IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 11 December 2023, accepted 7 January 2024, date of publication 15 January 2024, date of current version 24 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3354178

RESEARCH ARTICLE

Exploration of Power-Savings on Multi-Core Architectures With Offloaded Real-Time Operating System

GÖKHAN AKGÜN[1], BOZHIDAR KOLAROV[1], HENDRIK KALBERLAH[1], CORNELIA WULF[1], MICHAEL WILLIG[1], (Graduate Student Member, IEEE), JENS RETTKOWSKI[2], AND DIANA GÖHRINGER[1,3], (Member, IEEE)

[1]Chair of Adaptive Dynamic Systems, Technische Universität Dresden, 01069 Dresden, Germany
[2]Department of Electrical Engineering, Fachhochschule Dortmund–University of Applied Sciences and Arts, 44139 Dortmund, Germany
[3]Centre for Tactile Internet with Human-in-the-Loop (CeTI), Technische Universität Dresden, 01069 Dresden, Germany

Corresponding author: Gökhan Akgün (goekhan.akguen@tu-dresden.de)

This work was supported in part by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy-EXC 2050/1, Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop (CeTI)," Technische Universität Dresden, under Project 390696704, and in part by the Open Access Funding by Saxon State and University Library Dresden (SLUB).

ABSTRACT A Real-time Operating System (RTOS) manages the execution order of tasks with a scheduling algorithm to meet timing requirements. The scheduler frequently checks for ready tasks during context-switching. However, high task numbers can cause longer processing time in this routine. RTOSs are mainly implemented in software, but reconfigurable computing enables offloading to reduce, e.g., the processing time of context-switching. On the other hand, optimizing the energy efficiency of running applications is desirable. Power-saving techniques allow adapting current dissipation to required operating conditions. However, unplanned use can lead to missed deadlines in real-time applications. Therefore, real-time capability and energy efficiency have to be appropriately balanced. This work explores the impact of power-saving techniques on real-time requirements while supporting RTOS with offloading methodologies. A mapping strategy assigns tasks to Processing Elements (PEs) based on task dependency, inter-task/processor communication, and power consumption metrics. A multi-core architecture is designed with a Network-on-Chip (NoC) and four PEs in a 2D-mesh topology. The master PE manages the system architecture, executes the mapping strategy, and dynamically scales voltage to reduce power consumption while running an RTOS. The task scheduling is offloaded to the co-processor. On the other hand, each slave PE executes assigned tasks with an RTOS and performs an inter-task/processor communication. The task scheduling here runs on the reconfigurable hardware. Each slave PE locally adapts power with frequency scaling and clock gating. The experimental results show that co-processor offloading reduces scheduling overhead by 26.58%, and hardware offloading reduces it by 33.33%. Additionally, the proposed solution has reduced overall power by 47.27% and energy consumption by 89.47%.

INDEX TERMS Dynamic voltage and frequency scaling (DVFS), field programmable gate array (FPGA), multi-core architecture, power management, real-time operating system (RTOS).

I. INTRODUCTION

Real-time systems are ubiquitous in safety-critical applications such as avionics, automotive, and robotics. The correctness of the real-time system depends on the results of its calculations and the calculated time. A missed predefined deadline leads to malfunctions of the entire system. Therefore, all time-sensitive computations in safety-critical applications have to be executed before their deadlines [1]. Real-time Operating Systems (RTOSs) manage

The associate editor coordinating the review of this manuscript and approving it for publication was Vincenzo Conti.

the execution order of tasks with their scheduling policy and can thus meet the timing requirements of safety-critical applications. However, RTOSs are mainly implemented in software and sequentially executed on processors. They suffer from computational overhead, jitter, and large memory footprint [2]. Computational overhead is induced through interrupt management, task scheduling, resource allocation, deadlock mechanism, and various Application Programming Interfaces (APIs) [2]. On the other hand, jitter occurs due to the number of tasks, resources required, and system state. For instance, task scheduling in RTOS checks in regular intervals for a context switch. The execution time of the context switch is deviant and thus leads to a non-deterministic behavior at run-time. It is a critical problem for hard real-time systems and should not occur. However, the occurring jitter can be shortened or eliminated with reconfigurable systems. Task scheduling can be implemented in hardware or on a co-processor [1]. The approach would check the need for a context switch in parallel and change tasks on processors if necessary.

On the other hand, RTOS can use the data from design space explorations to adjust performance/power and monitor the embedded system at run-time. Power-saving techniques usually adapt the power consumption of embedded systems to existing workloads and operating conditions. One such popular power capping technique is Dynamic Voltage and Frequency Scaling (DVFS). Depending on the embedded device, the power consumption can be optimized, for instance, with onboard voltage regulators or Mixed-Mode Clock Manager (MMCM) modules [3]. This dynamically scales voltage and frequency to a particular operating point to minimize power dissipation at run-time. However, frequency scaling leads to a longer or shorter execution of tasks in RTOSs. For instance, a faster execution of tasks results in a long idle period where processors are busy waiting for the next scheduled task. A slower operation may lead to deadline misses and affect the real-time capability of running applications. It is becoming increasingly crucial for battery-powered embedded devices to meet reliability, real-time capability, and energy efficiency requirements with existing RTOSs.

Meeting this requirement becomes even more challenging when applications run on multi-core systems. A single processor design has limitations, such as the sequential execution of an application that multi-core systems improve. The application can be split into parts and run simultaneously on multiple processors. That allows the execution time to be shortened and optimization to be performed using power-saving techniques. With single-core systems, there are also performance losses for applications. In this case, the processor has to interrupt the execution to check the current power dissipation and adjust power with saving techniques. This criterion can also be better controlled and managed in multi-core systems. A dependency arises when an application is divided into parts and distributed among processors. The execution of these parts then requires a specific execution order. Before distributing these parts among processors, task parallelism and interconnections need to be examined. Therefore, a mapping strategy for multi-core systems is needed that distributes tasks among processors according to their dependencies, execution time, and communication. It can also manage task execution on processors while dynamically adjusting frequency/voltage and meet real-time requirements.

In this work, we have investigated the impact of power-saving techniques on real-time capability while supporting RTOS with offloading methodologies. For this purpose, we have designed a multi-core system with an open-source RTOS. Since the scheduling decision and task selection at RTOS consume processing time, offloading methodologies are examined more closely in this work. Therefore, task scheduling runs in hardware and on a co-processor. This improved the overall performance of the context switch, as will be discussed in more detail later. In addition, applications can be split into parts and executed in parallel on multiple processors simultaneously. However, in this case, the individual parts have a dependency that affects the total execution time. Therefore, it is necessary to investigate how dependency, inter-task communication, and the resulting execution affect each other. For this reason, we have performed a mapping strategy on a processor that considers the mentioned criteria in the initialization phase and allocates tasks to processor cores accordingly. At the same time, this processor manages the total power consumption and checks that no deadlines are missed in RTOS running processors. For this purpose, the power dissipation is optimized by DVFS and clock gating at run-time. The contribution of this work can be summarized as follows:

- A heterogeneous multi-core architecture for real-time systems that performs power-saving techniques and applies offloading methodologies to enhance RTOS scheduling performance.
- A task mapping strategy that assigns tasks to slave Processing Elements (PEs) based on task dependency, inter-task communication, and power consumption metrics.
- An architecture where the master PE monitors system requirements and deploys voltage scaling while slave PEs perform tasks in real-time and apply frequency scaling and clock gating.
- A discussion that elaborates on the trade-off between real-time capability and energy efficiency in the proposed system.

Section II introduces the related work in the area of power-saving techniques on FPGAs, offloading methodologies on RTOSs, and our use case. Section III describes this work's proposed methodologies and contributions. Section IV gives an overview of implemented hardware and software architecture. All results in this work are presented in Section V. The work is concluded in Section VI.

TABLE 1. Comparison of related works with the proposed work. EDF - Earliest Deadline First, LST - Least Slack Time, F-PP - Fixed-Priority Preemptive.

Features	[4]	[5]	[6]	[7]	[8]	[3]	Proposed Solution
Platform	ARM Cortex-A8	ODROID-XU3	XC7Z020	XC5VLX50T	XC2VP7	XC7Z020	XC7Z020
Real-Time Capability	Soft	Hard	×	Hard/Soft	Soft	Soft	Hard/Soft
Scheduling Algorithm	EDF	Look-Ahead	Cyclic	EDF, LST	F-PP	F-PP	F-PP
Operating System	Ubuntu Linux	In-house	XtratuM	eCos	FreeRTOS	FreeRTOS	FreeRTOS
Offloaded Task Scheduling	×	×	×	✓	✓	×	✓
Voltage Scaling	×	✓	✓	×	×	✓	✓
Frequency Scaling	✓	✓	✓	×	×	✓	✓
Clock Gating	×	×	×	×	×	×	✓
Power-Saving on Processor	✓	✓	✓	×	×	✓	✓
Power-Savings on FPGA	×	×	×	×	×	✓	✓

II. RELATED WORK

While offloading methodologies improve the scheduling performance of an RTOS, power-saving techniques lead to minimizing power dissipation. A Network-on-Chip (NoC)-based multi-core system enables an application to be split into tasks and executed simultaneously on PEs. A management unit can control the power-saving techniques, dynamically monitor power consumption, and assign tasks to PEs while considering real-time and power constraints. Table 1 categorizes representative works from the literature with different objectives considered in this work. Some results [7], [8] offloaded RTOS task scheduling into reconfigurable systems without applying power-saving techniques. Other works [3], [4], [5], [6] applied power-savings with operating systems, but only [3] used them with an open-source RTOS. None of the works used clock gating for power optimization under the considered constraints (all features shown in Table 1). A mapping strategy was only proposed in [5] for multi-core systems. However, it did not consider inter-task communication through a NoC. This section gives an overview of the state-of-the-art related to the proposed work in offloading methodologies, power-saving techniques, and task mapping strategies for multi-core systems.

An RTOS has a kernel with a scheduling algorithm. It frequently checks whether a context switch is required, selects the following execution task, and stores the context into its stack. Thus, designers can implement application functions without considering the synchronization, resource usage, and ordering of function calls. However, the scheduling process may be less predictable, and a high context switching rate may lead to overhead in the system because of complex and comprehensive algorithms for, e.g., control applications [7], [9]. Researchers have proven that task scheduling can be performed faster and the overall behavior more predictable when offloaded from running processors. This approach reduces run-time overhead and event response time due to smaller critical sections [10], [11]. Moreover, the system does not frequently have to be suspended by the system tick interrupted by an Interrupt Service Routine (ISR), which increases processor utilization and improves scheduling predictability [7]. For this purpose, task scheduling can be performed on the co-processor or hardware. There are also related works (FASTHARD [12], Silicon TRON [13], δ-Framework [14]) where the complete RTOS is replicated as an entity in hardware. Other works have offloaded and executed dedicated kernel services in hardware [15], [16], [17], [18]. For instance, in [15], the scheduler, inter-task synchronization (semaphore and mutex), and inter-task communication (mailbox and queue) ran in hardware. The proposed approach improved the performance of computing time by 31.67%. The hardware-based task scheduling reduces the computing time for the context switch, which was decreased by 83.28% in [7] for eCos with eight tasks, and 87.35% in [9] for μC/OS-II with three tasks. In descending order, the FreeRTOS scheduler checks all priority levels (starting from the highest priority) to release the next scheduled task [19]. In some priority levels, no tasks are available. Since all layers are checked individually, the processing time is wasted during context switching. Researchers in [8] investigated the execution of the FreeRTOS context switch. The experiments showed a latency of 1.73µs ±0.27µs for changing the context of five tasks in FreeRTOS (version 7.4.0). Binary tree-based scheduling was performed in hardware. The experimental results showed no deviation, and the proposed solution had a latency of 0.92µs. The hardware-based task scheduling thus removed jitter effects and reduced the latency by 46.82%. Tang and Bergmann in [7] differentiate between active and passive task scheduling. The software kernel frequently checks whether a context switch is required and requests the offloaded module for the next ready task in a critical section. This approach is associated with passive task scheduling. If the outsourced module triggers an interruption, it is an active task scheduling. In this case, RTOS responds to the interrupt and performs context switching. As a result, the active hardware-based task scheduling resulted in a 23.7% longer execution time for running applications on a soft-core processor in [7]. Similarly, a co-processor can perform task scheduling and trigger an interruption when context switching is necessary [11], [20]. Thus, the call of the context switching could be reduced five-fold in [11]. However, this approach needs a shared memory system to exchange necessary real-time information between processors.

FreeRTOS is a widely used open-source RTOS with one of the smallest memory footprints and runs on more than thirty different processor architectures [19]. Therefore, it is a suitable candidate for offloading strategies and extensions. Researchers have modified its kernel functionalities to perform an inter-task communication in a NoC [21], offloaded task scheduling into hardware [8], or extended it for

Symmetric Multi-Processing (SMP) support [22]. FreeRTOS runs on each PE in the NoC-based multi-core system in this work. FreeRTOS has 32 priority levels and checks each level individually to see if a ready task is waiting for execution [19]. It may happen that not all priority levels contain tasks. As a result, the task scheduling wastes processing time with individual checkings and leads to scheduling overhead. FreeRTOS provides, in the meanwhile, a solution (`config_USE_PORT_OPTIMISED_TASK_SELECTION`) for the presented problem and skips priority levels if these do not contain tasks [19]. Thus, FreeRTOS improves the previously introduced scheduling overhead and promises a better scheduling performance. Researchers in [8] did not explore this effect while comparing the scheduling performance with the proposed solution as it was tested with an older FreeRTOS version. In this research work, the FreeRTOS task scheduling runs on hardware and co-processor. The work examines the impact of active (on co-processor), and passive (as hardware-based) offloaded task scheduling on FreeRTOS and the resulting task performance. Besides, it considers the optimized task scheduling behavior in experimental studies and compares results with offloading methodologies.

On the other hand, power-saving techniques impact task execution in RTOS. A (low-power) task scheduling can continuously monitor the slack time and scale operating frequency at run-time without exceeding the maximum execution time. This approach was presented in [4] based on the earliest deadline first policy and resulted in an overall energy reduction of 27%. Another work in [6] presented a hypervisor architecture suitable for low-power applications where the scheduler dealt with the power awareness of the system and partitioned tasks concerning real-time requirements. It alternated the operating frequency between 400MHz and 32MHz, leading to a power reduction of 33%. In this context, the voltage was dynamically scaled from 1.05V to 0.9V, resulting in a 28% power reduction. The presented works are based on Linux OS [4] or XtratuM [6]. There are also many works related to low-power task scheduling algorithms that optimize the power dissipation of a system at run-time. A recent survey provides an overview of energy-optimized scheduling algorithms. We refer the reader to [23] for more information. However, few works have been published yet with RTOSs and power-saving techniques. The power management needs processing time to retrieve and control voltage regulators' values. The processing time takes between $2-5$ms for XC7Z020-CLG484-1 to read a single value from onboard voltage regulators because it has an I2C-based communication protocol (PMBus) and typically operates at a frequency of 400kHz [24], [25], [26]. The more values are accessed, the longer the processing takes. Thus, the power management is partitioned in a hypervisor architecture [6] or executed in parallel on a co-processor [3]. However, and to the best of our knowledge, no architectures have been presented in the literature that consider the impact of power-saving techniques on real-time capability while supporting RTOS with offloading methodologies. Besides, power-saving strategies have been widely investigated but not combined with RTOSs. It was studied in simulations [21] or in-house operating systems with a scheduling algorithm [5]. Therefore, we have designed a NoC-based multi-core architecture on reconfigurable systems with FreeRTOS running PEs. Furthermore, we have explored optimizing strategies' impact on real-time behavior and power consumption. Each PE adjusts its frequency based on the required workload and goes to sleep in idle phases. Besides, a power reduction also takes place through voltage scaling. Although [6] investigated the impact of DVFS on a hypervisor architecture, the effect on real-time behavior was not. In addition, the behavior was only examined for processors and not reconfigurable hardware.

Voltage scaling results in loss of the configuration if it reaches a certain voltage level. Researchers have explored this case in the context of power gating and identified 400mV as the threshold for the PL supply voltage (V_{CCINT} [27]) of XC7Z020-CLG484-1 [28]. Dynamic Partial Reconfiguration (DPR) technique (PCAP described in [29]) can be used to restore the original condition, but it has a timing overhead. In [28], the PCAP approach was used for a 4MB configuration file and the configuration time was about 35ms. The approach in [28] reduced power by 96%. A recent survey gives an overview of power-saving strategies for reconfigurable systems. We refer the reader to [30] for more information. However, power gating is unsuitable for real-time applications, and the system should not scale voltage below the threshold. Therefore, this work explores the impact of power-saving techniques with DVFS and clock gating on real-time capability while supporting RTOS with offloading methodologies. Since the power gating phase has a high configuration time, it would affect the real-time capability, and thus it is not considered in this work.

Various challenges arise in respective applications while dynamically scaling voltage and frequency for power reduction. Streaming applications are generally about completing computation under latency and performance constraints. For instance, frequency scaling improved the throughput of a streaming application by 37.32% [31]. In the same way, energy-aware task parallelism can be used to reduce the execution time of an application [32]. A resource allocation graph maps each task of an application according to the available resources on the reconfigurable system. The goal is to reduce the overall execution time of the entire application. Thereby, voltage and frequency modulation can further improve the performance and power budget of the system. In [32], the LEON3 processor accommodated the application and power management tasks. The results showed a power reduction of 28%. It was possible to close the trade-off between an efficient, low-power design and application-specific requirements using power-saving techniques.

On the other hand, multi-core architectures are driven by power walls, performance scalability, and reliability challenges [21]. Thus, application workloads are divided among tasks to scale performance while meeting power constraints. Mapping strategies are used to perform task scheduling while respecting these constraints. Two task mapping heuristics (Low Energy Communication based on Dependencies Neighbor (LEC-DN) and Nearest Neighbor (NN)) were performed on a PE for a 2 × 2 [21] and 3 × 3 [33] cluster. The LEC-DN heuristic kept high communication volume tasks together and attempted to place them in nearby PEs within the cluster. The NN heuristic examined the proximity of available resources around a PE to perform a task. The goal was to identify a PE within the cluster with the most unassigned tasks. The approach led to avoiding hotspots and balanced utilization of processing resources in the architecture. Researchers in [34] proposed a mapping strategy based on global priority in a centralized manner with a master-slave scheme. The advantage is that tasks have more PEs available for execution. Furthermore, decentralized mapping strategies have additional power consumption since the local management monitors the task scheduling within a cluster. A centralized mapping approach monitors the overall architecture and reduces thus power consumption through single execution. Therefore, the multi-core architecture in this work consists of a NoC with four PEs in a master-slave configuration. The master PE executes the (centralized) mapping strategy, monitors the underlying technology's power consumption and validates if all slave PEs meet the real-time requirements for assigned tasks.

While dynamically mapping tasks to PEs, it is also necessary to explore how dependencies, inter-task communication, and resulting execution influence each other. Therefore, a hierarchical clustering algorithm was proposed in [35] to consider the dependencies for task mapping. An (in-house) RTOS selected all ready tasks based on the Earliest Deadline First (EDF) scheduling algorithm and grouped related tasks according to their dependencies. A central hardware-based scheduler received the scheduling/assignment information and managed all hard real-time tasks in the NoC-based many-core system. The hardware-based task scheduling has flexibility in adjustment for other scheduling algorithms and use-cases [36]. It is possible to divide a central hardware scheduler into uniform partial schedulers to increase performance in a NoC-based multi/many-core system [37]. For instance, cluster-based task scheduling can also group tasks according to their periods to achieve a long idle period. This approach was presented in [38] and reduced energy consumption by 33.5% while meeting all real-time requirements. In addition, there are works in the literature that consider the influence of power-saving techniques on real-time constraints and assign tasks accordingly on a NoC-based multi/many-core system [39], [40]. However, these works combined the mapping strategy with DVFS which increases the mapping complexity due to the huge design space exploration. Some works minimized the run-time overhead by executing task mapping and DVFS separately [41]. An adaptive mapping was presented in [42] that gathered run-time information for estimating application performance and remapped applications to cores while applying DVFS at run-time. The results showed a reduction of 28% in energy consumption. However, this approach did not consider the communication overhead through a NoC. A recent survey provides an overview and classification of mapping algorithms on multi-core systems. We refer the reader to [43] for more information. This work obtains the task assignment using the mapping strategy presented in [44]. However, the solution in [44] was suitable for independent tasks and thus did not consider the overhead introduced by incoming messages through NoC in the task scheduling. Tasks, in this work, can communicate with other tasks (associated with inter-task) or processors through NoC (associated with inter-processor), resulting in task dependency. Therefore, this work extends the mapping strategy in [44] for dependent tasks and considers the impact of power-saving techniques on running tasks in task scheduling at run-time. Furthermore, researchers in [21] proposed a multi-core platform that consisted of a global, local manager, and slave PEs composed of a 4 × 4 NoC platform with 2 × 2 clusters. The global manager coordinated the application repository and mapped the tasks directly to the slave PE if the local manager requested it through performed mapping algorithm (LEC-DN and NN). Each PE ran FreeRTOS, and a task was automatically created when assigned to the slave PE and deleted after its execution. Besides, the FreeRTOS kernel was extended for inter-task and inter-processor communication through the network. This research work follows the same principle without adapting the FreeRTOS kernel for inter-task/processor communication and applies power-saving techniques while supporting RTOS with offloading methodologies. Each slave PE individually adjusts its operating frequency and turns the processor into sleep mode while being in the idle phase. The master PE scales the voltage for the reconfigurable hardware and optimizes the power dissipation at run-time.

This work involves mapping and scheduling tasks in an NoC-based multi-core system while considering task dependency, inter-task communication, and power consumption. There are several challenges in finding proper solutions for this problem:

Firstly, identifying an optimal solution for task mapping becomes complex and challenging due to the number of tasks and resources available. Different algorithms and heuristics ([21], [35], [39]) have been proposed to address and solve this problem. We have applied the proposed heuristic mapping strategy in [44] and extended it for dependent tasks (inter-task and processor communication) to tackle the problem.

Secondly, the problem involves balancing real-time capability and energy efficiency. For instance, frequency scaling leads to a longer or shorter execution of tasks in RTOSs while also impacting power consumption. A slower execution of tasks results in lowering power consumption on PEs, but it may lead to deadline misses and affects the real-time capability. Therefore, dynamic workloads and power

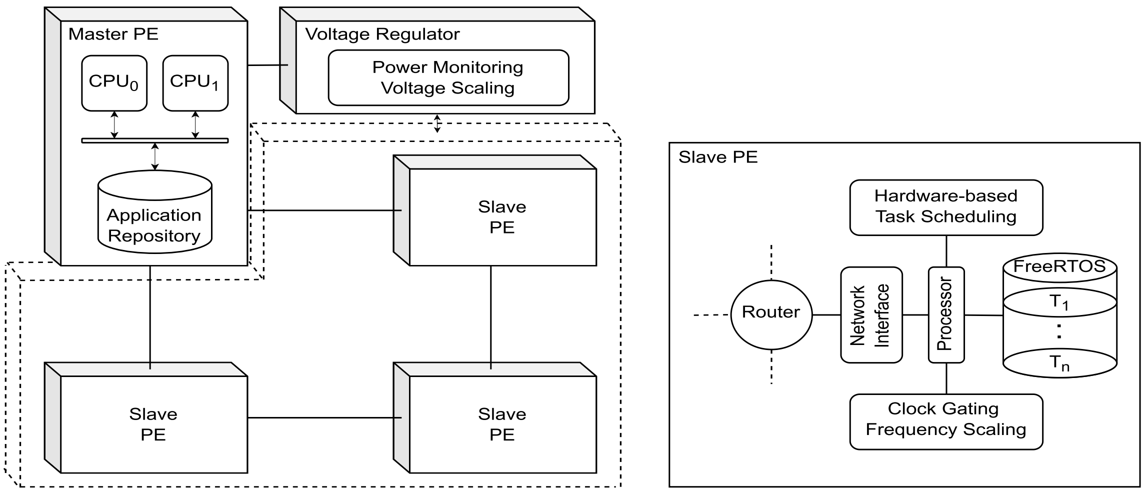


FIGURE 1. **Overview of the hardware architecture that contains a master PE and three slave PEs running FreeRTOS and connected through a NoC. We used an offloaded methodology to support each PE to improve context switching and thus result in real-time performance. Each slave PE performs clock gating and frequency scaling. The master PE performs voltage scaling, power monitoring, and mapping strategy. The dashed frame describes the reconfigurable hardware.**

consumption need to be monitored at run-time. Several works have proposed solutions for this problem, such as dynamically adapting task mapping based on processing time and power consumption [39], [40], [41], [42], [43]. In the same way, we have designed a heterogeneous multi-core architecture in a master-slave configuration. The master PE monitors power consumption, validates if all slave PEs meet the real-time requirements, and reassigns tasks to slave PEs if necessary. The slave PEs execute assigned tasks with RTOS while dynamically adapting frequency and applying clock gating in idle phases.

Thirdly, this problem involves task dependency through inter-task or task-processor communication. For instance, inter-processor communication introduces a communication overhead and impacts the application completion time. Several works [35], [36], [37], [38], [39], [40] have addressed this problem and proposed possible solutions. Our mapping strategy aims to assign dependent tasks primarily to the same slave PE to avoid communication overhead. Otherwise, the algorithm determines an appropriate slave PE based on a cost function.

Finally, the problem involves designing a proper offloaded scheduler for PEs, resulting in better RTOS scheduling performance. For instance, it is unpredictable when slave PEs receive messages in the NoC-based architecture. An RTOS reacts faster to incoming messages or events with offloaded task scheduling algorithms.

Several works ([7], [8], [9], [11], [19], [20]) proposed different solutions for this problem. We have proposed an adaptable solution for hardware-based task scheduling in this work. It can be used with any processor system without adapting RTOS kernel functionalities.

III. DESIGN METHODOLOGY

A multi-core system architecture is designed to explore the influence of power-saving techniques on real-time capability. Figure 1 gives an overview of the proposed architecture. The design consists of a NoC with four PEs in a 2D-mesh topology and master-slave configuration. Each PE executes FreeRTOS. The master PE monitors the underlying technology's power consumption and dynamically scales voltage to reduce its overall power. Besides, it hosts tasks in the application repository and performs task assignments for slave PEs. The mapping strategy considers task dependency, inter-task/processor communication, and power consumption to assign a task for execution on a slave PE. While executing tasks and RTOS services, the running FreeRTOS kernel is supported by an active offloaded task scheduling approach on the co-processor (depicted as CPU$_1$ in Figure 1). On the other hand, each slave PE has a homogeneous design on the reconfigurable hardware (depicted as a dashed frame in Figure 1). It consists of a MicroBlaze processor, a MMCM module for frequency scaling, clock gating for turning the processor into sleep mode, and offloaded task scheduling. Furthermore, a passive task scheduling approach supports all FreeRTOS running slave PEs at run-time. After the master PE's task assignment, each slave PE dynamically creates a task, executes assigned tasks, and performs an inter-task/processor communication to transmit task results.

A. OFFLOADING REAL-TIME SYSTEM COMPONENTS

One of the most widely used RTOSs on reconfigurable platforms is FreeRTOS. FreeRTOS is a portable and open-source operating system suitable for embedded applications. It meets the (hard) real-time requirement of an application at run-time. FreeRTOS supports semaphores and mutex for resource sharing and synchronization. A real-time scheduler executes functions (as tasks) in a specific order with their priorities. The scheduler consists of three types of scheduling algorithms. These algorithms are the fixed-priority preemptive scheduling (with and without time slicing) and the cooperative scheduling algorithm. Tasks with the same priority are executed according to the round-robin policy [19]. To understand the task scheduling functionality in software, we briefly present the execution flow as follows. The kernel of FreeRTOS measures the passing time using a tick-count variable. A timer interrupt increments this variable with a temporal resolution based on the chosen frequency. Each time the tick-count variable is incremented, the periodic scheduler checks whether a context switching is required. If an interrupt occurs during a task's execution, the task's context has to be saved in the stack. The scheduler checks the priorities of the tasks in the ready list. If there is a task with a higher priority in the ready list, this task next is granted a processing time when the interrupt routine completes. The operating system retrieves the next task's context from the stack and continues its execution. Otherwise, the context of the previous task is restored from the stack, and the execution is resumed.

The periodic call of the task scheduling service introduces additional overhead in software and eventually leads to jitter. Therefore, it is beneficial to investigate the potential of offloading the FreeRTOS task scheduling in more detail. In this work, the task scheduling runs first on a co-processor and second in hardware. In both cases, the task scheduling selects the next task from the ready list. FreeRTOS still pushes the context of the running task onto the stack and restores the context of the following ready task from the stack. In the case of the co-processor, an interrupt request is sent to FreeRTOS when context switching is required. Therefore, it does not frequently check anymore for context switching. The context of the next task to be executed was already selected by the co-processor and is available in the shared memory. In the case of the hardware-based task scheduling, FreeRTOS checks in regular intervals for context switching. If a context switching is required, FreeRTOS reads the next task to be executed from the offloaded reconfigurable module. As a result, the offloading approach removes the operating system's processing time for scheduling decisions and task selections.

B. POWER-AWARE REAL-TIME SYSTEM ARCHITECTURE

FreeRTOS has an idle task with the lowest priority. When other tasks in FreeRTOS are deleted, the idle task frees the allocated memory for the deleted task [19]. Otherwise, this task has no other function in FreeRTOS to perform. Therefore, the processor should enter sleep mode after freeing the memory space. Such an enhancement should also be made for running tasks with power-saving techniques. One such popular power capping technique is DVFS. It dynamically scales voltage and frequency to a certain operating point to minimize power dissipation at run-time. For instance, the power consumption can be optimized with onboard voltage regulators and MMCM modules on XC7Z020-CLG484-1 [3]. The onboard voltage regulators are used for scaling voltage and sensing the power consumption. The MMCM module is used to scale processor's frequency and other hardware components. However, frequency scaling leads to a longer or shorter execution of tasks in RTOSs. For instance, faster execution of tasks results in a long idle period where processors are busy waiting for the next scheduled task. The processor should be put into sleep mode with clock gating as a solution. A slower operation leads to deadline misses and affects the real-time capability of running applications.

Meeting this requirement becomes even more challenging when applications run on multi-core systems. A single processor design has certain limitations, such as sequential execution of an application that multi-core systems improve. The application can consist of different functions and be executed simultaneously on several processors. This shortens the time to complete the application and allows the processor to enter sleep mode. The overall power consumption is also taken into account in this work. Therefore, the single-core system only has a limited possibility of using power-saving techniques. In this case, the processor must interrupt execution to check the current power loss and adjust the power consumption with saving techniques. However, this leads to performance degradation in applications. This criterion can also be better controlled and managed with multi-core systems. A dependency arises when an application is divided into individual segments and distributed among processors. The execution of these segments thus requires a specific execution order. Before distributing segments among processors, task parallelism and interconnections have to be examined.

Therefore, a mapping strategy for multi-core systems is needed that distributes tasks to processor cores according to their dependencies, execution time, and communication. A heuristic mapping strategy is designed with the mentioned criteria in this work. Moreover, FreeRTOS runs on each processor using offloaded task scheduling. Each processor can locally optimize its performance and thus the power consumption through clock gating and dynamic frequency scaling. However, a power reduction also takes place through voltage scaling. Thus, the goal is to manage the execution of tasks on processing elements while dynamically adjusting the frequency/voltage on the XC7Z020-CLG484-1 and meeting real-time requirements with FreeRTOS.

IV. IMPLEMENTATION

A. HARDWARE-BASED TASK SCHEDULING

Since the essential part of the kernel is implemented as assembly code, context switching is executed within a few microseconds. However, deviations still occur during task selection at run-time. This can lead to a non-deterministic behavior and jitter because of the number of tasks, required resources, or system status. In the case of hard real-time requirements, jitter effects are unacceptable. Furthermore, the run-time behavior should remain deterministic. The goal should be to remove occurring jitter effects in RTOS. In such cases, offloaded task scheduling should be preferred. As proven in [8], a hardware-based task scheduling improved the real-time capability of RTOSs. The latency for context switching was reduced at least by 46.82%.

Algorithm 1 Hardware-Based Task Scheduling

```
1:  /* Update task list */
2:  #pragma HLS inline
3:  for All tasks in the task list do
4:      #pragma HLS pipeline
5:      if Task exists in the task list and has write state then
6:          Update task items in the task list
7:      else
8:          Add task into the task list
9:      end if
10: end for
    /* Search for the next ready task */
11: #pragma HLS inline
12: for All tasks in the task list do
13:     #pragma HLS pipeline II = 2
14:     if Task has ready state then
15:         Search for the highest priority task
16:     end if
17: end for
18: for All tasks in the task list do
19:     #pragma HLS pipeline
20:     if Task has ready state and highest priority then
21:         Search for the smallest count value (round-robin)
            and select this as the next scheduled task
22:     end if
23: end for
24: for All tasks in the task list do
25:     #pragma HLS pipeline
26:     if Task has ready state and highest priority then
27:         Update count value for all tasks with the same
            highest priority
28:     end if
29: end for
30: return Current task to FreeRTOS
```

Therefore, we have offloaded the task scheduling of FreeRTOS into hardware. We put a focus on the extensibility and adaptability of our implementation. Thus, we have implemented our approach with a high-level synthesis tool. Algorithm 1 shows the structure of the implementation. In the beginning, the hardware-based task scheduling returns the information of the recent ready task to FreeRTOS. If FreeRTOS sends a write command through the AXI interface, the task scheduling updates the task list with the new task entry (lines 2-10). However, it may happen that the task already exists in the list, and only its states need to be updated. Then, the task scheduling overwrites the respective task entries. If the task does not exist, the task scheduling adds it to the list. Once all tasks are updated, the task scheduling searches for the ready task in the list (lines 11-29). First, it identifies the highest priority in the ready list. Then, the task scheduling selects tasks with the round-robin policy. If multiple task items with the same priority are waiting for processing time, the task scheduling enables an execution one after another. The call of a task is monitored with a counter for this purpose. The task with the lowest counter reading and thus fewer calls in the same priority level is executed next. Once the task item with the highest priority is found in the ready list, the information is sent to FreeRTOS. All loops are pipelined to allow concurrent execution of operations [45]. Inlining functions lead to effectively sharing and optimizing operations within the functions [45]. Therefore, we have optimized our implementation with these pragmas in the Xilinx Vivado HLS tool. The AXI interface is used for sending/receiving task information. In this work, we have compared the performance of the task scheduling with the AXI4-Stream (AXI-S) and AXI4-Lite protocols (presented in Section V-B). Both protocols use a 32bit interface data width. In the case of AXI-S, the first bit shows if FreeRTOS reads the task status or writes a new task item into the ready list. The next bit reflects whether the task state is ready or not (blocked or suspended). The following six bits contain information about the priority of tasks. The information about the task pointer is in the remaining 24 bits that correspond to the internal task pointer `pxCurrentTCB` in FreeRTOS. In the case of AXI4-Lite, each task item (write, ready, priority, and task pointer) has its own 32bit width interface. A goal is to keep the implementation reproducible. Therefore, we have only added a few lines of code in `task.c` and `portmacro.h`. The AXI interface is only called in these files. Besides, the macro `config_USE_PORT_OPTIMISED_TASK_SELECTION` in `FreeRTOS_config.h` is extended to turn on the hardware-based task scheduling.

B. CO-PROCESSOR BASED TASK SCHEDULING

An alternative approach to offload task scheduling is the porting on the co-processor. The main difference to hardware-based task scheduling is the interprocessor communication and synchronization. When context switching is required, the co-processor sends an interrupt request to RTOS. The context for the next task to be executed is then available in the shared memory. RTOS fetches the task information from the shared memory and assigns it in the kernel to the current task control block (`pxCurrentTCB`). As a result, this approach removes the processing time which is usually required for scheduling decisions and task selections in RTOS. To investigate the timing behavior of task scheduling more deeply, we have conducted our implementation on an ARM Cortex-A9 dual-core processor of XC7Z020-CLG484-1. These processors can operate with a maximum frequency of 667MHz. In the case of the Programmable Logic (PL), the maximum operating frequency is 250MHz. API functions can change task states of FreeRTOS at run-time. Both processors also need to access and use data structures. Therefore, we have allocated a shared memory region to enable a shared use in the linker script. The memory region is divided into three partitions so that each processor has its memory area. The third partition is the shared memory region. Besides, synchronization mechanisms have been deployed to access data structures simultaneously. The instruction set of the ARMv7-A architecture provides certain operations to establish such a synchronization mechanism [46]. This mechanism prevents mutual exclusion within critical sections. The LDREX and STREX commands allow exclusive access to memory. LDREX loads the content of a memory address into a register and initializes the state of the exclusive monitor. It then tracks the synchronization operation. STREX performs a conditional store operation on a memory address. If the operation accesses the memory address, STREX updates the memory location and the corresponding register with a status bit.

```
1 volatile int* mutex; // lock variable
2 mutex = 0x1D158000 ;
3 *mutex = 0;
4 Xil_SetTlbAttributes(0x1D158000,0x15DE6);
5 lock_mutex (mutex);
6 /* critical section */
7 unlock_mutex (mutex);
```

LISTING 1. Use of the synchronization mechanism in the source code.

Listing 1 shows how the synchronization mechanism is deployed in source codes. The function (in line 4) allocates memory for the lock variable from the shared memory. The synchronization mechanism enables controlled access to the data structures and variables within the critical section (lines 5-7). The approach ensures consistency and prevents race conditions between processors. All shared variables and data structures of FreeRTOS are listed in Table 2. The interrupt routines are a basic mechanism of the RTOS kernel. The ARMv7-A architecture has a Generic Interrupt Controller (GIC) [46]. It manages interrupts that are triggered by the software and peripherals. The scheduler on the co-processor periodically calls an interrupt handler at a certain frequency. It checks whether a context switching is required for the running task in FreeRTOS. In context switching, the task scheduling selects a new ready task and sends an interrupt request with GIC to FreeRTOS. The task scheduling accesses data structures to select a ready task. Therefore, synchronization mechanisms are integrated into the task scheduling on the co-processor. Apart from using synchronization mechanisms, shared memory regions, and calling of interrupt routines, the task scheduling is offloaded into the co-processor. Besides, no further changes have been made to FreeRTOS.

C. MULTI-CORE POWER-AWARE SYSTEM ARCHITECTURE

A multi-core system architecture is implemented on XC7Z020-CLG484-1. The design consists of a NoC with four PEs in a 2D-mesh topology and master-slave configuration (shown in Figure 2a). The master PE consists of an ARM Cortex-A9 processor, a MMCM module for adjusting the frequency of routers, and Direct Memory Access (DMA) for enabling communication between the processor and router (shown in Figure 2b). Each slave PE has a homogeneous design on the hardware. It consists of a MicroBlaze processor, MMCM module for frequency scaling, clock gating for turning the processor into sleep mode and offloaded task scheduling (shown in Figure 2c).

TABLE 2. All resources in the shared memory used by FreeRTOS and the co-processor.

Shared Resources	Memory Address
.shared_pxReadyTasksLists	0x1D100000
.shared_pxDelayedTaskList	0x1D108000
.shared_pxOverflowDelayedTaskList	0x1D110000
.shared_ulSchedulerYieldRequired	0x1D118000
.shared_xTickCount	0x1D120000
.shared_uxSchedulerSuspended	0x1D128000
.shared_uxPendedTicks	0x1D130000
.shared_pxCurrentTCB	0x1D138000
.shared_xNextTaskUnblockTime	0x1D140000
.shared_xNumOfOverflows	0x1D148000
.shared_uxTopReadyPriority	0x1D150000
.shared_xSchedulerRunning	0x1D178000
.shared_xYieldPending	0x1D180000
.shared_pxNewTCB	0x1D188000

Because of the limited BRAM resources, the reconfigurable hardware consists of three slave PEs connected through a NoC. Each slave PE executes FreeRTOS. When a task is assigned through the mapping strategy to a slave PE, FreeRTOS automatically creates a task at run-time. FreeRTOS moves a task's stack to its heap on a context switch and thus allocates dynamic memory. Otherwise, the slave PE does not require heap memory. This work examined the FreeRTOS heap usage for the design. We have assumed that a FreeRTOS would schedule a maximum of 14 tasks at run-time. It resulted in the required memory size of 64KiB for FreeRTOS. A slave PE has a local repository to store assigned tasks from the master PE's mapping assignment. This work reserves a memory size of 32KiB for tasks in the local repository. The reconfigurable hardware of XC7Z020-CLG484-1 has 140 BRAM cells. The design would need 72 BRAM cells for a 2×2 and 192 BRAM cells for a 3×3 NoC architecture. Since a 3×3 NoC architecture exceeds available hardware resources, the hardware architecture consists of 4 routers arranged in a 2×2 mesh topology. In addition, it is beneficial due to the scalability to deploy a NoC in reconfigurable computing. Bus-based systems cannot be extended, whereas a NoC can be scaled by configuring or adding more routers to the existing architecture [47]. That is the reason why a NoC is used in the hardware design.

Applications are called periodically in our example task set. Therefore, a Timer IP monitors the execution time of running applications from master PE as shown in Figure 2b. When the defined period passes, the Timer IP calls an interrupt signal to indicate the end of a period to the processor. Otherwise, slave PEs use the Timer IP for the FreeRTOS tick as shown in Figure 2c.

All slave PEs have an individual clock domain and dynamically change the frequency with the MMCM module. The PL fabric clock is set to a maximum value of 100MHz. Experimental studies have shown that the MicroBlaze processor in this design cannot operate below

10MHz. The MMCM module thus adapts the operating frequency within these ranges. The MicroBlaze processor cannot directly receive/send a message from/to the router because of individual PE clock domains. Therefore, FIFOs are connected between the network interface and router to decouple different clock domains. As soon as a data packet is received, a FIFO triggers an interrupt signal to notify the processor of new incoming data packets. In the meantime, the processor prepares data packets for dispatch or processes other application sections.

The MicroBlaze processor has an AXI-S interface and is directly connected to FIFOs. The Processing System (PS) does not support an AXI-S interface. A DMA IP can transfer AXI Memory-Mapped (AXI-MM) messages to AXI-S based messages. Therefore, DMA is used to interface FIFOs to the ARM processor. In addition, the DMA component can directly access the DDR memory in PS. Thus, it is connected to the DDR controller with a separate high-bandwidth AXI4 High-Performance Port. The DDR controller is responsible for the arbitration between the DDR memory access requests of the ARM processor and DMA [48]. Hence, data packets are stored (read) simultaneously into (from) the DDR memory through DMA while executing the application on the processor. DMA has two interrupt signals for incoming (AXI-MM to AXI-S) or outgoing (AXI-S to AXI-MM) transmissions. It signals the ARM processor that packets are received or sent. The use of DMA overall reduces the compute and communication-intensive initial mapping phase on the ARM processor.

The used routers follow an XY-routing scheme. Besides, it utilizes round-robin arbitration and wormhole flow control. Therefore, FIFOs hold a single flit. Phit and flit sizes have been configured with 32bit width. The FIFO depth depends on the packet size and is accordingly defined. The NoC uses the XY-routing algorithm connected in a mesh topology and thus is deadlock-free. All routers have an AXI-S port and are directly connected to FIFOs. A packet contains a header and tail flit. The TLAST signal from the AXI-S interface represents a tail flit and signals to the router that a packet is fully transmitted or received. The header flit signals the destination in XY coordinates. For a detailed description of the router used, we refer the reader to [47] at this point.

FIFOs have to notify PEs that they are ready to transmit packets on the NoC. Therefore, a threshold is defined for the output FIFO to indicate its empty state. The threshold shows that the receiving PE has either read or is reading packets from its input FIFO. This is why FIFO depth corresponds to the packet size and operates in the store-and-forward policy to monitor the situational progress of packets within the NoC. Thus, packets are generated and fully present in the output FIFO before injecting them into the NoC. The wormhole control flow allows routers to accept only a single flit per input. When a packet has passed through the network, the number of flits in the sending PE's FIFO is reduced by one. A packet is guaranteed to reach its destination when it has passed a fixed number of routers. That can be determined by $Router_{\max} = Dim_x + Dim_y - 1$, where Dim_x and Dim_y are dimensions of the NoC. If the number of flits falls below this size, the package has arrived at the destination. Therefore, a 2×2 NoC requires FIFOs of a threshold size of 29. However, due to the limitations of the Xilinx FIFO IP, the threshold has been set to 27. When output FIFOs capture the fallen threshold, it triggers an interrupt signal to the MicroBlaze processor to indicate leaving packets.

All applications are executed on a 32bit MicroBlaze processor. The real-time configuration is selected, and the implementation is optimized for performance. Besides, the processor has no caches and two local memories for data and instructions with 64KiB and 32KiB address ranges. The first memory area is reserved for FreeRTOS and the second for the application. The floating-point unit is enabled (with extension and exception settings). The MicroBlaze processor has three AXI-S ports. The ports are connected to the input, output FIFO and outsourced task scheduling of FreeRTOS.

If the processor clock signal is stopped without prior indication, unpredictable effects may occur. An abrupt halt can result in data corruption during memory access or transition to an undefined processor state. Therefore, discrete ports (*sleep, hibernate, suspend*) are enabled on the configuration to manage processor sleep and wake-up. That allows the MicroBlaze processor to complete external accesses, such as memory transactions, and safely halt its pipeline, providing a safe state for clock gating. Xilinx provides in [49] a schematic for clock gating used in this work. For a detailed description of the clock gating, we refer the reader to [49]. The MicroBlaze processor can suspend itself with sleep instruction (*mbar16*) and initialize clock gating by activating its sleep signal [50]. However, an interrupt signal or a *dbg_wakeup* request can disable the clock gate and wake the processor. The debug request signal has not been used in this work. In addition, Timer IP or incoming data packets can trigger an interrupt signal. The *mbar* instructions have been executed within $2 + N$ or $8 + N$ cycles where N is the number of clock cycles to complete memory access [50]. The proposed schematic in [49] has only required a single clock cycle. Therefore, entering and leaving a clock gated state is negligible. Besides, the MicroBlaze processor and BRAM memory are only clock gated as shown in Figure 2c. The remaining PE components generate the interrupt signal or are responsible for system monitoring. Thus, these components have not been clock gated.

D. VOLTAGE SCALING AND POWER SENSING ON XC7Z020-CLG484-1

The onboard voltage regulators are used for scaling voltage and sensing the power consumption. The platform consists of a dual-core ARM Cortex-A9 processor and an FPGA [24]. It is equipped with three digital power controllers (UCD9248) by Texas Instruments (TI) [51]. These controllers provide ten power rails to supply PS, PL and other parts of the evaluation board. The power controllers are wired with the Power Management Bus (PMBus) which is connected to a

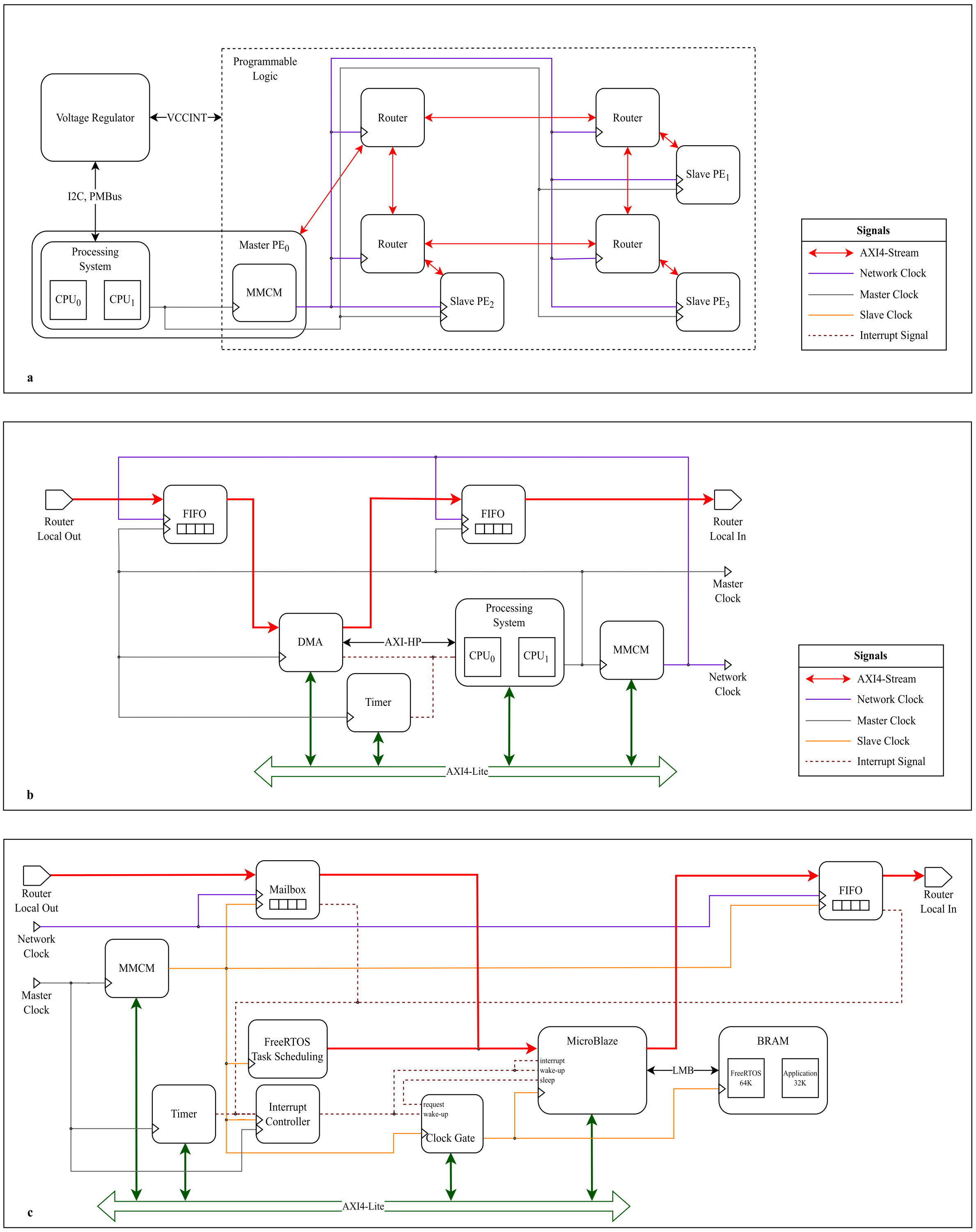


FIGURE 2. [a] Overview of the hardware architecture on XC7Z020-CLG484-1. The structure of the master PE and the connection to individual slave PEs are shown. The master PE consists of the PS that controls voltage regulators. The router has its clock frequency, which the PS controls through the MMCM module. [b] Overview of the hardware architecture of the master PE. MMCM adjusts the frequency of routers, and DMA enables the communication between the processor and router. [c] Overview of the hardware architecture of slave PEs. The MicroBlaze processor runs FreeRTOS. The scheduling decisions are made in hardware. The processor can be put into sleep mode when idle with clock gating. In all diagrams, all signals and the corresponding clock domains are color-coded.

1-to-8 channel I2C bus switch (PCA9548) [24]. Using the PMBus commands [51], it is possible to read the voltage and current on each power rail. In this research work, we investigate the characteristics of the PL internal supply

Algorithm 2 Voltage Scaling on XC7Z020-CLG484-1

```
1:  /* Scaling of the voltage ($V_{CCINT}$) to the set point */
2:  if set point $\leq$ 1.00V and set point $\geq$ 0.55V then
3:      if set point $\leq$ nominal voltage then
            /* Modify the following PMBus commands */
4:          POWER_GOOD_OFF = 85% of the set point
5:          POWER_GOOD_ON = 90% of the set point
6:          VOUT_UV_FAULT_LIMIT = 85% of the set point

7:          DVS with VOUT_COMMAND to the set point
8:          VOUT_MARGIN_HIGH = 105% of the set point
9:          VOUT_MAX = 110% of the set point
10:         VOUT_OV_FAULT_LIMIT = 115% of the set
            point
11:     end if
12:     if set point > nominal voltage then
            /* Modify the following PMBus commands */
13:         VOUT_MARGIN_HIGH = 105% of the set point
14:         VOUT_MAX = 110% of the set point
15:         VOUT_OV_FAULT_LIMIT = 115% of the set
            point
16:         DVS with VOUT_COMMAND to the set point
17:         POWER_GOOD_OFF = 85% of the set point
18:         POWER_GOOD_ON = 90% of the set point
19:         VOUT_UV_FAULT_LIMIT = 85% of the set point

20:         VOUT_MARGIN_LOW = 95% of the set point
21:     end if
22: end if
```

voltage (V_{CCINT}). The nominal voltage of PL is 1V [27]. Xilinx already provides a driver for XC7Z020-CLG484-1 to retrieve the voltage and current data [26]. Once the system setup is correctly initialized and the sensed data are read, the driver can adjust the voltage parameters at run-time. Algorithm 2 describes the voltage scaling with the PMBus commands on the platform. The steps have to be followed as the board has off-the-shelf protection mechanisms [25]. The mechanism has protection limits to prevent an overcurrent, overvoltage, and undervoltage fault occur on the technology. Without a proper setup of the PMBus commands, the fault management is active and turns off the board immediately. Thus, the appropriate commands have to be adapted to the redefined set points while scaling the voltage as shown in Algorithm 2 (lines 3-11 or 12-21). Based on previous research works, we have limited the voltage scaling between 1V and 550mV to prevent a faulty operation (line 2).

E. DESIGN OF THE HEURISTIC MAPPING ALGORITHM AND SYSTEM MONITORING

The master PE monitors the system state, analyzes task dependencies, creates a sorted task list, and determines where to run corresponding tasks. In this case, the task dependency is represented in a Directed Acyclic Graph (DAG). Here, a node represents the execution time of a task. The edges describe the time for incoming and outgoing messages from a task. These factors determine the weight quantity for each node and its edges. The master PE assigns priority to each node based on its weight quantity. The priority is obtained from the subordinate weight values in the respective task path. The highest accumulated weight value of a node corresponds to the highest priority. A task list sorted by these priorities represents a topological ordering of applications in DAG since a predecessor task is always weighted and prioritized higher than the subordinate weights of its successors.

Afterward, the master PE checks whether a slave PE can provide the required memory size for respective tasks and their parameters. Then, each initial node is assigned to a slave PE to take advantage of task-level parallelism. A task-to-task communication can occur at subsequent nodes. Therefore, a cost function is calculated for each node and slave PE. The task is then assigned to the respective slave PE with the lowest cost function. The cost function considers the following two cases. First, it checks whether there is already a predecessor node on the respective slave PE. If this is the case, inter-processor communication is not required. This leads to a cost reduction (otherwise to a cost penalty). Another case may occur as follows. One of the predecessor nodes may be executed on another slave PE. In addition, the execution may still be running and not completed. Therefore, the respective task must wait until the predecessor node completes its execution and receives the data through the network. The predecessor task with the latest execution should be identified, as the other predecessor tasks should already be completed. If this case exists, a penalty is included in the cost function. Each slave PE has a task list with some entries. These entries contain the total memory usage, the end time of the overall execution, and the number of assigned tasks. The task list is updated regularly and reflects the current status of the task assignment. The cost function identifies a suitable slave PE for a task. However, the identified slave PE may not have enough memory. Therefore, the task list is used to select a suitable slave PE with sufficient memory.

Algorithm 3 describes the steps of task mapping. The algorithm checks if a task has a predecessor (denoted as a parent in line 5). If this is not the case, the task is mapped to a slave PE with the lowest workload (denoted as utilization in line 6). This results in an even distribution of initial tasks to exploit task-level parallelism. If the task has a predecessor, the mapping strategy identifies the task with the lowest cost function (line 12). A slave PE can wait for an incoming message from a predecessor task. Therefore, the timing list is frequently updated (line 17). It also considers potential time savings due to task mapping. A task is finally assigned to a slave PE (lines 10, 16). The estimated completion time is updated after each mapping (line 19).

The master PE has a repository to host applications and their data. In this work, an executable code compiled for the MicroBlaze ISA is prepared and stored in the DDR memory. The master PE transfers the source code and its data through the NoC to slave PEs. Subsequently, the slave PE

Algorithm 3 Heuristic Mapping Algorithm

```
1:  /* Task assignment to slave PEs */
2:  Assignment (Prioritized task list $T$, List of slave PEs $P$)
3:  for Task $t_j \in T$ do
4:    if parent $(t_j) = \emptyset$ then
5:      $pe\_index$ ← Find slave PE with the lowest
        utilization and enough memory $(t_j)$
6:      if No slave PE is found then
7:        Return an error
8:      end if
9:      Add $t_j$ to assigned task list $L\ (p_{pe\_index})$
10:   else
11:     $pe\_index$ ← Find slave PE with the lowest cost and
        enough memory $(t_j)$
12:     if No slave PE is found then
13:       Return an error
14:     end if
15:     Add $t_j$ to assigned task list $L\ (p_{pe\_index})$
16:     Update the task timing data of $L\ (p_{pe\_index})$
17:   end if
18:   Update the end point of $p_{pe\_index}$
19: end for
20: return An assigned task list $L\ (p_i)$ to each slave PE
    $p_i \in P$ with $L\ (p_i) \subset T$
```

stores receiving packages into the local BRAM memory to execute them as part of a task. As a result, any application can be deployed on a slave PE after a task assignment. Moreover, the same application may run with different data in different tasks. That would introduce redundancy and cause memory overhead. These cases can also be better controlled, as applications and their data are sent separately, which improves memory usage on slave PEs. In this work, all applications were prepared as standalone applications to integrate these flexibly as tasks into FreeRTOS. We have reserved a 32KiB frame for each application in the DDR and BRAM memory.

The starting address of an application function is placed at the beginning of a frame. Therefore, a function pointer within a FreeRTOS task points to this address to execute an application function. When a slave PE receives the first package, it calls the *receive task* in FreeRTOS, which stores all received application packages into the local BRAM memory. After receiving and storing all packages, the *receive task* creates an *application task*. All application tasks have the same execution pattern. With this model, a single generic function can execute any application task associated with a slave PE without knowing the specific behavior of the task at design time. This is possible because multiple tasks can share functions of a FreeRTOS-based program without interfering with each other, provided there is no concurrent write access to global variables. The function immediately suspends or blocks the task state because it waits for the starting signal from the master PE. The master PE starts all slave PEs simultaneously to monitor the system status and perform run-time management. After receiving the start control message, the *application task* thus returns to the ready state and can be executed within the application hyperperiod specified by the master PE. The *application task* remains in the blocked state when a task waits for messages or execution completion from predecessors. It remains blocked until the *receive task* or a predecessor task mapped on the same slave PE sends a notification event. Afterward, the *application task* calls the assigned task with the function pointer and executes it according to the call definition in master PE. The function also returns a result after execution, which is used for inter-task communication. If the successor task is located on the same slave PE, the result is copied into the associated task memory region. Otherwise, the resulting message data is sent to the corresponding slave PE through the network.

All generated tasks attempt to complete their executions within a hyperperiod. However, FreeRTOS has preemptive task scheduling and executes tasks in a specific order with their priorities. For instance, tasks with the same priority are executed according to the round-robin policy. The master PE has created a sorted task list that contains the execution order of assigned applications for a slave PE. The slave PE can therefore set priorities according to the tasks it receives. However, this approach scales poorly with an increasing number of allowed maximum tasks in a slave since each different priority group increases the time required to perform a context switch. Therefore, slave PEs have a scalable mechanism that dynamically assigns a high priority to each application task according to its assignment. This process resembles the passing of a token from one task to the next after its completion. Hence, all slave PEs use the following six priority levels in FreeRTOS (from high to low):

- Level 5 - reserved for FreeRTOS service routines
- Level 4 - reserved for the *receive task*
- Level 3 - assigned to completed *application tasks*
- Level 2 - assigned to the *application task* to be performed next
- Level 1 - default priority for all *application tasks*
- Level 0 - reserved for the *idle task*

Level 0 is reserved for the idle task. Clock gating is performed within this task since no other task is processed. The *receive task* must preempt all other application-specific execution because it receives messages from PEs and manages the underlying system. More precisely, this task manages incoming messages, creates *application tasks*, adjusts the processor clock desired by the master PE, and releases the execution of tasks on request. Because of its crucial role, this task has the highest priority (Level 4). Otherwise, all created tasks are assigned to Level 1 except for the first generated task in Level 2. Thus, the FreeRTOS task scheduling will start executing the assigned task in Level 2. Besides, FreeRTOS allows changing task priorities dynamically. When an *application task* from Level 1 (2) attempts to raise (lower) its priority, i.e., pass the priority token, FreeRTOS will enforce a context switch before the task runs into the blocked state. Once a task completes its execution, it is set to Level 3 to prevent it from being replaced by another *application task*. Here, the next task to be executed is determined, and its priority is set to Level 2. Finally, the task status in Level 3 is blocked. If the last task reaches Level 3, a notification about completing all tasks is sent to the master PE. All tasks revert to Levels 1 and 2 at the beginning of each new hyperperiod. Note that hardware-based task scheduling is connected with each slave PE to improve scheduling overhead. Since the presented implementation is extensible and adaptable, only the described adjustments in Section IV-A are made to all slave PEs.

The master PE prepares all slave PEs to execute all tasks periodically before a defined deadline in real-time. Initially, a hyperperiod is defined in which all slave PEs have to complete all assigned tasks. The master PE determines a deadline for each slave PE. First, it identifies the slave PE with the latest expected execution time and checks whether this PE can execute all its tasks within the specified period. This period is defined as a deadline for this slave PE, even if the expected execution time ends earlier than the specified period. Otherwise, the specified period has to be adjusted to meet the deadline. As described before, there may be dependencies between individual tasks so inter-processor communication may be necessary. Therefore, all slave PEs should complete their execution before the identified latest execution time and defined deadline. The execution time is divided by the identified latest execution time, and the result is multiplied by the defined deadline, which is the resulting deadline for the slave PE. The underlying system architecture allows each slave PE to adjust its operating frequency using MMCM. Note that the NoC operating frequency corresponds to the maximum frequency set in the slave PE so that no bandwidth and message loss occur at run-time. Thus, all slave PEs perform frequency scaling based on the determined deadline. The master PE measures the total execution time of each slave PE and verifies that slave PEs meet the deadline as closely as possible without violating the schedule. In addition, it calculates an adjustable frequency (between $f_{\min}$ and $f_{\max}$) for each slave PE concerning the deadline and notifies them about it at the end of each hyperperiod. The slave PEs then set the proposed frequency locally.

FreeRTOS has an idle task at the lowest priority. If the task scheduling has no other ready tasks, then it calls the idle task that frees the allocated memory for deleted tasks and has no other function in FreeRTOS to perform. However, it is possible to call an idle hook function optionally within this task [19]. The idle hook function can be customized and allows the processor to be put into sleep mode. In this work, the function calls the sleep instruction of the MicroBlaze ISA. It triggers the clock gate module in PL (shown in Figure 2c), which puts the processor and memory into a sleep state. However, the processor is woken up in two cases. First, the FreeRTOS kernel periodically checks whether a context switching is required for a higher priority task. In addition, the processor may receive a package that triggers an interrupt and places the processor in the operating state. Otherwise, it remains in the idle task, and thus the processor is in the sleep state.

In the case of the voltage scaling, the master PE can only adjust the voltage of the entire PL region. Therefore, a minimum voltage level is identified at run-time in which all PEs can still be stably executed. The expected results have been distorted if the voltage value is below a particular voltage value. As a result, the PL region loses its configuration, and the MicroBlaze processors can no longer run an application. However, the configuration can be restored with DPR in Xilinx when the voltage reaches a stable value. The configuration has a timing overhead that is much higher than that of the voltage scaling. In [28], the PCAP approach was used for a 4MB configuration file and the configuration time was about 35ms. Due to the expected high configuration time, we see a threat in the real-time behavior and remove the consideration of power gating from this work. At the same time, scaling the voltage from the nominal mode to 0.4V took 2.73ms [28]. For this reason, voltage scaling is performed during the initialization phase. The slave PEs execute all applications at this set voltage level during the execution. The necessary steps for voltage scaling have already been described in Section IV-D.

V. EVALUATION

This section presents the experimental results obtained for the FreeRTOS task scheduling offloaded to the co-processor and hardware performed on XC7Z020-CLG484-1. The underlying technology allows the voltage to be dynamically scaled and the power dissipation to be monitored at run-time. However, power management required almost 5ms to retrieve only one power value from voltage regulators. Deadline violations occur during system monitoring. Therefore, the evaluations have been split. The second ARM Cortex-A9 processor executed first the (active) offloaded task scheduling and second monitored (only for power measurements) the power dissipation at run-time. The master PE monitors the system state and manages the task assignment (Section IV-E). Therefore, the master was the first ARM Cortex-A9 processor, and all MicroBlaze processors were slaves. Moreover, the master PE has a repository to host applications and their data. In this case, all applications for the repository were obtained from the BEEBS benchmark [52]. An initial measurement was performed with it for the task assignment. A MicroBlaze processor ran them within standalone applications and measured, for instance, the execution time in processor cycles. Table 3 gives an overview of the measured results. Here, *bss* indicates the memory size for variables, *data* for parameters, and *code* for application's functions. Afterward, the executable codes compiled for the MicroBlaze processor were prepared and stored in the DDR memory. The Microblaze processor had version 11.0, and the running FreeRTOS version was 10.1.2. A DAG graph with the selected functions from the BEEBS benchmark was

TABLE 3. Measurement of execution times and examination of memory size on a MicroBlaze processor.

Benchmark	Total Size	Code Size	Data Size	Bss Size	Result Size	Execution Cycles
qsort	2332	1768	80	484	80	4894
fdct	3068	2812	128	128	128	10650
matmul	2256	1052	0	1204	400	86184
fir	6488	660	2948	2880	2800	857349

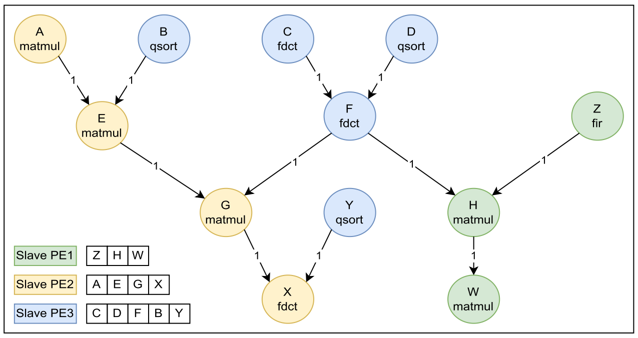


FIGURE 3. A DAG graph with dependency, partitioning, and calling order of tasks in each slave PE for the experimental study. The nodes' calling order and the DAG graph structure were arbitrarily selected.

prepared to examine the task assignment from the master PE to slave PEs. Figure 3 shows the graph's structure and tasks' interdependencies.

The edges show the sequence in which the respective result of the task should be sent to the successor node. In addition, it shows the distribution and calling order of tasks in the individual slave PEs. The goal was to explore the impact of power-saving techniques on real-time capability while supporting RTOS with offloading methodologies. Therefore, the research was conducted on an exemplary, arbitrarily selected task diagram to compare the different sets of results with each other. Nevertheless, the architecture enables the exploration of different benchmarks under different DAG graph structures.

A. CO-PROCESSOR BASED TASK SCHEDULING

As described in Section IV-B, we have offloaded the FreeRTOS task scheduling on the co-processor. The co-processor sent an interrupt request to FreeRTOS when context switching was required. Typically, a task is interrupted several times during execution to check for context switching. Therefore, we studied the effect of context switching during task execution. Figure 4 shows the results for task interruption with the BEEBS benchmark. The co-processor based task scheduling had a negligible deviation in all experiments. A small range of deviations only occurred in the *fdct* application. There was a more comprehensive range of deviations in the optimized and default software-based FreeRTOS task scheduling, which were highly irregular in all experiments. The time required for context switching could be reduced for all applications. By calling the context switching in a controlled manner, more computing time could be gained by a maximum of 3.68% for *fdct*, 6.35% for *qsort*, 0.67% for *matmul float*, and 0.25% for *matmul int*. Furthermore, we have measured the latency for the task scheduling and compared it with the default and optimized software-based FreeRTOS task scheduling. The task scheduling was running on the ARM Cortex-A9 processor at 667MHz. Table 4 shows the complete latency of the FreeRTOS task scheduling. All task scheduling had a deviation in the latency measurements. However, the co-processor based task scheduling still had the lowest deviation and thus led to lower jitter effects at run-time (also shown in Figure 4). Besides, the co-processor based task scheduling reduced the latency by 26.58% (18.3%) compared to the default (optimized) FreeRTOS task scheduling.

TABLE 4. Results of measured latency for the FreeRTOS task scheduling on co-processor.

Approach	Latency (in µs)	Variance (in µs)
Task scheduling on co-processor	0.29	±0.01
Optimized FreeRTOS scheduling	0.355	±0.015
FreeRTOS scheduling	0.395	±0.035

B. HARDWARE-BASED FREERTOS TASK SCHEDULING

As described in Section IV-A, we have accessed the hardware-based task scheduling through the AXI4-Lite and AXI-S protocols. We have compared latencies and consumed hardware resources for both protocols. Table 5 demonstrates the utilized hardware resources for XC7Z020-CLG484-1. The implementation did not require BRAM, DSP, and BUFG for both protocols. The round-robin policy had a significant impact on utilized hardware resources. Comparing both protocols showed that there was not much difference in the hardware resources consumed. Overall, the task scheduling also consumed few hardware resources with both protocols. In addition, we have investigated the latency of the hardware-based task scheduling for both AXI protocols. The AXI-S protocol needed 0.44µs to receive an update of the recent ready task. In the case of the round-robin policy, it took 0.65µs. Contrary, the AXI4-Lite protocol had a latency of 0.77µs with the round-robin policy. Comparing both protocols showed that the AXI4-Lite had a marginally higher latency than AXI-S. It was noticeable in both measurements that communication accounted for the essential latency time. The transaction latency was 0.63µs (0.75µs) for AXI-S (AXI4-Lite). Besides, we also changed the frequency of the task scheduling using the AXI4-Lite protocol to investigate the effect of frequency on latency. We run the task scheduling with 100MHz, 142MHz and 200MHz and measured an latency of 0.77µs, 0.74µs and 0.72µs. This approach was also insufficient to match the latency of the

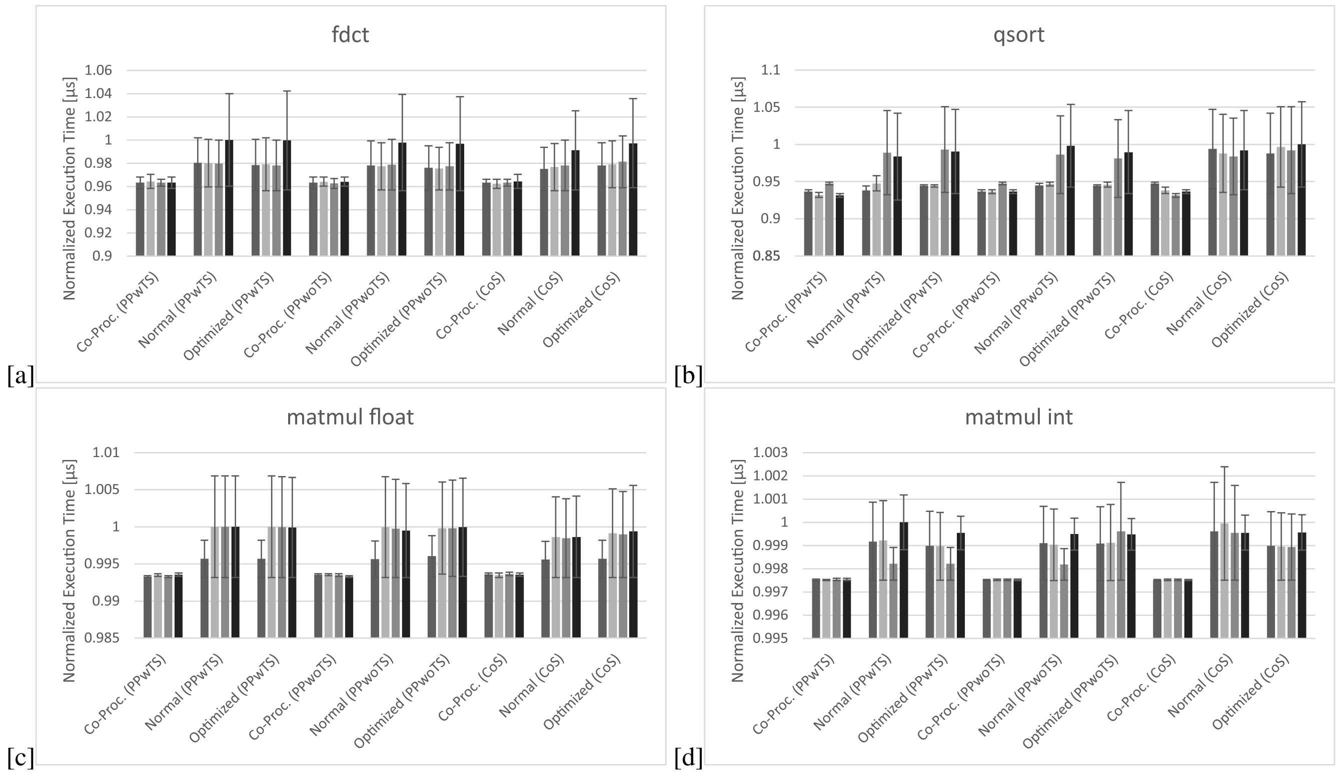


FIGURE 4. Four different applications ([a] fdct, [b] qsort, [c] matmul float, [d] matmul int) have been selected from the BEEBS benchmark [52]. A scheduling frequency of 500Hz, 1000Hz, 2000Hz and 4000Hz was set for each measurement. The experiments were performed with all three scheduling algorithms of FreeRTOS (PPwTS = Prioritized Preemptive Scheduling with Time Slicing, PPwoTS = Prioritized Preemptive Scheduling without Time Slicing, CoS = Cooperative Scheduling). Co-Proc. = Co-Processor based Task Scheduling, Normal/Optimized = Default/Optimized Software-based FreeRTOS Task Scheduling.

AXI-S protocol. In addition to the hardware execution, FreeRTOS also executes a small portion of assembly code for context switching. Table 6 shows the complete latencies for the hardware-based task scheduling with both AXI protocols, optimized, and default software-based FreeRTOS task scheduling. The default and optimized task scheduling in FreeRTOS had a deviation in the latency measurements. Contrary, the hardware-based task scheduling had no variance and thus no jitter effects in the execution. Overall, the task scheduling with the AXI-S protocol performed best and was 23.43% faster than the solution with the AXI4-Lite protocol. Besides, it was 33.33% faster than the default FreeRTOS task scheduling. Compared to the conducted measurements in [8], the improvement was 45.56%. But, the co-processor based task scheduling reduced the latency by 71%. However, the co-processor-based task scheduling ran at a frequency of 667MHz. Besides, the scalability of the task scheduling depends on the number of processors. The hardware-based task scheduling can be connected to any processor. Thus, it is adaptable and suitable for scalable systems. Moreover, the module can be integrated with just an AXI protocol call into FreeRTOS and thus requires no changes in the kernel as the co-processor-based implementation. The co-processor-based task scheduling required shared memory and synchronization mechanisms due to the shared resources. Due to the low utilized hardware resources and latency, we finally implemented the hardware-based task scheduling with the AXI-S protocol and connected it to the MicroBlaze processors.

C. HARDWARE UTILIZATION

Table 7 gives an overview of the hardware resource utilization of the proposed system architecture. It consumed 45.82% of the available slice logic blocks (LUTs) and 27.47% of the slice registers (FFs). Thus, the design can be extended for additional functionalities, e.g., hardware accelerators. The BRAM hardware usage was 51.43%. DMA, followed by MMCM consumed the most hardware resources in the master PE. In the case of a slave PE, the MicroBlaze processor had the most resource utilization. The hardware resources of the clock gate module used were negligible. Each slave PE had a homogeneous design structure and roughly equivalent resource consumption. Besides, the NoC router had a relatively low hardware utilization, as expected from the lightweight router based on XY routing scheme.

D. RUN-TIME MEASUREMENT

We performed run-time measurements for task assignment. Figure 3 shows how and in which order tasks were distributed to individual slave PEs. Furthermore, Figure 5 presents the conducted experimental results. We performed our investigation with different hyperperiods and identified the

TABLE 5. Resource consumption of the implemented hardware-based task scheduling on XC7Z020-CLG484-1.

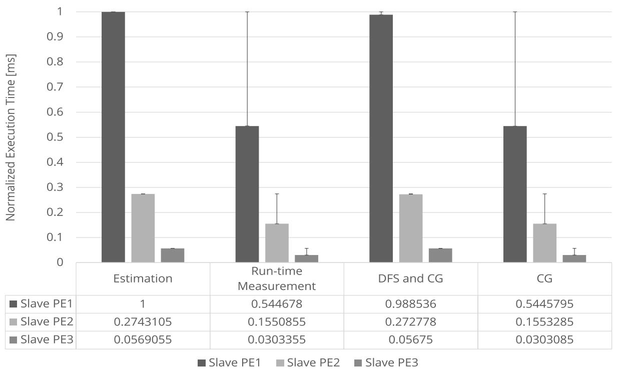
Resource	Scheduler with Round-Robin		Scheduler without Time-Slicing		Total
	AXI-Lite	AXI-Stream	AXI-Lite	AXI-Stream	Resources
LUT	758 (1.42%)	920 (1.72%)	642 (1.20%)	783 (1.47%)	53200
LUTRAM	55 (3.16%)	55 (3.16%)	31 (1.78%)	31 (1.78%)	17400
FF	1515 (1.42%)	1693 (1.59%)	1040 (0.97%)	1165 (1.09%)	106400

TABLE 6. Results of measured latency for the FreeRTOS task scheduling.

Approach	Latency (in µs)	Variance (in µs)
HW-scheduling (AXI-S)	0.98	0
HW-scheduling (AXI4-Lite)	1.28	0
Optimized FreeRTOS scheduling	1.295	±0.025
FreeRTOS scheduling	1.47	±0.36

same behavior in all of them. Therefore, we normalized all recorded values with an hyperperiod of 20ms. The task distribution estimates the total execution times on each slave PE. It has been found that slave PE1, PE2, and PE3 would take 10.37ms, 2.84ms, and 0.59ms, respectively. For this reason, the master defined a corresponding deadline of 20ms (normalized to 1ms), 5.48ms (0.27ms), and 1.13ms (0.05ms) for each slave PE. Since slave PE1 had the latest execution time, the hyperperiod was assigned, and the remaining two PEs got a period according to this value. After the estimation, a run-time measurement is performed to check the correctness of the estimated values. A total execution time of 10.89ms (0.54ms), 3.10ms (0.15ms), and 0.60ms (0.03ms) was measured, respectively. This resulted in a deviation of 4.76%, 8.25%, and 2.69% between the estimated and actual execution times. The deviation resulted from task execution in FreeRTOS and data packets in the network. Both behaviors are unpredictable and therefore have not been measured/determined at design time. The frequency of the individual PEs is then adjusted. In each iteration, the master PE checks whether the specified deadlines are met by all slave PEs. The frequency scaling and clock gating result shows that the slave PEs could always meet the deadlines, and the master PE could estimate a correct frequency of 55MHz, 57MHz, and 54MHz, respectively. At run-time, the behavior was observed, and if the execution time exceeded the defined deadline, the frequency was re-estimated. Note that the NoC operating frequency corresponds to the maximum frequency set in the slave PE. In this case, it was 57MHz. At a lower frequency, bandwidth and packet loss can occur. However, the master PE provided a correct frequency estimate every time. Therefore, no deadline was exceeded. In the last measurement, DFS was disabled, and the execution time was only measured with clock gating at 100MHz. Figure 5 shows that, as expected, clock gating did not affect the run-time measurement, and the initially measured condition was restored.

As Section IV-E described, the MicroBlaze processor goes into sleep mode when FreeRTOS calls the idle task. For this, the slave PE calls the sleep instructions and sends a sleep signal to the hardware module to bring him into



	Estimation	Run-time Measurement	DFS and CG	CG
■ Slave PE1	1	0.544678	0.988536	0.5445795
■ Slave PE2	0.2743105	0.1550855	0.272778	0.1553285
■ Slave PE3	0.0569055	0.0303355	0.05675	0.0303085

FIGURE 5. The run-time measurements from the individual slave PEs are shown. The execution time is normalized to the hyperperiod of 20ms. The error tolerance indicates the interval to the respective period. (DFS = Dynamic Frequency Scaling, CG = Clock Gating).

sleep mode. Besides, the clock gate module has an external wake-up signal that the interrupt controller triggers. There are two cases when the processor is woken up. The kernel of FreeRTOS measures the passing time using a tick-count variable. A timer interrupt increments this variable with a temporal resolution based on the chosen frequency. Each time the tick-count variable is incremented, the periodic scheduler checks whether context switching is required. Second, the processor receives packages from the NoC, which turns on the processor. We measured the latency, and the wake-up process took five clock cycles. To bring the processor into sleep mode took 30 clock cycles.

E. POWER AND ENERGY MEASUREMENT

We measured the PL power consumption without a design, and it was $P_{PL,stat} = 50.64$mW, which was considered in all measurements and removed from the results presented. The effect of power-saving techniques on the system architecture is presented in Figure 6. The master PE analyzes task dependencies, assigns tasks to slave PEs and monitors the performance with run-time measurements to check whether execution time meets defined hyperperiods (deadline) and thus real-time requirements. It predicts a frequency for local frequency scaling and adapts it if necessary at run-time. Figure 6a shows the results of DFS measurements indicating that power consumption was reduced from 370.14mW to 104.30mW with increasing periods. Moreover, Figure 6a also shows that the master PE assigned a new frequency to each slave PE at each period. Table 8 gives an overview of all frequencies that the master PE had determined and each slave PE set for DFS on each period. A measurement

TABLE 7. Resource consumption of the implemented complete system architecture on XC7Z020-CLG484-1.

	LUT	FF	BRAM	DSP	MMCM
Available Resources	53200	106400	140	220	4
Design Utilization	24378 (45.82%)	29231 (27.47%)	74 (52.86%)	15 (6.82%)	4 (100%)
Master PE	5258 (9.88%)	7014 (6.59%)	2 (1.43%)	0	1 (25%)
• DMA	1450 (2.72%)	2063 (1.94%)	2 (1.43%)	0	0
Slave PE	5859 (11.01%)	7112 (6.68%)	24 (17.14%)	5 (2.27%)	1 (25%)
• MicroBlaze Processor	2781 (5.23%)	2450 (2.30%)	0	5 (2.27%)	0
• Clock Gating	1	1	0	0	0
MMCM	1098 (2.06%)	1489 (1.39%)	0	0	1 (25%)
NoC Router	325 (0.61%)	167 (0.16%)	0	0	0

TABLE 8. Operating frequency results for each hyperperiod and slave PE, performed in frequency scaling.

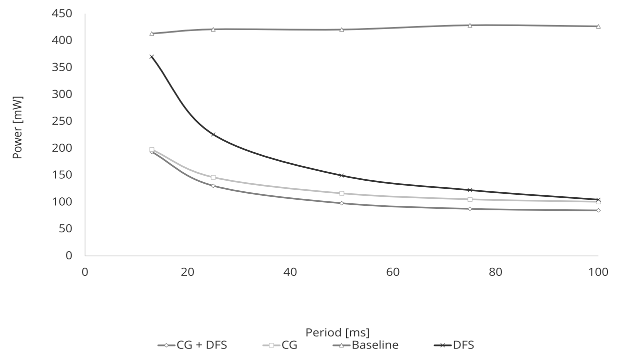
Hyperperiod	Slave PE1	Slave PE2	Slave PE3
13ms	11MHz	12MHz	11MHz
25ms	15MHz	16MHz	14MHz
50ms	22MHz	23MHz	21MHz
75ms	44MHz	45MHz	43MHz
100ms	83MHz	87MHz	85MHz

without power-savings (denoted as baseline in Figure 6a) was conducted at ($f_{\max}$) 100MHz and used as a comparison baseline. Figure 6b shows the achieved improvement by power-saving techniques related to no optimization in the design. Thus, power consumption was improved by 9.28% (67.52%) in the lowest (highest) period with DFS. However, clock gating with (and without) DFS resulted in better power optimization. The power was reduced from 192.72mW to 100.38mW for clock gating only and thus improved by 46.45% (68.35%) in the lowest (highest) period. The combination with DFS reduced power consumption by 47.41% (71.70%) in the lowest (highest) period. Each slave PE performed a clock gating whenever the processor was idle. Lowering the period reduced the available idle time across slave PEs and thus degraded the achievable power reduction by clock gating. Furthermore, the combination with DFS resulted in a slight improvement of the power consumption because frequency scaling increased idle time. However, clock gating can only serve a limited number of hardware components. In our case, it was the MicroBlaze processor and BRAM memory in the respective slave PE. All other components remain active as they trigger an interrupt to specific events. For instance, a FIFO triggers an interrupt to inform the processor about incoming packages that must continuously operate in the proposed design. On the other hand, DFS can be applied to all hardware components in PL and is in this sense more universally applicable. However, there is a task dependency associated with inter-task or processor communication in the system, whereby DFS cannot be fully exploited. Otherwise, the processors could have executed their tasks with the highest frequency and gone into sleep mode with clock gating earlier.

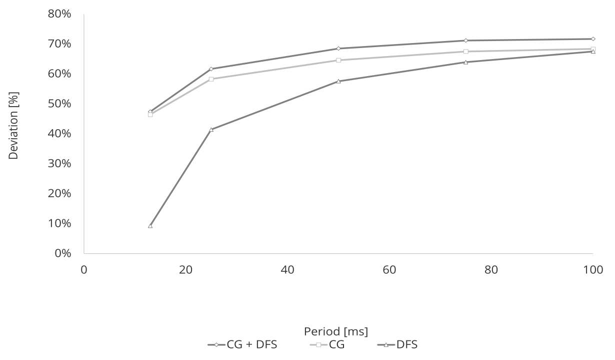
Figure 7 shows the dynamic power consumption with normal task execution without optimization (baseline), voltage scaling, and clock gating. In the experiments, the voltage was scaled down by 15%, and a hyperperiod of 50ms was set. In the beginning, the effect of voltage scaling was noticeable without a task execution. The power was at 149.7mW and reduced by DVS to 79.21mW. The baseline had a dynamic power dissipation of 421.13mW, and DVS reduced it by 27.63% to 304.74mW. As before discussed, the clock gating greatly impacted the power. Thus, it reduced the power by 73.32% to 112.35mW. However, a combination with DVS resulted in an overall saving of 85.39%. The improvement that clock gating brings to DVS was 79.81%. Figure 7 also depicts the dynamic behavior of clock gating on power consumption. Here, the power was initially 149.7mW. Afterward, the slave PEs were in an idle phase for 50ms, which resulted in power consumption of 84.47mW. During task execution, the power increased to 112.35mW. Thus, the design obtained a power reduction of 24.81% in slack times.

Figure 8a shows the power consumption resulting from DVFS in combination with clock gating for hyperperiods between 13ms and 100ms. The nominal voltage was 1V and was reduced in 0.05V steps to 0.85V. Below this threshold, results were getting corrupted, and FreeRTOS had a problem caused by a lack of heap memory [19]. This value was, therefore, the defined threshold for the measurements. The master PE specified an operating frequency on each hyperperiod (shown in Table 8) and thus scaled from 100MHz to the indicated frequency value. However, the power was reduced by a maximum of 47.27% (from 1V to 0.85V at a period of 100ms) with power-savings. In addition, all slave PEs met the specified deadlines while adapting power with optimization techniques at run-time. Besides, energy consumption is illustrated in Figure 8b-d. Here, a linear increase with a rising period can be seen. Each slave PE had a corresponding energy consumption according to the number of assigned tasks (as presented in Figure 3). Here, the baseline describes the case in which there is no optimization in the system. Clock gating and frequency scaling reduced power consumption by 53.34% at 13ms and 80% at 100ms (compared to 1V). Voltage scaling achieved additional reductions. It diminished the energy consumption by 69.33% at 13ms and 89.47% at 100ms while scaling the voltage to 0.85V. If the underlying hardware allows voltage scaling, the design can significantly reduce power/energy consumption and thus result in more savings.

The mapping strategy proposed in [44] was compared with a random algorithm. Therefore, we have compared the random and heuristic mapping algorithm's impact on power consumption (Figure 9). The power consumption was



[a]



[b]

FIGURE 6. **A comparison of power-savings (clock gating and DFS) supported by design has been conducted. [a] presents the resulting power measurements for a hyperperiod of 13ms, 25ms, 50ms, 75ms, and 100ms. [b] shows the deviation to the baseline without optimization according to the measured results in [a]. (CG = Clock Gating, DFS = Dynamic Frequency Scaling).**

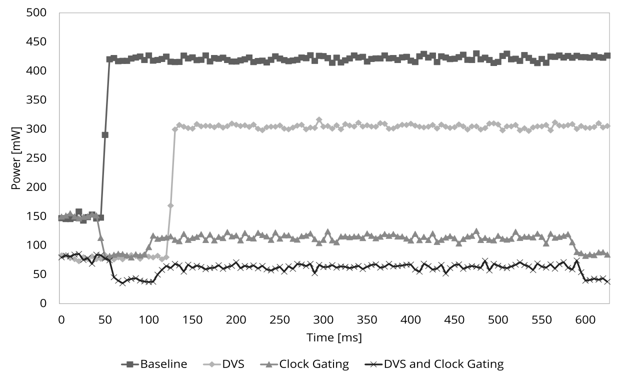


FIGURE 7. **Effect of voltage scaling and clock gating on dynamic power dissipation. The baseline describes the task execution without optimization. The nominal voltage is at 1V and was scaled to 850mV. The measurements were performed for a hyperperiod of 50ms.**

measured for different hyperperiods between 11 and 100ms. The number of messages on the edges of the DAG graph for inter-task communication (Figure 3) was increased step-wise from one (1M) to a maximum of nine (9M) messages in the experiments. Each slave PE had 32KiB memory available for tasks, so the maximum number of messages could be increased to nine in this work. Figure 9 shows only all tasks successfully mapped through the mapping strategy in the respective hyperperiod without exceeding the deadline. Slave PEs had sufficient processing time for task execution at higher hyperperiods, resulting in longer idle times and lower power consumption. Moreover, Figure 9 shows that mapping was not possible for lower periods because tasks exceeded the defined deadline and thus violated the real-time requirement. The influence was notable for hyperperiods below 22ms. The reason was that more messages were sent or tasks were suspended for a certain time due to the inter-task dependence. The proposed mapping strategy aims to assign dependent tasks to the same slave PE and thus avoid messaging through the network. Besides, it monitors the execution time for each slave PE and assigns tasks accordingly. Therefore, the heuristic mapping assigned tasks more efficiently at lower periods than the random algorithm without violating real-time requirements. This was why only the proposed solution could map tasks with nine messages to slave PEs. Nevertheless, both algorithms resulted in a significantly similar power consumption for all hyperperiods with one message (1M HM/RM). However, the heuristic mapping could meet all real-time requirements until an hyperperiods of 11ms, whereby the random mapping algorithm successfully assigned tasks until 13ms. On the other hand, the proposed mapping strategy resulted in more power-saving for high number of messages. For instance, the heuristic mapping resulted in a 24.99% lower power consumption at 22ms for eight messages compared to the random algorithm. Thus, the proposed mapping strategy can better meet real-time requirements while consuming less power.

F. IMPACT OF POWER-SAVING TECHNIQUES ON REAL-TIME CAPABILITY

One aspect of this work was to balance real-time capability and energy efficiency. This work considered the real-time capability differently at master PE and slave PEs. On the one hand, hardware-based task scheduling led to better scheduling performance and improved context-switching by 33.33% at slave PEs. The approach resulted in no variance and, thus, no jitter effects in the execution. The impact on power consumption was not measurable because of the infinitesimally short processing time of the context switch and the longer retrieving time of a power value. However, the hardware-based task scheduling enabled FreeRTOS, for instance, to react faster to incoming messages with enhanced scheduling performance in the design.

On the other hand, the master PE checked whether all slave PEs met the real-time requirements. In case of a deadline violation on a slave PE, the master PE computed a new frequency. The slave PE then scaled operating frequency based on the computed frequency. This reduced the power dissipation while meeting real-time requirements. A lower energy consumption was consumed at lower hyperperiods. However, the experiments showed that lower hyperperiods led tasks to exceed the defined deadline, thus violating the real-time requirement. This could be prevented with

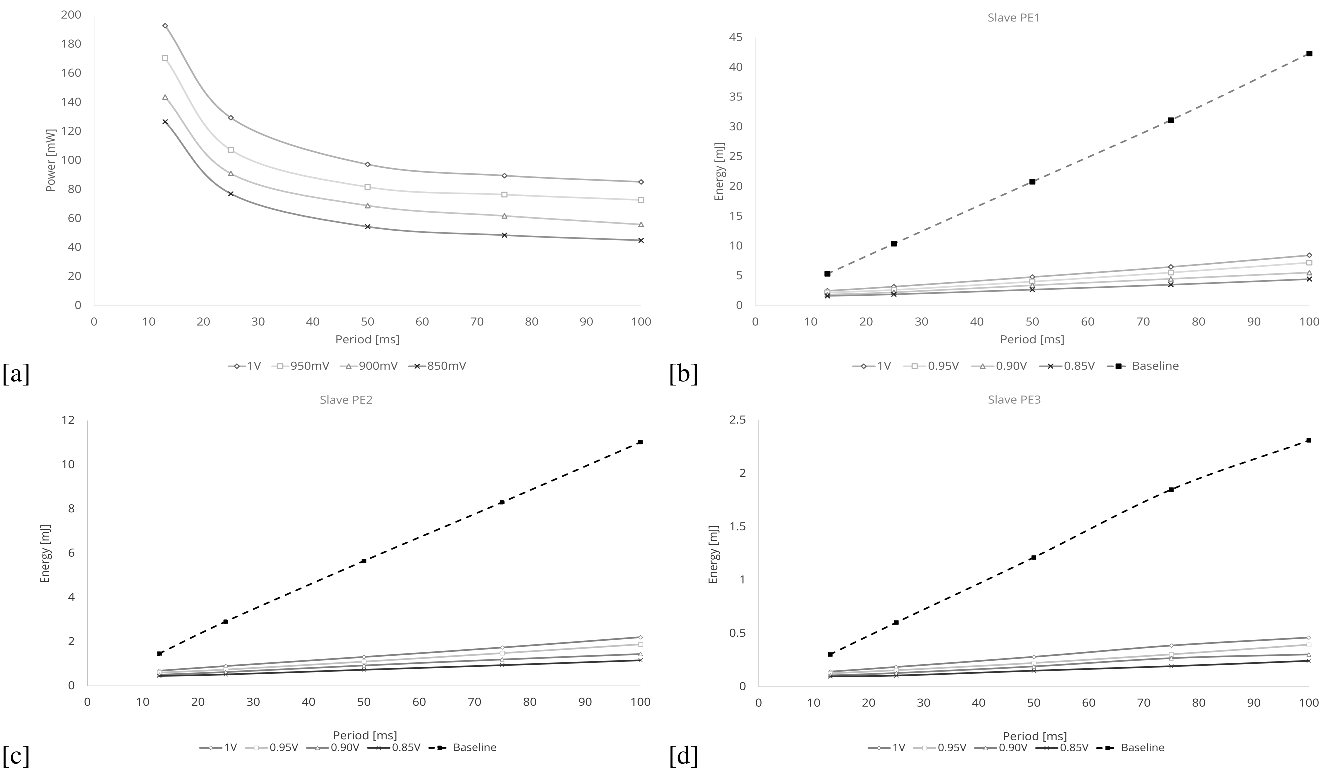


FIGURE 8. **The power-saving techniques provided by the architecture (clock gating and frequency scaling) were linked to voltage scaling. The impact of voltage scaling on power and energy consumption on each slave PE is presented. All measurements were conducted for a hyperperiod of 13ms, 25ms, 50ms, 75ms, and 100ms. [a] presents the power consumption for applying DVFS and clock gating for the application. [b]-[d] show the energy consumption for the respective slave PE.**

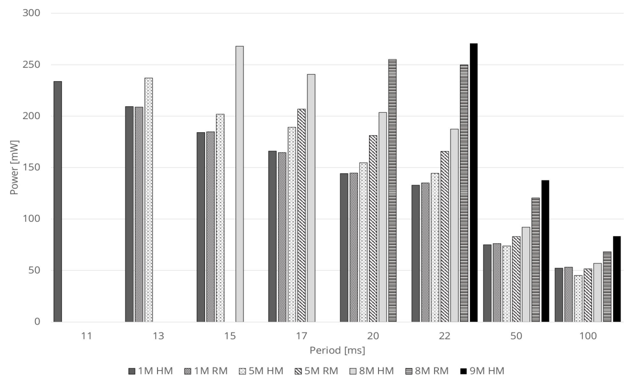


FIGURE 9. **Effect of random (RM) and heuristic mapping (HM) strategy on power consumption. The number of messages represented as edges in the DAG graph was increased by 1, 5, 8, and 9 (9M) messages. Clock gating was performed. The overall power consumption was measured.**

the power-saving techniques used while meeting the real-time requirements. Besides the frequency scaling, slave PEs applied clock gating while executing the idle task in FreeRTOS. The master PE scaled the voltage for the reconfigurable hardware and, thus, all slave PEs. This also reduced power consumption while meeting all real-time requirements. The master PE executed the heuristic task mapping that aimed to assign dependent tasks to the same slave PE to avoid inter-processor communication, thus reducing power consumption. The evaluations showed that the proposed strategy could map tasks with nine messages to slave PEs while reducing power and meeting real-time requirements. Using these various techniques at different levels in the system architecture led to reducing overall power consumption while not violating real-time requirements.

VI. CONCLUSION

This work investigated the impact of power-saving techniques on real-time requirements while supporting RTOS with offloading methodologies. The proposed solution was performed on a heterogeneous multi-core architecture with an NoC and four PEs in a 2D-mesh topology. The master PE monitored power consumption, dynamically scaled voltage, and assigned tasks to slave PEs with a heuristic mapping strategy. All slave PEs applied frequency scaling and clock gating to reduce power. The task scheduling was offloaded on a co-processor and in hardware. As future work, it is planned to compare the proposed heuristic task mapping with other mapping algorithms. The hardware-based task scheduling will be enhanced with a power-aware scheduling algorithm.

ACKNOWLEDGMENT

The authors would like to thank Hans-Harro Horn for his support in implementing hardware-based task scheduling in this work.

REFERENCES

[1] C. Wulf, M. Willig, G. Akgün, and D. Göhringer, "Operating systems for reconfigurable computing: Concepts and survey," in *Towards Ubiquitous Low-power Image Processing Platforms*. Cham, Switzerland: Springer, 2021, pp. 61–78.

[2] A. B. Lang, K. H. Andersen, U. P. Schultz, and A. S. Sørensen, "HartOS—A hardware implemented RTOS for hard real-time applications," in *Proc. IEEE Int. Conf. Program. Devices Embedded Syst.*, vol. 45, Jan. 2012, pp. 207–213.

[3] G. Akgün, L. Kalms, and D. Göhringer, "Resource efficient dynamic voltage and frequency scaling on Xilinx FPGAs," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Cham, Switzerland: Springer, 2020, pp. 178–192.

[4] S. Li and F. Broekaert, "Low-power scheduling with DVFS for common RTOS on multicore platforms," *ACM SIGBED Rev.*, vol. 11, no. 1, pp. 32–37, 2014.

[5] B. Ranjbar, T. D. A. Nguyen, A. Ejlali, and A. Kumar, "Power-aware runtime scheduler for mixed-criticality systems on multicore platform," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 10, pp. 2009–2023, Oct. 2021.

[6] T. Poggi, P. Onaindia, M. Azkarate-Askatsua, K. Grüttner, M. Fakih, S. Peiró, and P. Balbastre, "A hypervisor architecture for low-power real-time embedded systems," in *Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2018, pp. 252–259.

[7] Y. Tang and N. W. Bergmann, "A hardware scheduler based on task queues for FPGA-based embedded real-time systems," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1254–1267, May 2015.

[8] J. Pereira, D. Oliveira, S. Pinto, N. Cardoso, V. Silva, T. Gomes, J. Mendes, and P. Cardoso, "Co-designed FreeRTOS deployed on FPGA," in *Proc. Brazilian Symp. Comput. Syst. Eng.*, Nov. 2014, pp. 121–125.

[9] I. Bahri, M. A. Benkhelifa, and E. Monmasson, "HW-SW real-time operating system for AC drive applications," in *Proc. Int. Symp. Power Electron. Power Electron., Electr. Drives, Autom. Motion*, Jun. 2012, pp. 194–199.

[10] M. Sindhwani, T. Oliver, D. L. Maskell, and T. Srikanthan, "RTOS acceleration techniques—Review and challenges," *Proc. 6th Real-Time Linux Workshop*, 2004, pp. 123–128.

[11] M. Vetromille, L. Ost, C. A. M. Marcon, C. Reif, and F. Hessel, "RTOS scheduler implementation in hardware and software for real time applications," in *Proc. 17th IEEE Int. Workshop Rapid Syst. Prototyping (RSP)*, Jun. 2006, pp. 163–168.

[12] L. Lindh, "FASTHARD—A fast time deterministic HARDware based real-time kernel," in *Proc. 4th Euromicro Workshop Real-Time Syst.*, Jun. 1992, pp. 21–25.

[13] T. Nakano, Y. Komatsudaira, A. Shiomi, and M. Imai, "VLSI implementation of a real-time operating system," in *Proc. Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 1997, pp. 679–680.

[14] V. J. Mooney and D. M. Blough, "A hardware–software real-time operating system framework for SoCs," *IEEE Design Test Comput.*, vol. 19, no. 6, pp. 44–51, Nov. 2002.

[15] S. E. Ong, S. C. Lee, N. B. Z. Ali, and F. A. B. Hussin, "SEOS: Hardware implementation of real-time operating system for adaptability," in *Proc. 1st Int. Symp. Comput. Netw.*, Dec. 2013, pp. 612–616.

[16] M. Song, S. H. Hong, and Y. Chung, "Reducing the overhead of real-time operating system through reconfigurable hardware," in *Proc. 10th Euromicro Conf. Digit. Syst. Design Architectures, Methods Tools (DSD)*, Aug. 2007, pp. 311–316.

[17] S. Chandra, F. Regazzoni, and M. Lajolo, "Hardware/software partitioning of operating systems: A behavioral synthesis approach," in *Proc. 16th ACM Great Lakes Symp. VLSI*, New York, NY, USA, Apr. 2006, pp. 324–329.

[18] M. Silva, T. Gomes, and S. Pinto, "Agnostic hardware-accelerated operating system for low-end IoT," in *Proc. IEEE 28th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2022, pp. 21–30.

[19] R. Berry, *Mastering the FreeRTOS Real Time Kernel*. Accessed: Jan. 17, 2024. [Online]. Available: https://www.freertos.org/Documentation/RTOS_book.html

[20] M. Varela, R. Cayssials, E. Ferro, and E. Boemo, "Real-time scheduling coprocessor for NIOS II processor," in *Proc. VIII Southern Conf. Program. Log.*, Mar. 2012, pp. 1–6.

[21] G. Abich, M. G. Mandelli, F. R. Rosa, F. Moraes, L. Ost, and R. Reis, "Extending FreeRTOS to support dynamic and distributed mapping in multiprocessor systems," in *Proc. IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Dec. 2016, pp. 712–715.

[22] E. Qaralleh, D. Lima, T. Gomes, A. Tavares, and S. Pinto, "HcM-FreeRTOS: Hardware-centric FreeRTOS for ARM multicore," in *Proc. IEEE 20th Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2015, pp. 1–4.

[23] G. Xie, X. Xiao, H. Peng, R. Li, and K. Li, "A survey of low-energy parallel scheduling algorithms," *IEEE Trans. Sustain. Comput.*, vol. 7, no. 1, pp. 27–46, Jan. 2022.

[24] *ZC702 Board User Guide*, document UG850, Xilinx, San Jose, CA, USA, 2019, pp. 1–78.

[25] *InTune Automatically Compensated Digital PoL Controller with Driver and PMBus Telemetry*, document MAX15301, Maxim Integrated, San Jose, CA, USA, 2013, pp. 1–30.

[26] *Zynq-7000 AP SoC Low Power Techniques Part 3—Measuring ZC702 Power with a Standalone Application Tech Tip*. Accessed: Apr. 30, 2023. [Online]. Available: https://xilinx-wiki.atlassian.net

[27] *Zynq-7000 SoC: DC and AC Switching Characteristics*, document DS187, Xilinx, San Jose, CA, USA, 2018, pp. 1–72.

[28] M. Hosseinabady and J. L. Nunez-Yanez, "Run-time power gating in hybrid ARM-FPGA devices," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, pp. 1–6.

[29] *Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices*, document XAPP1159, Xilinx, San Jose, CA, USA, 2013, pp. 1–96.

[30] G. Akgün, M. Ali, and D. Göhringer, "Power-aware computing systems on FPGAs: A survey," in *Proc. 31st Int. Conf. Field-Programmable Log. Appl. (FPL)*, Aug. 2021, pp. 45–51.

[31] X. Wei, Y. Liang, T. Wang, S. Lu, and J. Cong, "Throughput optimization for streaming applications on CPU-FPGA heterogeneous systems," in *Proc. 22nd Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2017, pp. 488–493.

[32] S. M. A. H. Jafri, M. A. Tajammul, A. Hemani, K. Paul, J. Plosila, and H. Tenhunen, "Energy-aware-task-parallelism for efficient dynamic voltage, and frequency scaling, in CGRAs," in *Proc. Int. Conf. Embedded Comput. Syst., Architectures, Modeling, Simulation (SAMOS)*, Jul. 2013, pp. 104–112.

[33] M. Mandelli, G. Castilhos, G. Sassatelli, L. Ost, and F. G. Moraes, "A distributed energy-aware task mapping to achieve thermal balancing and improve reliability of many-core systems," in *Proc. 28th Symp. Integr. Circuits Syst. Design (SBCCI)*, Aug. 2015, pp. 1–7.

[34] J. Ma, F. Fu, Z. Liu, Z. Wu, and J. Wang, "μC-OS II-based operating system design for cluster in NoC-based MPSoC," in *Proc. IEEE Int. Conf. Signal Process., Commun. Comput. (ICSPCC)*, Aug. 2013, pp. 1–5.

[35] A. Norollah, Z. Kazemi, N. Sayadi, H. Beitollahi, M. Fazeli, and D. Hely, "Efficient scheduling of dependent tasks in many-core real-time system using a hardware scheduler," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2021, pp. 1–7.

[36] A. Norollah, Z. Kazemi, D. Derafshi, H. Beitollahi, and M. Fazeli, "Protecting security-critical real-time systems against fault attacks in many-core platforms," in *Proc. CPSSI 4th Int. Symp. Real-Time Embedded Syst. Technol. (RTEST)*, May 2022, pp. 1–6.

[37] D. Derafshi, A. Norollah, M. Khosroanjam, and H. Beitollahi, "HRHS: A high-performance real-time hardware scheduler," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 4, pp. 897–908, Apr. 2020.

[38] C. Wulf, M. Willig, and D. Goehringer, "RTOS-supported low power scheduling of periodic hardware tasks in flash-based FPGAs," *Microprocessors Microsyst.*, vol. 92, Jul. 2022, Art. no. 104566.

[39] L. Mo, Q. Zhou, A. Kritikakou, and J. Liu, "Energy efficient, real-time and reliable task deployment on NoC-based multicores with DVFS," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2022, pp. 1347–1352.

[40] L. Mo, A. Kritikakou, and O. Sentieys, "Controllable QoS for imprecise computation tasks on DVFS multicores with time and energy constraints," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 8, no. 4, pp. 708–721, Dec. 2018.

[41] W. Qiu, Y. Chen, D. Chen, T. Su, and S. Yang, "Run-time hierarchical management of mapping, per-cluster DVFS and per-core DPM for energy optimization," *Electronics*, vol. 11, no. 7, pp. 1–19, 2022.

[42] K. R. Basireddy, A. K. Singh, B. M. Al-Hashimi, and G. V. Merrett, "AdaMD: Adaptive mapping and DVFS for energy-efficient heterogeneous multicores," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2206–2217, Oct. 2020.

[43] M. Gupta, L. Bhargava, and S. Indu, "Mapping techniques in multicore processors: Current and future trends," *J. Supercomput.*, vol. 77, no. 8, pp. 9308–9363, Aug. 2021.

[44] Z. Wu, F. Fu, L. Wang, J. Wang, and F. Lai, "Energy-aware dynamic scheduling for NoC-based MPSoCs," in *Proc. Academic Int. Symp. Optoelectronics Microelectron. Technol.*, Oct. 2011, pp. 308–312.

[45] *SDx Pragma Reference Guide*, document UG1253, San Jose, CA, USA, 2019, pp. 1–103.

[46] *ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition*, document ID040418, Arm Limited, Cambridge, U.K., 2018, pp. 1–2720.

[47] J. Rettkowski and D. Göhringer, "RAR-NoC: A reconfigurable and adaptive routable network-on-chip for FPGA-based multiprocessor systems," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig14)*, Dec. 2014, pp. 1–6.

[48] *Zynq-7000 SoC Technical Reference Manual*, document UG585, Xilinx, San Jose, CA, USA, 2021, pp. 1–1825.

[49] *7 Series FPGAs Clocking Resources*, document UG472, Xilinx, San Jose, CA, USA, 2018, pp. 1–114.

[50] *MicroBlaze Processor Reference Guide*, document UG984, Xilinx, San Jose, CA, USA, 2018, pp. 1–388.

[51] *UCD92xx Digital PWM System Controller PMBus Command Reference*, document SLUU337, Texas Instruments, Dallas TX, USA, 2018, pp. 1–50.

[52] J. Pallister, S. J. Hollis, and J. Bennett, "BEEBS: Open benchmarks for energy measurements on embedded platforms," 2013, *arXiv:1308.5174*.

GÖKHAN AKGÜN received the bachelor's and master's degrees in electrical engineering and information technology from Ruhr University Bochum, in 2014 and 2017, respectively. He is currently pursuing the Ph.D. degree in computer science with the Chair of Adaptive Dynamic Systems (ADS), Technische Universität Dresden, Germany. His current research interests include reconfigurable computing, real-time operating systems, power management, and hardware-software co-design of signal/image processing algorithms.

BOZHIDAR KOLAROV received the Diploma degree in information systems engineering from Technische Universität Dresden, Germany, in 2022. He completed the thesis titled "Multiobjective Task Mapping in a Multi-Core Architecture" with the Chair of Adaptive Dynamic Systems (ADS).

HENDRIK KALBERLAH received the degree in computer science with Technische Universität Dresden. He completed the thesis entitled "Implementation of an RTOS Scheduler on the Co-Processor of a Xilinx System-on-Chip" with the Chair of Adaptive Dynamic Systems (ADS), in 2019.

CORNELIA WULF is currently pursuing the Ph.D. degree. She studied computer science with Johann Wolfgang Goethe-Universität, Frankfurt am Main, and during her residence in England with FernUniversität Hagen. Since August 2018, she has been an Assistant Researcher in computer science with the Chair of Adaptive Dynamic Systems (ADS), Technische Universität Dresden, Germany. She collected work experience as a Research Assistant with FernUniversität Hagen and the industry development of machine control programs.

MICHAEL WILLIG (Graduate Student Member, IEEE) received the degree in computer science in Leipzig and Munich. He is currently pursuing the Ph.D. degree in computer science with the Chair of Adaptive Dynamic Systems (ADS), Technische Universität Dresden, Germany. He is a Student Member of the FernUniversität Hagen for Industrial and Organizational Psychology. His computer science professional work included different industrial areas, such as SCADA with real-time OS QNX, development of domain-specific languages (DSL), medical data mining, and software development for embedded systems up to cluster server environments (J2EE).

JENS RETTKOWSKI received the B.Sc. degree in microtechnology from the Westphalian University of Applied Sciences, in 2011, the M.Sc. degree in electrical engineering and information technology from Ruhr University Bochum, Germany, in 2014, and the Ph.D. degree from Technische Universität Dresden, Germany, in 2021. He is currently a Professor of electronics with the Fachhochschule Dortmund–University of Applied Sciences and Arts, Germany. His research interests include self-adaptive networks-on-chip, multiprocessor systems-on-chip, and reconfigurable computing.

DIANA GÖHRINGER (Member, IEEE) received the Ph.D. degree (summa cum laude) in electrical engineering and information technology from the Karlsruhe Institute of Technology (KIT), Germany, in 2011. From 2013 to 2017, she was an Assistant Professor and the Head of the MCA (application-specific multi-core architectures) Research Group, Ruhr University Bochum (RUB), Germany. She is currently a Professor with the Chair of Adaptive Dynamic Systems (ADS), Technische Universität Dresden, Germany. She is the author and coauthor of more than 200 publications in international journals, conferences, and workshops. Her research interests include reconfigurable computing, multiprocessor systems-on-chip, networks-on-chip, simulators/virtual platforms, hardware-software-co-design, and runtime systems. She serves as a technical program committee member for several international conferences and workshops. She is a reviewer and a guest editor of several international journals.