Article

Runtime Adaptive IoMT Node on Multi-Core Processor Platform

Matteo Antonio Scrugli *, Paolo Meloni, Carlo Sau and Luigi Raffo

Depratment of Electrical and Electronic Engineering, University of Cagliari, 09124 Cagliari, Italy; paolo.meloni@unica.it (P.M.); carlo.sau@unica.it (C.S.); raffo@unica.it (L.R.)
* Correspondence: matteo.scrugli@unica.it

Abstract: The Internet of Medical Things (IoMT) paradigm is becoming mainstream in multiple clinical trials and healthcare procedures. Thanks to innovative technologies, latest-generation communication networks, and state-of-the-art portable devices, IoTM opens up new scenarios for data collection and continuous patient monitoring. Two very important aspects should be considered to make the most of this paradigm. For the first aspect, moving the processing task from the cloud to the edge leads to several advantages, such as responsiveness, portability, scalability, and reliability of the sensor node. For the second aspect, in order to increase the accuracy of the system, state-of-the-art cognitive algorithms based on artificial intelligence and deep learning must be integrated. Sensory nodes often need to be battery powered and need to remain active for a long time without a different power source. Therefore, one of the challenges to be addressed during the design and development of IoMT devices concerns energy optimization. Our work proposes an implementation of cognitive data analysis based on deep learning techniques on resource-constrained computing platform. To handle power efficiency, we introduced a component called Adaptive runtime Manager (ADAM). This component takes care of reconfiguring the hardware and software of the device dynamically during the execution, in order to better adapt it to the workload and the required operating mode. To test the high computational load on a multi-core system, the Orlando prototype board by STMicroelectronics, cognitive analysis of Electrocardiogram (ECG) traces have been adopted, considering single-channel and six-channel simultaneous cases. Experimental results show that by managing the sensory node configuration at runtime, energy savings of at least 15% can be achieved.

Keywords: adaptive system; health information management; Internet of Things; low-power electronics; multi-core processing; neural networks; remote sensing; runtime; wearable sensors

check for updates

Citation: Scrugli, M.A.; Meloni, P.; Sau, C.; Raffo, L. Runtime Adaptive IoMT Node on a Multi-Core Processor Platform. *Electronics* **2021**, *10*, 2572. https://doi.org/10.3390/electronics10212572

Academic Editor: Manuel E. Acacio

Received: 3 August 2021
Accepted: 14 October 2021
Published: 21 October 2021

1. Introduction

The next generation of biomedical devices is making great strides in the scientific community. Thanks to the use of integrated System-of-Systems (SoSs), it is possible to efficiently connect wearable sensor nodes, medical devices, and applications with healthcare information systems, which can be very useful in several scenarios. In the hospital environment, it is possible to increase the effectiveness of monitoring and therefore the treatment of patients. If applied in a domestic environment, in addition to greatly improving communication between the patient and the healthcare provider, it is possible to significantly reduce public medical costs. It is estimated that this market value could reach $136 billion by the end of 2021 [1].

The Internet of Things (IoT) paradigm, or in this case the Internet of Medical Things (IoMT), leads to additional benefits for the constant monitoring of vital parameters, data transmission and collection, and server-side or edge-side analysis. Currently, in the literature, some critical points are being widely questioned, and alternative solutions are being proposed to improve aspects concerning: responsiveness, scalability, privacy, and security. In Reference [2], different aspects of state-of-the-art devices are discussed, and the crucial points regarding the improvements still needed on IoMT devices in the field of ambient assisted living are highlighted. For the scalability aspect, in Reference [3], the authors propose a multi-agent system and services that allow the creation of dynamic, easily extended, and scalable solutions and do not require technical knowledge on the underlying technologies. In Reference [2], it is shown how proprietary IoT device solutions can be weak from a privacy and security standpoint. It is then demonstrated how a collaborative IoT network can lead to greater resistance to malicious attacks or how the use of end-to-end encryption methods prevents man-in-the-middle attacks. In our previous work [4] we show how, by moving cognitive and non-cognitive processing on-edge, there are not only improvements in responsiveness but also in power consumption. Reference [2] also discusses how the fusion of on-edge cognitive processing and the potentialities related to IoT networks brings numerous benefits in terms of robustness in contrasting environmental changes, responsiveness, human intervention reduction, and lower energy consumption.

Most of the efforts aiming in this direction focus on the adoption of an edge-computing approach, and also in our case, we focus the purpose of our work in this direction. Moving task processing directly to the node, even partially, rather than transmitting all of the raw data sampled by the sensor to the cloud, can lead to several benefits. For example, processing the data also means having the ability to obtain more compact information to transmit, or even, the node may decide to cease communication. Less data transmission by the node leads to energy savings and lower bandwidth requirements for communication between the node and the cloud. Again, near-sensor processing allows for immediate feedback, leading to improvements in both latency and low server reachability. Finally, decreasing the amount of information transmitted by the node reduces the risk that sensitive information can be intercepted by malicious agents. However, constraints on battery lifetime and power consumption for IoMT nodes are still very demanding, and a local execution of fairly complex data analysis tasks requires careful tuning of platform and application.

In this work, we present a wearable IoMT device capable of supporting the on-edge processing task, making progress regarding the dynamic hardware/software reconfiguration that better adapts the node to the variation of workload or operating mode and aims at optimizing its energy efficiency. At the base of our work, we present the component called Adaptive runtime Manager (ADAM), which dynamically manages the hardware/software reconfiguration of the device, in this case, on a multi-core device. ADAM can be invoked by an external reconfiguration message, or it can trigger itself internally: for example, it can be invoked if a certain event is detected within the sampled signal. As we see better later, ADAM creates and manages a network of processes for each operating mode, and the different processes communicate through FIFOs. The reconfiguration of the process network consists of enabling or disabling certain processes and then rerouting the FIFOs correctly. ADAM can also manage hardware parameters (such as system frequency, supply voltage, and power gating) to respond optimally to workload variations, lowering power consumption, and respecting real-time constraints. The prototype named Orlando and produced by STMicroelectronics was considered in our study.

The remainder of this paper is as follows: Section 2 describes the landscape of related works in the literature; Section 3 presents the proposed template for the node, the reference target platform, and the selected application model; Section 4 talks about the proposed solution for making the system adaptive; and Section 5 discusses our experimental results. Finally, Section 6 outlines our conclusions.

2. Related Works

In the literature, it is possible to find numerous works proposing an IoMT sensor network system in both hospital and home settings [5–7]. Most of these studies exploit a cloud-based analysis: data are usually encapsulated in standard formats and sent to remote servers for data mining. Most of these studies involve the use of cloud-based systems that collect and analyze the raw data received from the sensor, which is often wearable and portable. Many of these IoMT devices, whether commercial or not, provide days or weeks of autonomy [8,9]. However, few systems propose to move the processing task directly to the node, which is used only for the transmission of the raw signals sampled by the sensor [10–13]. The use of artificial intelligence is gaining momentum in many areas, often used because of the simplicity of implementation and the high accuracy potentially offered by these techniques. Their effectiveness has already been widely discussed on high-performance computing platforms. Some examples are [14], which uses a 3.5 GHz Intel Core i7-7800X CPU, RAM 32 GB, and a GPU NVIDIA Titan X (Pascal, 12 GB), [15]; where a NVIDIA GeForce GTX 1080 Ti (11 GB) is used; or [16], based on a i7-4790 CPU at 3.60 GHz. However, the migration of state-of-the-art cognitive techniques to low-power and resource-constrained devices still remains an open question.

There is a growing body of work that places artificial intelligence (AI) at the heart of processing mechanisms, often charged with recognizing certain detectable events from the acquired signal. In References [17,18], the authors exploit Artificial Neural Networks (ANNs) to detect specific conditions from the collected data. In particular, in Reference [18] the authors propose a system to detect the emotional state of the individual, i.e., happiness or sadness. Power-saving techniques are applied, such as enabling on-edge processing or reconfiguring the device at run-time based on workload or operating mode choice. The anomaly detection on an Electrocardiogram (ECG) trace using convolutional neural network techniques is considered as a use case. In our study, we extend what was already discussed in [4] in order to apply dynamic hardware/software optimization mechanisms on more complex state-of-the-art platforms with multiple cores.

On the one hand, the community is working on the design of low-power devices capable of supporting processing that exploits artificial intelligence techniques, and these types of devices include accelerators, parallelization elements, and flexible power management. In the market or in the literature, there are devices based on Systems-on-Chip (SoCs) or Systems-on-Module (SoMs) [19–21], embedded GPUs [22], or FPGA-based accelerators [23]. Among the different solutions regarding hardware accelerators, Adaptive Compute Acceleration Platform (ACAP) produced by Xilinx [24] combines the potential of three types of engines: Arm cores (scalar engines), the programmable logic (Adaptable Engines), and the new vector processor cores (AI engines). This tightly coupled hybrid architecture allows more dramatic customization and performance increase than any one implementation alone. In Reference [25], the author presents a new type of architecture based on fixed-point arithmetic. The substantial difference compared to traditional solutions is the choice of the representation scale, which in this case occurs during compilation, while in the case of floating-point arithmetic, it is performed during execution. Other distinctive features of the proposed method are universal superscalar architecture and asymmetric structure of working registers.

On the other hand, there is a remarkable work of software development that tries to optimize firmware, middleware, or libraries that optimize the mathematical tools used to implement solutions that exploit AI, and some of them specifically created to run on specific platforms. Among these, we find: CUDNN [26] for GPUs ARM-NN for microcontrollers based on ARM Cortex processors [27] and CMSIS (Cortex Microcontroller Software Interface Standard). Other important techniques, which we do not deal with in our work, are pruning and quantization, widely discussed in the literature and which lead to significant improvements in terms of energy efficiency. In Reference [28], the authors discuss very thoroughly both the pruning and quantization compression techniques by testing them individually or simultaneously with different frameworks that support them. They analyze their strengths and weaknesses and provide practical guidance for compressing networks. Google Brain has proposed a new data format called Brain Float 16 (BF16), and it has gained wide adoption in AI accelerators from Google, Intel, Arm, and many others. The purpose is to minimize the prediction accuracy degradation due to a lowering of the data precision, with a consequent increase in throughput. The main difference comes from truncating the Floating Point 32 (FP32) mantissa field from 23 bits to 7 bits [29]. In Reference [30], the authors propose a hardware implementation method of MRBF-TS systems.

Starting from [4], we have made a step forward in the adaptivity approach, to consider all levels of complexity exposed by the current landscape, including parallelism in hardware and software, custom programming models, and multiple architecture knobs to be managed. Other works have previously focused on the adaptive management of complex processing platforms. Reference [31] presents a runtime approach to reconfigure core-to-task mapping and the degree of parallelism of the application when the available resources or the application workload changes, targeting shared-memory platforms. This work, however, focuses on fault tolerance and not on dynamically changing workload. Moreover, it is not tested on a dynamically manageable processing chip. Reference [32] presents an approach for workload self-organization on multi-cores. However, differently from our approach, computing tasks are seen as indivisible, and only their mapping to the different cores is changed. In Reference [33], tasks can be duplicated on multiple cores in order to have more freedom in pipeline organization; however, this is not tested on data-dependent workload and on cognitive computing.

Tables 1 and 2 resume main approaches in the literature about CNN workload partitioning and mapping and about dynamic voltage and frequency scaling in CNN-based designs. To the best of our knowledge, our work is complementary to both these kinds of strategies, being the first attempt to bring together dynamic remapping of CNN operators and consequent dynamic management of the hardware setup.

There are different ways of distributing the workload (convolutional layers) on multiple cores. One of the possible strategies is called the "kernel-level", and it involves the distribution of the kernels of a layer-*nth*, and therefore the output features calculation of the layer-*nth*, in a balanced way across all cores. Many state-of-the-art libraries adopt this solution, for example ARM-CL [34], tengine [35], or NCNN [36]. In Reference [37], the authors indicate two approaches for workload partitioning in CNNs: layer-level and kernel-level strategy. The "layer-level" strategy involves assigning each layer to a group of cores; the number of cores per group being variable; and since not all cores are used for the layer-*ith*, pipelining techniques being used in order to maximize the throughput. The authors present *Pipe-it*, a framework capable of estimating the computational load of all layers and, using a layer-level strategy, very efficiently distributes the workload on the available cores in heterogeneous multi-core platforms. The layer-level strategy is equivalent to the method that we use in this work; however, in Reference [37], it is not applied at runtime but only during design time design space exploration. In Reference [38], the authors exploit the layer-level strategy obtaining good results as already demonstrated with the Pipe-it framework; in addition, they show the importance of taking into account the cache resources per core. The distribution of the workload that takes this fact into account minimizes the inter-core feature-map data movement overhead, finally demonstrating how, in the use case they considered, there is a 73% performance improvement. Also in [38], strategies are chosen only at design time. The reference platform that we used, Orlando board, lends itself well to the optimizations proposed in [38], due to the presence of intra-core memories that can reduce the features transfer, but the complexity of a generic neural network may require a quantity of memory for features that often exceeds what is usually made available by data caches.

Another aspect dealt with in our work and already widely discussed in the literature concerns Dynamic Voltage and Frequency Scaling (DVFS). It has been discussed in the literature about how to use the DVFS in order to better manage the temperatures of the processing units. In Reference [39], they tackle the problem of overheating in different ways: by judiciously selecting tasks with different thermal characteristics as well as alternating the processor's active/sleep mode or by exploiting the DVFS potential offered by the platform. In References [40,41], the authors show how they dynamically choose the system output quality under temperature constraints; in the use cases they considered, the system output quality is highly dependent on the application of DVFS. In Reference [42], on the other hand, the authors deepen the aspect linked to the reliability, and the use of DVFS takes into account the minimization of the thermal cycles that stress the chip. In Reference [43] and in

Reference [44], the authors exploit DVFS techniques with the sole purpose of maximizing energy efficiency. In particular, the first ones refer to general-purpose CPU systems, while the second work focuses on the CPUs of commercial smartphones and how much the system frequency plays a fundamental role in them in terms of energy minimization.

Similarly to References [43,44], in our work, DVFS techniques are used to maximize energy efficiency; furthermore, techniques to respond to CNN-based workload variations by the use of task-mapping techniques at runtime on a multi-core platform are discussed. There are other works that combine CNN applications with DVFS techniques on ASICs, CPU, or GPU-based devices. In Reference [45], the authors show how, through the DVFS techniques and the choice of the precision of the Deep Neural Network (DNN), it is possible to reach a certain level of inference accuracy and power consumption under latency constraints. In Reference [46], the authors develop a principled approach and a data-driven analytical model to optimize the granularity of threads during CNN software synthesis. In Reference [47], the authors propose a CNN-based low-power facial recognition system, and the DVFS mechanism is the basis of their method to increase energy efficiency. In Reference [48], authors use a performance-power analytical model fitted on a parametrized implementation of a Deep Learning (DL) accelerator in a 28-nm FDSOI technology to explore a large design space and to obtain the Pareto points that maximize the effectiveness of DVFS in the sub-space of throughput and energy efficiency.

Table 1. Qualitative comparison with CNN workload partitioning and mapping studies.

Work/Framework	CNN Workload Partitioning and Mapping		Runtime
	Kernel-Level	Layer-Level	
ARM-CL [34]	✓		
tengine [35]	✓		
NCNN [36]	✓		
Pipe-it [37]	✓	✓	
[38]	✓	✓	
Our work	✓	✓	✓

Table 2. Qualitative comparison with dynamic voltage and frequency scaling, especially in CNN-based designs studies.

Work	DVFS Constraints		DVFS on CNN	Dynamic Partitioning and Mapping
	Temperature	Energy		
[39]	✓			
[40]	✓			
[41]	✓			
[42]	✓			
[43]		✓		
[44]		✓		
[45]		✓	✓	
[46]		✓	✓	
[47]		✓	✓	
[48]		✓	✓	
Our work		✓	✓	✓

Summarizing, as main novel contribution in this work we propose:

- We presented a hardware/software/firmware architecture template involving a remotely controlled wearable IoMT device that performs cognitive data processing.
- Its validation on a state-of-the-art data analysis based on a CNN as an example computational load.
- Evaluation of the effectiveness of dynamic optimization techniques on multi-core devices using anomaly classification from an ECG signal as a use case.

3. Reference Architecture

We refer to an overall network of the IoMT system composed of three levels, as depicted in Figure 1. At the lowest level, we find the sensor nodes that communicate via Bluetooth Low Energy (BLE) technology with the upper level. Inside each node is the ADAM component, which is sensitive to reconfiguration messages or changes in workload and is responsible for the reconfiguration of the device. The middle level includes multiple gateways, which play the role of intermediary between the sensors network and the cloud. The top level is cloud based, data are collected securely, and the healthcare provider has the ability to view or analyze the data. Through a web interface, it is possible to manage data but also remotely control the device, thus determining which operating mode should be enabled. Our work considers and describes only the sensory node component.

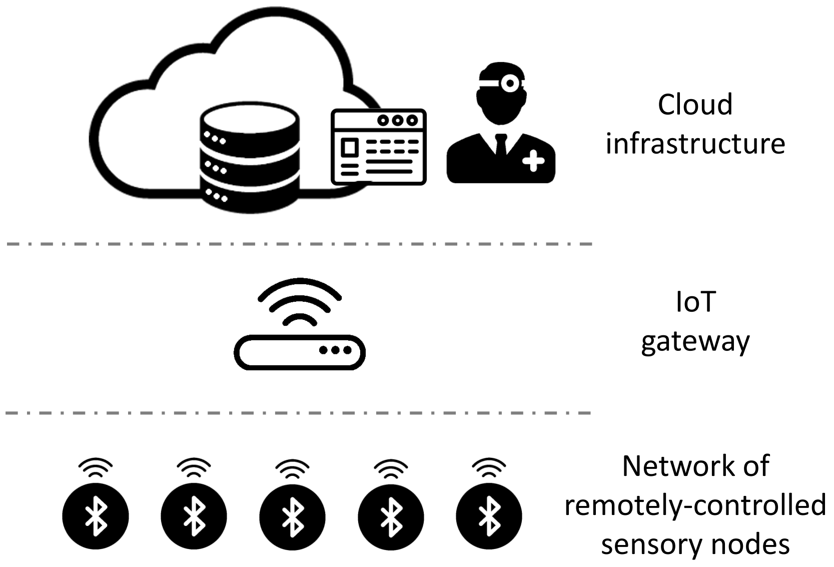

Figure 1. General overview of the proposed system.

3.1. Sensor Nodes

The structure of the sensory node is shown in Figure 2, and it too can be divided into layers which is described in this section.

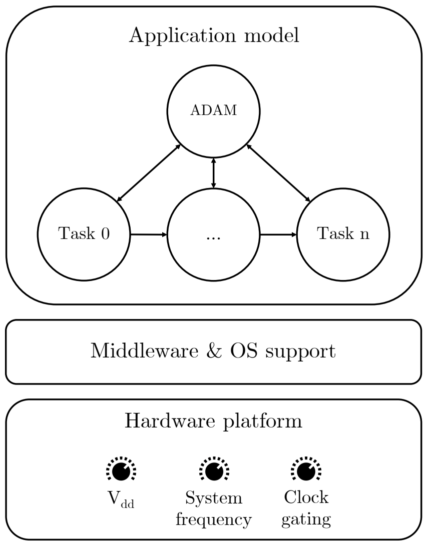

Figure 2. IoMT node architecture overview.

The lowest level consists of the hardware platform, usually, and it is based on microcontrollers or low-power devices. They take care of sampling different signals from the external environment and at the same time have the ability to analyze the data, and they also implement the communication system with the gateway. The node we considered in our work is a multi-core device called Orlando [20] and produced by STMicroelectronics, which raises the challenge compared to our previous work.

At the middle level, there is the middleware/operating system layer. The operating system allows for the easy management of scheduling and software threads. At the same level of the operating system, there are the middleware/firmware components that offer a set of primitives to better manage the hardware of the platform (e.g., power mode, performance counting, and operating frequency) and the APIs that allow continuous management and control of the status of the peripheral (energy and power status, remaining battery life).

At the top level, there is the software application; at this level, all tasks are executed according to the application model, and a precise characterization of the execution based on the process network makes it easier to dynamically manage the system configuration. At the same level of the application model, we added the component called ADAM, which is triggered by reconfiguration messages from the healthcare worker or from internal events such as battery level or workload variation. ADAM works with the application model to be able to configure the process network and communicates with the middleware/operating system layer to exploit the reconfiguration tools, such as activation/deactivation of tasks and restructuring of the inter-task connectivity, or the management of hardware parameters such as supply voltage or frequency scaling. The effectiveness of these components has already been demonstrated previously in the single-core case [4].

In this work, we extend ADAM to target multi-core advanced IoT platforms capable of executing more complex applications, thus enhancing near-sensor processing possibilities. This exposes additional challenges when it comes to the dynamic runtime management of the platform:

- Modern multi-processor IoT nodes, especially the plethora of prototype solutions currently designed by the community to support AI-related workloads and optimized for low power, have limited OS support. To try our approach on Orlando, we had to implement adaptivity on a bare-metal system, exploiting the platform-specific set of APIs to manage the application model, the process network, and the related operating modes.
- The availability of more cores requires, when switching operating mode, the adaptation of the parallelism level exploitable within the application structure. The workload imposed by a given mode must be optimally partitioned between the available processing elements, using splitting/merging and pipeline methods.

In the following sections, a detailed description of each level of the node is provided.

3.1.1. Hardware Platform

The hardware architecture of the Orlando chip is represented in Figure 3.

The chip is very flexible. It integrates:

- An on-chip reconfigurable data-transfer fabric to improve data reuse and reduce on-chip and off-chip memory traffic.
- An ARM-based host subsystem with peripherals.
- A range of high-speed IO interfaces for imaging and other types of sensors.
- A chip-to-chip multi-link to pair multiple devices together.
- A power-efficient array of Digital Signal Processors (DSPs) to support complete real-world computer vision applications. Eight DSP clusters are present, each composed by 2 DSPs, 4-way 16 kB instruction caches, 64 kB local RAMs, and a 64 kB shared RAM.
- 4 $\times$ 1 MB SRAM banks.

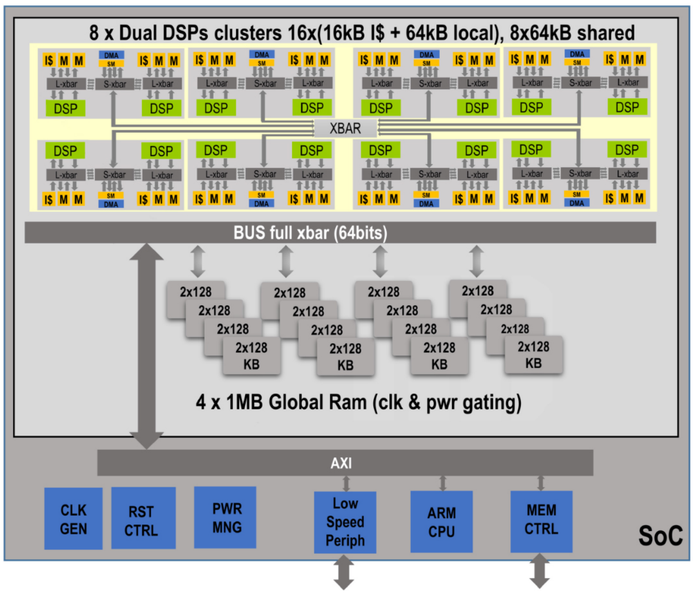
8 x Dual DSPs clusters 16x(16kB I$ + 64kB local), 8x64kB shared
DSP
XBAR
BUS full xbar (64bits)
2x128 KB
4 x 1MB Global Ram (clk & pwr gating)
AXI
CLK GEN
RST CTRL
PWR MNG
Low Speed Periph
ARM CPU
MEM CTRL
SoC

Figure 3. SoC top-level block diagram.

As far as our work is concerned, the attention is focused on DSP cores. As architecture knobs available for dynamically changing the platform setup, we consider the activation and deactivation of processing elements (DSP cores) and changes to system frequency and supply voltage. Still being in the prototyping phase, the Orlando board has not been subjected to normal post-production testing. Therefore, under the guidance of the manufacturer, we made an empirical investigation to characterize the device. Several experimental tests were made to characterize the relationship between the supply voltage of the chip V_{dd} and the system frequency, and they were used to obtain the results that are presented in Section 5. Therefore, a lookup table was obtained that can be consulted at any time by the ADAptive runtime Manager, in order to correctly set the minimum power supply voltage necessary for the required system frequency. Figure 4 shows the output of the characterization process, and for illustration purposes, the curve that approximates the trend of the supply voltage in relation to the system frequency is also shown.

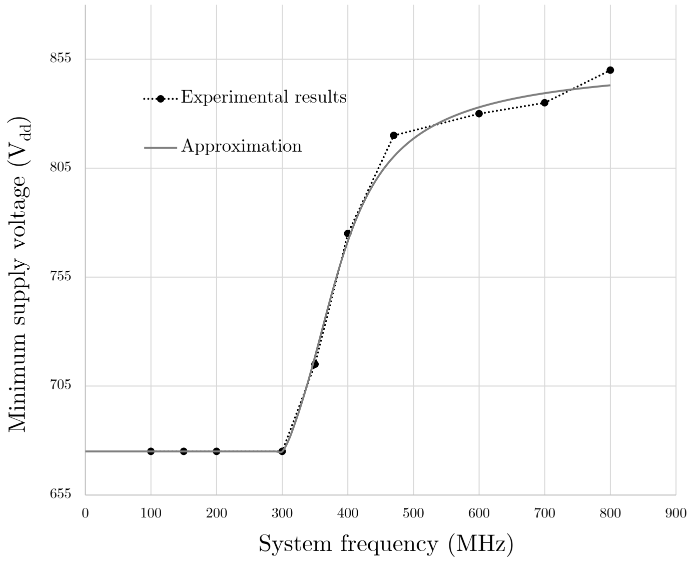

Figure 4. V_{dd} measurement and approximation for each system frequency.

3.1.2. Middleware

With the selected hardware platform, it is not currently possible to exploit the support of a real-time OS. We need to consider tasks in the same chain to be prospectively executed by independent hardware cores communicating through a set/hierarchy of shared memories. The overall management is implemented using platform-specific low-level primitives provided by STMicroelectronics, named RPC APIs. We have used them to manage communication and synchronization between the DSPs in the platform and to manage the operating state of the processing elements, setting to sleep mode those that are stalled on input (no input data from FIFO), or output channels (the output FIFO is full), or that are not assigned with a task. We present the RPC APIs that we have used in Table 3, concerning functions to turn on and off DSPs, and Table 4, concerning functions adopted for synchronizing DSPs while accessing FIFOs in a mutually exclusive way.

Table 3. Core activation and deactivation functions.

Function Name	Description
`sleep()`	It is invoked by the DSP that intends to go to sleep; once invoked, the DSP is placed in a low-power state and remains in this state until it receives a wake-up signal.
`wakeup(...)`	Once this function is invoked, a wake-up signal is sent to the specified core.

When idle, a DSP executes a `rpc_serve()` function. In this way, the core waits from an activation message from other cores, and it is set into a sleep state (through `sleep()` function) until a request is received, activating it again (through `wakeup()` function). The requests are stored in a queue, one per DSP, stored in the main shared memory (4×1 MB SRAM banks) and served with a round-robin priority scheme.

Table 4. Synchronization functions, which are mainly used to read/write on the same FIFO in a mutually exclusive way.

Function Name	Description
`mutex_init(...)`	Initialization of the mutual exclusion.
`mutex_lock(...)`	Request mutual exclusion.
`mutex_unlock(...)`	Release mutual exclusion.

In Table 5, we show the functions usable to send activation messages and to check the execution of the assigned tasks on a remote DSP.

Table 5. Call functions, used by the ADAM system to manage the execution of tasks on the cores.

Function Name	Description
`rpc_call(...)`	Execute a function passed as an input on a remote processor. From the inputs, it is possible to choose whether `rpc_call` is blocking or non-blocking.
`rpc_check(...)`	Check the execution status of a certain function call on a specific core (non-blocking).
`rpc_wait(...)`	Wait for the conclusion of a function on a specific core (blocking).

3.1.3. Application Model

In the following section, we describe the structure of the application model used in our application, and the model is based on a network of processes (compliant with the dataflow process network model of computation [49]). In order to simplify the communication and to avoid data loss in the case in which the system is not able to momentarily support the entire workload, processes exchange information via FIFOs, using *read* and *write* blocking primitives. One FIFO is initialized for each task that expects to receive input data. FIFOs are stored into the shared 4×1 MB SRAM banks and are split into data and management parts. The data part is spread within the SRAM: data chunks are stored dynamically by cores running production tasks, while they are freed by cores running consumption tasks. The management part contains a FIFO of pointers referencing the different dynamically allocated chunks of data (tokens), together with the counters for controlling the circular buffer and the FIFO status. Following the topology of the operating mode under execution, the outputs of the source tasks are coherently connected to the input FIFOs of the target tasks, and the connection is performed simply by providing a common FIFO reference. In this way, the connections among tasks can be easily modified by changing the FIFO reference provided to the same tasks. The processes may be potentially executed in parallel, in the case of available processing resources, in order to improve performance using a software pipeline. In particular, for each sensed variable to be monitored, we build a chain of tasks that operate on the sensed data (Figure 5).

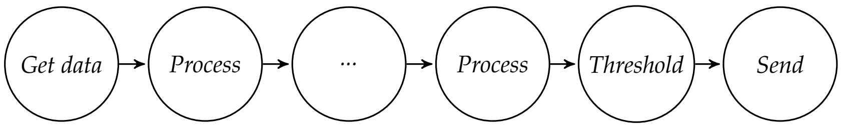

Figure 5. A simple chain of tasks: circles represent the tasks while the arrows represent the FIFOs that put them in communication.

Each sensor node is associated with a specific network of processes, which is dynamically modified according to the selected operating mode, enabling or disabling certain tasks.

We have identified four types of tasks that make up all possible network configurations:

- *Get data* task: deals with the sampling of the signal acquired by the sensor.
- *Process* task: there may be multiple processing tasks that allow multiple processing levels to be enabled. The choice of a different level of processing affects the required transmission bandwidth, the detail of the information that can be obtained from the node, and the energy consumption.
- *Threshold* task: allows filtering the information transmitted to the cloud, further reducing the energy consumption related to the communication with the server. In fact, once the signal has been processed, the system evaluates whether it is useful to send the results or not.
- *Send* task: takes care of packaging and sending data to the cloud.

Considering the selected process network model, the activation/deactivation of tasks or entire chains corresponding to sensors can be implemented by:

- enabling/stopping the periodic execution of the involved task;
- reconfiguring the FIFOs to reshape the process chain accordingly.

Therefore, it is possible to identify different configurations, defining operating modes characterized by different levels of processing, information detail, and communication bandwidth.

3.1.4. Adaptive Runtime Manager

An independent continuously running task has been introduced, and it takes care of the runtime reconfiguration of the platform, and the reconfiguration is performed by the ADAM component. The latter can be invoked by external or internal events; in particular, it monitors:

- receiving reconfiguration messages from the cloud.
- the workload. For example, a variation of the detection rate of an event in the acquired signal can lead to a more frequent invocation of a processing task. Consequently, there may be a need to reconfigure the device in case the real-time constraints are no longer respected.
- other system variables, such as the remaining battery charge.

If from the continuous monitoring ADAM detects one of the events listed above, it has the ability to act on the platform in different ways:

- act on individual tasks or on the entire task chain by enabling or disabling the constituent elements;

- decide when to enable the sleep mode state of peripherals, computing units, or the entire device;
- act on the system frequency and supply voltage;
- reroute the FIFOs data flow according to the selected operating mode;
- efficiently split the workload into available resources.

An example of reconfiguration can be deduced from Figure 6; in this case, it is possible to switch from an operating mode that involves the use of all the tasks and therefore enables in-place processing, to an operating mode that only sends raw data to the server.

In our previous work [4], it was shown in detail how the entire system and application model is managed in cases of operating mode or workload changing. As we see in more detail in the next sections, in this work, we further decline the model to consider CNN operators as an independent task. In this way, we can direct sub-parts of the network to different processing elements. We can exploit merging, by mapping tasks on the same core, or splitting, by partitioning tasks into parallel sub-tasks that can be mapped independently.

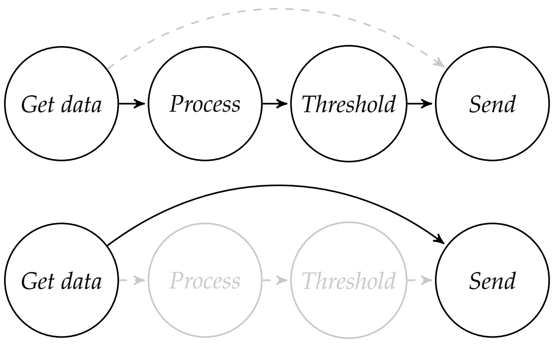

Figure 6. Two possible configurations of a generic system.

4. Adaptivity in Advanced Multi-Core Hardware Platforms

Modern data analysis algorithms, such as those relying on neural networks and deep learning, are characterized by critical demands in terms of computing power. They are composed of multiple layers, and each layer is usually processing *tensors*. Thus, such algorithms intrinsically expose additional parallelism to be exploited when more processing elements are available.

To comply with this complexity, we extended the application model described in Section 3.1.3, enabling the representation of lower-level building components of the *Process* tasks. For example, individual layers composing a *Process* task representing a CNN can be represented themselves as single tasks and communicate with each other through FIFOs. In this way, tasks can be independently mapped to physical cores, and throughput can be improved.

Figure 7 shows an example where each layer in a CNN is mapped to an independent core. The lower part of the figure shows a legend explaining how layers, cores, and FIFOs are represented. In this case, the throughput is obviously determined by the layer with a longer execution time, which limits the pipeline rate.

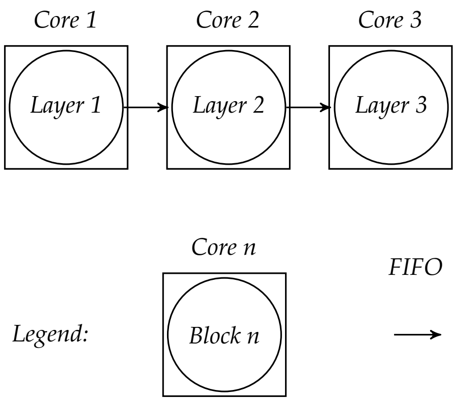

Figure 7. One core for each convolutional layer.

4.1. ADAM for Multi-Cores

When multi-core platforms are enabled, ADAM can act to change the mapping and partitioning of the software tasks a d to create a software pipeline configuration that optimally fits with the required workload. The objective is to balance pipeline stages and to set up an optimal frequency-voltage operating point for the platform.

We have defined a workload-partitioning mechanism, called *splitting*, that enables one to divide a single task to be executed in parallel on several cores, as depicted in Figure 8, to reduce the duration of a limiting pipeline stage and to improve the overall throughput. In this case, ADAM, depending on a selected optimization policy, is in charge to activate, when needed, a set of supporting cores sharing the initial workload, called *helpers*. The lower part of the figure shows a legend explaining how blocks, cores, and FIFOs are represented.

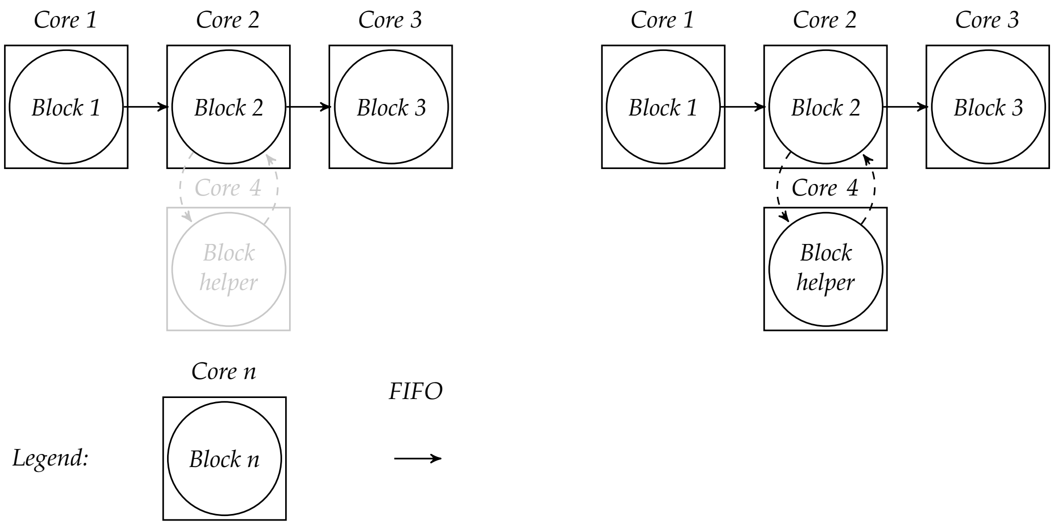

Figure 8. Subdivision of the workload given by the *blocks* into several cores.

At the moment, we have tested the splitting technique on Orlando, referring to the typical structure of neural network layers as a specific application use case. The splitting of a layer with one or more *helpers* is operated asking the *helpers* to compute part (half in case of one single *helper*) of the layer's output features.

4.2. Splitting Policies

ADAM can implement different policies that may be used to combine splitting and frequency/voltage scaling to dynamically adapt to changing workloads.

In this work, we have implemented two policies:

- the first policy, which we call ADAM-FF (Frequency First), which tries to minimize the working system frequency as the main objective;
- the second policy, that we call ADAM-IF (Idle First), which is more indicated for systems that have less reactive frequency management, which tries to set as many processing elements as possible in sleep mode.

Both policies envision the system to be set, at the start-up, in a mapping configuration, called hereafter *baseline* setup, that balances pipeline stages as much as possible. To this aim, we merge tasks (layers in our CNN-related experiments) in *blocks*, until we obtain groups that are as similar as possible to each other in terms of execution time. The merging of the operators in one single block is performed at design time. The merged block is represented as a single process network; thus, there is no performance degradation due to the scheduling of multiple nodes on the same processing element. Figure 9 shows an example of a balanced pipeline: the first and second layers are processed by core 1; the third layer is processed by core 2; and core 3 instead processes layers 4 and 5, merged together in one single *block*. As can be seen from Figure 9 and for all subsequent figures, the first layer, belonging to the first block, is not clearly visible on core 1; due to the computational complexity, the execution time is extremely reduced compared to the other layers. Again, Figure 9 and the figures representing the timing of the pipeline highlight the first stage of the pipeline of each core. What each core does after the first pipeline stage is nothing more than the same work repeated for all the other stages. In Figure 9 (and similar ones), the pipeline stages following the first one are shown in gray, in order to make the graph more readable, while the empty spaces indicate the sleep state of the cores.

We envision the definition of the baseline setup to be identified offline, at design time, by manual profiling or using adequate existing system-level design tools, such as those described in [50,51].

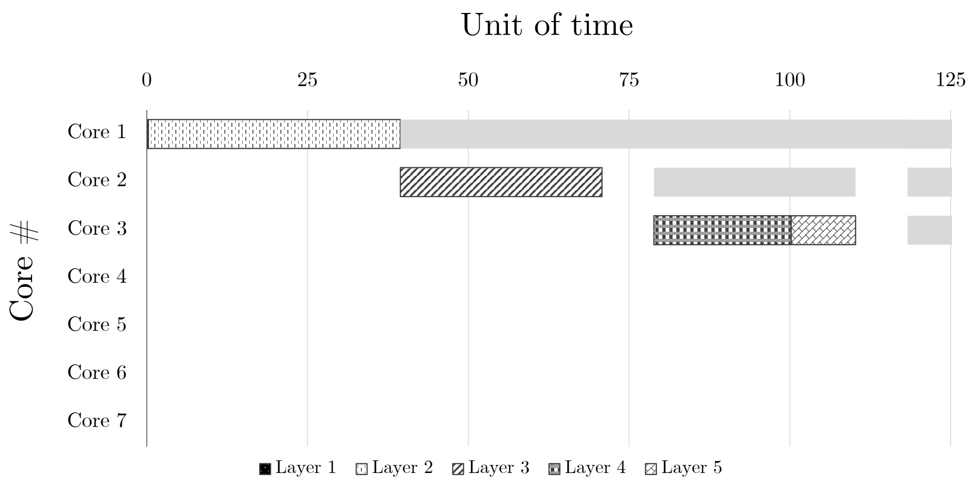

Figure 9. Example of balanced pipeline (gray shadows show the potential execution of successive pipelined computations of the same application).

At this point, thanks to the splitting mechanism, it is possible to divide the workload of each task (being an independently mapped layer or a *block*) between several cores, activating one or more *helpers*. Figure 10 shows an example of how this is used to reduce the execution time of the three stages of the pipeline (the first stage is shared by three cores, and the second and third stages by two cores each). By reducing the limiting length of the stages, the pipeline can switch at a higher rate; thus, the throughput is improved.

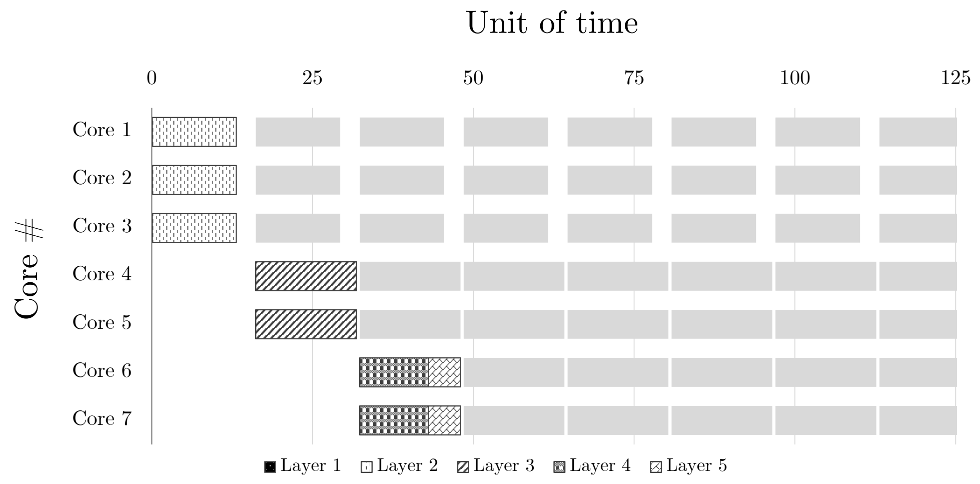

Figure 10. Example of redistribution of the workload over multiple cores (gray shadows show the potential execution of successive pipelined computations of the same application).

ADAM-FF policy is outlined in Algorithm 1. In this policy, the system starts from the baseline setup and, independently from the actual workload to be supported, applies splitting iteratively to throughput-limiting tasks until all the available processing elements are used as *helpers*. After this phase, ADAM enters in a routine, that may be triggered by a timer (as in the pseudo-code) or by other external events. It monitors the workload and increases the system frequency (adapting the voltage accordingly) when a higher performance level is required to support real-time constraints or reduces it when constraints are more relaxed.

Algorithm 1: ADAM-FF policy algorithm.

```
setup baseline;
while there are cores available do
    locate the bottleneck;
    apply the splitting mechanism on the bottleneck;
end
while true do
    wait trigger from periodic timer;
    check workload;
    if real-time constraints are not respected then
        Increase system frequency;
        Increase supply voltage;
    else
        Decrease system frequency;
        Decrease supply voltage;
    end
end
```

ADAM-IF is outlined in Algorithm 2. Again, the system starts from the baseline. It sets the system frequency to be capable of respecting worst-case real-time constraints, corresponding to the highest workload, when the maximum splitting of the tasks is applied. At this point, ADAM keeps monitoring the workload and adds or removes *helpers* to meet the needs posed by real-time at any monitoring step. When constraints are relaxed, the system uses the minimum number of cores, leaving the others to wait when to be activated as *helpers*, moving them from idle to active state only when more performance is needed.

The two policies are tested on Orlando hardware platform and considering a reference CNN use case in Section 5.

Algorithm 2: ADAM-IF policy algorithm

```
setup baseline;
set system frequency and supply voltage in worst-case;
while true do
    wait trigger from periodic timer;
    check workload;
    if real-time constraints are not respected then
        enable helper cores;
    else
        disable helper cores;
    end
end
```

4.3. Splitting Model on Orlando

As already described in Section 3.1.1 and as visible in Figure 3, Orlando chip mounts 8 clusters containing: 2 DSPs, 2 instruction cache memories (each of 16 kB) 2 local 64 kB memories, and a memory of 64 kB shared between the two DSPs. In general, it is possible to exploit these local cluster memories in order to optimize access to memory using the layer-level strategy mentioned in [37]. This optimization was not possible in this case due to the size of data (CNN weights) which forces the adoption of the 4 $\times$ 1 MB SRAM banks.

In order to better explain the splitting method adopted on Orlando, the function `rpc_call()` seen in the Table 5 is better described. Table 6 describes the input parameters associated with the aforementioned function.

Table 6. `rpc_call(...)` function arguments.

Input Parameter	Description
`int flags`	The first parameter specifies how the function is executed. Between the two mainpossibilities we find: - RPC_SYNC, request will be blocking until completion. - RPC_ASYNC, request will be executing asynchronously.
`void *func`	It's the pointer to the function to be executed on the specified core.
`int core_caller`	Indication of the core that executed the `rpc_call` function.
`int core_helper`	Core on which the pointed function will be executed.
`int n_parameters`	Number of parameters that the pointed function takes as input.
`int *ret`	Pointer to the return variable of the specified function.
`varargs`	Input parameters to the previously specified function.

Each *block* is associated with one or more `rpc_call()` functions, generally one for each layer of the neural network. For example, in `*func` a convolutional function pointer is specified. In `varargs` the structure containing all the data pointers useful for the convolution is provided. If the n-th core is in a sleep state, once an RPC function is executed, by assigning the value n to the `core_helper` variable, the n-th core is awakened from the sleep state and performs the function specified in `*func`.

The steps to enable core *helper* activation and workload splitting are described in the following list:

1. The ADAM system constantly calculates how many *helper* cores must be enabled for each i-th *block*;
2. The core dedicated to the i-th *block* is constantly informed by the ADAM system on how many *helper* cores are assigned to its *block*. Within this core, as many `rpc_call()` functions are performed as there are *helper* cores specified by ADAM on the i-th *block*;
3. Furthermore, the core dedicated to the i-th *block* takes care of passing data in a coherent way to the *helper* core. For example, if two *helper* cores are assigned for the i-th *block*, the convolutional kernel pointer of each `rpc_call()` function that awakens a core *helper* is changed; in particular, a third of the kernels is associated with each core *helper*

(so that each core *helper* calculates one-third of the output features), and the remaining third is used within the calling core.

5. Results

In this section, we show the results obtained with the proposed extension of ADAM for multi-core platforms, and in particular, considering the Orlando board from STMicroelectronics [20]. In Section 5.1, we describe more in detail the adopted use case, modeled according to the application model introduced in Section 3.1.3. This use case was evaluated on the target hardware platform in two different conditions, to show the potentials of the proposed approach: single channel, considering one sensor connected to the sensing node and whose results are shown in Section 5.3, and multi-channel, where multiple sensors are connected to the sensing node and whose results are shown in Section 5.4.

5.1. Use Case

In order to test the effectiveness of our system, we propose an architecture that allows constant monitoring of a patient's ECG signal, capable of detecting cardiac anomalies, by exploiting artificial intelligence techniques. Connected to the ADC of the reference platform, Orlando, there is the AD8232 sensor module developed by Analog Devices (https://www.analog.com/en/products/ad8232.html, accessed on 3 August 2021). For the considered application model, several operating modes have been originally envisioned [4]. In this work, besides targeting an advanced multi-core hardware platform, for which ADAM was extended, we focus on the CNN processing operating mode.

As shown in Figure 11, We identified five different task types with respect to the selected use case:

- *Get data*: takes care of acquiring the signal from the AD8232 module;
- *Peak*: analyzes the ECG signal to detect peaks and calculate the heart rate, and the amount of information sent to the server is greatly reduced;
- *CNN*: using cognitive analysis based on concurrent neural networks, cardiac abnormalities are detected in the ECG tracing. Signal frames around the peaks detected by the previous processing task are considered. Also in this case, the amount of information sent to the server is greatly reduced;
- *Threshold*: decides whether or not the results from the enabled processing levels should be sent to the cloud; for example, if the heart rate is within a normal range there is no need to transmit the data;
- *Send*: packages and sends the data to the server.

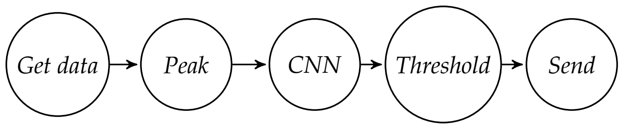

Figure 11. ECG application model for the CNN processing operating mode [4].

The *Process data* tasks, according to the application model presented in Section 3.1.3, are then two: *Peak* and *CNN*. The peak detection algorithm on the ECG signal is based on a derivative filter, which was chosen for its simplicity of implementation and the low computational capacity it requires. The adopted CNN recognizes anomalies on the ECG signal with an accuracy of 88%, and the inference process leads to three different output classes: Normal Sinus Rhythm, Atrial Fibrillation, or Other Rhythm [52]. The neural network consists of 13 layers, each involving a one-dimensional convolution, a batch normalization, a ReLU, and a dropout stage. Only three layers have also a max pooling stage with a pooling size of two between ReLU and dropout. The overall size of the data transferred to the cloud is 6 bytes (1 heartbeat datum represented with 8 bit, 1 classification label data represented with 8 bit, and 1 timestamp represented with 32 bit). This network requires a huge computational power, thus deeply stressing the capabilities of the adopted hardware platform. It constitutes a proper test bench for the proposed approach, which adapts the exploited level of parallelism to changing workload and operating conditions.

5.2. Experimental Setup

We have executed the CNN described in Section 5.1 on the Orlando multi-core platform, adopting ADAM for the dynamic adaptation of the processing, as discussed in Section 4. The baseline has been chosen manually at design time and is shown in Figure 12. Two layers are mapped on core 1; core 2, 3, and 4 execute one layer each, while in core 5, eight layers have been merged and mapped together.

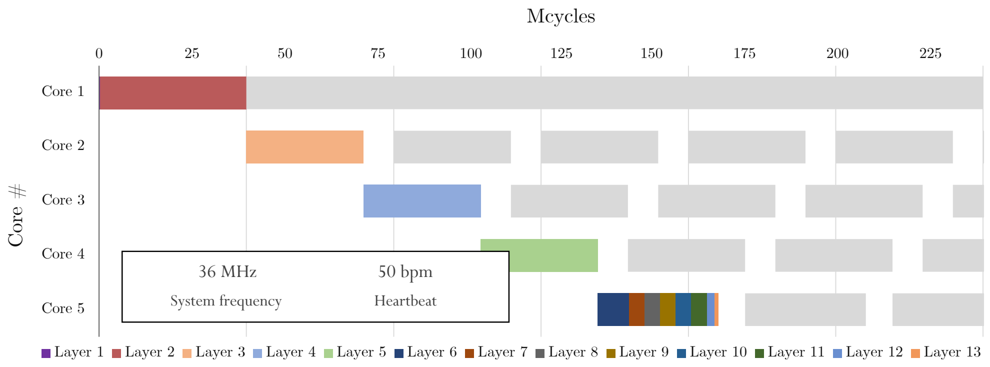

Figure 12. The baseline setup for the selected use case on Orlando.

The Orlando prototyping board provides pins to measure all the device current supply, and therefore a digital oscilloscope and a hall effect probe were used to evaluate power consumption corresponding to different workload conditions. We forced the system to sustain three different workloads in order to show how the system reacts and dynamically adapts itself to this variation. For this purpose, we adopted three workload conditions by fixing the average heart rate, respectively, to 50 bpm, 100 bpm and 200 bpm. Such values are chosen to represent the low, moderate, and high cardiac activity of a healthy individual. We considered 200 bpm as the overall maximum workload to be supported by the system. For testing purposes, dummy data were adopted in order to activate the CNN, evaluate the execution times, and measure the related power consumption.

To assess the benefits of the proposed splitting technique and of its combination with dynamic voltage and frequency scaling, we compared ADAM solutions with two more static policies, namely Fixed Topology (FT) and Static System Frequency (SSF). Overall, four policies were then considered in the reported experiments:

- FT: splitting support is not available, and the application is split according to the baseline setup, and, to meet real-time constraints for the maximum heart rate (200 bpm), a starting system frequency is also selected. The resulting mapping is kept equal during execution, while the frequency can be tuned to optimize consumption.
- SSF: no frequency scaling neither splitting support are available. The baseline setup is used for splitting application, and the system frequency is then selected in order to meet real-time constraints for the given maximum heart rate (200 bpm). The resulting mapping and frequency are kept equal during execution.
- ADAM-FF: the proposed ADAM approach for runtime adaptation is enabled, and the frequency-first policy is considered, with the main goal of minimizing the system operating frequency while meeting real-time constraints.
- ADAM-IF: the proposed ADAM approach for runtime adaptation is enabled, and the idle-first policy is considered, with the main goal of minimizing the number of idle cores while meeting real-time constraints.

Please consider that the supply voltage V_{dd} is not directly considered by the proposed approach, rather it is set according to the adopted frequency, as resulting from the preliminary study shown in Section 3.1.1 which led to the frequency-voltage pairing depicted in Figure 4.

5.3. Single Channel

In ADAM-FF, splitting is used at the start-up to balance the pipeline as much as possible using all cores. At 50 bpm, ADAM-FF minimizes the system frequency, and to comply with real-time constraints in this condition, it is settled around 9 MHz (frequency numbers are rounded to consider that available precision in clock generation is 1 MHz). When the workload increases to 100 bpm, ADAM-FF compensates by increasing the frequency, doubling it to around 18 MHz. The same happens when moving to 200 bpm, requiring a frequency increase to 36 MHz.

Thus, 36 MHz corresponds to the frequency required to support the worst-case workload with complete splitting. When using ADAM-IF, this value is set for all the workload levels. At 50 bpm, the system uses the baseline mapping setup represented in Figure 12. Five out of sixteen processors (DSPs) are in active mode while the others are in idle state. When passing to 100 bpm, six *helpers* are activated, to create the pipeline configuration represented in Figure 13. Obviously, for 200 bpm, all cores are activated.

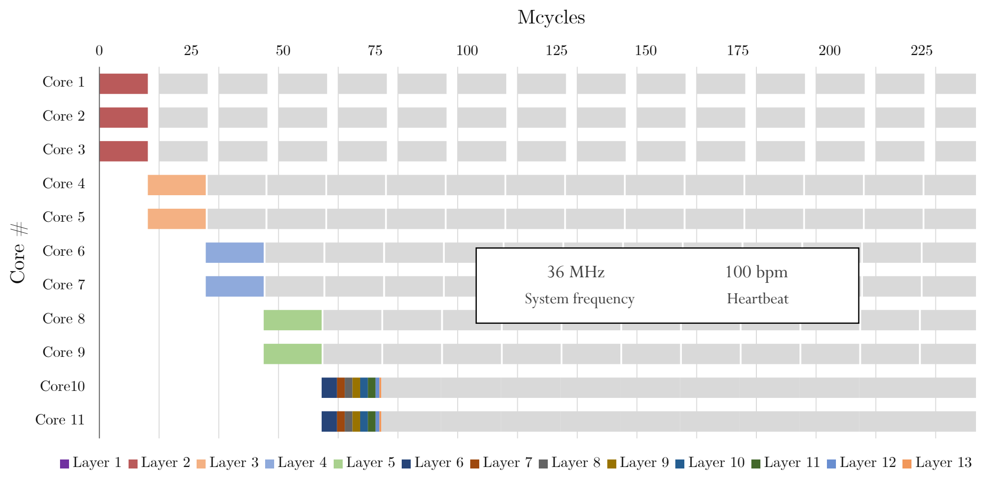

Figure 13. Dataflow on cores with a balanced pipeline and medium workload (ADAM-IF with maximum 100 bpm).

In FT, frequency is the only knob usable to comply with varying workloads. Five cores are always used, while frequency is set to 33 MHz for 50 bpm, 66 MHz for 100 bpm, and 132 MHz for the worst case.

Finally, in SSF, the system uses five cores and is always clocked to 132 MHz.

The results for single-channel power consumption are presented in Figure 14. For 50 bpm, ADAM-IF consumes slightly more than FT. In fact, the two policies lead, for a different system frequency, which in the ADAM-IF case is higher, to the use of the same number of cores. As may be noticed, in Orlando, in this case, ADAM-FF is the best solution for every workload condition. It saves around 5% on average when compared with ADAM-IF and up to 15% with respect to more static policies (SSF, FT). This is basically due to the amount of power that depends on the system frequency: by dividing the workload into several cores, it is possible to achieve a significant system frequency reduction. In devices where some components cannot be completely shut down, this method can be very effective for energy saving. When targeting devices that do not provide effective support for rapid and low-overhead frequency adaptation, changing the system frequency at runtime can be impossible. In this case, ADAM-IF can still be a good policy for saving power consumption. It is possible to save up to 15% power with respect to SSF, by changing the partitioning and using splitting instead of frequency to improve performance.

Savings appear to be overall limited in the proposed experimental results. This is mainly due to the fact that for the considered use case Orlando works in a frequency region that is lower than 300 MHz. In this region, V_{dd} is always set to 675 mV, as depicted in Figure 4. Lower frequencies do not enable the use of lower voltages; thus, using splitting instead of increasing clock speed does not provide maximum benefits.

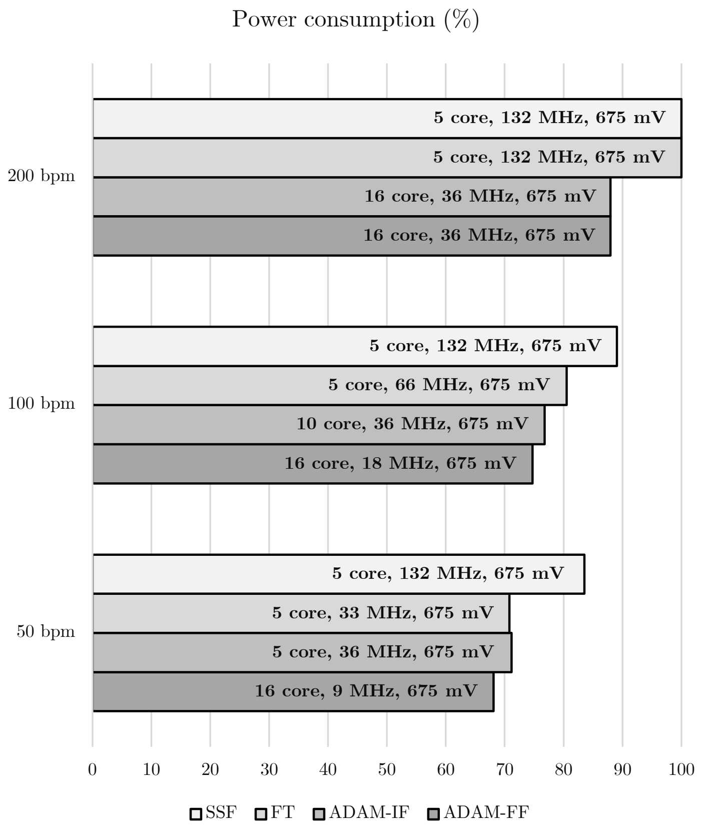

Figure 14. Comparison of power consumption considering different adaptation policies (with or without ADAM) and different workloads.

5.4. Multi-Channel

In order to explore the full potential of the proposed approach, we compare the proposed ADAM adaptation policies on a prospective benchmark that requires heavier workloads to be supported. For example, we can envision using Orlando to implement an embedded microserver analyzing multiple ECG channels (e.g., a single data collector in a hospital room, performing in-place analysis for all the patients). For this purpose, six different signals coming from 6 AD8232, each monitoring a different patient, are considered. These signals are computed by the hardware platform concurrently, stressing the available 16 cores and forcing the system to move to frequencies that imply a modification of the supply voltage. In this case, always considering the baseline setup depicted in Figure 12 as initial splitting for the application, the starting frequency necessary to meet real-time constraints with the maximum workload (200 bpm) is 789 MHz, which requires increasing V_{dd} to 843 mV with respect to the 675 mV of the single-channel experiments. This baseline setup and 789 MHz operating frequency are, as occurred for single-channel, the configuration adopted with the SSF policy for all the tests.

Figure 15 shows the results for the multi-channel experiments. ADAM-FF resulting frequencies are now equal to 52 MHz for 50 bpm, 108 MHz for 100 bpm, and 216 MHz for 50 bpm, all requiring the minimum supply voltage (675 mV) and employing the entire 16 available cores. This is, again, the best policy for power consumption, reaching a saving which is more than 80% with respect to SSF in the 50 bpm case.

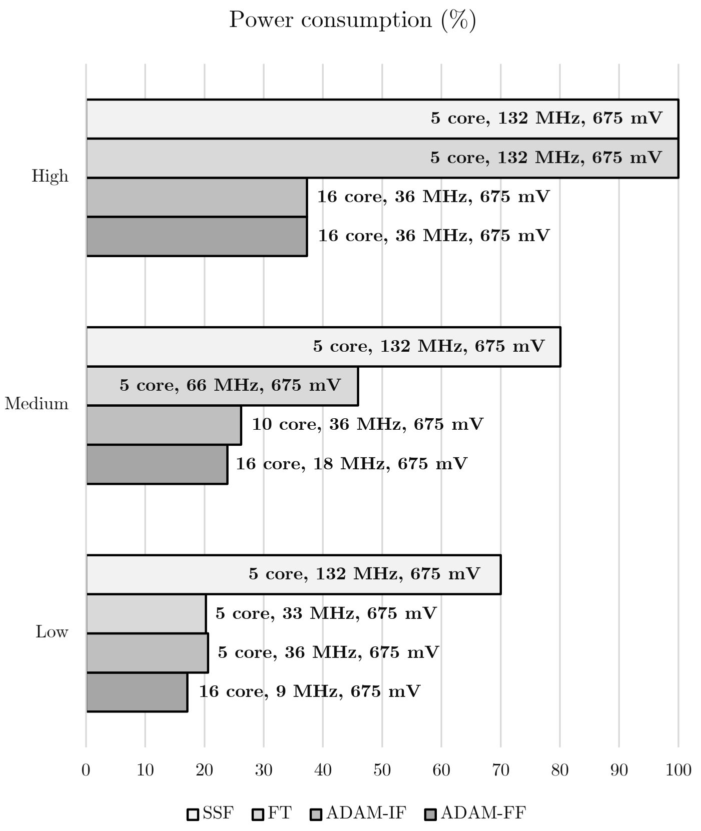

Figure 15. Comparison of power consumption in different ADAM configurations with high workloads (processing of six ECG streams).

ADAM-IF, instead of minimizing frequency, aims at minimizing active cores. For this reason, it employs only 5 cores for 50 bpm, 10 cores for 100 bpm, and all 16 cores for 200 bpm, with an operating frequency equal to 216 MHz, requiring the minimum supply voltage (675 mV). The overall power saving of the ADAM-IF policy with respect to SSF is slightly lower than ADAM-FF, but still consistent and close to 80% in the best case. By always using the same number of cores, five, the FT policy is instead adopting a frequency which is 198 MHz, 395 MHz and 789 MHz, respectively, for 50 bpm, 100 bpm and 200 bpm, and requiring in turn a V_{dd} equal to 675 mV, 767 mV and 843 mV. This, again, leads to a saving of the FT policy with respect to the ADAM-IF one for 50 bpm due to the overhead of running the same ADAM task. In any case, with higher workloads (100 bpm and 200 bpm) ADAM-IF saves always more than 50% power with respect to FT.

Increasing the overall computing load within the hardware platform, for all the considered workloads (50 bpm, 100 bpm and 200 bpm), both ADAM-based policies provide much more significant savings with respect to non-splitting policies. This is true especially considering the highest workload cases: up to 60% power reduction is achieved when higher performance is required.

6. Conclusions

We proposed a template of a hardware/software/firmware architecture concerning a remote-controlled IoMT wearable device able to recognize cardiac anomalies by means of deep learning algorithms. We have introduced a component called ADAM, able to manage the hardware/software configuration of the device in order to better manage the energy efficiency at runtime.

The workload distribution, through adaptive pipeline management techniques and process splitting, allows for an energy saving of at least 15% with respect to a static system. For the same system that manages six ECG channels at a time, the energy saving rises to 60% compared to a static system. In both cases, the reconfiguration policy called ADAM-

FF (Frequency First), which exploits all the available cores and acts exclusively on the system frequency, is the one giving the best performance. The obtained results confirm the potential of data-dependent runtime architecture management.

Author Contributions: Conceptualization, M.A.S., P.M., C.S. and L.R.; Data curation, M.A.S.; Formal analysis, M.A.S., P.M. and C.S.; Funding acquisition, L.R.; Methodology, M.A.S., P.M. and C.S.; Software, M.A.S.; Writing—original draft, M.A.S. and P.M.; Writing—review and editing, M.A.S., P.M. and C.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by EU Commission for funding ALOHA Project (H2020) under Grant Agreement n. 780788. This work was also supported by the joint research and development project F/050395/01-02/X32, INSIEME: Intelligent Systems for Integrated Health Management, CUP:B28I17000060008, funded by Italian MISE (Ministero dello Sviluppo Economico), D.M. 1 June 2016, Axis 1, action 1.1.3. of the National Operative Program «Imprese e Competitività» 2014–2020 FESR, Horizon 2020–PON I&C 2014-20.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Research, A.M. Internet of Things (IoT) Healthcare Market-Global Opportunity Analysis and Industry Forecast, 2014–2020. 2016. Available online: https://www.alliedmarketresearch.com/iot-healthcare-market (accessed on 3 August 2021).
2. Maskeliūnas, R.; Damaševičius, R.; Segal, S. A Review of Internet of Things Technologies for Ambient Assisted Living Environments. *Future Internet* **2019**, *11*, 259. [CrossRef]
3. Zouai, M.; Kazar, O.; Haba, B.; Saouli, H. Smart house simulation based multi-agent system and internet of things. In Proceedings of the 2017 International Conference on Mathematics and Information Technology (ICMIT), Adrar, Algeria, 4–5 December 2017; pp. 201–203. [CrossRef]
4. Scrugli, M.A.; Loi, D.; Raffo, L.; Meloni, P. *A Runtime-Adaptive Cognitive IoT Node for Healthcare Monitoring*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 350–357. [CrossRef]
5. Yang, Z.; Zhou, Q.; Lei, L.; Zheng, K.; Xiang, W. An IoT-cloud Based Wearable ECG Monitoring System for Smart Healthcare. *J. Med. Syst.* **2016**, *40*, 286. [CrossRef] [PubMed]
6. Roberts, L.; Michalák, P.; Heaps, S.; Trenell, M.; Wilkinson, D.; Watson, P. Automating the Placement of Time Series Models for IoT Healthcare Applications. In Proceedings of the 2018 IEEE 14th International Conference on e-Science (e-Science), Amsterdam, The Netherlands, 29 October–1 November 2018; pp. 290–291. [CrossRef]
7. Macis, S.; Loi, D.; Pani, D.; Raffo, L.; Manna, S.L.; Cestone, V.; Guerri, D. Home telemonitoring of vital signs through a TV-based application for elderly patients. In Proceedings of the 2015 IEEE International Symposium on Medical Measurements and Applications (MeMeA) Proceedings, Torino, Italy, 7–9 May 2015; pp. 169–174. [CrossRef]
8. Kaewkannate, K.; Kim, S. *The Comparison of Wearable Fitness Devices*; IntechOpen: London, UK, 2018. [CrossRef]
9. Kaewkannate, K.; Kim, S.C. A comparison of wearable fitness devices. *BMC Public Health* **2016**, *16*, 433. [CrossRef] [PubMed]
10. Ghasemzadeh, H.; Jafari, R. Ultra Low-power Signal Processing in Wearable Monitoring Systems: A Tiered Screening Architecture with Optimal Bit Resolution. *ACM Trans. Embed. Comput. Syst.* **2013**, *13*, 9:1–9:23. [CrossRef]
11. Tekeste, T.; Saleh, H.; Mohammad, B.; Ismail, M. Ultra-Low Power QRS Detection and ECG Compression Architecture for IoT Healthcare Devices. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2019**, *66*, 669–679. [CrossRef]
12. Wang, C.; Qin, Y.; Jin, H.; Kim, I.; Granados Vergara, J.D.; Dong, C.; Jiang, Y.; Zhou, Q.; Li, J.; He, Z.; et al. A Low Power Cardiovascular Healthcare System with Cross-layer Optimization from Sensing Patch to Cloud Platform. *IEEE Trans. Biomed. Circuits Syst.* **2019**, *13*, 314–329. [CrossRef]
13. Adimulam, M.K.; Srinivas, M.B. Ultra Low Power Programmable Wireless ExG SoC Design for IoT Healthcare System. In *Wireless Mobile Communication and Healthcare*; Perego, P., Rahmani, A.M., TaheriNejad, N., Eds.; Springer: Cham, Switzerland, 2018; pp. 41–49.
14. Labati, R.D.; Muñoz, E.; Piuri, V.; Sassi, R.; Scotti, F. Deep-ECG: Convolutional Neural Networks for ECG biometric recognition. *Pattern Recognit. Lett.* **2018**, *126*, 78–85. [CrossRef]
15. Baloglu, U.B.; Talo, M.; Yildirim, O.; Tan, R.S.; Acharya, U.R. Classification of myocardial infarction with multi-lead ECG signals and deep CNN. *Pattern Recognit. Lett.* **2019**, *122*, 23–30. [CrossRef]
16. Li, Y.; Pang, Y.; Wang, J.; Li, X. Patient-specific ECG classification by deeper CNN from generic to dedicated. *Neurocomputing* **2018**, *314*, 336–346. [CrossRef]
17. Tabal, K.M.R.; Caluyo, F.S.; Ibarra, J.B.G. Microcontroller-Implemented Artificial Neural Network for Electrooculography-Based Wearable Drowsiness Detection System. In *Advanced Computer and Communication Engineering Technology*; Sulaiman, H.A., Othman, M.A., Othman, M.F.I., Rahim, Y.A., Pee, N.C., Eds.; Springer: Cham, Switzerland, 2016; pp. 461–472.
18. Magno, M.; Pritz, M.; Mayer, P.; Benini, L. DeepEmote: Towards multi-layer neural networks in a low power wearable multi-sensors bracelet. In Proceedings of the 2017 7th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI), Vieste, Italy, 15–16 June 2017; pp. 32–37. [CrossRef]

19. Flamand, E.; Rossi, D.; Conti, F.; Loi, I.; Pullini, A.; Rotenberg, F.; Benini, L. GAP-8: A RISC-V SoC for AI at the Edge of the IoT. In Proceedings of the 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), Milano, Italy, 10–12 July 2018; pp. 1–4. [CrossRef]
20. Desoli, G.; Chawla, N.; Boesch, T.; Singh, S.; Guidetti, E.; De Ambroggi, F.; Majo, T.; Zambotti, P.; Ayodhyawasi, M.; Singh, H.; et al. 14.1 A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28 nm for intelligent embedded systems. In Proceedings of the 2017 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 5–9 February 2017; pp. 238–239. [CrossRef]
21. Google®. Google TPU. 2020. Available online: https://cloud.google.com/tpu (accessed on 3 August 2021).
22. NVIDIA®. Embedded Systems for Next-Generation Autonomous Machines. 2019. Available online: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems (accessed on 3 August 2021).
23. Meloni, P.; Capotondi, A.; Deriu, G.; Brian, M.; Conti, F.; Rossi, D.; Raffo, L.; Benini, L. NEURAghe: Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs. *ACM Trans. Reconfig. Technol. Syst.* **2018**, *11*, 1–24. [CrossRef]
24. Vissers, K. Versal: The Xilinx Adaptive Compute Acceleration Platform (ACAP). In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '19, Seaside, CA, USA, 24–26 February 2019; Association for Computing Machinery: New York, NY, USA, 2019; p. 83. [CrossRef]
25. Przybył, A. Fixed-Point Arithmetic Unit with a Scaling Mechanism for FPGA-Based Embedded Systems. *Electronics* **2021**, *10*, 1164. [CrossRef]
26. NVIDIA®. NVIDIA cuDNN. 2020. Available online: https://developer.nvidia.com/cudnn (accessed on 3 August 2021).
27. arm Developer. Cortex Microcontroller Software Interface Standard. 2016. Available online: https://developer.arm.com/tools-and-software/embedded/cmsis (accessed on 3 August 2021).
28. Liang, T.; Glossner, J.; Wang, L.; Shi, S.; Zhang, X. Pruning and Quantization for Deep Neural Network Acceleration: A Survey. *arXiv* **2021**, arXiv:cs.CV/2101.09671.
29. Ward-Foxton, S. Artificial Intelligence Gets Its Own System of Numbers. 2020. Available online: https://www.eetimes.com/artificial-intelligence-gets-its-own-system-of-numbers/ (accessed on 3 August 2021).
30. Dziwiński, P.; Przybył, A.; Trippner, P.; Paszkowski, J.; Hayashi, Y. Hardware Implementation of a Takagi-Sugeno Neuro-Fuzzy System Optimized by a Population Algorithm. *J. Artif. Intell. Soft Comput. Res.* **2021**, *11*, 243–266. [CrossRef]
31. Tuveri, G.; Meloni, P.; Palumbo, F.; Seu, G.P.; Loi, I.; Conti, F.; Raffo, L. On-the-fly adaptivity for process networks over shared-memory platforms. *Microprocess. Microsyst.* **2016**, *46*, 240–254. [CrossRef]
32. Jahn, J.; Henkel, J. Pipelets: Self-organizing software Pipelines for many-core architectures. In Proceedings of the 2013 Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France, 18–22 March 2013; pp. 1516–1521. [CrossRef]
33. Choi, Y.; Li, C.H.; Silva, D.D.; Bivens, A.; Schenfeld, E. Adaptive Task Duplication Using On-Line Bottleneck Detection for Streaming Applications. In Proceedings of the 9th Conference on Computing Frontiers, CF '12, Cagliari, Italy, 15–17 May 2012; Association for Computing Machinery: New York, NY, USA, 2012; pp. 163–172. [CrossRef]
34. arm. Arm Compute Library. Available online: https://developer.arm.com/ip-products/processors/machine-learning/compute-library (accessed on 3 August 2021).
35. OAID. Tengine. Available online: https://github.com/OAID/Tengine (accessed on 3 August 2021).
36. Tencent. NCNN. Available online: https://github.com/Tencent/ncnn (accessed on 3 August 2021).
37. Wang, S.; Ananthanarayanan, G.; Zeng, Y.; Goel, N.; Pathania, A.; Mitra, T. High-Throughput CNN Inference on Embedded ARM big.LITTLE Multi-Core Processors. *arXiv* **2019**, arXiv:1903.05898.
38. Wu, H.I.; Guo, D.Y.; Chin, H.H.; Tsay, R.S. A Pipeline-Based Scheduler for Optimizing Latency of Convolution Neural Network Inference over Heterogeneous Multicore Systems. In Proceedings of the 2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), Genova, Italy, 31 August–2 September 2020; pp. 46–49. [CrossRef]
39. Huang, H.; Chaturvedi, V.; Quan, G.; Fan, J.; Qiu, M. Throughput Maximization for Periodic Real-Time Systems under the Maximal Temperature Constraint. *ACM Trans. Embed. Comput. Syst.* **2014**, *13*, 1–22. [CrossRef]
40. Yu, H.; Ha, Y.; Wang, J. Thermal-aware frequency scaling for adaptive workloads on heterogeneous MPSoCs. In Proceedings of the 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 24–28 March 2014; pp. 1–6. [CrossRef]
41. Yu, H.; Ha, Y.; Wang, J. Quality Optimization of Resilient Applications under Temperature Constraints. In Proceedings of the Computing Frontiers Conference, CF'17, Siena, Italy, 15–17 May 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 9–16. [CrossRef]
42. Ma, Y.; Chantem, T.; Dick, R.P.; Hu, S. Improving System-Level Lifetime Reliability of Multicore Soft Real-Time Systems. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* **2017**, *25*, 1895–1905. [CrossRef]
43. Weissel, A.; Bellosa, F. *Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management*; Association for Computing Machinery: New York, NY, USA, 2002; [CrossRef]
44. Vogeleer, K.D.; Memmi, G.; Jouvelot, P.; Coelho, F. The Energy/Frequency Convexity Rule: Modeling and Experimental Validation on Mobile Devices. *arXiv* **2014**, arXiv:cs.OH/1401.4655.
45. Nabavinejad, S.M.; Hafez-Kolahi, H.; Reda, S. Coordinated DVFS and Precision Control for Deep Neural Networks. *IEEE Comput. Archit. Lett.* **2019**, *18*, 136–140. [CrossRef]

46. Motamedi, M.; Fong, D.; Ghiasi, S. Machine Intelligence on Resource-Constrained IoT Devices: The Case of Thread Granularity Optimization for CNN Inference. *ACM Trans. Embed. Comput. Syst.* **2017**, *16*, 1–19. [CrossRef]
47. Bong, K.; Choi, S.; Kim, C.; Yoo, H.J. Low-Power Convolutional Neural Network Processor for a Face-Recognition System. *IEEE Micro* **2017**, *37*, 30–38. [CrossRef]
48. Santoro, G.; Casu, M.R.; Peluso, V.; Calimera, A.; Alioto, M. Design-Space Exploration of Pareto-Optimal Architectures for Deep Learning with DVFS. In Proceedings of the 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy, 27–30 May 2018; pp. 1–5. [CrossRef]
49. Lee, E.; Parks, T. Dataflow process networks. *Proc. IEEE* **1995**, *83*, 773–801. [CrossRef]
50. Pimentel, A.D. Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration. *IEEE Des. Test* **2017**, *34*, 77–90. [CrossRef]
51. Meloni, P.; Loi, D.; Deriu, G.; Pimentel, A.D.; Sapra, D.; Moser, B.; Shepeleva, N.; Conti, F.; Benini, L.; Ripolles, O.; et al. ALOHA: An Architectural-aware Framework for Deep Learning at the Edge. In Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications, INTESA '18, Turin, Italy, 13–18 October 2018; ACM: New York, NY, USA, 2018; pp. 19–26. [CrossRef]
52. Goodfellow, S.; Goodwin, A.; Eytan, D.; Greer, R.; Mazwi, M.; Laussen, P. Towards understanding ECG rhythm classification using convolutional neural networks and attention mappings. In Proceedings of the 3rd Machine Learning for Healthcare Conference, Palo Alto, CA, USA, 17–18 August 2018.