A Predictable Task Migration Mechanism with Partial Application Stalling on a MPSoC

Tuzzi, A.

Award date:
2020

Department of Mathematics and Computer Science
Electronic Systems Research Group
Verintec Solutions B.V.

A Predictable Task Migration Mechanism with Partial Application Stalling on a MPSoC

Master Thesis

MSc candidate

Alberto Tuzzi

(s: 1284274)

Supervisors:

Dr. D. Goswami
Dr. Ir. A. Nelson

Committee members:

S. Tabatabaei Nikkhah
Dr. S. Zinger

Eindhoven, February 2020

Dedication

To my family, my girlfriend, my friends.

"I think it is possible, for ordinary people, to choose to be extraordinary"
E. Musk

Preface

The work in this thesis project and the whole master program journey would have not been possible without family, friends and colleagues, who supported me throughout this period.
First, I would like to thank my supervisors: Dip Goswami and Andrew Nelson. Not only for providing me with a challenging project to complete my studies but also for their work ethics and passion for research that always inspired me after every meeting.
A special thanks to my project advisor Shayan Tabatabaei Nikkhah, who was always available to help me in solving major issues regardless of his undergoing PhD duties.
I would also like to thank Martijn Koedam for the support on any technical issue that came along during the project development.
I would like to thank all of my colleagues at the Technische Universiteit Eindhoven who inspired me with their passion and hard work: Inaki, Frederik, Saharsh and, especially, Bharat who was teammate for most of the academic assignments.
I would like to thank all of my closer friends who helped me through my academic years and many steps of life: Niccolò, Luca, Davide, Simone, Karolis, Alessandro, Christian, Luca and Pietro.
A special thanks to my girlfriend and front line supporter Ieva, whose unconditional love helped during the hardest times to get back on track for pursuing my dreams.
On top of all, a big thanks to my family. My parents Lucia and Antonio for all they have done for me and my brother in order for us to grow as human beings and be able to pursue our dreams. A special thanks to my younger brother Alessandro, who has been the closest person to me in my life not only in terms of blood, but also passions and interests and whose presence always reminded me to do the best in being the older brother, and, by doing so, was the main factor into the completion of my adulthood maturity process.

Abstract

On MPSoC platforms, events such as a change in the requirements of the application, the necessity to load new functionalities or faulty resources are the reason why these often need a Resource Management System (RMS) capable of enacting resource management techniques with the objective of modifying resource allocations and bindings to overcome the aforementioned situations. In many cases, the undertaken technique is task migration, where applications are stalled and their tasks moved in execution to other resources on the platform. However, stalling the application completely leads to a more impactful performance degradation by introducing a larger time overhead. In fact, with total stalling, the application does not deliver output or does not operate intermediate jobs that lead to the it, during the migration time window. Moreover, if the environment is not predictable, the migration process overhead cannot be estimated and its behaviour will be then timely unpredictable, e.g. no upper bound can be defined beforehand or, most importantly, at runtime (by the RMS, for instance). In this thesis work, we propose a solution to this issue: we design a task migration mechanism that does not stall the entire target application and we make it predictable by computationally modelling the target application and running it on a predictable host MPSoC; we refer to this approach as *partial-stalling*. Being the target application modelled with the data-flow model of computation, we implement the partial-stalling task migration among two tiles as a dynamic rebinding of the involved actors (the task wrappers) and edges (in the form of FIFO buffers) that enable communication between them, leaving the non involved ones unaffected and free to keep executing where conditions to fire are met. With analysis purposes, the mechanism itself is divided in phases, using the communication barriers in it as phase boundaries, and for each one of them the upper execution time bound is predicted. The time model for the whole migration is then derived, combining the phases, in order to predict the Worst-Case Migration Time ($WCMT_{migrated\_task}$), and experimented on the presented case study. The results of the experimentation show that the developed time model is capable of computing an upper bound for any migration with a reasonable pessimistic error (of about 25%, at most 55% in particular approximation situations). Here we also show that the overall time overhead δ, introduced by the partial-stalling migration on the response time of the application, is always smaller or at most equal to the overhead introduced by the same migration with a full-stalling approach; demonstrating also that the latter, in any scenario, is always an upper bound to the former and therefore a non optimal approach.

Contents

Contents ix

List of Figures xi

List of Tables xv

Abbreviations xix

Definitions xxi

1 Introduction 1
- 1.1 Basic Concepts . . . 2
  - 1.1.1 Multi-Processor System on Chip . . . 2
  - 1.1.2 Data-Flow Model of Computation . . . 2
  - 1.1.3 Predictability and Composability . . . 4
- 1.2 Problem Statement . . . 5
- 1.3 Case Study . . . 5
  - 1.3.1 Image Processing Application: JPEG Decoder . . . 5
  - 1.3.2 Verintec MPSoC . . . 7
- 1.4 Thesis Structure . . . 9

2 Methodology 11
- 2.1 Proposed Solution . . . 11
  - 2.1.1 Dynamic Reconfiguration . . . 12
  - 2.1.2 Partial-Stalling Task Migration . . . 12
  - 2.1.3 Project Milestones . . . 13
- 2.2 Related Work . . . 14

3 Design and Implementation of the Solution 19
- 3.1 Test Application . . . 19
- 3.2 Multiprocess Instance . . . 20
- 3.3 RTOS Library `pose` Expansion: the Migration API . . . 21
  - 3.3.1 Task Duplication . . . 21
  - 3.3.2 Migration API . . . 22
- 3.4 Migration Daemon . . . 24
- 3.5 Migration Sequence . . . 26
- 3.6 Partial-Stalling Task Migration Example . . . 29

4 Time Model and Experimentation 33
- 4.1 Time Model . . . 33
  - 4.1.1 Migration Phases Worst-Case Times . . . 33
  - 4.1.2 Time Division Multiplexing Worst-Case Scenario . . . 35
  - 4.1.3 Subsequent Migrations . . . 39

4.2 Experimentation on the Case Study and Results 39
4.2.1 First Measured Migration: *CC* task 40
4.2.2 Second Migration: *IQZZ* task 45
4.2.3 Subsequent Migrations: *CC* then *IQZZ* 51
4.2.4 Impact of δ on the application $WCRT$ 52
4.3 Summary of the Results 55

5 Conclusions **57**

Bibliography **59**

A Verintec MPSoC **61**

B JPEG decoder **63**

C Migration API **65**

List of Figures

1.1 A generic MPSoC structure made of multiple processing and memory blocks that can communicate through an interconnect, as depicted in [8]. 2
1.2 The Data-Flow modelling happens by means of three elements: actors, edges and tokens 3
1.3 An example DF application graph made of three actors 4
1.4 Gantt chart showing the actor execution flow of the example application from Figure 1.3 4
1.5 The data-flow model of the *JPEG decoder* 6
1.6 Partial-Stalling of the *JPEG decoder* in case *IQZZ* is being migrated (red). Orange tasks are also involved due to direct communication and are also stalled, cyan tasks keep executing 7
1.7 A schematic of the used Verintec image 8
1.8 A possible TDM scheduling table for a VEP on *Verintec* 9

2.1 An example DF graph with the highlighted *RoC* of the *yellow* task 12
2.2 An high level example of a partial-stalling task migration mechanism 13
2.3 Gantt chart showing a mock task execution flow during the partial stalling task migration from the example of Figure 2.2 13
2.4 In a large MPSoC, a task migration might have the secondary duty of freeing space for, like in the depicted example, a cluster of processing units that are to be assigned to other applications 15
2.5 In several task migration implementations, the context or state of the migrated task is intermediately saved on a shared memory space where both source and destination PEs have access 16

3.1 The data-flow model of the simplistic test application 20
3.2 Structure comparison of the *POSIX* multiprocess instance and the *Verintec* platform 21
3.3 Task duplication of the test application on two tiles 21
3.4 The `pose` hierarchy of structures 22
3.5 Migration state machines describing the update of Tasks and FIFOs attributes with the Migration API 23
3.6 With the migration API expansions, the suspension of portions of the application is now possible 23
3.7 Application mapping from the *POSIX* multiprocess to the *Verintec* platform . . . 24
3.8 Function calls in the migration sequence. The communication steps that occur in the migration sequence are the horizontal arrays 26
3.9 Migration actions flow diagram: Source Phase I 27
3.10 Migration actions flow diagram: Dest. Phase I 27
3.11 Migration actions flow diagram: Source Phase II 28
3.12 Migration actions flow diagram: Dest. Phase II 28
3.13 Migration actions flow diagram: Source Phase III 29
3.14 Migration actions flow diagram: Dest. Phase III 29

3.15 The test DF application initial mapping 30
3.16 Migration of *SQR* sequence source PHASE I: suspension of *SQR* and the *RoC* . . 30
3.17 Migration of *SQR* sequence dest. PHASE I: suspension of the *RoC* and activation of *SQR* 30
3.18 Migration of *SQR* sequence source PHASE II: push of the FIFO data and deactivation of *SQR* 31
3.19 Migration of *SQR* sequence dest. PHASE II: pull of the FIFO data 31
3.20 Migration of *SQR* sequence source PHASE III: resumption of the *RoC* 31
3.21 Migration of *SQR* sequence dest. PHASE III: resumption of *SQR* and the *RoC* . . 32
3.22 Gantt chart showing a possible test application execution flow before, during and after the migration of *SQR* 32

4.1 Table schematization of the migration phases 35
4.2 An example of the TDM frame structure used in the experiments. The *migration daemon* is part of the System Application 36
4.3 The worst-case scenario TDM frame sequences graphical analysis tools 36
4.4 The worst-case scenario assuming no preemptions happen for the migration mechanism 37
4.5 The worst-case scenario assuming a preemption happens for phase S_{II} 37
4.6 The worst-case scenario assuming a preemption happens for phase D_{II} 38
4.7 The graph shows a comparison of the expected time amount needed to move each of the seven FIFO buffers 40
4.8 The JPEG decoder experimentation initial mapping 40
4.9 The graph in figure represents both the computed and measured execution cycles of the source migration functions stacked one onto each other for the *CC* task migration 42
4.10 The graph in figure represents both the computed and measured execution cycles of the destination migration functions stacked one onto each other for the *CC* task migration 42
4.11 The graph in figure shows the difference between the WCMT foreseen by the model and the worst obtained measurement for the *CC* task migration 45
4.12 The JPEG decoder mapping after the *CC* migration 45
4.13 The graph in figure represents both the computed and measured execution cycles of the source migration functions stacked one onto each other for the *IQZZ* task migration 47
4.14 The graph in figure represents both the computed and measured execution cycles of the destination migration functions stacked one onto each other for the *IQZZ* task migration 47
4.15 The graph in figure shows the difference between the WCMT foreseen by the model and the worst obtained measurement for the *IQZZ* task migration 50
4.16 The JPEG decoder mapping after the *IQZZ* migration 50
4.17 Comparison of the migration phases duration of the *CC* and *IQZZ* migrations . . 51
4.18 Comparison of the migration phases duration of the *VLD* migration 51
4.19 The graph in figure shows the difference between the WCMT foreseen by the model and the worst obtained measurement for the subsequent migrations of *CC* and *IQZZ* 52
4.20 The Gantt chart in figure shows an hypothetical median case flow scenario for the *IQZZ* migration 53
4.21 The Gantt chart in figure shows an hypothetical worst-case flow scenario for the *IQZZ* migration 53
4.22 The Gantt chart in figure shows an hypothetical best case flow scenario for the *IQZZ* migration 54

A.1 A schematic of the *Verintec* virtualization 61
A.2 The *PYNQ-Z2* board 62

B.1 `cat.jpg`, the used test image to keep the JPEG decoder application running . . . 63

List of Tables

4.1 The JPEG decoder FIFOs and their size 39
4.2 The single migration API functions worst-case execution times for the *CC* migration scenario 41
4.3 The migration phases worst-case execution times for the *CC* migration scenario . . 41
4.4 Comparison of the single functions expected execution times and measured execution times for the *CC* migration scenario. The relative error is also shown 42
4.5 The subscenario I TDM frames of the tiles for the *CC* migration 43
4.6 The subscenario I measured values for the *CC* migration 44
4.7 The subscenario II TDM frames of the tiles for the *CC* migration 44
4.8 The subscenario II measured values for the *CC* migration 45
4.9 The single migration API functions worst-case execution times for the *IQZZ* migration scenario 46
4.10 The migration phases worst-case execution times for the *IQZZ* migration scenario 46
4.11 Comparison of the single functions expected execution times and measured execution times for the *IQZZ* migration scenario. The relative error is also shown 47
4.12 The subscenario I TDM frames of the tiles for the *IQZZ* migration 48
4.13 The subscenario I measured values for the *IQZZ* migration 48
4.14 The subscenario II TDM frames of the tiles for the *IQZZ* migration 49
4.15 The subscenario II measured values for the *IQZZ* migration 49
4.16 The subsequent migrations scenario TDM frames of the tiles 51
4.17 The subsequent migrations measurement results 52

B.1 The *JPEG decoder* actors worst case response times for decoding `cat.jpg` 63

Listings

3.1 Pseudocode for the behaviour determination of the *migration daemon* 24
3.2 Pseudocode for the source behaviour of the *migration daemon* 25
3.3 Pseudocode for the destination behaviour of the *migration daemon* 25
4.1 The generic `time()` function 34
C.1 Simplified C code representation of the suspension and resumption of the τ adjacent tasks API functions 66
C.2 Simplified C code representation of the push state and pull state API functions . . 67
C.3 Timing equations to compute the Suspension/Resumption API functions and Activation/Deactivation API functions worst case timings 69
C.4 Flag reading and writing worst case timings 70
C.5 Timing equations to compute the Suspension/Resumption of τ adjacent tasks API functions worst case timings 70
C.6 Timing equations to compute the Push FIFO state/Pull FIFO state of τ FIFO buffers API functions worst case timings 71

Abbreviations

ACB ..... Application Control Block

API ..... Application Program Interface

DF ..... Data-Flow

FCB ..... FIFO Control Block

FIFO ..... First In First Out

LM ..... Local Memory

MoC ..... Model of Computation

MPI ..... Message Passing Interface

MPSoC ..... Multi-Processor System on Chip

NoC ..... Network on Chip

OS ..... Operating System

PE ..... Processing Element

PSTM ..... Partial-Stalling Task Migration

RMS ..... Resource Management System

RoC ..... Region of Communication

RTOS ..... Real Time Operating System

SA ..... System Application

SDF ..... Synchronous Data-Flow

SM ..... Shared Memory

SoC ..... System on Chip

TCB ..... Task Control Block

TDM ..... Time Division Multiplexing

VEP ..... Virtual Execution Platform

WCET ..... Worst-Case Execution Time

WCMT ..... Worst-Case Migration Time

WCRT ..... Worst-Case Response Time

Definitions

API ..... Interface meant to simplify communication between software parts

COMPSoC ..... Predictable and Composable SoC virtual platform design flow

Daemon Middleware . Software running transparently in background

Design Time ..... Phase of design of the mechanism; e.g. before any use case application

Destination Tile ..... Tile to which a task to migrate is moved

Embedded System ... Computer system running within a larger system

Reconfiguration ..... Element remapping mechanism (happens at runtime if dynamic)

Measurability ..... Characteristic of a system of being quantifiable

μkernel ..... Software virtualization of kernel application on a virtual platform

Run-Time ..... Stationary phase of the software execution; e.g. main program

Source Tile ..... Tile from which a task to migrate is moved

Startup-Time ..... Transient phase of the software execution; e.g. initialization

System Application .. Privileged system management super application

Task ..... Sub-element of a software application

Task Suspension ..... Temporary suspension of the task scheduling after the last execution

Task Deactivation .... Permanent deactivation of the task scheduling

Time Overhead ..... Additional execution time introduced by a reconfiguration mechanism

Upper Bound ..... Design limit on a system measure; e.g. execution time

Virtual Platform ..... Device that makes use of virtualized processing and memory blocks

Chapter 1

Introduction

Resource management techniques such as task migration, are used by resource management systems (RMS) to achieve different hardware usage settings; in the case of task migration, this happens by moving the execution and state of the software, among resources. In scenarios such as dynamic change of software requirements, need to free resources for new functionalities or faulty execution on a processing unit, task migration may be applied to change the use of a resource or free/load it completely.
State of the art task migration techniques (that we will discuss in Section 2.2) use full application stalling to bestow the migration but do not address the impact of the overhead introduced by this matter; full stalling causes the target application to not deliver output or to not compute intermediate steps that lead to it, for the duration of the mechanism, adding time overhead to the worst-case response time ($WCRT$) of the application execution. Then, the more frequently a RMS triggers migrations, the more the overall performance may be prone to degradation. This is not ideal for time-critical applications such as applications in medical software, automotive or avionics, where high performance and fault tolerance are high priority requirements. In this thesis we aim to tackle this by designing a task migration mechanism able to perform the duty without stalling the entire target application (we will address this novel behaviour, from now on, as *partial-stalling*), allowing the application to execute part of its duties without waiting, and aiming therefore to reduce the cumulative overhead caused by the migrations.
Of course, the impact of this introduced overhead will be proportional to the worst-case reconfiguration time of the applied technique (that we will call worst-case migration time, $WCMT$, in the case of task migrations). Nonetheless, estimating worst-case timings for these mechanisms is not always possible; preventive or runtime measurability of time, comes from defined characteristics of the used hardware architecture (and firmware), with the most important being predictability, a concept that describes every event occurring on the platform as being bound by a maximum event response time (later explained in Section 1.1.3). Henceforth, in this thesis we also focus on the importance of having a predictable environment and how this leads to the achievement of a predictable mechanism. The MPSoC under study is, in fact, predictable itself (see 1.3.2) and so is the model of computation used on the target application (see 1.1.2). The achieved predictability of the MPSoC-application system is then essential to develop a mathematical time model that gives the possibility to estimate an upper bound for the developed mechanism in a given mapping. We will see that such timing model predicts an upper bound on the $WCMT_X$ for a migration X; combining this with the object application $WCRT$, also an estimation of the overall time overhead introduced by the migration X on the application response time is given.
Before proceeding with the main dissertation, important concepts for the understanding of this document will be introduced in the first section of the chapter.

1.1 Basic Concepts

In order to have a better understanding of certain technical aspects often mentioned in this thesis, this section will go through them in more detail. Starting from the definition of *Multi-Processor System on Chip* (MPSoC), we will go then through an introduction of the *Data-Flow* (DF) *Model of Computation* (MoC), and close with the definitions of *Predictability* and *Composability* for an MPSoC.

1.1.1 Multi-Processor System on Chip

A *Multi-Processor System on Chip* (MPSoC) is, first of all, a *System on Chip* (SoC): an integrated circuit that has all the component blocks of a computer system (Processing Elements, Memory Blocks and Input/Output Ports). If the Processing Elements (PEs) are two or more, then the minimum requirement for the SoC to be an MPSoC is met. In such device, the communication between blocks is achieved through a shared channel that is generally named *Interconnect* [8]. Such entity can be as complex as an actual network made of buffers and address tables, to store, redirect and retrieve the data and thus bestow communication between a *master* block and a *slave* block (a *Network on Chip*, NoC)[14]; or it can be, for instance, a more flexible and easier to manipulate shared memory space with memory mapped accesses as in our case study (1.3). A graphical example of the described MPSoC generic structure can be seen in Figure 1.1; the blocks in the illustration represent the aforementioned SoC composing parts.

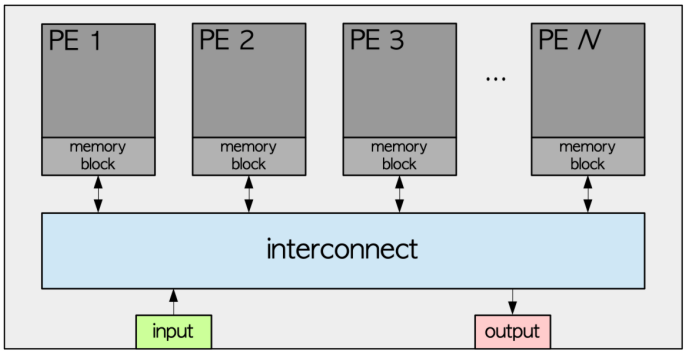


Figure 1.1: A generic MPSoC structure made of multiple processing and memory blocks that can communicate through an interconnect, as depicted in [8].

These platforms are used in embedded applications, including time-critical applications, and are therefore considered an embedded system device. An MPSoC will be used in the case study challenged by this thesis; it will be introduced in Section 1.3.2 of this introduction chapter.

1.1.2 Data-Flow Model of Computation

Another important information to know about our study is the model of computation (MoC) of the object application(s). In order to have an analyzable environment overall, the latter must be, in fact, computationally modelled. When dealing with measurements on embedded applications, it is important that such applications are modelled by following a MoC since this allows to get measures (in terms of time, memory usage, energy consumption and more, depending on the used model).

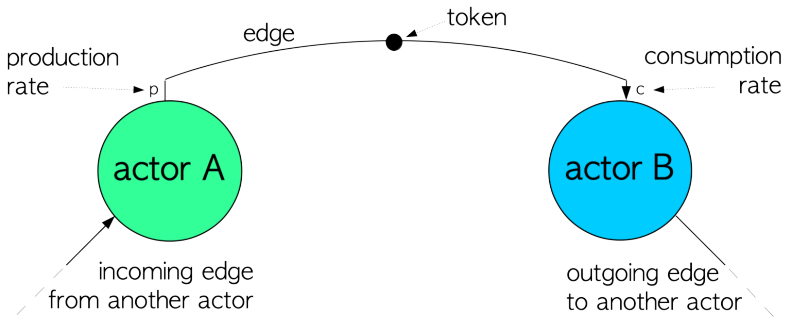


Figure 1.2: The Data-Flow modelling happens by means of three elements: actors, edges and tokens

In this thesis work we are interested in measuring time. In the case of data-flow (DF) MoCs, applications are sub-divided into single *actors* that fire and create *tokens* when the necessary number of incoming tokens are available and there is available space to store the newly generated ones. We identify the actors, in our work, with the software tasks; although we must take in account that actors *are stateless*. Actors, in fact, merely execute their duty over tokens (the data), regardless of the previous executions, and then reset; in order to *save the state* in this model, *self edges* must be added on the actor where to save tokens that have information useful for future executions. In this way, when they fire, they have the necessary information on the incoming token from the self edge, and, when done executing, they save the necessary information on the newly generated token. Hence, allowing us to consider the two, within the boundaries of our work, as equipollent. Communication and data dependencies are implemented through *edges* that connect the actors and allow to store tokens and move them. These entities, illustrated in Figure 1.2, quantize the three main elements of the application: tasks (into actors), data (into tokens) and communication dependencies (into edges). Considering the DF application singularly, without considering the platform it is running on, we can be sure that the quantization introduced by the model allows to have defined response times of the single actors, with defined communication times through the edges and with defined data size for tokens [11]. In such model, the application has, therefore, a predictable behaviour (*worst-case execution times* for actors are definable) and therefore it is timely measurable [3]. Our target application shall be in fact modelled as a *synchronous data-flow application* (SDF). In this type of DF application, parameters such as firing rules for actors and buffer sizes in the edges, do not change; moreover, the actors never change in number nor duty. With such model a standard worst-case execution scenario of the application can be defined, it is then possible to do timing analysis.
In a whole application-platform system, however, the global measurability depends not only on the application but also on the platform. This one must in fact have a *predictable* behaviour (a concept that will be explained right away, in 1.1.3).
One more important information, is that the single DF actors, as anticipated, are no other than the application tasks wrapped up into unit of execution entities. This introduced *atomicity* of tasks helps to make possible and to standardize operations over the application elements. Such characteristic will be utterly exploited in this thesis project (as we will see later in Section 2.1) during the implementation phase.

To define the cumulative *WCRT* of the SDF model, we consider the example graph of Figure 1.3 and we assume that the *input data* is always available when actor A is ready to fire (for instance, images that are already stored in memory in the case of an image processing application). In such scenario, the graph *repetition vector*, namely the minimal ratio of firing of the actors in a graph iteration, is $[2, 1, 2]$ (relative to $[A, B, C]$). Meaning that every iteration of the graph that leads to *output data* delivered by actor C, needs two executions of A, one of B and again another two of C (these are shown, stacked, on the Gantt chart in Figure 2.3). This implies that the *WCRT* of

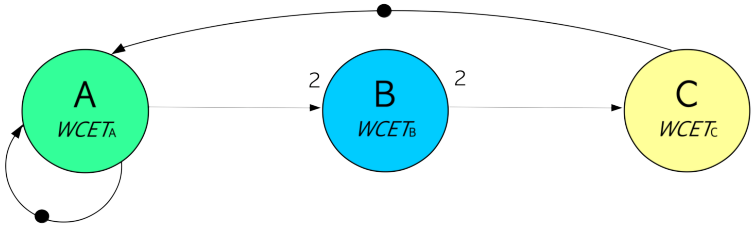


Figure 1.3: An example DF application graph made of three actors

an iteration of the graph would be the following:

$$WCRT_{app} = 2 * WCET_A + WCET_B + 2 * WCET_C$$

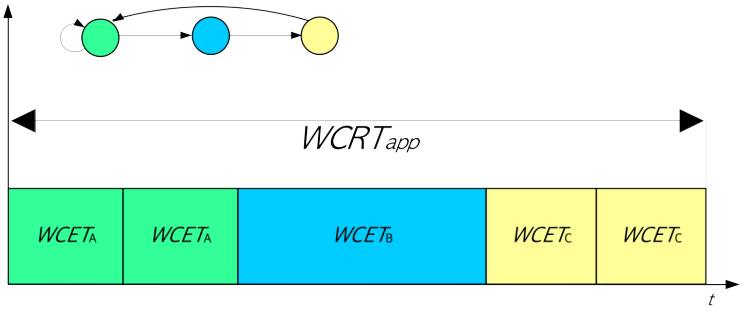


Figure 1.4: Gantt chart showing the actor execution flow of the example application from Figure 1.3

1.1.3 Predictability and Composability

Predictabiliy is a characteristic which, within an embedded platform, means that any kind of executed operation or mechanism is bounded by a worst-case response time. Predictability is hence a key to *measurability*, since having any kind of process bound by a worst-case response time means that global response times for defined mechanisms can be defined [13] (where for mechanism we intend a programmed chain of events on the MPSoC). The heterogenous MPSoC provided by *Verintec* [1] (that we will refer to often as the Verintec platform, for simplicity), which is going to be the platform of our case study (1.3.2), is also predictable (being architecturally based on the heuristics of the predictable platform guidelines of *CompSOC* [12]).
The Verintec platform happens also to be composable (another characteristic inherited by CompSOC). *Composability* is a characteristic of an MPSoC where any operation done by the platform to execute a certain application α does not affect in any way (time or memory wise) any other concurrently running β application [13]; composability in our project is not among the primary concerns, but it will also play a role in the validity of the presented solution implementation (as it will be explained in Section 2.1) and is useful to ensure global predictability of the environment (ensuring that interference from concurrent applications are predictable).
In conclusion, since we need our implemented platform-application system to be measurable as a whole to answer the problem of this project (more detail in 1.2), we need our final implementation to be predictable. We should then be able to compute an *upper bound* for the response time for all the sub-operations in any implemented mechanism. To achieve a predictable mechanism, we

need either any interference on the application to be predictable or no interference at all (this latter happens with composability). Therefore, the use of both *data-flow* (a predictable MoC) and *Verintec* (a predictable platform) must happen. At the same time, composability ensures that any interference from concurrent applications is already taken in account. Having these two elements, ensures that a mathematical model for timing analysis can be in the end crafted. This will be fundamental to determine the time upper bound and evaluate the impact of the implemented solution on the general problem that will be introduced in the next section.

1.2 Problem Statement

From the introduced background, two research questions are derived:

Q1: Can a dynamic task migration that acts without stalling the target application entirely, be designed and implemented?

Q2: Can the time overhead introduced by the partial-stalling task migration mechanism, be quantifiable in terms of migration time beforehand?

From the research questions, we can now derive the problem:

P: Design and Development of a partial-stalling task migration mechanism and juxtaposition of a preventive time model for the developed mechanism.

Hence, the focus of this thesis will move, from now on, on introducing the solution to P, answering in this way Q1 and Q2.

1.3 Case Study

The case study that is being considered in this thesis project is the following: a predictable (and composable) MPSoC with a relatively small number of PEs running a media processing application modelled in data-flow MoC (the platform details are given in Section 1.3.2). In a scenario like this where the availability of resources is dynamic and might become constrained, dealing with time requirements influences the whole platform resource usage. We want to, in fact, reproduce a situation where it can be possible to use the resources (PEs in this case) in different ways, with different loads.
It is, therefore, possible to remap the target media processing application in different ways over multiple processing elements as long as the resource constraints are not overtaken.

1.3.1 Image Processing Application: JPEG Decoder

The target application is named *JPEG decoder*. The purpose of this application is to decode an image file in the JPEG format to obtain the binary version of it. Since the JPEG format is the most used *lossy* format globally, a *JPEG decoder* (often together with the corresponding *JPEG encoder*) can be often part of bigger embedded applications that operate media processing and that might need operations over JPEG encoded images.

Data-Flow Model

The DF model of the application is formed by five actors (the software tasks): *VLD* (Variable Length Decoding), *IQZZ* (Zig-Zag Inverse Quantization), *IDCT* (Inverse Discrete Cosine Transform), *CC* (Color Conversion) and *RASTER* (Rasterization). The data dependencies can be seen in Figure 1.5 where the DF model of the application is represented. In our case study, the *input data* is always available for *VLD* and the buffers on the edges have the minimum possible size to achieve the repetition vector of $[1, 10, 10, 1, 1]$ (relative to $[VLD, IQZZ, IDCT, CC, RASTER]$),

so *VLD* cannot fire again until *output data* is delivered, e.g. *RASTER* executes and frees the edge between the two.

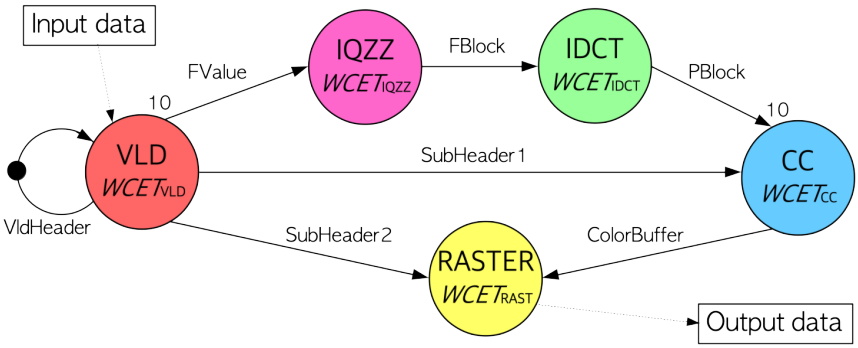


Figure 1.5: The data-flow model of the *JPEG decoder*

The worst-case execution times of the single actors depend on the image that is being decoded, so to ensure the consistency of the results presented in Section 4.2, we will use the same test image for every experimentation. More details about the workflow of the DF model, the worst-case execution time values and the test image, are given in Appendix B.
By following the example in Section 1.1.2, every single iteration is equal to:

$$WCRT_{jpeg} = (WCET_{VLD} + 10 * WCET_{IQZZ} + 10 * WCET_{IDCT} + WCET_{CC} + WCET_{RAST})$$

In this thesis, we are interested in showing the impact of our solution mechanism on the worst-case response time of the target application. In fact, we do not expect the throughput to change because of the mechanism (it rather depends on the mapping of the application over the tiles), we do expect on the other side to have an increase in the JPEG decoder worst-case response time, every time an execution of the solution mechanism happens, of a quantity δ:

$$WCRT^{mig}_{jpeg} = WCRT_{jpeg} + \delta$$

where the upper index *mig* indicates that a migration introduced overhead on the the $WCRT$ and δ symbolizes that time overhead introduced by the $WCMT$ on the $WCRT$.

Partial-Stalling Scenario

Assuming the use of the anticipated partial-stalling approach for the task migration mechanism on the JPEG decoder, we consider the following scenario: we see on Figure 1.6 an example of how a partial-stalling would affect the application (seen in its DF model graph form), in case *IQZZ* was issued for migration.

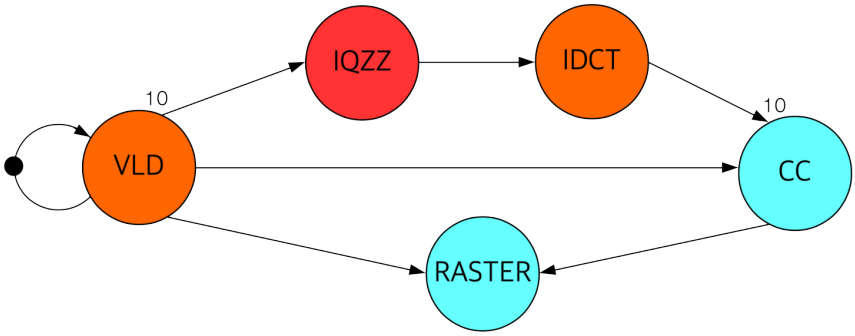


Figure 1.6: Partial-Stalling of the *JPEG decoder* in case *IQZZ* is being migrated (red). Orange tasks are also involved due to direct communication and are also stalled, cyan tasks keep executing

The tasks that need to be stalled during migration are identified as *Region of Communication*, a concept later described in 2.1.
In a classic (*full-stalling*) migration the iteration WCRT would be fully incremented by the Worst-Case Migration Time, e.g. δ equals $WCMT_{migrated\_task}$:

$$WCRT_{jpeg}^{fs\_mig} = WCRT_{jpeg} + \delta$$

where $\delta = WCMT_{IQZZ}$.

With the *partial-stalling* approach, δ will also depend on how much the migration lasts, but from this amount we will have to take away some time where the tasks not part of the *Region of Communication* can keep on executing (if conditions to fire are met) and is therefore not accountable as migration time:

$$WCRT_{jpeg}^{ps\_mig} = WCRT_{jpeg} + \delta$$

where $\delta \leq WCMT_{IQZZ}$.

Basically, the generic case of the same migration with full-stalling would be itself already an upper bound for the same migration with the partial stalling approach. In this thesis, we will quantify δ in Chapter 4, after an experimentation on our case study.

1.3.2 Verintec MPSoC

The MPSoC used in the presented thesis is, as anticipated, the *Verintec MPSoC* [1]. It is a virtual platform, meaning that the hardware configuration of the MPSoC is a software artifact built over the actual hardware board (a *PYNQ-Z2* FPGA) which provides the real hardware elements. Such virtual platform can then be configured with different hardware tiles, resulting in different instanciatable images of the Verintec MPSoC. The instance that has been used in the analysed case, consists of three PE tiles (that use *MicroBlaze* (mb) cores as physical processors) that can run, each, a maximum of three applications at the same time (the dark blocks in Figure 1.7). Each tile T has a 128KB portion of combined instruction and data memory (*I/D MEM*) that we will often refer to as *local memory* (LM), and the local memory of each tile cannot be accessed by the other two tiles (to ensure *composability*). The *interconnect* that bestows intra-platform communication is made of *shared memory blocks*; the only means of communication between each couple of processing tiles is in fact a 64KB shared memory tile (the lighter grey blocks in Figure 1.7), bidirectionally accessible by means of *memory mapping* (MMIO). The platform features then three of these shared memories (one for enabling communication between each pair of PEs, forming a full mesh interconnect structure).
More details about the platform and the virtualization concept are given in Appendix A.

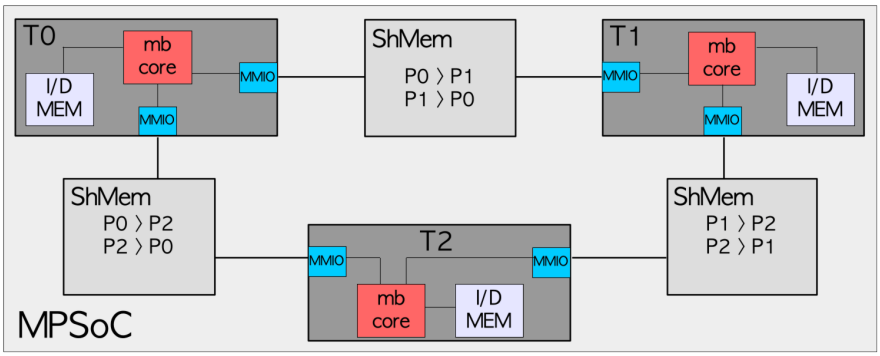


Figure 1.7: A schematic of the used Verintec image

Real-Time Operating System

The virtualization architecture of Verintec is based on the same virtual platform design flow described by *COMPSoC*. In a more general purpose version of the latter, the Real-Time Operating System (RTOS) was formed by a single software entity called *CompOSe* RTOS that wrapped up all of the necessary *inter and intra application scheduling* functionalities [13]. The OS provided by *Verintec*, on the other hand, is formed through the unwrapped and combined action of the `pose` framework and `vkernel` μkernel (which has the same purpose and functionality of the μkernel used in CompSOC, *comik* [17]). The latter is used to compose the firmware of the platform and has mainly the duty of managing the successful scheduling of *virtual execution platforms* (VEPs) (*inter-application scheduling*), while the former is the medium for mapping those VEPs on the μkernel (also as DF applications, like in our case study) and manage the scheduling of the single jobs within them (*intra-application scheduling*) [13]. The `pose` half of the OS is going to be the one to be expanded for the purposes described later in Chapter 2 (the expansion itself is discussed later in Section 3.3).

Intra-tile scheduling

On the Verintec platform, VEPs are scheduled by means of *Time Division Multiplexing* (TDM). We can now identify the VEPs as the single target applications that are to be scheduled and executed on the platform. In such technique, applications have assigned and limited time slots in which they can execute. The totality of the slots forms the *TDM frame*: a sequence of slots that happen cyclically.
In the used platform instance, up to three applications per time can execute (on each tile) within an unlimited number of slots (that have to be anyway defined and that can also have different time lengths among them). In addition to the three applications, a super application, the *System Application* (SA), can be scheduled to execute; the SA can be programmed to have any kind of platform or application management purpose (this will be exploited later to run a support *daemon* on the SA, see Section 3.4). In this scenario, the TDM frame can be initialised with a number N of slots of different duration that can be singularly assigned either to be *idle* or to schedule one of the four (three regular, plus SA) applications. After each allocated slot, is a context switch time to be taken in account (of 4096 cycles) where the state of the VEP is saved; an example of TDM frame is given in Figure 1.8 where the *CS* intervals represent the just described context switch time. This type of scheduling makes the concurrent execution of the applications composable, since each one of them has a defined time space to execute singularly without affecting the others.

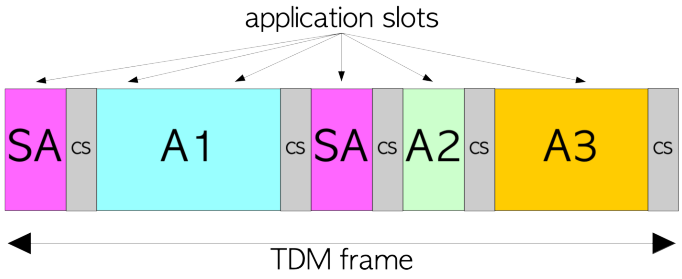


Figure 1.8: A possible TDM scheduling table for a VEP on *Verintec*

Intra-application scheduling

The *pose* library (written in the *C* language), as anticipated, is used for shaping and executing applications in a data-flow model. By doing so the intra-application scheduling is consequentially managed: the wrapped up tasks are in fact schedulable following a certain static scheduling policy. In our case under examination, the task scheduling will follow the *Round Robin* heuristics, where these are scheduled to execute cyclically whenever the necessary conditions (firing rules, availability of tokens to consume and necessary space in the production buffer) are met.
The source code of this entity will be subject to expansion in this study; the process will be described in Section 3.3.2.

1.4 Thesis Structure

The introduction chapter we just went through gave a general overview of the setting, the problem and the case study challenged in this thesis. Following, Chapter 2 will introduce the proposed solution to tackle the presented problem in our case study and a view of the current state of the art concerning similar implementations and scenarios; after this, the rest of the thesis will focus on the practical work for the project. First, in Chapter 3 the design decisions and tools for the implementations will be described; the focus will then move on the latter, where the shape of the solution is given and shown (with also the aid of a visual example). Finally, the evaluation of the implementation is presented and discussed in Chapter 4; here the mathematical time measurement model is presented and then used for real experimentation on the case study. The results from such step are then discussed for the evaluation of the solution. To wrap up, the conclusions over the project challenges and innovations will be given in Chapter 5; the chapter will also give a glance of the possible future works that can see this thesis work as an inspiration or starting point for a new project.

Chapter 2

Methodology

After the investigation of the problem was discussed in introduction (Section 1.2), this very chapter will focus on the proposed solution, with detail on the practical mechanism and the state of the art in the field.

2.1 Proposed Solution

As stated before, the purpose of this thesis project is to research a resource management mechanism according to the P problem. The most obvious solution to a generic case would be to implement a design that takes in account an everlasting worst-case scenario in the system; this approach is the furthest from optimal and would cause many other issues, resource misuse among all. To name one of the others, the scalability of the system would be extremely lowered (adding a new object applications would probably break apart any requirement in time of already mapped applications). A different way from the worst possible case design would be to dynamically change the resources allocated to the applications as resource management systems (RMS) do. Such techniques are known as dynamic reconfigurations (see 2.1.1). In this case, when an application needs more resources because of, say, a runtime change of a requirement, the RMS dynamically allocates more resources to it. Although, dynamic allocation of resources leads to expansion of the VEP on which the app is deployed and the expanded VEP may be mapped (bound) to physical resources of other tiles. Henceforth, since physical resources cannot be "moved", we need a mechanism to migrate the object application and its state (if we cannot reset the execution of it); this can be realized through a *partial application migration* mechanism; this would be the solution to *Q1* (research question 1, 1.2).
In order for the partial migration mechanism to work, the elements to migrate must be put to sleep. The process, within the interested application, should pause parts of it that are only directly communicating with the elements to move (the *Region of Communication*, *RoC*). The reason we want this behaviour, relies in the fact that, during a migration, not pausing the RoC might result in unwanted production or consumption of data relevant to the part of the application to migrate. This would alter the initial state (pre-migration) of the part to migrate, and such state might have been already moved, causing post-migration inconsistencies in the application data. It is because of this characteristic, that we refer to this migration mechanism as *partial-stalling* (detail is given in Section 2.1.2). To give a generic example, as shown on 2.1, we have a *dummy* application in a DF model. Task *yellow* is to be migrated (and its RoC is highlighted), the rest of the tasks can keep on firing while the RoC undergoes the migration as long as conditions on incoming and outgoing tokens are met. We must stress here, that with *pausing* an element (a task or whatever other task in the RoC), in this project, we mean that we enforce that element to be *unschedulable*; the element will execute until it is finished (wait for *quiescence*) once the pausing is issued and then not execute newly, since unscheduled. This *quiescence time*, shown in Figure 2.3, does not account for the migration time overhead, since it does not affect the application execution. In addition,

from now on, when talking about migration on our case study, we define it as *the change of the binding of the actors and respective FIFOs to the allocated resources* being the target application modelled in data-flow.
Implementing such mechanism in a predictable environment, allows the former to be predictable as well; this will lead to the solution to *Q2* (research question 2, 1.2). Being the Verintec MPSoC also composable, such migration mechanism shall not affect other applications that are executing normally.

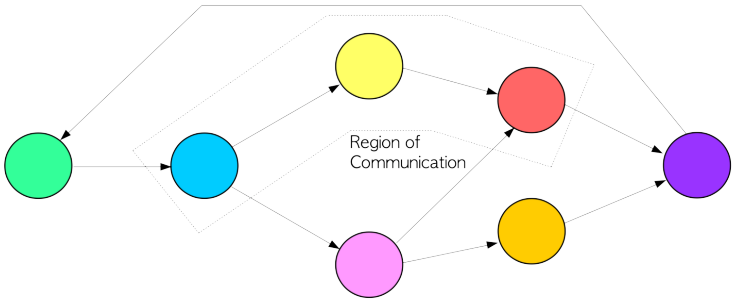


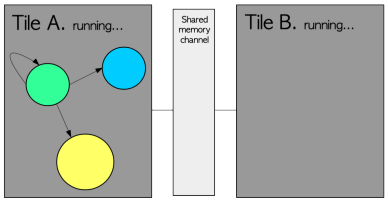
Figure 2.1: An example DF graph with the highlighted *RoC* of the *yellow* task

2.1.1 Dynamic Reconfiguration

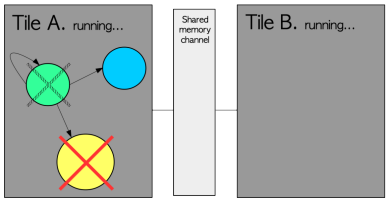
Because of the physical limits of embedded systems previously introduced, it became of primary importance to avoid misuse of the available hardware. The performance improvement comes then from the dynamic management of the available hardware: runtime resource management techniques that not only allow to satisfy software with dynamic requirements, but also aim to reduce performance degradation over time by redistributing workload for certain processing elements.
A dynamic reconfiguration of an embedded application is a mechanism where certain subelements are re-configured during runtime in order to use a platform resource differently or even use a different one. Such reconfigurations can have, for example, the objective of minimizing the use of an overheated processing unit (*Dynamic Voltage Frequency Scaling* or *Dynamic Remapping/Loading*), remove ghost memory allocations (*Dynamic Memory Desegmentation*), reserve more processing power for dynamic priority jobs (*Dynamic Task Migration*) or introducing new functionalities in the system (*Dynamic Loading*).
As aforementioned, in the presented study we will adopt a *variant* of the task migration technique; an introduction to the latter follows.

2.1.2 Partial-Stalling Task Migration

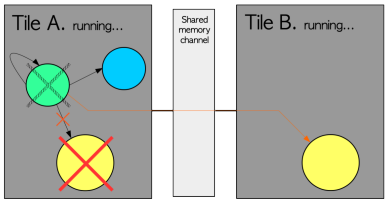
First of all, an actor (a stateless task) is a portion of an application that can be wrapped up as a single atomic unit of its own; for instance, in a *face recognition* application, one of the sub parts of the application is an *edge detection* filter: the edge detection will be a task of its own within the face recognition application. In a generic task migration mechanism, the execution of the task is moved from a processing element to another (for example, the *yellow* task in Figure 2.2a is moved to the second available tile as it can be seen in 2.2d). When the task is not independent and communicates directly with one or more fellow tasks from the same application (there is data dependency), these latter need also to be put momentarily to sleep (suspended) for the duration of the migration process, to allow the RMS to move the state (in our case the buffers that store the tokens and any necessary counter and pointer).
In a partial-stalling task migration, the total stalling of execution of an application is avoided. This is done by limiting the sleep phase to only the portion of the application interested by the reconfiguration (the previously mentioned *RoC*); namely, *the task to migrate and the directly communicating (or adjacent) ones*, as visible on Figure 2.2b and Figure 2.1. Also, we assume that



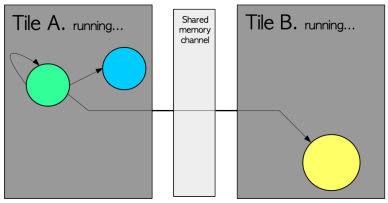
(a) An application divided in tasks (the colored circles) is running on one PE of an example MPSoC where two Tiles communicate through a shared channel (a shared memory space)



(b) *Yellow* is paused and so is *Green* (because it directly communicates with the former) to allow the migration of the first while *Blue* keeps executing (if conditions to fire are met)



(c) *Yellow*'s execution is now moved to *Tile B* and the communication channel with *Green* is remapped on the shared channel



(d) The partial-stalling task migration process is complete and the application is now distributed on both the PEs

Figure 2.2: An high level example of a partial-stalling task migration mechanism

the buffers set at the borderline of the *RoC* (therefore the buffers on the edges of the adjacent tasks that do not connects such task with the task to migrate) are big enough to not become full during a migration due to the tasks that keep on executing.
Figure 2.3 depicts the task flow of the shown example and the main purpose of the partial-stalling approach: the *blue* task, not part of the RoC of *yellow*, keeps executing during the migration window.

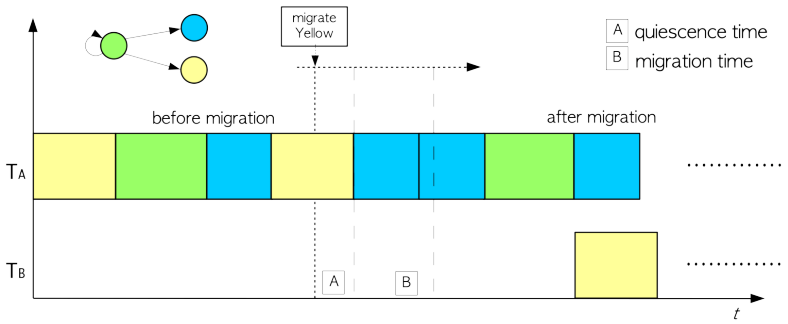


Figure 2.3: Gantt chart showing a mock task execution flow during the partial stalling task migration from the example of Figure 2.2

2.1.3 Project Milestones

The solution the project wants to give, is a software solution. In our case, it will happen by means of a dynamic reconfiguration technique that will not affect the whole application execution.
Of course, when taking in account the dynamic reconfiguration method to overcome the main problem, more lower level problems pop up; these are related to the initial state of our components. The current platform OS, specifically the initial state of the intra-application scheduling library (`pose`, seen in Section 1.3.2), does not support task migration, therefore it is still not possible

to achieve the task migration with the current state of the tools. Moreover, we want our task migration to not stall the application entirely during migration time. However, there is a way to support the partial stalling task migration; to achieve this `pose` must be updated and expanded. In the end, by implementing our solution, we want to be able to timely estimate and evaluate our implementation. Also this step can be supported by the predictable platform-application case study.
The main objectives of this thesis are therefore:

- Expansion of the OS in order to support partial-stalling task migration on a data-flow modelled application.
- Achieve a working implementation on the case study.
- Craft the design time timing estimation model.
- Experiment the timing model on the case study implementation in different configurations.

By completing these, we will have enough material and data to answer *Q1* and *Q2* and therefore offer a solution for *P*.

2.2 Related Work

Before addressing our study directly, a view of the related state of the art is given in this section. For a more logical information flow, the consulted literature will be organized in two paragraphs: in the first one, studies involving embedded platforms resource management techniques in general will be introduced and in the second one, the focus will move on studies that specifically make use of runtime task migration.

Literature Studies that involve General Resource Management Mechanisms

A. Shabbir et al. [21] highlight the limits of centralised resource management in MPSoCs by presenting a distributed resource management system that improves the the scalability of the platform, e.g. the MPSoC handles an increase in the number of applications better with a distributed resource manager. The applications that are run are in the data-flow MoC, nonetheless only dynamic task mapping is implemented and task migration is not supported.
O. Arnold et al. [4] propose a runtime management unit for heterogeneous MPSoCs called *CoreManager*. Such manager applies dynamic task scheduling among the PEs by using a scheduling-specific instruction set purposely developed. Through the designed instruction set, a better energy consumption and memory usage is achieved, however, nothing is proposed to improve the task communication within applications.
In S. Sina et al. [22] an experimentation of application dynamic loading over an MPSoC in time-critical systems is presented. At first an overview of the hardware and software is given, then the worst-case computation model is introduced. The goal of the study is to demonstrate that applications the dynamic loading of applications on a composable and predictable platform can be also designed in a composable and predictable fashion. In order to achieve this, the study presents a computation model for the worst-case loading time. Even thought the benchmarking from the *coarse grained* point of view is provided, the study does not dive deeply in the detail of the execution cycles of the loading mechanism, giving just rough estimates.
M. Mandelli et al. [15] present a resource management heuristic for MPSoCs with a large number of PEs. In this study, the tile management is distributed by organizing these into clusters, managing these locally by means of a master PE. Through this mechanism, the task allocation overhead for dynamically entering applications is minimised since the number of PE to parse for availability is reduced. However, runtime minimisation of inter-application communication is not

optimised. The clusters are, in fact, non reconfigurable and the task are not migratable, so tasks that are forced to be *distant* if mapped on PEs on different clusters. Such issue is solved in the follow-up study by G. Castilhos et al. [7] where runtime reorganisation of clusters is implemented together with task migration. The study shows how investing processing time to migrate tasks has a positive impact on the overall application execution (inter-application communication is reduced).

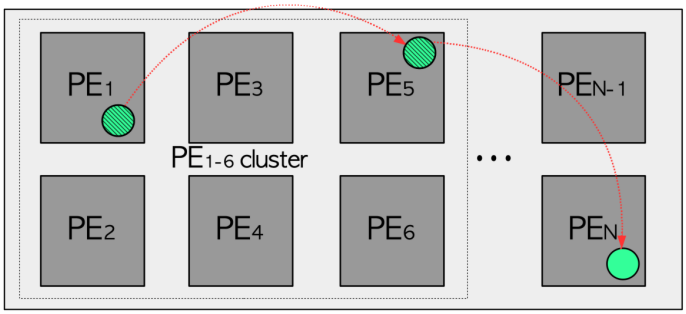


Figure 2.4: In a large MPSoC, a task migration might have the secondary duty of freeing space for, like in the depicted example, a cluster of processing units that are to be assigned to other applications

Even though the last presented work does implement dynamic reconfiguration with the goal of having application that better use the available resources, the general limits of using runtime reclustering and task migration are not directly addressed, especially in terms of time overhead. In that specific case, the stalling time to reconfigure is shown to be worth the investment in such environment with multiple PE clusters: when a reclustering happens, applications are moved through task migration to non involved clusters. However, such experimentation does not take in account small MPSoCs with minimal number of PEs or overloaded MPSoCs with constrained resources, where any dynamic operation would involve the whole tile-application environment an there would be no space for concurrent executions.

Literature Studies that involve Dynamic Task Migration

P. Tendulkar et al. [23] investigate on resource management with task migration on a composable and predictable MPSoC, of a SDF modelled application. In this work, they introduce a system-level resource manager running as an additional application that does not affect the other applications in time thanks to the composability of the platform. However, The implemented task migration makes use of strict migration points where the target application is entirely paused, due to the fact that just the execution of the task is migrated (meaning that the only migration points occur when the state of the task is *empty*, e.g. no tokens on incoming and outgoing edges). In addition, investigation of the time overhead introduced by the reconfiguration is not detailed and no models are given. G.M. Almeida et al. [16] focus mainly on a message passing framework for MPSoC in their work. Task migration is part of the study, being the previously mentioned framework ideal to manage dynamic reconfiguration scenarios that involve more than one PE. Task migration is achieved through the establishment of *migration points*, namely time windows in the PE task execution schedule where there is no data *on fly* due to communication. During the migration points the application can be stalled and the context of the task to move migrated. However, the full application stalling is what we aim to avoid in our study.
In the follow-up work, still from G.M. Almeida et al. [2], the focus moves on the task migration previously introduced. This study shows that the developed task migration mechanism introduces overhead, but introduces also an improvement in performance because of the achievable

load balancing. Nonetheless, target applications are still fully paused during a migration and no estimation of the overhead is given.
F. Fu et al. [9] present a task migration mechanism that focuses on preserving the task execution time by introducing a low overhead on migration. In this research, the MPSoC is formed by one master PE and a number of slave PE that execute the tasks. The migration is managed through the use of a *MPI* purposely designed to initialise, manage and finalise the migration. Such approach is responsible for the reduction of the overhead compared to other works. We will use a similar mechanism in our work, but, again, the used MPSoC here does not move the focus of the work on resource management as we would like to do.
V. Nollet et al. [19] propose a resource management method for a reconfigurable hardware NoC; within the hardware management heuristic, runtime task migration is implemented in order to free space for the reconfigurable tiles to be moved when necessary. A benchmarking method is also provided to evaluate the effectiveness of the migration; this provides certain parameters such as: *reaction time* (time between the issue and the start of migration), *freeze time* (time during which the task is suspended) and *residual dependencies* (the undesired dependencies left from a task on the source tile of the migration). However, such migration is implemented only to avoid a deadlock in a tile reconfiguration mechanism of a large MPSoC and, once again, resource usage management to solve resource constraints is not the main focus of the study.
E. Briao et al. [6] present a dynamic task allocation mechanism on a simulated NoC, based on *bin-packing* (assignment based on processor utilisation) and *linear clusterization* (assignment that takes into account clusters of highly communicating tasks to minimize inter-process communication) algorithms. In order to allow dynamic load balancing through the mentioned algorithms, a task migration mechanism is implemented. However, the mechanism is achieved through a total idling of the destination processor since avoiding a big migration overhead and estimating it, is not a one of the goals of the study.

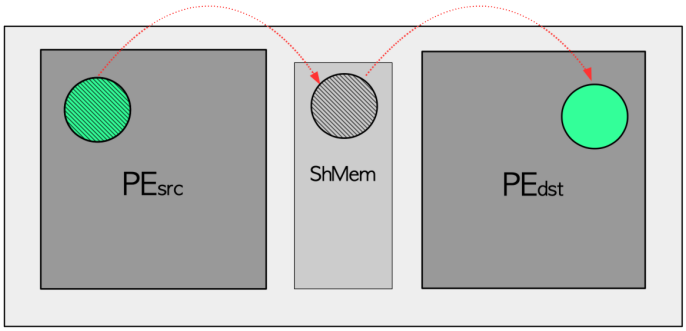


Figure 2.5: In several task migration implementations, the context or state of the migrated task is intermediately saved on a shared memory space where both source and destination PEs have access

Again V. Nollet et al. [18] show a novel migration mechanism where the operation is initiated by using the processor debug registers (commonly present in modern μprocessors) as means to communicate and initiate a migration. Such method has minimal overhead, but is, however, limited by the number of available registers.
S. Bertozzi et al. [5] present a lightweight task migration mechanism over a *NUMA* (Non Uniform Memory Access, which means that access times for local and shared memories differ) MPSoC in which the migration is implemented with two fundamental elements. The first is a *MPI* (Message Passing Interface) that allows the *master* and *slave* processor to communicate, and therefore issue and manage the migration by only sharing a *bit*, reducing, by means of it, the migration overhead. The second is the processing *context* saving on the shared memory; the task state is in fact saved on the shared memory so that a safe resuming of the application can happen on the migration

destination tile. Both of the elements, are used and moved by a middleware, a *deamon*, present on each tile. Such study is relevant to ours: it happens on a MPSoC with really similar structure to ours (i.e. PEs that communicate through shared memory spaces see Figures 1.7). Moreover the model they used for the message passing and the state (context) saving on the shared memory is convenient to our case. Although, the resource budget again does not match our case study and no benchmarking is given.
B. Pourmohseni et al. [20] propose a predictable task migration mechanism on a MPSoC, in which possible migration outcome mappings of the target application are pre-explored at design time through *Design Space Exploration*. Also in this study the migration is achieved by full stall of the object application and, although they do provide a model to compute beforehand upper bound of the migration, the article does not address the overall weight of full stalling on the global migration time overhead.
L. Gantel et al. [10] present a task migration mechanism over a reconfigurable MPSoC in which the application tasks are already initialised on every PE even if non active (*task duplication*). This means that the necessary source code is already loaded on each tile and does not need to be moved at each migration. A migration is therefore triggered for a task on a source tile; the task is deactivated, the context is saved on the shared memory (just like in [5]) and moved to the destination tile; the migrated task is activated on the destination tile, where the source code was already loaded. The relevancy of this study relies in the fact that the necessary structures to run a task are already loaded on each tile. Dynamic loading of instruction data is, in fact, not relevant to our study and also expensive in terms of additional execution time; for this reason we will also go for pre-loaded task code in our implementation. These relevant studies, nevertheless, focus on offering a general solution to a general cases while and, by doing so, the used MPSoCs are relatively large in terms of resources, while we are focused on scenarios where there are constraints on resources. In such bigger MPSoCs a task migration might have the power moving an application completely in another region of the platform, freeing, for instance, a cluster of six PEs to make space for a new application, as shown in Figure 2.4. Of course, this is not a case for a small MPSoC.

Chapter 3

Design and Implementation of the Solution

Investing a portion of execution time to migrate tasks to other tiles, in a parallel fashion respect to the application execution, is our aimed solution. The initial RTOS state does not support the functionalities of *pausing* tasks (e.g. unschedule and wait for quiescence) and *moving* (e.g. rebind) the involved FIFO buffers among the tiles. Because of this, a phase of software expansion to obtain the necessary framework for the objective has to take place.
After a phase of preparation, the morphology of the available tools was took into consideration to develop a migration mechanism. In order to achieve a *partial-stalling task migration* following the guideline introduced in 2.1.2 and, at the same time, preserve the *predictability* of the system, the most logical approach is to update and expand the `pose` library (which is already a framework designed for modelling DF applications and manage the task scheduling within it, but not designed for task migration whatsoever). Together with the thesis supervisors, we decided to approach the obstacle by developing first debug mediums to speed up any verification of the designed and implemented software expansions.
In this chapter, we will firstly see two necessary tools developed for coding and debug purposes: the *test application* and the *POSIX multiprocess instance*. Secondly, the *pose* expansion, that we name the *Migration API* and the implementation of the *migration daemon* who uses the API to make the migration mechanism possible. Then, the dynamics of the implemented partial stalling task migration will be shown in detail with a focus on the steps taken by the *migration daemon* to move the data; the addition of a visual example will be also given for ease of comprehension.

3.1 Test Application

As introduced in 1.1.2, the adopted *MoC* is the *data-flow* model: the target application will be therefore subdivided into actors. This software morphology does not only make the dynamic management operations over the application predictable (thanks to the data-flow MoC, as explained in 1.1.2), but it is also ideal for designing a *partial-stalling task migration* kind of reconfiguration (see 2.1.2). Having actors that are scheduled singularly makes possible to identify, for each one of them, *regions of communication* (the portion of the application defined by a certain task plus all of the other ones that directly have incoming or outgoing communication with it).
During design and implementation of the task migration solution, the need for a test application of simple structure and trivial operations was really important for debug purposes. For such phase, a test application was quickly implemented and used. The application, modelled in the data-flow MoC and consisting of four actors, follows in Figure 3.1.
The test application generates a random floating point number in the *RND* actor. The same number is squared in *SQR* and doubled in *DBL*. At last, the two intermediate outputs of SQR and

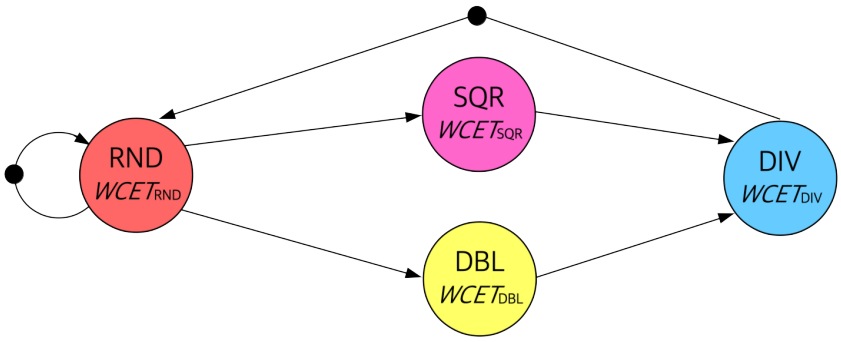
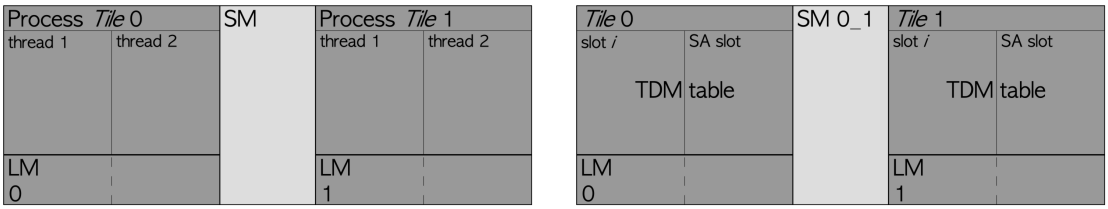


Figure 3.1: The data-flow model of the simplistic test application

DBL are divided by the *DIV* actor. Basically, if random number *num* is generated, the output will be half of it ($num^2/2num = num/2$). The job is trivial, but useful to determine swiftly whether the reconfiguration process had been successful while leaving the application functionality unaltered. The application starts thanks to two initial tokens put into the consumption buffers of *RND*; the self edge on this latter has no algorithmic purpose, it is there for debug purposes (e.g. testing the migration of a task with a self edge).
A dataflow model of the described application can be observed in figure 3.1.

3.2 Multiprocess Instance

The necessity of having a design platform as an environment to test the effectiveness and consistency of every single step of the expansion, led to the development of the *Multiprocess Instance* on a *Linux* machine. The objective of the multiprocess instance is to recreate the behaviour that is expected on the *Verintec* MPSoC in order to make the test implementation derived from the design, *portable* to the embedded platform. On this latter, processors are unable to access each other local memory to respect the composability principle; they are henceforth allowed to communicate just by using the NoC made of a full mesh of bidirectional shared memory spaces.
To emulate the processors, two different `POSIX` processes are initiated (in brief, two distinguished `main.c` files are run). One of these two cannot access the memory allocation reserved to the other and vice versa, recreating the local memory enclosure we see on the embedded processors (LM, Local Memory on Figure 3.2a). To enable the communication, moreover, a `POSIX` shared memory for inter process communication (`IPC`) instance is created, accessible by both processes through an identifier (SM, Shared Memory on Figure 3.2a). The memory behaviour is now emulated.
When considering the platform, the *firmware*, supported by the `vkernel` μkernel library (same as `comik` on COMPSoC [17]), makes the simultaneous run of applications possible through TDM scheduling. This will not be the case on the multiprocess instance since the library is not portable to a Linux machine; the approach to simultaneously run applications follows. In our study case we have two applications: the test DF application and the *migration daemon*. To run these simultaneously on the multiprocess instance, two threads (through `pthreads`) are initiated on each process (these emulate a TDM slot, see Figure 3.2a, even though they do not come after each other sequentially, but seem to follow a *round robin* scheduling). The first thread will be responsible for the execution of the test application, the second of the migration daemon.
The `POSIX` multiprocess instance is now ready; an high level schematic is given in Figure 3.2a. Such tool is built considering the parallel structure present on *Veritnec* which is represented on Figure 3.2b; once the initial implementation is ready, it will be portable from one to the other.



(a) POSIX

(b) Verintec

Figure 3.2: Structure comparison of the *POSIX* multiprocess instance and the *Verintec* platform

3.3 RTOS Library `pose` Expansion: the Migration API

Modelling of data-flow applications and management of inter-scheduling and execution is the main purpose of `pose`. Enabling partial-stalling task migrations, however, is not yet; the expansion of the framework offered by the library becomes then a necessary step. In order for the partial-stalling task migration to work, we need the possibility to *suspend* the execution of the involved tasks within the *RoC*, to be sure that no data *on flight* would threaten the consistency of the context data to be migrated. As mentioned previously in the dissertation, the suspension happens by making the task unschedulable and letting it execute until completion if it was under execution once the suspension was issued. In fact, to achieve a clean migration, we must wait until the involved elements are in a quiescent state; this is enforced by not allowing tasks of the RoC to be scheduled again.
The designed approach for the migration requires the application to be fully mapped on every PE by means of *task duplication*, this will be discussed before taking a look at the migration API.

3.3.1 Task Duplication

The pre-concept for the designed software add-ons to work according to design, is the use of *Task Duplication* at system startup-time. Task duplication is a technique where every necessary actor (and FIFO) is loaded on each tile as if the application is to be executed exclusively on that PE. This is already possible with the current state of `pose`, since loading of tasks and FIFOs is a primary functionality of the library.
With this technique, a bigger memory footprint is left on each local memory (and shared memory). However, a lot of time is saved in a migration process, where the only dominant time consuming step remaining, is to move the content of the buffers (since both tasks and FIFO buffers are already there and no dynamic loading of them will be needed). The same task duplication technique is used in [10].
An example of task duplication on our test application can be seen in Figure 3.3: all the necessary actor and FIFO instances are loaded at startup-time.

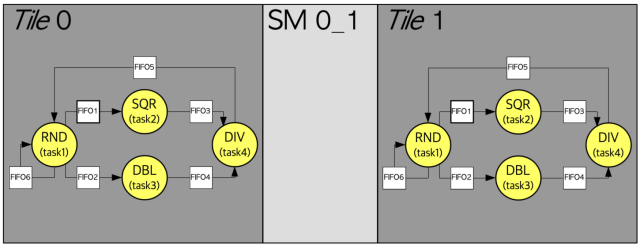


Figure 3.3: Task duplication of the test application on two tiles

3.3.2 Migration API

The `pose` library is used to model DF applications by creating a hierarchy of instances of three different `structs`, as shown on Figure 3.4. To allow *suspension/resumption*, *deactivation/activation* and *state move* functionalities, expansions have been operated in two different areas of the `pose` library: the control block structures and the control block manipulation functions within the source `C` code.

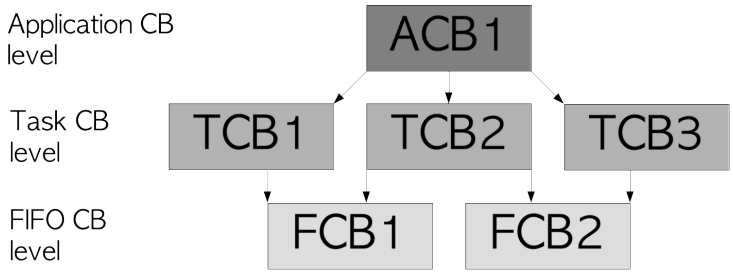
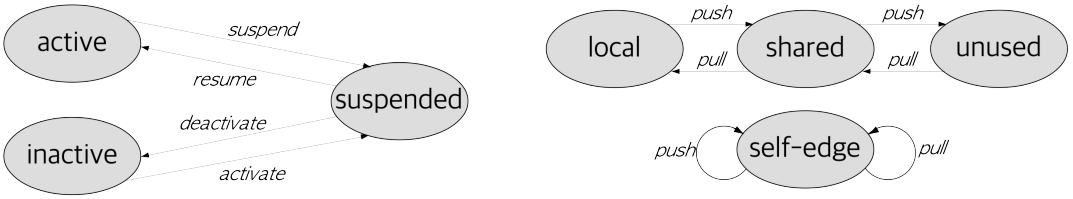


Figure 3.4: The `pose` hierarchy of structures

- Structures: Relevant data, flags and pointers for each entity of the data-flow model of the application are wrapped in `struct` instances called *control blocks* (CBs). One of such entities is the application itself (`ACB`, Application Control Block), the others are the tasks and the FIFO channels within it: these last two are subject to update.
  In the `TCB` (Task Control Block), two flags are added: `active` and `suspended`. The activeness flag signals whether the task is being scheduled on the tile from a global point of view, in other words, if it is active on the specific tile or not. The suspension flag has also the purpose of making the task unschedulable, but in this case the meaning is that the task is unschedulable due to ongoing dynamic reconfiguration (like the task migration). Once the reconfiguration is finished, the task is unsuspended and the prosecution of operation within the tile will depend only on the more global activeness flag. The presence of the two flags ensures that *RoC* tasks that are supposed to be unactive on one of the two tiles, do not get accidentally activated when a resumption happens. To sum up, the `active` flag has a permanent effect (task active or not on a tile), the `suspended` flag, instead, has a temporary effect, it allows to unschedule, and therefore put to sleep, the tasks involved into a migration to avoid *on flight* data to damage the veracity of the state of the application to migrate.
  In the `FCB` (FIFO Control Block), the `locality` flag is added to signal the location of a FIFO on a tile (self-edge, local, shared, unused). During a migration scenario, the locality flag allows the migration daemon to know how to update the FIFO channel. Such flag will in fact signal if the channel is a self edge, a communicating channel between two tasks on same tile (*local*) or on different ones (*shared*), or even an *unused* channel (between two *inactive* tasks on that tile). This information is necessary in order to update the FIFO channels correctly and to their most optimal position, starting from their current nature. Through this mechanism, no *residual dependencies* are left behind on the migration source tile (as in [19]).
- Functions: The manipulation of the control blocks to support a migration is made possible by the addition of several functions that operate at `TCB` and `FCB` level.
  At task level, functions to *suspend/resume* a task τ and its adjacent communicating task τ_{adj_i} and functions to *activate/deactivate* a task τ are added.
  At FIFO level, a function to be used by the source to *push* the FIFO data and state is added together with a function to *pull* the backed up information, to be used by the destination.

With the Migration API the tasks and the intermediate FIFO attributed describing the state of

them are then updated. The effect of the migration functions is shown in Figure 3.5 where two state machines are represented. The one in Figure 3.5a represents how the state of the task changes during a migration: on the source tile task τ will start as active and end as inactive following a *pause* and a *deactivate*; on the destination tile τ will go from inactive to active in the opposite fashion, with an *activate* and then a *resume*. In case the task is part of the τ_{adj_i}, then it will just go from active to suspended during the migration and again to active. The other one, on Figure 3.5b represents how the locality of the FIFOs involved in a migration on a tile changes: a self-edge preserves its nature regardless of which action is happening; FIFOs on normal edges can go from local, passing by shared to unused, with the direction of the change depending if, on that tile, a *push* or a *pull* is happening for that FIFO.
The expansion details and the API functionality are explained in detail on Appendix C. All of these adaptations and newly added elements are used to craft the *migration daemon* that will be introduced in Section 3.4.



(a) Task attribute state machine

(b) FIFO locality state machine

Figure 3.5: Migration state machines describing the update of Tasks and FIFOs attributes with the Migration API

Within a partial-stalling task migration, the task to migrate is suspended along with the directly communicating tasks (the *region of communication*) since the FIFO channels between τ and τ_{adj_i} are going to be moved and must be therefore quiescent at migration time. In the case of our test DF application on Figure 3.6, migrating the *SQR* actor (*orange*, suspended and to be migrated) means that in the process also *RND* and *DIV* are suspended (*yellow*, suspended), while leaving *DBL* non-stalled (*cyan*, active). To achieve a predictable and composable migration, we need a background entity that will make the migration process transparent from the point of view of the target data-flow application. Such entity will be a part of the platform System Application (SA), that we will call *migration daemon*. This is going to be described in the next section.

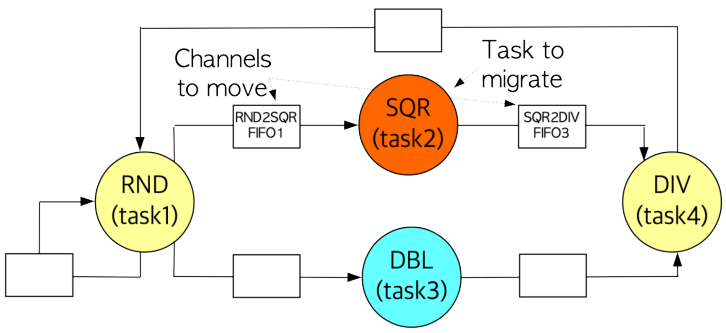
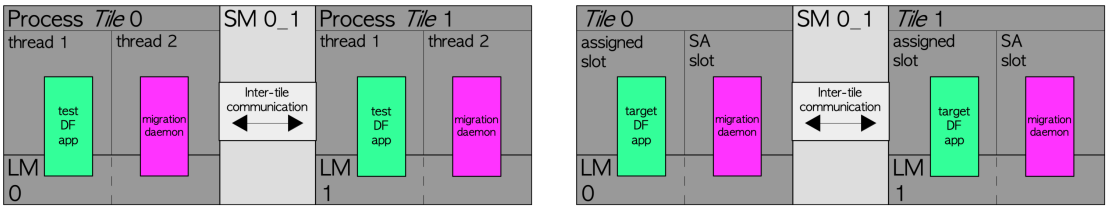


Figure 3.6: With the migration API expansions, the suspension of portions of the application is now possible

3.4 Migration Daemon

To make the task migration parallel and transparent from the test application scope (or any other application that will be executed on the platform), the implementation of a middleware was a necessary step: the *migration daemon*.



(a) POSIX

(b) Verintec

Figure 3.7: Application mapping from the *POSIX* multiprocess to the *Verintec* platform

Such application makes sure that the application(s) keeps executing while a migration is ongoing (in the mapping example on Figure 3.7b we can observe that the *daemon* executes in its own System Application slot as an independent application). The purpose of the daemon (on the model of the one implemented by [5]) is to enable the message passing between the two involved PEs (achieving in this way migration initialisation and finalisation), to save and move the communication context (therefore move all the involved FIFO channel data, namely the state of the tasks) and, of course, to direct the necessary activation/deactivation and suspension/resumption of tasks by using the migration API expansion, described previously in Section 3.3.2. Since the daemon is meant to alter the CBs of the application, to preserve composability, we must assemble such application as a part of the *system application* (a *super* application that will be present on the Verintec platform). Also, if we were to assemble it on the same VEP used by the target application, implementing a partial stalling mechanism would not be possible because only one application per time would be scheduled for execution.
The SA is present on each one of the tiles, meaning that there will be a *daemon* instance on each PE to manage upcoming migrations. Of course a tile might behave as *source* or *destination* for a migration; the first step for the *middleware* introduced in this section is therefore to determine what will be the behaviour for the tile it runs on, during a migration. Such step is shown in Listing 3.4 where pseudocode resembling the actual C lang code is given.

```
/* The daemon checks whether to behave as source
 * or destination by verifying the state of the
 * task to migrate on which the daemon instance is running
 */

void migration_daemon(int t_id)
{
    t = get_task(t_id);

    if(is_active(t))
    {
        task_migration_source(t);
    }
    else
    {
        task_migration_destination(t);
    }
}
```

Listing 3.1: Pseudocode for the behaviour determination of the *migration daemon*

Once the behaviour for the *daemon* is determined, the migration can start. In Listing 3.4 we can see the pseudocode steps that make up the *source* behaviour, while in Listing 3.4 the ones that make up the *destination* behaviour. The main concern of the source is to pause the *RoC* and deactivate the task to migrate to move its context out, while the main concern for the destination is to move the context of the task to migrate in by also pausing the *RoC* and activating the former. With a closer look we can notice that the two present some synchronization steps that are dependent on each other (in the form of `while` loops). Source and destination must in fact communicate in order to complete the migration process successfully; this happens by exchanging, on each step, a byte of information to grant the passage of the barriers (in a similar way as [9] and [5] implement their MPIs).

```
/* In the source behaviour, the task to migrate
 * is suspended and so are the adjacent tasks
 * that are active. The context of the task is
 * pushed and the latter deactivated. When
 * the destination is done, the RoC tasks are
 * resumed to complete.
 */

void task_migration_source(task t)
{
    os_migration_suspend_task(t);

    os_migration_suspend_adjacent_tasks(t);

    while(!migration_grant_push)
    {
        /* waiting for destination tile to be ready for context move phase */
    }

    os_migration_push_task_state(t);

    migration_grant_pull();

    os_migration_deactivate_task(t);

    while(!migration_pull_done)
    {
        /* waiting for destination tile to finish pulling */
    }

    resume();

    os_migration_resume_adjacent_tasks(t);
}
```

Listing 3.2: Pseudocode for the source behaviour of the *migration daemon*

```
/* In the destination behaviour, the
 * adjacent tasks that are active, are suspended.
 * the migrated task is activated, the context is
 * pulled and the RoC tasks are resumed in the end
 */

void task_migration_destination(task t)
{
    os_migration_suspend_adjacent_tasks(t);

    migration_grant_push();

    os_migration_activate_task(t);

    while(!migration_grant_pull)
    {
        /* waiting for source tile to finish pushing */
```

```
    }

    os_migration_pull_task_state(t);

    migration_pull_done();

    while(!resume)
    {
        /* waiting for source tile to synch */
    }

    os_migration_resume_task(t);

    os_migration_resume_adjacent_tasks(t);
}
```

Listing 3.3: Pseudocode for the destination behaviour of the *migration daemon*

The operations and the interaction between source and destination create a fixed sequence of events that characterize every possible migration. Such sequence of function calls is summarised in Figure 3.8. All details of this migration sequence are explained in the following section (3.5) and shown in an example later on (in Section 3.6).

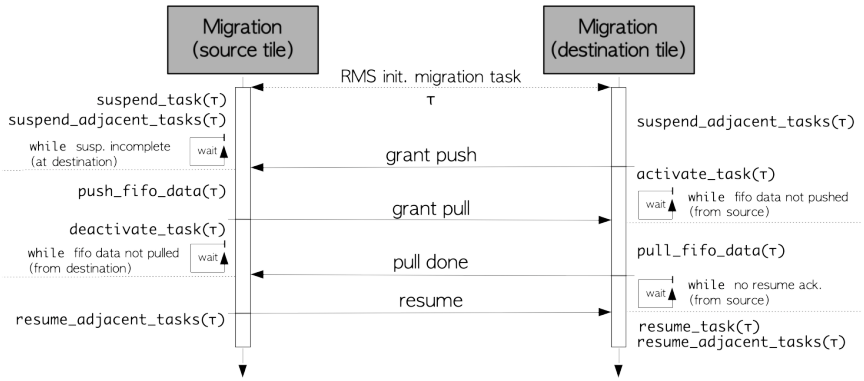


Figure 3.8: Function calls in the migration sequence. The communication steps that occur in the migration sequence are the horizontal arrays

3.5 Migration Sequence

The migration middleware introduced in Section 3.4, was then implemented to use the API functions (discussed in Section 3.3.2). More precisely, the daemon has been equipped with two different behaviours, one to deal with being a source of migration and one to deal with being a destination. In this way, providing the daemon to each tile enables *bidirectional* possibilities for task migration. The migration sequence resulting from the combined action of the source PE migration daemon and the destination one, has been implemented to ensure the consistency of the data left behind on source, the data to be moved among processors and the data at found at destination. The sequence will be seen in detail in this section.
Once a migration for a task τ is issued, a series of fixed steps, of both communication and manipulation, happen. The designed flow of events is shown in Figure 3.9; in the image, the grey blocks symbolise steps that have a variable execution time depending on the application that is undergoing reconfiguration and on the exact state of it (explaining the need for synchronization points between source and destination processor).
A more detailed sequence diagram of the `pose` function calls is given on Figure 3.8. The explanation of the necessary steps for the migration to happen follow hereby. An external entity, that we will call *Resource Management System*, monitors the resource usage for the platform. At a

given point in time t_0 the *RMS* triggers a migration for task τ: the *daemon* determines whether its execution tile is to behave as source or as destination, phase I begins for both tiles.

Source Phase I: τ and *RoC* Suspension (Figure 3.9)

In case of *source* behaviour, τ is suspended alongside all the other adjacent tasks (in the RoC) active on the tile and then enters a loop where it waits for the destination to complete the parallel duties.

Destination Phase I: *RoC* Suspension (Figure 3.10), Push Grant and Activation of τ

On the other side, in case of *destination* behaviour, the *daemon* suspends the τ adjacent tasks active here and communicates with the *daemon* on the source to allow the latter to continue to its second phase (by means of the *push grant* signal). At the same time, the destination uses the time the source uses now for pushing, to already put τ in an active state; after this activation, it enters a loop to wait for the source to finish pushing.

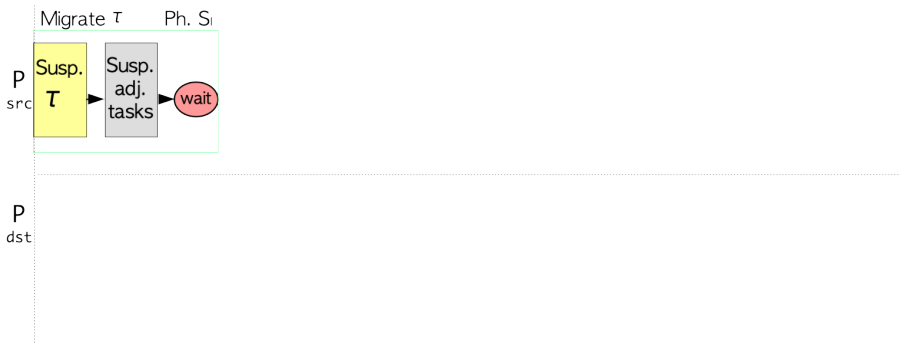


Figure 3.9: Migration actions flow diagram: Source Phase I

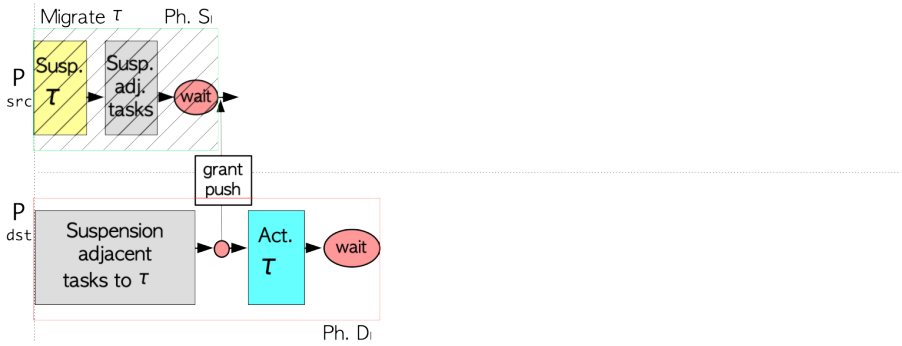


Figure 3.10: Migration actions flow diagram: Dest. Phase I

Source Phase II: Push of FIFO Data, Pull Grant and Deactivation of τ (Figure 3.11)

In its phase II, the source pushes the state data of the producing and consuming FIFOs of τ (tokens and counters are moved to the most optimal position depending on the locality of the

FIFO, to shared if the FIFO is gonna bestow communication between tasks on different tiles). The state push mechanism is similar to the one seen in the studies by [5] and [10]. When the source *daemon* is done pushing, it communicates with the *daemon* on the destination to allow the latter to exit the wait and continue to its phase II (with the *pull grant* signal). After this, as also the destination tile did, the source tile uses the time the destination uses for pulling to already put τ in an unactive state; after the deactivation, the processor enters a loop to wait for the destination to finish pulling and therefore synchronize.

Destination Phase II: Pull of FIFO Data and Pull Done (Figure 3.12)

In its second phase, the destination pulls the state data of the producing and consuming FIFOs of τ that have been previously pushed by the source (tokens and counters are moved to the most optimal position depending on the locality of the FIFO, either from shared to local or leaving them shared if the FIFO is gonna bestow communication between tasks on different tiles). The state pull is also, consequently, a mechanism similar to the one seen in in by [5] and [10]. Once the destination *daemon* is done pulling, it communicates with the *daemon* on the source to allow the latter to exit the wait and continue (with the *pull done* signal).

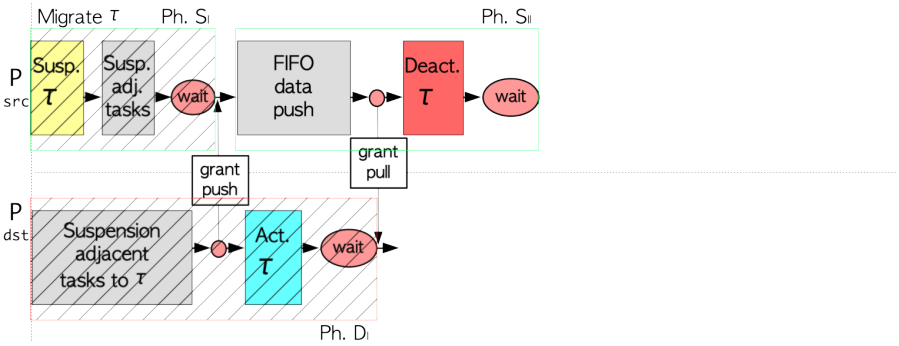


Figure 3.11: Migration actions flow diagram: Source Phase II

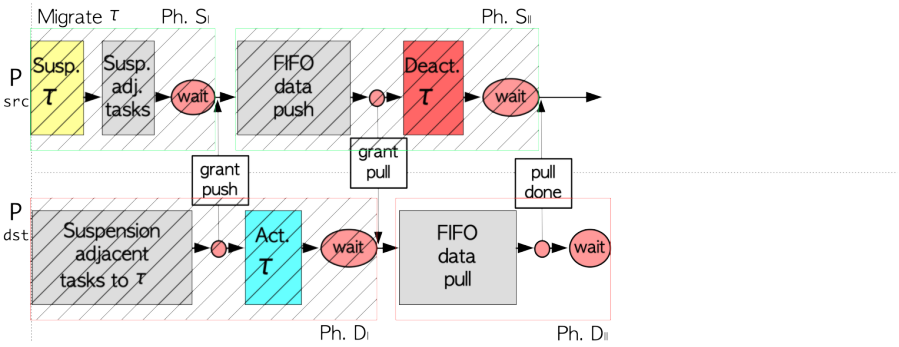


Figure 3.12: Migration actions flow diagram: Dest. Phase II

Source Phase III: Acknowledgement and *RoC* Resumption (Figure 3.13)

Once the *pull done* signal is received, acknowledgement is given through the *resume* signal (which allows the destination tile to enter its last phase). After this, the source tile completes phase III by resuming the previously suspended *RoC* tasks.

Destination Phase III: τ and *RoC* Resumption (Figure 3.14)

After the destination tile completes the pull, it enters a loop to wait for the source to acknowledge and allow it to exit the wait. After the acknowledgement happens (through the *resume* signal), the destination tile resumes τ and the previously suspended τ adjacent tasks. The migration can be considered complete after this step.

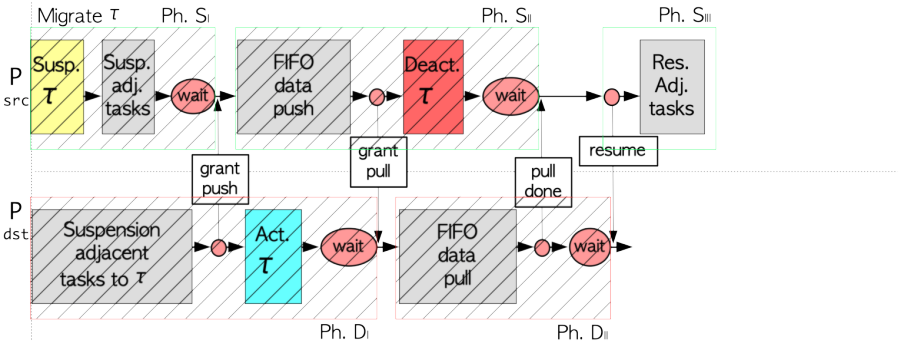


Figure 3.13: Migration actions flow diagram: Source Phase III

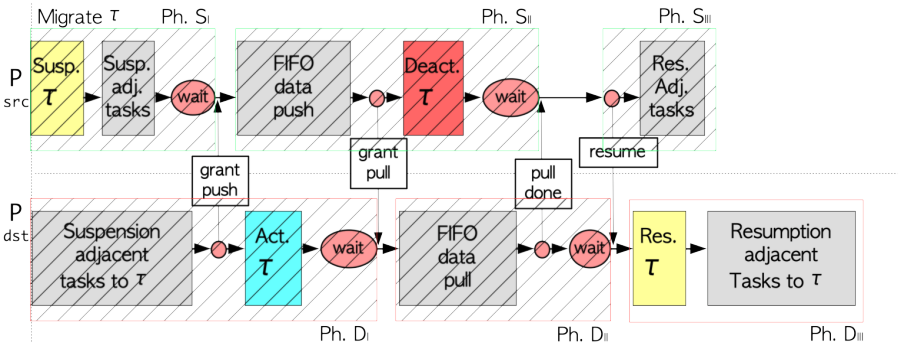


Figure 3.14: Migration actions flow diagram: Dest. Phase III

3.6 Partial-Stalling Task Migration Example

Hereby a visual *PSTM* example is shown from a high level point of view. We assume the mapping of the test DF application to be the *initial scenario* shown on Figure 3.15, where all tasks active on `T0` and inactive on `T1`. The *daemon* is part of the *System Application* and therefore has its own allocation in the TDM table (Figure 3.7b) and a shared communication channel to bestow communication between the *daemons* on each tile. The task to be migrated is task *SQR* of the test DF application.

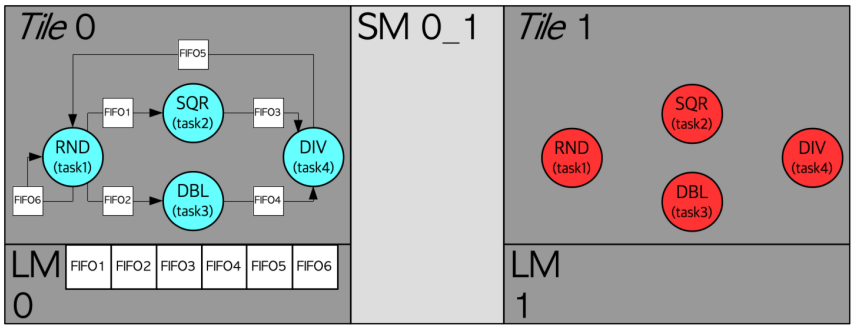


Figure 3.15: The test DF application initial mapping

At first phase I begins on source (`T0`) and destination (`T1`). On `T0`, the task is suspended; the suspension of the RoC tasks follows (3.16)). *DBL* is not directly involved, it is not suspended and can therefore continue to execute (while execution conditions are met). On `T1`, in this case, adjacent tasks are all inactive, so there is no suspension; after granting the push, *SQR* is already put in active state (3.17).

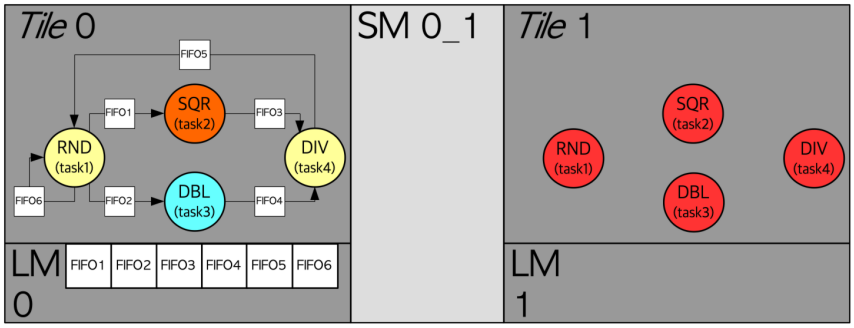


Figure 3.16: Migration of *SQR* sequence source PHASE I: suspension of *SQR* and the *RoC*

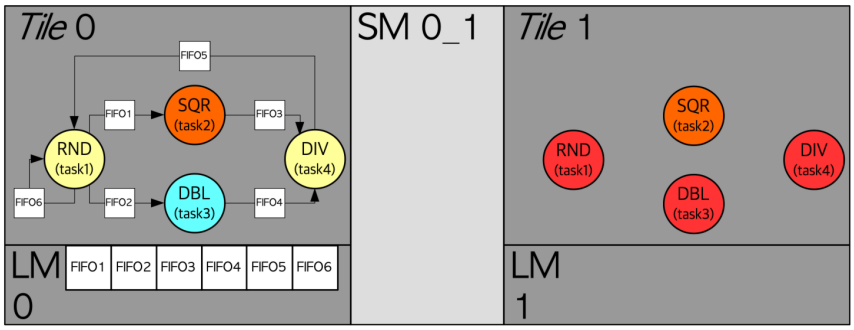


Figure 3.17: Migration of *SQR* sequence dest. PHASE I: suspension of the *RoC* and activation of *SQR*

Phase II follows on the tiles. On `T0` the involved FIFO data is therefore pushed on the shared memory and the FIFO locations are updated, where necessary; pull is granted afterwards. Then,

SQR is put to inactive state (Figure 3.18 depicts the outcome of this phase). The pull happens on `T1` and the FIFO locations are updated also here at the destination tile, where necessary and in the optimal positions (3.19).

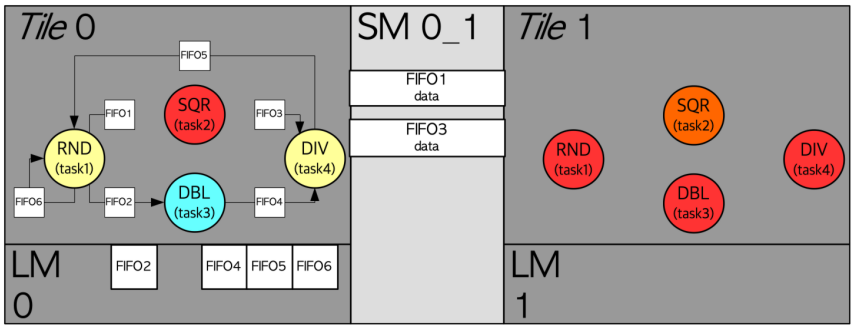


Figure 3.18: Migration of *SQR* sequence source PHASE II: push of the FIFO data and deactivation of *SQR*

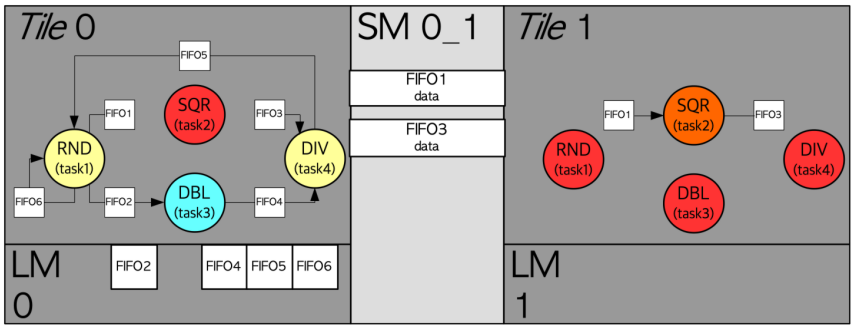


Figure 3.19: Migration of *SQR* sequence dest. PHASE II: pull of the FIFO data

After this, phase III begins. The source acknowledges the happened pull and resumption of the previously suspended RoC tasks happens (3.20). When receiving the acknowledgement, the destination resumes the migrated task *SQR* (and any RoC task in any other case where these are active). The migration is finalised (3.21).

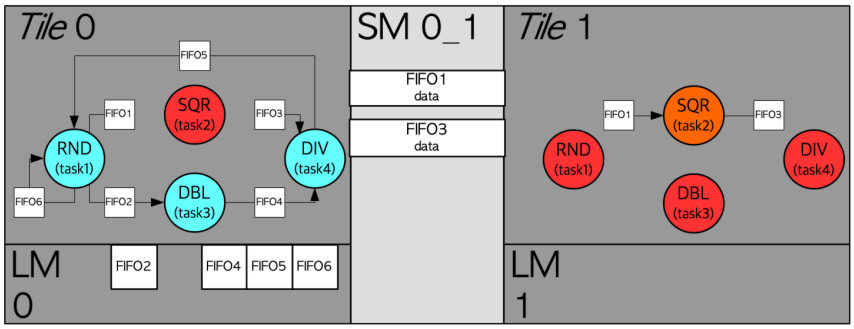


Figure 3.20: Migration of *SQR* sequence source PHASE III: resumption of the *RoC*

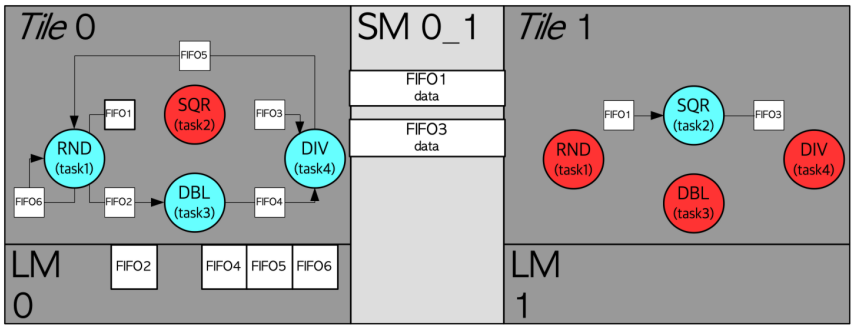


Figure 3.21: Migration of *SQR* sequence dest. PHASE III: resumption of *SQR* and the *RoC*

Before the migration, `T0` was bound to execute all the four tasks while `T1` was completely unused. The hypothetical scenario emulated in this example sees an overuse of `T0` and an underuse of `T1`; here the *RMS* commands a migration for *SQR* in order to achieve a better resource usage in terms of PEs.
On Figure 3.22 a hypothetical, yet possible, execution flow scenario of the test DF application tasks is represented in a *gantt* chart. Before, during and after migration flows are shown; the migration is issued during the execution of one of the adjacent tasks (*RND*) adding a RoC quiescence wait time before starting the migration. Task *DBL*, which was not involved in the RoC, is able to fire during the migration process. After migration, normal execution resumes, however, *DIV* is data dependent on *SQR*, and that is the reason it *starves* until *SQR* completes a new execution on its new tile.

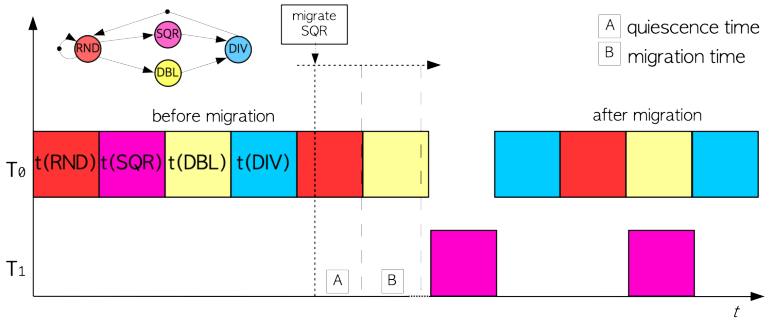


Figure 3.22: Gantt chart showing a possible test application execution flow before, during and after the migration of *SQR*

Chapter 4

Time Model and Experimentation

A reconfiguration measure such as a partial-stalling task migration, like, in general, any dynamic reconfiguration, introduces time *overhead*. Henceforth, the most interesting aspect is to be researched in the time delay introduced by this overhead and how this relatively affects the rest of the application execution. The use of a predictable MPSoC such as the Verintec platform and a predictable MoC such as the data-flow model, implies the *time predictability* of any operation done by the platform, including the PSTM; i.e. the operations execution times can be estimated in their worst-case at design time, before mapping and execution. Because of this, it was decided to evaluate the experimentation through timing analysis.
First, the time model for the PSTM is to be designed, Section 4.1 gives details about how this is crafted and how the model works. Secondly, such model is used to make measurements in our case study; Section 4.2 goes over this. Section 4.3 reflects over the previously obtained measurements and evaluations over the implemented migration solution are given.

4.1 Time Model

The developed time model has the objective of defining the worst-case execution time of a partial-stalling task migration *for any given task of the target application in any given mapping of the target application over the MPSoC tiles and with any given TDM scheduling slot assignment.* The $WCMT$ will depend on two macro factors: from the application fine grained point of view, the expected execution time of the migration mechanism itself and, from the platform coarse grained point of view, the execution of the latter over the defined TDM scheduling tables of the involved tiles. Combining the two is the key to a reliable timing model for computing an upper time bound for any possible migration in the application-platform system. The calculation must then follow a bottom-up fashion, starting from the most atomic elements: the migration API functions.

4.1.1 Migration Phases Worst-Case Times

Regarding the migration mechanism worst-case response times of the phases, the time measurement model was retrieved by analyzing the implementation that made use of the test DF application with the timer available in the processor. The functions that form the migration API, have both a timely *static* and (occasionally) a timely *dynamic* execution part (depending on the application mapping and characteristics). The execution time of the migration API functions would in fact mainly depend on different application parameters such as: number of tasks adjacent to the task to migrate, number of self edges of the task to migrate, task activeness, number of production and consumption FIFOs, token size, buffer capacity, FIFO locality.

Time Computing Functions

The analysis process consisted, at first, into constructing, for each function, execution time equations, defining precisely the static and dynamic parts (where present), highlighting all the involved variables (listed above) that influenced the dynamic part.
Secondly, the same were measured in the implementation featuring the test application with the use of a processor clock timer; each time measure was equalized to the corresponding time computing equation. Finally, by reverse arithmetic, with the objective of constructing general equations applicable to any object application, the weight for each of the variables, and therefore the dynamic parts of the equations, were defined. By changing the parameters of the test target application and therefore obtaining new time measures, the aforementioned weights were calibrated until the expected measures had a (pessimistic) error of around 10%.
Each function in the migration mechanism can be then summarized as follows:

$$time(migration\_function) = time(static\_overhead) + time(dynamic\_part) \tag{4.1}$$

where the number of cycles of the dynamic part is, depending on the considered migration function, either 0 or a function of the number of tasks adjacent to the task to migrate, number of self edges of the task to migrate, number of production and consumption FIFOs, token size, buffer capacity or FIFO locality:

$$time(dynamic\_part) = f(n_{adj}, n_{self}, n_{prod\_fifos}, n_{cons\_fifos}, token\_size, buffer\_size, fifo\_locality) \tag{4.2}$$

At the end of the process, the `time()` function for the migration API functions will be shaped as shown in Listing 4.1.1.

```
time(migration_api_function) =
            static_number_of_cycles +   // usually func. call and loop overhead
            dynamic_number_of_cycles

where

dynamic_number_of_cycles = f(n_adj_tasks, n_self_edges, n_prod_fifos, n_cons_fifos,
     token_size, buffer_size, fifo_locality)
```

Listing 4.1: The generic `time()` function

More detail about the time measurement of each of the migration API functions is shown in Appendix C.

Migration Phases

For the migration mechanism as a whole, we take in account the division in phases, as described previously in Section 3.5. These are represented and made of pseudo function names (for ease of representation) on the schematic in Figure 4.1. When calculating the global WCMT at design time, computing this bound on the migration phases is the first step.
Separating each phase is a synchronization barrier due to the communication between the tiles. For each phase one can then define a worst-case migration phase time. For computing these worst-case time bounds on the source and destination migration phases, we do *not* take into account the cycles spent waiting for synchronizations; these waiting latencies will be accounted in the second part of the WCMT computation (Section 4.1.2), when we will consider the bigger picture with both the tiles involved.
As the depicted phases suggest, once the atomic migration API steps worst-case timings are computed, building the equations is straightforward. For the source phase I, the worst-case time will be the sum of the cycles spent to suspend task τ and the adjacent tasks to the latter active on the tile:

$$WC\_S_I = time(SRC\_suspend(\tau)) + time(SRC\_suspend\_adj(\tau)) \tag{4.3}$$

Phase	Source Tile	Destination Tile	Phase
S_I	susp τ, susp_adjacent τ while (no grant to push) ←	suspend_adjacent τ grant_push	D_I
S_{II}	push_state τ grant_pull →	activate τ while (no grant to pull)	
	deactivate τ while (pull not complete) ←	pull_state τ pull_done	D_{II}
S_{III}	grant_resume →	while (resume not granted)	
	resume_adjacent τ	res τ, res_adjacent τ	D_{III}

Figure 4.1: Table schematization of the migration phases

Similarly, for destination phase I, we account the cycles spent to suspend the *RoC* tasks active on the tile. In addition, we sum the time spent to send the *push grant* signal and the time to activate τ:

$$WC\_D_I = time(DST\_suspend\_adj(\tau)) + time(DST\_grant\_push) + time(DST\_activate(\tau)) \quad (4.4)$$

Regarding source phase II, the worst-case time will be the sum of the cycles to push the state of τ, the cycles necessary to send the *pull grant* signal and the cycles to deactivate τ:

$$WC\_S_{II} = time(SRC\_push\_state(\tau)) + time(SRC\_grant\_pull) + time(SRC\_deactivate(\tau)) \quad (4.5)$$

Destination phase II, on the other hand, accounts the time spent to pull the state of τ plus the one spent on sending the *pull done* signal:

$$WC\_D_{II} = time(DST\_pull\_state(\tau)) + time(DST\_pull\_done) \quad (4.6)$$

The bound on last source phase, phase III, is expected to be the sum of the cycles expired to send the *resume* signal and the cycles invested to resume the *RoC* tasks that had been suspended in phase I:

$$WC\_S_{III} = time(SRC\_grant\_resume) + time(SRC\_resume\_adj(\tau)) \quad (4.7)$$

As soon as the *resume* signal is received, destination phase III begins. The expected bound is given by the sum of the time to resume τ on its new tile and the adjacent tasks previously suspended in phase I:

$$WC\_D_{III} = time(DST\_resume(\tau)) + time(DST\_resume\_adj(\tau)) \quad (4.8)$$

4.1.2 Time Division Multiplexing Worst-Case Scenario

The classic worst-case scenario where one has to wait for a whole TDM cycle before the application can start to execute, is not enough in the specific case of our *Partial Stalling Task Migration*. The migration mechanism is, in fact, not only dependent on two different TDM frames from two different tiles (source and destination), but it also involves communication steps between the *migration daemons* (each running in the System Application slot of the TDM frame, see Figure 4.2) on each one of them, introducing the risk of nested worst-case response time scenarios to the communications.

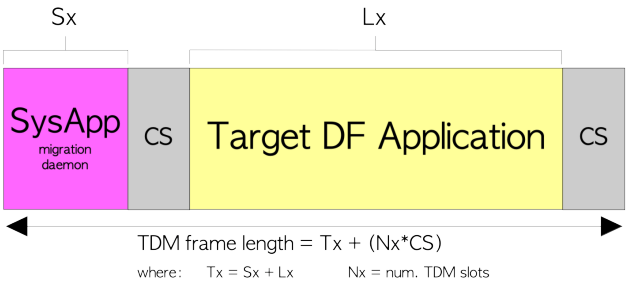


Figure 4.2: An example of the TDM frame structure used in the experiments. The *migration daemon* is part of the System Application

For the purpose of demonstrating the final formula to compute the upper bound on the *WCMT* we consider the graphical example depicted on Figure 4.3. First of all, we consider our demonstrative TDM frames to be made of just two slots, one for the SA and the other for the target DF application. Since our model will be proportional to the TDM frame size, we are, in such way, sure to compute the most pessimistic upper bound; both the SA and the application have in fact only one slot per TDM cycle to execute and preemptions would postpone the completion to the next TDM frame (depending of course on the ratio between execution times and the slot sizes). Also, on the platform, the clocks of the different tiles are synchronized, so we assume this condition is met for developing our model.

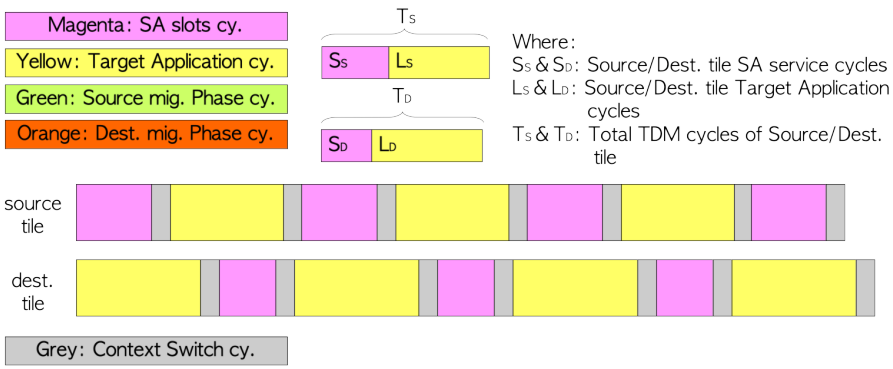


Figure 4.3: The worst-case scenario TDM frame sequences graphical analysis tools

The first requirement to have a worst-case happens when all of the phases experience a waiting time, i.e. *they get to the wait step without having received the necessary signal to exit it.* This situation happens when the SA (who hosts the *migration daemon*) service slot of the source tile is in a chronologically antecedent position compared to its destination tile counterpart (as depicted on the example in 4.3).
At first, for simplicity, we now consider the single phase bounds, WC_S_I, WC_D_I, WC_S_{II}, WC_D_{II}, WC_S_{III}, WC_D_{III}, to be all smaller than the allocated service slots on source and destination TDMs (S_S and S_D), so that we are sure they execute each within one service slot (without getting preempted). The execution of the migration phases is now represented on Figure 4.4.
We can now start to build our upper bound mathematical formula. By following the representation on Figure 4.4, we can observe that there will be, first of all, a time component equal to the *maximum possible discrepancy* between S_S and S_D. We will call this cycle amount $max\phi$ and compute it as follows:

(4.9)

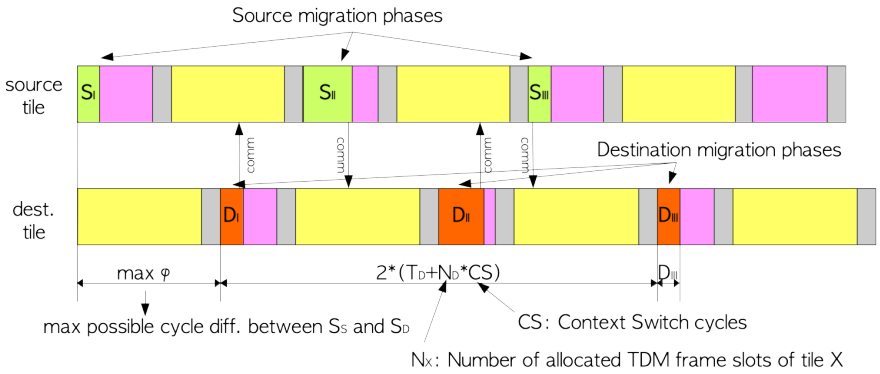


Figure 4.4: The worst-case scenario assuming no preemptions happen for the migration mechanism

$$max\phi = \begin{cases} N_D * CS + L_D & \text{if } T_S \geq T_D \\ (T_S + N_S * CS) - S_D, & \text{otherwise} \end{cases}$$

where, N_X represents the number of distinct slots of the X TDM table, CS the cycles elapsed during context switch between the cycles (4096), and L_X the cycles of the TDM that do not service SA (computed as $L_X = T_X - S_X$ with T_X being the total cycles of the set TDM frame, as suggested in Figure 4.3).
Now, observing again Figure 4.4, we can notice that, in this worst-case scenario, the migration is *dominated temporally by the destination tile* since it will inevitably conclude here with destination phase III. The end of the $max\phi$ time, coincides with the begin of the destination tile migration phases; in the example, the time elapsed from there to the begin of the last phase, equals the time of two destination TDM cycles ($2 * T_D$) plus also the must be accounted context switches ($2 * (N_D * CS)$). To $max\phi$ and this second component just described, we add in the end the migration finalization of the third destination phase (WC_D_{III}).
We have now three of the four components of our upper bound (WCMT) equation:

$$WCMT = max\phi + 2 * (T_D + N_D * CS) + WC\_D_{III} + P_{wait} \tag{4.10}$$

The missing component, P_{wait}, represents the *impact of preemptions* in the migration phases on the WCMT. On Figure 4.4 we considered, in fact, worst-case times for the phases smaller than the service cycles. However, this might not be the case in certain situations. On Figures 4.5 and 4.6 we can observe the effects of the preemptions of S_{II} and D_{II}, respectively.

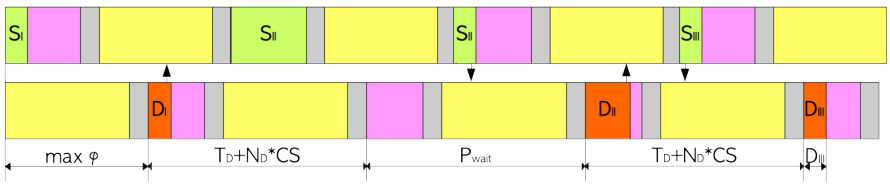


Figure 4.5: The worst-case scenario assuming a preemption happens for phase S_{II}

On 4.5 we can see that the effect of the preemption on the source tile, can result in waiting cycles

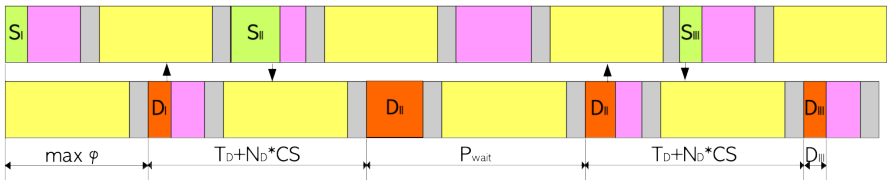


Figure 4.6: The worst-case scenario assuming a preemption happens for phase D_{II}

on the *dominating* destination TDM. Therefore, one must, first of all, compute the likelihood of preemptions on the source tile by considering the ratio between the worst-case bound on the phases and a single service slot:

$$P_S = \frac{WC\_S_I + WC\_S_{II} + WC\_S_{III}}{S_S} \tag{4.11}$$

Although this approach is pessimistic, since not all the phases execute on a single service slot in a worst-case scenario, it ensures that no preemptions are missed in the calculation. In computing an upper bound we must in fact not miss preemptions, we thus tolerate this calculation.
Now that we have the probability of the number of preemptions of the source side of the migration, we must compute the impact of these on the destination tile. Since *every preemption will trigger a missed TDM cycle* on the source tile, this is rather straightforward: the ratio of the total number of the missed source cycles and the size of the destination TDM, will result in the likely number of missed TDM frame cycles on the destination tile. The described computation follows:

$$P_{S2D} = \frac{P_S * (T_S + N_S * CS)}{(T_D + N_D * CS)} \tag{4.12}$$

After this, we move to the opposite example, depicted on Figure 4.6, where the preemption happens on the destination tile. In this case, we will just need to compute the likelihood of preemptions on the destination tile, since we assumed the destination tile as timely dominant. The computation goes in the same direction as Equation 4.11:

$$P_D = \frac{WC\_D_I + WC\_D_{II} + WC\_D_{III}}{S_D} \tag{4.13}$$

Considering that every preemption causes a waiting destination TDM frame, we have now all the necessary to build the last component of the $WCMT$ equation, P_{wait}. Before proceeding, we must consider that, for ease of representation, the preemption on source and destination have been depicted separately and for just one of the phases of the migration; in a real worst-case scenario they can happen on source and destination at the same time and also multiple times. Because of this, we combine the effects of the expected preemptions on source and destination, by summing them linearly, and approximating to the upper integer to quantize the additional spent times to the size of the destination TDM frame.
The P_{wait} equation follows:

$$P_{wait} = \lceil P_D + P_{S2D} \rceil * (T_D + N_D * CS) \tag{4.14}$$

The $WCMT$ equation (4.10) is now complete in all of its components.
The *ceiling approximation* is certainly overpessimistic compared to the softer *nearest integer ap-*

proximation, if we consider that a likely number of preemptions smaller than $x.5$ would be approximated to $x+1$ preemptions instead of x preemptions, but ensures that the computed $WCMT$ is a mathematical upper bound.

4.1.3 Subsequent Migrations

Another additional scenario we want to analyze is the following: a migration B was issued before the end of a migration A, meaning that the former will start as soon as the latter is done; e.g. the migrations are not decoupled and must be considered as whole in the calculation. In this case the total worst-case of the two migrations will be the sum of the two worst-cases plus an additional component, that is an intermediate worst-case wait between the two, that will amount to a maximum of one destination tile TDM cycle:

$$WCMT_{AB} = WCMT_A + (T_D + N_D * CS) + WCMT_B \tag{4.15}$$

A generalization of the formula for n migrations follows:

$$WCMT_n = \sum_{i=1}^{n} WCMT_i + \sum_{i=1}^{n-1} (T_D + N_D * CS) \tag{4.16}$$

4.2 Experimentation on the Case Study and Results

An experimentation using the introduced timing model is done in this section, where we will compute the upper bound for two migrations on the JPEG decoder (with four different TDM frame settings in total). By taking a look at the worst-case timing of the migration API functions (in Appendix C), we can expect the bottleneck of the migration mechanism time to happen when FIFOs are transferred in location (where every transfer of a single byte of information B, takes 14 cycles). These are also depicted on Figure 4.7 where the impact of the FIFO size on the time to move it, is visualized. Below, is a table showing the sizes of the FIFOs (*token_size*buffer_capacity*) for the JPEG decoder application data-flow model (Table 4.1).
Regarding the execution times of the single JPEG decoder actors, their duration can vary and depends on the test image being used; Therefore, one single test image was used through the experimentation to preserve the predictability of the environment. This matter is not central for the purposes of this study, but more detail about the worst-case actor execution times and the test image is given in Appendix B.
By taking a look at the total size of the buffers, we can already expect the most enduring migration mechanisms in our case study to be the ones that involve FIFO 2 (3200 B) or FIFO 7 (3072 B), that are both about as five times as big than the third FIFO in size (number 6, with 640B). For this reason we will take into considerations two migrations, one involving FIFO 7 and the other involving FIFO 2 (of course they cannot be moved in a same migration since they do not share a same producing nor consuming task). First we will predict the worst-case time for the migration

FIFOlabel	*Routing*	*token_size* (B)	*buffer_capacity*	*total_size* (B)
1	*VLD* local	448	1	448
2	*VLD* to *IQZZ*	320	10	3200
3	*VLD* to *CC*	60	1	60
4	*VLD* to *RASTER*	52	1	52
5	*IQZZ* to *IDCT*	256	1	256
6	*IDCT* to *CC*	64	10	640
7	*CC* to *RASTER*	3072	1	3072

Table 4.1: The JPEG decoder FIFOs and their size

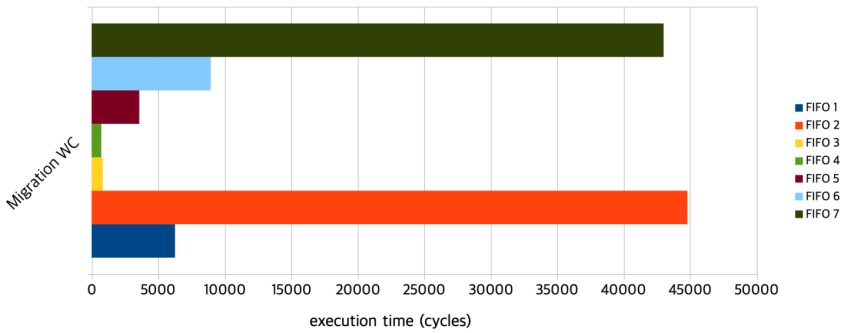


Figure 4.7: The graph shows a comparison of the expected time amount needed to move each of the seven FIFO buffers

of the *CC* actor from `T1` to `T0` (default mapping is shown on Figure 4.8). Then, from the resulting mapping of this first migration (Figure 4.12)), the migration of the *IQZZ* actor also from `T1` to `T0` (that will result in Figure 4.16). As anticipated, within the scenario, we study two subscenarios that differ in terms of the designed TDM frame. However, in both cases, the TDMs will be made of two slots: one for the System Application (that hosts the *migration daemon*), and the other for the execution of the JPEG decoder itself (on the model of the example shown on Figure 4.2). The reason for this decision is that our model would give a less precise bound for TDMs that are made of more slots, while in this section of the study, we are more interested in validating the model in a more critical situation (where the upper bound is expected to be relatively closer to the measured WCMT).

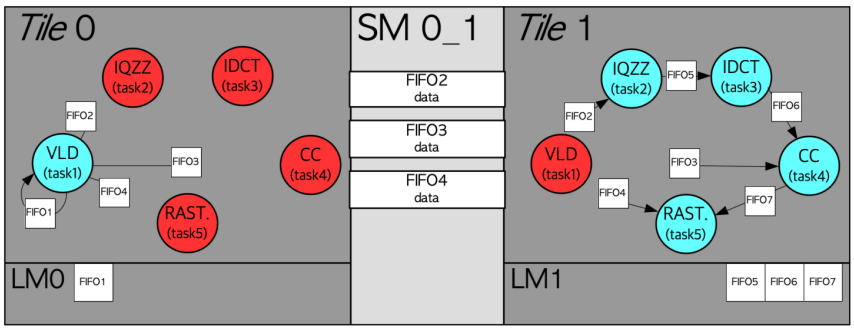


Figure 4.8: The JPEG decoder experimentation initial mapping

4.2.1 First Measured Migration: *CC* task

In this scenario, the application is running with the mapping depicted in Figure 4.8. By migrating *CC* we identify also *IDCT*, *RASTER* and *VLD* in its RoC: These listed tasks are paused during the target migration. Also, the FIFO buffers that bestow communication to and from *CC* are the ones to be moved; these are number 3, 6 and 7. FIFO 3 is in a *shared* locality state, so we expect the mechanism to make it fully *local* once the migration of *CC* is complete; this means that moving this buffer will be more impactful when *pulling*. FIFOs 6 and 7, on the other hand, are fully *local* on `T1`, so we expect the mechanism to make them *shared* once the migration is complete (as suggested by the state machine we saw earlier in Figure 3.5b); meaning that moving these buffers is heavier when *pushing*.

Step I: API functions worst-case timings

As shown in the demonstration of the timing model of Section 4.1. We start computing the WCMT from the single worst-case timing of the executed migration API functions. By using the derived formulas (available in Appendix C), the computed values are displayed on Table 4.2

Migration API function	$WCET$ (cycles)
$SRC\_suspend(CC)$	8
$SRC\_suspend\_adj(CC)$	178
$SRC\_push\_state(CC)$	52277
SRC_grant_pull	6
$SRC\_deactivate(CC)$	8
SRC_grant_resume	6
$SRC\_resume\_adj(CC)$	176
$DST\_suspend\_adj(CC)$	176
DST_grant_push	6
$DST\_activate(CC)$	9
$DST\_pull\_state(CC)$	1199
DST_pull_done	6
$DST\_resume(CC)$	7
$DST\_resume\_adj(CC)$	175

Table 4.2: The single migration API functions worst-case execution times for the CC migration scenario

Step II: Migration phases worst-case timings

We go on by determining the upper bounds of the single migration phases by using the Equations from 4.3 to 4.8. The values are shown on Table 4.3.

Migration Phase	$WCET$ (cycles)
S_I	186
D_I	191
S_{II}	52291
D_{II}	1205
S_{III}	182
D_{III}	182

Table 4.3: The migration phases worst-case execution times for the CC migration scenario

We now do a first experimentation, by measuring the empirical values from the same described migration on the platform. The obtained values are shown on 4.4, where the previously predicted cycles for the API functions executions are shown besides the actual measured cycles.
As we can see on Figure 4.9, the bottleneck of the migration mechanism happens, as anticipated, when moving the FIFO buffers and, in this specific case, when pushing the FIFOs to the shared memory for migration. In the graph, the `push_state` function execution takes up basically the whole mechanism cycles; in this particular case it is even more explicit due to the need to move FIFO 7, which has the relatively big size of 3072B.
Of a total predicted number of cycles 54237, the actual measurement is 53082, resulting in the following delta: $\Delta = 54237 - 53082 = 1155$ This Δ means that the predicted values were just 2.17% more than the actual ones. Also, taking a look at Figure 4.9, we can visualize the almost non-perceivable difference between the two. The precision is such due to the fact that the single atomic API functions are expected to execute always with the same number of cycles with a given

Migration API function	$WCET$ (cycles)	Measured ET (cycles)	Δ
$SRC\_suspend(CC)$	8	8	0%
$SRC\_suspend\_adj(CC)$	178	162	9.88%
$SRC\_push\_state(CC)$	52277	51313	1.88%
SRC_grant_pull	6	6	0%
$SRC\_deactivate(CC)$	8	8	0%
SRC_grant_resume	6	6	0%
$SRC\_resume\_adj(CC)$	176	160	10%
$DST\_suspend\_adj(CC)$	176	166	6.02%
DST_grant_push	6	6	0%
$DST\_activate(CC)$	9	9	0%
$DST\_pull\_state(CC)$	1199	1060	13.11%
DST_pull_done	6	6	0%
$DST\_resume(CC)$	7	7	0%
$DST\_resume\_adj(CC)$	175	165	6.06%
total	54237	53082	2.17%

Table 4.4: Comparison of the single functions expected execution times and measured execution times for the CC migration scenario. The relative error is also shown

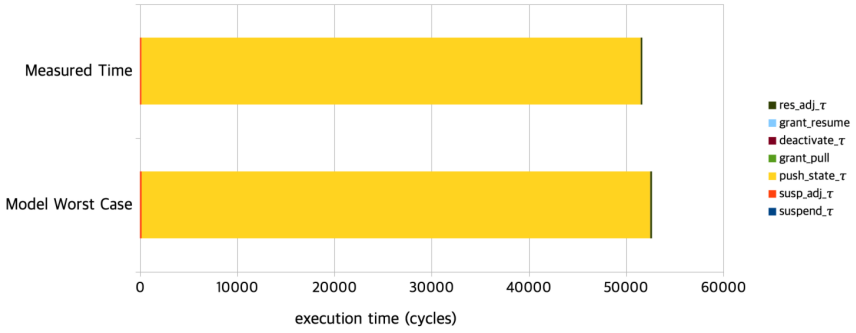


Figure 4.9: The graph in figure represents both the computed and measured execution cycles of the source migration functions stacked one onto each other for the CC task migration

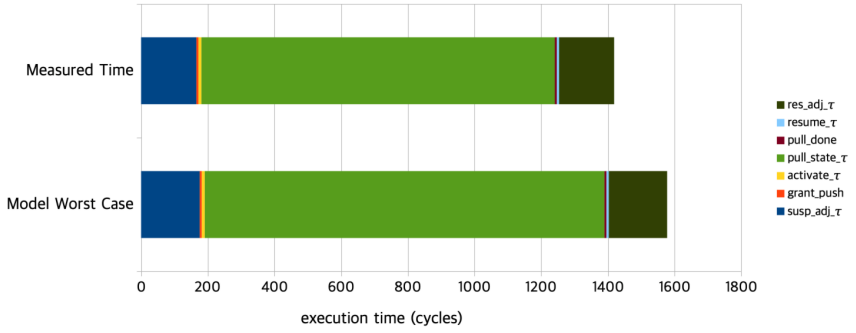


Figure 4.10: The graph in figure represents both the computed and measured execution cycles of the destination migration functions stacked one onto each other for the CC task migration

application mapping (except for the *push* and *pull* ones). This expectation is perfectly accounted with the designed model (see Appendix C) that, up to this stage of the measurement, keeps the Δ small. As we will see right away, the biggest contribution to the global migration time Δ, comes when we consider the bigger migration picture with the source and destination TDM frames interactions to bestow the migration.

Subscenario I: $T_S \geq T_D$

In the first case, we assign for the source tile `T1` a bigger TDM frame then the destination tile `T0`. The values are shown in Table 4.5, where L_X is the number of slot cycles assigned to the JPEG application (named L because of it is seen as a latency time from the migration point of view) and S_X is the number of service slots cycles for the SA (and therefore the *daemon*); both for a generic tile X.
Being FIFO 7 of 3072B in size, we already know that at least 43008 cycles (3072*14) will be spent on moving only this FIFO from `T1` to the shared memory. Therefore, to enhance the likelihood of preemption of the migration phases, as shown in 4.1.2, we assign the two S_X to be around the same order of magnitude (50000 for the destination tile and 70000 for the source).

$Tile$	L_X (cycles)	S_X (cycles)
0	100000	50000
1	140000	70000

Table 4.5: The subscenario I TDM frames of the tiles for the *CC* migration

To compute the WCMT, we start with $max\phi$; by following Equation 4.9 we obtain the following:

$$max\phi = N_D * CS + L_D = 108192 cycles$$

Following the global worst-case time Equation 4.10 defined previously in Section 4.1 the WCMT, assuming no preemptions happen, is:

$$WCMT_{np} = max\phi + 2 * (T_0 + N_0 * CS) + WCD_{III} = 108192 + 308192 + 182 = 416566 cycles$$

We now compute the likelihood of preemptions happening:

$$P_0 = \frac{(WCD_I + WCD_{II} + WCD_{III})}{S_D} = \frac{(191+1205+182)}{50000} = 0.03$$
$$P_1 = \frac{(WCS_I + WCS_{II} + WCS_{III})}{S_D} = \frac{(186+52291+182)}{70000} = 0.75$$

We proceed to calculate the influence of the preemptions of the source tile on the destination tile:

$$P_{1to0} = \frac{P_1 * (T_1 + N_1 * CS)}{(T_0 + N_0 * CS)} = \frac{0.75 * (218192)}{158192} = 1.03$$

The total number of expected preemptions according to the model follows:

$$P_{tot} = \lceil P_0 + P_{1to0+} \rceil = \lceil 1.06 \rceil = 2$$

Hence, the WCMT considering preemption, according to Equation 4.10 is:

$$WCMT = WCMT_{np} + P_{tot} * (T_0 + N_0 * CS) = 416566 + 316384 = 732950$$

Timer Measurements Subscenario I

With this mapping and TDM configuration, 10 simulations of the *CC* migration were run and timed with the embedded processor timer. The results are shown on Table 4.6.

Average Measured MT (of 10 measurements) (cycles)	Measured $WCMT$ (cycles)
410748	474493

Table 4.6: The subscenario I measured values for the CC migration

With a measured WCMT of 732950, the time model successfully predicted an upper bound, with an error of:

$\Delta = 732950 - 474493 = 258457 cycles$
$\Delta_{\%} = (\Delta / 474493) * 100 = 54.47\%$

Subscenario II: $T_S < T_D$

In the second subscenario, we assign for the destination tile `T0` a bigger TDM frame then the source tile `T1`. The values are shown in Table 4.7.

$Tile$	L_X (cycles)	S_X (cycles)
0	150000	80000
1	90000	40000

Table 4.7: The subscenario II TDM frames of the tiles for the CC migration

To compute the WCMT, we proceed now in the same way as subscenario I, starting with the maximum service slots discrepancy $max\phi$:

$max\phi = (T_S + N_S * CS) - S_D = (T_1 + N_1 * CS) - S_0 = 138192 - 80000 = 58192 cycles$

The global worst-case time follows, beginning with the assumption that no preemptions happen:

$WCMT_{np} = max\phi + 2 * (T_0 + N_0 * CS) + WCD_{III} = 58192 + 476384 + 182 = 534758 cycles$

We now compute the likelihood of preemptions happening:

$P_0 = \frac{(WCD_I + WCD_{II} + WCD_{III})}{S_D} = \frac{(191+1205+182)}{80000} = 0.02$
$P_1 = \frac{(WCS_I + WCS_{II} + WCS_{III})}{S_D} = \frac{(186+52291+182)}{40000} = 1.32$

We proceed to calculate the influence of the preemptions of the source tile on the destination tile:

$P_{1to0} = \frac{P_1 * (T_1 + N_1 * CS)}{(T_0 + N_0 * CS)} = \frac{1.32 * (138192)}{238192} = 0.77$

The total number of expected preemptions according to the model follows:

$P_{tot} = \lceil P_0 + P_{1to0} \rceil = \lceil 0.79 \rceil = 1$

Hence, the WCMT considering preemption, according to Equation 4.10 is:

$WCMT = WCMT_{np} + P_{tot} * (T_0 + N_0 * CS) = 534758 + 238192 = 772950$

Timer Measurements Subscenario II

With this mapping and other TDM configuration, 10 simulations of the CC migration were run and timed with the embedded processor timer. The results are shown on Table 4.8.

Average Measured MT (of 10 measurements) (cycles)	Measured $WCMT$ (cycles)
479745	662964

Table 4.8: The subscenario II measured values for the *CC* migration

With a measured WCMT of 772950, the time model successfully predicted an upper bound, with an error of:

$\Delta = 772950 - 662964 = 109986 cycles$
$\Delta_{\%} = (\Delta / 662964) * 100 = 16.59\%$

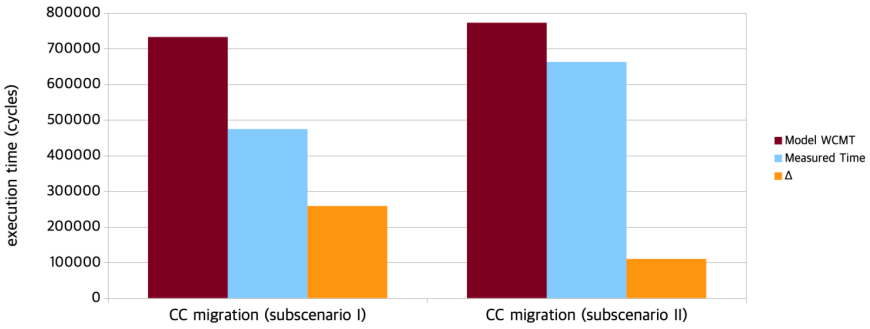


Figure 4.11: The graph in figure shows the difference between the WCMT foreseen by the model and the worst obtained measurement for the *CC* task migration

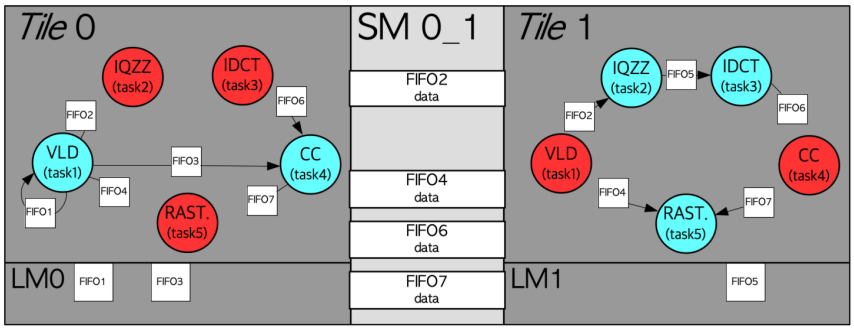


Figure 4.12: The JPEG decoder mapping after the *CC* migration

4.2.2 Second Migration: *IQZZ* task

In the second scenario, the application is running with the mapping depicted in Figure 4.12. By migrating *IQZZ* we identify also *IDCT* and *VLD* in its RoC: These listed tasks are paused during the target migration. Also, the FIFO buffers that bestow communication to and from *IQZZ* are the ones to be moved; these are number 2 and 5. FIFO 2 is in a *shared* locality state, so we expect the mechanism to make it fully *local* once the migration of *IQZZ* is complete (as suggested by the state machine we saw earlier in Figure 3.5b); this means that moving this buffer will be more impactful when *pulling*. FIFO 5, on the other hand, is fully *local* on `T1`, so we expect the mechanism to make it *shared* once the migration is complete; meaning that moving this buffers is heavier when *pushing*.

Step I: API functions worst-case timings

Also for this scenario, we start computing the WCMT from the single worst-case timing of the executed migration API functions. The computed values are displayed on Table 4.9

Migration API function	$WCET$ (cycles)
$SRC\_suspend(IQZZ)$	8
$SRC\_suspend\_adj(IQZZ)$	135
$SRC\_push\_state(IQZZ)$	3801
SRC_grant_pull	6
$SRC\_deactivate(IQZZ)$	8
SRC_grant_resume	6
$SRC\_resume\_adj(IQZZ)$	134
$DST\_suspend\_adj(IQZZ)$	135
DST_grant_push	6
$DST\_activate(IQZZ)$	9
$DST\_pull\_state(IQZZ)$	45017
DST_pull_done	6
$DST\_resume(IQZZ)$	7
$DST\_resume\_adj(IQZZ)$	134

Table 4.9: The single migration API functions worst-case execution times for the *IQZZ* migration scenario

Step II: Migration phases worst-case timings

Again, we go on by determining the upper bounds of the single migration phases, as did for the first scenario. The values are shown on Table 4.10.

Migration Phase	$WCET$ (cycles)
S_I	143
D_I	150
S_{II}	3815
D_{II}	45023
S_{III}	140
D_{III}	141

Table 4.10: The migration phases worst-case execution times for the *IQZZ* migration scenario

We do a first experimentation also for this *IQZZ* migration. We measure the empirical values from the execution of the migration on the platform. The obtained values are shown on 4.11, where the previously predicted cycles for the API functions executions are shown besides the actual measured cycles.
As we can see on Figure 4.14, like in the first migration experiment, the bottleneck of the migration mechanism happens when moving the FIFO buffers and, in this other case, when pulling the FIFOs from the shared memory to local. In the graph, the `pull_state` function execution takes up almost the whole mechanism cycles; this time it happens due to the need to move FIFO 2, which has a relatively big size of 3200B.
In this case the cumulative delta is: $\Delta = 49412 - 46160 = 3252$ Therefore the predicted values were 7.04% more than the actual ones. Again the predicted atomic migration API values are precise (see the graph on Figure 4.14), but, also in this case we expect the biggest contribution to the global Δ to come from the following global migration analysis.

Migration API function	$WCET$ (cycles)	Measured ET (cycles)	Δ
$SRC\_suspend(IQZZ)$	8	8	0%
$SRC\_suspend\_adj(IQZZ)$	135	121	11.57%
$SRC\_push\_state(IQZZ)$	3801	3796	0.13%
SRC_grant_pull	6	6	0%
$SRC\_deactivate(IQZZ)$	8	8	0%
SRC_grant_resume	6	6	0%
$SRC\_resume\_adj(IQZZ)$	134	120	11.67%
$DST\_suspend\_adj(IQZZ)$	135	122	10.65%
DST_grant_push	6	6	0%
$DST\_activate(IQZZ)$	9	9	0%
$DST\_pull\_state(IQZZ)$	45017	41824	7.63%
DST_pull_done	6	6	0%
$DST\_resume(IQZZ)$	7	7	0%
$DST\_resume\_adj(IQZZ)$	134	121	10.74%
total	49412	46160	7.04%

Table 4.11: Comparison of the single functions expected execution times and measured execution times for the $IQZZ$ migration scenario. The relative error is also shown

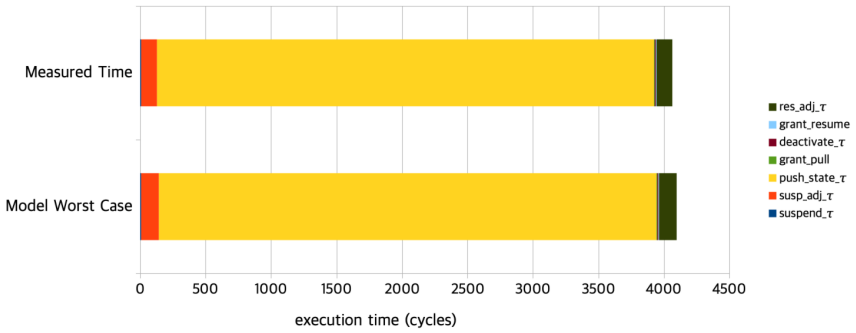


Figure 4.13: The graph in figure represents both the computed and measured execution cycles of the source migration functions stacked one onto each other for the $IQZZ$ task migration

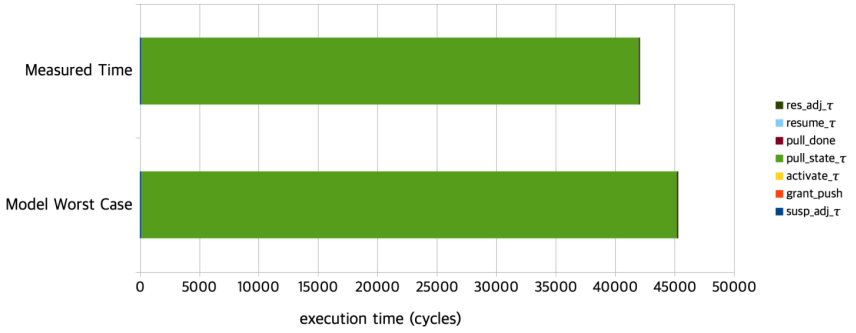


Figure 4.14: The graph in figure represents both the computed and measured execution cycles of the destination migration functions stacked one onto each other for the $IQZZ$ task migration

Subscenario I: $T_S \geq T_D$

In the first subscenario of this second experiment migration, we again assign for the source tile `T1` a bigger TDM frame then the destination tile `T0` (values are shown in Table 4.5. Being FIFO 2 of 3200B in size, we already know that 44800 cycles (3200*14) will be spent on moving only this FIFO from `T1` to the shared memory. Therefore, to enhance the likelihood of preemption of the migration phases, as shown in 4.1.2, we assign the two S_X to be again around the same order of magnitude.

$Tile$	L_X (cycles)	S_X (cycles)
0	90000	40000
1	150000	80000

Table 4.12: The subscenario I TDM frames of the tiles for the *IQZZ* migration

As we did for the first migration of *CC* we compute the WCMT the same way. We start with $max\phi$:

$max\phi = N_D * CS + L_D = 98192 cycles$

Following the global worst-case time Equation 4.10, the WCMT, assuming no preemptions happen, is:

$WCMT_{np} = max\phi + 2 * (T_0 + N_0 * CS) + WCD_{III} = 98192 + 276384 + 141 = 374717 cycles$

We now compute the likelihood of preemptions happening:

$P_0 = \frac{(WCD_I + WCD_{II} + WCD_{III})}{S_D} = \frac{(150+45023+141)}{40000} = 1.13$
$P_1 = \frac{(WCS_I + WCS_{II} + WCS_{III})}{S_D} = \frac{(143+3815+140)}{80000} = 0.05$

We proceed to calculate the influence of the preemptions of the source tile on the destination tile:

$P_{1to0} = \frac{P_1 * (T_1 + N_1 * CS)}{(T_0 + N_0 * CS)} = \frac{0.05 * (238192)}{138192} = 0.09$

The total number of expected preemptions according to the model follows:

$P_{tot} = \lceil P_0 + P_{1to0} \rceil = \lceil 1.22 \rceil = 2$

Hence, the WCMT considering preemption, according to Equation 4.10 is:

$WCMT = WCMT_{np} + P_{tot} * (T_0 + N_0 * CS) = 374717 + 276384 = 651101$

Timer Measurements Subscenario I

With this mapping and TDM configuration, 10 simulations of the *IQZZ* migration were run and timed with the embedded processor timer. The results are shown on Table 4.13.

Average Measured MT (of 10 measurements) (cycles)	Measured $WCMT$ (cycles)
380783	422506

Table 4.13: The subscenario I measured values for the *IQZZ* migration

With a measured WCMT of 651101, the time model successfully predicted an upper bound, with an error of:

$\Delta = 651101 - 422506 = 228595 cycles$
$\Delta_{\%} = (\Delta / 422506) * 100 = 51.66\%$

Subscenario II: $T_S < T_D$

In the second subscenario, we assign for the destination tile `T0` a bigger TDM frame then the source tile `T1`. The values are shown in Table 4.14.

$Tile$	L_X (cycles)	S_X (cycles)
0	140000	70000
1	100000	50000

Table 4.14: The subscenario II TDM frames of the tiles for the *IQZZ* migration

To compute the WCMT, we proceed, one last time, with the same procedure, beginning from $max\phi$:

$max\phi = (T_S + N_S * CS) - S_D = (T_1 + N_1 * CS) - S_0 = 158192 - 70000 = 88192 cycles$

By following the usual equation with the assumption that no preemptions happen:

$WCMT_{np} = max\phi + 2 * (T_0 + N_0 * CS) + WCD_{III} = 88192 + 436384 + 141 = 524717 cycles$

We now take in account preemptions by computing first the likelihood of them happening:

$P_0 = \frac{(WCD_I + WCD_{II} + WCD_{III})}{S_D} = \frac{(150+45023+141)}{70000} = 0.65$
$P_1 = \frac{(WCS_I + WCS_{II} + WCS_{III})}{S_D} = \frac{(143+3815+140)}{50000} = 0.08$

We proceed by calculating the influence of the preemptions of the source tile on the destination tile:

$P_{1to0} = \frac{P_1 * (T_1 + N_1 * CS)}{(T_0 + N_0 * CS)} = \frac{0.08 * (218192)}{158192} = 0.11$

The total number of expected preemptions according to the model follows:

$P_{tot} = \lceil P_0 + P_{1to0} \rceil = \lceil 0.76 \rceil = 1$

Hence, the WCMT considering preemption, according to Equation 4.10 is:

$WCMT = WCMT_{np} + P_{tot} * (T_0 + N_0 * CS) = 524717 + 218192 = 742909$

Timer Measurements Subscenario II

With this mapping and other TDM configuration, 10 simulations of the *IQZZ* migration were run and timed with the embedded processor timer. The results are shown on Table 4.15.

Average Measured MT (of 10 measurements) (cycles)	Measured $WCMT$ (cycles)
365589	607104

Table 4.15: The subscenario II measured values for the *IQZZ* migration

With a measured WCMT of 742909, the time model successfully predicted an upper bound, with an error of:

$$\Delta = 742909 - 607104 = 135805 cycles$$

$$\Delta_{\%} = (\Delta / 607104) * 100 = 22.36\%$$

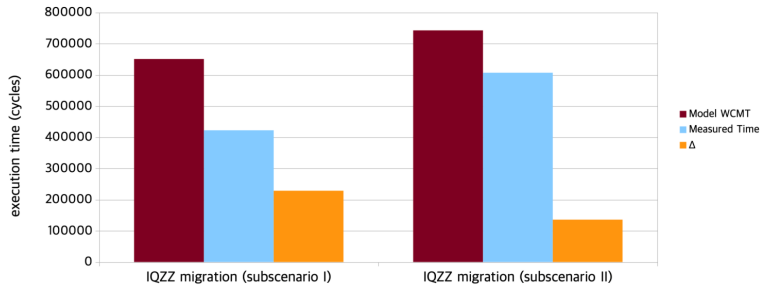


Figure 4.15: The graph in figure shows the difference between the WCMT foreseen by the model and the worst obtained measurement for the *IQZZ* task migration

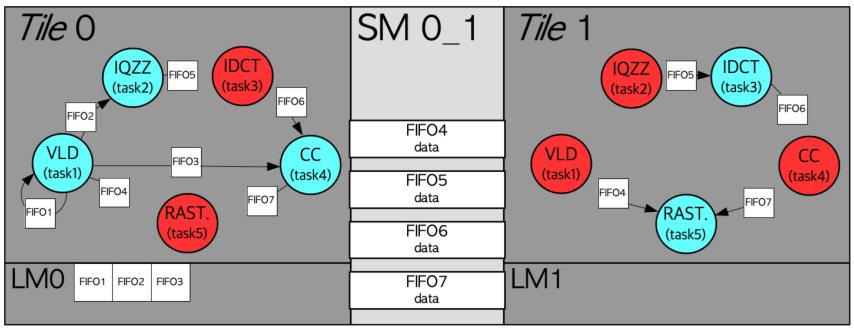


Figure 4.16: The JPEG decoder mapping after the *IQZZ* migration

Before proceeding to the subsequent migration experiment, it is interesting to notice how the model points out how heavy is the influence of the application variables on the migration phases duration. On Figure 4.17 we can see a graph that compares the phases for the *CC* and *IQZZ* migrations (the values can be observed on Tables 4.3 and 4.10); the opening and closing phases (I and III) are almost static (and neglectable), while the phase II for both source and destination is heavily influenced on the amount of data (the FIFOs) to move among memories, making this phase the main bulk of the mechanism.

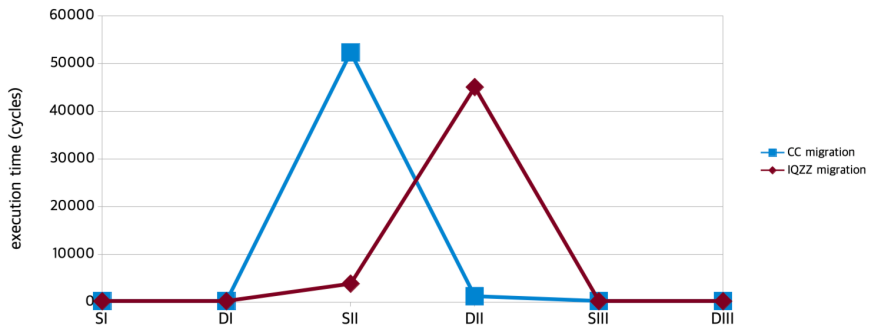


Figure 4.17: Comparison of the migration phases duration of the *CC* and *IQZZ* migrations

One can conclude that, when migrating a task, at least one of the two migration II phases, will be a bottleneck among the phases timings; this unless there are any self edges on the migrated task. In case of FIFOs with a *self edge* locality, both push and pull are expected to be influenced proportionally by the size of the FIFO. If we were to hypothetically migrate the *VLD* task now, from the resulting mapping in Figure 4.16, we would obtain the worst-case computed values depicted on the graph of Figure 4.18.

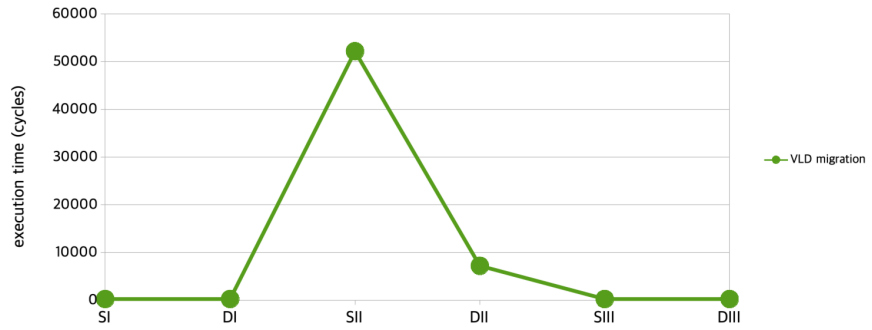


Figure 4.18: Comparison of the migration phases duration of the *VLD* migration

4.2.3 Subsequent Migrations: *CC* then *IQZZ*

We now experiment the two migrations that we saw before on the same TDM frame instances (on Table 4.16), issuing the *IQZZ* migration right after the end of the *CC* one. The scenario is now the same as the one analyzed in Section 4.1.3.

$Tile$	L_X (cycles)	S_X (cycles)
0	200000	20000
1	300000	30000

Table 4.16: The subsequent migrations scenario TDM frames of the tiles

Using the already used model heuristics, we compute the WCMT for both the migrations, obtaining the following:
$WCMT_{CC} = 1249334$
$WCMT_{IQZZ} = 1457485$

Then, we go one step further, considering the subsequent case and using Equation 4.16:

$WCMT_{CC,IQZZ} = WCMT_{CC} + (T_0 + N_0 * CS) + WCMT_{IQZZ} = 1249334 + 208192 + 1457485 = 2915011$

Now, the same scenario, analyzed and measured on the platform gave the results shown on Table 4.17.

$AveragemeasuredMT$ (of 10 measurements) (cycles)	$measuredWCMT$ (cycles)
2200720	2289979

Table 4.17: The subsequent migrations measurement results

With a measured WCMT of 2915011, the time model in its subsequent migrations expansion successfully predicted an upper bound, with an error of:

$\Delta = 2915011 - 2289979 = 625032 cycles$
$\Delta_{\%} = (\Delta / 2289979) * 100 = 27.29\%$

As we can see, the error remains around the same order of magnitude. This proves the reliability of the model, where the introduced error is only dependent on the single migrations and does not propagate in case of subsequently issued ones. The same graphic confrontation of the model computed WCMT and the worst measured time keeps the same shape as in the first two experiments, as plotted on Figure 4.19.

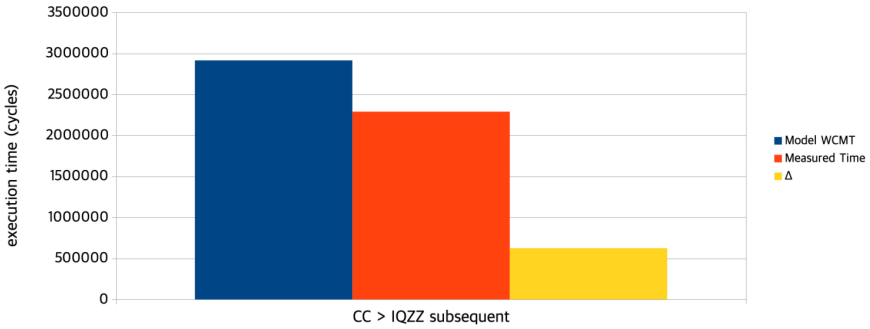


Figure 4.19: The graph in figure shows the difference between the WCMT foreseen by the model and the worst obtained measurement for the subsequent migrations of *CC* and *IQZZ*

4.2.4 Impact of δ on the application $WCRT$

If we assume the JPEG decoder in its minimal form, e.g. with the minimum possible buffer sizes, as the version shown in this thesis (with repetition vector [1,10,10,1,1]), then we can discuss the impact the partial-stalling approach on such migration experiment has on the execution flow of the tasks (and the cumulative JPEG *WCRT*). If we take, for instance, a scenario in which this migration happens during the execution of the *non-RoC* tasks *CC* or *RASTER* tasks (which is likely, since the latter is the longest in execution on our JPEG decoder with the given test image, see Appendix B), the migration of *IQZZ* (that endures 742909 in its worst-case) can start right away and finish before *RASTER* (that can have a worst-case execution time of 1453095) is complete or after by just a quantity δ which is anyway smaller than the *WCMT*; in this case the cumulative JPEG *WCRT* would be increased by 0 in the best case and a quantity δ in the other

cases, by following what has been anticipated in 1.3 about the JPEG $WCRT$:

$$WCRT_{jpeg}^{ps\_mig} = WCRT_{jpeg} + \delta$$

where $0 \leq \delta \leq WCMT_{IQZZ}$.

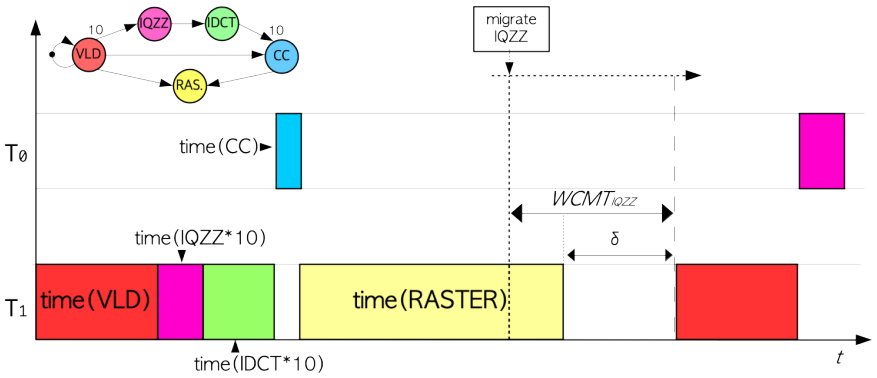


Figure 4.20: The Gantt chart in figure shows an hypothetical median case flow scenario for the *IQZZ* migration

The scenario is depicted on the Gantt chart on Figure 4.20; of course, this is not a worst-case scenario, but proves the advantage introduced by the partial-stalling approach. Apart from this median case, we can also identify a worst-case and a best case scenario for the partial-stalling migration of *IQZZ*.
In the first one, the worst-case scenario happens when the migration for *IQZZ* is issued when one of the actors that feeds the non-RoC tasks in terms of tokens, is executing, for instance *VLD*; in this case the migration happens as soon as the latter is done executing, but the non-RoC tasks (*CC* and *RASTER*) cannot execute because they are starving: the *WCMT* will be completely added to the JPEG *WCRT*, as it would happen for the general case of the full-stalling approach:

$$WCRT_{jpeg}^{ps\_mig} = WCRT_{jpeg} + WCMT_{IQZZ} = WCRT_{jpeg}^{fs\_mig}$$

The scenario is depicted on the Gantt chart on Figure 4.21.

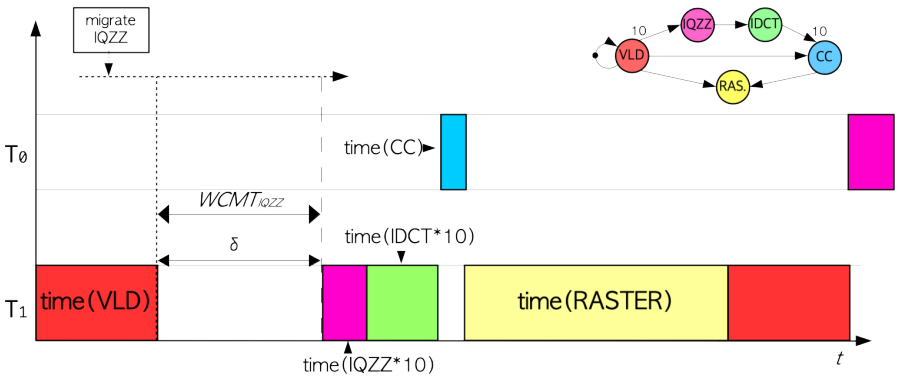


Figure 4.21: The Gantt chart in figure shows an hypothetical worst-case flow scenario for the *IQZZ* migration

In the second one, the migration of *IQZZ* is issued when *IDCT* is having the last of its 10 executions, meaning that the migration will start when this is done, and the non-RoC tasks will

have all of the necessary tokens available to execute, e.g. they will not starve. So the mechanism starts at the same moment as *CC* starts executing; in this case the JPEG *WCRT* would be not increased, if the *WCMT* is smaller than the response times of the non-RoC tasks, or would be increased by the minimal δ quantity that we will call ϵ:

$$WCRT_{jpeg}^{ps\_mig} = WCRT_{jpeg} + \epsilon$$

where

(4.17)

$$\epsilon = \begin{cases} 0 & \text{if } WCMT_{migrated\_task} \leq WCRT_{non\_RoC} \\ WCMT_{migrated\_task} - WCRT_{non\_RoC} & \text{otherwise} \end{cases}$$

with

$$WCRT_{non\_RoC} = \sum_{\tau \notin RoC, after\_migrated\_task} WCET_{\tau}$$

with *migrated_task* being *IQZZ* in our experiment and the $WCRT_{non\_RoC}$ being the sum of the *WCET*s of *CC* and *RASTER*. The scenario is depicted on the Gantt chart on Figure 4.22.

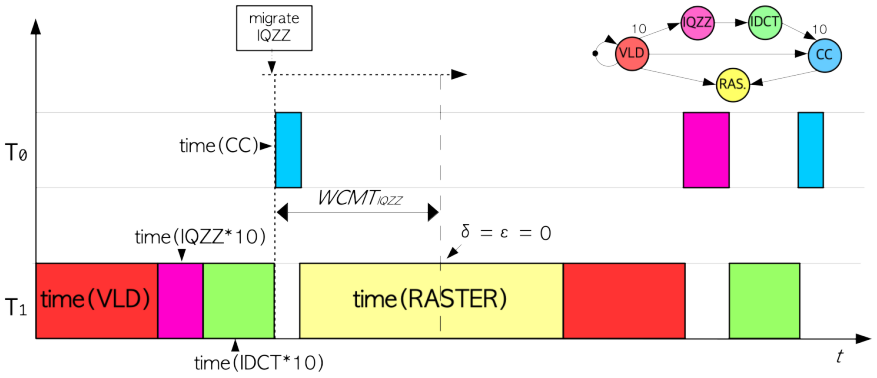


Figure 4.22: The Gantt chart in figure shows an hypothetical best case flow scenario for the *IQZZ* migration

In the scenario of our case study (*WCET*s are provided in Appendix B) with the same setting of the experimental migration of *IQZZ* (subscenario I, for instance), we would have the following values:

$WCRT_{jpeg} = 2155376 cycles$
$WCMT_{IQZZ} = 651101 cycles$

with $WCMT_{IQZZ}$ being 30.2% of the cycle size of $WCRT_{jpeg}$. We would thus define:

$$WCRT_{jpeg}^{IQZZ\_ps\_mig} = WCRT_{jpeg} + \delta$$

where $0 \leq \delta \leq WCMT_{IQZZ} = 651101$ (and, in case of best situation: $\delta = \epsilon = 0$). Meaning that the relative time overhead added to $WCRT_{jpeg}$, would oscillate between 0% (in the best case) and 30.2% (in the worst case).

4.3 Summary of the Results

From the experimentation of the time model on the case study, we have seen that the former can be used to compute an upper bound on the designed partial-stalling task migration. In the *CC* and *IQZZ* migration examples that we saw, the computed WCMTs do not differ more than 55% compared with the corresponding empirical values from the measurement results. Nonetheless, the 50+% errors are obtained when the likely number of preemption just passes the nearest integer (1.06 and 1.22) triggering particularly pessimistic estimations. The results of the measurement show in fact that the most pessimistic prediction happens when approximating the number of preemptions to the upper integer; when the likely number of preemptions gets closer to the upper integer, like 0.77 and 0.76, the final error is definitely smaller, 16.59% and 22.36% respectively. Considering that, with the used implementation, triggering the real worst-case is unlikely, we can expect the actual Δ to be even smaller than in the presented results in a real use case; we could then say that the upper bound computed with the presented time model is, yes, pessimistic, but just by a quantifiable magnitude (as we saw in Figures 4.11, 4.15 and 4.19).
Δ is clearly increased when we consider the interaction of the TDM frames of the two tiles during the migration phases. This becomes more visible if we take a look at the single migration API functions execution times (on Tables 4.4 and 4.11). Before taking in account the effect of the TDM frame interaction between the two tiles and the preemptions, the error, is kept small, with a maximum of 7.04% for the *IQZZ* migration.
In any kind of task migration where moving the state of the task to migrate is necessary, the state moving functionality is always the bottleneck. In fact, regarding the results on the migration API functions execution of our implemented mechanism, it appears clear that the bigger are the involved FIFOs and the more the execution of the *push_state* and *pull_state* will be significant, dwarfing the execution time of the other functions (as we saw with our experimentation, in Figures 4.17 and 4.18). The model computed FIFO moving times compared to the resulting ones from timer measurement, demonstrated that the bigger these are and the more the preemptions of the migration phases will be likely (in relation with the size of the SA service slot, as we saw) stretching in this way the total migration time.
From these considerations, we can affirm that the time estimation model of the presented partial-stalling migration mechanism suggests that a parameter compromise must be found depending on the use case. Given an application, the most likely tasks to be migrated of it and the frequency of migration the resource manager using the mechanism might apply, a TDM frame with a certain SA slot number can be derived to have a least possible overhead while still benefit of such migration measure.
Least but not last, we saw in the second migration example of actor *IQZZ* (Section 4.2.2), an analysis of the impact of the computed $WCMT$ on the total cumulative application $WCRT$. Here, it was proven that the time overhead δ is at most as impactful as the best possible case of the same migration with full stalling approach, following in:

$$WCRT_{app}^{ps\_mig} \leq WCRT_{app}^{fs\_mig} \tag{4.18}$$

the $WCMT_{migrated\_task}$ is therefore the upper bound on the overhead δ, introduced by the migration of the $migrated\_task$:

$$0 \leq \delta \leq WCMT_{migrated\_task} \tag{4.19}$$

with the addition that, for PSTM, the overhead δ can be also null in the best case situation.

Chapter 5

Conclusions

We have seen in this thesis a variant technique of the task migration. The dissertation began from the design of the mechanism, passing by the implementation and culminated with the experimentation; the latter was made possible through a detailed demonstration of the purposely crafted timing model. In essence, the presented work has shown that, in a predictable environment, it is possible to innovate and redesign resource management techniques while also giving an estimation of the expected performance (in this case, in terms of execution time). The designed *Partial-Stalling Task Migration* proved itself to be optimal compared to the same mechanism with the classic full-stalling approach, especially in scenarios where the target application has tasks with long execution times). Here, the long execution time of a task A, can be invested into executing a migration of another task Z that has no communication constraints with A. Compared to the state of the art seen in 2.2, this thesis work offers an alternative view of the general task migration technique, focusing on a more resource constrained environment, but, most importantly, an environment that offered beforehand and runtime measurability and therefore the possibility of estimating (with the same platform and the same application MoC) any kind of use case applied to it. Moreover, this thesis showed that building a design time estimation model with a limited error introduction, like the one that has been shown, in a system with a predictable platform and predictably modelled applications is always possible.

Future Work

Considering the outcome of this thesis, many branches can take place to open the development of new works. The main upgrades that can be implemented concern the software; it is also important to mention that calling them *upgrades* strongly depends on the scenario and use case in consideration.
The first to mention is the following. At startup time, task duplication, namely the load of all the application actors and involved FIFO channels on every involved tile, might be avoided. This happens by dynamically loading (or removing) the needed (or not) tasks and FIFOs only when a task migration happen. The advantage of this approach is that the memory footprint is reduced both locally and globally; not having a task allocated on a tile means that all the the needed control blocks for the actor and the FIFO buffers are not there. The major drawback is that, of course, each migration process will be significantly longer since, every time, a task control block and 2 or more FIFO control blocks will have to be loaded before moving the context.
Moving on, the partial-stalling characteristic of the task migration mechanism from the presented thesis might be exploited for migration-time execution optimisation. For example, both consumption and production buffers of tasks out of the *region of communication* of the task to be migrated, might be dynamic in size in order to allow these to execute as much as possible during the migration window. Even though this is a way to keep a significant throughput during migration, additional buffer backpressure, which is already a drawback of partial-stalling migrations with static buffer sizes, will be an outcome of such technique. Also, making the buffer size dynamic

might affect predictability principle; moreover, more time during migration will be spent to expand the buffers (and it will be time spent on tasks and FIFOs that are not involved in the region of communication).
The current implementation allocates all the necessary control blocks and buffers sequentially in order to avoid memory segmentation. Such approach is not optimal, for MPSoCs with reduced memory space. An improvement on this side would be to dynamically allocate portions of the CBs only when these are necessary. They are not used, in fact, for the whole execution time, especially if a task is not active on that specific PE or if a FIFO buffer is unused (which happens when both of the producing and consuming tasks are not active on the local tile). This improvement would definitely reduce the memory footprint, but it would also mean that now the memory is prone to segmentation, especially because the core of the involved CB is still allocated. Henceforth, this approach must come together with dynamic loading and memory desegmentation, meaning that more processing time will be invested despite the need of high throughput from RT applications.
Migrating more than one task ad the time, is also a possible future implementation outcome. In such scenario, an additional migration step would have to be taken to intersect the involved RoCs and verify in which way the buffers should be moved. Nonetheless, many more FIFOs will be involved, making necessary further analysis to check whether migrating one task after another with the current implementation has still less overhead than this hypothetical multiple task migration.
Another way to speedup the migration process, would be to dynamically manage the TDM frames during a migration. In case there are empty slots that are not allocated or not being used, these could be temporarily used to execute the SA and therefore the migration daemon, allocating in such way more processor cycles per each TDM cycle for the migration. In such approach, it must also be considered that by reallocating slots one must be careful to not impact the already allocated applications that are still executing; this would in fact affect both predictability and composability.
Another approach might as well take in account the interruption in the middle of the migration process due to a new decision from the resource manager. Here, instead of ending the migration regularly, and then re-migrate the involved task back, a backtracking mechanism could be implemented so that if a migration decision is retracted, this can be done right away without waiting for the migration to end plus redoing all of the steps in the opposite way to re-migrate.
To conclude, regarding the overhead estimation, in this thesis we assumed the pessimistic scenario where *input data* is available at any time for the first actor whenever *output data* for the previous input has been delivered (like a DF back edge from the last actor to the first). In a "closer to real", more precise and less pessimistic analysis, one should take in account that input data might arrive periodically and that the first actor might start computing over it as soon as it is available, if the production buffers are big enough to store new tokens before the old ones are parsed (e.g. a new iteration enters the pipeline). This might be the case of raw input coming from a camera for example. Estimating the overhead introduced by the migration in such scenario is more complex but would also be more precise, because the δ would be spread over several iterations instead of just one (as shown in Chapter 4); giving a more fine grained view of δ. We started developing the mathematics to account also this case, but since it was incomplete, it has not been included in this thesis.

Bibliography

[1] Verintec solutions, 2020. 4, 7

[2] Gabriel Marchesan Almeida, Sameer Varyani, Rémi Busseuil, Gilles Sassatelli, Pascal Benoit, Lionel Torres, Everton Alceu Carara, and Fernando Gehm Moraes. Evaluating the impact of task migration in multi-processor systems-on-chip. In *Proceedings of the 23rd symposium on Integrated circuits and system design*, pages 73–78, 2010. 15

[3] Hadi Alizadeh Ara, Marc Geilen, Amir Behrouzian, Twan Basten, and Dip Goswami. Compositional dataflow modelling for cyclo-static applications. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 121–129. IEEE, 2018. 3

[4] Oliver Arnold and Gerhard Fettweis. Adaptive runtime management of heterogenous mpsocs: Analysis, acceleration and silicon prototype. In *2014 International Symposium on System-on-Chip (SoC)*, pages 1–4. IEEE, 2014. 14

[5] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of the Design Automation & Test in Europe Conference*, volume 1, pages 1–6. IEEE, 2006. 16, 17, 24, 25, 28

[6] Eduardo Wenzel Brião, Daniel Barcelos, and Flávio Rech Wagner. Dynamic task allocation strategies in mpsoc for soft real-time applications. In *2008 Design, Automation and Test in Europe*, pages 1386–1389. IEEE, 2008. 16

[7] Guilherme Castilhos, Marcelo Mandelli, Guilherme Madalozzo, and Fernando Moraes. Distributed resource management in noc-based mpsocs with dynamic cluster sizes. In *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 153–158. IEEE, 2013. 15

[8] Taho Dorta, Jaime Jiménez, José Luis Martín, Unai Bidarte, and Armando Astarloa. Reconfigurable multiprocessor systems: a review. *International Journal of Reconfigurable Computing*, 2010, 2010. xi, 2

[9] FangFa Fu, Liang Wang, Yu Lu, and Jinxiang Wang. Low overhead task migration mechanism in noc-based mpsoc. In *2013 IEEE 10th International Conference on ASIC*, pages 1–4. IEEE, 2013. 16, 25

[10] Laurent Gantel, Salah Layouni, Mohamed El Amine Benkhelifa, François Verdier, and Stéphanie Chauvet. Multiprocessor task migration implementation in a reconfigurable platform. In *2009 International Conference on Reconfigurable Computing and FPGAs*, pages 362–367. IEEE, 2009. 17, 21, 28

[11] Marc Geilen. Synchronous dataflow scenarios. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):1–31, 2011. 3

[12] Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, Ashkan Beyranvand Nejad, et al. Virtual execution platforms for mixed-time-criticality systems: the compsoc architecture and design flow. *ACM SIGBED Review*, 10(3):23–34, 2013. 4

[13] Kees Goossens, Martijn Koedam, Andrew Nelson, Shubhendu Sinha, Sven Goossens, Yonghui Li, Gabriela Breaban, Reinier van Kampenhout, Rasool Tavakoli, Juan Valencia, et al. Noc-based multiprocessor architecture for mixed-time-criticality applications., 2017. 4, 8

[14] Shashi Kumar, Axel Jantsch, J-P Soininen, Martti Forsell, Mikael Millberg, Johny Oberg, Kari Tiensyrja, and Ahmed Hemani. A network on chip architecture and design methodology. In *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*, pages 117–124. IEEE, 2002. 2

[15] Marcelo Mandelli, Guilherme M Castilhos, and Fernando G Moraes. Enhancing performance of mpsocs through distributed resource management. In *2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012)*, pages 544–547. IEEE, 2012. 14

[16] Gabriel Marchesan Almeida, Gilles Sassatelli, Pascal Benoit, Nicolas Saint-Jean, Sameer Varyani, Lionel Torres, and Michel Robert. An adaptive message passing mpsoc framework. *International Journal of Reconfigurable Computing*, 2009, 2009. 15

[17] Andrew Nelson, Ashkan Beyranvand Nejad, Anca Molnos, Martijn Koedam, and Kees Goossens. Comik: A predictable and cycle-accurately composable real-time microkernel. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–4. IEEE, 2014. 8, 20

[18] Vincent Nollet, Prabhat Avasare, J-Y Mignolet, and Diederik Verkest. Low cost task migration initiation in a heterogeneous mp-soc. In *Design, Automation and Test in Europe*, pages 252–253. IEEE, 2005. 16

[19] Vincent Nollet, Théodore Marescaux, Prabhat Avasare, Diederik Verkest, and J-Y Mignolet. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Design, Automation and Test in Europe*, pages 234–239. IEEE, 2005. 16, 22

[20] Behnaz Pourmohseni, Stefan Wildermann, Michael Glaß, and Jürgen Teich. Predictable run-time mapping reconfiguration for real-time applications on many-core systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 148–157, 2017. 17

[21] Ahsan Shabbir, Akash Kumar, Bart Mesman, and Henk Corporaal. Distributed resource management for concurrent execution of multimedia applications on mpsoc platforms. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 132–139. IEEE, 2011. 14

[22] Shubhendu Sinha, Martijn Koedam, Rob Van Wijk, Andrew Nelson, Ashkan Beyranvand Nejad, Marc Geilen, and Kees Goossens. Composable and predictable dynamic loading for time-critical partitioned systems. In *2014 17th Euromicro Conference on Digital System Design*, pages 285–292. IEEE, 2014. 14

[23] Pranav Tendulkar and Sander Stuijk. A case study into predictable and composable mpsoc reconfiguration. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 293–300. IEEE, 2013. 15

Appendix A

Verintec MPSoC

The Verintec MPSoC is an FPGA based platform constructed with *hardware virtualization* (the used FPGA is the *PYNQ-Z2* board, shown in Figure A.2). This technique has the aim of adding a layer of abstraction between the applications and the hardware (the virtualization layer, the VEP in Figure A.1), that creates a virtualized environment for them to be deployed on. The *µkernel* (which is an entity named *hypervisor* in general virtualization techniques) is the vector that provides the virtual resources for the VEP, and therefore for the application to execute.

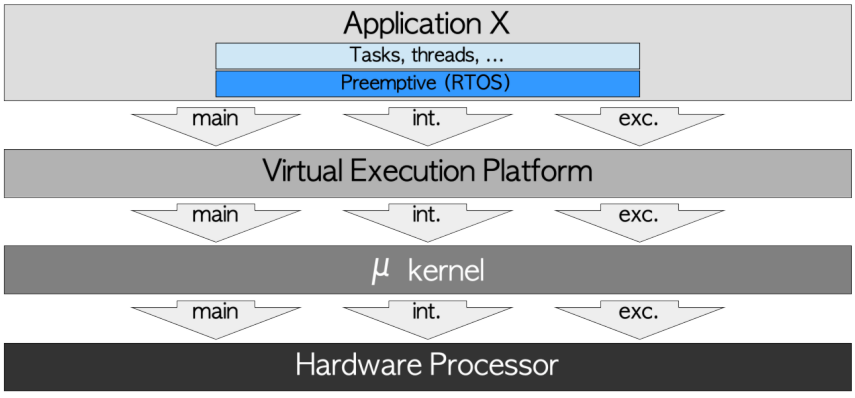


Figure A.1: A schematic of the *Verintec* virtualization

The hardware processor can run software (*main*) but also interrupt (*int.*) and exception handlers (*exc.*). These are virtualized through the virtual processor layer, therefore the application uses indirectly the hardware processor by using the virtual one.

Figure A.2: The *PYNQ-Z2* board

The memory, on the other side, is instanciated by mapping the *BRAM* blocks present on the FPGA; The routings are then statically defined by means of virtual memory mapping. The virtual processor can then access the addresses directly by having the virtual address; however, it cannot access the global physical addresses, since they might be part of a different virtualization (of another processor), notching out the *composability* principle.

Appendix B

JPEG decoder

The JPEG decoder application has been introduced in Chapter 1, where also the data-flow model has been shown (on Figure 1.5). By using the processor timer, the single JPEG decoder actor execution times where measured; the detected actor worst case execution times for the case study using the `cat.jpg` (Figure B.1) test image are shown below on Table B.1.

Actor	$WCET$ (cycles)
VLD	489888
$IQZZ$	7609
$IDCT$	10283
CC	33473
$RASTER$	1453095

Table B.1: The *JPEG decoder* actors worst case response times for decoding `cat.jpg`

Figure B.1: `cat.jpg`, the used test image to keep the JPEG decoder application running

JPEG Data-Flow Algorithm

A quick overview of the *JPEG decoder* data-flow is given, for ease of comprehension following the graph on Figure 1.5 is advised. The *VLD* actor takes as global input a block of the input JPEG encoded image, and decodes its JPEG format header data. This first task, when complete, feeds

the rest of the graph by outputting 10 FValue tokens in the direction of the *IQZZ*, a SubHeader1 token towards the *CC* and a SubHeader2 token towards *RASTER*. The *IQZZ* performs the inverse quantization on the FValues and outputs, for each of them, a PValue, that is then processed by the *IDCT* task. The latter performs a cosine transform on the PValues and outputs, for each of them, a PBlock. The *CC* can now fire, having the 10 necessary PBlocks and the SubHeader1, generating a ColorBuffer on the edge that goes to the last task: *RASTER*. This one, fires, operating the rasterization on the color converted image from the ColorBuffer with the use of the information on the SubHeader2. A binary decoded block of the initial image is given as global output.

Appendix C

Migration API

Updated Functions

Several minor/major changes have been done on some already existing `pose` library functions in order to support the migration mechanism. The most relevant ones are listed here:

- `void os_add_task(... int active ...)` This is the function that creates the `TCB` for a given task. The function initialises the relevant `TCB` flags for migration: `active` with the passed value and `suspended` with 0.

- `void os_add_fifo(... int locality ...)` This is the function that creates the `FCB` for a given FIFO channel. The function now allows the `FCB` to store the buffer and counters shared and backup allocations, so that they are retrievable when necessary (on FIFO update due to migration). The function also initialises the `locality` flag with the passed value.

- `int os_check_task_fr (int task_id)` This function determines whether a task is schedulable or not. It is used by another function, `TCB* os_schedule_task()`, to schedule the next available task. An `if` statement check inside of it has been added to make a task with id `task_id` unschedulable if it is suspended or inactive.

Migration API Functions

In this subsection more detail to the add-on functions of `pose` library is given. These are fundamental for obtaining a working non-stalling task migration.

- `void os_migration_[activate/deactivate]_task(int task_id)`
  Sets the `active` flag for the task with id `task_id` to value 1 for activation and 0 for deactivation.

- `void os_migration_[resume/suspend]_task(int task_id)`
  Sets the `suspended` flag for the task with id `task_id` to value 1 for suspension and 0 for resumption.

- `void os_migration_[resume/suspend]_adjacent_tasks(int task_id)`
  Sets the `suspended` flag, for the the tasks adjacent to the task with id `task_id` and active on the involved tile, to value 1 for suspension and 0 for resumption.

- `void os_migration_push_task_state (int task_id)`
  Initialises the push to the shared memory of the context of each FIFO that allows the task with id `task_id` to communicate (namely the FIFOs on the task consuming and producing edges).

- `void os_migration_fifo_out(FCB * fifo)`
  Pushes the context of the FIFO pointed by `*fifo` to the shared memory and updates the FIFO channel depending on the current source tile `locality` of it to the most optimal position.

- `void os_migration_pull_task_state (int task_id)`
  Initialises the pull from the shared memory of the context of each FIFO channel that allows the task with id `task_id` to communicate (again, the FIFOs on the task consuming and producing edges).

- `void os_migration_fifo_in(FCB * fifo)`
  Pulls from the shared memory the context of the FIFO pointed by `*fifo` and updates the FIFO depending on the current destination tile `locality` of it.

- `void os_migration_copy_buffer(void * dst, void * src, int size)`
  Copies a FIFO buffer of size `size` from the source pointer by `*src` to the destination pointed by `*dst`. The function is used by `os_migration_fifo_out` and `os_migration_fifo_in` to move the FIFO buffer content.

The workflow of the most complex functions, to suspend/resume the adjacent tasks and to push/pull the task state, is shown on the following listings. The code is a simplified pseudo C code for ease of depiction.

```
/*
 * RESUME ADJACENT TASKS
 * resumes adjacent tasks to the passed task
 * that are active (within the tile)
 */
void os_migration_resume_adjacent_tasks(task t) {

  /* Retrieve producing tasks to resume */
  for( j = 0; j < num_prod_fifos; j ++ ){
    /* ignore self edges, the task adjacent through a self edge is the passed task
        itself */
    if (fifo_locality != 0){
      if (task_on_edge->active){
        os_migration_resume_task(task_on_edge);
      }
    }
  }

  /* Retrieve consuming tasks to resume */
  for( j = 0; j < num_cons_fifos; j ++ ){
    /* ignore self edges, the task adjacent through a self edge is the passed task
        itself */
    if (fifo_locality != 0){
      if (task_on_edge->active){
        os_migration_resume_task(task_on_edge);
      }
    }
  }
}

/*
 * SUSPEND ADJACENT TASKS
 * suspends adjacent tasks to the passed task
 * that are active (within the tile)
 */
void os_migration_suspend_adjacent_tasks(task t) {

  /* Retrieve producing tasks to suspend */
  for( j = 0; j < num_prod_fifos; j ++ ){
    /* ignore self edges, the task adjacent through a self edge is the passed task
        itself */
```

```
    if (fifo_locality != 0){
      if (task_on_edge->active){
        os_migration_suspend_task(task_on_edge);
      }
    }
  }

  /* Retrieve consuming tasks to suspend */
  for( j = 0; j < num_cons_fifos; j ++ ){
    /* ignore self edges, the task adjacent through a self edge is the passed task
        itself */
    if (fifo_locality != 0){
      if (task_on_edge->active){
        os_migration_suspend_task(task_on_edge);
      }
    }
  }
}
```

Listing C.1: Simplified C code representation of the suspension and resumption of the τ adjacent tasks API functions

```
/*
 * MIGRATE FIFOS - DESTINATION
 * retrieves the the write/read counters
 * and the buffer content and relocates
 * the fifos on the destination tile to the
 * most optimal position according to the
 * previous locality for the passed FIFO
 */
void os_migration_fifo_in(fifo f){
  switch(fifo_locality) {
    case 0:
      // fifo is a self edge, retrieve data from shared location
      current_writec = shared_writec;
      current_readc = shared_readc;
      os_migration_copy_buffer(current_buffer, shared_buffer, total_buffer_size);
      break;
    case 1:
      // shouldn't occur.
      break;
    case 2:
      // fifo is shared, it will become of locality 1 so update the current ptrs to
          local
      writec = local_writec;
      readc = local_readc;
      buffer = local_buffer;
      // retrieve the data from the shared location
      current_writec = shared_writec;
      current_readc = shared_readc;
      os_migration_copy_buffer(current_buffer, shared_buffer, total_buffer_size);
      fifo_locality = 1;
      break;
    case 3:
      // fifo was unused, it will become of locality 2 so update the current ptrs
          to shared
      writec = shared_writec;
      readc = shared_readc;
      buffer = shared_buffer;
      fifo_locality = 2;
      break;
    default:
      break;
  }
}

/*
```

```
 * MIGRATE FIFOS - SOURCE
 * backs up the the write/read counters
 * and the buffer content and relocates
 * the fifos on the source tile to the
 * most optimal position according to the
 * previous locality for the passed FIFO
 */
void os_migration_fifo_out(fifo f){
  switch(fifo_locality) {
    case 0:
      // self edge, data must be pushed to shared location
      shared_writec = current_writec;
      shared_readc = current_readc;
      os_migration_copy_buffer(shared_buffer, current_buffer, total_buffer_size);
      break;
    case 1:
      // fully local fifo within the tile, data must be pushed to shared location
      shared_writec = current_writec;
      shared_readc = current_readc;
      os_migration_copy_buffer(shared_buffer, current_buffer, total_buffer_size);
      // fifo will become of locality 2, so update current ptrs to shared
      writec = shared_writec;
      readc = shared_readc;
      buffer = shared_buffer;
      fifo_locality = 2;
      break;
    case 2:
      // shared fifo, will become of locality 3 so do nothing
      fifo_locality = 3;
      break;
    case 3:
      //doesn't occur.
      break;
    default:
      break;
  }
}

/*
 * PULL FIFOS - DESTINATION
 * triggers the inwards migration of producing
 * and consuming FIFOs of the passed task
 */
void os_migration_pull_task_state(task t){

  /* Check producer fifos */
  for( j = 0; j < num_prod_fifos; j ++ ){
    /* self edge will appear also in consuming fifos loop so we skip it here */
    if (fifo_locality == 0)
      continue;
    os_migration_fifo_in(fifo_on_edge);
  }

  /* Check consumer fifos */
  for( j = 0; j < num_cons_fifos; j ++ ){
    os_migration_fifo_in(fifo_on_edge);
  }
}

/*
 * PUSH FIFOS - SOURCE
 * triggers the outwards migration of producing
 * and consuming FIFOs of the passed task
 */
void os_migration_push_task_state(TCB* task){
  /* Check producer fifos */
  for( j = 0; j < num_prod_fifos; j ++ ){
```

```
    /* self edge will appear also in consuming fifos loop so we skip it here */
    if (fifo_locality == 0)
      continue;
    os_migration_fifo_out(fifo_on_edge);
  }

  /* Check consumer fifos */
  for( j = 0; j < num_cons_fifos; j ++ ){
    os_migration_fifo_out(fifo_on_edge);
  }
}
```

Listing C.2: Simplified C code representation of the push state and pull state API functions

Migration API Timing Model

As described in the thesis, the derived timing model is made of two parts: one dependant exclusively on the migration mechanism and the other dependant on the TDM scheduling of the applications, including the SA (who runs the *migration daemon*). In this appendix more detail is given to the timing functions that are used to compute the migration mechanism expected upper bound at design time.
By taking another look at Figure 4.1 we can see which are the involved migration API functions to be time measured. The timing functions to compute their own upper bound singularly follow.

Suspension/Resumption and Activation/Deactivation of a task τ

The functions to suspend/resume/activate/deactivate a task τ do not involve any of the variables that influence the execution time. The involved API functions, in fact, fulfill their purpose just by manipulating the `TCB` of τ with no data involved. Therefore the dynamic part of the timing equation will be equal to zero, meaning that any task of any application, on the *Verintec* platform, will take the cycles shown here on C to be suspended/resumed.

```
time(SUSPEND(t)) =
            7 +                                  // func. overhead
            1 =                                  // execution
            8 cycles

time(RESUME(t)) =
            6 +                                  // func. overhead
            1 =                                  // execution
            7 cycles

time(ACTIVATE(t)) =
            7 +                                  // func. overhead
            2 =                                  // execution
            9 cycles

time(DEACTIVATE(t)) =
            7 +                                  // func. overhead
            1 =                                  // execution
            8 cycles
```

Listing C.3: Timing equations to compute the Suspension/Resumption API functions and Activation/Deactivation API functions worst case timings

Writing/Reading communication barrier flags

We have seen in Chapter 3 that the *migration daemons* on the source and destination tile communicate over the shared memory to assure the correct transfer of context data, and we have seen

that this happens by using *mutex* flags on the shared memory channel between the two tiles. Also in this case, their timing would be the same for any application on the *Verintec* MPSoC, since the operation is, in essence, just a write/read of a variable.

```
time(WRITE_BARRIER_FLAG) = 6 cycles

time(READ_BARRIER_FLAG)  = 12 cycles
```

Listing C.4: Flag reading and writing worst case timings

Suspension/Resumption of tasks adjacent to τ

In order to ensure that the *Region of Communication* of task τ is quiescent during the migration process, we have seen that the tasks within the region must be also suspended (and resumed once the mechanism is complete). Differently form the single suspension/resumption of τ, this time we have variables involved: the number of self edges on τ and the number of incoming and outgoing edges on τ.

```
time(SUSPEND_ADJACENT(t)) =
            37 +                                    // func. overhead
            7  +                                    // prod. loop overhead
            foreach(self_edge)
                28 +                                // loop cycle
            foreach(producing_edge)
                41 +                                // loop cycle
                ifactive(task_on_edge)
                    2                               // suspension of involved task
            7  +                                    // cons. loop overhead
            foreach(self_edge)
                28 +                                // loop cycle
            foreach(consuming_edge)
                41 +                                // loop cycle
                ifactive(task_on_edge)
                    2                               // suspension of involved task

time(RESUME_ADJACENT(t)) =
            37 +                                    // func. overhead
            7  +                                    // prod. loop overhead
            foreach(self_edge)
                28 +                                // loop cycle
            foreach(producing_edge)
                41 +                                // loop cycle
                ifactive(task_on_edge)
                    1                               // resumption of involved task
            7  +                                    // cons. loop overhead
            foreach(self_edge)
                28 +                                // loop cycle
            foreach(consuming_edge)
                41 +                                // loop cycle
                ifactive(task_on_edge)
                    1                               // resumption of involved task
```

Listing C.5: Timing equations to compute the Suspension/Resumption of τ adjacent tasks API functions worst case timings

Push/Pull of tasks τ context

The most time expensive parts of the migration are related to the push and pull of the migration mechanism (on source and destination). The long duration of these API functions happens because their dynamic part is proportional to many variables, such as: the number of self edges on τ, the number of incoming and outgoing edges on τ, call of nested functions, buffer size and token size.

```
time(PUSH_FIFO(f)) =
            25 +                                    // func. overhead
            10 +                                    // switch overhead
            switch(fifo_locality)
                case(0): 15 +                       // 0: fifo was on a self edge
                         (14*buffer_size*token_size) // 14 cycles to transfer a B
                case(1): 16 +                       // 1: fifo wads fully local
                         (14*buffer_size*token_size) // 14 cycles to transfer a B
                case(2): 15                         // 2: fifo was shared

time(PULL_FIFO(f)) =
            25 +                                    // func. overhead
            10 +                                    // switch overhead
            switch(fifo_locality)
                case(0): 15 +                       // 0: fifo was on a self edge
                         (14*buffer_size*token_size) // 14 cycles to transfer a B
                case(2): 16 +                       // 2: fifo was shared
                         (14*buffer_size*token_size) // 14 cycles to transfer a B
                case(3): 15                         // 0: fifo was unused

time(PUSH_STATE(t)) =
            20 +                                    // func. overhead
            7  +                                    // prod. loop overhead
            foreach(self_edge)
               28 +                                 // loop cycle overhead
            foreach(producing_edge)
               41 +                                 // loop cycle overhead
               time(PUSH_FIFO(fifo_on_edge))        // push involved fifo
            7  +                                    // cons. loop overhead
            foreach(self_edge)
               28 +                                 // loop cycle overhead
            foreach(consuming_edge)
               41 +                                 // loop cycle overhead
               time(PUSH_FIFO(fifo_on_edge))        // push involved fifo

time(PULL_STATE(t)) =
            20 +                                    // func. overhead
            7  +                                    // prod. loop overhead
            foreach(self_edge)
               28 +                                 // loop cycle overhead
            foreach(producing_edge)
               41 +                                 // loop cycle overhead
               time(PULL_FIFO(fifo_on_edge))        // pull involved fifo
            7  +                                    // cons. loop overhead
            foreach(self_edge)
               28 +                                 // loop cycle overhead
            foreach(consuming_edge)
               41 +                                 // loop cycle overhead
               time(PULL_FIFO(fifo_on_edge))        // pull involved fifo
```

Listing C.6: Timing equations to compute the Push FIFO state/Pull FIFO state of τ FIFO buffers API functions worst case timings