MASTER

Temporal-behavior evaluation of a network on chip

van Berkel, R.T.J.

Award date:
2015

Link to publication
Temporal-behavior Evaluation of a Network on Chip

Author:
ing. R.T.J. van Berkel

Host Organization:
Alten PTS
Beukenlaan 44
5651 CD Eindhoven
The Netherlands

Committee TU/e:
dr. ir. R.J. Bril (supervisor)
prof. dr. K.G.W. Goossens

Supervisor Alten:
ir. Bas van den Berg

December 12, 2014
Abstract

Within this thesis we evaluate whether or not an application with real-time requirements can run side by side with other applications on a commercial off-the-shelf (COTS) platform. This COTS platform exists of a multiprocessor system on chip, containing a network on chip (NoC) of which the temporal behavior is unknown. The focus of this thesis is on answering the question whether or not the NoC can provide temporal guarantees, such that the real-time requirements of the application can be met. The specific application considered in this thesis is a multicopter flight controller. We assign dedicated processing power and instruction/data memory to the control software, allowing us to focus on the NoC as the only shared resource between applications.

In this thesis, we first describe the resource requirements and timing requirements of the application and the deployment of the application on the platform. Next, we model the temporal behavior of the platform, in particular the NoC, and express the temporal behavior of the application in terms of the platform characteristics. The temporal behavior of the platform is subsequently evaluated by a set of experiments, using artificial load to simulate traffic on the NoC from other applications. Based on these experiments we conclude that the platform model properly describes the temporal behavior of the NoC. Moreover, we conclude that the NoC can provide temporal guarantees to the control application, i.e. the timing requirements of the control application can be met.
Acknowledgments

In the first place I would like to thank Alten PTS for the chance to put my knowledge into practice and offering me the wonderful opportunity to work as a consultant within the technical automation area. In particular I want to thank Bas van den Berg. I learned a lot thanks to Bas during my graduation and when I needed support, he was ready for me despite his busy schedule. In addition, I would like to thank Dr. R.J. Bril from the Technical University of Eindhoven for his scientific support and control of the graduation process at times when it was necessary.

Next, I would like to thank my beloved girlfriend, Maartje van Nistelhooy, for the support she gave me at times that I was a little less motivated due to setbacks. Last but not least I would like to thank my parents for their moral and financial support. Thanks to my parents, I was able to study, to develop myself and to prepare myself for the future. I am eternally grateful for that.
Alten

Alten PTS is a leading service provider in the field of technical automation.

Alten PTS has 250 employees in the Netherlands and is part of the international Alten group, which has 16,000 employees worldwide and has been active since 1988 in 15 countries.

Each year, Alten offers multiple students the opportunity to bring their knowledge into practice through an internship or graduation project. One of those graduation projects is described within this thesis.
4 Platform model and response time estimations

4.1 Platform model

4.1.1 No shared resources

4.1.2 Gyroscope and accelerometer

4.1.3 Magnetometer

4.1.4 Motors

4.1.5 I2C bus

4.1.6 NoC

4.1.6.1 L4_PER interconnect

4.1.6.2 L3 interconnect

4.1.6.3 NIs and the bridge

4.1.7 GPIO

4.1.8 M3 subsystem

4.2 Deployment Application Q

4.2.1 Execution time $\tau_{Q-reading}$ and $\tau_{Q-writing}$

4.2.2 Execution time $\tau_{Q-processing}$

4.2.3 Execution time $\tau_{Q}$

4.3 Event response time Application Q

4.3.1 Best-case event response time

4.3.2 Worst-case event response time

4.4 Deployment Application G

4.5 Event response time Application G

4.5.1 Best-case event response time

4.5.2 Worst-case event response time

4.6 Summary and findings

5 NoC latency experiments

5.1 Setup and RTT experiment concept

5.2 Experiment for model validation

5.3 Frequency scaling experiment

5.4 Simulated load experiment

5.5 Use case experiment

5.6 Summary and findings

6 Conclusion

6.1 Future work

A Quadcopter hardware

A.1 Frame (a)

A.2 Motors (b)

A.3 Propellers (c)

A.4 ESCs (d)

A.5 Battery (e)

A.6 Components (f) - (i)

A.7 Sensor board (j) and Pandaboard ES (k)

B Hardware used during experiments

B.1 Pandaboard ES (OMAP4460)

B.2 Micro SD card plus adapter

B.3 Laptop

B.4 USB to serial converter

B.5 Wave generator

B.6 Logic analyzer

C Software used during experiments
### List of Figures

1. Alten logo ................................................................. 2

2. Picture of the composed quadcopter .................................. 11
2.1 Picture of the composed quadcopter .................................. 11
2.2 Quadcopter protection cover ............................................ 12
2.3 Simplified hardware overview ........................................... 12
2.4 Revised attitude hold algorithm ....................................... 13
2.5 Quadcopter on experimental setup .................................... 13

3. Multiple-byte write and read I2C messages .......................... 16
3.1 Multiple-byte write and read I2C messages .......................... 16
3.2 Pandaboard ES Top View [25] ........................................ 17
3.3 OMAP4460 Architectural Block Diagram [7] ......................... 18
3.4 L3 Interconnect Architectural Block Diagram [9] ..................... 18

4. Architectural overview .................................................... 23
4.1 Architectural overview .................................................. 23
4.2 L4_PER packetization [29] ............................................ 25
4.3 Logical path from the M3 subsystem to the sensors/motors ...... 26
4.4 Overview of NoC components and corresponding packet transfer time . 27
4.5 Architectural overview including event and response, Application Q ........................................ 28
4.6 UML sequence diagram: response on environmental event .......... 29
4.7 time line: best-case response on environmental event (an optimal instant) . 30
4.8 time line: worst-case response on environmental event (a critical instant) . 31
4.9 Logical path from the M3 subsystem to the GPIO module .......... 32
4.10 Architectural overview including event and response, Application G ........................................ 33
4.11 UML sequence diagram: response on GPIO input change .......... 33
4.12 UML sequence diagram best-case event response time ............... 34
4.13 UML sequence diagram worst-case event response time ............. 34

5. RTT $t_1 - t_0$ in the case of rising edge detection .................. 36
5.1 RTT $t_1 - t_0$ in the case of rising edge detection .................. 36
5.2 Experimental setup .................................................... 37
5.3 Result RTT control experiment ........................................ 37
5.4 Simulated load experiment overview .................................. 39
5.5 Expected L3 interconnect architecture ................................ 40
5.6 Use case experiment overview ........................................ 41

A. Quadcopter hardware components ..................................... 46
A.1 Quadcopter hardware components ..................................... 46

B. Experimental setup ..................................................... 51
B.1 Experimental setup ..................................................... 51
List of Tables

3.1 Partial L3 interconnect connectivity matrix .............................................. 19
4.1 M3 instructions Application G ................................................................. 32
5.1 Result RTT control experiment ................................................................. 38
5.2 Frequency scaling results ......................................................................... 38
5.3 RTT Application G with simulated load on the NoC .................................... 40
5.4 Result use case experiment ......................................................................... 41
A.1 NTM Prop Drive 28-30S 800KV / 300W Brushless Motor (short shaft version) specifications table ................................................................. 48
## List of abbreviations

- **CPU**: Central Processing Unit
- **COTS**: Commercial Off-The-Shelf
- **DSP**: Digital Signal Processor
- **EEPROM**: Electrically Erasable Programmable Read-Only Memory
- **GPIO**: General Purpose I/O
- **I²C**: Inter-Integrated Circuit
- **L1**: Level 1
- **MMU**: Memory Management Unit
- **MPSoC**: Multi Processor System on Chip
- **MPU**: Multi-core Processing Unit
- **NoC**: Network on Chip
- **OCM**: On-Chip Memory
- **OS**: Operating System
- **PID**: Proportional, Integral and Derivative
- **PWM**: Pulse-Width Modulation
- **RAM**: Random-Access Memory
- **ROM**: Read-Only Memory
- **SAR**: Save And Restore
- **SRAM**: Static Random-Access Memory
- **RTT**: Round-Trip-Time
- **TDM**: Time Division Multiplexing
- **TLB**: Translation Look-aside Buffer
Chapter 1

Introduction

1.1 Background

Over the years so-called multicopters have become more and more popular. A multicopter is a type of unmanned aerial vehicle (UAV) which is lifted and propelled by multiple propellers. Thanks to their maneuverability and stability characteristics, multicopters are used in many different fields of expertise nowadays. In order to guarantee both maneuverability and stability, algorithms are being developed and provided to the end user in the form of multicopter controller boards. Today a variety of commercial off-the-shelf (COTS) controller boards are available.

COTS multicopter controller boards are often dedicated to cost-efficiency and do not allow integration of additional functionality because of their lack of computational resources [3, 4, 5]. E.g. one might want to extend the controller with a vision algorithm such that environmental information can be captured. In practice one often has to add processing power and memory capacity in such a situation.

Alten is building a multicopter from COTS parts and typical multicopter control software is being implemented on a heterogeneous MPSoC called the OMAP4460. The control software is considered a real-time piece of software. From this point we will refer to the real-time multicopter control software as Application Q. Once finished, it should be possible to integrate additional software on the OMAP4460 based controller board, but also to guarantee that Application Q will never miss its execution deadlines.

1.2 Context

In order to allow (real-time) applications to run side by side on a single MPSoC, one typically applies one or more of the following strategies: one either implements virtualization and/or partitioning of the platform or one simply treats all the applications as if they are all real-time. In [33] both dedicated and shared resources are partitioned into so-called virtual platforms. Each virtual platform provides a guest OS or just a single application with virtualized platform interfaces. The virtual platforms are scheduled using a strict time division multiplexed (TDM) scheduler. The approach allows real-time applications to be verified and executed in isolation, on a mixed time-criticality system. However, for creation of predictable virtual platforms, the approach assumes predictable (temporal) behavior of underlying hardware, which is not the case in our situation with the OMAP4460. In [34], researchers show that it is possible to run a non real-time OS and a real-time OS side by side on the OMAP4460 by means of virtual partitioning. The two OSs use their own dedicated processor core for execution. The real-time OS uses a piece of dedicated memory for the timing critical instructions and data. Furthermore, both OSs use their own non-shared peripherals. MMUs and caches are disabled to prevent non-deterministic behavior. However, as indicated in the paper, the NoC, used for communication between the processor cores and memories for example, is shared between multiple hardware components. The (temporal) behavior of the NoC may impair the real-time ability of the real-time OS. Hence, only taking shared processors and memories into account is not sufficient, a closer analysis of the (shared) NoC is required.

Within the real-time community most research on schedulability of mixed time-criticality systems only takes shared processors and memories into account, while only very little work focuses on having a shared NoC. In [35], an off-line schedulability analysis approach is presented which allows prediction of the worst-case network packet latency. In [36], the researchers propose an analytic model which is able to predict the average packet latency and router blocking time. And finally, in [37], simulations are performed to evaluate the performance of the NoC while injecting various types of traffic into the NoC.
All three approaches allow a certain evaluation of packet latency's in various types of NoCs. However, all three approaches have something in common. All three require knowledge about the NoC under evaluation. The approach described in [36] for example takes as input an application communication graph, a topology graph, a mapping vector, and a routing matrix. In our situation where we are using the NoC from the OMAP4460, we are missing crucial information related to the NoC. The available information is insufficient to derive a model of the temporal behavior of the NoC and it is not possible to apply the just described NoC packet latency evaluations. Hence, we have to come up with a different approach to evaluate the NoC packet latency.

1.3 Problem definition

We want to integrate multiple pieces of software, including Application Q, on the OMAP4460. Both Application Q and the other pieces of software have to use the OMAP4460 NoC. However, it is not known how much time the NoC needs to deliver packets. Moreover we do not know to what extent the packet latency may increase when multiple pieces of software use the NoC simultaneously. I.e. at this time we cannot guarantee that Application Q will never miss its execution deadlines.

1.4 Goal

The main goal of the thesis is to answer the following main question: can we guarantee that the temporal requirements of Application Q will always be met, independent of future functionality that will be integrated on the OMAP4460? From the main question we derive the following sub-questions:

1. Given the available information about the OMAP4460, can we derive a platform model which includes the expected temporal behavior of the NoC?

2. Given the platform model and the deployed Application Q, can we derive an estimation on the worst- and best-case response time of Application Q in isolation? I.e. we assume only Application Q is deployed on the platform, what are the timing bounds?

3. Can we validate the platform model and the derived response time estimations by means of experiments?

4. To what extent does potential additional traffic on the NoC affect the response time of Application Q?

5. Are the timing bounds of Application Q sufficiently tight such that we can guarantee that the timing requirements of Application Q are met?

1.5 Approach

First we collect minimal information about the multicopter hard- and software such that we know how the OMAP4460 communicates with external hardware for example and what the characteristics of Application Q are. Next, we deploy Application Q on the OMAP4460 in such a way that we avoid the temporal consequences of having shared processors and memories. This leaves us with the task to evaluate the (temporal) behavior of the NoC. Then, we introduce both a platform model and a model of Application Q deployed on the platform. The platform model includes an approximation of the NoC (temporal) behavior. We base the platform model on the available information about the OMAP4460. Based on the model, we derive the worst- and best-case response time of Application Q. Afterwards, we validate the application model and the response time estimations by means of hands-on experiments. So-called round-trip-time (RTT) experiments are performed to evaluate the temporal behavior of the NoC. I.e. we measure how much time is required to send packets over the NoC and back. Last but not least we introduce artificial load on the NoC in an attempt to interfere with the traffic from Application Q. We perform this final experiment to determine to what extent potential additional traffic on the NoC affects the response time of Application Q. We use the results of the experiments to answer our main research question.
1.6 Chapter overview

In Chapter 2, we provide information about the multicopter hard- and software. In Chapter 3, we discuss the OMAP4460 and we explain how Application Q will be deployed such that we avoid the temporal consequences of having shared processors and memories. Then, in Chapter 4, we derive the platform model and estimations on the worst- and best-case response time of Application Q. In Chapter 5, we describe the results from experiments that have been performed to validate the platform model and to determine to what extent potential additional traffic on the NoC affects the response time of Application Q. The thesis concludes with an interpretation of the results and future work.
Chapter 2

Multicopter hard- and software

In [1], sections 2.1 and 2.2, the state-of-the-art with respect to a specific multicopter type, called a quadcopter, is described. It was earlier investigated what is important to keep in mind when composing a quadcopter from COTS hardware components and it was investigated how the software, Application Q, should look like. This chapter provides a brief description of the actual composed quadcopter, developed Application Q and the tuning and testing procedure. Note that in [1], a number of requirements related to the quadcopter hard- and software is listed. Due to the fact that the focus of our work during the project shifted towards deployment of Application Q instead of development, the quadcopter hard- and software requirements were relaxed to nice-to-haves.

2.1 Hardware

In Figure 2.1 a picture of the composed quadcopter is presented.

![Figure 2.1: Picture of the composed quadcopter](image)

The span of the quadcopter is 70cm (from propeller tip to propeller tip) which allows the quadcopter to maneuver through an open door which is typically 90cm wide. Furthermore, the COTS hardware components are selected as such that the quadcopter should be able to carry an additional weight of 1kg and should be able to achieve a flight time of approximately 15 minutes (without the additional 1kg of weight). Mention that no safety features are yet implemented, instead a CAD design was created for 3d printing of protection covers for the propellers. The protection covers should decrease the probability that the quadcopter can cause any injury or damage to other people and materials. The CAD design of the covers is presented in Figure 2.2.

In the next subsections we will discuss the following hardware components; (1) the sensors, (2) the actuators, (3) the controlling system and (4) the \( I^2C \) bus which interconnects the components. Note that the quadcopter actually contains a lot more hardware components. For more information about the hardware components and the hardware components which are not mentioned here, please refer to Appendix A.
2.1.1 Sensors

For the purpose of our project we use three different sensors which are integrated on a single PCB. The PCB can be accessed via I²C communication. The three different sensors include a triple-axis gyroscope, triple-axis accelerometer and a triple-axis magnetometer. The gyroscope measures an angular rate of change (°/s) in three directions (x, y, z). The accelerometer measures the acceleration in g-forces (g) the sensor endures in three directions (x, y, z). Together with the gyroscope, the accelerometer is used to determine the (relative) attitude of the quadcopter, expressed in roll, pitch and yaw. The third sensor, the magnetometer, measures the magnetic field of the earth (milli-gauss) in three directions and is used to determine the (absolute) heading of the quadcopter. All three sensors generate 16 bits of data for each direction at a configurable fixed timing interval.

2.1.2 Actuators

The quadcopter has four motors which are independently controlled by a pulse-width modulation (PWM) driver. The PWM driver is on its turn controlled via I²C communication. The PWM driver controls the power to the connected motor, the power is varied between 0 and 100%. Attached propellers transform the rotational speed of the motors into a downward force. Cooperation of the four motors make the quadcopter move into a certain direction. A controller should make sure the quadcopter moves in the desired direction.

2.1.3 Controlling system

A COTS development board called the Pandaboard ES, which contains the OMAP4460, was used to deploy Application Q on. Application Q does the processing of sensor data and controlling the power of all four motors. The OMAP4460 offers a lot of processing power, internal memories and a lot of connectivity options like the I²C bus. More information about the OMAP4460 and how we deployed Application Q can be found in Chapter 3.

2.1.4 I²C bus

The sensors, actuators and the controlling system are interconnected to each other by a communication bus called the I²C bus [24]. The three hardware components and the I²C bus are graphically represented in Figure 2.3.

![Figure 2.2: Quadcopter protection cover](image)

![Figure 2.3: Simplified hardware overview](image)
2.2 Original software

Application Q which runs on the Pandaboard ES must read data from the sensors and make sure that the power of each motor is continuously controlled in such a way that the quadcopter is able to maintain its current attitude in air. In [1] a quadcopter attitude hold algorithm was suggested which in short performs the following; (1) read sensor data, (2) fuse the sensor data to obtain the attitude of the quadcopter, (3) feed the result into the PID controllers to obtain the correct motor offsets and (4) send the sum of the offsets to the motors.

2.3 Revised software

The suggested attitude hold algorithm was incomplete. The suggested algorithm includes three independent attitude (roll, pitch, yaw) correcting PID controllers. Such a PID controller receives a desired attitude and the actual attitude of the quadcopter. The difference between the desired and actual attitude, the error, is calculated and, depending on the PID parameters, an offset for all four motors is generated. The sum of the three independent attitude offsets (roll, pitch and yaw) is then provided to the motors as set-points. The original software assumes that linearly increasing the set-point (or power) of a single motor does result in linear increased thrust, this is not the case. The actual thrust that is generated from the motors is dependent on a lot of factors. In other words, the thrust is not controlled and we have an open-loop system. In order to be able to control the thrust of the motors, to create a closed loop system, a slightly different attitude hold algorithm was implemented. The revised algorithm uses raw gyroscope data as feedback from the motors to control the thrust from each motor. An overview of one of the three controllers, in this case the roll controller, is provided in Figure 2.4. Together with the pitch and yaw controllers, all necessary offsets are generated for the motors. The roll, pitch and yaw offsets get summed and then send to the motors. The revised attitude hold algorithm is based on open-source software available from [2].

![PID-controller](image)

Figure 2.4: Revised attitude hold algorithm

2.4 Tuning and testing

For tuning and testing of the (revised) attitude hold algorithm, an experimental setup was used which is shown if Figure 2.5.

The figure shows the quadcopter which is fixed on a free rotating shaft from the experimental setup. The setup allows tuning of the roll and pitch controllers, one at a time. The yaw controller was not tuned nor checked. When tuned correctly, the quadcopter should be able to balance on the free rotating shaft.

In order to allow tuning and testing of the attitude hold algorithm, we have to execute the software on the Pandaboard ES. A pre-installed Ubuntu server image [19] was used to host the developed software.

![Quadcopter on experimental setup](image)

Figure 2.5: Quadcopter on experimental setup
on the Pandaboard ES. The Ubuntu server OS does not provide software with any temporal guarantees and sporadic delays may occur any time due to kernel tasks and background applications for example. However, in spite of the sporadic executional delays of the attitude hold algorithm, we were able to perform coarse tuning of the roll and pitch controller and to check the functioning of the attitude hold algorithm. It was checked whether the quadcopter was able to actually balance on the shaft. Since the main focus of this thesis is not on the development of quadcopter control software nor on the performance of the attitude hold algorithm, we did not derive any figures on the performance and we did no further testing and improvement of the software.
Chapter 3
Software deployment

So far, a quadcopter was built using COTS components and Application Q was developed and tested. The next step is to deploy Application Q on the OMAP4460 and focus on providing Application Q with temporal guarantees. In this chapter, an abstraction of the Application Q is formulated and a set of temporal and architectural requirements is described in Section 3.1. In Section 3.2 we give a brief overview of the OMAP4460 architecture which is located at the Pandaboard ES. Last but not least, in Section 3.3 the chapter concludes with a justification and explanation of the made decisions related to the deployment of Application Q.

3.1 Application Q

The name Application Q hides the purpose and the functionality of the application. In this section we describe the properties and requirements of Application Q we need in order to decide how to deploy Application Q on the OMAP4460.

3.1.1 Properties

Application Q implements an iterative execution of a number of sequential actions and computations. In short, Application Q reads data from a set of sensors, performs the necessary computations and then writes data to a set of actuators.

3.1.2 Temporal requirements

Based on open-source software [2], which includes a main control loop which runs at a 100Hz update frequency, we derived the following temporal requirements for Application Q. Application Q can be seen as a task $\tau_Q$, which is periodic released every 10ms ($T_Q = 10ms$) and has a relative hard deadline of $D_Q < T_Q$. Within the 10ms, Application Q has to read from the set of sensors, perform computations and then send data to the actuators. After the release of the $k^{th}$ instance of task $\tau_Q$, a.k.a. job $\tau_{Q,k}$, at time $a_{Q,k}$ (absolute activation time), the absolute finalization time $f_{Q,k}$ may be delayed for arbitrary time $t_D$ as long as the finalization time minus the activation time, is smaller than the relative deadline, $f_{Q,k} - a_{Q,k} < D_Q$. I.e. the response time of each job $R_{Q,k} = f_{Q,k} - a_{Q,k}$ should always be smaller than the relative deadline $D_Q$, $\forall k \in \mathbb{N} | R_{Q,k} \leq D_{Q,k}$.

3.1.3 Architectural requirements

The architectural requirements are split in three categories; processing, memory and I/O. All three categories are treated in the next subsections. Preferably, Application Q does not share any resources with other applications such that we avoid the need to resolve the temporal consequences of having shared resources.

3.1.3.1 Processing

The processing requirement is based on the specifications from COTS controller boards, e.g. [3, 4, 5], which contain (among other features) a functionality very similar to Application Q. Preferably a 32-bit RISC processor is used, which supports an operating frequency greater than $20MHz$ and has
single precision floating point (32-bit) support or (preferably) acceleration. Since the majority of the processing consists of calculations with floating point numbers, especially multiplications, the presence of single precision floating point (32-bit) support or hardware acceleration is no luxury. One might be able to omit using floating point numbers, but, as a consequence, precision might be lost for example.

### 3.1.3.2 Memory

Like the processing requirement, the memory requirement is based on the specifications of COTS controller boards. E.g. the controller board from \[4\] contains a 64kB programmable flash program memory. Since the COTS controller board implements more functionality than just Application Q, we used the figure as a requirement.

### 3.1.3.3 Peripherals

Application Q needs to read from a set of sensors and write to a set of actuators. Both the set of sensors and the set of actuators are connected to an \(I^2C\) bus and hence we need an \(I^2C\) controller at our disposal such that we can join the bus and exchange data with the sensors and actuators. The \(I^2C\) controller has to support at least two types of \(I^2C\) messages, which are represented in Figure 3.1.

![Figure 3.1: Multiple-byte write and read \(I^2C\) messages](image)

In the figure, the so-called multiple-byte write and read \(I^2C\) messages are represented \[24\]. The multiple-byte write message contains an \(I^2C\) address (8 bits), a register address (8 bits), 16 bits of data, 4 ACK bits, 1 start bit and 1 stop bit. In total the message is 38 bits long, see Equation (3.1). The multiple-byte read message contains two times the 8 bits \(I^2C\) address, a register address (8 bits), 16 bits of data, 4 ACK bits, 1 NACK bit, 1 start bit and 1 stop bit. In total the message is 48 bits long, see Equation (3.2).

Each execution of Application Q, nine 16 bits values have to be read from the set of sensors and four 16 bits values have to be written to the set of actuators. Application Q is periodic released with \(T_Q = 10\, ms\) and therefore the \(I^2C\) controller should support a transmission rate of at least 58300bps, see Equation (3.3).

In practice, one often distinguishes between standard (100kbps), fast (400kbps) and high speed (3.4Mbps) \(I^2C\) transmission rates \[24\]. In our case, the standard \(I^2C\) transmission rate is sufficient.

\[
data size I^2C write message = 8 + 8 + 16 + 4 + 1 + 1 = 38\text{bits}
\]

\[
data size I^2C read message = 2 \cdot 8 + 8 + 16 + 4 + 1 + 1 + 1 = 48\text{bits}
\]

\[
I^2C\text{ transmission rate } \geq \frac{1}{T_Q} \cdot (9 \cdot 48 + 4 \cdot 38) = \frac{1}{10 \cdot 10^{-3}} (9 \cdot 48 + 4 \cdot 38) = 58300\text{bps}
\]

### 3.2 Pandaboard ES

During the project we used the development platform called the Pandaboard ES which contains the OMAP4460. In this section we shortly introduce the Pandaboard ES and dive into the architecture of the OMAP4460. We explored the OMAP4460 in order to identify how Application Q can be deployed such that the earlier listed requirements can be met.
3.2.1 Top view

In Figure 3.2, a top view of the Pandaboard ES is shown. The Pandaboard ES provides the user with a lot of connection possibilities like USB, HDMI, SD card, LCD etc. and therefore the Pandaboard ES looks like a very complex general purpose machine. However, the most interesting part of the Pandaboard ES, the OMAP4460, is located just above the center of the board and is highlighted with a red box. The OMAP4460 is the brain of the Pandaboard ES, the rest of the components located at the Pandaboard ES are dedicated to power regulation and to provide access to as many of the features of the OMAP4460 to allow easy development. More information about the Pandaboard ES can be found in [25].

![Figure 3.2: Pandaboard ES Top View](image)

3.2.2 The OMAP4460

Figure 3.3 provides a block diagram of the OMAP4460 architecture.

The OMAP4460 consists of a NoC (highlighted in green), a number of computational subsystems (the dark blue rectangles) and a lot of other multi-functional hardware components (the light blue rectangles). The multi-functional components include on-chip memory, external memory interfaces etc. The subsystems contain one or more processes which can potentially be used to do the processing for Application Q and hence we had to select an appropriate subsystem to deploy Application Q on. The components from the OMAP4460 that were relevant while making the decision which subsystem to use to deploy Application Q on are described in the subsequent subsections. E.g. the audio back-end subsystem, abbreviated as ABE, is not described since we are not interested in processing audio. The descriptions are based on available information from the OMAP4460 technical reference manual (TRM). For more information refer to [7].

3.2.2.1 The OMAP4460 NoC

The NoC allows the subsystems and the multi-functional hardware components to exchange data. Without such an NoC, the computational subsystems would be useless since they would not be able to access data from a sensor or from a hard disk for example. The OMAP4460 NoC consists of two levels of interconnects which cooperate with each other. The two levels are called level 3 (L3) and level 4 (L4). The L3 interconnect consists of one interconnect, the L4 interconnect consists of five independent interconnects. The subsystems and the multi-functional hardware components are connected to the NoC by interfaces. Bridges connect the L3 interconnect with the L4 interconnects.

3.2.2.1.1 L3 Interconnect  The L3 interconnect is the main level interconnect from the OMAP4460 NoC and therefore the most important component of the NoC. An overview of the L3 interconnect and the components connected to it, either subsystems or other multi-functional hardware components, is given.
in Figure 3.4. Note that the components in bold are within the scope of this thesis and are described in the subsequent subsections.

![L3 Interconnect Architectural Block Diagram](image)

Figure 3.4: L3 Interconnect Architectural Block Diagram

In the figure we distinguish two types of modules that are connected to the L3 interconnect: (1) the initiator (or master) modules, highlighted in yellow, which are able to initiate read and write requests to the interconnect and (2) the target (or slave) modules, highlighted in white, which can only respond to requests delivered by the L3 interconnect. Note that there exist modules that are both initiator and target, e.g., the Dual Cortex-M3 subsystem.

Even though it looks like that each and every target module is reachable from all initiator modules, in practice it is not the case. Table 3.1 shows a selection of the functional paths that exists between the L3 interconnect initiator modules and the L3 target modules. Note that a cell contains a + sign when such a functional path exists. For the full connectivity matrix, refer to [9].

As can be seen from the table, not all modules have a functional path between them and hence not all target modules are reachable from each initiator module connected to the L3 interconnect. In the case of Application Q, a functional path to an I²C controller should exist.

The OMAP4460 contains four I²C controllers which are connected to one of the L4 interconnects called the L4 peripheral interconnect (L4_PER). The OMAP4460 has four separate bridges which connect the L3 interconnect with the L4_PER interconnect: L3_SN_L4_PER0, L3_SN_L4_PER1, L3_SN_L4_PER2 and L3_SN_L4_PER3. Hence, from the subsystem we choose to deploy Application Q on, at least one of those bridges should be reachable. Furthermore, the target modules called...
Table 3.1: Partial L3 interconnect connectivity matrix

L3_SN_L4_CFG and L3_MN_MMC should be reachable. The L3_SN_L4_CFG target module is a bridge which connects the L3 interconnect with an L4 interconnect called the L4 configuration interconnect (L4_CFG). Reachability of L4_CFG is required because it offers access to configuration registers from large number of hardware components. Reachability of the target module called L3_MN_MMC is required because it offers access to the off-chip SDRAM.

3.2.2.1.2 L4 Interconnects The OMAP4460 contains five independent L4 interconnects, we only consider two of them; the L4_CFG and the L4_PER. Both the L4_CFG and L4_PER can be accessed directly via the L3 interconnect. An overview of the connectivity is given in Figure 3.5. All the components in bold are within the scope of this thesis and some of them are described in the subsequent subsections. Others in bold are not discussed in this chapter, we discuss them later in this thesis.

3.2.3 A9 subsystem

The first component connected to the L3 interconnect we describe is called the Cortex-A9 subsystem. The A9 subsystem is the main processing subsystem of the OMAP4460. In practice, this subsystem is often assigned to boot and run Linux and eventually load firmware to other subsystems. Among others, the subsystem contains a dual core A9 processor, running at a maximum clock speed of 1.2GHz, which implement the ARMv7-A architecture profile [26]. Furthermore, the subsystem contains 32kB instruction and 32kB data level 1 (L1) caches per CPU core, shared 1MB level 2 (L2) cache, 48kB bootable read-only memory and vector floating point unit co-processors. For more information refer to [8].
3.2.4 DSP Subsystem

The second component connected to the L3 interconnect we describe is the digital signal processing (DSP) subsystem. The DSP subsystem includes a 32-bit fixed-point media processor running at a maximum clock speed of 720 MHz, 32kB L1 cache and 128kB L2 cache. The processor itself does not include an address translation unit and therefore, in order to be able to use all devices from the OMAP4460 connected to the NoC, the subsystem has a dedicated memory manage unit (MMU) for accessing L3 interconnect address space. For more information refer to [9].

3.2.5 M3 subsystem

Next to the A9 subsystem and the DSP subsystem, the OMAP4460 contains a subsystem called the Cortex-M3 subsystem. The M3 subsystem includes two M3 cores running at a maximum clock speed of 400 MHz, 32kB shared L1 cache, 64kB L2 RAM memory and 16kB bootable ROM. The two M3 cores implement the ARMv7-M [27] instruction set architecture and have 2 - 12 cycle 32-bit hardware division and single-cycle 32-bit multiplication computational acceleration. Like the DSP subsystem, the M3 subsystem contains a dedicated MMU for accessing L3 interconnect address space. For more information about the M3 subsystem refer to [10].

3.2.6 On-Chip Memory

The OMAP4460 on-chip memory space is hierarchical and consists of the levels L1, L2, L3 and L4. The L1 and L2 memories can be found within the A9, DSP and M3 subsystems, the L3 and L4 memories are connected to the L3 and L4 interconnects respectively. Next to the memories within the subsystems we have 56kB on-chip memory (OCM) RAM (L3), 4kB save-and-restore (SAR) ROM (L4) and 8kB SAR RAM (L4). Both SAR memories are mainly used for context saving purposes and are therefore outside of the scope of this thesis. The OCM RAM on the other hand can be used for any purpose and therefore we might want to use it to (temporarily) store data. For more information refer to [11].

3.2.7 Off-chip memory

The OMAP4460 contains two interfaces for off-chip memories. In the case of our Pandaboard ES, one of the interfaces, the extended memory interface (EMIF), is already in use by 1GB of SDRAM memory. The EMIF can be accessed via the DMM which is connected to the L3 interconnect. The other interface called the general purpose memory controller (GPMC) is not yet used and can potentially serve as an interface between the OMAP4460 and several types of off-chip memories. For more information refer to [11].

3.2.8 Peripherals

Among others, the OMAP 4460 contains the following peripherals: I2C controllers, multiple UART devices, general purpose I/O (GPIO), timers and multimedia controllers (MMCs). All four different devices are reachable via the L4_PER interconnect. The OMAP4460 contains four multi-master high-speed I2C controllers. The I2C controllers support the standard, fast and high speed transmission modes. One of I2C controllers is used for the purpose of Application Q. The UART from the OMAP4460 is mainly used for debugging. Most of the pre-installed images, on-line available for the OMAP4460 [19], setup the UART as serial communication between the OMAP4460 and a host PC. For more information about the I2C controllers and the UART devices refer to [13]. Both the GPIO and timers are used for experimental purpose in the case of this thesis, we discuss them later. More information about the GPIO and timers can be found in [14] and [15] respectively. Last but not least we have the MMCs. The MMCs are used as interface between the subsystems and multimedia devices like an SD card. Since the OMAP4460 allows booting from an SD card, one simply downloads an image to the SD card, plugs it into the SD card slot from the Pandaboard ES, provide power to the Pandaboard and the OMAP4460 automatically starts booting. For more information about the MMCs refer to [16].

3.3 Deployment solution

After exploration of the Pandaboard ES we decided to reserve the entire M3 subsystem for deployment of Application Q and to allow deployment of other future applications on the other subsystems. In the
subsequent subsections we justify our decision, we explain how the deployment is achieved and what the remaining issues are related to our approach.

3.3.1 Justification

In the first place we wondered which of the subsystems from the OMAP-4460 were feasibly to deploy Application Q on. Since the DSP did not seem to support 32-bit floating point operations from scratch but fixed-point operations instead, it did not seem logical to use the DSP subsystem for deployment of Application Q. Another reason for not deploying Application Q on the DSP subsystem is the lack of experience with (programming) DSPs. Both the A9 and M3 subsystems on the other hand do support 32-bit single precision floating point operations. Both the A9 and the M3 subsystem meet the processing requirements; both subsystems contain two RISC processors of which the clock frequencies are high enough and the instruction sets offer 32-bit single precision floating point operations. Furthermore, both subsystems contain large enough memories inside the subsystem to already fulfill the memory requirement. Both subsystems can reach the \(^2C\) controllers. Hence, both subsystems are feasible to deploy Application Q on.

Preferably, Application Q does not share any resources with the future applications and thus we examined the possibility of fully assigning either the A9 or M3 subsystem to Application Q. Both subsystems are feasible to do so, but as a consequence, the capacity of the assigned subsystem that is not used by Application Q (CPU cycles and memory space) will be lost.

We choose to deploy Application Q on the M3 subsystem. One of the reasons for deploying Application Q on the M3 subsystem instead of the A9 subsystem is because the M3 subsystem is a subsystem with small processing power and memory capacity compared to the A9 subsystem. Fully assigning the A9 subsystem would be big waste of both processing and memory capacity. Furthermore, it is important to notice that the A9 subsystem can reach the M3 subsystem but not the other way around, see Table 3.1.

This implies that the A9 subsystem can load firmware for the M3 subsystem and do the initialization while the opposite is not possible. Last but not least, a project found on-line [17] proved that is possible to deploy an application like Application Q on a M3 based processing board.

3.3.2 How it is done

For the installation, we re-used an example project [18] which uses the A9 subsystem to (1) load firmware for the M3 subsystem, (2) initialize the M3 subsystem, (3) provide the M3 subsystem with a pointer to the firmware on (4) wake up one of the two M3 processors. The firmware from the example project contains an OS called FreeRTOS and a very simple example application which is implemented as a periodic task. We reused the source code from that project to write our own firmware including Application Q and we built our own mechanism to load the firmware and to initialize the M3 subsystem.

On the A9 subsystem we run a pre-installed Ubuntu server image [19] which we extended with a module which adds a device node to the Ubuntu device tree. Simply by copying fresh compiled firmware to the device node, the M3 subsystem gets initialized, the firmware gets loaded into the M3 subsystem local memory, the M3 processors is automatically brought out of reset and Application Q starts running.

Because of our approach we can be sure that the M3 processor we use for running Application Q will never be used by the future applications since they will be deployed on a different subsystem and cannot access the M3 processors. In order to make sure that we do not share memory space with the future application we had to make sure that Application Q does not use any piece of memory from outside the M3 subsystem. That requirement was met by only adding the memories from the M3 subsystem to the address space from the M3 subsystem. We did the management of the address space by manually configuration of the MMU translation look-aside buffer (TLB). We put three addresses in the TLB, (1) the address of the 64kB RAM which is inside the subsystem, (2) the address of the \(^2C\) controller configuration registers which can be reached via the L4_CFG interconnect and (3) the address of the \(^2C\) controller itself which is connected to the L4_PER interconnect. Both the firmware for the M3 subsystem and the required execution and data memory space fits in the 64kB RAM inside the subsystem.

3.3.3 Shared NoC

By fully assigning the M3 subsystem to Application Q, we avoid the need of resolving the temporal consequences of having shared processors and memories, in the favor of Application Q. The issues that remain with the just described solution is that the \(^2C\) controllers and the configuration registers are outside the M3 subsystem and hence they can potentially be used by the future applications that will be
deployed on the other subsystems. The same holds for the NoC which we have to use in order to reach the \textit{I}^2\textit{C} controllers and configuration registers. The configuration registers have to be accessed only once before Application Q starts executing and therefore we do not care about them. Since the usage of one of the \textit{I}^2\textit{C} controllers is very specific it is valid to assume that the \textit{I}^2\textit{C} controllers will not be used by the future applications. The NoC on the other hand must be (exhaustively) used by the future applications and thus we have to study the behavior NoC in order to be able to say something about the effect on the temporal requirements of Application Q of sharing the NoC with the future applications.
Chapter 4

Platform model and response time estimations

Within this chapter we introduce a platform model which describes the temporal behavior of the platform, in particular the NoC. Next, we use the model to derive an estimation on the worst- and best-case event response time of deployed Application Q to an environmental event. In Subsection 3.1.2 solely a requirement related to the response time of a job of Application Q, $R_{Q,k}$, is provided. However, in the next chapter we describe experiments that were performed on the outside of the platform. Measurements by using a scope can not be performed within a processor for example. I.e. we need to use some off-chip pins to connect our measurement device to. Therefor, within this chapter, we also derive an estimation on the response time to an environmental event. An environmental event can be generated on the outside of the platform and then one can monitor for a response.

In Subsection 4.1 the platform model is briefly explained. In Sections 4.2 and 4.4 we express the temporal behavior of the deployed application in terms of the platform characteristics. In Sections 4.3 and 4.5 we derive event response time estimations. Finally, in Section 4.6 we summarize what we have learned from modeling the platform and deriving response time estimations.

4.1 Platform model

In Figure 4.1 an architectural overview is graphically represented. Note that components that are not used by Application Q are omitted in the overview.

In the figure one can distinguish four colors: light blue, dark blue, green and red. Light blue represents either a computational subsystem (e.g. the M3 subsystem), a piece of memory or I/O (e.g. an I2C controller). Green represents a network component. A network component can either be the L3 subsystem or the network interface of the processor. Red represents an on-chip memory component.

Figure 4.1: Architectural overview
interconnect, the L4_PER interconnect or the $I^2C$ bus. Dark blue represents either a network interface or a bridge. Last but not least, red represents an environmental event or response.

Within the next subsections we derive temporal assumptions for the individual hardware components from the platform. The assumptions are based on information from Sections 3.2 and 3.3 and [28, 29, 24, 25, 20, 21, 22, 8, 10, 14, 6, 7, 13].

4.1.1 No shared resources

To allow estimation of the event response time of our application deployed on the platform, no sharing of resources is assumed. I.e. when deriving the event response times we do not take any interference into account. Within our equations we will use a delay time $t_D$ equal to zero. In practice however, in our particular situation, due to our deployment strategy, the NoC is potentially shared by multiple applications and hence interference might actually occur when accessing the NoC [30].

4.1.2 Gyroscope and accelerometer

Each $10\text{ms}$ ($T_{\text{Sensors}} = 10\text{ms}$), new sensor data (x, y and z direction) is available and stored into FIFO buffers. From both sensors we have to read $3 \times 16$ bits (16 bits per direction). Both the gyroscope and the accelerometer have three FIFOs, each having 32 entries, which can be accessed via $I^2C$. Since the FIFO buffers are used we do not have to read the sensor data from these two sensors at a Nyquist frequency. Reading the sensors is done at a rate equal to the rate at which the sensors produce data, $100\text{Hz}$. If the FIFO buffer is empty, the last value will be returned.

4.1.3 Magnetometer

Each $100\text{ms}$, new sensor data is available (x, y and z direction). In total $3 \times 16$ bits have to be read. The magnetometer does not have internal FIFOs, but reading will be performed at a frequency of $100\text{Hz}$ ($T_{\text{Sensors}} = 10\text{ms}$), which is 5 times the Nyquist frequency $\frac{100\text{ms}}{2} \Rightarrow 20\text{Hz}$, such that none of the magnetometer samples are missed by the controlling system.

4.1.4 Motors

Each $2.5\text{ms}$ ($T_{\text{Motors}} = 2.5\text{ms}$) the device checks its internal registers for the required motor thrust settings and makes sure the four motors spin at the right speed. In total $4 \times 16$ bits have to be written to the device. Writing to the motors is done at a rate of $100\text{Hz}$. When the device checks it internal registers and no new data is available, the device simply does not change the motor speed.

4.1.5 $I^2C$ bus

The $I^2C$ bus is configured to run in $I^2C$ fast mode and therefore it can transmit $400\text{kbps}$ per second. Based on Equation (3.1) and (3.2) and the $400\text{kbps} I^2C$ transmission rate, we derive the required $I^2C$ transmission time for reading and writing 16 bits of data. An $I^2C$ write message is 38 bits long and therefore it takes $95\mu s$, see Equation (4.1), to transmit the message over the bus. Note that $C$ represents the time that is required to fulfill a certain task, in this case the transmission of an $I^2C$ message. An $I^2C$ read message is 48 bits long and therefore it takes $120\mu s$, see Equation (4.2), to transmit the message over the bus.

$$C_{I^2C\text{-write}} = \frac{1}{f_{I^2C}} = \frac{1}{400 \cdot 10^3} \cdot 38 = 95\mu s \quad (4.1)$$

$$C_{I^2C\text{-read}} = \frac{1}{f_{I^2C}} = \frac{1}{400 \cdot 10^3} \cdot 48 = 120\mu s \quad (4.2)$$

4.1.6 NoC

The NoC connects the M3 subsystem with, among others, peripherals like $I^2C$ controllers and GPIO. The NoC consists of network interfaces, two types of interconnects and a bridge. It is assumed that the NoC components and the M3 subsystem operate in sync. This assumption is based on the fact that the components use a clock frequency which decent from one single physical clock, the M3 subsystem clock.
4.1.6.1 L4_PER interconnect

The L4_PER interconnect is a piece of hardware that, due to the lack of public available information, can be seen as a black box. I.e. something goes in and eventually something comes out at the right destination. For that reason, the L4_PER interconnect is represented as a cloud in Figure 4.1. Within the interconnect, possibly some packet switching is performed and therefore we assume that it takes some time before a packet reaches its destination. However, the amount of time is unknown. What we do know is that the L4_PER interconnect uses a 100 MHz clock signal and at least 32 bits can be sent in parallel. Therefore we assume that transferring a packet with 32 bits data and 32 bits addressing overhead over the L4_PER interconnect takes 20 ns, see Equation (4.3).

\[
C_{L4} = \frac{1}{f_{L4}} \cdot 2 = \frac{1s}{100 \cdot 10^6} \cdot 2 = 20\text{ns} \tag{4.3}
\]

Note we assumed an addressing overhead of 32 bits per packet for address storage etc. The assumption regarding the parallel transportation and the overhead on top of the data is based on online available information [29] presented in Figure 4.2.

![L4_PER packetization](image)

Figure 4.2: L4_PER packetization

4.1.6.2 L3 interconnect

Like the L4_PER interconnect, the L3 interconnect is a component which lacks the availability of public information regarding its temporal behavior. The L3 interconnect is therefore also represented as a cloud. What we do know is that the L3 interconnect uses a 200 MHz clock signal which descends from the M3 subsystem and at least 32 bits can be sent in parallel. Therefore we assume that transferring a packet with 32 bits data and some additional overhead over the L3 interconnect takes 10 ns, see Equation (4.4).

\[
C_{L3} = \frac{1}{f_{L3}} \cdot 2 = \frac{1s}{200 \cdot 10^6} \cdot 2 = 10\text{ns} \tag{4.4}
\]

4.1.6.3 NIs and the bridge

The network interfaces (NIs) unpack the packets delivered by the NoC components (L3 and L4_PER interconnect) and provide the data from the received packet to the connected device. Vice versa, the data from the connected component is packed with an address attached to it and put on the NoC for delivery.

The bridge is a special NI. The bridge transfers the data packets from clock domain one (the L3 interconnect) to clock domain two (the L4_PER interconnect) and the other way around.

Since there is no information available regarding the time the NIs require to handle the data, it is assumed that the NIs use the same clock as the interconnect they are connected to. I.e. the NI connecting the M3 subsystem and the L3 interconnect uses a clock running at 200 MHz. The NI connecting GPIO to the L4 interconnect, the NI connecting the \(\mu\)C controller to the L4 interconnect and the bridge connecting the L3 interconnect and the L4 interconnect require 20 ns to handle the data, see Equations (4.5), (4.6) and (4.7) respectively. The NI connecting the M3 subsystem and the L3 interconnect needs 10 ns, see Equation (4.8), to handle the data.

\[
C_{NI-GPIO} = \frac{1}{f_{L4}} \cdot 2 = \frac{1s}{100 \cdot 10^6} \cdot 2 = 0.02\mu\text{s} \tag{4.5}
\]

\[
C_{NI-\mu\text{C}} = \frac{1}{f_{L4}} \cdot 2 = \frac{1s}{200 \cdot 10^6} \cdot 2 = 0.02\mu\text{s} \tag{4.6}
\]

\[
C_{Bridge} = \frac{1}{f_{L4}} \cdot 2 = \frac{1s}{100 \cdot 10^6} \cdot 2 = 0.02\mu\text{s} \tag{4.7}
\]
\[ C_{NI-M3} = \frac{1}{f_{L3}} \cdot 2 = \frac{1s}{200 \cdot 10^6} \cdot 2 = 0.01 \mu s \] (4.8)

### 4.1.7 GPIO

GPIO is used for the purpose of Application G which we discuss in section 4.4. The GPIO component uses an interface clock running at 100MHz but it is unknown how much time the GPIO requires to actually set or reset one of the pins. It is assumed that the GPIO module needs one clock tick at 100MHz to actually set or reset one of the pins, hence it takes 10ns, see Equation (4.9), to set or reset a pin.

\[ C_{GPIO} = \frac{1}{f_{L4}} = \frac{1s}{100 \cdot 10^6} = 0.01 \mu s \] (4.9)

### 4.1.8 M3 subsystem

The M3 subsystem is already discussed in subsection 3.2.5.

### 4.2 Deployment Application Q

In this section we express the temporal behavior of Application Q in terms of the modeled platform characteristics. In Figure 4.3 a reduced architectural overview of the platform is provided again. The red lines in the figure represent the logical paths of packets traveling from the M3 Subsystem to the sensors and motors and vice versa.

![Logical path from the M3 subsystem to the sensors/motors](image)

Figure 4.3: Logical path from the M3 subsystem to the sensors/motors

From this point we model Application Q as a task \( \tau_Q \), which is periodic released every 10ms. \( \tau_Q \) interacts with three different asynchronous tasks labeled \( \tau_{Sensors}, \tau_{Motors}, \) and \( \tau_{Environment}. \) All interactions are initiated by \( \tau_Q. \) Both task \( \tau_Q \) and task \( \tau_{Sensors} \) is periodic released at an interval of \( T_Q = T_{Sensors} = 10ms. \) \( \tau_{Motors} \) is released every \( T_{Motors} = 2.5ms. \) \( \tau_{Environment} \) is observed as a sporadic task, since environmental events can occur any time. The execution time of \( \tau_{Sensors}, \tau_{Motors} \) and \( \tau_{Environment} \) is assumed to be negligible \( C_{Sensors} = C_{Motors} = C_{Environment} = 0, \) data is directly available/consumed after the release of the task. The execution time \( C_Q \) is calculated in the next subsections by calculating the execution times of the sub-jobs from \( \tau_Q \) labeled \( \tau_Q-reading, \tau_Q-processing \) and \( \tau_Q-writing. \)

#### 4.2.1 Execution time \( \tau_Q-reading \) and \( \tau_Q-writing \)

Each time \( \tau_Q \) is released, 9 values have to be read from sensors and 4 values have to be written to the motors. Reading and writing from/to sensors/motors can only be performed by the \( I^2C \) controller in our case and hence interaction over the NoC has to take place between the M3 subsystem, on which
Application Q is deployed, and the \( I^2C \) controller. Each of the NoC components discussed in Subsection 4.1.6 consume time to transfer packets. The NoC components together consume \( C_{\text{NoC}} = 0.08 \mu s \) to transfer a packet from the M3 subsystem to the \( I^2C \) controller and vice versa, see Equation (4.10). Mention the we assumed that none of the NoC resources is shared among applications and we took a delay time of \( t_D = 0 \) into account. An overview of the NoC components and the corresponding required times to transfer a packet are graphically represented in Figure 4.4.

\[
C_{\text{NoC}} = C_{\text{NI-M3}} + C_{\text{L3}} + C_{\text{Bridge}} + C_{\text{L4}} + C_{\text{NI-I}2C} + t_D \tag{4.10}
\]

\[
C_{\text{NoC}} = 0.01 \mu s + 0.01 \mu s + 0.02 \mu s + 0.02 \mu s + 0.02 \mu s + 0 = 0.08 \mu s
\]

![Figure 4.4: Overview of NoC components and corresponding packet transfer time](image)

Before the \( I^2C \) controller actually reads a value from a certain sensor, a few things have to be arranged. E.g. the sensor \( I^2C \) address and the amount of bits we want to receive has to be send to the \( I^2C \) controller. Afterwards the \( I^2C \) controller starts reading the sensor value. Once the \( I^2C \) controller has finished reading, another few steps have to be taken to actually retrieve the result from the \( I^2C \) controller. For our convenience it is assumed that configuring the \( I^2C \) bus controller and reading the result requires ten \( I^2C \) controller register writes and five \( I^2C \) controller register reads in total. This assumption is based on the \( I^2C \) programming guide found in [13]. A single register write includes a single NoC transaction while a single register read includes two NoC transactions, since a result has to be returned. In total 20 NoC transactions with the \( I^2C \) controller have to be performed to retrieve a single sensor value. The required time for 20 NoC transitions with the \( I^2C \) controller, labeled \( C_{\text{I}2\text{C-controller}} \), is equal to 1.6\( \mu s \), see Equation (4.11).

\[
C_{\text{I}2\text{C-controller}} = \# \text{NoC-transactions} \cdot C_{\text{NoC}} = 20 \cdot 0.08 \mu s = 1.6 \mu s \tag{4.11}
\]

In Subsection 4.1.5 we calculated the time the \( I^2C \) controller needs to read/write a single value from/to a sensor/motor, see Equations (4.2) and (4.1) respectively. The last thing we have to take into account to calculate \( C_{\text{Q-reading}} \) and \( C_{\text{Q-writing}} \) is the time the M3 subsystem needs to setup the NoC transactions. From assembly analysis, we know that putting a single register read/write request on the NoC takes only one instruction cycle and therefore putting all read and write request on the NoC takes 0.05\( \mu s \), see Equation (4.12).

\[
C_{\text{CPU}} = 20 \cdot \frac{1}{f_{\text{CPU}}} = 20 \cdot \frac{1}{200 \cdot 10^6} = 0.05 \mu s \tag{4.12}
\]

\( C_{\text{Q-reading}} \) can now be calculated by taking the sum over \( C_{\text{I}2\text{C-controller}}, C_{\text{I}2\text{C-read}} \) and \( C_{\text{CPU}} \) and multiplying the result by nine, since we have to read nine sensor values. We do the same for calculating \( C_{\text{Q-writing}} \), we take the sum over \( C_{\text{I}2\text{C-controller}}, C_{\text{I}2\text{C-write}}, \) and \( C_{\text{CPU}} \) and multiply the result by four, since we have to write four motor values. The resulting calculations are presented in Equations (4.13) and (4.14).

\[
C_{\text{Q-reading}} = 9(C_{\text{I}2\text{C-controller}} + C_{\text{I}2\text{C-read}} + C_{\text{CPU}}) = 9(1.6 \mu s + 122 \mu s + 0.05 \mu s) \approx 1100 \mu s \tag{4.13}
\]

\[
C_{\text{Q-writing}} = 4(C_{\text{I}2\text{C-controller}} + C_{\text{I}2\text{C-write}} + C_{\text{CPU}}) = 4(1.6 \mu s + 97 \mu s + 0.05 \mu s) \approx 400 \mu s \tag{4.14}
\]
4.2.2 Execution time $\tau_{Q-processing}$

In [1], it was derived that one instance $\tau_{Q,k}$ requires the M3 subsystem to perform $344$ scalar arithmetic operations. For our convenience, to avoid the need of analyzing assembly, we assume that $1 \cdot 10^3$ instruction cycles per instance $\tau_{Q,k}$ have to be performed. Since the M3 subsystem runs at a clock frequency of $400\,MHz$, performing all $1 \cdot 10^3$ instructions cycles takes about $2.5\,\mu s$, see Equation (4.15).

$$C_{Q-processing} = \frac{\#\text{instructions}}{f_{CPU}} = \frac{1 \cdot 10^3}{400 \cdot 10^6} = 2.5\,\mu s$$ (4.15)

4.2.3 Execution time $\tau_Q$

Finally, to calculate the execution time of $\tau_Q$, we take the sum over $C_{Q-reading}$, $C_{Q-processing}$ and $C_{Q-writing}$, the result is presented in Equation (4.16).

$$C_Q = C_{Q-reading} + C_{Q-processing} + C_{Q-writing} = 1100\,\mu s + 2.5\,\mu s + 400\,\mu s \approx 1.5\,ms$$ (4.16)

From this figure, we can already conclude that, assuming our platform model is correct and none of the NoC components are shared and $\tau_Q$ is processed directly after the release of the job, that the requirement from Subsection 3.1.2 is met. In this case, $R_{Q,k}$ is equal to $C_Q$, which is smaller than $D_Q = 10ms$. Furthermore, we can conclude that the processing time, $C_{Q-processing}$ are negligible compared to $C_{Q-reading}$ and $C_{Q-writing}$, because the latter are 2 orders of magnitude larger than the former. The main reason that $C_{Q-reading}$ and $C_{Q-writing}$ are 2 orders of magnitude larger than $C_{Q-processing}$ is because of the time the $I^2C$ bus requires to transport data. I.e. the transmission rate of the $I^2C$ bus is the bottleneck.

4.3 Event response time Application Q

At this point we know how much time Application Q requires to read the sensors, process the data and then write to the motors. Next, we derive estimations on the best- and worst-case response time of Application Q to an environmental event. In Figure 4.5 the architectural overview of the platform is graphically represented again, this time an environmental event and a response is presented in the figure as well.

The four tasks labeled $\tau_Q$, $\tau_{Sensors}$, $\tau_{Motors}$ and $\tau_{Environment}$ are asynchronous tasks and the arrival time of each individual task is crucial with respect to the event response time of Application Q. In Figure 4.6 a UML sequence diagram is presented which includes the four asynchronous tasks. $L_1$, $L_2$ and $L_3$ represent the inter-arrival time of two consecutive tasks. In the next two subsections we discuss a situation in which the inter-arrival times are minimal and maximal respectively.
4.3.1 Best-case event response time

In the best-case situation, the four tasks are released directly after each other in the sequence \( \tau_{\text{Environment}}, \tau_{\text{Sensors}}, \tau_{\text{Q}} \) and \( \tau_{\text{Motors}} \). The best-case situation called the optimal instant is graphically represented in Figure 4.7.

(1) An environmental event arrives, (2) directly after the arrival of the event, the sensors take a new sample, (3) directly after the arrival of a new sample, Application Q reads the sensor data, does the necessary processing and sends data to the motors and (4) the motors read the motor data and the actuation takes place. Hence, the best-case response time Application Q to an environmental event is equal to \( BR_Q = 1.5\, ms \), see Equation (4.17).

\[
BR_Q = C_{\text{Environment}} + C_{\text{Sensors}} + C_Q + C_{\text{Motors}} = 0 + 0 + 1.5\, ms + 0 = 1.5\, ms
\]  

(4.17)

4.3.2 Worst-case event response time

In the worst-case situation, task B, which requires data from task A, is released first and directly afterwards task A is released. In other words, task B just misses the data from task A and the data from task A will be processed by B the next release of task B. The worst-case situation, called the critical instant, is graphically represented in Figure 4.8.
Figure 4.7: time line: best-case response on environmental event (an optimal instant)

(1) The sensors take a new sample, but no environmental event has yet occurred, (2) directly after the sampling from the sensors, the environmental event arrives, (3) just before the sensors take a new sample, task $\tau_Q$ is released, but the sensor data is not yet available, (4) the sensors take a new sample because the next instant of $\tau_{Sensors}$ is released and the event is captured by the sensors, (5) the next instant of $\tau_Q$ is released and the event is now captured by Application Q, (6) just before $\tau_Q$ wants to write data to the motors, task $\tau_{Motors}$ is released such that it just misses the data from $\tau_Q$ and (7) the next instant of $\tau_{Motors}$ is released such that it captures the motor data and the motors perform the actuation. Hence, the worst-case response time of Application Q to an environmental event is equal to $WR_Q = 23.5 ms$, see Equation (4.18).

\[ WR_Q = T_{Sensors} + T_Q + C_Q - \text{Reading} + C_Q - \text{Processing} + T_{Motors} \]  
\[ WR_Q = 10ms + 10ms + 1100\mu s + 2.5\mu s + 2.5ms \approx 23.5ms \]

None of the requirements in this thesis describe what the maximum response time to an environmental event might be. However, if one wants to minimize these environmental event response times, one of the options would be to synchronize the delivery and the demand of data between two tasks which have a data dependency. Another option would be to increase the sample rate of the sensors for example.

### 4.4 Deployment Application G

Because of our deployment strategy of Application Q, the only resources that Application Q will share in the future with potential other applications, are the components from the NoC. However, as can be observed from the calculations in Section 4.2 the time the NoC requires to transfer packets contributes only very little to the execution time of $\tau_Q$. The transfer rate of the $I^2C$ bus is the bottleneck. The transfer rate of the $I^2C$ bus is an order of magnitude 3 smaller than the transfer rate of the NoC. I.e. the time the $I^2C$ controller needs to read the sensors and to write to the motors is the major contribution to the execution time of $\tau_Q$. Therefore, we replace Application Q by a much simpler application called...
Application G, which includes only very little functionality and relatively more NoC transactions. This allows a considerably higher contribution of the NoC transfer time to the execution time. I.e. the temporal effects of variations in the NoC transfer time will be better observable.

Application G implements the following functionality: (1) we read the status of a GPIO input pin and (2) we write the exact same status to a GPIO output pin. The status of a GPIO pin can either be a logic '1' or a logic '0'. The functionality is implemented in an infinite while loop and will therefore run forever. The reason for using a GPIO module instead of an I²C bus is that the GPIO module is less complex. What happens on our platform is summarized below:

1. From software running on the M3 subsystem we initiate a GPIO input pin read request to the NoC
2. A packet containing the read request is transmitted over the NoC from the M3 subsystem to the GPIO module. The logical path is graphically represented in red in Figure 4.9.
3. The GPIO module actually reads the status of the GPIO input pin and puts the result back on the NoC.
4. A packet containing the result is transmitted back over the NoC from the GPIO module to the M3 subsystem.
5. The M3 subsystem checks the status of the GPIO input pin and then initiates a GPIO output pin write request to the NoC depending on that status.
6. A packet containing the write request is transmitted over the NoC from the M3 subsystem to the GPIO module.
7. The GPIO module actually writes the GPIO output pin depending on the status stored in the packet.
8. We go back to step 1.

The steps and the logic path under consideration are graphically represented in Figure 4.9.

Unlike the situation described in Section 4.2, there are only two asynchronous tasks in this situation; task $\tau_G$ and task $\tau_{Environment}$. In this particular situation a transition from a logic '0' to a logic '1' and the other way around is considered an environmental event on which Application G has to respond. Task $\tau_G$ is a continuous running task while $\tau_{Environment}$ is considered a sporadic task. $C_{Environment}$ is again equal to zero. In order to calculate $C_G$, we have to take a closer look at the implementation of Application G. In
Table 4.1, the Assembly instructions related to Application G are presented. Furthermore, the number of clock cycles the M3 processor requires to execute the instructions are given and it is indicated whether or not the NoC is involved if when executing a certain instruction.

<table>
<thead>
<tr>
<th>#</th>
<th>M3 instruction</th>
<th>M3 cycle count</th>
<th>NoC involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ldr r0, GPIO-input</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>2</td>
<td>and r1, r0, #32</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>lsl r0, r1, #1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>str r0, GPIO-output</td>
<td>2</td>
<td>√</td>
</tr>
<tr>
<td>5</td>
<td>b (instruction 1)</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: M3 instructions Application G

As soon as the NoC gets involved, see instructions 1 and 4, we have to take $C_{NoC}$ into account again when we want to derive an estimation on the execution time, see Equation (4.10). It is assumed that the GPIO needs some time to actually read/write the value and therefore we have to add $C_{GPIO}$ from Equation (4.9). Last but not least the M3 requires some time, see the M3 cycle count in the table, to actually put the read/write request on the NoC. Hence instruction 1 takes 0.18 µs, see Equation (4.19). Instruction 4 takes 0.1 µs, see Equation (4.20), to complete. For instructions 2, 3 and 5, the NoC is not involved and thus the required time is equal to the cycle count divided by the CPU clock frequency, see Equations (4.21) and (4.22). The total execution time of $\tau_G$ is equal to 0.3 µs, see Equation (4.23).

\[
C_{instruction-1} = \frac{\text{cycle count}}{f_{CPU}} + C_{NoC} + C_{GPIO} + C_{NoC}
\]

\[
C_{instruction-1} = \frac{2}{400 \cdot 10^6} + 0.08 \mu s + 0.01 \mu s + 0.08 \mu s \approx 0.18 \mu s
\]  

\[
C_{instruction-4} = \frac{\text{cycle count}}{f_{CPU}} + C_{NoC} + C_{GPIO} = \frac{2}{400 \cdot 10^6} + 0.08 \mu s + 0.01 \mu s \approx 0.1 \mu s
\]  

\[
C_{instruction-2} = C_{instruction-3} = \frac{\text{cycle count}}{f_{CPU}} = \frac{1}{400 \cdot 10^6} \approx 0.003 \mu s
\]

\[
C_{instruction-5} = \frac{\text{cycle count}}{f_{CPU}} = \frac{4}{400 \cdot 10^6} = 0.01 \mu s
\]

\[
C_G = \sum_{i=1}^{5} C_{instruction-i} = 0.18 \mu s + 0.1 \mu s + 0.003 \mu s + 0.003 \mu s + 0.01 \mu s \approx 0.3 \mu s
\]
4.5 Event response time Application G

In Figure 4.10 the architectural overview of the platform is graphically represented again. Note that this time the environmental event and the response are related to the GPIO module instead of the actuators and motors which are connected to the I2C bus.

In Figure 4.11 the UML sequence diagram is presented related to Application G.

The response time of Application G to an environmental event depends on when the environmental event arrives. We consider both optimal and critical arrival times of the environmental event in the next subsections.

4.5.1 Best-case event response time

In the UML sequence diagram presented in Figure 4.12 the red arrow represents the optimal arrival time of the environmental event, just before the GPIO module samples the input. I.e. when calculating the best-case event response time only $C_{GPIO}$ and $C_{NoC}$ from $C_{instruction-1}$ has to be taken into account and instructions 2, 3 and 4. The estimated best-case event response time is therefore equal to $0.2\mu s$, see Equation (4.24).

$$BR_G = C_{GPIO} + C_{NoC} + \sum_{i=2}^{4} C_{instruction-i}$$

$$BR_G = 0.01\mu s + 0.08\mu s + 0.003\mu s + 0.003\mu s + 0.1\mu s \approx 0.2\mu s$$
4.5.2 Worst-case event response time

Worst-case, the environmental event arrives just after the GPIO samples the input pin, see the UML sequence diagram presented in Figure 4.13. Due to the fact that during the first iteration we just miss the environmental event, the environmental event is captured the next iteration.

The estimated worst-case event response time is equal to 0.5μs, see Equations (4.25), (4.26) and (4.27).

\[
WR_G = C_{NoC} + C_G + \sum_{j=2}^{4} C_{instruction-j} \quad (4.25)
\]

\[
\sum_{j=2}^{4} C_{instruction-j} = 0.003 \mu s + 0.003 \mu s + 0.1 \mu s \approx 0.1 \mu s \quad (4.26)
\]

\[
WR_G = 0.08 \mu s + 0.3 \mu s + 0.1 \mu s \approx 0.5 \mu s \quad (4.27)
\]

4.6 Summary and findings

Based on available information of the platform, we have created a platform model. Next we have expressed the temporal behavior of Application Q in terms of the platform model characteristics. Then we concluded, given a number of assumptions, that the requirement given in Subsection 3.1.2 can be met. In addition, we noticed that most of the execution time of Application Q is due to the speed of the \( I^2C \) bus. Afterwards, we determined the response time of application Q to an event which takes place on the
outside of the platform. We did that because during the experiments described in the next chapter, we observe what happens on the outside of the platform. Then we replaced Application Q by a much less complex application, Application G. We did that because the response time of Application Q is mainly determined by the $I^2C$ bus, while we want evaluate the behavior of the NoC. Application G comprises so little functionality and is not dependent on the relatively slow $I^2C$ bus, allowing the transmission times on the NoC to play a significant role in the execution time. This allows us to analyze variations in the NoC transmission times better. We use application G whatsoever during our experiments instead of Application Q. Finally, we determined the response time of application G on a environmental event which occurs on the outside of the platform. We use those response times as comparison material during our experiments.
Chapter 5

NoC latency experiments

In Chapter 4, we derived an estimation on the best- and worst-case event response time of Application G to an environmental event. It was assumed that the NoC was not shared and thus the only traffic on the NoC was the traffic from Application G. We did not take any interference into account. Within this chapter, we present the results of experiments that were performed to validate the results from Section 4.5. Furthermore, we tried to introduce interference to delay the packets traveling between the M3 subsystem and the GPIO module. The results from our experiments are also representative for Application Q, which uses I2C instead of GPIO, since the fact that both peripherals are memory mapped and the exact same resources from the NoC are involved. The only difference is the functionality of the peripheral.

In the next subsection, the setup of the experiment is described, and the concept of the so-called round-trip-time (RTT) experiment is briefly explained. In Section 5.2, we introduce the results from an experiment that was performed to validate the results from Section 4.5. The same results are also used during the remainder of the experiments for comparison. In Section 5.3, the result of scaling the M3 subsystem, L3 interconnect, and L4_PER interconnect frequencies with respect to the RTT are presented. We performed those experiments to validate the relation between the clock frequencies and the RTT. In Section 5.4, we present the results of experiments we performed to determine the effect of additional traffic on the L3, L4_PER, and L4_CFG interconnects with respect to the RTT. We try to introduce interference on the NoC. In Section 5.5, we discuss the results of a use case experiment.

5.1 Setup and RTT experiment concept

The RTT is defined as the time it takes for a signal to be sent plus the amount of time it takes for an acknowledgment of that signal to be received. In our case, with Application G, this means that we have to put a signal, an artificial environmental event, on the GPIO module and then measure the time until the exact same signal comes out of the GPIO module again. The concept is graphically represented in Figure 5.1. In the figure, $t_0$ represents the time at which the signal (rising edge transition) is put on the GPIO module. $t_1$ represents the time at which the signal comes out of the GPIO module again. The RTT is calculated by subtracting $t_0$ from $t_1$.

![Figure 5.1: RTT $t_1 - t_0$ in the case of rising edge detection](image)

The RTT experiment concept was implemented using a $1k\, Hz$ square wave generator and a logic analyzer. The wave generator was used to generate the artificial environmental events. The logic analyzer was used to measure the time between the occurrence of an input transition and the occurrence of an output transition. The logic analyzer collects samples at frequency of $100\, MHz$. The experimental
setup is presented in Figure 5.2. More information about the hardware and software used during the experiments can be found in Appendix B and Appendix C.

![Experimental setup diagram]

**Figure 5.2: Experimental setup**

### 5.2 Experiment for model validation

Within this section we introduce the results from an experiment that was performed to validate the results from Section 4.5. We use the results from this experiment also during the remainder of the experiments for comparison. Hence, this experiment can also be seen as control experiment. During the experiment only Application G was running on the platform and we expected a best- and worst-case RTT of 0.2µs and 0.5µs respectively. The result of the experiment is presented in Figure 5.3.

![Result RTT control experiment graph]

**Figure 5.3: Result RTT control experiment**

Within the figure, the X axis represents the RTT expressed in micro seconds and the Y axis represents the ratio of occurrences. The graph is based on 10^5 collected RTT values. The experiment was repeated multiple times to check for consistency. In the graph, each red bar represents a subset of values which are bucketed using a bucket size of 0.05µs. The length of the bar represents the ratio of values that are
within that 0.05µs range. E.g. the first bar from the left-hand side represents that about 7.5% of the values is within the range 0.25 – 0.3µs. Results are rounded to 0.05µs due to accuracy of 0.02µs of the logic analyzer taking samples at 100MHz sampling frequency, see Equation (5.1). In the equation, a Nyquist factor of 2 is taken into account.

\[
\text{accuracy} = \frac{1}{f_{\text{logic-analyzer}}} \cdot 2 = \frac{1s}{100 \cdot 10^6} \cdot 2 = 0.02\mu s
\]

Equation (5.1)

In Table 5.1 some statistics are presented.

<table>
<thead>
<tr>
<th></th>
<th>min.</th>
<th>max.</th>
<th>avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25µs</td>
<td>0.75µs</td>
<td>0.50µs</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Result RTT control experiment

From the statistics we derive that the lower bound is equal to 0.25µs and the upper bound is equal to 0.80µs. The result is not entirely consistent with our expectation, both the the expected best and worst-case RTT is smaller than the observed lower and upper bound respectively. Below we describe two plausible causes:

- We assumed the M3 subsystem, L3 interconnect and L4_PER interconnect clock signals are synchronized. If this is not the case then for each hop in the NoC between two different frequency domains, worst-case a delay of one clock period from the slowest clock domain is introduced. In Equation 5.2 we calculate what the delay would be in that case. For a single round trip, we have to use the NoC three times, two times for reading the GPIO input pin and one time for writing the GPIO output pin. Each time we use the NoC we have three hops, M3 → L3 → L4 → NI_GPIO. Hence, in the worst case situation, we have to take an additional 0.08µs delay into account, see Equation 5.2. Taking the additional delay into account for our platform model, the worst-case RTT would be 0.58µs, which is already close to 0.6µs.

- Furthermore, any routing algorithm might be implemented at the NoC level. Depending on the arrival of a packet some time might expire before the packet is routed. However, crucial information about the NoC is missing to come up with reasonable figures at this point. Extensive future work might contribute to a more accurate model of the platform. However, from this point we will use the measured 0.8µs worst-case RTT to draw our conclusions.

\[
3 \left( \frac{1}{f_{L3}} + \frac{1}{f_{L4}} + \frac{1}{f_{NI_{GPIO}}} \right) = \frac{3s}{200 \cdot 10^6} + \frac{3s}{100 \cdot 10^6} + \frac{3s}{100 \cdot 10^6} \approx 0.08\mu s
\]

Equation 5.2

5.3 Frequency scaling experiment

The platform allows frequency scaling of the M3 subsystem, L3 interconnect and L4_PER interconnect clocks. During the experiment which is described in this section we changed the clock frequencies to examine what the effect is on the RTT. This experiments allows us to validate the relation between the clock frequencies and the RTT. The clocks from the L3 and L4_PER interconnect descent from the M3 subsystem clock and hence scaling the M3 subsystem clock frequency also scales the L3 and L4_PER interconnect clock frequencies. In Chapter 4 we have seen that \(C_{CPU}, C_{NOC}\) and \(C_{GPIO}\), see Equations (4.12), (4.10) and (4.9), depend on the clock frequencies and therefore we expect that when we halve the clock frequencies the RTT will double. The result is presented in Table 5.2. Note that in the table, the first row represents the same results as obtained in the previous experiment. For each of the three experiments, 10\(^5\) RTT values were collected.

<table>
<thead>
<tr>
<th></th>
<th>(f_{CPU})</th>
<th>(f_{L3})</th>
<th>(f_{L4})</th>
<th>min.</th>
<th>max.</th>
<th>avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>400MHz</td>
<td>200MHz</td>
<td>100MHz</td>
<td>0.25µs</td>
<td>0.75µs</td>
<td>0.50µs</td>
<td></td>
</tr>
<tr>
<td>200MHz</td>
<td>100MHz</td>
<td>50MHz</td>
<td>0.55µs</td>
<td>1.55µs</td>
<td>1.05µs</td>
<td></td>
</tr>
<tr>
<td>100MHz</td>
<td>50MHz</td>
<td>25MHz</td>
<td>1.05µs</td>
<td>2.95µs</td>
<td>2.00µs</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Frequency scaling results

From the table it can be observed that the minimum, average and maximum observed RTT indeed doubles when the clock frequencies are halved. Hence the RTT scales linear with the clock frequency.
5.4 Simulated load experiment

Up to this point no delay time $t_D$ because of interference at the NoC was taken into account. In this section, experiments are performed where the NoC is shared between multiple applications running on different subsystems. We have two applications running: (1) we have Application G running on the M3 subsystem and (2) we have another application, Application L, running on the A9 subsystem. The A9 subsystem generates artificial load on the NoC to a certain destination by continuously polling some register of the destination. Application L is implemented bare-metal on the A9 subsystem and exists only of a few assembly instructions to ensure maximum load is generated on the NoC. We performed the experiment six times, each time Application L simulated load on the NoC to a different target node. In Figure 5.4, an overview of the experiment is graphically represented. The red line represents the logical path from the M3 subsystem to the GPIO module. The orange lines represent the six different logical paths from the A9 subsystem to a target node. The six different target nodes are labeled L3_DMM_CFG, L3_GRPC_CFG, L4_PER_UART1, L4_PER_GPIO2, L4_CFG_SPINLOCK and OFFCHIP_SDRAM. The prefixes L3_, L4_PER_ and L4_CFG_ represent that the target node is connected to the L3, L4_PER or L4_CFG interconnect respectively. The prefix is followed by the name of the target node.

![Diagram of simulated load experiment](image)

Figure 5.4: Simulated load experiment overview

The L3 connectivity matrix presented in Table 3.1 and Figure 3.3 from Subsection 3.2.2.1 suggest that there exist separate paths from a NoC initiator node to a NoC target node. Furthermore, in [29] some information can be found on the architecture of the L3 interconnect transaction layer. Based on this information, we expect that the architecture of our L3 interconnect must look like the architecture presented in Figure 5.5.

Based on the expected L3 interconnect architecture, we expect that only traffic from the A9 subsystem to the GPIO module will affect the RTT. Both the A9 and the M3 subsystem try to access the same hardware component and hence they might have to wait for each other to complete the transaction. The results of the experiment are presented in Table 5.3. For each of the six experiments, we collected $10^5$ RTT values.

The results of the experiment show that the simulated load on the NoC does not affect the observed RTT. The observed lower and upper bound is equal to $0.25\mu s$ and $0.80\mu s$ respectively. The small deviations in the table are the result of the rounding that was applied.
Our presumption that the traffic from the A9 subsystem to the GPIO module would affect the RTT was wrong. A plausible explanation for this phenomenon is that the RTT is not affected because of some arbitration mechanism. E.g. it can be the case that access to the GPIO module is scheduled by a time division multiplexing scheduler. Each initiator node would have a certain time slot to access the GPIO module. Another plausible explanation can be that traffic from the M3 subsystem to the GPIO module always has a higher priority than traffic from the A9 subsystem to the GPIO module. A closer future analysis is required to confirm that explanation.

5.5 Use case experiment

The load we simulated on the NoC in the previous section did not influence the measured RTT of application G. However, we might have overlooked something and therefore we performed a final use case experiment. The use case experiment is based on a pre-build image [31] which is executed from the A9 subsystem. The pre-build image includes a Ubuntu distribution which is adapted such that it runs on the OMAP4460. On top of Ubuntu we run an application called XBMC Media Center. We use the media center to playback a full HD 1080p movie. When playing the movie, XBMC uses a lot of resources from the OMAP4460, including the IV AHD subsystem, the on-chip GPU, the display subsystem, the SDRAM, USB, SD card etc. I.e. a lot more resources and a lot more different logical paths via the different interconnects are addressed. We play the movie while performing the RTT experiment again with Application G. Because of the results from the experiment in the previous section we expected that the load on the NoC introduced by XBMC would not affect the RTT of application G. An overview of the experiment is provided in Figure 5.6.

The result of the experiment is presented in Table 5.4. The observed lower and upper bound is equal to 0.25\(\mu s\) and 0.75\(\mu s\) respectively. No deviations were observed in comparison with the control experiment. Hence, even under use case circumstances no interference is observed with respect to execution of Application G.
Within this chapter we have described an experimental procedure to measure the event response times we have determined theoretically in Section 4.5. Then we used the experimental procedure to demonstrate the correctness of the platform model. The result shows that our theoretically determined best-case response time is very close to reality. However, the theoretically determined worst-case event response time was found to be too small. We expected a worst-case response time of 0.5\(\mu s\). In reality, the worst-case event response time is equal to 0.8\(\mu s\). We have described two possible causes and, where possible, we have described what the impact would be on the theoretical derived event response times. We were unable to verify possible causes of the deviation, because of lack of information about the platform. Then we repeated the experiment in which we varied the NoC clock frequencies. We were able to demonstrate by that experiment that the clock frequencies have a linear relationship with the event response times. Then we did an experiment where we injected artificial load on the NoC. We have done that in six different ways. Each time we injected the load in a different way so that each time one or more NoC components were shared with Application G. In none of the six different experiments the measured event response time of Application G was affected. Because of the results from the experiments with artificial load injections, we suspected that the response times of Application G are never influenced by any form of traffic on the NoC whatsoever. In order to enforce that presumption, we performed a final experiment, which is based on a use case of the platform. Instead of injecting artificial load on the NoC, we ran an OS on the platform with a multimedia application on top that plays back an HD movie. During the execution of the use case experiment a lot of different transactions are active on the NoC. The results of the experiment have been shown that even with this use case experiment the event response times of Application G are not affected.

Looking at the results of all experiments together, we conclude that guarantees can be given to the event response times of Application G and thus guarantees can be given to the response times of Application Q. However, because the actual measured worst-case event response time of Application G was higher than expected, we need to adjust the in Subsection 4.2 calculated response time of Application G.
Q. The response time of application Q would be $R_{Q,k} \approx 1.5 ms$ according to our calculations. Because the contribution of the NoC to the response time of Application Q is so small, $R_{Q,k}$ would not change significantly even if we assume that in the worst-case situation, the transactions on the NoC require 2, 3, 4 or even 10 times more time than calculated. So we can conclude that despite the fact that we do not know exactly the execution time of Application Q, $R_{Q,k}$ will be less than $D_{Q,k}$ and hence the requirement described in 3.1.2 is met.
Chapter 6

Conclusion

We have shown that it is possible to integrate Application Q on the OMAP4460 MPSoC together with other pieces of software, while being able to provide Application Q with temporal guarantees. We assigned dedicated processing power and instruction/data memory to Application Q such that Application Q has to use only one shared resource, which is the NoC. By means of modeling and hands-on experiments we evaluated the NoC (temporal) behavior. The evaluation allowed us to conclude that traffic on the NoC from other pieces of software will not interfere with traffic from Application Q. Furthermore, the evaluation allowed us to come up with response time bounds for Application Q.

For the purpose of the evaluation of the NoC (temporal) behavior, we derived a platform model. The platform model includes a basic representation of computational and instruction/data storage resources and the expected (temporal) behavior of the NoC. Given the platform model and the deployment of Application Q, we derived estimations on the worst- and best-case response time of Application Q in isolation. Based on the theoretical derived response time estimations, we could already conclude that the timing requirements of Application Q are met. I.e. assuming no interference, the relative response time is smaller than the relative deadline. However, since the platform model is based on significant number of (NoC) assumptions, we had to validate the model. Furthermore, we had to evaluate the possible temporal consequence of deploying additional applications on the platform. We addressed both requests by means of NoC round-trip-time (RTT) experiments. The experiments allowed us to validate the platform model and to show that additional load on the NoC does not influence the response time of Application Q.

6.1 Future work

We limited our work to the OMAP4460. However, our modeling and experimental approach should also be applicable on other MPSoCs for which the (temporal) behavior of the NoC is unknown. Whenever one has the possibility to simulate traffic on the NoC and to setup measurements as described in this thesis, one should be able to at least setup similar experiment.

For the purpose of the NoC evaluation we measured the RTT of packets traveling over the NoC between a dedicated subsystem, on which Application Q runs, and a peripheral device. We used a different subsystem to inject traffic into the NoC. The experiment allowed us to conclude that the temporal requirements of Application Q will always be met. However, if one is interested in a more global understanding of the NoC behavior, one could reverse the experiment. I.e. could find it interesting to measure the RTT of packets traveling between a different subsystem and a peripheral device and inject load from the dedicated subsystem.

Another optional experiment would be to use interrupts when accessing peripherals instead of the polling based method. Often separate paths next the NoC are used for interrupt signals and hence are also not affected by the traffic on the NoC.
Bibliography


Appendix A

Quadcopter hardware

This appendix describes (1) the made design decisions with respect to the individual quadcopter components, (2) the properties of the components and (3) the relation between the individual components. First, an overview of the quadcopter components is given in Figure A.1. Afterwards each and every individual component (labeled a - k) is briefly described.

Figure A.1: Quadcopter hardware components
In practice, one often buys components a-f from the figure in combination with a flight controller and a radio transmitter plus receiver. Then, after assembly, one can start flying the quadcopter by using the radio transmitter (manual remote control). However, in our case, it is not allowed to use a COTS flight controller. Instead, we use a general-purpose computational development platform (Pandaboard ES) which does not include any motion tracking sensor or any other quadcopter related piece of hardware. The advantage of using such a general-purpose platform is that it can be used for an almost unlimited variety of applications. E.g. on-board video processing is one of the many applications one could consider to implement. Unfortunately, using such a general-purpose platform comes with the consequence that additional hardware components are required next to components labeled a-f in Figure A.1.

In next sections each individual hardware component is discussed. The components are ordered in chronological order. I.e. the first discussed component is also the first component that we considered.

A.1 Frame (a)

The frame is the support structure of the quadcopter. Selection of the frame is basically a trade-off between robustness, weight, size, cost and mounting options. E.g. a frame made from thick stainless steal is very robust but also very heavy. More weight implies more required thrust from the motors and more thrust implies more power consumption and last but not least more power consumption results in shorter time of flight. Hence, weight is a very important factor which also determines the required parameters for other quadcopter parts such as the motors and the power supply. In practice, people who build quadcopters pick their frame from a web shop depending on their application. E.g. people who want to realize quadcopters that fly in formation indoors[5] choose a small frame, on the other hand people who want to build a quadcopter that is able to carry a human[6] choose a big frame. For this project, a frame with a span of 450mm (motor axis to motor axis), called the SK450[1], was selected. The main frame is built from glass fiber while the arms are constructed from polyamide nylon. The weight of the frame is equal to 393g. Propellers from up to 10 inches in diameter can be used. The frame is solid, lightweight, it meets the span requirement (including 10 inch propellers) and it has enough mounting options for a computational platform and other components like vision cameras etc.

A.2 Motors (b)

The motors of the quadcopter are assigned, in combination with the attached propellers, to generate lift. In practice the so-called brushless motors[7] are used because of their efficiency (85 − 90%), speed accuracy and great dynamic bandwidth (related to how fast the motor can change its speed). In order to determine which motors were needed, first an estimation of the total weight of the quadcopter was made, including the eventually extra weight that it has to carry in the future. Once the weight of the quadcopter is determined one can calculate the total amount of thrust that has to be generated in order to make sure that the quadcopter is able to fly and eventually perform some acrobatic maneuvers. The minimum thrust-to-weight ratio is $2 : 1$ for flying and at least $3 : 1$ if the quadcopter also has to perform acrobatics[8]. Besides the lift criterion, there are three other criteria for selecting the right motors:

- Weight, the lighter the better.
- Operating voltage, because this determines the number of battery cells that are required.
- Max. current, this determines which electronic speed controller (ESC) is required.

The weight of the quadcopter was estimated at about 1kg without payload and 2kg with payload. Hence, in order to achieve the $2 : 1$ thrust-to-weight ratio we have to at least generate 4kg of thrust. We picked motors called NTM Prop Drive 28-30S 800KV / 300W Brushless Motor (short shaft version) from the Hobbyking web shop[1]. Specifications of the picked motors are presented in Table A.1.

---


### A.3 Propellers (c)

The selection of the propellers was based on the motor specifications described in the previous section and online discussions. Also the 10 inch diameter limitation which applies for the selected frame played an important role during the selection. In general the following rule is applied: if one want to achieve high air speeds, one should select small propellers with a large pitch. On the other hand, if one want to fly at slower air speeds with more control, then larger propellers with a smaller pitch angle should be selected. For our project we selected propellers with a diameter of 10 inch and a 4.8 inch pitch. According to table A.1 each motor should then be able to deliver a maximum thrust of 1.27kg.

### A.4 ESCs (d)

After considering the motors and propellers, one has to consider the ESCs. An ESC is an electronic circuit with the purpose to vary an electric motor’s speed, its direction and possibly also to act as a dynamic brake. An ESC is typically controlled with a pulse-width modulated (PWM) signal. Selection of the ESCs is done by answering the following questions: (1) What is the maximum current that one motor may pull? And (2), what refresh rate for updating the motor speed do we need? We answered the first question by simply applying the motor specifications. The motor may potentially pull 17.3A assuming an operation voltage of 18.5V. However, most ESCs do not support voltages higher than 14.8V and therefore we assumed a higher current pull (>20A). The second question can be answered by examining the experiences from quadcopter hobbyists. Nowadays a lot of quadcopter builders use an ESC with a refresh rate of 400Hz. We decided to use Turnigy Multistar 30 Amp ESCs. As the name suggests the ESCs support a maximum current pull of 30A. Furthermore, the ESCs support an PWM update frequency of 400Hz. The ESCs should be used with a 2 to 4 cells LiPo battery.

### A.5 Battery (e)

The fifth quadcopter component we consider is the battery. The battery is required for powering the motors and any other piece of electronic hardware that is mounted on the quadcopter. While picking the right battery the following rules of thumb are of high importance: (1) the battery has to be as light as possible to keep the quadcopter as light as possible, (2) the discharge rate of the battery should be high enough, i.e. the battery should be able to survive peak current pulls from the motors, (3) the voltage of the battery has to be high enough to power the motors and (4) the capacity of the battery should be big enough to allow reasonable time of flight. The trade-off between the weight of the battery and capacity usually depends on the quality of the battery. The required discharge rate on the other hand depends on the motors and ESCs we picked. The required discharge rate can be calculated using the following formula \( \text{Discharge rate} = \frac{\text{Max. current}}{\text{Battery capacity}} \) [7]. The maximum current (Max. current) is the sum of currents through each motor running at full load plus the maximum current required by the power supply to power the electronic devices connected to it. The voltage of the battery depends on the number of battery cells and must be adapted to the voltage requirement of the motors. Each battery cell of a LiPo battery delivers a voltage of 3.7V. In practice, LiPo batteries with 2 to 6 cells are used and hence the voltage is typically within the range 7.4V – 22.2V.

For the purpose of our project we used a 4000mAh, 4 cell LiPo battery which supports a maximum discharge current between 140A – 280A which is larger than the maximum the ESCs can potentially pull.

---

---

---

---
$30A \cdot 4 = 120A$. The weight of the LiPo battery is 427g.

A.6 Components (f) - (i)

Component (f) is power distribution PCB which distributes the power from the LiPo battery among the four motors and the DC-DC converter (component (i)). Component (g) converts commands which are received from the $I^2C$ bus into PWM signals for the the ESCs. The level shifter, component (h), is required because the $I^2C$ to PWM converter expects an $I^2C$ signal with a logic high rated between $3.3V - 5V$ while the Pandaboard ES only supports an $I^2C$ signal with logic high rated at $1.8V$. We use a DC-DC converter to convert the varying voltage of the LiPo into a fixed $5V$ which is required to power the Pandaboard ES, the level shifter, the $I^2C$ to PWM converter and the sensors.

A.7 Sensor board (j) and Pandaboard ES (k)

Both the sensor board and the Pandaboard ES is discussed in Chapter 2.
Appendix B

Hardware used during experiments

In Figure B.1 the experimental setup is presented again. In the next sections some more information about the used hardware is provided.

B.1 Pandaboard ES (OMAP4460)

Information regarding the Pandaboard ES development board which is used during the project is provided below:

- Name: Pandaboard ES (OMAP4460)
- Revision: B1
- PCBA: 750-2170-002 REV B
- S/N: U4460511607

B.2 Micro SD card plus adapter

Information regarding the SD card which is used in combination with the Pandaboard ES development board during the project is provided below. The SD card contains the software images used during the experiments.

- Name: SanDisk Ultra micro SD card
- Size: 8GB
- Type: Micro SD HC-I
- Speed: UHS class 1 (10MB/s)
- Adapter: SanDisk micro SD to SD card adapter

B.3 Laptop

Information regarding the laptop which is used during the experiments to monitor the results is provided below:

- Name: ASUS N51Vf-SX019C Notebook
- Processor: Intel® Core™2 Duo Processor T6400
- Memory: 2048 MB, DDR2 800 MHz SDRAM,
- Graphics card: NVIDIA® GeForce® GT 130M, with 1GB VRAM
- Harddisk: 2.5" SATA 320GB, 5400rpm
B.4 USB to serial converter

Information regarding the USB to serial converter which is used during the experiments for communication between the laptop and the Pandaboard ES is provided below:

- Name: Sitecom USB to Serial cable CN-104
- OS support: Windows and Linux
- Compatibility: USB 1.1 and USB 2.0

B.5 Wave generator

Information regarding the wave generator which is used for the experiments is provided below:

- Name: Wavetek Meterman FG2C Function Generator
- Frequency range: \(0.3Hz - 3MHz\) (used: 1kHz)
- Waveforms: Sine, triangle and square (used: square wave)
- Used output: Main output terminal (50Ω)
- Waveform adjustments: Duty cycle, TTL/CMOS, DC offset and amplification

B.6 Logic analyzer

Information regarding the logic analyzer which is used for the experiments is provided below:

- Name: Saleae Logic 16
- Sample frequency: Max. 100MHz (used: 100MHz)
- Number of inputs: 16 (used: 2)
- Input voltage range: \(-0.9V\) to \(6V\)
Appendix C

Software used during experiments

A list of the software used during the experiments is given below:

- Bare metal c code on the Cortex A9 processors, to simulate NoC traffic.
- Bare metal c code on the Cortex M3 processors, to implement Application G.
- Picocom, version 1.4, for serial communication between the Pandaboard and the laptop.
- Saleae Logic, version 1.1.15, for logic analyzer measurements.
- Java code, to parse the logic analyzer data, the tool searches for signal transitions.
- Gnuplot, to create plots from the logic analyzer data.
- The R project, to extract statistical information from the logic analyzer data.