suitable for dual-core CPUs. Two approaches for solving of conflicts (i.e. situations when multiple CPU cores want to use the coprocessor at the same time) were designed and compared [26]. After that, we added a support for scheduling non-real-time tasks in the same scheduler by using priorities instead of deadline values [27]. Finally, an improved form of the scheduler optimized in terms of timing precision, chip area costs and power consumption was proposed in [28].

Beside of our coprocessors, there are also other solutions existing. In [19], EDF algorithm is also used but with the maximum number of tasks being only 64, while the other approach uses priorities instead of deadlines that is not optimal for hard real-time systems [20]. There are also other solutions based on priorities or static scheduling [10, 21, 22]. The scheduler, we have already presented, is an efficient solution for simpler embedded systems containing hard real-time tasks and employing one single-core CPU. However, as systems grow in their complexity, more performance is required, which often leads to use of multi-core CPUs. In that case, a more complex task scheduler is needed in order to provide multi-core CPU support. The suitability of the EDF algorithm was deeply analyzed, and it was concluded that EDF is suitable for uniform multiprocessor systems [19]. Therefore, we decided to design completely new real-time task scheduling coprocessor that would be optimal for either dual-core or quad-core real-time embedded systems.

3. Related work on MIN/MAX queues

Deadline-based task schedulers for real-time systems perform task sorting according to their deadlines very intensively for implementation of decision, which task to select for execution. Therefore, data sorting in a form of MIN/MAX queues represents the core functionality of task schedulers implemented in hardware [19, 20, 24, 27].

The research presented in this paper is focused on data sorting in real-time systems, where the lowest value of the sorted data (i.e. the earliest deadline) is needed. Thus, the goal is to implement the MIN queue. These queues contain items that are being sorted. The items consist of these values [33, 34]:

- *SORT_DATA* – the items shall be sorted according this value. Thus, if a queue is the MIN queue, then the output of the queue is the item with the lowest *SORT_DATA* value. If a queue is the max queue, then the output of the queue is the item with the highest *SORT_DATA* value.
- *PAYLOAD* – a bit vector that represents some data that is relevant to the application. The payload does not affect the sorting decisions. This value can also serve as identification number of the item or as a pointer/address to memory.
- *VALID* – this is a 1-bit value that informs, whether the item is valid or empty. If this value is high (i.e. logic 1), then the item is valid. Otherwise, the item is empty. If the output of the queue is an empty item (i.e. the first item within the queue is empty), then the whole queue is empty.

The MIN queue must provide these three instructions identified by 2-bit opcode [33, 34]:

- *00 – NOP* – no operation is performed. The output of the queue will remain unchanged. This is used to keep the accelerator idle. This feature is required.
- *01 – INSERT* – a new 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 *SORT_DATA* remains to be the output of the MIN/MAX queue. This feature is required.

- *10 – POP* – the first item within the queue is removed from the queue. The remaining items update their positions so that the second item becomes the new first item of the queue. Thus, the output of the queue is updated to provide the new first item as a result. This feature is required.

The attributes of the MIN queue are described from the hardware as well as real-time computing points of view. These attributes and their requirements are [33, 34]:

- Constant instruction response time (the number of clock cycles needed for instructions to provide an updated output). The requirement is to have constant time complexity, i.e. all instructions of the accelerator are providing output in constant number of clock cycles. This means that the clock cycles amount does not change either by having various items count in the queue (i.e. changing actual number of values in the queue) or by having various queue capacity. Constant response time improves predictability and determinism of the whole real-time system [1].
- High performance. This attribute depends on the clock frequency multiplied by the amount of clock cycles needed for usage of one instruction. The clock frequency depends on the critical path length of the accelerator as well as the critical path of other parts of the system within the same clock domain. There is no reason to achieve significantly lower critical path length for the accelerator than for the rest of the clock domain. The number of clock cycles needed for calling the accelerator instructions should be as low as possible.
- Low chip area costs. This attribute depends on the implementation technology. For ASICs, this is evaluated either by the number of transistors used or by real dimensions of the manufactured chip. For FPGAs, this is evaluated by the number of logic resources (e.g. LUTs, registers and RAM bits), depending on the selected FPGA device.

Several architectures for data sorting in MIN/MAX queues have been developed, and can be used for task scheduling of real-time systems. Nevertheless, they suffer from scalability issues due to increasing critical path length and resource cost with regards to increasing capacity of schedulers (i.e. the maximum number of tasks supported). The most popular architectures include FIFO with MUX Tree [10, 11, 14, 20], Shift Registers [17, 19, 22, 23], DP RAM Heapsort [18] and Systolic Array [24-29].

The FIFO approach is the least scalable in terms of critical path length due to the complexity of the MUX Tree part, which contains too long critical path (if higher capacity is selected). It is also very inefficient from chip area point of view [10, 11, 14, 20].

The Shift Registers architecture is more efficient approach than the previous one but there is still a problem with the critical path length. This architecture consists of homogenous cells, where each cell is composed of a comparator, control logic and a set of registers to store one item. The cells can exchange items with their neighbours, where each cell has two neighbours (they are connected within one line). All cells are receiving the same instruction simultaneously from the input of the queue. The more cells the queue contains, the longer the critical path is due to the bus width for providing instructions simultaneously and due to the control signals exchange between all cells. Thus, this architecture can be used for small capacities only. An example of Shift Registers architecture containing eight cells is displayed in Fig. 1 [17, 19, 22, 23].

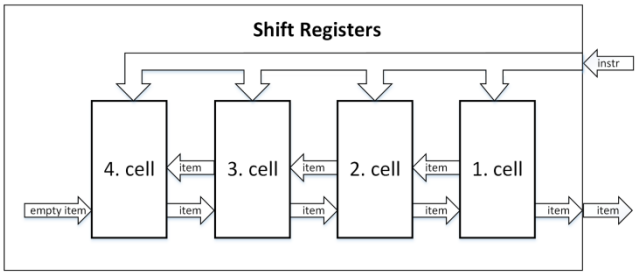

Fig. 1. Shift Registers architecture example [17].

DP RAM Heapsort is relatively efficient sorting architecture due to adoption of dual-port RAM for storage of items within the architecture. However, it is not possible to perform INSERT or POP instructions independently. This architecture can perform POP and INSERT instructions only together. Therefore, this architecture is not usable for implementation of MIN/MAX queues [18].

Systolic Array architecture is very similar to the Shift Registers. However, the critical path problem is solved by pipelining. It contains homogenous cells that are connected within one line. Each cell is a neighbour to one other cell to the left and to the right, except the first and the last cell in the queue. The first cell is the only cell that provides its output to the output of the whole queue and that receives instructions from the input of the queue. The instructions are gradually propagated from the first cell to the last cell (one cell per each clock cycle) in the similar way as instructions are propagated through pipeline stages in the pipelined CPUs [24-29].

Fig. 2 shows an example of the Systolic Array architecture containing 15 cells. The first cell from the right is the first cell of the queue, thus it behaves as an interface to the external environment too. The only signals propagated in parallel are clock and reset signals [24-29].

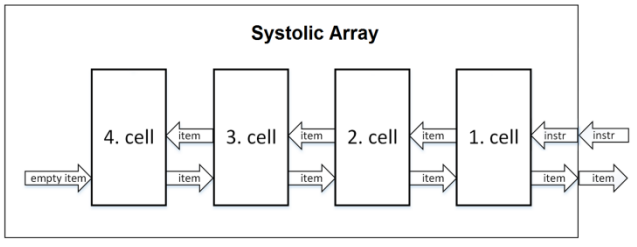

Fig. 2. Systolic Array architecture example [24-29].

Each cell of the Systolic Array represents one pipeline stage including pipeline register. Thus, it takes *N* clock cycles to propagate one instruction through the whole structure, where *N* is the number of cells. However, due to the reason how pipelining works, each cell is executing a different instruction at the same time. Furthermore, the output of the whole queue is updated in the beginning already and thus, it takes only 2 clock cycles to read an updated output of the queue (1 clock cycle is needed to update the first cell and 1 clock cycle is used for reading from the updated cell). The MIN/MAX queue based on this architecture can accept a new instruction every 2 clock cycles, i.e. the instruction response time is 2 (regardless of the cell count) [24-29].

The Rocket Queue architecture was designed as an improvement to the Systolic array architecture inspired by DP RAM Heapsort. The Rocket Queue is structured into levels. There are two types of levels: duplicating levels and merged levels. An example of the Rocket Queue architecture is depicted in Fig. 3, where three duplicating levels and 11 merged levels were used. The amount of duplicating levels may be changed to lower or higher value. Nevertheless, more than 5 duplicating levels are not recommended due to the increasing critical path length [30, 31].

One of the most resource consuming parts of the queues is a comparator. While Systolic Array and Shift Registers are using one comparator per each cell, the Rocket Queue architecture employs a single comparator for all cells within the same level. Thus, the number of comparators in Rocket Queue depends on the amount of levels, not on the number of cells, which leads to lower resource costs of the Rocket Queue architecture [30, 31].

Among these architectures, only the Systolic Array and Rocket Queue architectures meet the requirements described in Section 2. Within these two architectures, the Rocket Queue is more efficient in terms of resource costs and therefore, it is the most suitable architecture among all existing ones [30, 31]. Furthermore, Systolic Array is able to remove an item from any position according to its ID, which is necessary for flexibility and extensibility of task schedulers. For example, inter-task synchronization may need to temporarily remove some tasks from the queue of ready tasks or reschedule a task (i.e. remove the task and schedule it again with other deadline/priority).

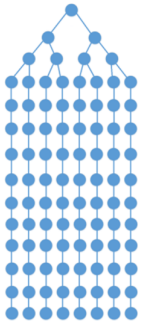

Fig. 3. Rocket Queue architecture with three duplicating levels [30, 31].

4. Proposed task scheduler

This paper presents two versions of tasks schedulers. Both versions implement EDF algorithm, which is known to be optimum solution for hard RT systems in terms of its decision-making [1-4]. The difference between these two versions is that one version supports CPUs that can execute two tasks in parallel (called dual-core CPUs) [26-28], while the second version of the scheduler supports CPUs that can execute four tasks in parallel (called quad-core CPUs) [37].

The proposed schedulers are using one global EDF-based schedule that is shared for all CPU cores. The top *M* tasks with the earliest deadline values are selected for running at the present time, where *M* is the number of CPU cores. The users of the proposed schedulers do not assign real-time tasks to specific CPU cores, like it is common in hypervisors nowadays. Instead of that, the CPU power (i.e. computing resources) are shared for all real-time tasks, which balances the usage of all CPU cores automatically, resulting in the maximum CPU utilization at all times.

The proposed task schedulers are designed as a coprocessor unit that provides two instructions:

- *schedule_task* – this instruction is used to add and schedule a new task to the scheduler. The scheduler updates the list of scheduled tasks and if the new task has a stricter deadline constraint than any of the currently running tasks, then a preemption is performed in the scheduler. If a preemption occurs, the scheduler informs the relevant CPU core to perform a task switch.
- *kill_task* – this instruction is used to deschedule one of the existing scheduled task among the currently running tasks. The running task is removed from the scheduler and the task with the earliest deadline among all ready tasks is selected for execution.

4.1. Top module of the scheduler

The top module of both versions of the task scheduler (i.e. dual-core scheduler and quad-core scheduler) is consisting of the following three components:

- Ready Queue module – contains ready tasks (i.e. tasks that are ready for execution but are not running)
- Running Tasks module – contains running tasks
- Semaphore module – handles conflicts of attempts to use the task scheduler by more CPU cores at the same time

The organization of these three blocks within the top module for dual-core version of the scheduler is described by a block diagram displayed in Fig. 4. 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 – Running Tasks component which 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 preempted. This component also reads the task with the lowest deadline value among the ready tasks (from the Ready Tasks). The control signals from Semaphore module are driven to the Ready Tasks indirectly through the Running Tasks module.

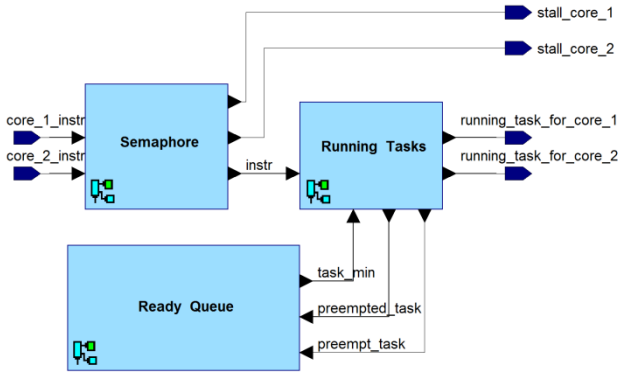

Fig. 4. Top level module for dual-core version of the scheduler.

The top module of the quad-core version of the scheduler is very similar to the previously presented dual-core version. The only difference is that the interfaces of the top module, Semaphore and Running Tasks components are extended for communication with four CPU cores. The block diagram for the quad-core version of the scheduler is displayed in Figure 5.

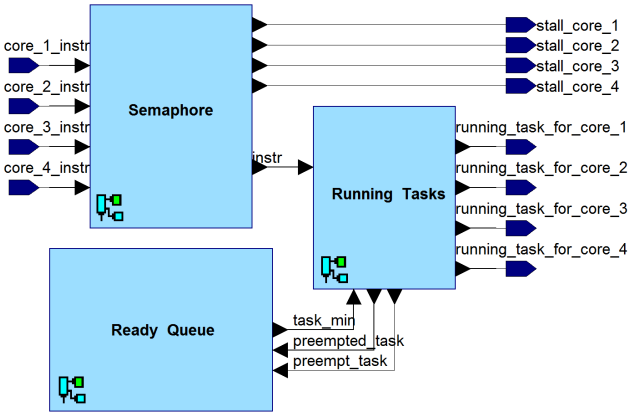

Fig. 5. Top level module for the quad-core scheduler.

4.2. Running Tasks module

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. Whenever any of the running tasks is killed, the task with the earliest deadline among the ready tasks is inserted to the Running tasks module. 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 earlier deadline than any of the running tasks, then this task replaces 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 stores the preempted task, otherwise the new task is inserted to the Ready Tasks module. This module maintains two remaining tasks – running tasks (tasks that are being executed by the CPU cores). The running tasks are using different logic that ensures that there are no redundant task switching operations. The new architecture is shown in Fig. 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 [26-28].

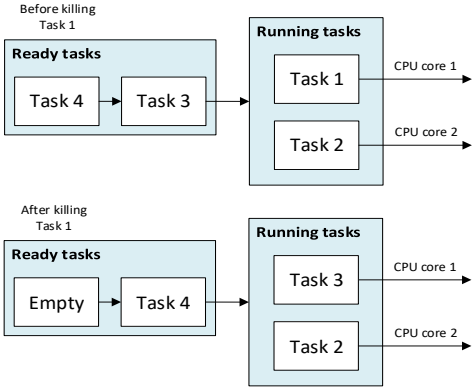

Fig. 6. Example of task kill operation in Running Tasks [26-28].

The second example (illustrated in Fig. 7) is the 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, the 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 [26-28].

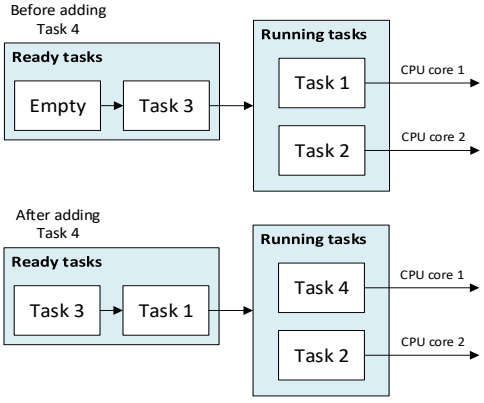

Fig. 7. Example of task add operation in Running Tasks [26-28].

Fig. 8 shows the logic circuit that implements the Running Tasks module with support of two CPU cores. The circuit contains four sets of registers (task ID of CPU core 1, task deadline of CPU core 1, task ID of CPU core 2, and task deadline of CPU core 2) and one 1-bit register (in the upper-right part of the circuit) that remembers which of these two tasks has higher deadline value. For this comparison, one comparator is used. The second comparator is used for comparison of the task with the higher deadline value with deadline of the new, incoming task.

The NAND and AND logic gates are used only for enabling of the registers that hold the running tasks.

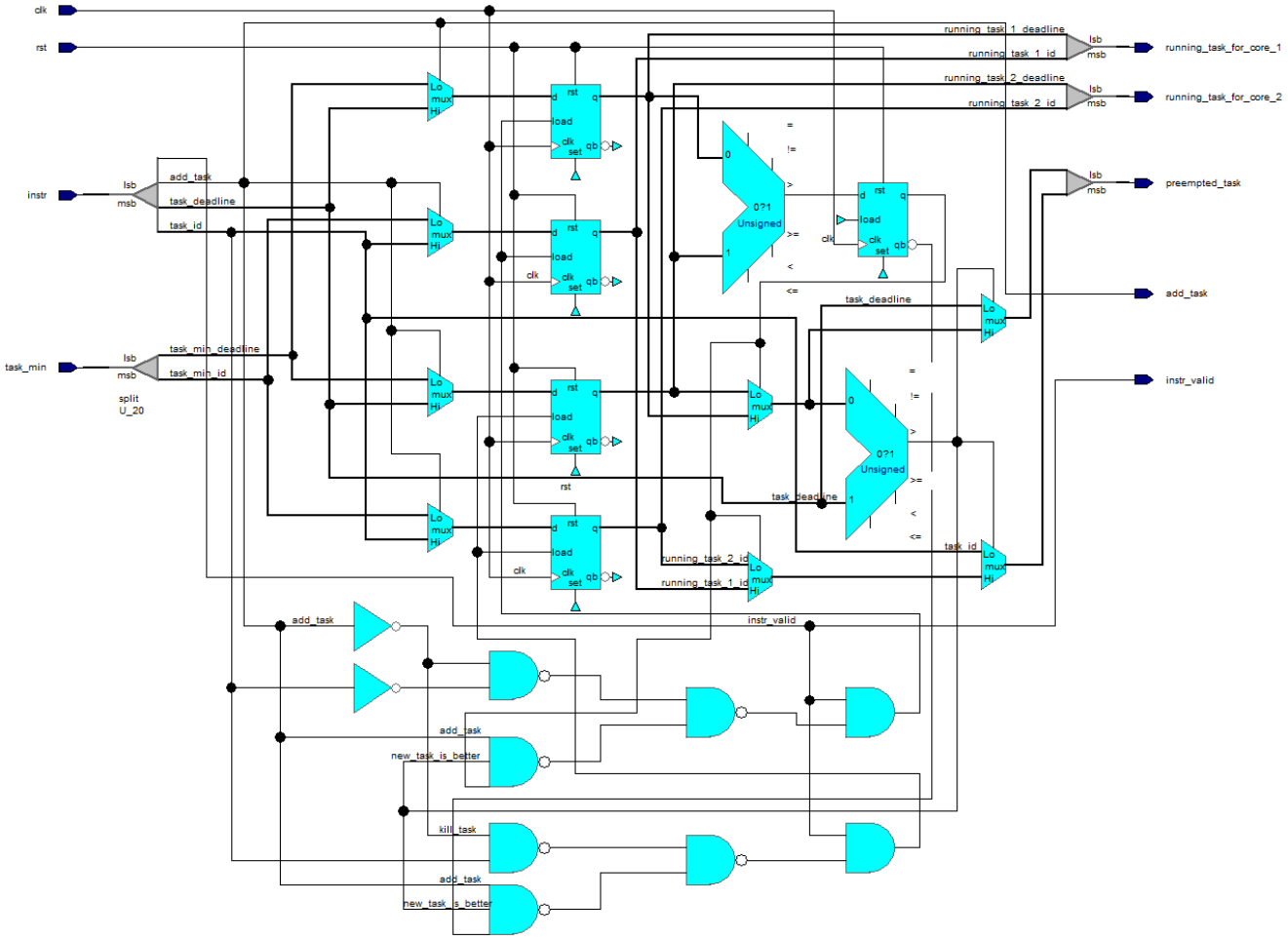

Fig. 8. Logic circuit of Running Tasks module.

For quad-core CPU support, the Running Tasks component has to be extended so that it holds four tasks rather than only two tasks. These tasks are the tasks that are supposed to be running in the CPU cores. The dual-core version of the Running Tasks component contained two comparators for comparison of the deadlines between the *running_task_for_core_1*, *running_task_for_core_2* and the *new_task* provided from the Control Unit. The decision logic within the Running Tasks component needed only one clock cycle in dual-core version.

In quad-core version of the Running Tasks component, there are five comparators needed. Due to the critical path length requirements for the combination logic (mainly consisting of the comparators), the decision logic is performed in two clock cycles. In order to remember the temporary results from the first clock cycle, additional (intermediate) registers containing two tasks and two bits are needed. The first clock cycle is used for performing two comparisons in parallel – compare *running_task_for_core_1* to *running_task_core_2*, and compare *running_task_for_core_3* to *running_task_core_4*. The tasks with the higher deadline values are being stored to the intermediate registers and two additional bits are used for storing the identification of the tasks within this module. The second clock cycle replicates the original logic that was used in the Running Tasks component designed for dual-core CPUs. This means that the deadlines from the temporary results (stored in intermediate registers) are compared to each other and to the new task provided from the Control Unit. According to these computations, preemption can occur, which means that one of the running tasks can be preempted and replaced by a new task. Regardless of the preemption occurrence, one of those tasks is further provided to the Ready Tasks module anyway [37].

4.3. Semaphore module

What if two or more CPU cores decide to add a new task or kill an existing task at the exactly 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 time (one instruction of the scheduling per 100 instructions), then the probability of a conflict occurrence is 0.01% for dual-core CPUs. So the conflict would occur very rarely but still it has to be taken into account.

If a conflict occurs, it is resolved dynamically by the semaphore module. This module is responsible for choosing which CPU core can use the coprocessor at the corresponding time. The second core will use the coprocessor 2 clock cycles later and thus, it must be stalled for 2 clock cycles in that case.

The Semaphore module for dual-core CPU systems consists mainly of multiplexers for selecting the instruction. In addition to that, there is only one D Flip-Flop (DFF) 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 the DFF. The best-case execution time is one clock cycle. However, it is important to keep in 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 four clock cycles because in the worst-case scenario, a conflict occurs every clock cycle. Fig. 9 shows the logic circuit that implements Semaphore module handling conflicts for two CPU cores.

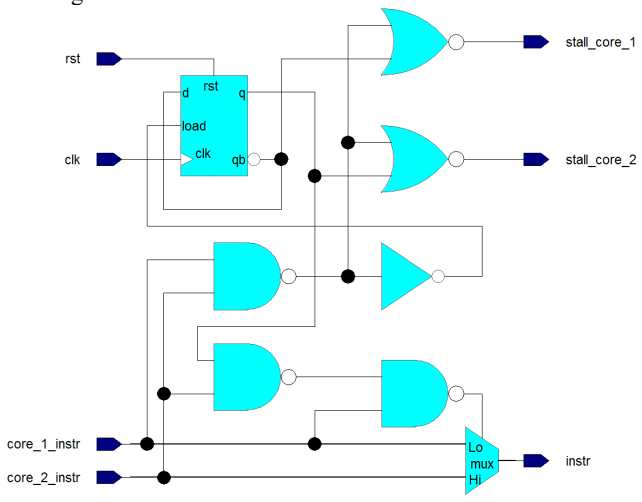

Fig. 9. Semaphore module circuit for dual-core systems.

For quad-core systems, the situation is much more complicated because there are more possible combinations of the conflicts that may occur. Even bigger conflicts (i.e. three or more CPU cores intending to use the coprocessor at the same time) can occur too. There are eleven possible combinations of conflicts in total, represented by the CPU core numbers. There exist six versions of conflicts when two CPU cores are in conflict: CPU core 1 and CPU core 2 conflict named as *1-2,* CPU core 1 and CPU core 3 named as *1-3,* CPU core 1 and CPU core 4 named as *1-4,* CPU core 2 and CPU core 3 named as *2-3,* CPU core 2 and CPU core 4 named as *2-4,* CPU core 3 and CPU core 4 named as *3-4.* These four versions of conflicts can occur when three CPU cores are trying to use the scheduler at the same time: CPU core 1, CPU core 2 and CPU core 3 conflict named as *1-2-3,* CPU core 1, CPU core 2 and CPU core 4 conflict named as *1-2-4,* CPU core 1, CPU core 3 and CPU core 4 conflict named as *1-3-4* and CPU core 2, CPU core 3 and CPU core 4 conflict named as *2-3-4.* The last possible combination is when all four CPU cores are trying to use the scheduler at the same time, which is named as *1-2-3-4*.

There are two requirements for the semaphore module, primary and secondary. The primary requirement is that there is specified a maximum possible number of delays (CPU stalls) caused by the conflicts and that this number is relatively low. Such a requirement is crucial because the scheduler is intended for real-time systems. The secondary requirement is fairness from the point of view of the CPU cores – each CPU core has approximately the same amount of possibilities to win to use the scheduler instantly [37].

The proposed solution for the new Semaphore module consists of a 2-bit counter that is used for representation of four states. These four states are called: 1234, 2143, 3412 and 4321. Each of these states implicitly specifies the priority order that is used for selecting a winner whenever any of the eleven possible conflicts occurs. For example, the 1234 state means that the CPU core 1 has higher priority than core 2, CPU core 2 has higher priority than core 3, and CPU core 3 has higher priority than core 4. Whenever a conflict occurs, the state is changed to the next one by incrementing the 2-bit counter. We decided to reduce the total number of 24 possible permutations or priority orders to only 4 orders defined by the four states because in this way, the state machine responsible for decision of which CPU wins the conflict is much simpler, resulting in simpler design and smaller hardware. The four orders were chosen so that these orders are symmetric, fair and they are rotating after every conflict. Whenever a conflict occurs, the order is changed to the next order by updating the state machine moving from the current state to the next state.

The states are specified by 2-bit counter in the following way [37]:

- value “00” represents state/order 1234. The next value is “01”.
- value “01” represents state/order 2143. The next value is “10”.
- value “10” represents state/order 3412. The next value is “11”.
- value “11” represents state/order 4321. The next value is “00”.

All combinations of conflicts with respect to the actual state and the corresponding winners are listed in Table I. Each line of the table represents a possible scenario of conflicting CPU cores. The columns represent the four possible orders, where one of them is selected at a given time depending on the current state of the state machine. One can observe that both requirements for the Semaphore are met because the maximum possible number of losses for any CPU core is three in a row (i.e. the CPU core can lose 0, 1 2 or 3 times at most), and the winning of CPU cores is evenly distributed. Due to the rotating behavior of states/orders, it is guaranteed that one instruction will take 2*M* clock cycles in the worst-case scenario (the case when all CPU cores want to use the scheduler all the time), where *M* is the number of CPU cores. The best-case scenario is 2 clock cycles. Thus, for quad-core CPUs, one instruction can take 2 to 8 clock cycles depending on the occurrence of the conflicts.

TABLE I. TABLE OF WINNERS FOR QUAD-CORE SEMAPHORE

	1234	2143	3412	4321
1-2	1	2	1	2
1-3	1	1	3	3
1-4	1	1	4	4
2-3	2	2	3	3
2-4	2	2	4	4
3-4	3	4	3	4
1-2-3	1	2	3	3
1-2-4	1	2	4	4
1-3-4	1	1	3	4
2-3-4	2	2	3	4

	1234	2143	3412	4321
1-2-3-4	1	2	3	4

Based on the description above, Fig. 10 shows a block diagram of the Semaphore module for quad-core CPUs. This module consists of Conflict Detector module, Winner Selector module, AND gate, two D-FFs, and three multi-bit multiplexers. Both, Conflict Detector and Winner Selector modules, need from the input instructions only the bit that specifies, whether the instruction is valid (i.e. whether the CPU core is trying to use the task scheduler or not). The Winner Selector module performs the decision, which CPU instruction is selected among the currently valid instructions. The decision is represented by signals *SEL1*, *SEL00* and *SEL01*, which are used as control inputs for the multiplexers that select one of the CPU instructions and provides the selected instruction to the output called *instr*. The *instr* output is used by the Running Tasks module.

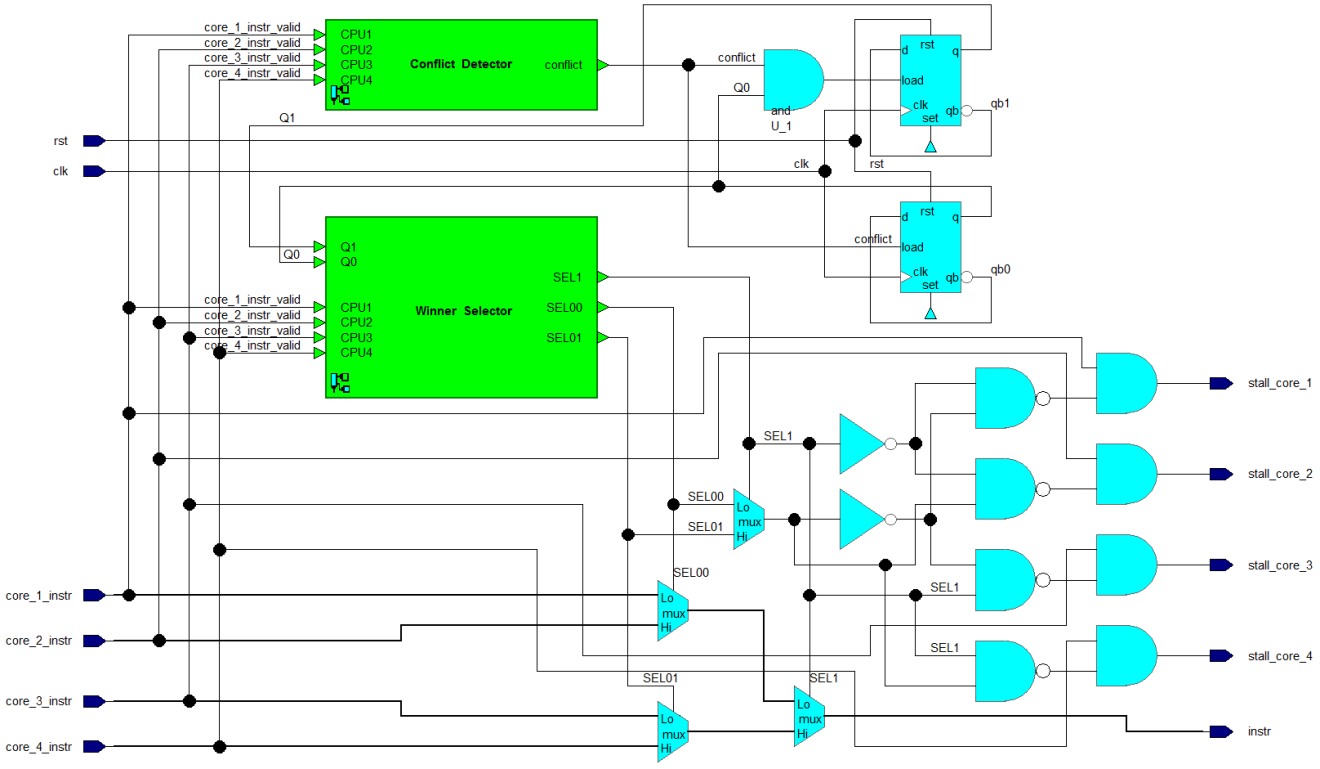

Fig. 10. Block diagram of Semaphore module.

Fig. 10 also shows that the Semaphore module provides four 1-bit output signals to the CPU cores, each for one CPU core. These signals are called *stall_core_#* and they are used for informing the particular CPU core that its request to use the task scheduler has been rejected due to a conflict with another CPU core. The CPU core that receives the "stall" signal should wait until the conflict is resolved. Eventually, the CPU can execute other instructions while waiting for the conflict to be resolved. The *core_#_instr_valid* and *stall_core_#* signals are used as a handshaking mechanism between the CPU and the scheduler.

Fig. 11 shows the logic circuit that represents the Conflict Detector module. The whole circuit consists of six 2-input NAND gates and one 6-input NAND gate. The module detects the situation, when at least two CPU cores want to use the scheduler at the same clock cycle. Therefore, whenever there are at least two inputs driven by logic 1, then the output *conflict* shall be logic 1. If it is not the case or only one CPU core has a valid instruction, then the output value shall be logic 0.

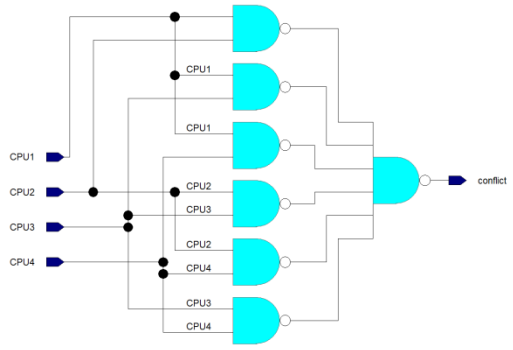

Fig. 11. Logic circuit of Conflict Detector module.

The Winner Selector module is displayed in Fig. 12. The *SEL1* output is generated by three 2-input NOR gates and one 3-input NOR gate. The *SEL00* and *SEL01* outputs require two 2-input NAND gates each. This circuit contains two inverters for generating negated inputs too. The Winner Selector module performs the same decision logic that was described in Table I.

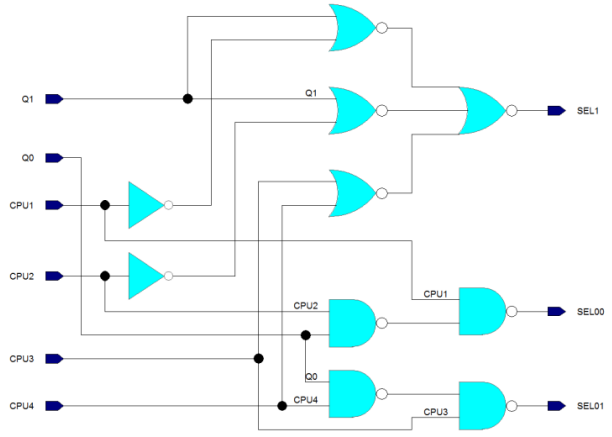

Fig. 12. Logic circuit of Winner Selector module.

4.4. Ready Queue module

The Ready Queue module is responsible for storing and sorting of all tasks that are ready for execution. This module is implemented as a sequential circuit, which always provides on the output the task with the earliest deadline among all tasks stored in this circuit. Thus, the task with the smallest deadline value associated to this task is always at the beginning (and output) of the Ready Queue. The Ready Queue module can accept a request for a inserting a new task into the queue but only one at a time. Alternatively, the task with the earliest deadline (i.e. at the beginning of the queue) can be popped, which means that this task is removed from the queue and the rest of the queue is updated. Whenever either a new task is inserted or the output task is removed, the queue has to reorganize in such a way that a new task with the earliest deadline is selected for the queue output (i.e. the queue keeps updating/reorganizing).

The Ready Tasks module alone represents the core EDF functionality, which is responsible for answering the question: Which task has the earliest deadline? This is the task that is supposed to be selected for execution. Even when the scheduler has already the Running Tasks module that is supposed to contain the tasks that are selected for the execution, the Running Tasks module needs to know, which task among all ready tasks is the best candidate for execution next. For this selection, the Ready Tasks module provides the answer.

As it was already mentioned in Section 3, there exist multiple architectures for implementation of sorting MIN/MAX queues. In [26-28], the Ready Tasks module was implemented using Shifts Registers architecture. In this paper, the proposed Ready Tasks module is implemented by Heap Queue architecture, which is much more efficient architecture that should result in much better task scheduler in terms of chip area, power consumption and timing as well. The Heap Queue 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 only. This means that each level is duplicating the number of items that can be stored within such s 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.

The duplicating levels of Rocket Queue and Heap Queue architectures form a binary tree that realizes a heapsort algorithm displayed in Fig. 13 [18].

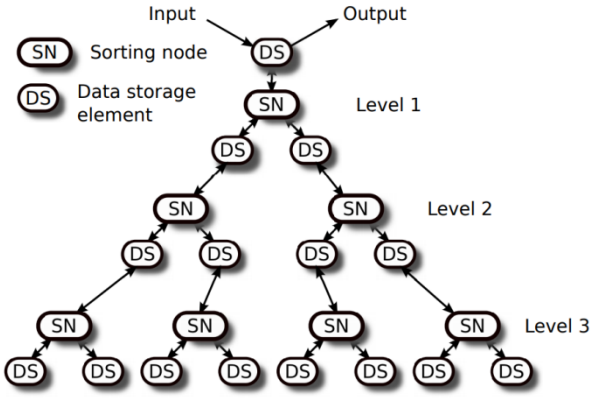

Fig. 13. Heapsort algorithm based on a binary tree [18].

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 informatics theory too [33, 34].

Fig. 14 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 neighboring 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 particular 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 [33, 34].

One can also notice that the Control Unit performs a combination of several Sorting Nodes employed in heapsort algorithm displayed in Fig. 13. 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 a single Control Unit saves significant portion of combinational logic within the queue [30, 31].

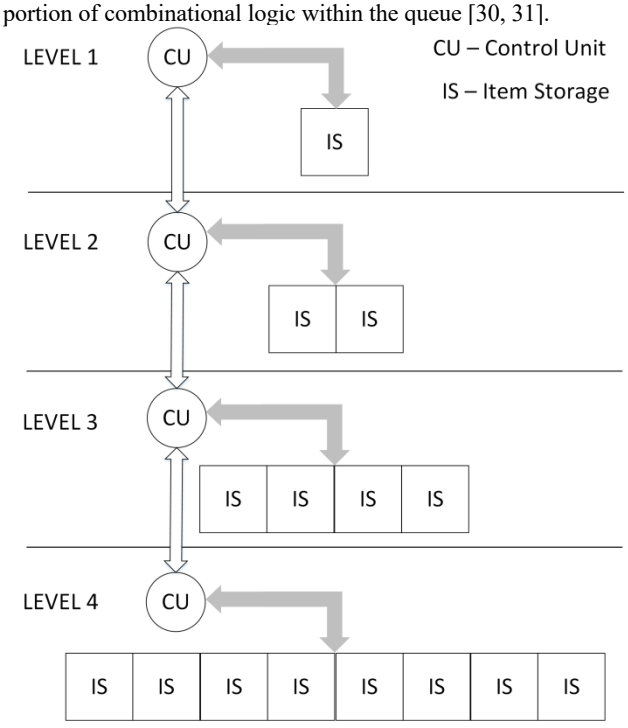

Fig. 14. Top level module for quad-core version of the scheduler [33, 34].

Since the Heap Queue architecture uses 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 [30, 31]. 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 usage of the MIN/MAX queue, this limitation may be acceptable or not. Thus, the Heap Queue architecture is suitable only for those cases, 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 cases, the Rocket Queue architecture remains to be the optimum solution for hardware acceleration (e.g. Worst-fit memory allocation algorithm) [33, 34].

The Heap Queue architecture is very similar to the DP RAM Heapsort architecture [18]. 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 [30, 31]. 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 [33, 34].

The items used in Heap Queue consist of two values – *ID* and *DATA*. The *ID* is used for identification of the item and the *DATA* is used for sorting the items within the queue. The EDF algorithm sorts tasks, where each task has its unique *ID* and one deadline value. Therefore, the implementation of EDF algorithm is achieved when Heap Queue item represents one EDF task, which means:

- *ID* of the item in Heap Queue is used as the task ID in EDF-based Ready Tasks module.
- *DATA* of the item in Heap Queue is used as the deadline value in EDF-based Ready Tasks module.
- *schedule_task* operation of EDF is using the INSERT instruction of Heap Queue.
- *kill_task* operation of EDF is using the POP instruction of Heap Queue.

5. Verification

The proposed task scheduler and its variations were described in SystemVerilog language and then, verified by simulations in a form of a coprocessor unit. The following variations of the scheduler were verified:

- Proposed task scheduler based on Heap Queue for dual-core CPUs
- Proposed task scheduler based on Heap Queue for quad-core CPUs
- Existing task scheduler based on Systolic Array
- Existing task scheduler based on Shift Registers for dual-core CPUs

Besides SystemVerilog language, a simpler version of Universal Verification Methodology (UVM) was used for the verification phase as well. Since the interface of our coprocessor unit is relatively simple, the UVM usage could be simplified too. In this case, one transaction in UVM is just one instruction performed in two clock cycles and thus, there is no need to use agents for interfacing the device under test (DUT). We used only one test procedure generating constrained random inputs, predictor and scoreboard. The test procedure is generating

millions of instructions with fixed opcode and UID but with randomized sorting values. The predictor is a module that predicts the DUT output according to the inputs (it behaves just like a DUT but at higher level of abstraction similar to high-level software languages). The description of the predictor is pure sequential and high level. The predictor uses SystemVerilog queue structure and *sort()* function for ordering the items in the queue. The testbench used for the verification is shown in Fig. 15.

The correct behavior of all the designed coprocessors was verified through 1 000 000 test iterations, each consisting of 510 instructions randomly generated by the test procedure. In this test, 50% of instructions were *schedule_task* and the other 50% of instructions were *kill_task*. Full capacity of the Ready Tasks module was used in these tests. The following configuration parameters were used for the coprocessor verification: 8-bit ID values, Ready Queue Capacity set to 255 and 32-bit width of random deadline values.

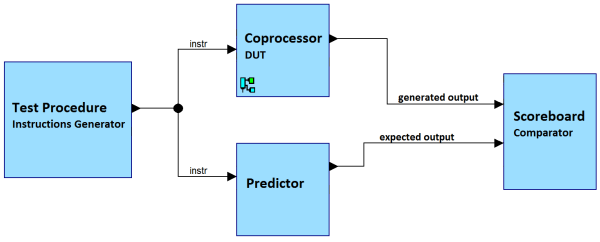

Fig. 15. Test bench architecture.

6. Synthesis results

We have performed an FPGA synthesis of four task schedulers in total. Two of them are the proposed task schedulers based on Heap Queue (Proposed Solution CPU2 is a version designed for two CPU cores and Proposed Solution CPU4 is designed for four CPU cores). The other two solutions are existing schedulers, Existing SA Solution CPU1 and Existing SR Solution CPU2 from [26-28]. The Existing SA Solution CPU1 is based on Systolic Array architecture and supports only one CPU core. The Existing SR Solution CPU2 is based on Shift Registers architecture and supports two CPU cores using the semaphore approach [26-28]. The target device for the synthesis was Intel FPGA Cyclone V (5CSEBA6U23I7), and the clock frequency of 100 MHz was targeted. A comparison has been performed for Adaptive Logic Module (ALM) consumption.

In Table II, the comparison of ALM consumption for various maximum numbers of tasks is presented, where all four schedulers are compared. The Ready Queue Capacity number represents the configuration of Ready Queue in the scheduler, which defines the maximum number of tasks that can be stored into Ready Queue of the scheduler. The Ready Queue capacity is varying from 31 to 32767 and bit width of task ID is always the minimum possible (e.g. 5 bits for 31 tasks or 6 bits for 63 tasks). The total number of tasks that can be stored into the scheduler are increased by two for the Proposed Solution CPU2 and by four for the Proposed Solution CPU4 due to the storage of tasks in the Running Tasks module. The deadline bit-width is 32 for all schedulers. The synthesis of the existing solution was not successful for higher Ready Queue Capacities due to too high consumption of the ALM resources.

TABLE II. CONSUMPTION OF ALM RESOURCES

Ready Queue Capacity	*Proposed Solution CPU2*	*Proposed Solution CPU4*	*Existing SA Solution CPU1*	*Existing SR Solution CPU2*
31	1 544	1 624	2 458	3 629
63	1 968	2 058	4 874	6 906
127	2 296	2 391	10 119	13 754
255	2 913	3 011	21 757	26 255
511	3 403	3 503	-	-
1023	3 801	3 906	-	-
2047	4 312	4 418	-	-
4095	4 492	4 600	-	-
8191	5 092	5 191	-	-
16383	5 592	5 702	-	-
32767	6 163	6 283	-	-

In Table III, the comparison of RAM bits consumption for various maximum numbers of tasks is presented, where all four schedulers are compared. The existing scheduler based on Systolic Array does not consume any RAM bits in FPGA because this architecture simply does not use any memories as storage of scheduled tasks, but only registers. One can see that the consumption of RAM bits for the proposed task schedulers is increasing exponentially. This is expected because the Ready Queue Capacity is increasing exponentially as well. Thus, the memory consumption scales in fact linearly with respect to this parameter.

TABLE III. CONSUMPTION OF RAM RESOURCES

Ready Queue Capacity	*Proposed Solution CPU2*	*Proposed Solution CPU4*	*Existing Solutions*
31	912	912	0
63	2184	2184	0
127	4864	4864	0
255	10344	10344	0
511	21584	21584	-
1023	44600	44600	-
2047	91680	91680	-
4095	187976	187976	-
8191	384568	384568	-
16383	785960	785960	-
32767	1605144	1605144	-

In addition to the FPGA synthesis, we also performed ASIC synthesis for the same task schedulers (except for Existing SR Solution CPU2), in order to analyze their scalability when implemented in ASIC as well. For this purpose, we decided to use 28 nm TSMC High Performance Mobile process with 2 GHz clock frequency and 0.9 V power supply voltage. The chip area (in μm^2) results are displayed in Table IV. The results show that the proposed task schedulers require lower chip area than the existing Systolic Array task scheduler as long as the Ready Queue Capacity is 63 or more.

Since the Heap Queue architecture is using SRAM-based memories of various depths, too small memories are synthesized into flip-flops. One flip-flop bit is significantly less efficient than one SRAM bit in terms of chip area and power consumption. The Heap Queues with smaller capacity are using mostly flip-flop based memories, which causes that the overall results are relatively poor. However, as queue capacity is increasing, most of the memory bits are realized by SRAM. Therefore, the Heap Queue outperforms other architectures the most when higher queue capacity is selected.

TABLE IV. CHIP AREA CONSUMPTION

Ready Queue Capacity	*Proposed Solution CPU2*	*Proposed Solution CPU4*	*Existing SA Solution CPU1*
31	12 007	12 215	11 167
63	21 597	21 782	23 271
127	39 877	40 061	48 097
255	45 990	46 202	98 945
511	62 545	62 767	203 085
1023	81 177	81 390	415 358
2047	133 647	133 873	850 377
4095	236 564	236 794	1 738 448

The following table (Table V) shows the total power consumption for the synthesized task schedulers with respect to the Ready Queue Capacity. The total power consumption represents a sum of static power consumption (i.e. leakage power) and dynamic power consumption. For the dynamic power consumption, it is assumed that the queues are actively used once every 20 clock cycles (i.e. every 20th instruction of CPU is a valid instruction of the tested coprocessor). The results are presented in microwatts (µW).

TABLE V. TOTAL POWER CONSUMPTION

Ready Queue Capacity	*Proposed Solution CPU2*	*Proposed Solution CPU4*	*Existing SA Solution CPU1*
31	2 357	2 392	3 129
63	4 148	4 184	6 417
127	6 883	6 925	13 069
255	8 171	8 214	26 953
511	9 866	9 909	55 045
1023	11 721	11 760	111 747
2047	18 206	18 243	234 515
4095	30 765	30 807	478 279

The overall comparison of the synthesized task schedulers is presented in Table VI. One can conclude that the proposed task schedulers are more efficient in terms of chip area costs in ASIC implementation and ALM consumption in FPGA than the existing task scheduler. The resource costs of the proposed scheduler designed for four CPU cores are only negligibly higher than the costs of the scheduler designed for two CPU cores. However, the performance gain caused by running four tasks in parallel instead of only two tasks is significant, as the total performance can be increased by 100%. Both proposed task schedulers are better than the existing task scheduler offering higher performance due to the fact that the proposed task schedulers are compatible with running of two or four CPU tasks in parallel, increasing the performance of the whole real-time system to 200% (when two CPU cores are used) or to 400% (for four CPU cores).

If we compare the proposed HW-implemented task schedulers with software implementation of EDF, it is clear that the proposed schedulers execute EDF instructions in constant and much shorter time regardless of the number of tasks that are scheduled within the system, which is impossible for software implementations of EDF schedulers.

From the results presented in Table VI, it is expected that the proposed solution is also applicable and scalable in terms of chip area costs for even higher amount of CPU cores too. However, the design effort due to design complexity, the maximum number of CPU stalls in a row, the worst-case execution time and critical path length (which affects the maximum clock frequency) are the most limiting factors for scaling to more than four CPU cores.

TABLE VI. OVERALL COMPARISON OF THE SCHEDULERS

Criterion	Scheduler Version		
	software scheduler	*existing SA scheduler*	*proposed schedulers*
Chip Area Costs	no	100%	46% *
Best Case Execution Time	*tens of clock cycles*	2 clock cycles	2 clock cycles
Worst Case Execution Time	*thousands of clock cycles*	2 clock cycles	2 clock cycles
CPU Cores	1	1	2 or 4
CPU Throughput	80% to 98%	100%	200% or 400%

* applies for the Ready Queue Capacity parameter of 255.

7. Discussion and conclusion

Two different improvements of hardware-implemented EDF-based task schedulers were proposed and presented: resource cost decrease caused by usage of Heap Queue sorting architecture for implementation of Ready Queue in the scheduler and overall real-time system performance increase caused by providing a support for CPUs that can execute two or four real-time tasks simultaneously.

The synthesis results show that the change of sorting architecture of Ready Queue module from Systolic Array to Heap Queue can reduce the resource costs of the task scheduler by 84% of ALMs in FPGA and 84% of chip area in ASIC, depending on the scheduler capacity.

The second improvement is the added support of CPUs that can run two or four tasks (programs) in parallel. Due to the parallelism offered by such CPUs, the overall real-time system performance is significantly improved at a cost of negligible increase of resource costs needed to implement this support. In multi-core systems, a conflict can occur whenever at least two CPU cores attempt to use the coprocessor at the same clock period. Our research was focused on solving this problem efficiently in terms of both performance, determinism and chip area costs. The 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 cores are losers that are stalled in the meantime. Each core is stalled at most three times in a row because the selection of the winner and loser is always switched deterministically. This ensures fairness of the approach in any cases too. The semaphore approach can be theoretically further used for even more CPU cores (e.g. 8 or 16), however, the design and timing complexity of the semaphore is not very well scalable for such extensions at the moment. The proposed solution is definitely not suitable for larger numbers of CPU cores (e.g. 16 or more).

In terms of timing, all presented schedulers were synthesized for 100 MHz clock frequency in FPGA and 2 GHz clock frequency in 28 nm ASIC. These schedulers perform each instruction (*task schedule* or *task kill*) in constant time, regardless of the current amount of tasks present in the scheduler and the scheduler capacity (i.e. the maximum number of tasks). Both response time and throughput are two clock cycles. Compared to the software implementation of EDF algorithm, this is a huge benefit in terms of scheduling performance and system determinism.

According to the synthesis results, it has been shown that the proposed task scheduler improvements based on using new Heap Queue architecture and support of two or four CPU cores can be used for significant improvement of performance, determinism and reliability of task schedulers used in systems and applications that belong to safety-critical and hard real-time domain. Consequently, the whole real-time system using such an improved task scheduler would be improved too. For example, more tasks could be executed without causing deadline misses, or these tasks could contain more time-consuming features, while the execution on time would be still guaranteed by the improved task scheduler and due to the fact that more powerful CPUs could be used.

The proposed task schedulers were designed in a form of coprocessor units that can be implemented either together with an open-source CPU (e.g. RISC-V CPUs) on a single ASIC chip, or the coprocessor can be implemented in FPGA that is closely connected to an existing CPU (typically called FPGA SoC). The proposed architecture of task scheduling coprocessors is designed to be versatile. The architecture remains unchanged regardless of what application is the scheduler used for, provided that the maximum number of tasks and deadline bit width parameters are large enough.

The proposed task schedulers are not intended to fully replace existing software implementations of operating systems, rather to be combined with these software solutions in a reasonable way. This means that selected functions of software-implemented operating systems can call the instructions of the proposed coprocessors that implement the hardware-accelerated task scheduling. The proposed schedulers do not distinguish between periodic, aperiodic and sporadic tasks. All tasks are scheduled and prioritized according to their deadlines only, regardless of whether the task is periodic or not. The periodical behavior is supposed to be handled on higher level (in software part of the OS). Thus, whenever a new task is scheduled using the *schedule_task* instruction, it is expected to be executed only once from the view of the proposed HW-implemented task scheduler. Periodic tasks can be achieved by software extension of the HW-implemented task scheduler by periodic usage of the *schedule_task* instruction in the software part of the operating system. Task synchronization and sharing mechanisms are not considered within the proposed HW-implemented task scheduler neither, and it is expected to handle these problems by software part of the OS. The software part can use the *kill_task* instruction to temporarily deschedule a running task followed by a *schedule_task* instruction using a new deadline for the task. In this way, any running task that is blocked by other tasks with later deadlines can be easily rescheduled in order to become unblocked.

From reliability point of view, the hardware implemented features can be used for increased reliability of real-time systems, as they can perform the same algorithm that can be implemented in software. Thus, a redundancy can be performed by combining the software implementation with hardware realization relatively easily as well [38, 39].

Acknowledgement

This work was supported in part by the Slovak Research and Development Agency under grant APVV-15-0254 and by the Slovak Republic under grant VEGA 1/0905/17.

References

[1] R. Mall, Real-Time Systems: Theory and Practice, 2nd edition, 2008, ISBN 978-81-317-0069-3.

[2] C. A. O'Reilly, A. S. Cromarty, “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, doi: 10.1117/12.948443.

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

[4] G.C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 2011, doi: https://doi.org/10.1007/978-1-4614-0676-1.

[5] S. Heath, Embedded Systems Design, Newnes, 2003, ISBN: 0750655461.

[6] I. Lee, J. Y.-T. Leung, S. H. Son, Handbook of Real-Time and Embedded Systems, Chapman & Hall/CRC, 2007, ISBN: 9781584886785.

[7] M. Joseph, Real-time Systems Specification, Verification and Analysis, Prentice Hall International, London, 2001.

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

[9] M. Pohronská, Utilization of FPGAs in Real-Time and Embedded Systems, in M. Bielikova, ed.,Proceedings in Informatics and Information Technologies Student Research Conference, Vydavateľstvo STU, 2009.

[10] C. Ferreira, A. S. R. Oliveira, Hardware Co-Processor for the OReK Real-Time Executive, 2010.

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

[12] A. B. Lange, K. H. Andersen, U. P. Schultz, A. S. Sorensen, HartOS - a Hardware Implemented RTOS for Hard Real-time Applications, 2012, s. 207-213.

[13] S. Liu, Y. Ding, G. Zhu, Y. Li, Hardware scheduler of Real-time Operating. in: Advanced Science and Technology Letters Vol.31, 2013, s. 159-160.

[14] G. Bloom, G. Parmer, B. Narahari, R. Simha, Real-Time Scheduling with Hardware Data Structures, 2010.

[15] M. Varela, R. Cayssials, E. Ferro, E. Boemo, Real-time scheduling coprocessor for NIOS II processor, Proc. VIII Southern Conf. Programmable Logic, 2012, pp. 1-6, doi: 10.1109/SPL.2012.6211775.

[16] R. Chandra, O. Sinnen, Improving Application Performance with Hardware Data Structures, 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), Atlanta, GA, 2010, pp. 1-4, doi: 10.1109/IPDPSW.2010.5470740.

[17] S. W. Moon, Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches, IEEE Transactions on Computers, 2000, pp. 203-212, doi: 10.1109/RTTAS.1997.601359.

[18] W. M. Zabołotny, Dual port memory based heapsort implementation for fpga, Proceedings of SPIE, 2011, doi: 10.1117/12.905281.

[19] Y. Tang, N. W. Bergmann, A Hardware Scheduler Based on Task Queues for FPGA-Based Embedded Real-Time Systems, IEEE Transactions on Computers, 2015, pp. 1254-1267, doi: 10.1109/TC.2014.2315637.

[20] J. Starner, J. Adomat, J. Furunas, L. Lindh, Real-Time Scheduling Co-Processor in Hardware for Single and Multiprocessor Systems, Proceedings of the EUROMICRO Conference, 1996, pp. 509-512, doi: 10.1109/EURMIC.1996.546476.

[21] S. E. Ong, S. C. Lee, SEOS: Hardware Implementation of Real-Time Operating System for Adaptability, Computing and Networking (CANDAR), 2013 First International Symposium, 2013, pp. 612-616, doi: 10.1109/CANDAR.2013.110.

[22] K. Kim, D. Kim, Ch. Park, Real-Time Scheduling in Heterogeneous Dual-core Architectures, Proceedings of the 12th International Conference on Parallel and Distributed Systems, 2006, doi: 10.1109/ICPADS.2006.90.

[23] L. Kohutka, Hardware task scheduling in real-time systems in IIT.SRC 2015, Student Research Conference, 2015.

[24] L. Kohutka, M. Vojtko, T. Krajcovic, Hardware Accelerated Scheduling in Real-Time Systems, Engineering of Computer Based Systems Eastern European Regional Conference, 2015, pp. 142-142, doi: 10.1109/ECBS-EERC.2015.32.

[25] L. Kohutka, V. Stopjakova, Hardware Accelerated Task Scheduling in Real-Time Systems, Adept, 2016.

[26] L. Kohutka, V. Stopjakova, Hardware-Accelerated Task Scheduling in Real-Time Systems: Deadline Based Coprocessor for Dual-Core CPUs, International Symposium on Design and Diagnostics of Electronic Circuits and Systems, 2016.

[27] L. Kohutka, V. Stopjakova, Task scheduler for dual-core real-time systems, 23rd International Conference Mixed Design of Integrated Circuits and Systems, 2016, pp. 474-479, doi: 10.1109/MIXDES.2016.7529789.

[28] L. Kohútka, V. Stopjaková, Improved Task Scheduler for Dual-Core Real-Time Systems, 2016 Euromicro Conference on Digital System Design (DSD), Limassol, 2016, pp. 471-478, doi: 10.1109/DSD.2016.44.

[29] F. Klass, U. Weiser, Efficient systolic arrays for matrix multiplication, in Proc. Int. Conf. Parallel Processing, Austin, Tex., Aug. 1991, vol. III, pp. 21-25.

[30] L. Kohutka, V. Stopjakova, Rocket Queue: New Data Sorting Architecture for Real-Time Systems, 20th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2017, pp. 207-212, doi: 10.1109/DDECS.2017.7934573.

[31] L. Kohutka, V. Stopjakova, A New Efficient Sorting Architecture for Real-Time Systems, 6th Mediterranean Conference on Embedded Computing (MECO), 2017, pp. 1-4, doi: 10.1109/MECO.2017.7977221.

[32] C. L. Liu and James W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, J. ACM 20, 1, 1973, pp. 46-61, doi:https://doi.org/10.1145/321738.321743.

[33] L. Kohutka, L. Nagy, V. Stopjaková, A Novel Hardware-Accelerated Priority Queue for Real-Time Systems, 2018 21st Euromicro Conference on Digital System Design (DSD), Prague, 2018, pp. 46-53, doi: 10.1109/DSD.2018.00023.

[34] L. Kohutka, V. Stopjakova, Heap Queue: A Novel Efficient Hardware Architecture of MIN/MAX Queues for Real-Time Systems, 2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Budapest, 2018, pp. 5-8, doi: 10.1109/DDECS.2018.00008.

[35] K. Churnetski, Real-time scheduling algorithms, task visualization, Computer Science Department Rochester Institute of Technology, 2006.

[36] A. Mohammadi, S. G. Akl, Scheduling Algorithms for Real-Time Systems, School of Computing, Kingston, Ontario, 2005, doi: 10.1.1.536.9002.

[37] L. Kohútka, 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, pp. 1-6, doi: 10.1109/AIEEE.2017.8270538.

[38] D. E. Rosenheim, R. B. Ash, Increasing reliability by the use of redundant machines, IRE Trans. on electronic computers, vol. EC-8, pp. 125-130, 1959.

[39] B. J. Flehinger, Reliability improvement through redundancy at various system levels, IBM J. Res. Develop., vol. 2, pp. 148-158, 1958, doi: https://doi.org/10.1147/rd.22.0148.