Alma Mater Studiorum · Università di Bologna

scuola di ingegneria e architettura

dipartimento di
ingegneria dell'energia elettrica e dell'informazione
"Guglielmo Marconi"
DEI

Corso di Laurea in Automation Engineering M

—

Tesi di Laurea
in
Hardware-Software Design Of Embedded Systems M

DESIGN AND PROTOTYPING OF A FREERTOS-BASED POWER CONTROL FIRMWARE FOR HPC PROCESSORS IN GAP8

Relatore:
Chiar.mo Prof. Luca Benini

Presentata da:
Giovanni Bambini

Correlatori:
Prof. Andrea Bartolini
Prof. Christian Conficoni
Dr. Ing. Simone Benatti

III Sessione
Anno Accademico 2018/2019

Abstract

With the increase in demand of efficiency and compute density in High Performance Computing sectors in the last years, the role of the Power Controller System (PCS) has growth in importance. Naif thermal capping and power capping solutions are not capable of delivering the desired energy-efficency on a wide set of working conditions and workloads for the processors in this market segment. For this reason advanced control policies are needed. This thesis aims to introduce a new generation of PCS based on high-performances multicore microcontrollers and the adoption of real-time software layers for more flexible and efficient firmware development.

Contents

List of Figures

List of Tables

List of Abbreviations

AV	Autonomous Vehicle
CPU	Central Processing Unit
DMA	Direct Memory Access
EPI	European Processor Initiative
FC	Fabric Controller
HPC	High Power Computing
L1	Level 1 Memory
L2	Level 2 Memory
OS	Operating System
PCS	Power Controller System
PULP	Parallel Ultra Low Power
PVT	Process, Voltage, Temperature Sensor
RTOS	Real-Time Operating System
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter
VR	Voltage Regulator

Chapter 1

Introduction

The last decade witnessed the end of the Moore's cycle (transistor dimensions scaled down by 0.7x every technology generation), at which the microprocessor industry answered with a progressive increase of power density by stacking multiple cores on the same die. This requires that each computing unit must operate not only at its **maximum performances**, but also at its **maximum efficiency**. Furthermore, in the last decade the demand of data center has dramatically increased due to not only the grow of scientific research computation needs, but also to the huge spread of internet and web applications. Data centers require massive computing and fast data access at an affordable cost, with a limited power budget and under a secure environment. To operate in this power and thermally constrained conditions, processors for the data center market require to **dynamically adjust** their performance to adapt to the executed software workload and environmental conditions.

High computing performances is becoming an established paradigm in several sectors, such as **Advanced Driver Assistance Systems** (ADAS) and **Autonomous Vehicle** (AV). In these systems generally the computing units are heterogeneous to provide efficient computation toward specific functions and the power capping is necessary not only to preserve the lifetime and the performances of the chip, but also to preserve the battery time

of the overall system. Such systems also require high security and safety constraints. Therefore, the power management holds an important role and requires a degree of flexibility to meet different power requirements of diversified and complex systems.

In these market branches a dynamic management of power consumption has to be achieved to handle high performances in a context where these performances cannot be sustained in static conditions for an extended period of time. Power consumption is an important element since it impacts significantly in the operating cost and it is a thermal limit to the performance and the lifetime of the processor. With these considerations the importance of the power management is clearly a key element in the processor.

The **Power Controller** (or **Power Controller System** PCS) is an embedded control unit integrated in the processor which handles the control of the operation point (i.e. frequency, voltage, ...) of operation of each computing unit of the chip and therefore the power consumption of the whole board. The objective of the PCS is to deliver the minimum necessary power to let each core execute the given operation at the given frequency within the thermal limit imposed. This is achieved by hard real-time control on the temperature of the chip and by predicting the power required by each type of workload, estimating also the power loss across the various components of the board.

The state-of-the-art power controllers present limitations when faced with complex control conditions with an increasing amount of heterogeneous computing units to be controlled due to the limited computing capability that doesn't allow the utilization of advanced algorithms. This limitations are worsened in that the PCS is developed *"bare metal"* without any software layer, increasing the complexity and reducing the flexibility[1].

Lately, in these years a tendency has emerged to further optimize power consumption by linking the activity flow of applications running in the processor to power management policies[2]. Although a standard has not yet

[1] The Intel PCU and IBM OCC are analized in [7] and [8]

[2] As described in [5].

been defined, these tendencies have to be taken into account in the development of the PCS to meet flexibility and adaptability for future requirements. This means that the PCS has to interact with those applications adding a complexity layer in the development of the PCS firmware.

The work of this thesis is part of a open-source project under development at the UNIBO Multitherman Laboratory in Bologna, to design and develop an innovative Power Controller aimed at HPC processor and AV market. In particular the purpose of this thesis is to study, develop and test the firmware of the power controller. The firmware should implement advanced control features, should be able to manage a large number of heterogeneous computing units, should have the flexibility to interact with running application and should be easy to update and upgrade. These requirements have been taken into account by choosing an open-source low-power microcontroller with accelerators as PCS and developing the firmware over a RTOS layer. To test the implementation I developed the firmware on a reference platform (GAP8). To test the functionality I designed a hardware in the loop simulation framework where a STM32 is used for the emulation of the power and thermal dissipation plant while the GAP8 runs the designed firmware.

The key results I have achieved in this thesis are:

1. To the best of the author knowledge, this is the first power controller system firmware for **RISC-V** based microcontroller developed on **FreeRTOS**.

2. The firmware is capable of **CPU-level power capping** and **core-level thermal capping**. The power-capping consists of a model-based open-loop controller coupled with an online model adaptation algorithm. The core-level thermal capping is based on a closed-loop PID controller. The firmware developed can handle, in a single RISC-V core running at 250MHz, up to 63 cores units, with the thermal-capping and the power capping controls executed every 500us and the model adaptation algorithm executed every 1ms (considering to occupy only 50% of the control period for computation).

3. The **implementation** and **testing** of the PCS on a commercial RISC-V based SoC, namely Greenwaves GAP8.

4. The implementation of a **Hardware in the Loop** simulation framework with the GAP8 firmware implementation and a STM32 board used for emulating the physical plant. The STM32 board emulates 9 cores and can execute more than 300 iterations of the model every 100ms. In the same 100ms the GAP8 firmware is capable of emulating the 500us control step.

5. Verification with the HiL framework that the controller is working. I measure for a synthetic test that the temperatures stay in a 3°C range of the thermal limit and the power budget is respected.

6. The characterization of FreeRTOS overheads in terms of task switching latency on the RISC-V (GAP8) architecture. In GAP8 I measured for the Notification system an overhead of 1319 cycles to activate a task and 1233 cycles to block a task and perform a context switch.

1.1 The European Processor Initiative - EPI EU H2020 Project

The European Processor Initiative (EPI) is a project that aims to deliver a high-performance, low-power processor to achieve independence in HPC and to advance the research in the field within EU borders by creating competences in high-end chip design. The project gathers 27 partners from 10 European countries and the strategic plan is to support the next generation of computing and data infrastructures. The mission is to use this processor to build an Exascale machine by 2023, but also to deliver a flexible product to go beyond HPC market addressing the needs of European industry (mostly autonomous car market).

According to the roadmap of the project, the first version of the EPI

processor will be completed by the end of 2021. This version is called RheaR1 and is based on ARM ZEUS core combined with other accelerators specialized for different applications. The launch of the product will be between 2021 and 2022; the second version of EPI is expected to be completed in 2023 and it will be based on ARM TITAN[3]. The University of Bologna in conjunction with the Institute of Technology in Zurich (ETH Zürich) is assigned to develop the Power Control Unit of RheaR1.

According to [1] a possible PCS architecture can be derived from open-source design of microcontrollers as the one presented in section 2.5. In the work of this thesis the firmware of a PCS is developed over a PULPissimo-derived microcontroller to deploy an advanced control policy. As a HPC case study and as an application target for the PCS that is being developed, the EPI processor is considered the chip to be controlled.

[3]european-processor-initiative.eu/project/epi/

Chapter 2

The Control Structure

In this chapter is presented the assumed interface for the PCS I/O, based on the interfaces of HPC processors on the market and on the interface of RheaR1. A hardware platform is also proposed as the physical implementation of the PCS.

2.1 The Processor Plant

There are many different processors and models to describe them. Generally processors include:

- **Temperature sensors**, required to perform thermal control;
- **Frequency/Voltage actuators** to impose the output computed by the power controller system, required to carry out a control action;
- other sensors and inputs may also be present, such as sensors to measure the power consumption.

In this thesis we consider a processor plant as shown in Figure 2.1.

The thermal sensor is a PVT sensor which measure Process, Voltage and Temperature. There is a thermal sensor for each core of the chip. The control action is performed by modulating the voltage and the frequency applied to

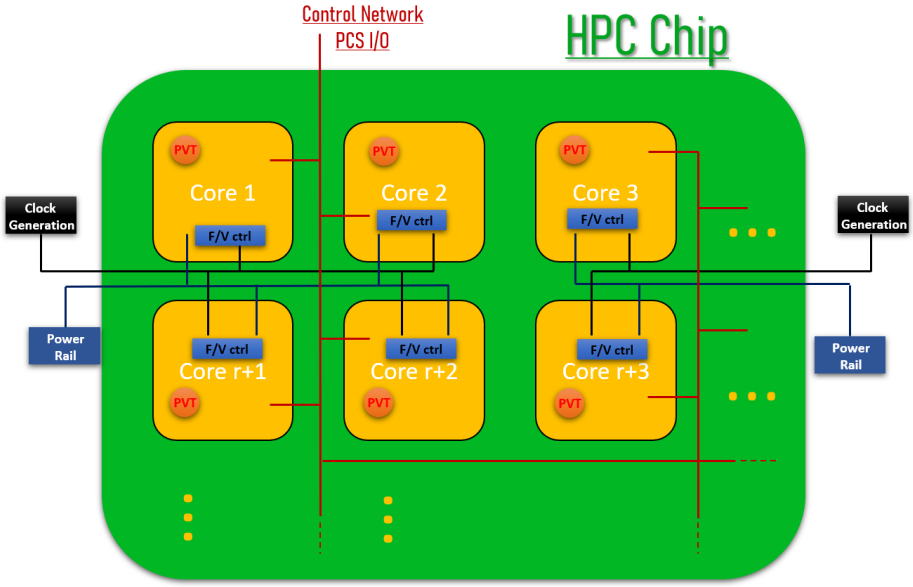

Figure 2.1: The supposed Block-Scheme of the Processor Plant

the cores. There are several techniques differing by implementation cost and chip area, timing performances, value accuracy and power efficiency. Furthermore, the voltage and frequency outputs generated by the PCS are related by a specific function that strongly depend on the model of the chip. For this project a simple **Dynamic Frequency Scaling** (DFS) for each core is implemented. To adapt the developed PCS in a specific system, a simple software output generation module can be added. Particular attention should be paid to the effects that different outputs control value can exhibit on the overall control performances. A **Control Network** implemented through a ring bus, connects the PCS with the chip cores and it transmits both inputs and outputs.

To measure the absorbed power of the chip, sensors on the **Voltage Regulators** (VR) of the power rails can be exploited. The frequency at which the sensors can be queried to obtain the values is assumed of $1ms$ for our project.

2.2 The Power Controller System

As described in the Introduction, the Power Controller System (PCS) is an important component of the processor. It interfaces to all the physical sensors and actuators, the OS and the user's applications. The main objective of the power controller is to periodically read the status of the computing elements and sets the operating point (Voltage and Frequency) according to the power management policies. In this work the power controller carries out a hard **real-time thermal control** on the cores and a **power capping control** of the whole board.

In addition to those metrics, the power controller may read the power consumption of the board and may receive from the OS parametric requirements in terms of performance level (Target Frequency), power budget, and other information such as the characteristics of the workload to be executed. On the basis of these parameters, the power management policy determines the best operating point while ensuring the thermal stability, the power budget and the application constraints.

This output information flow can be divided into in-band and out-of-band services. The in-band services are delivered to the applications and operating systems running on the processing elements of the chip, the out-of-band services are delivered to the system administrator and system management tools through the Board Management Controller (BMC). These services consist of out-of-band power telemetry, system-level power capping, reliability and serviceability.

2.2.1 The Proposed Hardware-Software Interface

In this section the interface between the PCS firmware and the plant hardware is described. The interface include the PCS hardware inputs and outputs and the services that the PCS provides to the applications running on the processor chip and to the system administrator. In particular:

- Inputs:

- **Target frequency** per core
- **Power cap** of the board and power cap of board sections
- **Expected workload** possibly over a given horizon
- Additional **performance parameters**, such as the Binding Constraint vector

- Outputs:
  - **Telemetry data**: mean and peak frequency, voltage and temperature per core
  - **Errors** relative to the execution of the power control application, either software, relative to the measurements or to the communications
  - Additional desired parameters, such as the causes of frequency reduction with respect to target values or the parameters describing model identification
  - **Out-of-Band** additional parameters as reliability and serviceability

2.3 The Control Structure and Policies

The main goal of the power controller system is to perform thermal and power management of the considered computing platform. The thermal management is stricter since a failed control may result into hardware damage or the shut down of the system (which will result in a crash of the running application). This negative effect must be avoided considering the importance of safety in applications such as the AV and to not impact performances and costs in HPC server environment.

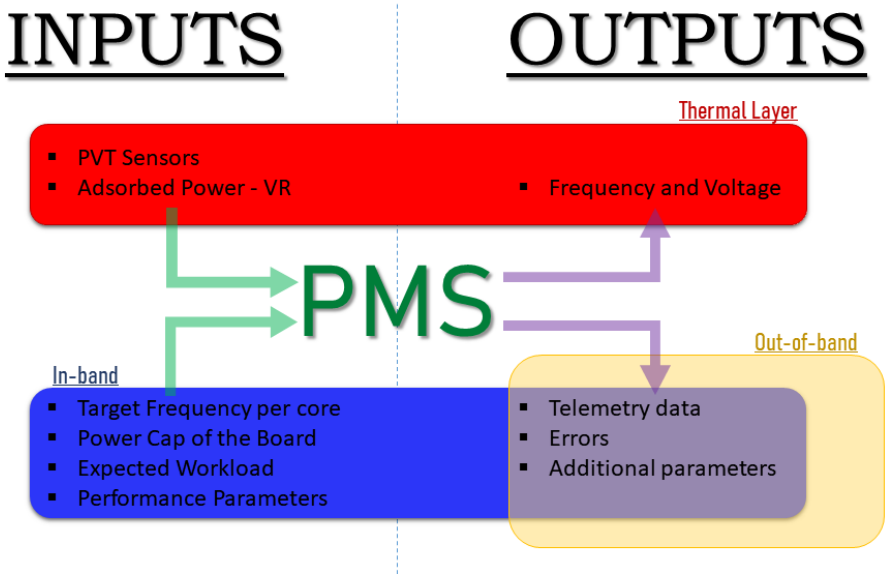

Figure 2.2: Hardware-Software Interface

2.3.1 Control Problem Formulation

A straight control procedure[1] is to preserve the given target frequencies of the computing unit while keeping the entire system within its thermal and power limits. Specifically, the system has to be kept within given thermal limits and it has a total power limit (inequality constraint), and its computing units are bounded by a minimum and maximum frequency and voltage, which can conceivably be specific and variable per unit.

$$T_{Si}(x,t) \leq T_{CRIT}, \qquad \forall x \in V_{Si}, t \in \mathbb{R}_{\geq 0}, \tag{2.1}$$

$$\sum_{i=1}^{n_s} P_i \leq P_{budget} \tag{2.2}$$

$$f_{min} \leq f_i \leq f_{max}, \qquad i = 1, \ldots n_s. \tag{2.3}$$

$$V_{dd\_min} \leq V_{dd\_i} \leq V_{dd\_max}, \qquad i = 1, \ldots n_s. \tag{2.4}$$

[1]The model definition and the Control structure described in this chapter are presented by Christian Conficoni in [2]

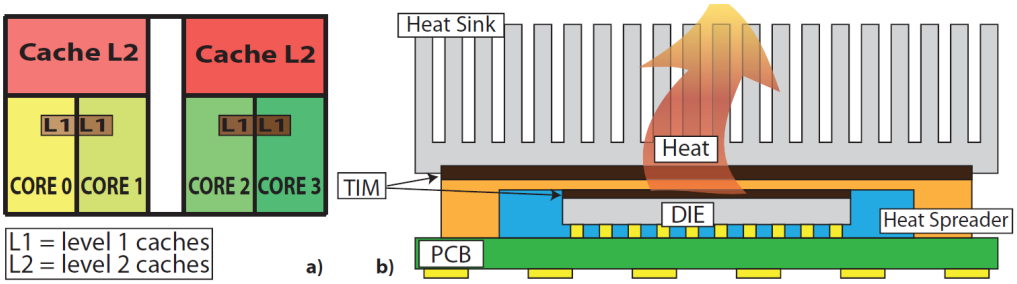

Figure 2.3: Considered Chip Thermal Structure

where T_{CRIT} is a critical temperature threshold not to be exceeded for the silicon device.

Other advanced constraints may be added. For our control structure two further constraints are deployed: a spatial constraint on the power consumption per chip section, differentiated by the connection to different external powerlines, and cores binding constraints, to exploit the parallel computation of HPC.

$$\sum_{i=1}^{n_j} P_i \leq P_{quad\_j\_max}, j = 1, \ldots, n_{quad} \tag{2.5}$$

Since HPC often executes parallel operations which mostly consist of executing the same operation on different data by deploying operations onto multiple cores, it is expected that all the cores will complete their computations in the same time interval. This characteristic can be exploited to reduce the power consumption of the system without affecting the performances, by aligning the frequencies of each core of the group that is executing parallel computations, to the minimum frequency at which these cores are running.

$$f_{ti} = f_{tj} \text{ if } B(i,j) = 1 \tag{2.6}$$

where $B(i,k)$ is a $n_s \times n_s$ symmetric matrix whose entries are 0 if the cores are not subject to binding constraints, 1 otherwise.

2.3.2 The Chip Model Definition

The basic architecture considered is showed in Figure 2.3 and it represents a standard planar silicon SoC layout with a passive copper heat-spreader placed over it. The physic heat-equations that describe the heat flow are:

$$\rho_{Si} c_{Si} \frac{\partial T_{Si}(x,t)}{\partial t} = k_{Si} \nabla^2 T_{Si}(x,t) + q(x,t), \qquad x \in V_{Si}, t \in \mathbb{R}_{\geq 0} \tag{2.7}$$

$$\rho_{Cu} c_{Cu} \frac{\partial T_{Cu}(x,t)}{\partial t} = k_{Cu} \nabla^2 T_{Cu}(x,t), \qquad x \in V_{Cu}, t \in \mathbb{R}_{\geq 0}, \tag{2.8}$$

with the following boundary and initial conditions:

$$\begin{aligned}
&T_{Si}(x,0) = T_{Si,0}(x), \qquad \forall x \in V_{Si} \\
&T_{Si}(x,t) = T_{\partial Si}(x,t), \qquad \forall x \in \partial V_{Si} \backslash \partial V_{Cu} \\
&T_{Cu}(x,0) = T_{Cu,0}(x), \qquad \forall x \in V_{Cu} \\
&T_{Cu}(x,t) = T_{\partial Cu}(x,t), \qquad \forall x \in \partial V_{Cu} \backslash \partial V_{Si} \\
&T_{Si}(x,t) = T_{Cu}(x,t) \\
&k_{Si} \nabla T_{Si}(x,t) = k_{Cu} \nabla T_{Cu}(x,t), \qquad \forall x \in \partial V_{Si} \cap \partial V_{Cu}, t \in \mathbb{R}_{\geq 0},
\end{aligned} \tag{2.9}$$

where $T_{Si}(\cdot)$ and $T_{Cu}(\cdot)$ are the silicon chip and copper heat-spreader temperatures, defined in the volumes V_{Si}, V_{Cu}, respectively. $q(\cdot) \geq 0$ is the volumetric thermal power generated by internal sources, while ρ_{Si}, ρ_{Cu}, c_{Si}, c_{Cu}, k_{Si}, k_{Cu} are the density, specific heat and thermal conductivity of the two materials.

The inputs to this system are the power supplied to the chip and the ambient temperature. The power is given by the sum of the power supplied to each core plus an unknown component due to dissipation and other chip functionalities. The power supplied to each core can be modelled with a non-linear map $h(\cdot)$ taking as inputs the Voltage and frequency at which the core is operating and the workload that the core is executing. In this model the unknown component is considered negligible and treated inside the non-linear map with the workload parameter. The core can perform a variety of

different operations (load and store, SVE, indexing, branches, logical, ...) and each one of them consumes different amounts of power.

$$P_i(t) = h_i(f_i(t), V_{ddi}(t), w_i(t)) = \int_{V_{s,i}} q(x,t)dV \tag{2.10}$$

The non-linear map should be invertible in an open connected interval $\mathcal{F}$ to ensure that given the power consumption and knowing the workload, a unique pair of frequency and supply voltages producing that power can be found. There are several models that describe this non-linear mapping, they are strongly related to the type of computing platform that is being controlled and generally they are obtained through empirical identification. A general HPC model will be used in this case, but adopting a different model has no significant modifications in the proposed control strategy.

$$P_i = P_{stat} + P_d = I_{cc}V_{ddi} + w_i f_i V_{ddi}^2 \tag{2.11}$$

To obtain a model to be used conveniently for control design purposes, ODEs can be obtained from the PDEs 2.7 and 2.8 by employing Euler discretization of the derivatives with respect to the spatial coordinates x. This operation introduces some degree of approximation but it can be shown that feasibility of the thermal constraints can be guaranteed by focusing on the hot spots of the die[2].

n_c finite elements (cells) are selected, each related to a power source, then two thermal states, one associated to the local temperature in the silicon die and the other to the local temperature of the metallic heat spreader, are associated to each element. The ODE associated to a generic element becomes:

[2]From [2]

$$\begin{aligned}\dot{T}_{Si,i}&=\frac{P_i}{C_{Si,i}}+\frac{T_{Cu,i}-T_{Si,i}}{C_{Si,i}R_{Si,v,i}}+\sum_{j\in\mathcal{N}\{i\}}\frac{T_{Si,j}-T_{Si,i}}{C_{Si,i}R_{Si,h,ij}}\\ \dot{T}_{Cu,i}&=\frac{T_{Si,i}-T_{Cu,i}}{C_{Cu,i}R_{Si,v,i}}+\frac{T_{amb}-T_{Cu,i}}{C_{Si,i}R_{Cu,v,i}}+\sum_{j\in\mathcal{N}\{i\}}\frac{T_{Cu,j}-T_{Cu,i}}{C_{Cu,i}R_{Cu,h,ij}},\end{aligned} \tag{2.12}$$

where $C_{Si,i}$, $C_{Cu,i}$ are the thermal capacitances, $R_{Si,v,i}$, $R_{Cu,v,i}$, $R_{Si,h,ij}$, $R_{Cu,h,ij}$, $j \in \mathcal{N}\{i\}$ are the thermal resistances and T_{amb} is the ambient temperature. These lumped parameters are obtained from spatial discretization procedure.

For simplicity the spatial discretization is achieved by relating each finite element with an entire core. More complex modelling can be accomplished by increasing the number of finite elements, but consider that the number of states will increase significantly. Collecting all the temperatures in a vector $T = (T_{Si,1}, T_{Cu,1}, \ldots, T_{Si,n_s}, T_{Cu,n_s})^T$, the final State Space model is rewritten as:

$$\dot{T} = AT + B_pP + B_ET_{amb}, \qquad T_{Si} = CT, \tag{2.13}$$

2.3.3 The Proposed Control Solution

The optimization problem can be mathematically formulated as:

$$\begin{aligned}&\min_f \sum_{j=0}^{N-1} |f_t(k+j|k) - f(k+j|k)|_R^2\\ &\text{subject to:}\\ &T_{Si,i}(k+j+1|k) \leq T_{CRITi} \quad \forall i=1,\ldots,n_s \quad \forall j=0,\ldots,N\\ &f_{min}\mathbf{1}_{n_s} \preceq f(k+j|k) \preceq f_{max}\mathbf{1}_{n_s}, \quad \forall j=0,\ldots,N-1\\ &V_{dd\_min}\mathbf{1}_{n_s} \preceq V_{dd}(k+j|k) \preceq V_{dd\_max}\mathbf{1}_{n_s}, \quad \forall j=0,\ldots,N-1\\ &f_{ti} = f_{tj} \text{ if } B(i,j) = 1\\ &\sum_{i=1}^{n_s} P_i \leq P_{budget}\end{aligned} \tag{2.14}$$

where k denotes the (discrete) time variable, N is the number of steps assumed as the horizon and n_s is the number of elements/cores.
$f = (f_1, \ldots, f_{n_s})^T$ are the decision variables of the problem, i.e. the frequencies to be assigned to the cores by solving the problem above with the set of target frequencies $f_t = (f_{t,1}, \ldots, f_{t,n_s})^T$. $R \in \mathbb{R}^{n_s \times n_s}$ is a symmetric positive definite weight matrix and $|\cdot|_R$ is the norm induced by it. T_{CRITi} are the critical temperatures on the n_s points selected to represent the thermal state of the die. Finally, $\mathbf{1}_{n_s}$ denotes a column vector of ones of the same dimensions of f and $\preceq$ denotes an element-wise inequality.

The optimization problem described by 2.14 is a convex Quadratic Problem, and in principle can be solved by means of efficient numerical iterative optimization algorithm[3]. However, this optimization procedure cannot be performed offline to produce the optimal sequence of frequencies/voltages because:

- the workload is not available for the entire time horizon
- the optimization relies on the exact knowledge of the chip model; any mismatch would produce prediction errors which could lead to sub-optimal or even unfeasible solutions.

and even if it can be solved on-line, for example by deploying receding horizon approaches like in the Model Predictive Control Framework, carrying out the optimization is computationally heavy and unattainable to be performed for a large amount of finite elements on a microcontroller (the Power Controller).

An advanced approach could employ distributed algorithm, but such approach is not straightforward mainly due to the coupling constraints, and the necessity of a state-observer to obtain the heat-spreader temperature which is generally not available. For these reasons a first approach is based on a simple sub-optimal algorithm.

To match the difference in requirements between thermal and power limit control a two-layer algorithm is proposed, which is composed by a:

[3]From [2]

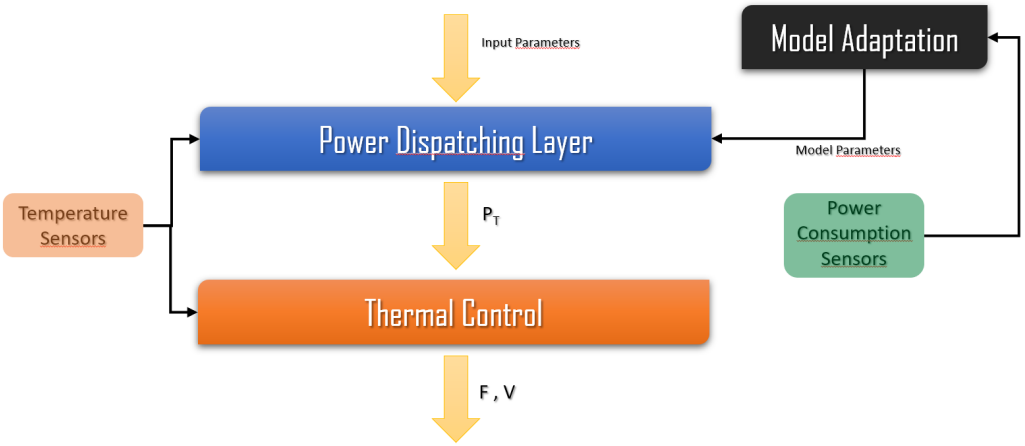

Figure 2.4: Hierarchical Control Structure

Power Dispatching Layer which is in charge of distributing the requested power (computed through the target input frequency and the chip power model) among cores, according to power limitations and other constraints like the frequency bindings

Thermal Regulator which allocates core frequencies (and associated power) according to values computed by the Power Dispatching Layer in order to meet temperature constraints related to the cores actual thermal state.

The scheme of this hierarchical structure is shown in Figure 2.4.

2.3.4 The Power Dispatching Layer

The first step that the Power Dispatching Layer has to perform is to estimate the power consumption of the board based on the parameters received by the OS. Based on the model 2.11 the power for each core can be computed knowing the target frequency, the corresponding voltage and workload; then the power consumption of the chip is obtained by summing all the core power consumptions.

The computed total power is compared to the given power budget.

$$\Delta = \sum_{i=1}^{n_s} P_{ti} - P_{budget} \tag{2.15}$$

If $\Delta > 0$ an algorithm should be deployed to reduce the power consumption and stand within the boundaries. A simple heuristic algorithm is proposed to achieve power reduction based on the thermal room of each core ($T_{CRITi} - T_{Si,i}$). The frequency cutting will not be uniform but proportional to the core temperature (the hotter the core is, the more its power is cut). The binding constraint is enforced in the procedure to obtain advanced policies. In particular:

1. Compute weights to reduce cores power :

$$\alpha_i = \frac{1}{T_{CRITi} - T_{Si,i}} \tag{2.16}$$

2. Evaluate and enforce binding constraint directly on α_i:

$$\alpha_i = max\{\alpha_i, \alpha_j\} \text{ if } B(i,j) = 1 \tag{2.17}$$

3. Normalize the weights α_i as:

$$\bar{\alpha}_i = \frac{\alpha_i}{\sum_{i=1}^{n_s} \alpha_i} \tag{2.18}$$

then cut cores powers as follows

$$P_{ti}^* = P_t - \bar{\alpha}_i \Delta \tag{2.19}$$

2.3.5 The Thermal Regulator

In the thermal Controller the limitation of the output is imposed based on the thermal states of each core and considering the thermal dynamics.

To perform this control action a PI regulator for each core is proposed. The PI takes into account the dynamics of the system, is computationally light and does not strongly rely on accurate model parameters identification. Furthermore each PI is distinct and independent of others, allowing flexibility and decentralization in the code implementation.

The philosophy of the thermal layer is to reduce the power indication received by the Power Dispatcher layer only when the temperature of the core is approaching its critical value. To this aim the reference input of the regulator is set to be a constant value computed as the temperature limit minus a positive margin. The error produced will reflect the amount of control action to be applied to reach this constant value. The output of the regulator is saturated between $[-p, 0]$ obtaining a null control action if the temperature is less than the reference input value, and a negative control action in case the error is negative. This control action is subtracted to the target power received by the Power Dispatch layer to obtain the final output.

Even if it is convoluted, applying this control action allows to not reduce the frequency under the temperature threshold and still taking into account the thermal dynamics when the control action is applied. To obtain good performances and requested safety, the choice of the positive margin is crucial, considering also the delay originated from the non-linear saturation action performed in the last step.

The obtained power output is then converted into frequency and Voltage values by inverting the map 2.10. An additional step to further enforce the binding constraints can be added on the final frequency and voltage, but the choice depends on the trade-off between the gained power consumption efficiency and the additional computation time introduced.

2.3.6 Power Model Adaptation

In both layers the power model of the plant carries out a fundamental role. Although a perfect knowledge of the chip model has been assumed, in a realistic scenario there are parametric uncertainties and additive signal noise

which make this assumption not true.

To overcome this effect conservative bounds and parameters can be deployed, increasing the optimality gap with respect to the full information case. An alternative control choice is to update the model parameter while the system is running, performing a model identification.

The proposed technique is to apply a standard Least Mean Square (LMS) on the parameters of the model by comparing the sum of the computed power through the model and the actual measured consumed power given by the VR.

$$\theta(k+1) = \theta(k) - \nabla_\theta \left(\sum_{i=1}^{n_j} h_i(f_i(k), V_{ddi}(k), w_i(k) \right) \tag{2.20}$$

where θ is the vector of coefficients.

2.4 Control Requirements

The control structure proposed requires:

- a hard real-time deadline regarding the output of the control values (frequencies and voltages). This also implies that the computation of the power controller has to be time-bounded and executed with a precise periodicity
- reliable and high-frequency inputs for both sensors and parameters
- to communicate with multiple agents without interfering with the control action
- different time conditions for the layers and the model adaptation

2.5 The PCS Hardware Implementation

We considered as a target implementation of the PCS the architecture of the PULPissimo platform. PULP stands for Parallel Ultra Low Power and

it is an open-source project started as a joint effort between the Integrated Systems Laboratory (IIS) of ETH Zürich and Energy-efficient Embedded Systems (EEES) group of the University of Bologna in 2013 to explore new and efficient architectures for ultra-low-power processing[4].

PULPissimo is the advanced version of the PULPino basic microcontroller, featuring the presence of the logarithmic interconnect between the RISC-V core and the memory subsystem allowing multiple access ports. These allow to integrate a subsystem called micro DMA (uDMA) which is able to copy data directly between peripherals and memory, as well as optional accelerators.

The choice is supported by the PULPissimo modularity, power efficiency and multicores structure that allow to tune the power control performances based on the chip to be managed. The fact that is also open-source matches the open-source final goal of our project too.

The architecture is hierarchical and demand-driven, enabling efficient and complex control operation, and it combines a fabric controller (FC) core, which can be considered as a classic microcontroller, and a cluster of 8 cores with an architecture optimized for the execution of vectorized and parallelized algorithms. These are connected through the logarithmic interconnect. The multicore design of the PCS allows to implement and support more advanced and computationally complex control policies, for larger number of cores, while the efficiency of the PULP-based chip keeps the power consumption low[5].

2.5.1 Cores

All the eight cores of the cluster share the RV32IMFCXpulp instruction set architecture, while the fabric controller can be configured as either RV32IMC or RV32IMFCXpulp. The I (integer), C (compressed instruction), M (Multiplication and division) and a portion of the supervisor

[4]pulp-platform.org/projectinfo.html

[5]The following description of the Power Controller is extracted from [1]

ISA subsets are supported. These standard instruction sets are extended with specific instructions to optimize the performance of signal processing and machine learning algorithms. For more details on the instruction set architecture, see `https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf` and RISC-V User Manual: `github.com/openhwgroup/cv32e40p/blob/master/doc/user_manual.doc`.

2.5.2 Memory

There are two different levels of internal memory: a larger level 2 (L2) area of 512kiB which is accessible by all processors and DMA units and a smaller (Tightly Coupled Device Memory - TCDM) level 1 (L1) area shared by all the cluster cores (128kiB). The cluster-level 1 memory is connected to the cluster cores via a logarithmic interconnect that is sized to provide single-cycle access in 98% of cases and is designed to minimize conflicts and it supports atomic functionality to provide determinism in data access.

The instruction caches of the FC (1kiB) and cluster (4kiB) will automatically cache instructions as needed. The cluster instruction cache is shared between all the cores in the cluster. Generally, the cluster cores will be executing the same area of code on different data; hence, the shared cluster instruction cache exploits this to reduce memory accesses for loading instructions: this provides an ideal memory architecture for the execution of code implementing parallelized algorithms.

All elements of the PCS have a shared access to L2 memory area while multiple DMA units allow autonomous, fast, low power transfers between cluster L1 and L2 memory and between L2 memory and external peripherals.

2.5.3 Micro DMA subsystem and Interfaces

The uDMA subsystem provides direct transfer of data between AXI Master, L2 memory, and the different peripherals instantiated in it. It helps in relaxing the execution load of FC RI5CY core.

The uDMA has two ports connecting the logarithmic interconnect of the PCS toward the interleaved memory area. Out of the two ports, one is write-only and dedicated to the RX (Receive) channels, and one is read-only and dedicated to the TX (Transmit) channels. The ports towards the memory are 32-bit wide while the single-channel interfaces toward the peripherals may have smaller widths. The widths supported by the UDMA are 8,16 and 32bits, either set at design-time or run-time to fit the specific features of the attached peripherals.

Data transfer on TX and RX channels are completely decoupled and not synchronized. This means that if synchronization is needed for TX and RX to happen at the same time, it has to be implemented on the peripheral side. The CPU can enable or disable the channels, and when the channel is enabled data transfer to/from memory is always triggered by the peripherals, meaning that the CPU is not involved in the data transfer.

Starting a transaction on a channel requires only three accesses in addition to the peripheral configuration: the software has to program the source or target pointer, the transfer length in bytes and send the start command. At the end of each transaction on each channel, the uDMA generates a dedicated event to notify the fabric controller that is possible to queue another transaction. Each channel can have one transaction running, and one enqueued that is automatically started at the end of the running transaction to support continuous data transfers based on a double buffering scheme.

2.5.4 Performance Counters

Each RI5CY cores of the FC and the cluster provide a performance counter. These are 32-bit counter registers that can be configured to count various indexes, as the total number of cycles, instructions executed, data memory load or store, jumps, branches taken, data hazards and other advanced architect-based information as the number of cycles wasted due to L1/logarithmic interconnect contention. These performance counters are really useful at design time to measure the performance of the firmware, the

control and the applications and in a more advanced PCS firmware can also be exploited to achieve better performances or to fix execution issues at run-time by stopping the problematic features while leaving the hard real-time thermal control functions active.

Chapter 3

Firmware Architecture

In this chapter the power controller implementation and the design choices will be presented. The structure of the firmware is described to provide a basic knowledge for the results presented in Chapter 4.

3.1 Control Implementation

The hierarchical two-layer control structure with the additional power model adaptation is suitable to be implemented within different *"objects"* with different **timing and execution constraints** based on the **requirements** of the layer. The inner thermal layer has the highest frequency constraints and priority of execution, the power dispatching layer has a medium priority of execution but still a hard real-time deadline and can be executed with a slight lower frequency, and the model adaptation layer has a low execution priority and lower frequency of execution, tolerating also missed deadlines which will only impact in the model adaptation performances.

In this scheme also the I/O has different performance requirements: the PVT sensors and the Voltage/Frequency actuation are aligned with the thermal layer, the parametric inputs and the telemetry and control data are aligned with the power dispatching layer but they exhibit a higher latency and a lower execution priority with a soft real-time deadline. The VR sensor

Item	Frequency	Priority	Hard Real-Time
Thermal Layer	HIGH	HIGH	yes
Power Dispatching	Medium	Medium-High	yes
Model Adaptation	Medium-Low	Low	no
PVT Sensors	HIGH	HIGH	no
Voltage-Freq.	HIGH	HIGH	yes
Parametric Inputs	Medium	Medium-Low	no
Telemetry	Low	Low	no
Power Consumption	Medium	Low	no
Out-of-Band Services	-	Low	no

Table 3.1: Control Structure and I/Os Requirements

information are needed by the Model adaptation layer and so are aligned with its requirements. The out-band services has the lowest priority and generally they have no frequency constraints since they are queried occasionally.

Analyzing the current HPC processors architecture we consider for our project a ring control network that is running at $\sim$ 250MHz with a delay of $\sim$ 250ns per transaction to fetch and write the control data of a 36 core processor, every step of the power controller needs about 50us of data transaction. Considering this overhead and starting from a simpler first implementation approach, the two control layers are unified under the same frequency and priority requirements, given by the stricter thermal layer values. With this choice the power capping control performances are increased, while the parametric inputs, the telemetry and control data retain their lower requirements and can be aligned to other layers. For simplicity, the Power model adaptation and this I/O services are merged under compromised frequency and priority values.

3.2 Using a RTOS

For the Firmware structure we had two options: to write a *"bare meta"* code with a finite state machine that regulates all the firmware structures and steps, or to write the code over a **real-time infrastructure** (OS). The second option was chosen to try something innovative (since the first option is the industrial standard) and for all the benefits that developing over an OS brings, such as:

- **scalability** and **portability**, given by the abstraction that a RTOS provides to the user application;
- **modularity**, **upgradability** and **maintainability** (along with better **readability**) of the code;
- more **flexibility** and the possibility to execute code with **mixed computation requirements** (periodic, continuous, event-driven, soft and hard real-time)

Developing over a OS bears also several challenges and raises concern over the required performances (considering the characteristics of the system over which the PCS is performing its control action) and the **safety** and **reliability** of the entire system.

FreeRTOS was chosen as the real time OS, because it is a valid and well-established product with the necessary characteristics and tools to obtain the control requirements described in 2.4. It is also open source which granted us the possibility to look at the source code while developing and it is aligned with the scope of the PCS project. Lastly FreeRTOS provides a safety critical version, called safeRTOS which is pre-certified for all standards and can be used in medical, industrial and automotive applications[1]

[1]More information at `freertos.org/FreeRTOS-Plus/Safety_Critical_Certified/SafeRTOS.html`

3.3 FreeRTOS

FreeRTOS is an open source real time OS designed mainly to run on low power and resources-strict microcontrollers and it is capable to comply with soft and hard real-time requirements. It is released under the GPL license with some restrictions and exceptions, and can be used for commercial purposes withholding the product's intellectual properties.

Thanks to its features FreeRTOS allows the execution of pre-emptive, cooperative or hybrid scheduling solutions with maximum flexibility in assignation and modification of tasks priority, and with the ability to block tasks and suspend the scheduler. The kernel provides to the application developer some tools and objects and these are: Queues, Binary, Counting and Recursive Semaphores, Mutexes, Recursive Mutexes, Direct to Task Notifications, Software Timers, Event Groups, Tickhook and Idle hook functions. It also allows the verification of stack overflow at run-time, the collection of data and statistics on run-time tasks, and for some architectures the possibility for a complete model for interrupt nesting.

It is distributed with a core code folder common to all platforms, which contains all the structures, functions and data sets of the firmware along with the APIs for all the objects that it provides. To port the code on the several platforms, there are specific code folders which mainly contain efficient code to perform the actions related to the hardware, like the context switch and the systick generation. FreeRTOS is compatible with more than 40 architectures and 15 toolchains. The configuration of the FreeRTOS kernel is performed by modifying the FreeRTOSConfig.h file[2].

3.3.1 The Scheduler

In FreeRTOS there are four states in which a task can be in: **Running State**, **Ready state**, **Blocked state** and **Suspended state**.

[2]For additional information on FreeRTOS refer to [3], [4], `freertos.org/` and Analisi e Test del Sistema Operativo FreeRTOS per Microcontrollori

Running State: the tasks that are in Running state are the ones that are currently executing. In a **single core** system (this is the general case) only one task can be in the running state. If there is no task in running state, the scheduler will select one to run among the available ones depending on the chosen scheduler configuration. If there are no available tasks it will execute the **idle task**.

Ready state: the tasks are in Ready state if they are ready and available to execute, but they are not currently executing. This is the pool from which the scheduler selects the task to run if there is currently no task in the running state.

Blocked state: the tasks are in the Blocked state if they are waiting for a synchronization or temporal event (or both). Tasks in blocked state cannot execute.

Suspended state: the tasks that are in Suspended state are ignored by the scheduler (so they produce no execution overhead) but are still not deleted.

The possible states flow is shown in the Figure 3.1.

The FreeRTOS scheduler is designed to select for execution the task in the Ready state with the highest priority. If two or more tasks in the ready state have the same priority the scheduler will choose the task which waited the most. In this way tasks with same priority will alternate in the execution even if the time slice configuration is not active. If the pre-emption is active, a task passing from the blocked state to the ready state will trigger a context switch. If the time slice configuration is active, tasks with the same priority will alternate every tick interrupt.

The assignment of priority is a key factor for the correct behavior of the application developed on FreeRTOS. The priority can span from 0 (lower value, same as the idle task) to the `(configMAX_PRIORITIES-1)` which is the maximum value. The priority can be changed after the creation of the

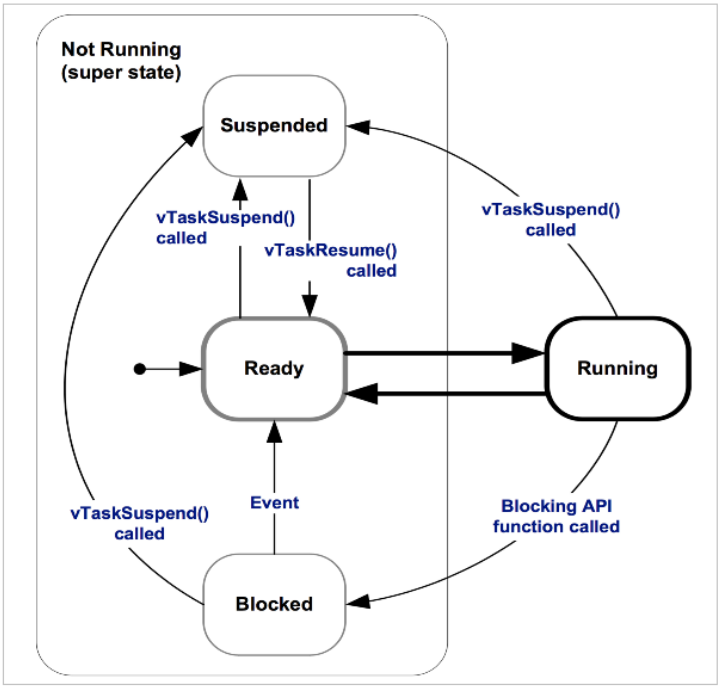

Figure 3.1: Possible states flow of FreeRTOS tasks

task allowing to implement dynamic priority algorithms like Earliest Dead-Line First (EDF algorithm), which are not directly supported by the kernel itself through parameter configuration.

3.3.2 The Tasks

The **task** is the key object of FreeRTOS. It is implemented as a C function with a **definite prototype** and can be thought as a program on its own. A task should contain an infinite loop and must not be allowed to return and to execute past the end of the function. If a task is no longer required it should explicitly be deleted by calling the proper API function. The tasks are created by the API function `xTaskCreate()`: this function takes the pointer to the C function implementing the task code with some additional creation parameters (priority, stack size, name), and it populates the task handle. The task handle is a pointer used inside FreeRTOS to reference the task in API calls.

```
void ATaskFunction ( void * pvParameters ) {
/* Variables can be declared just as per a normal function.
If the variable is declared static only one unique copy of
the variable is created for all the instances of the task
and it will be shared among them */
int32_t lVariableExample = 0;
/* A task will normally be implemented as an infinite loop.
   */
for ( ;; ) {
/* The code to implement the task functionality will go here.
    */
}
/* A task should never break out of the above loop, but in
case it happens then the task must be deleted before
reaching the end of its implementing function. */
vTaskDelete ( NULL ) ;
}
```

Figure 3.2: Prototype of the C Function implementing a Task

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
            const char * const pcName,
            unsigned short usStackDepth,
            void *pvParameters, UBaseType_t uxPriority,
            TaskHandle_t *pxCreatedTask );
```

Figure 3.3: FreeRTOS API Function Prototype of xTaskCreate()

3.4 Firmware Methodology

Considering the Control Requirements set in section 2.4, the chosen control implementation discussed in section 3.1 and the assumed Power Controller Interface described in section 2.2.1, the software implementation decisions and the design choices are presented.

3.4.1 Tasks of the Firmware

Following the control structure and based on the FreeRTOS architecture the power controller is implemented into 2 periodic Tasks, 1 event-triggered task and 1 Initialization task (note that there is also the IdleTask provided by FreeRTOS with the possibility to append application-specific code to it):

Initialization Task: this task gathers all initialization steps that are necessary for a proper functioning of the control structure. It creates the FreeRTOS objects needed for the application (semaphores and mutexes needed for data exchange and EOT notification), it assigns the initial values to the global and control variables, and it starts the timer for the interrupt generation. This task starts with the highest priority, then after it has fulfilled its job, it suspends itself waiting to be triggered again if needed.

Control Task: this is the most important task of the power controller. It runs with a period of 500us and is responsible for the thermal control and the power capping of the plant chip. It also gathers and manages telemetry data and checks for BMC request. It has the highest priority below the priority of the initialization task because it is the task with the stricter hard real-time requirements;

TaskOS: this task is responsible of updating the control model of the EPI processor to achieve a better control by the model parameters. It also exchanges commands and data with the upper layer OS and with user applications, and in future versions it may perform advanced control by

exploiting the cluster cores as an accelerator. It executes periodically with a periodicity that is a **multiple** (configurable) of the periodicity of the control task, and it has a priority lower than the control task to allow to be pre-empted by it.

BMC Task: this task is responsible to manage the out-of-band services. It is in charge of communicating with the administrator system and it collects, analyzes and signals errors that occurred in the power controller. In case of a double Socket Motherboard it disputed whether the BMC tasks of the two processor may talk to each other to obtain a joint regulation of the input values and the set-points for power usage across all board.

3.4.2 Obtain Lower Timings than RTOS Tick

The chosen periodicity of the control task is 500us. FreeRTOS has an internal periodic structure that is triggered by an interrupt with a frequency that can be defined by the user; this interval of time is called **Tick**, and is the basic time unit of measure inside FreeRTOS. This periodic interrupt calls a FreeRTOS internal function that is responsible of the management of the OS, of switching running tasks (if Round Robin is configured and there are two or more tasks with the same priority ready to execute), and executing context switches if some task reached its maximum waiting time (expressed in Ticks)[3]. The **Systick** (which is the event that triggers the Tick change) is generally set with a frequency of 1000Hz. To achieve the mentioned periodicity timings using only FreeRTOS tools, the only choice is to reduce the frequency of the sysick, and by doing so, increasing the overheads of the RTOS. This would negatively impact the performances of the power controller. Further, blocking the execution interval of the control task on the FreeRTOS would create limitations:

[3]If it has a periodicity greater than the current running tasks (and preemption is enabled)

- the control structure would be linked to the FreeRTOS structure, so RTOS run-time issues may impact more significantly on the control action;
- it is impossible to change the period of the control task at run-time. An advanced configuration could reduce the frequency at which it executes if deadlines are not properly met, or increase it if the slack time is enough to improve performances;
- the systick frequency could not be decreased to reduce overheads (since the proposed structure is mostly event and time-driven with preemption and all tasks have different priority, it runs without the necessity of *"tick context switches"*);
- the systick frequency could not be exploited to detect execution issues and unblock wrongly blocked tasks

To achieve the wanted periodicity without the limitations mentioned above, a **hardware timer** was set up to trigger an ISR with a specific frequency, modifiable at run-time. Inside this timer ISR the task that is desired to be activated is signaled with the FreeRTOS API `vTaskNotifyGiveFromISR()`. The **Notification system** is the fastest and most efficient method of blocking and activating tasks[4].

3.4.3 FreeRTOS Notification System

When the corresponding configuration parameter is enabled in the FreeRTOSConfig.h file, the direct to task notification system is activated, and an unsigned 32-bit int variable called **Notification Value** and a **Notification State** are associated to each task. The task notification allows tasks to interact with each other and synchronize with ISRs without the need for an external communication object. For this reason tasks notifications are much faster, more performing and efficient (in the use of RAM) than other kernel

[4] As stated in [4]

objects which would perform the same action. There are also some limitations but these are not a disadvantage for our project implementation.

By default the notification state of a task is set to `taskNOT_WAITING_NOTIFICATION`. When a task tries to read (take or wait) its notification value, if this request is blocking the task notification status is set to `taskWAITING_NOTIFICATION`; in this case the task exits the Running state and enters the Blocked state, waiting for its state to change, allowing the user also to specify a maximum block time. When a task receives a notification its notification state is set to `taskNOTIFICATION_RECEIVED`, and the notification value is updated with the value sent by the caller's API. When a task reads the notification value, its state is set back to `taskNOT_WAITING_NOTIFICATION`.

To better understand the notification system, the steps followed by the control task to set a task execution periodicity are described below and illustrated in Figure 3.4:

1. A task calls the API `ulTaskNotifyTake()`. Since the task notification state is `taskNOT_WAITING_NOTIFICATION` and no one had notified this task before, the task blocks itself, either for the number of ticks passed through the parameter `xTicksToWait` or indefinitely if `portMAX_DELAY` is passed;

2. After the set interval of time has passed, a timer interrupt is triggered and the corresponding ISR is called;

3. Inside the ISR the API `vTaskNotifyGiveFromISR()` (FromISR is the interrupt safe version of the analogue API) is called and it sets the notification state of the task passed as argument to `taskNOTIFICATION_RECEIVED`. The API returns a value that can be used to trigger a context switch if the task that the API unblocked has a priority higher than the current running task;

4. The task is unblocked and in the ready state; when the FreeRTOS

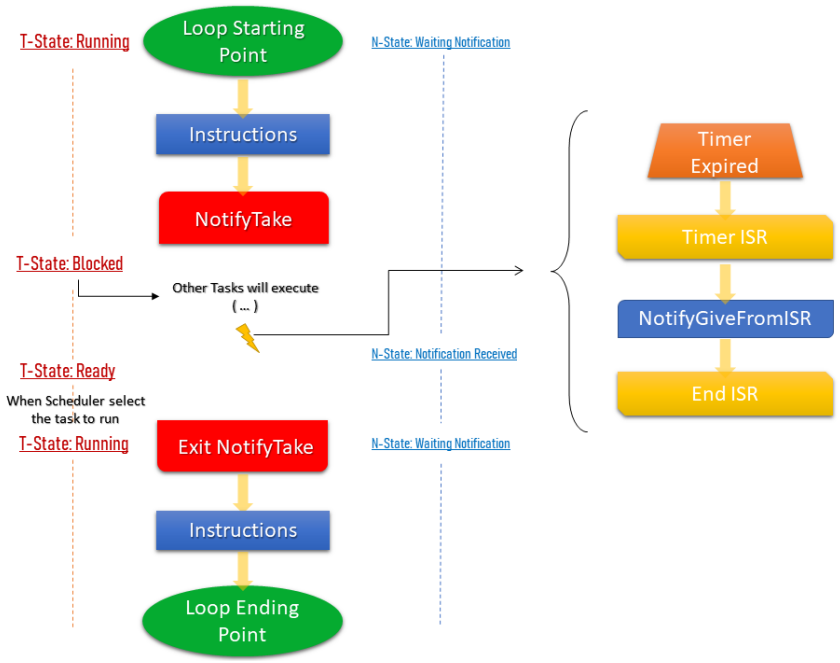

Figure 3.4: FreeRTOS Notification system from ISR

scheduler will put it in the running state it will execute again starting from the blocking call of the `ulTaskNotifyTake()`,

5. (until) the task calls again the `ulTaskNotifyTake()` (remember we are in an infinite for() loop).

3.4.4 How tasks access to the shared memory to communicate

The tasks need to exchange data, commands and information between them. In a real-time system that has to satisfy safety requirements, to avoid race conditions and corrupted data reading, a proper sequence must be followed. FreeRTOS has two objects that can be used to avoid these problems: the **Binary Semaphores** and the **Mutexes**. The binary semaphore is like a token that can be taken or given and it represent the availability of the resource that needs to be secured. The mutexes are binary semaphores with

Resource	Type	Periodic Ctrl	TaskOS	BMC
Parameter Inputs	Mutex	x	x	x
Telemetry	Mutex	x	x	x
Perf. Parameters	Mutex	x	x	x
Error Map	Mutex	x	x	x
SPI	Mutex	x	x	
Send EoT	Binary	x	x	
Receive EoT	Binary	x	x	

Table 3.2: Semaphores and Mutexes scheme

the **priority inheritance** mechanism, and in FreeRTOS Mutexes cannot be used inside ISRs. The priority inheritance aims to minimizes the negative effects of the **priority inversion** problem by raising the priority of the task that is holding the mutex to the highest priority of the tasks blocked attempting to obtain the same mutex.

The data are exchanged between task using mutexes. There is one mutex for each **global variable** that is accessed by more than one task. For variables that are accessed also by ISR and to obtain event-driven triggers (for example, to notify when the data are ready or a transaction has been correctly executed) the binary semaphore is used instead. In case a binary semaphore is used, we posed particular attention to make them able to block just one task and to not have other nested blocking calls to avoid priority inversion.

3.4.5 The Periodic Control Task

All the FreeRTOS tasks follow a **common structure**: there is an initialization that is executed only once when the task is started for the first time and then there is an **infinite for(;;) loop** where there is the core code of the task. The code inside the for() loop will be executed continuously. The Periodic Control task follows the same structure: it has an initialization

part where all the variables of the task are declared and initialized, and the infinite for() loop in which the control algorithm is executed cyclically. The loop starts with the call to the API function ulTaskNotifTake() which is the one that blocks the task and reactivates it after the periodic interval has passed. Then there is a main Control Sequence that is characterized as:

1. **Write** to the actuators (in this case the chip cores) the values computed at the previous iteration (k-1);
2. **Read** new input values;
3. **Compute** new Output values.

Intertwined with this main sequence there are other sections that are mainly dedicated to the management of data, to receive commands and input parameters from TaskOS, or to check numerical values error. These sections are mixed with the main control sequence for optimization reasons: they occupy a portion of the time delay necessary to send and receive data from the chip sensors and actuators. If the periodic control task has nothing to execute and the read/write transactions are not completed, the task blocks itself by means of Binary Semaphore waiting to be unblocked by the EoT ISR.

3.4.6 TaskOS

The taskOS has a similar structure to that of the control task: it has an initialization part and the infinite loop started with the call the `ulTaskNotifyTake()`. The code flow is described as:

1. **Read** the VRM values from the board
2. **Read** the telemetry values generated by the control task
3. **Compute** a model optimization of the chip based on these received values

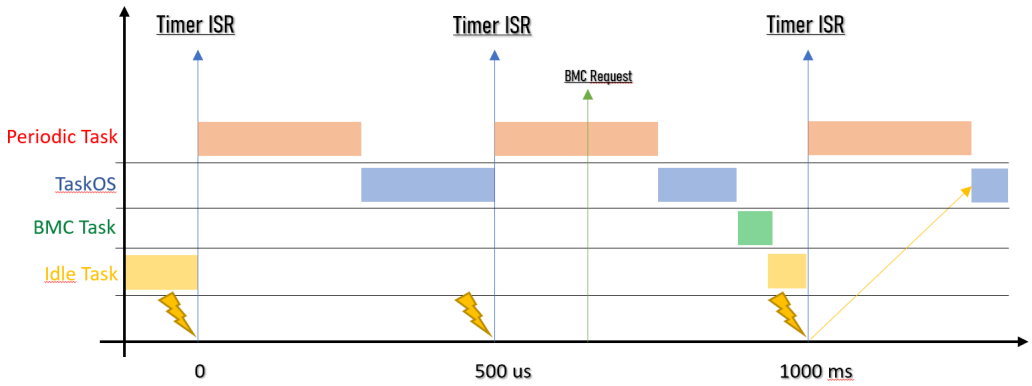

Figure 3.5: Task Scheduling Scheme

4. **Check** requests and new input commands from the "over OS"

5. **Write** the data to the shared memory

3.5 Code architecture and Libraries

The application code is divided in files, based on the category and the objective of the C functions, to obtain **modularity** and a better readability and organization. Some files are needed only for the tests illustrated in Chapter 4, others may be not necessary in some implementations and further libraries may be added effortlessly.

The **core files** are task.c and task.h which contain the definitions of the tasks, the declaration of the required data structures and the shared memory variables, and additional functions and macros for performing I/O and intertask communication.

Another relevant file is PCSConfig.h which contains a long list of parametric defines used for the configuration of all the aspects of the application. The configuration includes:

- the **plant** (chip and board) configuration and its parameters;

- the **control structure**, the **control parameters** and initial values;

- the **code structure**, the **desired features** and performances of the power controller;
- additional parametric elements of the code.

The mathematical functions, the PIDs, the power model and the model adaptation algorithms are gathered in the math.c and math.h files.

In the main.c there is the initialization of hardware resources, the timer ISR dedicated to set the periodicity of the tasks and the other ISRs related to the EoT signaling.

Additionally there are the SPI and simulation files which are focused on the implementation of the HiL test, the TimerMeasure files which contain the function to perform time measurements for the performances tests, and the printFloat libraries to help debugging the code.

Chapter 4

Experimental Results

In this chapter the tests and the obtained results relative to the firmware development will be presented. Firstly a brief paragraph will be dedicated to the Matlab simulated results, then some performance tests on the RTOS overheads and control execution timings will be discussed. In the end a HiL test will be presented to confirm the result obtained in the Matlab environment.

4.1 Experimental Setup

The hardware implementation of the PCS is based on the PULPissimo platform. To execute the tests the **Greenwaves GAP8** microcontroller is selected. GAP8 is a commercial PULP-based product with a design comparable to the PULPissimo microcontroller described in section 2.5. The tests were executed on a GAP8v1 board with multiple versions of the SDK, in particular the 2.7 and the 3.2_dev. The most relevant difference between GAP8 platform and the proposed PCS hardware implementation is the absence of the FPU. The lack of a dedicated computation unit for floating point numbers caused the necessity to adopt a simulated time environment to perform the HiL test since the floating point operations of the power controller are simulated through integer ALU operations.

The GAP8 SDK has also a FreeRTOS v.10.2.1 proprietary implementation that compiles with GCC. Unfortunately, during the execution of the tests, we noticed that the Greenwaves SDK was flawed by numerous bugs (of which multiple were critical bugs) and by several issues largely described in the appendix A. These circumstances caused prolonged delays and the transition between various versions of the SDK among the tests.

4.1.1 FreeRTOS Configuration

The FreeRTOS scheduler is configured with a **Prioritized Pre-emptive Scheduling** with **Time Slicing** enabled.

The systick frequency is set at 1000Hz, but it is a value still under consideration because lowering it reduces the overheads, but choosing a right value for it can be exploited to detect execution issues and avoid locks or tasks blocked indefinitely. The Notification system, the binary semaphores and the mutexes are enabled. Tickhook function and run-time stats are currently disabled to reduce overheads, but if in an advanced version the systick is exploited to check for run-time issues these two features could be activated too.

The other configuration parameters are not fundamental for the test configuration, but it is worth noticing that on the GAP8 both:

- `configKERNEL_INTERRUPT_PRIORITY`
- `configMAX_SYSCALL_INTERRUPT_PRIORITY`

are set, so the employment of **interrupt-safe versions** of freeRTOS APIs in ISR is permitted[1].

4.1.2 Chip Modelling: STM32 Nucleo

The thermal model computation to perform the HiL test is entrusted to a **STM32 F401RE Nucleo** microcontroller, running an ARM cortex-M4

[1]The behavior is described in `https://www.freertos.org/a00110.html`

core with FPU and MAC optimization. The code is compiled with the last CubeMX STM libraries.

The data connection between the two platform is carried on by a **half-duplex SPI** line. The GAP8 takes the role of the master while the STM Nucleo is the slave. The STM Nucleo has also an **UART connection** to a workstation to send the test data, sensor telemetry and to receive user commands.

The C code to simulate the chip plant is implemented to achieve a **continuous simulation**. After a short initialization, the STM Nucleo starts executing the thermal model by repeatedly calling a model-step-update function inside a while(1). This function computes a thermal discrete state space update taking in input the frequencies received by the GAP8 power controller and returning the vector with the core temperatures. To achieve a continuous computation of the state, the STM Nucleo application is structured with **asynchronous calls** to manage the SPI and UART transmissions which are then handled by the DMA. This is harmonized with the STM Nuleo characteristic of being the SPI slave.

4.1.3 SPI Implementation

The communication between the GAP8 and STM Nucleo boards is implemented with a **half-duplex single-slave SPI**. The master is the GAP8. It sends a 1MHz clock with low polarity, first-edge phase and most significant bit first characteristics. Every transmission consists of 4 8-bits packages for a total of **32 bits** communication.

As described in previous paragraphs, the SPI communication is managed through interrupts and DMA on the STM32. Further the two platform are connected through pin cables which could introduce some disturbances. For these reasons, and to overcome possible SPI bugs on the GAP8 side described in A, an **handshake system** between the two systems has been implemented.

When the GAP8 wants to communicate, it starts by sending a 32-bits

message specific for the type of communication that it wants to perform. Then it starts the SPI clock to read the STM Nucleo answer to the previously sent message. If the received 32-bits message matches a one-to-one specific 32-bit value corresponding to the previously sent message, the handshake has been completed correctly and the action that the GAP8 wanted to perform can be started. If it doesn't match, in order not to impact performances and to not miss deadlines, the GAP8 action is aborted and the execution of the power controller continues. For example if the GAP8 wants to read the temperature of the cores and the handshake fails, the control action computation will continue with the previously received temperature values and an error related to the missed handshake is set.

Once the handshake is completed, a further control on the messages is integrated in the 32-bits packages. The **first 8 bits** describe the type of operation, the **second 8-bits** package stores the core identification number of which the following information are related, and the **last two 8-bits** packages store the actual values. With this technique an additional control is performed inside each communication to keep the the systems synchronized.

To stabilize the SPI clock, the test involving SPI transaction are executed with the GAP8 clock set to 50MHz.

GAP8 SPI

Regarding the GAP8 side of the SPI, due to the problems described in the appendix A, the SPI communication has been forced to be implemented with **asynchronous calls** with particular attention to the maximum number of transmission enqueued in the uDMA (as described in section 2.5, 2 is the maximum number of operations that the uDMA can store).

The task that wants to start an SPI transaction takes a mutex which is related to the SPI hardware resource. In this way only one task can communicate and problems with the uDMA are avoided. Then the task that holds the mutex starts the SPI communication and blocks itself waiting for a binary semaphore to be given. The binary semaphore represents the **SPI**

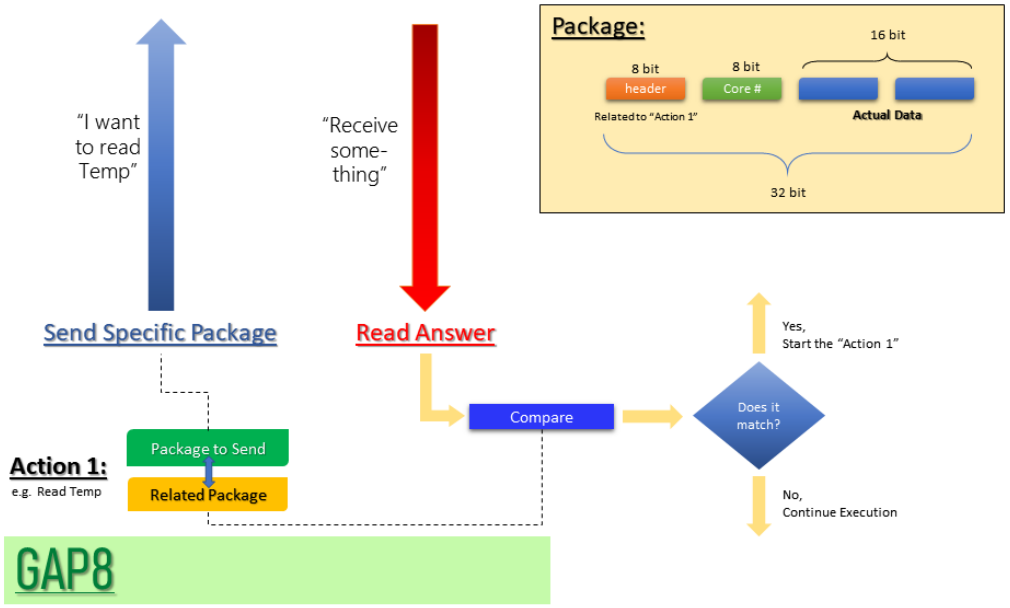

Figure 4.1: SPI Handshake and Package Structure

EoT signal: an ISR will give the binary semaphore (unblocking the task) when the SPI transaction enqueued is completed. The ISR is called by the GAP8 SPI APIs. There are two binary semaphores, one corresponding to the `SPI_send()` and one to the `SPI_receive()` functions.

When a task finishes its SPI-related operation (after being unblocked by the last binary semaphore), it releases the SPI-related mutex for other tasks to use the SPI resource. Binary semaphores are used because mutexes cannot be called inside ISRs.

4.2 Test Bed

The first test is performed in Matlab to verify the quality of the power controller. Then performance measurement tests are executed to establish the overheads of FreeRTOS and the computation timings of the control structure. The measurements are analyzed to check if the chosen values and requirements can be satisfied. The tests are executed on the GAP8 platform

```
/* Reset all to 0 */
asm volatile ("csrw 0x79F, %0" :: "r" (0));

/* Configure */
// Only Configured for Cycle
uint32_t maskConfig = (PCER_CYCLE_Msk & ~(0x1UL << 17U));
asm volatile ("csrw 0x7A0, %0" : "+r" (maskConfig) );

/* Enable */
uint32_t maskEnable = (1 << PCMR_GLBEN_Pos)
              (1 << PCMR_SATU_Pos);
asm volatile ("csrw 0x7A1, %0" :: "r" (maskEnable));
```

and the measurements are obtained with the performance counters which is the most precise and direct method to obtain cycle and instruction values along with other interesting information such as the performed load/store or the stalls.

Finally the Hardware in the Loop test is performed to verify the PCS implementation. Additional tests are performed to set up the HiL experiment, measuring SPI and floating point performances. SPI connection tests were very much needed to find and fix the numerous bugs afflicting the GAP8 SDK and environment[2].

4.2.1 GAP8 Measurements Tools

The tools to perform the measurements are based on querying the internal **performance counters** to measure the number of cycles and the number of instructions that are executed for the corresponding part of code under analysis. The instructions to initialize the performance counter and to start the recording are shown below:

and the function to read the register is:

Before performing the measurements, the overhead introduced by the read function was analyzed to obtain more accuracy. The value of the overhead is 44 cycles. This value is quite high, but this is mainly due to the fact that we

[2]As largely described in Appendix A

```
/* Read */
uint32_t value = 0;

/* Cycles */
__ASM volatile ("csrr %0, 0x780" : "=r" (value));
/* Instructions */
//asm volatile ("csrr %0, 0x781" : "=r" (value));
/* LD stall */
//asm volatile ("csrr %0, 0x782" : "=r" (value));
/* Istr Stall */
//asm volatile ("csrr %0, 0x784" : "=r" (value));
/* LD */
//asm volatile ("csrr %0, 0x785" : "=r" (value));
/* Store */
//asm volatile ("csrr %0, 0x786" : "=r" (value));

return value;
```

are using global variables to store the data extracted from the performance counters: the global variables are stored in the L2 memory, which requires higher number of cycles to perform load and store.

4.3 Matlab Control Performances

We created the state space matrixes of the chip model proposed in 2.13 in Matlab and a simulation of the chip model and the proposed power controller was developed in Simulink. The adopted parameters of the model are illustrated in the Table 4.1. The PID values are showed in Table 4.2. A zero holder with 500us of delay is added at both input and output of the power controller and Gaussian noise is added to the thermal state of the plant model to simulate a more realistic scenario.

The objective of the test is to analyze the performances of the power controller and the quality of the control action with respect to the optimization problem described in 2.14. The input values to the power controller are illustrated in Table 4.3. In particular two test were performed: in the first test the frequency input is a step with $f_i = f_{max} = 3Ghz$ to observe both

Parameter	Value
$R_{Si,v,i}$	5 [K/W]
$R_{Si,h,ij}$	42 [K/W]
$R_{Cu,v,i}$	290 [K/W]
$R_{Cu,h,ij}$	2.4 [K/W]
$C_{Si,i}$	$1 \times 10^{-3}[J/K]$
$C_{Ci,i}$	$1.2 \times 10^{-2}[J/K]$
T_{CRIT}	80 $[°C]$
T_{amb}	25 $[°C]$

Table 4.1: Model Parameters of the Simulink Test

Parameter	Value
Proportional	2.0
Integral	0.1
Derivative	0
Anti Wind-Up Limiter	$-0.75 \cdot P_T^*$
Output Saturation	$[0, -P_T^*]$

Table 4.2: PID Parameters of the Simulink Test

the power capping and the thermal control, in the second test a variable frequency input is given to observe the behavior with non-constant inputs. Random workloads have been assigned to each core, in order to introduce variations in the cores power consumption. The outputs are shown in Figure 4.2, 4.3 and fig:result2b.

From Figure 4.2 (b) it can be observed that the thermal controller behaves correctly limiting the temperature of each core to $T_{Crit} - T_{margin}$. Figures 4.2 (a) and (b) illustrate the PID behavior of one core and the error going to zero. The temperature oscillations showed in 4.2 (d) are limited in a 2°C

Parameter	Value
Frequency	3.0 [GHz]
Power Budget	42.0 [K/W]
Based Workload	100 [CPI]
Initial Temperature	25.0 [°C]
Fixed Voltage	1.1 [V]
Temp. Limit	87.0 [°C]
Temp. margin	10.0 [°C]

Table 4.3: Input Parameters of the Simulink Test

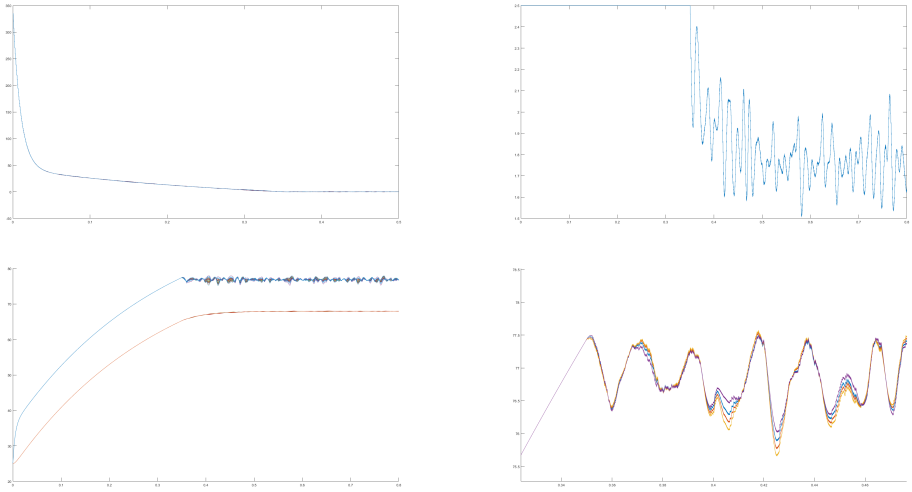

Figure 4.2: Simulation results of Test 1 with constant Frequency Input: (a) PID errors, (b) PID output of core 5, (c) Cores and Heat-spreader (orange) Temperatures, (d) Zoom-in on the Cores Temperatures at steady state.

range. In the test with variable inputs the thermal controller still manages to limit the temperature as we can see from Figure 4.3 (top).

From Figure 4.4 (d) we can observe that the power capping works properly limiting the input frequency of the cores.

In Figure 4.2 (c) and 4.4 (c) we can notice strong changes in the PIDs output. This behavior is probably caused by the non-linearities introduced by the saturation. A filter to smooth out this oscillation (or to simply reduce their amplitude) can be added in the hardware implementation to avoid highly variable execution timings and a performance decrease due to the binding constraints enforcement.

4.4 FreeRTOS Overheads and Performances

To evaluate the feasibility of using a RTOS and to analyze its overheads and performance with a focus on the available slack time, several tests are performed to asses the timings related to the RTOS management.

The first batch of tests are executed to evaluate the **activation time**

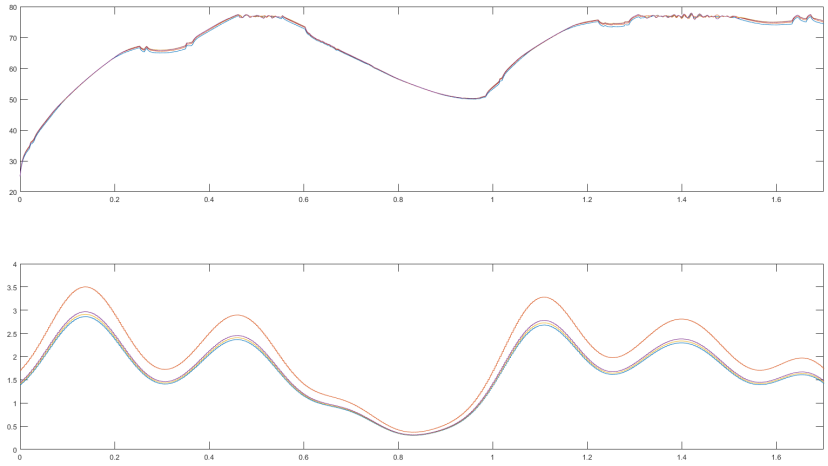

Figure 4.3: Temperatures of the Cores (top) and Frequency Input (bot)

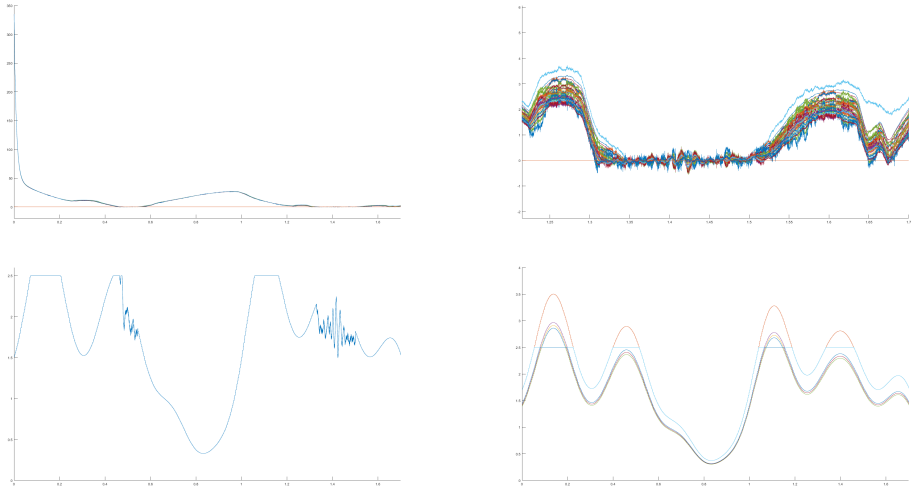

Figure 4.4: Simulation results of Test 2 with variable Frequency Input: (a) PID errors, (b) Zoom-in on the PID errors, (c) PID output of core 5, (d) Power capping of cores 1,3, 5,7.

of the periodic task through the notification system of FreeRTOS and the time needed by the periodic task to block itself and perform a **context switch** to another task. The analysis of these two values is important to evaluate the performance and the efficiency of developing the power controller over a RTOS in comparison to a *bare metal* implementation which should introduce lower overheads. Another important parameter to analyze is the **jitter** introduced by FreeRTOS, mainly due to the Systick function execution which has a configured period of 1ms. The analysis of these data led to some improvement considerations for the hardware design of the PCS.

The activation time of the task, in the environment of our test, is the sum of:

- the time related to the execution and the return from the **TIMER1 ISR** (excluded the time necessary by GAP8 to perform a hardware context switch)
- the execution time of the FreeRTOS API **vTaskNotifyGiveFromISR()** used to unblock the periodic task
- the execution time of the FreeRTOS API **portYIELD_FROM_ISR()** which checks if the unblocked task has a periodicity higher than the currently executing task and if a Context Switch is necessary
- the time to perform the FreeRTOS **Context Switch**
- the time to **conclude** the FreeRTOS API vTaskNotifyTake() which was called by the periodic task to block itself. This is the final step to unblock the task.

The "change of task" time is instead the sum of:

- the time to execute the **first part** of the FreeRTOS API vTaskNotify-Take() used by the Periodic Task to block itself
- the time to perform the FreeRTOS **context switch**

	Activation Time (in Cycles)	**Context Switch** (in Cycles)
Mean	1810,40	1452,58
Standard Dev.	8,78	13,16
Range (Max - min)	34	55
Max min	1829 1795	1481 1426
kurtosis	-0,4982	-0,7131

Table 4.4: Results of the Performance Test

- some minor additional time due to some code on the second task used to measure the "change of task" time

The test was repeated for 40 iterations, to get results reflecting the variability of timings that can be measured on a real-time scheduler.

4.4.1 Results

The obtained results are shown in Table 4.4. The mean value of the activation time of the periodic task is **1810 cycles** with a standard deviation of 9.8 cycles. At 50 MHz it means that the overhead due to the activation of the Periodic Task is **36us**. Considering 500us as the value of the execution interval of the control task, the percentage needed for the management of the periodicity is **7.2%** of the interval time. The mean value of the Context Switch Time is **1453 cycles** with a standard deviation of 13.2 cycles. At 50Mhz it is **29us**, **5.8%** of the interval time. Observing these values, the performance degradation and the overheads introduced by the FreeRTOS task and run-time managements are considerable, occupying a total amount of **20% of the available time** in the periodic interval. To better understand the causes of these overheads, the number of instructions executed can be observed: the activation time of the periodic task generates 331 instructions, while the context switch time generates 271. Each instruction is executed, in average, in 5.40 cycles.

These values are too high and required further tests to investigate the causes. Therefore the measurements were repeated examining the number of Load/Store, Load stall and cycles waiting for instruction fetches.

	Activation Time			**Context Switch**		
	Mean	Std. Dev.	Range	Mean	Std. Dev.	Range
Loads	41	0	0	35	0	0
Stores	43	0	0	24,44	1,22	3
Load Stalls	30, 46	0,50	2	30	0	0
Cycles I-Miss	313,23	16,44	71	172,39	12,57	42

Table 4.5: Results of the Performance Test - Causes Investigation

Observing the data in the Table 4.5 related to the activation time, the 74,3% of the loads are load stall and there are about 313 cycles wasted for instructions miss. For each load stall GAP8 takes around 12 cycles to access L2 memory. If there were no instructions miss and load stalls, the cycles necessary could be reduced as:

$$cycles_{a_n} = 1810 - (30 \cdot 12) - 313 = 1137 \tag{4.1}$$

A similar computation is done also for the context switch time:

$$cycles_{cs_n} = 1453 - (30 \cdot 12) - 172 = 921 \tag{4.2}$$

Focusing on the **jitter analysis**, it can be observed that the Range parameter (the difference between the maximum and the minimum value) for the activation time is **34 cycles** and for the context switch time is **55 cycles**. This values show that there is almost no jitter, considering they are less than the 3% of the measured values and considering that the measures were not extremely precise due to the difficulties and the bugs afflicting the versions of the SDK with which the tests were performed[3].

[3]For more information refer to Appendix A

The range and the standard deviation values of the slack time measured between the execution of the Low Priority Task and the following TIMER1 Interrupt were observed to study the jitter: the range is 87 cycles and the standard deviation is 20.3 cycles. Also these values show a negligible jitter. Furthermore the values range of the cycles waiting for instructions miss is 71 cycles for the activation time and 42 cycles for the context switch time, revealing that the measured jitter may be mostly caused by hardware execution.

4.4.2 Improvements

The obtained results show that some improvements may be done on the FreeRTOS implementation for GAP8. The main optimization can be carried out by storing the FreeRTOS variables, lists and structures in the L1 to be accessed faster (considering the high percentage of load stalls). The instructions miss overhead is instead more related with compiler and hardware optimization.

We sent this analysis to the GreenWaves developers and then the test was executed again on the SDK 3.2_dev. The new measured activation time occupies **1319 cycles** in average (27% cycles improvement) and the context switch time **1233 cycles** (15% cycles improvement).

The results demonstrate that optimization is possible, and better performances may be achieved with a better hardware structure. This thesis is part of a PCS hardware-software co-design project, and the test highlighted the necessity of a L1 memory with bigger dimension and a FreeRTOS different implementation to avoid as much as possible instruction fetch misses and load stalls.

4.5 Measurement of Control Execution Time

In this paragraph the execution timings of the power control structure are analyzed to study the performance of the code implementation and develop

considerations on the right value for the periodicity of the control task, the frequency at which the PCS should run and if there is enough slack time to add further advanced control features. The **scalability** of the code is also a main objective of the test, since the control structure execution time is strongly related to the parameter of the number of cores to be controlled. To this purpose the test was repeated for a different number of cores, specifically for 1, 16, 36 (nominal) and 72 cores.

The measurements are divided in these sections:

1. **Alpha Reduction**: in this section the alpha reduction is computed. This section executes only if the Computed Power is greater than the maximum imposed power
2. **PID section**: in this section the PIDs for each core is computed and the output is saturated and added to the target power
3. Computation of **power bindings**
4. **Frequency computation** and final frequency binding computation

Other two timings were recorded that are not part of the control flow but belong to the periodic control task. These two sections of code manage the telemetry data recordings and the exchange of data and commands between tasks. They are:

- **Section A:** telemetry management. This part manages the telemetry of the cores to be sent to the OS when requested. This section has two separate cycles measurements because there is a part of code that only executes with a proper periodicity; the split is implemented to achieve better performances. The part that is dedicated to the collection of data always executes, the second part that manages the data and sends them to the readable shared memory, executes periodically to reduce the overhead due to the writing in the shared memory.
- **Section B:** in this section the periodic control task reads the commands such as Target Frequencies, Workload, Total Power Budget,

Core bindings and model coefficients. Then these data are managed for optimization and computation purposes. This section is executed with a periodicity equal to the TaskOS periodicity; when it is not executed, it occupies a negligible amount of cycles.

An indicative value is also provided for the model adaptation execution.

The test was repeated for 40 iterations, to get results reflecting the variability of timings that can be measured on a real-time scheduler. Since these tests are run with integers, the generated values has no mathematical nor control meaning, thus to achieve a worst case scenario the alpha reduction was forced to be always active and for binding section a constant vector was provided. The GAP8 speed was set to 250MHz which should be the nominal speed of the PCS.

4.5.1 Results

The results are shown in the Tables 4.6 and 4.7

	36 Cores (Nominal)	
	Mean	Range
Section A	3869	8
S. A Complete	9153	31
Section B	6696	795
Alpha Reduction	6156	57
PIDs Computation	7498	49
Power Bindings	2324	40
Frequency Comp.	4806	56

Table 4.6: Results of the Control Test (in cycles) - 36 Cores (Nominal)

The 36 cores test is the starting point of our analysis. It can be observed that the section A (telemetry management) has a high impact on the performances; the measurement with both part executing (with the data update in the shared memory) has a greater amount of cycles than the execution

	1 Core		16 Cores		72 Cores	
	Mean	Range	Mean	Range	Mean	Range
Section A	43	0	1719	9	7663	3
S. A Complete	1012	20	6736	27	17206	27
Section B	1960	73	4004	391	11120	1506
Alpha Reduction	879	79	2672	76	11439	77
PIDs Computation	1026	52	3892	61	14358	47
Power Bindings	841	53	1554	70	3661	47
Frequency Comp.	901	47	2673	60	8742	108

Table 4.7: Results of the Control Test (in cycles) - 1, 16, 72 Cores

of the 36 PIDs. Section B (command management and power computation) has a high number of cycles too. The computation of the PIDs along with the computation of alpha reduction takes a total amount of 13.650 cycles (379 cycles/Core). This is a meaningful parameter since it is the core control part and the value is a good result for a non-optimized code and memory usage. The computation of the power bindings has a low impact on the performances (about 65 cycles/core). The sum of all timings gives a total of about 36.634 cycles. Considering that the GAP8 runs at 250MHz, in a 500us period there are **125.000 cycles**: the control part of the 36-core test takes **29,3%** of the total available cycles.

Considering this percentage and the overhead derived for FreeRTOS management (1% for task activation, 1% for each context switch, plus some cycles related to the systick management plus the TaskOS cycles) at 250MHz there is a considerable amount of available cycles to perform advanced control algorithm or to increase the frequency of the periodic task. Further, the cluster of the GAP8 can add another performance boost.

The analysis can proceed by observing the 72-core and the 1-core tests: the 72-core test (along with the 16-core test) is useful to comprehend the scalability of the code with respect to the number of cores; the 1-core test can give information about the **fixed timings** of the code execution.

The sum of all timings of the 72-cores test is about 66.526 cycles, with an increment of 29.892 cycles with respect to the 36-cores test. The ratio

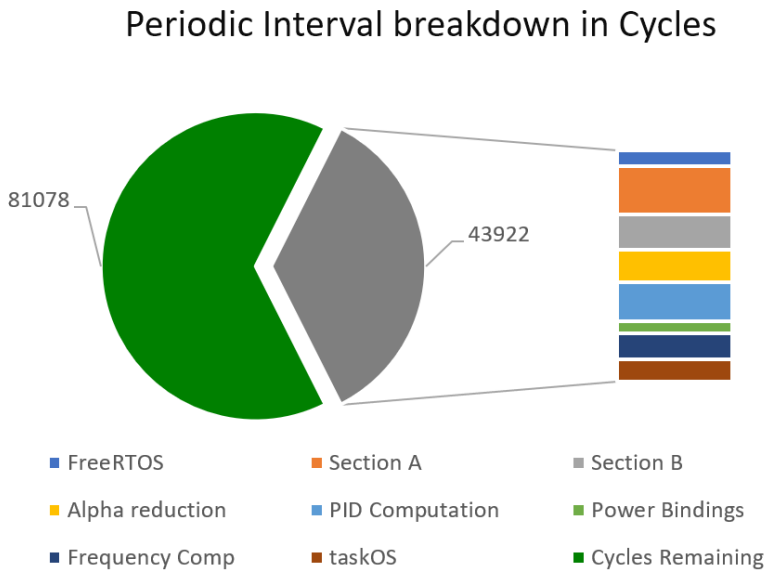

Figure 4.5: Distribution of cycles in a Control Period

between the timings is 1,816. A similar result is observable for the 16-core test (21.533 total cycles, ratio: 0,59). This is due to the fixed cycles (cycles to execute instructions that are not related to the number of cores) observable from the 1-core test data: comparing the fixed cycles to the cycles of the 36-cores test, the fixed cycles are the 11% of the total for the Alpha Reduction and PID part, but they reach the 27% for the Section B and 34% for the power bindings section.

Considering the total cycles and the average of the comparison of the tests, the scalability law is defined as:

$$Cycles = 5721 + 899 \cdot \#Cores \tag{4.3}$$

From the equation 4.3 an high value of fixed time is observable. It is equal to **4,5%** of the total cycles amount in the time period. The variable time is the **0,72%** of the total amount of cycles. To show that the obtained variable time value is good, consider a scenario in which the control structure has to run for only the 50% of the available time in the period (to allow other tasks

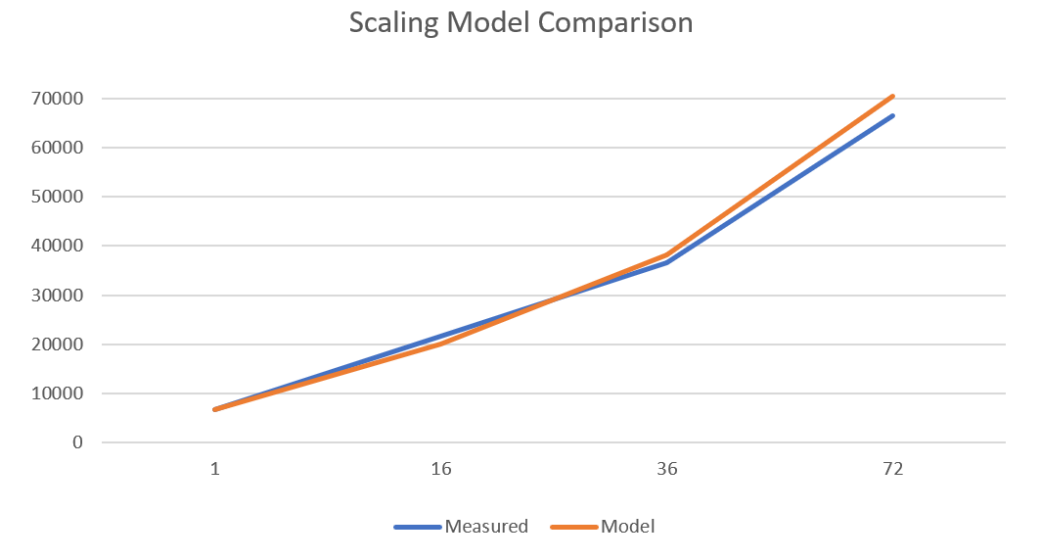

Figure 4.6: Cycles Scaling Comparison

to execute). The total number of cores that can be controlled is:

$$i_{Core} = \frac{50\% - 4,5\%}{0,72\%} = 63 \tag{4.4}$$

The TaskOS execution was also measured: it occupies $4310,41$ cycles with a range parameter of 10 cycles.

4.5.2 Improvements

The main concern analysing these data was related to the high value of the fixed timings and the high number of cycles needed to execute section A and section B parts. The focus was to improve the code to reduce these numbers.

As explained in section 3.4.4, the tasks use mutexes to access the shared memory. This action may strongly impact on the number of cycles (even in the case that the task is not blocked waiting for the mutex to be released). An improvement made was to create for each task variables that are copies of the shared memory structure. With this choice the memory footprint of

the program increases, but the performances (considering that the data are not to be fetched in the slower L2 memory) should increase since the shared memory is acceded only once per task iteration.

The new values are:

	36 Cores (Nominal)		
	New Opt.	Old	Improvement
Section A	7515	9153	17,10%
Section B	4343	6696	35,14%
PIDs Computation	4604	7498	39,60%
Frequency Comp.	6807	4806	-41,50%

Table 4.8: Improvement of Control Timings (in cycles) - 36 Cores (Nominal)

The performance improvement is remarkable, with the 17% and the 35% of less cycles in the Section A and section B respectively, and a 39,6% in the PIDs. The frequency computation part has 42% more cycles because it includes the final part of the loop where the shared memory is accessed through the mutex. Overall the amount of cycles is **31.750**, **13,3%** fewer cycles with respect to previous measurements.

4.6 Hardware in the Loop Test

The purpose of the test is to verify the results obtained in Matlab presented in section 4.3 and to test the correct implementation of the control structure on a real-time firmware.

4.6.1 Floating Point Test

A test similar to the one described in section 4.5 with 36 cores and floating point numbers was performed to determine the loss of performances that the simulated floating point arithmetic brings in GAP8: this analysis is necessary to evaluate a simulated time for the HiL test. The comparison is illustrated in Figure 4.8.

Figure 4.7: Picture of the Hardware in the Loop Test

The periodic task executes in 307.065 cycles i.e. $8,38$ **times** the cycles needed with integer computation.

4.6.2 SPI Performances

The Figure 4.9 illustrates the SPI communication. To understand the delay and the overhead that the SPI communication introduces, we need to consider:

- an average **delay** of $120us$ per transaction due to the structure with binary semaphores we implemented (the delay is showed in Figure 4.10;
- each transaction takes **32us** (32 bits at 1MHz);
- the periodic control task performs $2 + 2 + n_{core} \cdot 2$ SPI transactions every periodic execution due to the handshake structure described above (considering the cases in which the handshakes are achieved correctly);

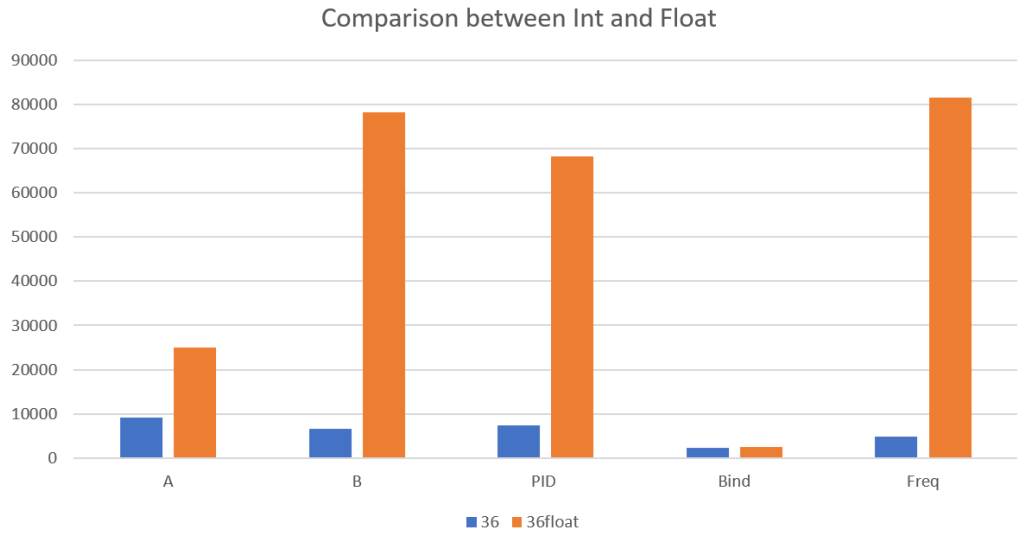

Figure 4.8: Comparison between cycles of the 36-Core Integer and Float Test

- the TaskOS performs $2 + n_{core} \cdot (3) + 1$ SPI transactions to read user input and send data;

Considering the nominal 36 cores, the periodic task introduces every periodic cycle as overhead of $76 \cdot 152 = 11.552us$, and the taskOS introduce $111 \cdot 152 = 16.872us$ every two periodic cycles.

4.6.3 Test Set-Up

From sections 4.6.1 and 4.6.2 the total amount of time necessary for each periodic interval has to be greater than:

$$T_{comp} > 307.065 * 2 \cdot 10^{-2} + 11.552 + 16.872 = 34.566us \tag{4.5}$$

For this reason the simulation is executed in a **simulated time** environment such that every **100ms** corresponds to a 500us periodic task cycle (200:1). The values are set in floating point, but they are converted to fixed 16-bits integers when transmitted through SPI. For timings computation reasons and

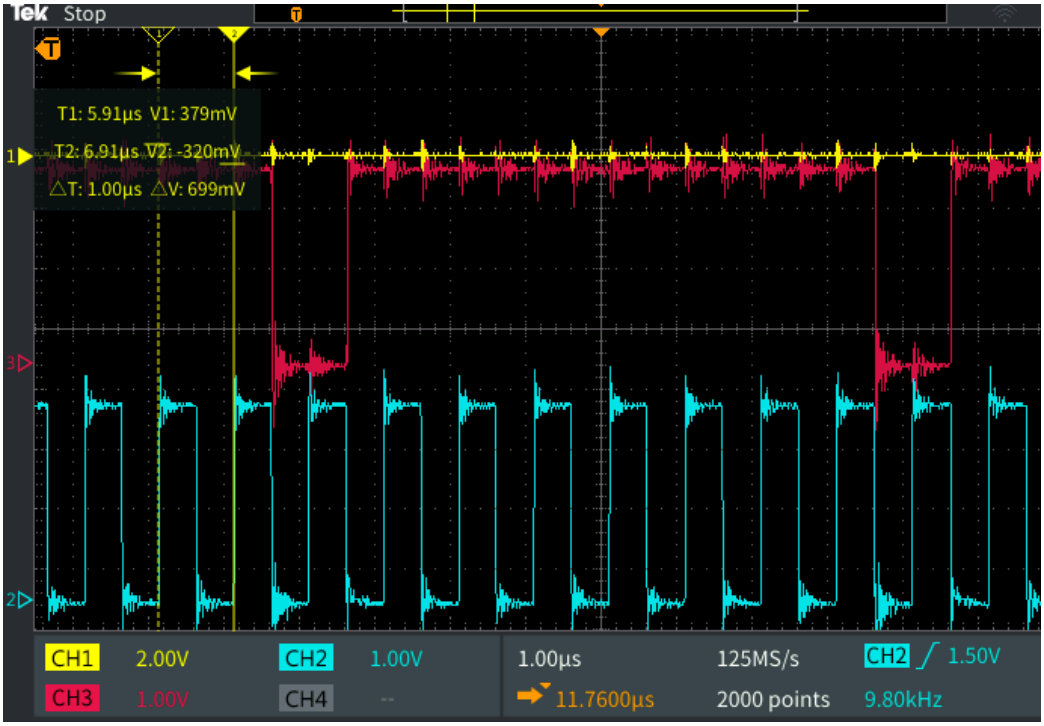

Figure 4.9: SPI Communication

connections overheads, the HiL test is performed with only 9 active cores to be controlled (they represent a quadrant of the hypothesized 36 cores chip).

The state matrixes for the model are obtained through a discretization of the Matlab thermal model with a sample time T_s based on the desired simulated time. The tests showed that the thermal model on the STM32 executes approximately 300 state-update steps every 100ms. This value is chosen as the simulated time of the periodicity of the control task based on the results obtained in 4.6.1 and taking into account the overheads introduced by the SPI analyzed in 4.1.3. The T_s is then computed as:

$$T_s = (300 \cdot \frac{100000}{500})^{-1} = \frac{1}{60000} \tag{4.6}$$

Two tests are performed: one to enlighten the thermal PID control and the other to observe the power capping policy. The inputs parameters are changed every 30s (simulated time). The applied inputs for the two tests are

Figure 4.10: SPI Delay between each SPI transaction

shown in Tables 4.10 and 4.10.

In the first test the maximum frequency and workload is given as input to each core to stress the thermal limit. The power budget in enough to not trigger the power capping control algorithm. In the second test a lower frequency is given to try not to saturate the thermal limit to better show the power capping policy. The power budget is reduced at input 3 and 9. In the second test the cores were also divided in two groups C_1 and C_2 to show the different control action: the first group has cores 0, 1, 3, 5, 7, 8, the second group has the cores 2, 4 and 6.

4.6.4 Results

The Results are showed in Figures 4.11 and 4.13. In Figure 4.11 we can observe the thermal control that avoid reaching temperatures higher than 78°C. The zoom in Figure 4.12 shows the oscilation at the limit tempera-

ture. In input 3 and 9 the frequencies (and thermal oscillations) are reduced probably because the higher workload parameter activated the power capping control algorithm.

In Figure 4.13 we can observe that the power capping algorithm meets the power budget constraints at inputs 3 and 9. At inputs 5 and 6 we can see that different workloads cause different cores temperatures and control actions. With these inputs the temperature oscillations are no longer present because the heat exchanged between cores is no longer enough to heat them above the imposed temperature limit.

# input	**Frequency** (GHz)	**Workload**	**Power Budget** (W)
1	$2,5$	$0,533$	22.50
2	$2,5$	1.244	22.50
3	$2,5$	1.778	22.50
4	$2,5$	1.244	22.50
5	$2,5$	1.244	22.50
6	$2,5$	1.244	22.50
7	$2,5$	1.244	22.50
8	$2,5$	1.244	22.50
9	$2,5$	1.778	22.50
10	$2,5$	1.244	22.50

Table 4.9: Input Paramters for each Core of HiL Test 1

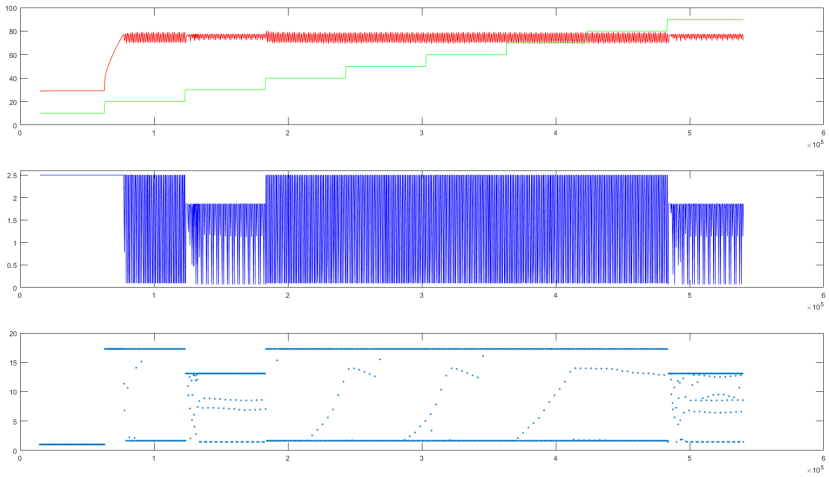

Figure 4.11: Results of HiL Test 1, Temperature (top), Frequency (mid), Total Power Consumption (bot).

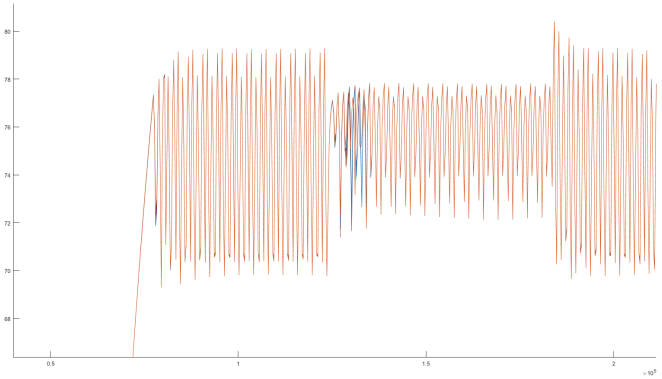

Figure 4.12: Results of HiL Test 1, Zoom on the Temperature

# input	**Frequency (GHz)**	**Workload C_1**	**Workload C_2**	**Power Budget (W)**
1	$1,8$	$0,533$	$0,533$	22.50
2	$1,8$	1.244	1.244	22.50
3	$1,8$	1.244	1.244	11.25
4	$1,8$	1.244	1.244	22.50
5	$1,8$	1.244	$0,533$	22.50
6	$1,8$	1.244	$0,533$	22.50
7	$1,8$	1.244	1.244	22.50
8	$1,8$	1.244	1.244	22.50
9	$1,8$	1.778	1.244	0.80
10	$1,8$	1.244	1.244	22.50

Table 4.10: Input Paramters for each Core of HiL Test 2

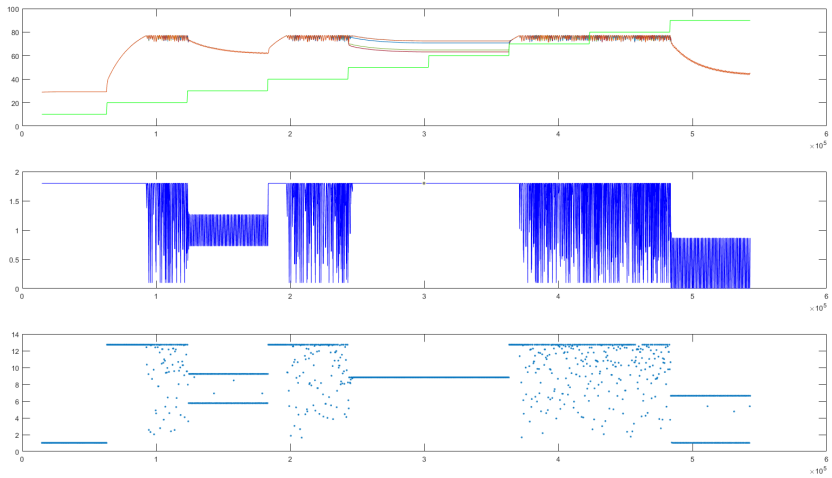

Figure 4.13: Results of HiL Test 2, Temperature (top), Frequency (mid), Total Power Consumption (bot).

Conclusions

The results obtained in Chapter 4 revealed that a thermal and power capping control is deliverable by a RISC-V based microcontroller with a ROTS software layer. The overheads introduced by the RTOS structure are not negligible but can be optimized, and at higher clock speed of the microcontroller they are not too significant. The experiments proved high flexibility and modularity regarding the firmware code structure. In future works, additional features and advanced control policies can be implemented effortlessly and there is a large available computation time in the 500us control period.

In this project I developed the first (or one of the first) PCS for a RISC-V based SoC. The obtained results and the numerical analysis may lead to a larger adoption of this architecture. The PCS implementation on the GAP8 board delivered the expected control in the expected timings even if the test was done in a simulated environment. The potential for the HPC market is there, and heterogeneous CPU and boards can exploit the flexibility and the higher performance of multicore RISC-V PCS.

The problems and bugs encountered in this thesis, especially in setting up the HiL experiment, reveal that the ecosystem is still immature. Projects like this one can help fill the gaps and the open-source philosophy helps on the matter too.

Future directions of works are towards the optimization of RTOS implementation (mainly regarding instruction fetch miss and load stalls) and implementation of advanced control algorithms modules to add to the PCS firmware. In the control solution we proposed a sub-optimal control policy;

we justify it with the high computation requirements that optimal solution to an optimization problem requires. Further tests may analyze the performance impact of online optimization solutions executed on a PULPissimo microcontroller. If the implementation is feasible an interesting future work could be to develop an optimized optimal control algorithm that exploits the parallel computation of the PULP cluster.

The code I developed for the PCS firmaware can be found in the `git.eees.dei.unibo.it/giovanni.bambini/epi_pmu` GitLab repository.

Appendix A

Lesson Learnt

GAP8, still a work-in-progress platform

Every electronic product is not a standalone, but it is inserted in an ecosystem made of software, application layers (SDK, Toolchain, IDE and API) and also made of user experiences and posts on forums where users share problems, bugs and fixes. GAP8 is a new product, but not only it is missing the ecosystem, it is also lacking almost any documentation or some important notes on how to develop application based on FreeRTOS. So, I want to dedicate a section of this thesis to talk about all the problems that I faced, in the hope that the ones who will come next may have to go through less debug sessions.

The first thing to point out is that if you have a problem installing the SDK, with the installation blocking at the step

```
[ 33\%] Performing download step (git clone) for 'ftdi'
```

this is probably caused by the fact that the installation performs a check on the internet network you are using to install the SDK, and some networks will block the connection if this kind of check is performed. To overcome this problem you just try on another network that does not block the connection.

SDK 2.7

I started developing on GAP8 with the SDK version 2.7. In this SDK FreeRTOS was implemented directly on top of GAP8 hardware, as it should be, but the hardware-dedicated APIs (such as SPI, I2C, Timers, DMA, ...) were not the same as the basic GAP8 APIs: they were rewritten for FreeRTOS without bringing any optimization or improvements correlated to the structure of the RTOS[1].

The first main bugs I encountered were related to the **Timer ISR** and the **priority management**. On GAP8 there are 2 64-bit Timers, one on the Fabric controller and one in the cluster. Each of these timers is divided into two 32 bit timers. We are now focusing on TIMER0 and TIMER1 that are the two 32 timers of the fabric controller. FreeRTOS uses TIMER0 for its own systick generation, so even if there are API for TIMER0, avoid them completely because they will crush the FreeRTOS firmware when called.

We wanted to program a basic test, in which a timer interrupt periodically awakes a blocked task through the usage of FreeRTOS Notification system 3.4.3(it can be also done with semaphore). The timer interrupt service routine suffered of a bug that prevented FreeRTOS to execute properly. The bug was described by the developers as "the priority is switched from Machine mode to User mode and there are problems with MRET when returning from IRQ handler" and it was fixed. This issue spawn in the SDK 3.0 a **personalized timer API** to set up this timer interrupt behavior. Also a **documented example** is now present in the SDK.

Another important issue was the printf function. In the SDK 2.7 the printf has issues that makes the code showing peculiar and inconsistent results. Sometimes the FreeRTOS scheduler execution doesn't start if there is a printf in specific positions of the code, other times it starts only if there is a printf in some proper location. I found that the best practice to adopt here is to store all the data that are desired to be printed in a huge array and execute

[1]When in this chapter I talk about APIs, I am referring to the specific FreeRTOS implementation of those APIs.

a time-gated test. When the test is finished, all the tasks but one should be suspended. Then the remaining task may perform a printf of all the collected data (in our case was a simple printf inside a for(), cycling for the number of data collected).

The printf behavior was fixed in the SDK 3.0. Also is worth considering that some of the unusual behaviors described above may had been originated from other bugs that are described below.

The next challenge was the SPI. It took more than two full months of work to both understand how the SPI APIs works on GAP8 and to fix various bugs that the SPI implementation had throughout SDK versions that were updated to fix the bugs I was finding.

To use the SPI on GAP8 you need to program the uDMA to perform the transaction. The main problem in the SDK 2.7 was caused by the names and the descriptions of the functions (and the lack of documentation or examples). For example the API function SPI_MasterTransferBlocking doesn't actually send anything, because to send you need first to configure the uDMA by calling multiple SPI_MasterTransferBlocking by following a precise structure ending with an EOT. Some pre-configured messages (as SOT, CFG and EOT) are present in the SDK, if you want to grep them navigating through files. Otherwise the actual SPI functions you need to use are: This problem is no longer present in SDK versions starting from 3.0, where the uDMA configuration for SPI transaction is hidden inside the APIs.

The SPI on version 2.7 suffered of a lot of other issues as the presence of spurious bad signals in both clock and MOSI signals and other configuration bugs. The main issue was that the SPI makes the system crash in a lapse of time that varies between 1 and 4 minutes. These and other undefined SPI bugs were fixed in the version 3.2 of the SDK.

I want also to note that SDK 2.7 had issues with GPIOs: I couldn't find a way to use more than one GPIO. When more than one GPIO were used, the output signal of each GPIO was depending on both GPIOs state, as they were interfering with each other. I think this issue was related to uDMA

bugs, and it was aggravated by the contemporaneous utilization of the SPI (which also utilizes the uDMA), resulting in counterintuitive behaviors of the code.

Ultimately it seems there are also some inconsistencies with the compiler which appears to delete some important part of the code trying to optimize it. I tried to remove the optimization in the GCC options, but the code would stop compiling returning errors. This issue is also present in the versions 3.* of the SDK.

Because of all these issues, but mainly for the uDMA critical bugs, I recommend not to use versions before the 3.2 if you want to use FreeRTOS on GAP8. Also SDK versions before 3.0 are no longer supported but still contains these bugs.

SDK 3.*

The versions 3.* of the SDK fixed multiple bugs, related mainly with the uDMA queue. Starting from the 3.0 version Greenwaves developers unified all APIs (they have also other two layers called PulpOS and MBED) under common layer called PMSIS. In my opinion few questionable choices regarding the GAP8 FreeRTOS implementation were made in the implementation of the PMSIS layer:

1. the Idle task of FreeRTOS seems to never execute; as stated in FreeRTOS documentation the idle task has an important role, and it should execute sometimes to clear destroyed resources[2]. For this reason in this FreeRTOS implementation it is recommended to not delete tasks or other FreeRTOS objects;

2. from FreeRTOS examples (and from other sample code that was sent to us by email from the developers) it seems that tasks creation and hardware initialization is preferable to be made after the FreeRTOS

[2] `https://www.freertos.org/RTOS-idle-task.html`

scheduler has started. So instead of executing the initialization steps and task creations in the main, you have to transfer all the code into a "main function task" that is going to be passed to the pmsis_kickoff() function. This API function will create a task with priority (tsIdleTaskpriority + 1) to execute the code of "the main". This is a questionable choice, mostly from a safety and determinism point of view, and because the priority of this "main task" is the lowest above the idle task, as soon as a task is created, if pre-emption is enabled it will pre-empt this main task stopping the initialization;

3. from FreeRTOS examples (and from other sample code that was sent to us by email by developers), the main task (the one called by pmsis_kickoff() is never suspended or blocked, but is utilized to check for end of application (that is a thing that should never occur in a FreeRTOS application that is generally suppose to run forever).

4. the developers used tickhook and idlehook functions to implement code, forcing the user to keep these two functions active.

Further, the fixes that this SDK version brought enlightened new issues that probably were also present in the previous versions, but they were hidden by other problems and bugs. The first is a bug related to the stack overflow signaling: when all tasks are in blocked state (for example waiting for a time-driver or event-driven activation), the FreeRTOS application execution will be terminated with the error: "Stack Overflow : -taskName- ! Exiting !" if the configCHECK_FOR_STACK_OVERFLOW in FreeRTOSConfig.h is active, otherwise it will stop returning "commands completed".

The developers said that this problem is generated if the parameter portMAX_DELAY is passed to a blocking API. In that case the task goes into a "suspended state". In their implementation of FreeRTOS they activated the tickhook function and implemented code in it. This code executes a check for suspended tasks "and if suspended tasks are equal to the number of tasks you defined in main application(value given for TaskHandle_t tasks[NBTASKS +

X], NBTASKS is equal to 1(for Idle Task), and the X value is for number of user task), it simply exits(since Idle is the only one running, meaning other tasks have probably ended running). What happened is that -all tasks- were suspended, then Idle Task is executing and it deletes user tasks before exiting. But just before exiting, Timer IRQ comes and tries to unblock a blocked task, which has been deleted by Idle Task. And that is why you have a Stack Overflow for user task, because its allocated memory space has been taken away by the scheduler."[3].

To solve this problem they suggested two alternatives:

1. disable Idle Hook in FreeRTOSConfig.h by setting the parameter #define configUSE_IDLE_HOOK to 0.

2. in the `/demos/gwt/gap8/common/config_files/FreeRTOS_util.c` file delete the tickhook code that is responsible of this behavior

Another issue is present if you try to use openOCD option. Regarding the UART printf, which with microcontrollers is often a simple and preferred method to debug your application, the developers are switching from the debug bridge, which they say is way too aggressive and disturb things in preemptive systems like FreeRTOS by polling on the JTAG way too hard[4], to OpenOCD. After installing openOCD following the instructions on the site[5], to enable it add the flag io=host in your Makefile. If you encounter the error message:

```
Open On-Chip Debugger 0.10.0+dev-gf77209d5 (2019-12-16-10:10)
Licensed under GNU GPL v2
For bug reports, read
                http://openocd.org/doc/doxygen/bugs.html
embedded:startup.tcl:26: Error: Can't find -f
in procedure 'script'
at file "embedded:startup.tcl", line 26
```

[3]Developer mail
[4]Developer mail
[5]`greenwaves-technologies.com/setting-up-sdk/`

to fix it you need to additionally source the file configs/gapuino.sh .

Finally, if you are trying to go around this `pmsis_kickoff` main task initialization structure (I tried and it seems to work fine), I suggest to modify the SDK code by commenting the xTaskCreate instruction in the `./pmsis/pmsis_backend/pmsis_backend_native_task_api.c` file. Other procedures may interfere with the correct execution of your freeRTOS application. An example is that if `pmsis_kickoff` is never called, but instead the function vTaskStartScheduler() is simply called in the main (note that this should be the common procedure), any printf call after the scheduler has started will result in the crash of the application.

If otherwise you continue use their procedure with this main task spawned by the `pmsis_kickoff` function, I still suggest to change the priority at which this main task is created, by modifying the value in the file `./pmsis/pmsis_backend/pmsis_backend_native_task_api.c`. If you are changing the priority, you still need to fix one further issue with the priorities that is illustrated in the next paragraph, otherwise the application will not work properly, will not start or it will crash.

FreeRTOS priority system

The FreeRTOS priority system allows to develop real-time application with complete interrupt nesting implementation. In some hardware portings interrupts are not completely disabled even inside critical sections[6]. In the FreeRTOSConfig.h file of GAP8 implementation there are two parameters: `configKERNEL_INTERRUPT_PRIORITY` and `configMAX_SYSCALL_INTERRUPT_PRIORITY`. The first sets the interrupt priority used by the RTOS kernel itself. Interrupts that call FreeRTOS API functions must also execute at this priority. The second parameter defines the priority at which the interrupts that use the safe version of the API (the ones that have "FromISR" in their name) can be executed. Interrupts that do not call API functions can execute at higher priorities and therefore never

[6] `freertos.org/a00110.html#kernel_priority`

have their execution delayed by the RTOS kernel activity (within the limits of the hardware itself).

In the GAP8 FreeRTOS implementation, in the file `./pmsis/pmsis_backend/implementation_specific_defines.h` is defined the parameter `PMSIS_TASK_EVENT_KERNEL_PRIORITY` which is set to 2. I don't understand what this parameter actually does, but I found out that the maximum priorities possible in the GAP8 FreeRTOS implementation is 3. Considering that the 0 priority is the idle task and that at 1 priority you have this main task called by `pmsis_kickoff`, you have only 1 priority left to use for your application.

If you want to achieve a deeper range of value, not only you need to change the `configMAX_PRIORITIES` parameter inside the FreeRTOSConfig.h file, but also you have to change the value of `PMSIS_TASK_EVENT_KERNEL_PRIORITY` such that it is equal or greater than the value set on `configMAX_PRIORITIES`

spi

Since we started using the 3.2 SDK versions, we found several bugs relative to the SPI configuration: the SDK was updated at least four times to fix them. If in your application you are facing inconsistencies in the SPI clock you may need to switch to the latest version of the SDK.

Further if you are using a GAP8v1 board you will probably see the SPI clock speed changing every time a new SPI configuration is set (either at run-time or when restarting the application or the board). This is caused by the function `pi_freq_get(PI_FREQ_DOMAIN_PERIPH)` that on this platform seems to return a random value every time is called. This is probably related to the PLL hardware bug present in the first version of GAP8 board, but it may also be a SDK misconfiguration of the registers.

To fix this issue, I modified the file `./pmsis/implem/pmsis_driver/spi` by setting the variable `uint32_t periph_freq` inside the function `static uint32_t __pi_spi_clk_div_get(uint32_t spi_freq)` to a fixed value (I chose 500000).

In the last dev version of the SDK (concurrently to the delivery of this thesis) the SPI has a huge dealay (around 10ms) per transmission if the blocking APIs are used. To overcome this problem, the correspondent async versions of the SPI API functions should be used.

Bibliography

[1] A. Bartolini et al., "A PULP-based Parallel Power Controller for Future Exascale Systems," 2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Genoa, Italy, 2019, pp. 771-774.

[2] Notes on EPI Thermal and Power Control - Christian Conficoni, Bologna, 2019.

[3] Real Time Engineers Ltd., The FreeRTOS Reference Manual. API funcions and configuration options, 2015.

[4] Richard Barry, Mastering the FreeRTOS Real Time Kernel. A Hands-On Tutorial Guide, Pre-release.

[5] Eastep Jonathan et al., "Global Extensible Open Power Manager: A Vehicle for HPC Community Collaboration on Co-Designed Energy Management Solutions", in Springer International Publishing, year 2017.

[6] `powerstack.caps.in.tum.de/proto.html`

[7] J. Doweck et al., "Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake," in IEEE Micro, vol. 37, no. 2, pp. 52-62, Mar.-Apr. 2017.

[8] M. Ware et al., "Architecting for power management: The IBM POWER7 approach," HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, Bangalore, 2010, pp. 1-11.