3-15-2021

On-Device Deep Learning Inference for System-on-Chip (SoC) Architectures

Tom Springer

Elia Eiroa-Lledo

Elizabeth Stevens

Erik Linstead

On-Device Deep Learning Inference for System-on-Chip (SoC) Architectures

electronics

MDPI

Article

On-Device Deep Learning Inference for System-on-Chip (SoC) Architectures

Tom Springer *, Elia Eiroa-Lledo, Elizabeth Stevens and Erik Linstead *

Fowler School of Engineering, Chapman University, Orange, CA 92866, USA; eiroalledo@chapman.edu (E.E.-L.); estevens@chapman.edu (E.S.)

* Correspondence: springer@chapman.edu (T.S.); linstead@chapman.edu (E.L.); Tel.: +1-714-289-3159 (E.L.)

Abstract: As machine learning becomes ubiquitous, the need to deploy models on real-time, embedded systems will become increasingly critical. This is especially true for deep learning solutions, whose large models pose interesting challenges for target architectures at the "edge" that are resource-constrained. The realization of machine learning, and deep learning, is being driven by the availability of specialized hardware, such as system-on-chip solutions, which provide some alleviation of constraints. Equally important, however, are the operating systems that run on this hardware, and specifically the ability to leverage commercial real-time operating systems which, unlike general purpose operating systems such as Linux, can provide the low-latency, deterministic execution required for embedded, and potentially safety-critical, applications at the edge. Despite this, studies considering the integration of real-time operating systems, specialized hardware, and machine learning/deep learning algorithms remain limited. In particular, better mechanisms for real-time scheduling in the context of machine learning applications will prove to be critical as these technologies move to the edge. In order to address some of these challenges, we present a resource management framework designed to provide a dynamic on-device approach to the allocation and scheduling of limited resources in a real-time processing environment. These types of mechanisms are necessary to support the deterministic behavior required by the control components contained in the edge nodes. To validate the effectiveness of our approach, we applied rigorous schedulability analysis to a large set of randomly generated simulated task sets and then verified the most time critical applications, such as the control tasks which maintained low-latency deterministic behavior even during off-nominal conditions. The practicality of our scheduling framework was demonstrated by integrating it into a commercial real-time operating system (VxWorks) then running a typical deep learning image processing application to perform simple object detection. The results indicate that our proposed resource management framework can be leveraged to facilitate integration of machine learning algorithms with real-time operating systems and embedded platforms, including widely-used, industry-standard real-time operating systems.

Keywords: system-on-chip; deep learning; heterogeneous multiprocessor scheduling; feedback control; real-time operating systems; Internet-of-Things

Citation: Springer, T.; Eiroa-Lledo, E.; Stevens, E.; Linstead, E. On-Device Deep Learning Inference for System-on-Chip (SoC) Architectures. *Electronics* **2021**, *10*, 689. https://doi.org/10.3390/electronics10060689

Academic Editor: George A. Papakostas

Received: 16 February 2021
Accepted: 11 March 2021
Published: 15 March 2021

1. Introduction

Deep learning is currently revolutionizing many machine learning applications such as computer vision, natural language processing, and autonomous systems. Deep learning is based upon architectures of artificial neural networks containing many hidden layers. While training such architectures was once impractical, the availability of large volumes of data, hardware acceleration, and improved algorithms have made deep artificial neural networks a viable alternative for many applications. As a result, deep learning has demonstrated increasingly superior performance, even exceeding human accuracy in some domains.

Overall, deep learning solutions are deployed via a two-stage process. During the first stage, the DNN model is learned by presenting the network with a training data set

and, in the case of supervised learning, adjusting the network weights based on observed prediction error. For example, in the case of image processing, the network would be trained by providing it thousands of different images. Obviously, this stage is computationally intensive, often requiring multi-threaded GPUs or other high-performance computing clusters. Typically performed offline or in the cloud, this stage could take hours or even days, depending on the amount of data required to train the system adequately.

Inference is the next stage of the deployment pipeline, where the trained model is installed on the target hardware and used to infer or predict the output values used directly by the target processor. This target system (i.e., edge or end device) would then process its raw sensor data, providing that data as input to the deployed deep learning model, and using the model's output to classify the inputs. For example, an image classifier application could process raw camera data to recognize various categories of ordinary objects.

Depending on the application, the inference stage may have a much lower latency requirement, usually in the range of milliseconds instead of hours or days compared to the training phase. Examples of these types of systems would include autonomous vehicles, robotic systems, or other safety-critical embedded systems. As a result of these latency requirements, there has been a significant amount of effort focused on developing new hardware and software techniques for faster inference solutions.

On the hardware side, there are numerous reports [1–3] that custom-designed chips for machine learning will deliver an order of magnitude acceleration improvement over current platforms. While the training phase will likely still reside in traditional data centers, there is a much higher possibility that the inference phase will run on the edge in some embedded computing device. There are many reasons for moving computing closer to where sensors gather data, including reliability, limiting network bandwidth, providing improved security, or more effective resource management of deep learning applications. This challenge is gaining research traction, for example, in [4], authors discuss the deep learning memory challenge caused by scarce availability of memory. They propose a combination of techniques for deployment of next-generation of on-chip machine learning including hardware-aware DNNs that could allow for this stage to be on-chip as opposed to offline. In the work of [5], authors also acknowledge the importance of online scalability of deep learning algorithms and propose an architecture that supports an always-on inference and learning engine through on-chip learning. This algorithm leverages a crossbar array structure to perform vector-matrix multiplications and enables online training of multi-layer SNNs with memristive neuromorphic hardware. Their results show minimal performance loss when compared with current state-of-the-art technology. Further, authors of [6], developed an end-to-end framework, DNN+NeuroSim V2.0, to benchmark compute-in-memory-based architectures for on-chip training.

With the shift to on-device inference, the question becomes what type of embedded hardware and configuration will produce the fastest and most efficient deep learning platforms. One particular platform that is gaining significant traction is system-on-a-chip (SoC) architecture. SoCs include various peripherals and computing components on an integrated circuit that can satisfy even some of the most stringent processing requirements for an embedded application. For this reason, SoCs are often considered as an excellent solution for implementing deep-learning applications on embedded systems. In fact, many experts are predicting that most of the effort spent on accelerating inference performance will go towards embedded hardware solutions such as SoCs or other custom processing elements like field-programmable gate arrays (FPGAs).

However, many of those machine learning applications intended to execute on the targeted hardware platforms require real-time performance and must operate under strict timing constraints. For instance, consider the video stream from the camera on a self-driving car. The data must be processed in real-time with predictable behavior so that other critical subsystems (e.g., route planning) are able to respond in a timely manner to a rapidly changing environment (e.g., to avoid collisions). Or, perhaps, consider the cybersecurity domain, where fast and efficient intrusion detection is essential to protect the

system itself [7]. In both cases, the runtime performance of the underlying models must be deterministic.

The challenge is that traditional machine learning applications are developed for server, desktop, or hand-held-based platforms that cannot process input data in a predictable amount of time. The reason for this is two-fold. One reason is that the machine learning- based applications typically run on general purpose operating systems (GPOS), such as Linux, which by default are not real-time. The other is that SoCs' heterogeneous processing is routinely under-utilized in general purpose computing environments because embedded devices require an in-depth knowledge of various hardware architectures and interfaces.

To resolve some of the issues described in the previous paragraphs, we present an online feedback-based real-time scheduling and resource management framework (FC-RTS) for deep learning inference on embedded heterogeneous SoC-based architectures. In this work, we leverage a real-time operating system (RTOS) to replace current GPOSs for the purpose of providing more deterministic behavior and a higher granularity of control over hardware components. To effectively manage unpredictable inference times, we leverage feedback mechanisms to dynamically reallocate processing resource elements based on an application's criticality. This approach is to ensure that the most critical real-time tasks (e.g., collision avoidance) will not be affected during sudden workload spikes. To demonstrate our framework's utility, we integrate it with a computer vision task based on deep learning and convolutional neural networks. We choose this task for two reasons. First of all, deep learning models are some of the largest and most computationally intense models in the entirety of machine learning, and so the ability of our framework to perform well on a deep learning task would indicate that it could generalize to other machine learning approaches. Secondly, computer vision problems are a central part of autonomous systems research, and therefore of broad interest to the real-time machine learning community. The authors in [8] highlight the importance of timeliness for safety in computing systems for autonomous driving. Autonomous vehicles take in a myriad of real-time data which needs to be processed and assessed with deep learning algorithms. The vehicles, however, need to respond to safety threats almost instantly and therefore need an RTOS to satisfy the system's desired response time. By considering the above issues, this paper makes the following contributions: (1) The introduction of a real-time resource management framework that can support deep learning at the edge; (2) The integration of that framework with a major commercial real-time operating system (VxWorks) on a commodity embedded platform.

To describe this new framework, we organize the remaining sections of this paper as follows. The next section provides some background and overview to provide a reference for the methodologies and approaches used by our adaptive framework. Section 3 will present some related work involving deep learning inference as well as multiprocessor scheduling for heterogeneous systems (e.g., systems that utilize multiple cores of different types, such as a combination of CPUs and GPUs). Section 4 provides details on our framework's implementation, while Section 5 covers the performance analysis related to a specific application. Section 6 covers the physical implementation as well as the simulation environment and representative hardware implementation. Finally, Section 7 provides a summary of results discovered thus far as well as future work and recommended enhancements to FC-RTS.

2. Preliminaries

This section is used to provide some technical background information on the techniques used as well as the terminology needed to define our real-time management framework for SoC-based platforms.

2.1. Task Model

Each embedded machine learning application (e.g., image processing or collision avoidance) consists of one or more real-time tasks. A real-time task is defined as any task that must meet its timing constraints. The timing constraints are traditionally represented as a 3-tuple $(T,\ C,\ D)$ where T defines the task period (time interval where the task instance needs complete execution), C denotes the worst case execution time (WCET) of the task, and D defines the relative deadline of a task (relative to the start of the task as to when the task needs to finish execution). Therefore, each application can then be represented as a task set of one or more tasks consisting of $\Gamma_s = \{\tau_1, \tau_2, \ldots \tau_n\}$, where each task τ_i is defined as (T_i, C_i, D_i). It is assumed that each task τ_i is a constrained task, such that $C_i \leq D_i \leq T_i$.

Other important characteristics of real-time tasks include task priority, release time, absolute deadline and the utilization of a task. The priority of a task determines what task is chosen to execute first. The release time defines when a task is ready to execute, not necessarily when it is allowed to execute. The absolute deadline defines the relative time when the task is started compared to when it must complete. Note that a task which cannot miss a single deadline is classified as hard real-time, whereas a task that can tolerate some missed deadlines is known as soft real-time. Utilization is defined as the ratio between the period of the task and its WCET.

Additionally, there are three different categories of real-time tasks: periodic, sporadic, and aperiodic. A periodic task is where an instance of the task is released at the start of each period. A sporadic task does not have a period but has a minimum interval of time between releasing another instance of the task. An aperiodic task has no period nor time interval between tasks. Aperiodic tasks are especially challenging to plan for because they can happen at any time, such as when a system is interrupted to handle some asynchronous event. These characteristics are worth mentioning in that an embedded machine learning application may contain any combination of task types listed above and therefore needs to be accounted for when planning on doing any on-device inference.

2.2. Real-Time Scheduling

It is up to a scheduling algorithm to determine which tasks get to execute and for how long. There are several important characteristics of real-time scheduling algorithms that are worth mentioning. The first is whether a task is preemptive or non-preemptive. A preemptive task is one that can be interrupted at any time, allowing other tasks to run. In contrast, a non-preemptive task is allowed to continue uninterrupted. This is an important concept related to how we manage the scheduling of tasks in a heterogeneous processing architecture.

Another key characteristic of real-time scheduling is the notion of priority. In a priority-based scheduling scheme, the task with the highest priority gets to run first, where priority can be assigned either statically or dynamically. In a static or fixed priority scheme, each task is assigned a priority that does not change during the task's lifetime. An example of a fixed priority scheme is the Rate Monotonic (RM) algorithm where the priority of a task is determined by the inverse of its period. In a dynamic priority scheme, the priority of a task can change during the lifetime of the task. The most common example of dynamic priority scheduling is the Earliest Deadline First (EDF) algorithm, where the priority of a task can change depending upon the task with the nearest deadline that is ready to run.

2.3. Multiprocessor Real-Time Scheduling

Up to this point, the scheduling schemes described in Section 2.2 apply to systems with a single processor. However, SoC-type devices typically consist of multicore-based architectures. A few main multiprocessor or multicore scheduling algorithms are designed to manage multicore architectures: partitioned scheduling, global scheduling, and a third hybrid type that combines elements of the other two approaches. In the partitioned scheduling approach, tasks are assigned to a specific processor and not allowed to migrate to other processors even if those processors are under-utilized. This leads to the challenge

of efficiently assigning tasks to processors, which is analogous to the bin packing problem, which is proven to be NP-Hard [9].

Global scheduling solves the complexity problem of the partitioned approach because a single global scheduler assigns tasks. Another key advantage of global scheduling is that it typically requires fewer preemptions as the scheduler will only preempt a task if there is no idle processor. Global scheduling is more suitable for open systems where new tasks arrive dynamically, as a new task can be added easily to an existing schedule without assigning it to a particular partition. A primary disadvantage is the significant overhead incurred when a task is allowed to migrate to another processor or core.

Additional challenges are based upon the notion that SoCs are heterogeneous architectures with one or more general purpose processors (CPUs) with one or more synergistic coprocessors (e.g., GPUs or DSPs) meant to deliver increased performance and power efficiency. However, the multiprocessing scheduling mechanisms mentioned previously present certain obstacles in heterogeneous multiprocessor systems due to the significant overhead involved in the preemption of tasks running in the coprocessor. This overhead is an architectural side-effect related to the number of registers, pipeline stages, and cache flushes [10,11] associated with the coprocessor. Therefore, the traditional way of scheduling the coprocessor, to reduce this incurred overhead, has been to designate that the CPU schedules both the CPU-bound tasks and the coprocessor-bound non-preemptive tasks.

In practice, the GPU or DSP is considered a functional accelerator, much like a floating-point coprocessor, that responds to processing requests from the CPU [12]. However, the scheduling problems of heterogeneous multicore systems are complicated by the precedence constraints imposed by the master-slave relationship of the processor and coprocessor. The CPU invokes the coprocessor code in a similar manner as a remote procedure call (RPC), which is initially executed on the processor and then dispatched to execute specific instructions on the coprocessor. This means each task that utilizes a GPU or DSP will initially execute on the processor, at time t_n make a call to execute on the GPU which will complete without preemption then return control back to the CPU.

For instance, consider the example illustrated in Figure 1. At t_0 the processor executes for C_i^{pre} at t_3 makes a call to the GPU where it executes for C_i^{cp} then at t_{10} completes and returns to the CPU where the task instance finishes at t_{12}. Notice that the CPU code waits for each GPU-related operation to complete. In effect, a gap is created within each task instance because the GPU instructions are executed on the coprocessor.

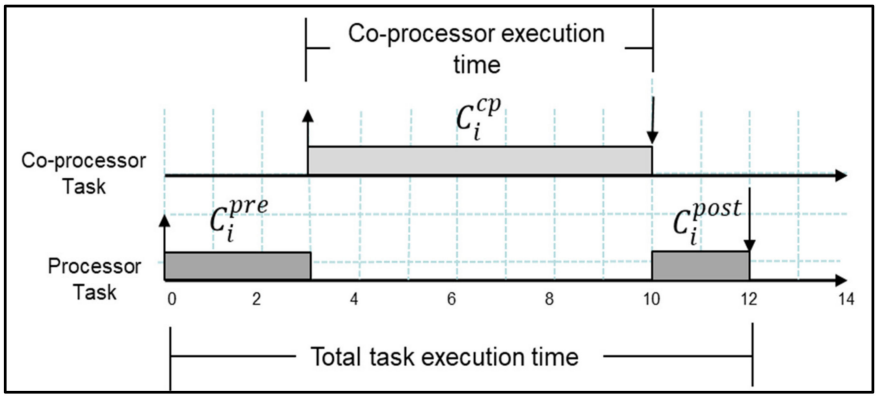


Figure 1. CPU-GPU task schedule.

This example scenario highlights several of the challenges associated with scheduling real-time GPU-based applications. One is the processing gap created in the processor when the CPU task is waiting on the GPU task to finish. In this example, the processor task is idle 50% of the time, resulting in a decrease in the system's overall schedulability. Another obstacle is the processors need to be allocated according to a predictable scheduling policy to ensure that task timing constraints are met. This is difficult because there is underlying

complex proprietary software that is used to execute the GPU's instructions, and that software is not designed for deterministic real-time processing.

To facilitate a real-time environment, one of the overall goals is to exploit those processing gaps to schedule other CPU-based tasks in order to improve the schedulability bound of the system. Another is to provide for more predictable behavior of the coprocessor by treating it as a special case of shared resource management in a multiprocessor system [13,14]. The final goal is to create a dynamic run-time environment that can adapt to changing processing requirements by re-allocating resources based upon the criticality or importance of a particular application.

3. Related Work

Authors in [15] presented a case study that examined various optimizations, algorithms, and platforms for machine learning in resource-scarce embedded systems. Their paper presented guidelines and recommendations for techniques to be used when implementing machine learning applications on resource-scarce devices. Authors emphasize that by moving applications to the network's edge devices, internet traffic is reduced, latency issues can be mitigated, network security improved, and real-time performance provided in the system. They acknowledge that in order for an end-device to meet strict real-time deadlines, the devices have to run either a bare-metal application or an RTOS instead of a GPOS. The reason is that an RTOS allows for the prioritization of tasks to meet deadlines where a standard GPOS cannot ensure the task's timing constraints are met.

Other work in [16] presented by researchers looked at making machine learning toolkits compatible with real-time systems, specifically general purpose embedded processors. Their work focused on porting Support Vector Machines (SVM) onto embedded processors across two communication networks. Their platforms consisted of an ARMv7 processor running Linux and a PPC440 processor running the Green Hills Integrity RTOS. The main effort of this work focused on optimizing runtime performance and the memory footprint of SVM libraries. Various runtime optimizations included removing doubles, disabling exceptions and in-lining functions. Other modifications included fixed-size vectors or caching dot product calculations. Their work mainly focused on application-level changes, with some minor changes made to the kernel by removing expensive function calls in the kernel. One particularly interesting finding mentioned by the authors in the paper was the discovery about how unpredictable the response times can be when presented with differing code and data structures.

Applying machine learning to embedded sensor systems was another approach investigated by researchers [17] looking into integrating machine learning techniques for Internet-of-Things (IoT) based applications. The authors' approach was to develop an efficient real-time realization of a Gaussian mixture model (GMM) for execution on an embedded sensor board. Their work presented real-time data analytics of sensor data with continuous training of the machine-learning platform. However, while real-time capabilities were advertised, there was no more mention of any specific timing constraints or consequences of missed deadlines. The implication was that the sensor data was processed as soon as it was received with minimal latency, which would not necessarily be classified as a real-time system.

The balance between hardware and software implementations of machine learning algorithms was examined by authors [18] to demonstrate the possibilities of moving the critical sections of the machine learning algorithm into hardware in the form of an embedded SVM. Their overall goal was to optimize the SVM by moving code back and forth between the software and the hardware—notably an FPGA. The authors argued that processor size could be traded against performance to fit into other deployment scenarios, thus providing a type of static adaptability.

An empirical study was conducted by Ogden and Guo [19] that analyzed the performance tradeoffs between on-device inference versus cloud-based inference for mobile applications. Their study concluded that on-device inference was suitable for certain tasks

on newer mobile devices with specific architecture optimized hardware. They also demonstrated that unpredictable or varying network conditions could lead to poor cloud-based inference throughput or response times.

A real-time GPU scheduling framework, called GPUSync [20,21], was introduced to address GPU-based real-time scheduling issues, such as assigning tasks to a specific GPU, managing memory transactions, and scheduling the computations. As GPU allocation can be viewed as a scheduling problem for mutually exclusive resources, their approach was to implement a synchronization-based methodology. Due to the high overhead costs associated with preemption in the GPU and since multiple tasks may be sharing the GPU(s), it is reasonable to treat the coprocessor execution as a critical section of code in the CPU. In this way, existing real-time locking protocols [21] can be applied to the GPU allocations, providing more determinism and predictability required by a real-time environment.

Authors in [22] used machine learning to manage complex tasks in integrated circuit technologies. The tasks are managed by an operating system whose main role is defining resource allocation and temporal scheduling. The authors proposed a neural network-based model for the design of a scheduler for a heterogeneous multiprocessor. However, this approach requires a static approach where all tasks have to be defined apriori. Any change to the task set requires a new scheduler to be modeled and developed. Practically speaking, modifying a new scheduler in an operating system kernel every time the task set changes is non-trivial. For monolithic operating systems like Linux, it is nearly intractable.

For some of these reasons, our motivation was to create an approach that did not require systemic changes to the kernel but instead to run on top of a commercially available real-time operating system. We believe from our research that our approach was the first to implement a scheduling architecture for a heterogeneous multiprocessor with real-time guarantees.

4. Real-Time Scheduling Framework

This section provides a description of the architecture (see Figure 2) used to provide for effective resource management in a constrained resource-scarce embedded environment. This framework leverages feedback control, limited preemption and a real-time operating system to satisfy the transient and steady performance requirements of a real-time system.

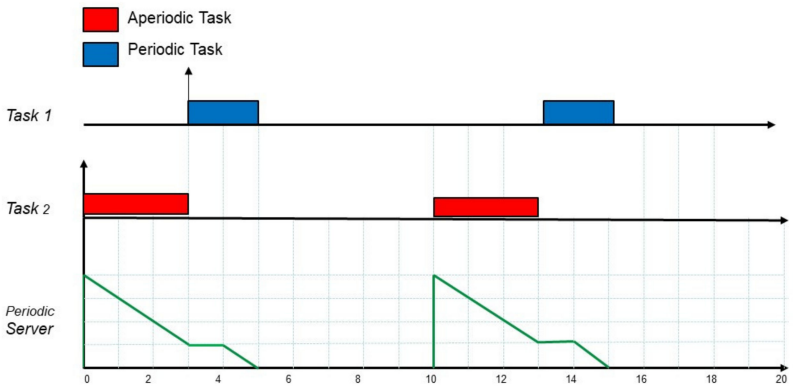


Figure 2. Periodic server.

4.1. Feedback Control

The purpose of using feedback control is to provide for an adaptive method of resource allocation by dynamically shifting resources (i.e., processor, coprocessor) when needed based upon the criticality of a task. Adaptive scheduling algorithms have been proven to be an effective method for achieving performance guarantees in unpredictable computing environments [23]. Our feedback control mechanism is based upon the feedback control real-time scheduling algorithm (FCS) presented by authors in [24] then expanded to support

multiprocessor systems similar to the algorithm presented by authors in [25]. We then further expanded and refined this architecture to support heterogeneous (SoC) architectures.

4.1.1. Task Model

Initially described in Section 2.1, the task model is expanded to support the scheduling architecture of the feedback control structure. Tasks are characterized by the following 4-tuple: $\{D_i, Ce_i, Ca_i, L_i\}$, where D_i defines the relative deadline, Ce_i the estimated execution time, Ca_i the actual execution time and L_i represents the criticality level of the task. Specifically, periodic tasks are defined as $\{P_i, Ue_i, Ua_i\}$, P_i denotes the invocation period, estimated CPU utilization is defined as $Ue_i = Ce_i/P_i$ and actual CPU utilization is defined as $Ua_i = Ca_i/P_i$.

Aperiodic tasks are unique in that they can reoccur at random times and are typically soft real-time tasks. Examples could include keyboard input or mouse movements. A motion detection sensor could be considered an aperiodic task. The problem is due to their inherent unpredictability, aperiodic tasks present scheduling problems in a real-time system. Hence, the goal is to complete each aperiodic task as soon as possible, without causing periodic tasks to miss their deadlines. One common solution used to create a predictable schedule for aperiodic tasks is to make them behave like a periodic task. The idea is to create a periodic task that is created to execute aperiodic tasks. That periodic task is known as a periodic server. A periodic server is defined as $S_i = (P_i, Q_i, L_i)$, where P_i, defines the task period, Q_i the task's budget (which defines how many units of time the server is to execute) and L_i which defines the criticality of the task. The periodic server (see Figure 2) never executes more than Q_i units of time during any time interval P_i. When a server is scheduled, it executes any aperiodic task waiting, consuming its budget until the task is completed or exhausted. During the next scheduling period, the full budget is replenished. In the example provided in Figure 2, task 3 is an aperiodic task and controlled by a periodic server. As the task executes, it consumes the server's budget $Q_i = 5$ until either the full budget has been consumed or the task ends and then the budget gets replenished every period, $P_i = 10$.

4.1.2. Feedback Control Parameters

In order to provide feedback control, variables used to measure the performance of the controller need to be defined. Similar to the parameters defined by authors in [24], we include deadline miss ration and CPU utilization extended to a multiprocessor platform. The miss ratio is defined as $M(k)$ at the k^{th} sampling instant where the number of missed deadlines is equal to deadline misses divided by the number of completed or rejected tasks over a sampling window. The sampling window is defined as $((k-1)W, kW)((k-1)W, kW)$, where W is the sampling period and k is called the sampling instant. Variable $U(k)$ determines the CPU utilization at the k^{th} sampling instant and is used to regulate deadline misses while managing overall system utilization.

Using those control parameters, we can now represent the desired performance of the real-time system, where deadline misses are defined by M_s and CPU utilization is defined as U_S. As an example, consider a system where very few or no deadline misses can be tolerated but lower CPU utilization is acceptable. In this case, we may set the control variables $M_s = 0$ and U_S. = 0.50. In control theory, the difference between the control variables and their actual values is known as the error. Miss ratio error is defined as $E_M = M_S - M(k)$ and the utilization error is defined as $E_U = U_S - U(k)$. The performance goal is to maintain an overall error ratio of zero.

To achieve an error ratio of zero, certain parameters can be changed dynamically by the system to affect the values of the controlled parameters. We can choose the total requested CPU utilization of all tasks in the system as the parameter to be changed dynamically, known as the manipulated variable in control theory. The idea is that by controlling the CPU utilization, we can effectively minimize the miss ratio of all tasks.

4.2. Feedback Control-Based Scheduler

The scheduler in our framework is based upon the FC-GEDF and FC-EDF scheduler presented by authors in [25,26], then extended to support co-operating tasks in a multiprocessor heterogeneous platform. FC-EDF was initially designed to support independent tasks in a uniprocessor platform and FC-GEDF was designed for homogenous multiprocessor platforms only.

The scheduler is dynamic where any task can be admitted to the system depending onto the loading conditions there is the potential for starvation of lower priority tasks. Periodic tasks are inserted directly into the task queue, while aperiodic tasks are assigned to a periodic server which in effect creates a periodic task. The schedule analyzer is used to create a schedulable task set that is provided to the scheduler for actual allocation onto the processor for execution.

The scheduler framework (see Figure 3) is comprised of a number of components which include a System Monitor, a PID controller, a Bandwidth Actuator and a Schedule Analyzer. The following subsections provide further details on each component.

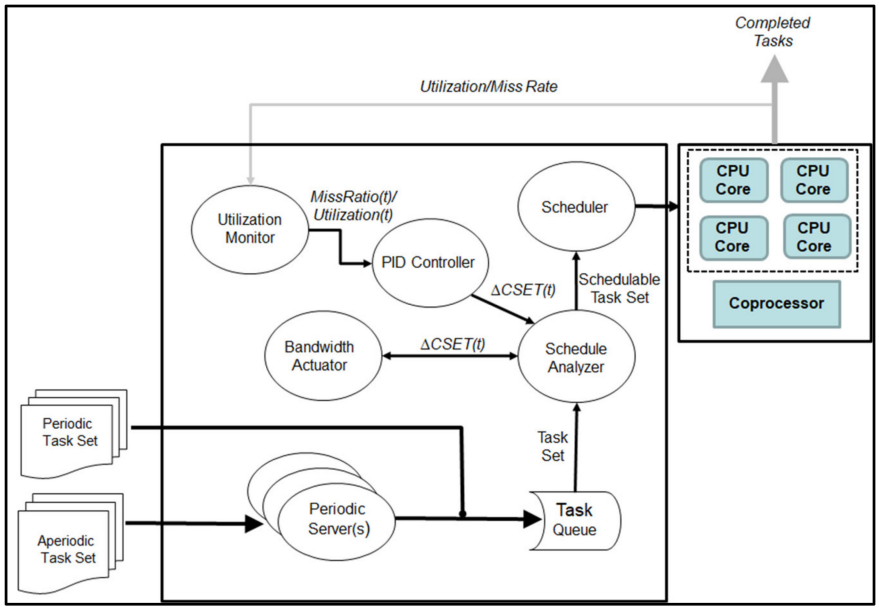


Figure 3. Feedback control real-time scheduler (FC-RTS).

4.2.1. Utilization Monitor

The Utilization Monitor samples the controlled parameters, miss ratio $M(k)$ and utilization ratio $U(k)$ and feeds that information back into the PID controller.

4.2.2. PID Controller

The PID controller is used to adjust the utilization $U(k)$ in order to drive the miss ratio $M(k)$ needs to be at or near zero. The goal is to maximize utilization but minimize or eliminate task deadline misses, especially in tasks with a high criticality value. According to authors in [27]. the controller is based upon the following formula:

$$D_B(k) = C_p error(k) + C_I \sum_{IW} error(k) + C_D error(k) - error(k - DW)/DW \quad (1)$$

where $D_B(k)$ refers to the change in bandwidth utilization (i.e., CPU utilization), $error(k) = M_S - M(k)$, the C_p, C_I, C_D variables refer to the proportional, derivative and integral parameters. The parameter IW defines the time window over which the integral errors are summed while DW defines the time window over which the derivative errors occur. The value $D_B(k)$ defines the output of the PID controller, where $D_B(k) > 0$ indicates that bandwidth utilization should be increased while $D_B(k) < 0$ indicates that the overall CPU bandwidth

should be decreased. It is important to note that the PID controller does not actually increase the bandwidth, just that it is merely suggesting on what it should be according to the controller. It is the job of the Schedule Analyzer to actually determine if the increased bandwidth is schedulable with no missed deadlines.

4.2.3. Schedule Analyzer

The Schedule Analyzer determines if the task set is actually schedulable based upon the increased bandwidth allocation requested by the controller. If the task set is determined to be un-schedulable by the analyzer, then the bandwidth from lower criticality tasks is borrowed until sufficient bandwidth is acquired. Note that this approach can lead to starvation of lower criticality tasks, but it is assumed that increased bandwidth allocation resulting in an overload condition $U_S > 1$ (i.e., not enough bandwidth in the system to handle all requests) is transitory. If the overload condition is long-term or permanent, then the recommendation is to offload some of the workload to other resources or utilize a more resource-available computing platform.

While there has been a significant amount of work in real-time schedulability analysis for symmetric multiprocessing [28–31], there are unique challenges presented when scheduling for asymmetric architectures composed by a general purpose processor and various other coprocessors, such as a DSP or GPU. The issue is how to verify that a coprocessor can provide some degree of real-time performance. This is because general purpose scheduling is preemptive, but execution in a coprocessor is usually non-preemptive, which might introduce longer or unpredictable blocking times potentially violating the real-time constraints of the system. One approach to this problem was proposed in [32], where the idea of co-scheduling was introduced. The basic idea of co-scheduling is to divide each task into separate code blocks, associating suitable deadlines to each block in order to meet the task deadline. The disadvantage of this method is that programmer or compiler intervention is required to create the code blocks. Another method proposed by authors in [33] was to re-arrange the scheduling in order to account for tasks that use the coprocessor. This is done by modeling the coprocessor activity as a blocking factor. So, when a task requests a coprocessor activity, it blocks for a time B_i waiting for the activity to complete. In this way, a task using the coprocessor blocks all the other tasks, requiring the coprocessor to free the regular tasks to execute on the master processor in the gaps created by the coprocessor activities.

In our framework, we implement a similar approach except we utilize a blocking primitive (i.e., semaphores) to suspend the task waiting on the coprocessor activity to finish allowing other tasks to execute while the cooperating task blocks. The advantage of this approach is that blocking primitives are already implemented in any standard RTOS, so there is no requirement to modify the scheduler to compute the blocking factor for the cooperating task. Therefore, in order to determine if a task set is schedulable, we must consider the blocking factors and how those factors affect the whole system. A couple things to consider are that cooperating tasks could be delayed by other cooperating tasks which may already hold the coprocessor or other higher priority regular tasks that may interfere with the cooperating task before it is scheduled. Therefore, as demonstrated in [33], the blocking factor *B_i* of a cooperating task could be computed using the following formula:

$$B_i = C_i^{cp} + B_i^{lp} + B_i^{hp} \tag{2}$$

where C_i^{cp} defines the execution time for the coprocessor, B_i^{lp} defines the lower priority blocking time and B_i^{hp} defines the blocking time associated with any higher priority tasks. The lower and higher priority task blocking times can be defined as follows:

$$B_i^{lp} = \max_{P_j < P_i} \left\{ C_j^{cp} \right\}$$

$$B_i^{hp} = \sum_{P_j<P_i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j^{cp}$$

where P_j is defined as a lower priority task than P_i in the task set T_n. Based upon the Equation (2), we can calculate an upper bound B_i on both cooperating and independent tasks using the following equation:

$$B_i = \begin{cases} C_i^{cp} + \underset{P_j<P_i}{max}\left\{C_j^{cp}\right\} + \\ \quad \sum_{P_j<P_i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j^{cp}, \quad cooperating \\ 0, \qquad\qquad independent \end{cases} \tag{3}$$

The value derived from Equation (3) is then used in Equation (4) by the Scheduler Analyzer to determine if the task is schedulable.

$$\forall_i = 1, \ldots, n \sum_{P_j \geq P_i} \frac{C_j}{T_j} + \frac{B_i}{T_i} \leq U_{lub}(i) \tag{4}$$

where $U_{lub}(i) = i\left(2^{\frac{1}{i}} - 1\right)$. This value is then used by the Bandwidth Actuator to assign CPU utilization to each task. If it is determined that the current task set is schedulable, then the tasks are allocated their assigned WCET. If the task set is deemed not executable, then the bandwidth from lower criticality tasks can be temporarily eliminated. In this case, the lower criticality task would be blocked causing the task to miss deadlines. The task would have to wait until additional bandwidth became available through the feedback process in order to have an opportunity to run again.

4.2.4. Fixed Priority Scheduler

At run-time, the fixed priority scheduler chooses the highest priority task that is ready to run. The priority is based upon the task period P_i so the shorter the period, the higher the task priority. Therefore, if the priority of $T_i > T_j$ then T_i would be scheduled first with its full bandwidth Ce_i then T_j would be scheduled next with its full estimated bandwidth and continue on until all ready tasks have be scheduled or an un-schedulable task set is detected.

In the event the task set is not schedulable, instead of the highest priority task receiving their estimated bandwidth Ce_i, the task with the highest criticality level will receive their full bandwidth. Therefore, the scheduler redistributes CPU bandwidth based upon the criticality level which means lower criticality tasks yield their bandwidth to higher criticality tasks. This greedy approach can lead to starvation, even for some high priority tasks, but this is acceptable in that during overload conditions, the highest criticality tasks are considered superior to lower criticality tasks.

4.2.5. Bandwidth Actuator

After the requested CPU bandwidth has been allocated to the highest criticality task, the lower criticality tasks need to be reallocated. The next lower criticality tasks are then assigned CPU bandwidth based upon the remaining utilization. The algorithm and description for bandwidth allocation is provided as shown in Algorithm 1.

Algorithm 1 Bandwidth Actuator

Input: A task set T_j with criticality levels less than the task set of T_i
Output: A new bandwidth value for each task that will maintain a schedulable system
1: **for** *each* $t_j \in T_j$ **do**
// Assign C_j full bandwidth and perform
// schedulability test
2: $C_j = AllocateBandwidth()$
3: $S = ScheduleAnalyzer\left(T_i,\ T_j\right)$
4: **while** *S not schedulable* **do**
// C_j could be reduced or 0
5: $C_j = AllocateNewBandwidth\left(T_j,\ C_j\right)$
6: **end while**
7: **end for**

The bandwidth actuator algorithm (Algorithm 1) works by iterating through all the tasks T_j which are of lower criticality than tasks T_i. In line 2, the new bandwidth is calculated based upon the remaining system utilization. The schedule analyzer (line 3) is invoked to analyze the calculated utilization. If the modified bandwidth renders the system un-schedulable, then a new utilization value is attempted based upon the previous failed value. The algorithm continues to reduce the bandwidth of lower criticality subsystems until a schedulable system is found.

5. Simulation Analysis Results

The performance analysis of the feedback-based real-time scheduling framework has been evaluated using a simulated set of tasks, similar to the same approach adopted by authors in [33] who evaluated the performance of multiprocessor DSP scheduling. Comparison performance was based upon the static priority tasks used by traditional schedulers, where the priority of the task does not change versus dynamic priority tasks that can change during the lifetime of the task based upon environmental conditions. Other heterogeneous multicore schedulers presented by other researchers are not necessarily applicable because they are either static-based scheduler, where the task set must be identified apriori, or not specific to real-time scheduling.

Both periodic and aperiodic task sets were generated using random values with a uniform distribution. The number of periodic tasks were varied from 2 to 20. Cooperating tasks were generated to be 50% of the total number of periodic tasks. Aperiodic tasks were not modeled to use the coprocessor. The number of aperiodic tasks were varied from 1 to 5. Periodic task periods were generated from 5 to 100 while periodic server periods were varied from 10 to 100. Independent periodic task worst case execution times $C_i' = C_i$ and cooperating worst case execution times $C_i' = C_i + C_i^{cp}$ were selected to create an overall utilization that varied from 0.35 to 0.90. The worst case execution time C_i^{cp} of cooperating tasks were generated to vary between 10% to 80% of the total task execution time C_i'. For our initial performance analysis, we modeled one processor and one coprocessor, with future work to include models consisting of multiple general purpose processors and special purpose coprocessors. The first experiment compared our adaptive feedback-based approach against the standard fixed priority scheduling algorithm in terms of overall utilization factor and task deadline misses.

The first experiment analyzed the standard fixed priority scheduling algorithm RM, the criticality of the task is listed to provide reference but was not considered for schedulability analysis. Figure 4 depicts the results of a typical simulation run. System utilization represents the overall utilization recorded in the system. The Miss Rate (Low-Criticality) represents the overall average miss rate for tasks that are considered to be less critical. The Miss Rate (High-Criticality) represents the overall average miss rate for tasks considered to be highly critical. Notice from the figure that both task types are meeting all deadlines

until utilization reaches around 70% which is the theoretical limit for RM scheduling. Then, as requested utilization begins to increase overall utilization begins to saturate at 100% and both task types start to increasingly miss deadlines. Note that both highly critical and lower criticality tasks are similar in miss rates. The reason is that fixed priority RM scheduling only considers the period of the task to assign priorities. The shorter the task period, the higher the priority. However, the importance of a task may not be proportional to how often that task runs, and some tasks may have a longer period but actually be more critical than a shorter period task.

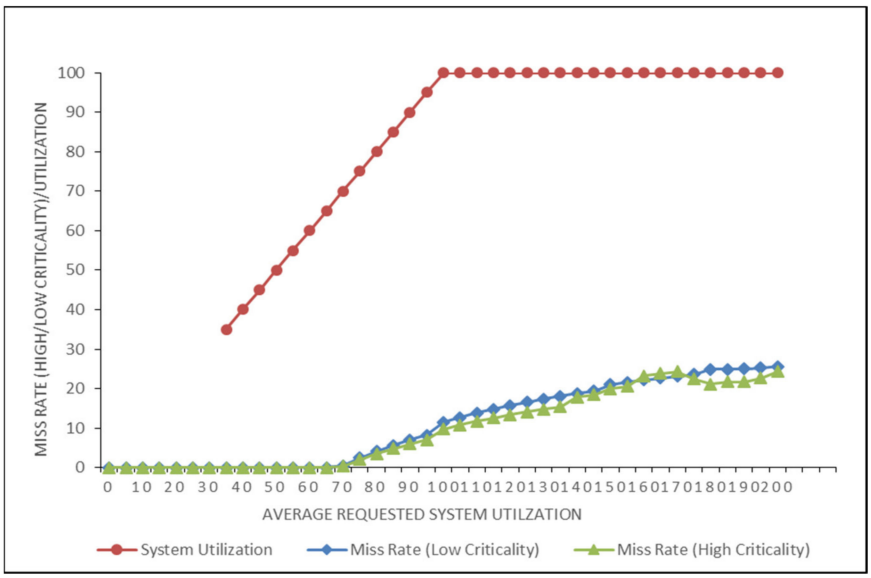


Figure 4. Static fixed priority scheduling.

Figure 5 illustrates the performance of our feedback-based scheduling framework as compared to the traditional fixed priority approach used in most embedded real-time applications. Notice in Figure 5 that around 70% of utilization tasks start to miss deadlines similar to what happens with the example presented in Figure 4. However, highly critical tasks do not start missing deadlines until around 95% total utilization while lower criticality tasks start missing deadlines at a higher rate than with fixed priority scheduling. This is due to the higher criticality tasks taking the bandwidth from the lower criticality tasks. Using our scheduling framework, we were able to effectively eliminate or limit the deadline miss rates for highly critical tasks. The next experiment analyzed how our feedback-based approach adapted to transient workload conditions. This is a case where the system is overloaded temporarily but eventually returns to a state where all tasks can be scheduled during nominal conditions. These types of transient events can occur because of hardware faults, unpredictable execution times or even software bugs. For this experiment, we modeled a sample representative task set, as described in Table 1. Tasks T_1, T_2, T_3 and T_4 are considered higher criticality since those tasks simulate reading sensors or controller actuators which are tasks that typically require low latency and a high degree of determinism. Task T_5 is the simulated image processing task and considered to be a lower criticality task. Task T_6 represents an aperiodic task which is used to represent asynchronous activity, such as processing a network packet or handling keyboard input.

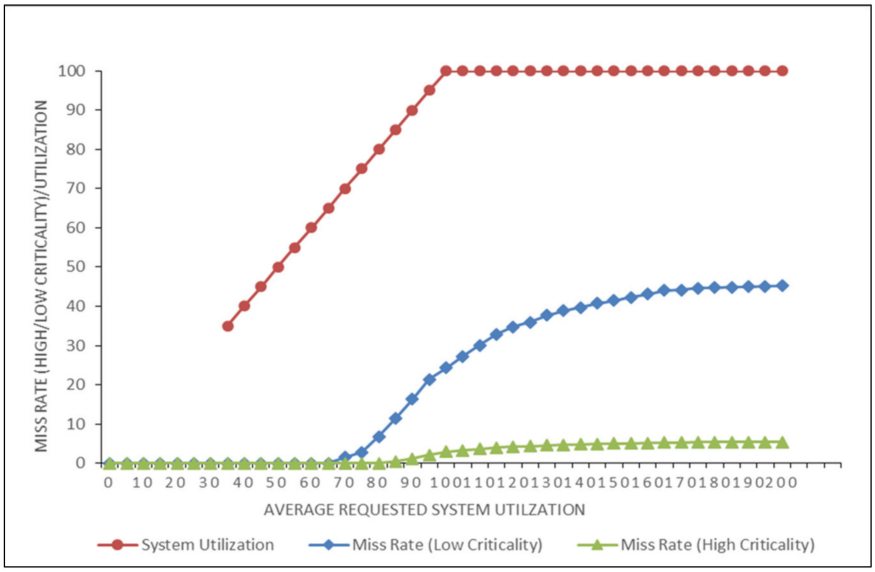


Figure 5. Adaptive feedback-based scheduling.

Table 1. Sample real-time task set.

Task	Period	Worst Case Execution Time (WCET)	Criticality Level
T_1	30	3	3
T_2	45	3	4
T_3	60	5	1
T_4	90	5	2
T_5	300	30	6
T_6	100	10	5

Similar to Figures 4 and 5, tasks can meet all their deadlines as long as overall utilization does not exceed 70%. At hyper-period 15 (a hyper-period is defined as the amount of time required for all tasks in the set to execute at least once), we model a transient overload of a requested utilization around 80% which results in tasks starting to miss their deadlines. It is at this point where the feedback scheduler utilizes the bandwidth actuator to reallocate bandwidth to higher criticality tasks. This is represented in Figure 6 where lower criticality task(s) are blocked from running in order to loan their bandwidth to the more critical tasks. Around hyper-period 21, utilization returns to a more nominal state where the bandwidth is restored to the lower criticality tasks. At hyper-period 42, another overload condition is modeled with a higher level of utilization. In this case, bandwidth approaches close to 95% which results in additional deadline miss rates because lower criticality task(s) are blocked in order to temporarily loan their bandwidth. However, it is important to note that after the feedback controller reaches a stable state, higher criticality tasks no longer miss their deadlines.

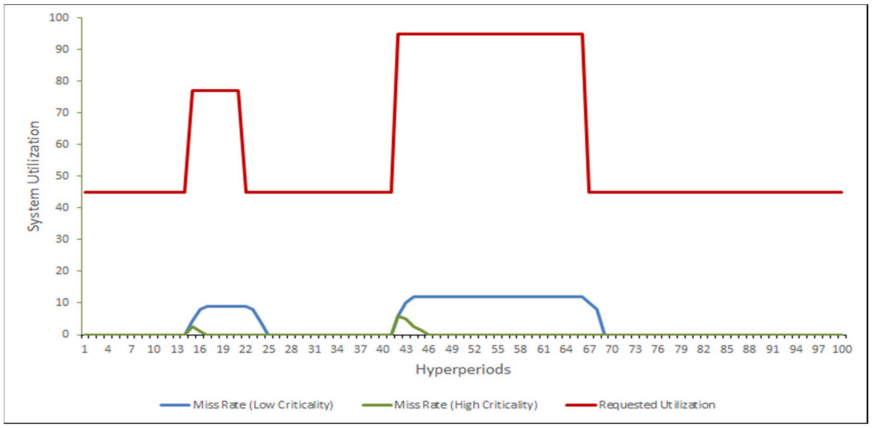


Figure 6. Feedback scheduler with transient overload.

6. Implementation

For our experimental setup, we implemented FC-RTS on an embedded SoC platform. We chose the Raspberry Pi 3 Model B+ for its Broadcom BCM28370B0, Cortex-A53 64-bit SoC processor, which is a common platform for Internet-of-Things (IoT) applications such as sensor monitoring and actuation control. The primary motivation for selecting the Raspberry Pi was to demonstrate that a commercial off the shelf (COTS) processor, not intended to provide deterministic behavior, could be used in a real-time DL environment without having to make any hardware changes. Since our work is primarily focused on the operating system scheduler, rather than the precise hardware specification, the Raspberry Pi 3 is computationally sufficient for the work described here.

As the Raspberry Pi is based upon Raspian which is a Linux-based operating system, a more deterministic real-time operating system is required. WindRiver's VxWorks 7.0 was selected because it is the de-facto industry standard operating system for real-time processing and WindRiver already provides a board support package (BSP) [34] for the platform. The BSP is the software support infrastructure required to run a RTOS on the Raspberry Pi. A picture of the targeted development platform is provided in Figure 7.

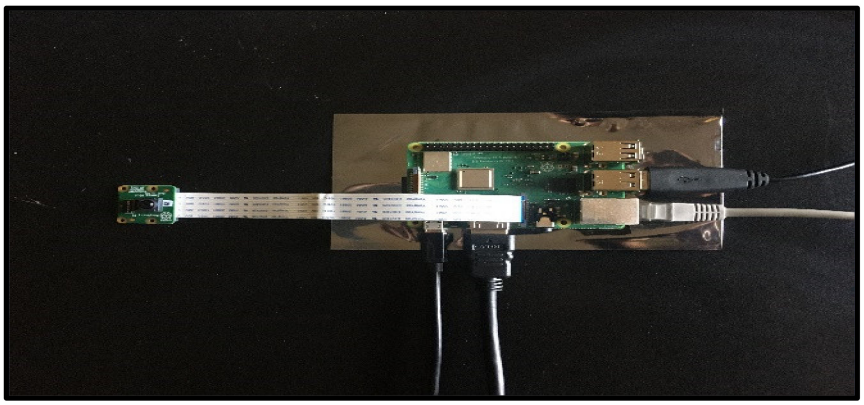


Figure 7. Target platform.

The native VxWorks scheduler can schedule tasks using either a preemptive priority-based or a round-robin scheduling policy. In VxWorks 6.x and greater, WindRiver introduced the concept of real-time processes (RTP) which more closely resemble processes in general purpose operating systems like Linux. Tasks in kernel mode or processes in RTP mode are scheduled in the same way. Processes are created with memory protection so kernel memory space, ISRs and direct hardware access are prohibited. Tasks that operate in kernel mode have full access to kernel resources and are not subject to the same limitations as processes in RTP mode.

We choose to implement our feedback-based scheduler in kernel mode because the overhead in RTPs is prohibitive and the scheduler needs access to the kernel resources for task management. Our scheduler was implemented on top of the native VxWorks scheduler as a type of extension or middleware that sits between the scheduler and the VxWorks native scheduler. The VxWorks RTOS provides functions to extend the capability so various kernel mechanisms can be customized to support feedback-based scheduling. For example, the scheduler can be extended with either a customized ready queue structure or to attach an interrupt handler that is executed at every clock tick.

The native VxWorks scheduler dispatches the highest priority task in the ready queue. Our approach utilizes the system called *tickAnnounceHookAdd( )* that is invoked at every tick interrupt and called before the native scheduler accesses the ready queue to dispatch the highest priority task. The ready queue is then manipulated by resuming a task *taskResume( )*, suspending a task *taskSuspend( )* or setting/changing priorities *taskPrioritySet( )*. The kernel's tick counter is also utilized to read *tickGet( )* and set *tickSet( )* as a means to manage the notion of time when the tick interrupt ISR is invoked.

The primary function of FC-RTS is to arrange either the periodic tasks or aperiodic tasks (that are serviced by a periodic server) in the ready queue at every period start. They are arranged according to their priority or criticality level depending upon whether overall task utilization exceeds available bandwidth. In order to support aperiodic tasks, the notion of a periodic server needs to be implemented. The periodic server itself is a task in VxWorks with its own task control block (TCB) and task event queue. Figure 8 illustrates the implementation of the required data structures needed to support a periodic server in VxWorks.

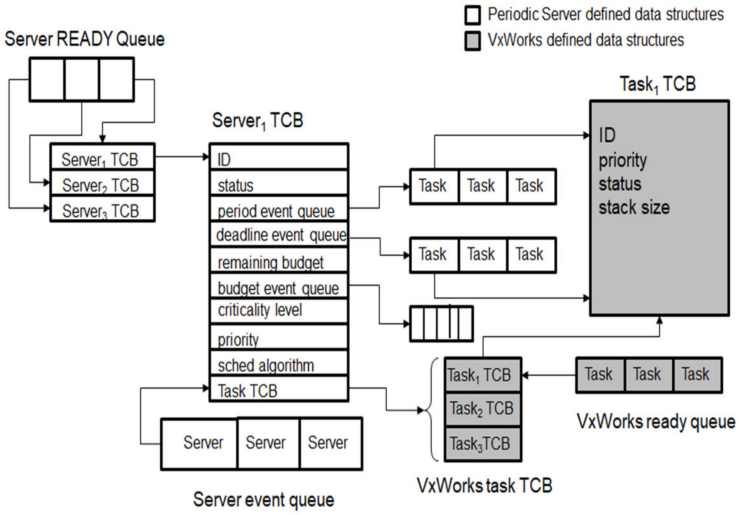


Figure 8. Implementation for periodic server.

The TCBs needed to support periodic serving in VxWorks are described as:

period_event_queue—is a reference to the server's event queue which contains the task period.

period—is the period of the server.

deadline_event_queue—is a reference to the server's task queue which holds the task deadline.

budget—is the server defined budget.

remaining_budget—is the current remaining budget of the server.

priority—is the server's priority.

criticality level—is the server's indication of criticality during overload conditions.

Task_TCB—is a list to the VxWorks TCB task list. It references those task TCBs that are associated with the server

In order to assess the applicability of our feedback-based real-time scheduling framework, we modeled a representative workload based upon common components found

in an edge IoT device. A task set was simulated to model the devices for the proposed development platform provided in Figure 9.

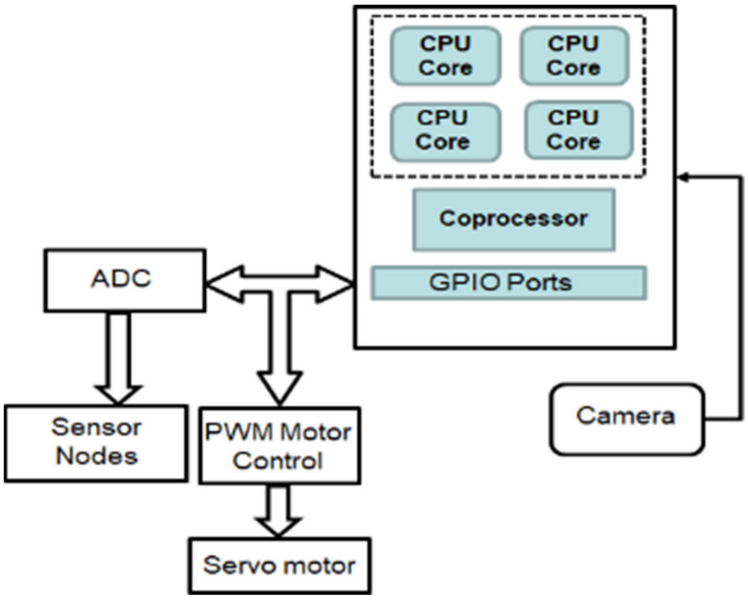


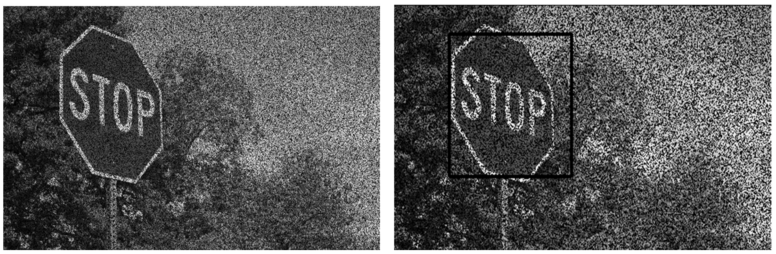
Figure 9. Simulated development platform.

Similar to the example task set provided in Table 1, we created two periodic tasks to command a simulated L298N motor driver for independently controlling two motors. Another group of tasks were created to simulate the monitoring of a series of sensor nodes. An image processing task was used to exercise the GPU coprocessor by processing sample images using OpenCV which had been ported to run on VxWorks by WindRiver Labs [35]. A couple other aperiodic tasks were created to simulate asynchronous events, such as network packet processing and keyboard processing. The purpose was to exercise the periodic server which manages the scheduling of the asynchronous tasks.

The actuator and sensor tasks were modeled with a simple spin loop that executed for the length of their respective WCET then blocked until the next invocation. The image processing task was created to perform simple object detection in two scenarios. One scenario was where the object could be detected without image enhancement (Figure 10a,b) and the other scenario was to simulate a faulty sensor that generated a noisy image requiring filtering (Figure 11a,b). The purpose was to model an average case processing scenario and a worst case processing scenario. For the aperiodic task, we created a periodic server to manage the simulation of the standard VxWorks console task to support the asynchronous events such as keyboard input or console output.

(**a**) (**b**)

Figure 10. (**a**) Original image; (**b**) Detected image.



(**a**) (**b**)

Figure 11. (**a**) Noisy image; (**b**) Detected image.

For the nominal and worst case scenarios we created the task set described in Table 2. For the nominal case, we modeled the simulated tasks to randomly vary their execution times between BCET and ACET, which are the best case execution time and average case execution times respectively. The image task was not modeled but instead performed the actual image processing using OpenCV. The BCET and ACET of the image task were averaged based upon numerous timing runs recorded when the image was processed without noise (Figure 10a). To determine m is critical in real-time systems, we analyzed the response times of the image process during 250 hyper-periods of the task set which is approximately 9 s. During the nominal case, system utilization ranged from $1.13 \leq U \leq 1.49$ which produces a correct schedule based upon the formula [36]:

$$U \leq m^2/(3m-2) \tag{5}$$

where $U = 1.6 \; for \; m = 4$. Refer to Figure 12 and notice that even while the system is clearly not overloaded, Raspbian Linux experiences significant jitter in the response times on the image processing task, even exceeding the response time thresholds on multiple occasions which is one indication that deadlines are being missed. Now compare that with our framework FC-RTS running in VxWorks, where the response time jitter has much less variation and no thresholds are exceeded indicating that no deadlines were missed.

Table 2. Simulated task set.

Task	Period	BCET	ACET	WCET	Criticality Level
Sensor_1	125	15	20	25	3
Sensor_2	125	15	20	25	4
Actuator_1	250	20	25	30	1
Actuator_2	250	20	25	30	2
Aperiodic	125	5	10	15	6
Image	360	250	295	358	**5**

For the worst case scenario case, we modeled the simulated tasks to randomly vary their execution times between ACET and WCET. The WCET of the image task was averaged over numerous timing runs recorded when the image was processed with noise and therefore requiring the more computational expensive filtering operation (Figure 11a). In this case, the utilization ranged from $1.49 \leq U \leq 1.75$ which indicates an overloaded system. Notice in Figure 13 that Raspbian Linux's increasing jitter results in a significant number of deadlines being missed while VxWorks FC-RTS does a much better job of minimizing the jitter. While FC-RTS does indicate that some deadlines are being missed, that is to be expected since the image task is considered to be less critical, making it susceptible to having to forfeit its bandwidth during overload events.

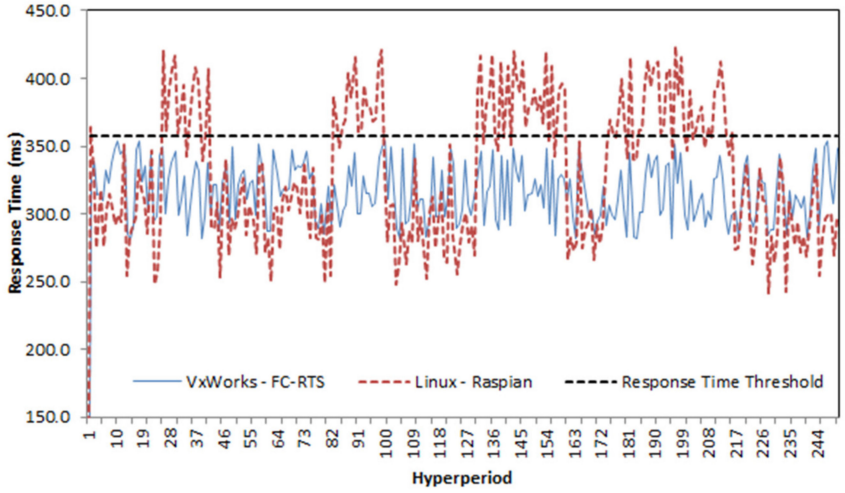


Figure 12. Response time nominal.

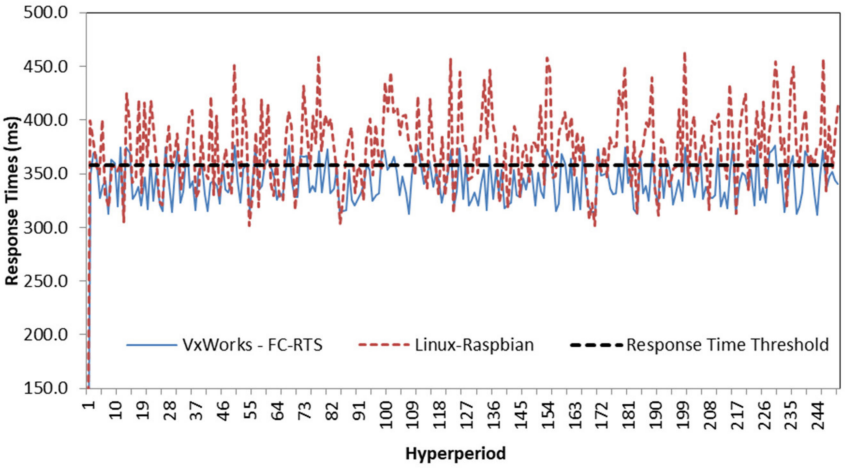


Figure 13. Response time worst case.

7. Conclusions and Future Work

In this paper, we explored some of the initial challenges in moving deep learning applications to edge devices. We explicitly looked at three areas: the real-time deterministic requirements of proposed edge devices and the limitations of resource-constrained embedded devices, the inherent unpredictability of deep learning applications and platforms as well as the problem of scheduling a set of tasks in a heterogeneous multiprocessor platform while still maintaining some level of real-time guarantees.

To provide real-time processing, we incorporated the industry standard RTOS VxWorks and enhanced it to incorporate the notion of a periodic server for managing asynchronous events, a leading cause of unpredictable behavior in a real-time system. To account for limited resources and unpredictability, we utilized a feedback control-based architecture to monitor the system and dynamically adjust resources to only the most critical tasks. For the problem of multiprocessor heterogeneous scheduling, we adopted synchronization protocols to provide an upper bound on the blocking times for cooperating tasks that use the coprocessor, resulting in predictable schedules. Using task set simulations, we were able to demonstrate through schedulability analysis that FC-RTS effectively manages critical resources and can still provide a high level of service, even when the system is severely overloaded. We were also able to demonstrate the practicability of this approach by incorporating it into a commercial RTOS on a Raspberry Pi processor. We further illustrated the usefulness of FC-RTS by running an actual image processing task using OpenCV, a common deep-learning application.

For future work, we plan on incorporating multiple coprocessor architectures in our schedulability analysis instead of just a single coprocessor. This work will incorporate an FPGA-based approach using the Zynq UltraScale+ RFSoC ZCU216 evaluation boards to evaluate versus a COTS-based processor approach such as the Raspberry Pi. We will also explore the integration of chips designed specifically for machine learning, such as the Google Coral and Intel Myriad processing units. Given the successful deployment of our framework on the Raspberry Pi, we anticipate the results will generalize to these more computationally-powerful target architectures, as well as to different machine learning applications outside of computer vision. We also plan on implementing and analyzing other scheduling protocols, such as EDF, to compare how it performs against RM. Finally, we plan on evaluating FC-RTS with the fully populated hard platform depicted in Figure 9 with actual hardware sensors and actuators instead of software-simulated ones.

Author Contributions: Conceptualization, T.S. and E.L.; methodology, T.S.; software, T.S.; writing—original draft preparation, T.S., E.E.-L., E.S. and E.L.; writing—review and editing, E.E.-L.; supervision, T.S., E.S. and E.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

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

References

1. Dhar, S.; Guo, J.; Liu, J.; Tripathi, S.; Kurup, U.; Shah, M. On-Device Machine Learning: An Algorithms and Learning Theory Perspective. *arXiv* **2019**, arXiv:1911.00623.
2. Sze, V.; Chen, Y.-H.; Yang, T.-J.; Emer, J.S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* **2017**, *105*, 2295–2329. [CrossRef]
3. Di Paolo Emilo, M. Bring Deep-Learning Inference to Embedded Applications. Available online: https://www.electronicdesign.com/industrial-automation/article/21808380/bring-deeplearning-inference-to-embedded-applications (accessed on 13 March 2021).
4. Conti, F.; Rusci, M.; Benini, L. The Memory Challenge in Ultra-Low Power Deep Learning. In *Time, Progress, Growth and Technology*; Springer International Publishing: Cham, Switzerland, 2020; pp. 323–349.
5. Payvand, M.; Fouda, M.E.; Kurdahi, F.; Eltawil, A.M.; Neftci, E.O. On-chip error-triggered learning of multi-layer memristive spiking neural networks. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2020**, *10*, 522–535. [CrossRef]
6. Peng, X.; Huang, S.; Jiang, H.; Lu, A.; Yu, S. DNN+ NeuroSim V2.0: An End-to-End Benchmarking Framework for Compute-in-Memory Accelerators for On-chip Training. *IEEE Trans. Comput. Des. Integr. Circuits Syst.* **2020**, *1*. [CrossRef]
7. Khan, M.A.; Kim, J. Toward Developing Efficient Conv-AE-Based Intrusion Detection System Using Heterogeneous Dataset. *Electronics* **2020**, *9*, 1771. [CrossRef]
8. Liu, L.; Lu, S.; Zhong, R.; Wu, B.; Yao, Y.; Zhang, Q.; Shi, W. Computing Systems for Autonomous Driving: State-of-the-Art and Challenges. *IEEE Internet Things J.* **2020**, *1*. [CrossRef]
9. Garey, M.R.; Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*; W. H. Freeman and Company: San Francisco, CA, USA, 1979.
10. Texas Instruments. *OMAP3 Platform*; Technical Report; Texas Instruments, Inc.: Dallas, TX, USA, 2009. Available online: http://www.ti.com/lit/ml/swpt024b/swpt024b.pdf (accessed on 13 March 2021).
11. Qualcomm, Inc. *Snapdragon*; Technical Report; Qualcomm: San Diego, CA, USA, 2011. Available online: http://www.qualcomm.com/media/documents/snaPdragons4-proces-sors-system-chip-solutions-new-mobile-age (accessed on 13 March 2021).
12. Baumgartl, R.; Hartig, H. Dsps as flexible multimedia accelerators. In Proceedings of the Second European DSP Education and Research Conference (EDRC'98), Paris, France, 6–11 September 1998.
13. Rajkumar, R. Synchronization in Multiple Processor Systems. In *Synchronization in Real-Time Systems: A Priority Inheritance Approach*; Kluwer Academic Publishers: New York, NY, USA, 1991; pp. 61–118.
14. Rajkumar, R.; Sha, L.; Lehoczky, J.P. Real-time synchronization protocols for multiprocessors. In Proceedings of the 1988 Real-Time Systems Symposium, Huntsville, AL, USA, 6–8 December 1988.
15. Hatcher, W.G.; Yu, W. A Survey of Deep Learning: Platforms, Applications and Emerging Research Trends. *IEEE Access* **2018**, *6*, 24411–24432. [CrossRef]
16. Haigh, K.Z.; Mackay, A.M.; Cook, M.R.; Lin, L.G. *Machine Learning for Embedded Systems: A Case Study*; BBN Technologies: Cambridge, MA, USA, 2015.
17. Lee, J.; Stanley, M.; Spanias, A.; Tepedelenlioglu, C. Integrating machine learning in embedded sensor systems for Internet-of-Things applications. In Proceedings of the 2016 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), Limassol, Cyprus, 12–14 December 2016; pp. 290–294.

18. Branco, S.; Ferreira, A.G.; Cabral, J. Machine Learning in Resource-Scarce Embedded Systems, FPGAs, and End-Devices: A Survey. *Electronics* **2019**, *8*, 1289. [CrossRef]
19. Ogden, S.S.; Guo, T. Characterizing the Deep Neural Networks Inference Performance of Mobile Applications. *arXiv* **2019**, arXiv:1909.04783.
20. Elliott, G.A.; Ward, B.C.; Anderson, J.H. GPUSync: A Framework for Real-Time GPU Management. In Proceedings of the 2013 IEEE 34th Real-Time Systems Symposium, Vancouver, BC, Canada, 3–6 December 2013; pp. 33–44.
21. Elliott, G.A.; Anderson, J.H. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Syst.* **2012**, *49*, 140–170. [CrossRef]
22. Chillet, D.; Eiche, A.; Pillement, S.; Sentieys, O. Real-time scheduling on heterogeneous system-on-chip architectures using an optimised artificial neural network. *J. Syst. Archit.* **2011**, *57*, 340–353. [CrossRef]
23. Franklin, G.F.; Powell, J.D.; Workman, M.L. *Digital Control of Dynamic Systems*, 3rd ed.; Addison-Wesley: Menlo Park, CA, USA, 1998.
24. Lu, C.; Stankovic, J.; Son, S.; Tao, G. Feedback control realtime scheduling: Framework, modeling, and algorithms. *Real-Time Syst.* **2002**, *23*, 85–126. [CrossRef]
25. Brandenburg, B.; Calandrino, J.; Block, A.; Leontyev, H.; Anderson, J. Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? In Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, St. Louis, MO, USA, 22–24 April 2008; pp. 342–353. [CrossRef]
26. Stankovic, J.; Lu, C.; Son, S.; Tao, G. The case for feedback control real-time scheduling. In Proceedings of the 11th Euromicro Conference on Real-Time Systems, Euromicro RTS'99, York, UK, 9–11 June 1999; pp. 11–20.
27. Lu, C.; Stankovic, J.A.; Tao, G.; Son, S.H. Design and evaluation of a feedback control EDF scheduling algorithm. In Proceedings of the 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054), Phoenix, AZ, USA, 1–3 December 1999; pp. 56–67.
28. Baker, T.P. Multiprocessor EDF and deadline monotonic schedulability analysis. In Proceedings of the 2003 International Symposium on System-on-Chip (IEEE Cat. No.03EX748), Cancun, Mexico, 5 December 2003; p. 120, 129.
29. Baker, T. An analysis of EDF schedulability on a multiprocessor. *IEEE Trans. Parallel Distrib. Syst.* **2005**, *16*, 760–768. [CrossRef]
30. Baker, T.P. *Further Improved Schedulability Analysis of EDF on Multiprocessor Platforms*; Technical Report TR-051001; Florida State University Department of Computer Science: Tallahassee, FL, USA, 2005.
31. Bertogna, M.; Cirinei, M.; Lipari, G. Improved Schedulability Analysis of EDF on Multiprocessor Platforms. In Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05), Balearic Islands, Spain, 6–8 July 2005.
32. Saewong, S.; Rajkumar, R.R. Cooperative scheduling of multiple resources. In Proceedings of the 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054), Phoenix, AZ, USA, 1–3 December 1999.
33. Gai, P.; Abeni, L.; Buttazzo, G. Multiprocessor DSP scheduling in system-on-a-chip architectures. In Proceedings of the 14th Euromicro Conference on Real-Time Systems, Euromicro RTS 2002, Vienna, Austria, 19–21 June 2002; pp. 231–238.
34. WindRiver Labs 2020. Available online: https://labs.windriver.com/vxworks-sdk (accessed on 13 March 2021).
35. WindRiver Labs 2019. Available online: https://labs.windriver.com/opencv (accessed on 13 March 2021).
36. Srinivasan, A.; Baruah, S. Deadline-based scheduling of periodic task systems on multiprocessors. *Inf. Process. Lett.* **2002**, *84*, 93–98. [CrossRef]