Composable and Predictable Power Management

Composable and Predictable Power Management

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.Ch.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op donderdag 6 november 2014 om 12.30 uur

door

Andrew Thomas Nelson

Master of Science
geboren te Craigavon, Verenigd Koninkrijk

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. Kees Goossens

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. Kees Goossens	Technische Universiteit Delft, promotor
Dr. Anca Molnos	CEA Leti, co-promotor
Prof. dr. Koen Bertels	Technische Universiteit Delft
Prof. dr. Ben Juurlink	Technische Universität Berlin
Prof. dr. Jose Pineda	Technische Universiteit Eindhoven
Dr. Said Hamdioui	Technische Universiteit Delft
Prof. dr. Henk Sips	Technische Universiteit Delft, reservelid

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Andrew Thomas Nelson

Composable and Predictable Power Management
Delft: TU Delft, Faculty of Elektrotechniek, Wiskunde en Informatica — III
Thesis Technische Universiteit Delft. – With ref. –

Met samenvatting in het Nederlands.

ISBN 978-94-6186-366-9

Subject headings: Power Management, Real-Time, Embedded Systems.

Cover image: From a T-shirt design worn at DATE 2012, “The balance of power is shifting. Energy just declared independence”.

Printed in The Netherlands

Acknowledgements

Trying to remember who all to thank is definitely a non-trivial part of the PhD. process. The research in this thesis has been carried out as part of a group effort to research composable and predictable System on Chips (SoCs) that we call CompSOC. I would therefore like to start by thanking my promoter Kees Goossens for not only providing guidance and encouragement throughout, but also for giving me the opportunity to be part of a group of like minded people who strive to keep applications from interfering in much the same way as some people obsessively try to prevent the different foods from touching on their plate. I also thank my co-promotor Anca Molnos for the many coffee fuelled discussions that helped ideas to form and new directions to be found.

At some point the talking has to end and the work just has to be done. Composable and predictable platforms, microkernels and operating systems do not just create themselves, and I would therefore like to thank everyone past and present who worked on the CompSOC platform, who are by now too numerous to start listing (but you know who you are). I would specifically like to thank Ashkan Beyranvand Nejad and Martijn Koedam, whom I collaborated with to create the Composable and Predictable Microkernel (CoMik) and the Predictable Operating System (POSe). Many thanks also go to the students who contributed work that assisted with my research, namely Sjoerd te Pas, Douwe van Nijnatten and Bas Vermaat.

Life is not all about work, and thankfully there has been some time to squeeze in fun in the past few years. In fact, it is hard to know where to begin with thanking all the people who helped me spend time away from behind a desk. Whether it is bar visits, Northern Ireland trips, weddings, Sinterklaas, cricket and sausage fests, or whatever, thanks again to Anca, Andreas, Ashkan, Bart, Benny, Björn, Davit, Elena, Eugenia, Golnoosh, Joyce, Jude, Karthik, Kees, Manil, Margriet, Martijn, Miran, Pao, Pavel, Radu, Richard, Sven, and the others who were accidentally omitted.

Thanks, to my family and Jo's family for their support and understanding during these last few years. Most of all, thanks to Jo for all the friendship and support she has given me during the highs and the lows of the last eleven years, of which the PhD. years must have been particularly trying of her patience. What am I going to be able to use as an excuse now?

Abstract

Composable and Predictable Power Management

The functionality of embedded systems is ever growing. The computational power of embedded systems is growing to match this demand, with embedded multiprocessor systems becoming more common. The limitations of embedded systems are not always related to chip size but are commonly due to energy and/or power constraints. While it can be possible to embed a more powerful Multiprocessor System on Chip (MPSoC), it is not always possible to provide an energy or power supply that meets its demands within the device's size and weight requirements. Power management through Dynamic Voltage and Frequency Scaling (DVFS) enables the device to be run at less than its maximum voltage and frequency, allowing high computational capability when necessary while conserving power at other times.

Embedded systems commonly perform real-time functionality. A real-time application has an associated formal model to verify that it meets its timing requirements. This formal model is used to perform a worst-case timing analysis to ensure that the application meets its requirements. These models incorporate the worst-case timing of the application's computation and communication. Timing changes due to power management must also be taken into account, complicating the verification process.

The drive for evermore functionality has led to mixed time-criticality systems, in which multiple applications of various timing criticalities share the same (hardware) resources. This complicates the verification process further as the timing interference due to shared resource contention must be taken into account. A monolithic verification effort is therefore traditionally required after system integration and must be carried out again if any modifications are made that affect the timing of any of the applications.

The problem that we aim to solve in this work is to *enable real-time applications to perform independent execution and power management without violating their timing requirements or invalidating the timing verification of concurrently executing applications*.

To solve this problem, we contribute the Composable and Predictable Microkernel (CoMik) to composably and predictably virtualise processors. When used in combination with composable and predictable memory controllers and interconnect (as provided by the Composable and Predictable System-on-Chip (CompSOC) platform), these virtual processors cannot interfere with each other's timing by even a single cycle. If whatever executes on the virtual processors (e.g. an Operating System (OS) or an application directly) has a real-time requirement, it can be verified independently of whatever executes on the concurrent virtual processors and other virtual resources.

To enable formally analysable application execution, we contribute the Predictable Operating System (POSe) that enables dataflow applications to be executed on the (virtualised) processor. We contribute a combined application and platform dataflow graph, including an algorithm to automate this process. When annotated with worst-case timings, the combined application and system graph is then used to verify that the application meets its timing requirement.

If the application's performance is better than its requirement (e.g. when the input or platform behaviour are better than worst case), its performance can be reduced using DVFS to achieve a reduction in power consumption. We contribute an off-line convex optimisation that uses the combined application and platform dataflow model to derive static run-time operating frequency levels to achieve low power consumption. The off-line technique is able to exploit static slack in the schedule, but not dynamic run-time slack due to variations in task execution times. Before dynamic slack can be used it must be possible to observe it. For this purpose, CoMik provides independent power, energy and timing accounting per virtual processor. This enables each virtual processor to be assigned individual power and energy budgets and POSe applications to be assigned timing budgets. We contribute a description and model of how energy and power budgets can be distributed between multiple virtual processors, enabling whatever executes on the virtual processor to perform composable independent power-management without affecting the ability of other virtual processors from using their entire budget allocation.

Using CoMik's accounting infrastructure, we also demonstrate how the quality of applications can be dynamically scaled to assist meeting timing, energy or power requirements. We further contribute a distributed dynamic power management policy that enables dataflow applications that are mapped onto multiple (virtual) processors to make distributed dynamic slack observations and local power-management decisions.

We demonstrate the applicability of the presented techniques on an implemented Field Programmable Gate Array (FPGA) prototype of a CompSOC hardware platform instance, using an H.263 decoder as a case-study application. We show that our techniques do not only work in theory, but that they are also implementable and implemented.

Contents

CHAPTER 1

Introduction

Embedded systems are now ubiquitous in everyday life. This is in part due to the omnipresent nature of technology that keeps us permanently tethered to the internet, such as mobile phones and tablet computers. Other uses of embedded technology are not always as obvious to the end user, e.g. by reducing usage complexity or increasing functionality of existing everyday objects, such as cars and washing machines. In these roles, it is typically used to perform real-time tasks that interact with the physical environment, e.g. fuel injection regulation in modern combustion engines. Whatever the role of the embedded system, the end user usually does not want to have to think much about it, or even know that it is there at all. To quote ubiquitous computing pioneer Mark Weiser [106]:

> '*The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.*'

While much has changed since 1991, many of the predictions contained in [106] came to fruition, and this quote is still as relevant today as when it was made.

Even though embedded systems are ubiquitous, it does not mean that there are no outstanding problems to solve. Everyone with a smartphone will be aware that one of the unpleasant consequences of this particular piece of technology "weaving" itself into our everyday lives, is the need for us to monitor and tend to its energy needs by plugging it in and recharging it all too frequently. We therefore proceed in this section to investigate the

consumer and industrial trends that led to this research into composable and predictable power management.

1.1 Consumer Trends

Demand for portable devices, such as smartphones and tablets, is increasing [93, 113]. These devices require a portable power source, which is commonly a battery that needs the user to plug it in to recharge it. Intervals between charges depend on both the capacity of the battery and rate at which the device consumes energy from the battery, i.e. its power consumption. Figure 1.1 presents the "Top 10 Smartphone Purchase Drivers" resulting from a recent consumer survey carried out by market research company IDC. This survey shows that battery life is the main purchase driver across all major smartphone operating systems. [It also indicates that weight/size is less of a concern so maybe manufacturers should think about increasing the battery capacity by putting in a larger battery.]

It is looking increasingly likely that smartphones and tablets are just the tip of the portable device iceberg. Consumer trend watchers are highlighting the rising trend of wearable devices as one of the major trends of 2014 [3, 56]. As an example, not content with the information relaying capability of their smartphone, consumers are expected to also want another smaller device on their wrist in the form of a smartwatch, which is inevitably another device that needs to be charged regularly.

Wearable devices form part of a larger embedded system trend called the "internet of things" [42, 57]. With Wireless Local Area Networks (WLANs) in almost every home, more and more consumer devices are being embedded with some form of internet connectivity, creating an internet of things. This is leading to another consumer trend in 2014 called "the internet of caring things". These are portable devices and sensors that perform a caring role, e.g. remote patient monitoring.

In summary, consumer trends are towards smarter connected portable devices. The smartphone is one such device that has been on the market for a number of years and a recent consumer survey presented in Figure 1.1 shows that battery life is a key concern when deciding which smartphone to purchase.

1.2 Industry Trends

Consumer demands for smarter devices have not gone unnoticed by various industries. For example, the automotive industry is increasingly catering for consumer gadget addictions [102]. The most interesting of these for the future direction of the automotive industry are the in-car gadgets that are automating the driving process, such as automated parking, with fully automated driving being the end goal [74].

Automation is practical for both ease of use and safety, but the demand for ever greater functionality has led to mixed criticality systems [21]. Mixed criticality generally refers to mixing applications with differing safety criticalities on the same hardware

Figure 1.1: Smartphone purchase drivers. *(Source: IDC's ConsumerScape360, by Michael DeHart; via: Twitter, Francisco Jeronimo,* `http://t.co/AS2VjrEF2x`*, 12th May 2014)*

platform. Reduced area, weight, power and cost of materials are all reasons to share hardware resources between applications, but potential interference due to resource sharing complicates the verification of safety critical applications. For example, uncharacterised non safety critical applications that share the same resources as a safety critical real-time application could cause unbounded timing interference on the shared resources making it impossible to guarantee the safety critical application's timing behaviour.

In this thesis, we focus on the particular problem of mixed time-criticality that forms part of the larger problem of mixed criticality in general. Real-time applications have either soft, firm or hard timing requirements, ranging from lowest to highest strictness of the requirement [14]. Safety critical real-time applications have hard timing requirements [22], as failure to meet the requirement may cause physical harm to the system and its surroundings, e.g. an electronic braking system. Firm real-time applications have strict timing requirements with the value of the application's output deteriorating rapidly after the requirement deadline, but do not pose a risk of physical harm, e.g. a software defined radio. Soft real-time applications have timing requirements but the value of the output does not deteriorate so rapidly after a missed deadline and can therefore be deemed functional while occasionally missing deadlines, e.g. an MP3 decoder. Non real-time applications are applications with no real-time requirement. In a mixed time-criticality system, applications of various timing requirements share the same platform resources.

Multi-core platforms compound the problem of verifying real-time applications in a mixed time-criticality system further, as applications can be mapped across multiple processors sharing not only the processors but also the interconnect that enables the processors to communicate. Temporal isolation is a simplification strategy that allows concurrent applications to execute on shared resources with statically bounded interference [75]. The timing of applications can therefore be verified in isolation as the worst-case interference is known in advance.

Virtualisation has been used for many years in desktop and server machines to share resources, and is becoming increasingly common for embedded systems [45, 46]. Virtualisation simplifies resource sharing by enabling applications and Operating Systems (OSs) to execute on virtual machines as if they were being executed on a physical hardware platforms, simplifying the programming/porting process. Combining virtualisation with temporal isolation creates virtual machines that are temporally isolated from each other. Real-time applications can therefore be mapped onto a virtual machine and verified independently from concurrent applications.

Power management of embedded systems enables applications to reduce the power consumption of the platform by performing Dynamic Voltage and Frequency Scaling (DVFS) or temporarily shutting down parts of the platform [15]. A reduction in power consumption is achieved in exchange for an increase in computation time. This complicates providing timing guarantees for real-time applications as the timing behaviour of the application depends on used DVFS levels. While much work has been carried out on power management of real-time systems [7, 47, 112], the power management of mixed time-criticality systems is still an open issue. This is also the case for embedded virtualisation. While

power management for virtual platforms is suggested in [46] for the context of migrating virtual machines off of a core to shut it down, performing virtualised power management using DVFS is unresolved.

In summary, there is a continuing trend for more functionality provided by embedded systems. Resource sharing enables functionality to be added at lower cost than additional dedicated hardware. This leads to an increase in mixed time-criticality systems. Temporal isolation simplifies the verification of real-time applications on shared resources while virtualisation simplifies programming shared resources. Power management of mixed time-criticality platforms and independent DVFS in virtual platforms are still open issues.

1.3 Problem Statement

Power management of applications running on embedded systems is important due to design considerations and consumer demands. These two factors are also driving the creation of mixed time-criticality systems. Verifying the timing behaviour of real-time applications in these systems is non-trivial, due to the possibility of inter-application timing interference. As power management through DVFS changes application timing, verifying application timing and bounding/limiting its effects on concurrent applications is a problem.

The problem that we aim to solve in this work is therefore to *enable real-time applications to perform independent execution and power management without violating their timing requirements or invalidating the timing verification of concurrently executing applications*.

1.4 Requirements

Given our problem statement, we break our proposed solution down into the topics of:

- **Composability:** The ability to share resources within strict budgets such that a system can be composed of independently executing and verifiable applications on Virtual Platforms (VPs).
- **Predictability:** The ability to abstract a VP to a formal model to predict behaviours.
- **Low Power:** To enable applications to reduce their power consumption.

We introduce these topics in the following sections.

1.4.1 Composability

Composability enables concurrent applications to share resources within set limitations/budgets. For timing, composability is sometimes referred to as temporal isolation. In

this thesis, we refer to the strictest form of composability where budgets for time, space and energy resources are strictly enforced, e.g. when time sharing a memory resource using Time Division Multiplexed (TDM) arbitration, each composable virtual processor gets a time allocation in the schedule and a space allocation (dedicated memory region), with the boundaries of that allocation being strictly enforced so that each thread can only use its allocated time/space and no more than that. By composably arbitrating all shared resources using (statically allocated) predictable budgets per composable thread, applications are cycle-accurately temporally isolated, i.e. they cannot interfere with each other's timing behaviour by even a single cycle.

The energy/power supply of an embedded system is also typically a shared resource, i.e. a battery. Our problem statement calls for independent power management and hence also composable sharing of the energy/power resource. Applying the same principles as temporal composability, each application must receive a statically allocated strictly enforced energy/power budget. Each application must be able to use this budget in its entirety, regardless of the behaviour of concurrent applications.

1.4.2 Predictability

Predictability is essential to give timing guarantees for real-time applications. We consider an application to be predictable if a formal abstraction can be created that can be used to derive predictions about the application's timing behaviour prior to execution. The accuracy of the formal model is determined by how close the predictions are to reality. For real-time applications, it is important that the model is not only accurate but also temporally conservative, i.e. that the timing of the application in reality can only be equal to or better than the prediction by the model.

For an application to be predictable it must be executed on a predictable hardware platform. It must therefore be possible to formally model the behaviour of the hardware on which the application executes. On a multi-core memory mapped platform with distributed shared memory, this means that it must be possible to conservatively model the application's execution on the processors and accesses to and from local and remote shared memories. The user- and system-software, processors, interconnect, memory controllers, memories and any other peripherals that the application might interact with must therefore all have deterministic timing bounds. The composable arbitration of resources must also be carried out in a deterministic manner so that it can be formally modelled.

1.4.3 Low Power

The purpose of power management is to reduce power and/or energy consumption. DVFS is a common mechanism that enables a (usually) monotonic trade-off in performance for a reduction in power consumption, i.e. a computation can be performed with lower power consumption, but the computation will take longer. To be able to use the DVFS mechanism while providing timing guarantees for real-time applications, it must therefore

be possible to incorporate the timing effects of the DVFS mechanism into the application's timing formalism.

Given our problem statement, multiple applications can share the processor. The interference caused by each application's power management strategy on the other applications must therefore be bounded and modelled as part of the timing formalism of the applications. This is not practical, as the bounds on the interference of all other concurrent applications would have to be known at design time. Using our strict definition of composability, applications know their budget allocation at design time that this budget is guaranteed to be interference free.

To achieve this, not only the arbitration of the DVFS mechanism must be composable, but also its effects. This means that a DVFS level set by one application cannot interfere with or be altered by other applications. The effects of DVFS also concern energy/power budget depletion rate. Energy and power budgets allocated to applications must also be guaranteed and strictly enforced, meaning that budgets must be able to be used in their entirety, regardless of the behaviour of other applications.

1.5 Contributions

We proceed to outline the set of contributions made by this thesis that meet the requirements of our problem statement. Figure 1.2 presents a high level overview of our proposed solution to the problem statement in Section 1.3.

1.5.1 Architecture

To enable composable and predictable execution of applications, we contribute the following to our Composable and Predictable System-on-Chip (CompSOC) platform's software stack:

- The Predictable Operating System (POSe) OS implements the dataflow Model of Execution (MOE) enabling applications to be structured, executed and analysed as dataflow graphs.
- The Composable and Predictable Microkernel (CoMik) microkernel[1] composably and predictably virtualises a physical processor into multiple virtual processors, enabling cycle-accurate composable sharing of the physical processor.

We contribute the following hardware to enable the CompSOC platform's composable and predictable software stack:

- Composable processing tile memory architectures. We propose multiple memory architectures that enable temporally composable sharing of local scratch pad memories with Direct Memory Access (DMA) support.

[1] The work on CoMik was carried out in collaboration with Ashkan Beyranvand Nejad [16].

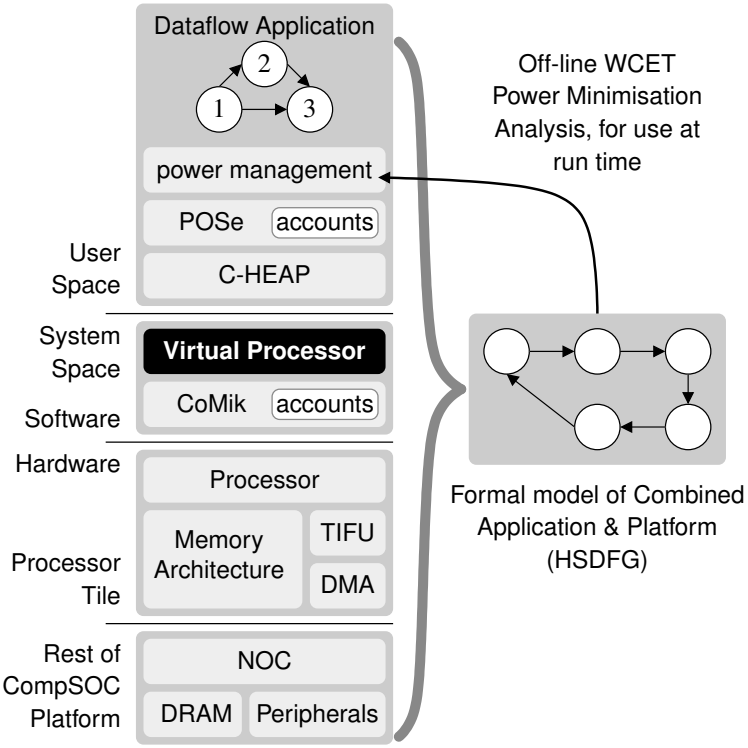


Figure 1.2: Composable power management for real-time dataflow applications.

- The Timer-centric Interrupt and Frequency Unit (TIFU)[1] is a hardware module that each processor tile has. It is used to coordinate interrupt and DVFS changes at programmable times. This module assists CoMik to provide virtualised interrupts and DVFS configuration, enabling each virtual processor to have an independent DVFS level.

1.5.2 Power Management

To achieve our objective of independent power management for real-time applications, we contribute the following

- A method to allocate composable per application energy/power budgets, such that each application can use its entire allocated budget regardless of the behaviours of other applications.

[1] The work on the TIFU was carried out in collaboration with Ashkan Beyranvand Nejad [16].

- An analysis method to derive low-power conservative static Voltage and Frequency Scaling (VFS) levels for real-time dataflow applications executing on the CompSOC platform.
- An analysis method to derive low-power conservative DVFS levels that are selected at run-time using a distributed closed control loop, for real-time dataflow applications executing on the CompSOC platform.
- An investigation into the use of quality scaling mechanisms in adaptive applications to assist run-time power management by trading a reduction in quality for a reduction in execution time. We carry out this investigation for an adaptive H.263 decoder application.
- A case study analysis of our static and dynamic power management techniques applied to an H.263 decoder.

1.5.3 Formalism

To be able to provide real-time guarantees, and therefore to also perform DVFS while guaranteeing not to violate requirements, we contribute the following:

- Dataflow timing abstractions for multiple inter-core First In First Out (FIFO) configurations, using the C-HEAP communication protocol.
- A method to generate combined application and platform Homogeneous Synchronous Dataflow Graphs (HSDFGs) that can be used for worst case analysis when the actors are annotated with their Worst-Case Execution Times (WCETs).

1.5.4 Summary

As illustrated in Figure 1.2, our contributions enable real-time dataflow applications to execute on composably virtualised hardware, including energy and power budgets, e.g. a virtualised battery. This enables each dataflow application to be independently formally modelled as a combined application and CompSOC platform HSDFG. These independent models enable worst case timing analysis of the application when actors are annotated with WCETs making them suitable for deriving timing guarantees for real-time applications. We use the application's formal model to derive suitable low-power DVFS levels that are guaranteed not to violate the application's real-time requirement. We contribute two power management techniques. One of our techniques derives static conservative VFS levels. Our other technique derives a set of conservative DVFS points that depend on application progress using a run-time power management closed control loop to monitor the application's progress and select an appropriate conservative DVFS operating point. We demonstrate our power management techniques applied to an implementation of an H.263 decoder on an Field Programmable Gate Array (FPGA) prototyped instance of the CompSOC platform.

1.6 Overview

The remainder of this thesis is organised as follows. We present details of the composable and predictable CompSOC platform in Chapter 2. We describe the hardware platform together with the dataflow formalism that we use to abstract the platform's timing. We conclude Chapter 2 by describing how the application and CompSOC hardware platform can be modelled together as an HSDFG. In Chapter 3, we give an overview of the power model we use to account for processor power consumption on an FPGA prototyped implementation of the CompSOC platform. We also describe how energy, power and time are composably accounted for using CoMik and POSe. This also includes a description of how energy/power budgets are composably allocated in the CompSOC platform, so that they can be used in their entirety regardless of the behaviour of concurrent applications.

We proceed in Chapter 4 to propose an off-line power management method to derive low power DVFS levels per core for a real-time dataflow application. The derived DVFS levels are guaranteed not to violate the application's timing requirements. The method we propose in Chapter 4 cannot make use of dynamic variations in task execution time. In Chapter 5, we therefore propose a run-time power management method that makes use of dynamic variations in task execution time to lower the power consumption of the application further than achievable with our static power management technique. We also investigate the use of quality scaling mechanisms in adaptive applications to trade a decrease in quality for a decrease in execution time, thereby enabling further reductions to the power consumption.

Having described our methods in the previous chapters, we apply them to an H.263 video decoder application in Chapter 6. We show that our power management techniques are not only theoretical, but can be implemented in practice using an FPGA prototyped CompSOC platform. We bring this thesis to an end in Chapter 7, by making conclusions about the work in this thesis and describing potential avenues for future research.

CHAPTER 2

The CompSOC: Mixed Time-Criticality Platform

Real-time applications require guarantees that they will meet their timing requirements. To be able to do this, the platform on which they execute must be predictable. This is further complicated whenever real-time applications share resources with other non real-time applications. The contention for shared resources must be taken into account when providing guarantees. This is not trivial, especially on mixed-time criticality systems where real-time and non real-time applications can share resources. The behaviour of non real-time applications might not be predictable, and therefore the contention for the resource might be unpredictable too.

In this chapter, we describe the CompSOC platform that provides a composable and predictable platform for the purpose of running applications of mixed time criticality. This is achieved by CompSOC's hardware and software framework, as illustrated in Figure 2.1. Dataflow applications are executed on the platform using the POSe OS and composably isolated using the CoMik microkernel. The CompSOC hardware platform enables worst-case bounds to be derived for both computation and communication. For real-time applications, a worst-case timing analysis is used to verify that the application meets its timing requirements. In Section 2.1 we explain how dataflow is used to analyse the behaviour of real-time applications.

The worst-case timings of the application depends on the hardware resources to which it is mapped. In Section 2.2, we describe the CompSOC hardware platform, an instance of which is illustrated in Figure 2.2a. We explain how the application's mapping to

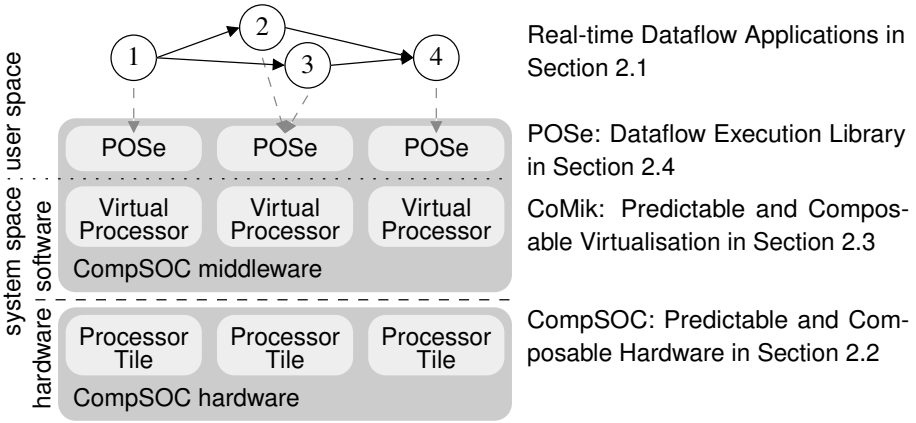


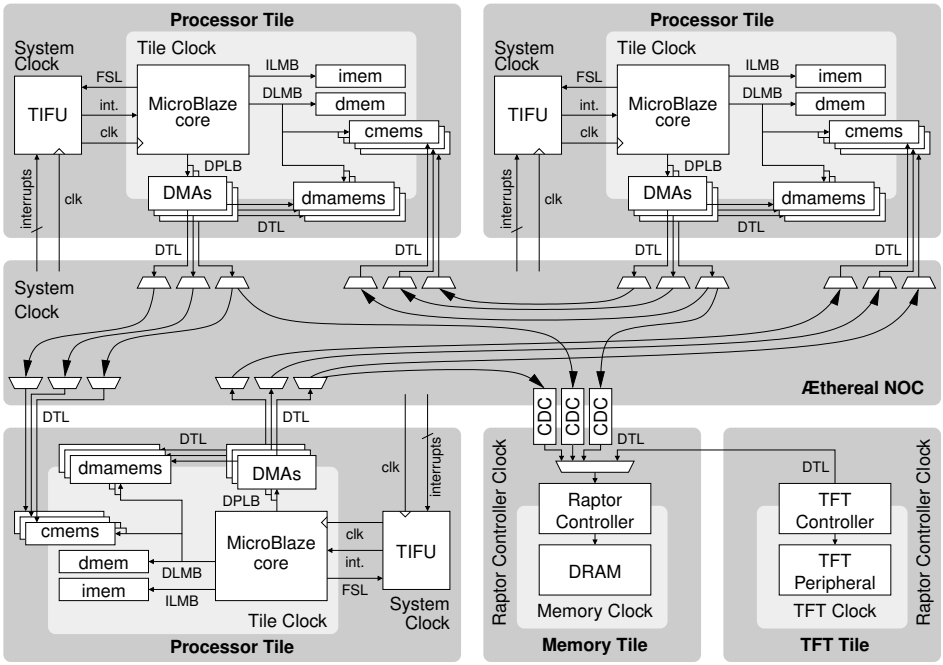
Figure 2.1: CompSOC high-level overview.

CompSOC's hardware resources affects its timing and how we take this into account in the application's dataflow model. Both the dataflow application's tasks and communication require physical resources and take time to complete. In Section 2.2.3, we explain how a mapped C-HEAP FIFO is modelled as a dataflow graph that when

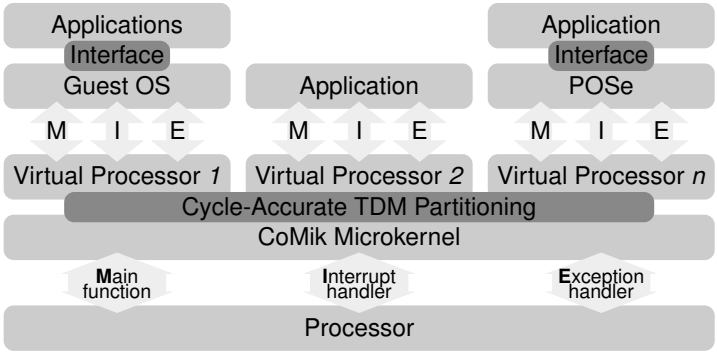
To enable composable and predictable communication between processing cores, we contribute composable memory architectures to the CompSOC processing tile in Section 2.2.1. We describe how the communication and memory architecture of the tile is configured to enable composable inter-tile communication, presenting two composable configurations and explain the trade-offs between them. To support precisely timed virtualisation of DVFS management and interrupts we further contribute the TIFU to the CompSOC processing tile and describe it in detail in Section 2.2.4.

Composable virtualisation provides temporal isolation between applications, enabling mixed time-criticality applications to share physical resources [31, 34]. To achieve this, we contribute the CoMik microkernel in Section 2.3. CoMik divides the processor into multiple composable virtual processors enabling applications of various criticalities to co-exist on the same physical processor without interfering by even a single cycle. This is achieved using TDM scheduling of virtual processors on the physical processor. By modelling the TDM scheduling as a dataflow latency-rate server, the composable virtualisation can be incorporated into the application dataflow graph for temporal analysis.

CoMik allows the user a choice of which OS and/or Model of Computation (MOC) to use. It provides an interface similar to the physical processor, taking a pointer to a main function, and interrupt handler and an exception handler, as illustrated in Figure 2.2b. To enable the execution of dataflow applications, we contribute the POSe Real-Time Operating System (RTOS) in Section 2.4. This is achieved by structuring applications following the dataflow paradigm, with tasks as actors that communicate via C-HEAP FIFOs as edges. We further describe how the POSe MOE is modelled as a dataflow graph



(a) CompSOC hardware platform instance.



(b) CompSOC software hierarchy.

Figure 2.2: CompSOC platform overview.

enabling the POSe OS to be incorporated into the application dataflow graph for temporal analysis.

After describing how real-time applications can be modelled and analysed as dataflow graphs (Section 2.1 and Section 2.4, respectively), and presenting the CompSOC platform (Section 2.2) that enables dataflow applications to be executed, in Section 2.5 we contribute an algorithm that takes the dataflow model of a mapped application that is mapped onto the CompSOC platform, and makes a combined application and CompSOC platform dataflow graph.

2.1 Real-time Dataflow Applications

Real-time applications have timing requirements that must be met. To provide assurances, the timing of the applications must be analysed. This is commonly achieved by formalising applications using a MOC. Many types of MOC exist, with varying degrees of analysability and expressiveness. The CompSOC platform is able to execute applications using the Kahn Process Network (KPN), Cyclo-Static Dataflow (CSDF), Synchronous Dataflow (SDF) and Homogeneous Synchronous Dataflow (HSDF) MOCs, ranging from most to least expressive. The choice of MOC is a trade-off, as generally the more analysable a MOC is then the less expressive it is also [14]. It is harder to express an application in a more restrictive MOC. As such, the most analysable MOCs are also the most restrictive to program for, adding to design time.

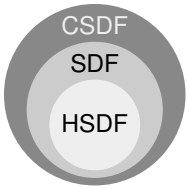
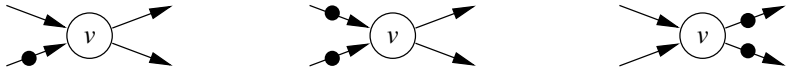


Figure 2.3: Dataflow expressiveness.

In this thesis, we focus solely on the dataflow MOCs due to their analysability. Figure 2.3 illustrates the hierarchy of expressiveness of the dataflow MOCs. Note that KPN is not a dataflow MOC.

SDF is more expressive than HSDF, and all HSDFGs are Synchronous Dataflow Graphs (SDFGs). Similarly, CSDF is more expressive than SDF, and all SDFGs are Cyclo-Static Dataflow Graphs (CSDFGs). It is shown in [14] that all CSDFGs can be represented as timing-equivalent HSDFGs. This means that applications can be modelled as either of the more expressive SDF or CSDF MOCs while still being temporally analysable using the HSDF formalism. The techniques presented in this thesis can be used on applications that follow the HSDF MOC, and on applications that follow other dataflow MOCs that are translatable to HSDF for temporal analysis, such as SDF and CSDF.



(a) Insufficient tokens. **(b)** Firing enabled. **(c)** Firing complete.

Figure 2.4: HSDF actor firing.

An HSDFG is a graph, with actors represented by the graph vertices and FIFO communication between actors represented by directed edges. Data is communicated along the edges of the graph in atomic tokens, represented as a black circle on the edge. Edges have an infinite token capacity, meaning that the number of tokens on an edge does not inhibit the production of more tokens on that edge. HSDF actors require a single token on each incoming edge before they are able to fire, i.e. the task has the data required for execution. This is illustrated in Figure 2.4 where in Figure 2.4a actor v is unable to fire as one incoming edge has no tokens, whereas actor v is enabled to fire in Figure 2.4b as there is one token on each of its incoming edges. Depending on the scheduling scheme actors can fire (at the earliest) as soon as they are able (Self-Timed Schedule (STS)) or possibly defer firing until a later moment, e.g. when using a Static-Periodic Schedule (SPS). When an actor fires, a token is consumed from each incoming edge. Upon completing execution, one token is produced on each outgoing edge, as illustrated in Figure 2.4c.

2.1.1 Resource Constraints

To execute a dataflow application, it must be mapped onto a suitable platform, such as the CompSOC platform. Whereas the only constraint on actor firing in an HSDFG is sufficient token availability, mapped dataflow applications also have resource constraints in addition to data dependencies. These implementation constraints are due to finite resource capacity, e.g. FIFO buffer capacity. In order to take them into account within the HSDFG framework, they must be modelled as a token dependency. This is achieved by adding additional HSDFG edges and tokens to the original application HSDFG.

Mapping

The nature and timing of the resource constraint depends on the resource to which the dataflow component is mapped. Dataflow applications consist of actors that are implemented by computational tasks and edges that are implemented by C-HEAP FIFO buffers. On a multi-processor system, individual tasks are mapped onto a processor. The worst-case execution time of the task depends on the type of processor to which it is mapped and the frequency at which the processor operates. The worst-case work of the task is the worst-case number of cycles that must execute on the processor before the task is complete, and therefore does not depend on the processor's frequency. This makes worst-case work a useful measurement of task execution duration for DVFS management.

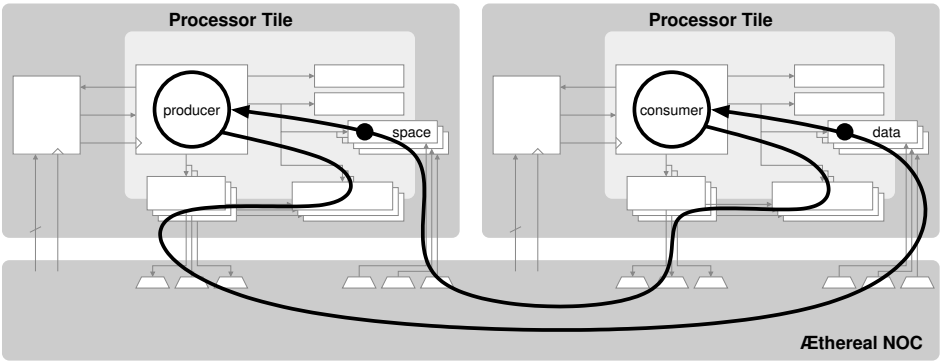


Figure 2.5: A C-HEAP FIFO mapping.

The application's communication must also be mapped to platform resources, as illustrated in Figure 2.5. The dataflow edge that is implemented as a C-HEAP buffer can be mapped in the local scratchpad memory of either the producing or consuming processing tile, or in a larger shared memory that is accessible via the Network on Chip (NoC). When accessing memory locations via the NoC, the communication actions must be mapped onto hardware communication resources, such as DMA modules and NoC connections. As with the computational tasks, the use of these resources costs time. The dataflow MOC does not allow graph edges to be annotated with an execution duration. For analysis, additional actors are therefore added to the edges of an application's dataflow graph, that represent the duration of the operation on the communication resources. This is explained in more detail for C-HEAP communication in Section 2.2.2.

Auto-concurrency constraint

An HSDF actor can execute infinitely many times auto-concurrently, with token availability being the only constraint, e.g. the actor illustrated in Figure 2.6a has no incoming edges and may fire infinitely many times auto-concurrently (at the same time in parallel) as it is always enabled to fire. For instance, depending on the scheduler, a task executing on a processor might be prevented from, or have limited auto-concurrency.

(a) Infinitely auto-concurrent. **(b)** Non auto-concurrent.

Figure 2.6: HSDF actor auto-concurrency.

An auto-concurrency constraint on an actor can be modelled in a HSDFG by adding an additional self-edge to the actor concerned. A self-edge is a directed edge where the

source and destination actor are the same. It is governed by the same rules as other HSDF edges, and therefore in order for the actor to fire, at least one token must be present on the edge before the actor is enabled to fire. As the actor will only produce a token on this edge after it has consumed one from it, auto-concurrency is limited by the number of initial tokens on the edge, e.g. one initial token completely prevents auto-concurrency, while two tokens allows two iterations of the actor to fire concurrently.

The scheduling schemes described in this thesis do not schedule application tasks auto-concurrently. They are therefore modelled by actors with a self-edge containing a single initial token. For simplicity of illustration, in this thesis self-edges are omitted from actors in HSDFGs diagrams, unless required in the diagram to differentiate actors without an auto-concurrency constraint, or it is otherwise stated.

Single-Resource Concurrency Constraint

When mapping an HSDF application to an actual Multiprocessor System on Chip (MPSoC) platform, it is not always possible, or desirable, for each actor to be mapped onto its own processor. As such, the application's tasks will have to share the processor that they are mapped onto. Instructions from only one task at a time can be processed on a single threaded processor, so scheduling must be used to arbitrate which task gets to execute and at what time.

In this thesis, task-level scheduling within an application is assumed to be performed co-operatively. A task that is executing, cannot be pre-empted by another task. As such, only one task belonging to an application can execute per processor at any given time. Under a STS, actors start firing as soon as they have sufficient tokens to do so, but multiple actors on the same processor may concurrently have sufficient tokens to fire, e.g. Figure 2.7a illustrates a single resource on which all three actors are able to fire concurrently. The single-resource constraint is therefore not captured by the STS.

(a) Concurrently enabled actors.

(b) Cycle of resource constraint control edges.

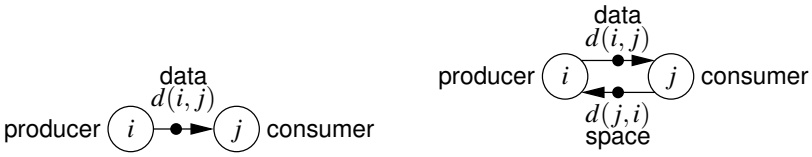
Figure 2.7: Single-resource constraint control edges.

Token availability is the only method to control the firing of HSDF actors that are executing following a STS. The single resource constraint is therefore taken into account using (so called) "control" edges in the HSDFG. For the example illustrated in Figure 2.7a, control edges are added in Figure 2.7b to ensure that only one actor can fire at a time. The control edges form a cycle, with one of the edges containing a single initial token, ensuring that only one actor at a time is able to fire. The actors therefore fire following

a Static-Order Schedule (SOS), starting with the actor that has the initial token on its incoming control edge (actor 1 in the example).

Finite FIFO Capacity

A FIFO buffer that is mapped onto a hardware platform has a finite capacity. The HSDF edges that represent inter-actor FIFOs have an infinite token capacity. A FIFO buffer with finite capacity is modelled using two directed edges. One edge representing the direction of data transfer between the two actors. The other edge represents the direction of transfer of free space in the buffer.



(a) Infinite buffer capacity. **(b)** Finite buffer capacity.

Figure 2.8: HSDFG FIFO buffer capacity representation.

2.1.2 HSDFG and Schedule Formalism

We continue by presenting the background material on HSDFG schedule formalism that enables temporal analysis of dataflow applications. In this section, we present schedule admissibility and analysis techniques for STS, Worst-Case Self-Timed Schedule (WCSTS) and SPS schemes. These are applied in Chapter 4 and Chapter 5 to enable conservative DVFS for real-time applications.

The firing time of an actor depends on the availability of tokens on its incoming edges and the scheduling scheme. Under a Self-Timed Schedule (STS), the actors start firing as soon as there are enough tokens to do so. Whereas in a Static-Periodic Schedule (SPS), actors start firing at static periodic intervals that are derived to ensure token availability. CompSOC executes dataflow applications following a STS. The actor firing times in an STS schedule depends on actual actor firing durations. For dataflow applications, these times depend on task execution times that can be data-dependent.

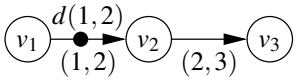


Figure 2.9: HSDFG composed of actors, edges and initial tokens.

A HSDFG G, such as that illustrated in Figure 2.9 is represented using the tuple (V,E,t,d). V is the finite set of annotated vertices. The vertices are dataflow actors. E is

the finite set of annotated directed edges that connect the vertices. An edge is represented by the tuple $(i, j) \in E$ where $i \in V$ is the actor *producing* tokens on the edge and $j \in V$ is the actor *consuming* tokens from the edge. The execution time of the k'th iteration $k \in \mathbb{N}$ of an actor i is given by $t(i,k)$, with $t : V \rightarrow \mathbb{R}^+$. The initial token occupancy of an edge (i, j) is given by $d(i, j)$, with $d : E \rightarrow \mathbb{N}$.

We use a schedule notation based on the one presented in [76]. The start time of the k'th iteration of actor i firing is given by $s(i,k) \in \mathbb{R}^+$. The worst-case execution time of a task is given by $t(i)$, as it does not depend on the iteration k. The worst-case execution time $t(i)$ satisfies, $\forall k \in \mathbb{N} : t(i,k) \leq t(i)$.

Using this formal representation, it is possible to express the scheduling schemes as a set of constraints that must be adhered to if the scheme is to be admissible.

Self-Timed Schedule (STS)

In a STS, actors fire as soon as they are enabled. It has been shown in [38] that an STS schedule is admissible (i.e. valid) if and only if Equation 2.1 holds for every edge $(i, j) \in E$ in the HSDFG.

$$s(j,k+d(i,j)) \geq s(i,k)+t(i,k) \tag{2.1}$$

For a single edge in isolation, such as that illustrated in Figure 2.8a, constraint Equation 2.1 ensures that the starting time of $s(j,k+d(i,j))$ an actor j consuming a token from the edge cannot be earlier than the start time of the actor i that produced that token $s(i,k)$ in addition to the time that it takes for the producing actor to complete its firing iteration $t(i,k)$. The token being consumed in graph iteration $k+d(i,j)$ is produced $d(i,j)$ iterations earlier, as the initial tokens on the edge are consumed first. All admissible schedules (not just STS) must satisfy Equation 2.1.

Monotonicity

A dataflow application has tasks modelled as actors that can have variable execution times. Due to the monotonicity of the dataflow MOC, a shorter execution time can only cause an earlier enabling of subsequent actors, and similarly a longer execution time can only cause a later enabling of subsequent actors. This is important for observing and using slack for power management, as an observed earlier start can be conservatively used to lower the DVFS level. From Equation 2.1, it follows that dataflow graphs execute monotonically. In the context of the dataflow MOC, monotonicity of execution means that an earlier enabling of an actor firing cannot cause a later enabling of a subsequent actor firing, and similarly a later enabling of an actor firing cannot cause an earlier enabling of a subsequent actor firing, i.e. from Equation 2.1, it follows that:

$$s_l(i,k) > s_e(i,k) \Rightarrow s(j,k+d(i,j)) \geq s_l(i,k)+t(i,k) > s_e(i,k)+t(i,k) \tag{2.2}$$

where $s_e(i,k)$ and $s_l(i,k)$ are earlier and later enablings of iteration k of actor i. Under STS, actors start firing as soon as they are enabled. The enabling time $s(j,k+d(i,j))$ of actor j that consumes tokens in iteration $k+d(i,j)$ produced by actor i, cannot be any earlier than the completion time of actor i in iteration k. For an actor firing duration of time $t(i,k)$, an earlier enabling $s_e(i,k)$ of actor i causes it to complete iteration k earlier than a later enabling $s_l(i,k)$. The earlier enabling $s_e(i,k)$ of actor i therefore cannot cause a later enabling $s(j,k+d(i,j))$ of actor j, and similarly a later enabling $s_l(i,k)$ of actor i cannot cause an earlier enabling of actor j. From Equation 2.1, it follows that:

$$t_l(i,k) > t_s(i,k) \Rightarrow s(j,k+d(i,j)) \geq s(i,k)+t_l(i,k) > s(i,k)+t_s(i,k) \qquad (2.3)$$

where $t_s(i,k)$ and $t_l(i,k)$ are shorter and longer firing durations of iteration k of actor i. For actor enabling time $s(i,k)$, a shorter firing duration $t_s(i,k)$ of actor i causes it to complete iteration k earlier than a longer firing duration $t_l(i,k)$. The shorter firing duration $t_s(i,k)$ of actor i therefore cannot cause a later enabling $s(j,k+d(i,j))$ of actor j, and similarly a longer firing duration $t_l(i,k)$ of actor i cannot cause an earlier enabling of actor j. The dataflow abstraction of the application is therefore monotonic in both:

1. Actor enabling times $s(i,k)$, as given by Equation 2.2.
2. Actor firing durations $t(i,k)$, as given by Equation 2.3.

which means that:

1. Scheduling an application following any scheduling scheme other than STS cannot improve the timing performance of the graph, because actors are fired as soon as they are enabled in an STS.
2. Worst-case actor firing durations produce graph timings that conservatively bound the timing performance of the graph using actual case actor timings.

We can therefore schedule an application graph following an STS, while conservatively analysing the application's timing following any scheduling scheme that satisfies Equation 2.1. Moreover, using worst-case task timings enables conservative analysis of the application's timing for all actual task execution times. In the following two sections, we will explain how WCSTS and SPS are used for conservative timing analysis of application graphs that execute as STS in the actual case.

Worst-Case Self-Timed Schedule (WCSTS)

A WCSTS is the STS of a dataflow graph in which all the actors fire for the duration of their worst-case execution time. Due to the monotonicity of dataflow execution, the actor firing times of a WCSTS cannot be earlier than those of a STS for the same graph. This is a useful simplification that enables the derivation of conservative guarantees where the precise duration of all executions of each actor is not known in advance, but where the duration has a worst-case bound, e.g. an HSDF modelled H.263 video decoder with data-dependent execution.

The throughput of an HSDFG is derived with a Maximum Cycle Mean (MCM) $\mu(G)$ analysis. For a WCSTS of a HSDFG, the graph executes in a periodic manner after $K(G)$ iterations. The period of the cyclical execution is $\mu(G) \cdot N(G)$, where $N(G)$ is the cyclicity of the graph [11]. The cyclicity $N : G \to \mathbb{N}$ of the graph is derived as follows:

$$N(G) = \operatorname*{lcm}_{\forall m \in M(G)} \left(\gcd_{\forall c \in C(m)} \left(\sum_{\forall e \in E(c)} d(e) \right) \right) \tag{2.4}$$

where $M(G)$ is the set of all maximally strongly connected subgraphs in graph G, $C(m)$ is the set of all cycles in subgraph m and $E(c)$ is the set of all edges, belonging to cycle c. The math operator gcd is the greatest common divisor and lcm is the least common multiple.

Throughput of the HSDFG is the rate at which the graph completes iterations. In an HSDFG iteration, all actors complete a single firing. The average throughput of the WCSTS of a HSDFG is therefore $\mu(G)^{-1}$ over any $N(G)$ graph iterations in the cyclic regime. The MCM $\mu : G \to \mathbb{R}$ is calculated as [76]:

$$\mu(G) = \max_{\forall c \in C_s(G)} \frac{\sum_{\forall i \in V(c)} t(i)}{\sum_{\forall e \in E(c)} d(e)} \tag{2.5}$$

where $C_s(G)$ is the set of all simple cycles in graph G and $V(c)$ is the set of all actors belonging to cycle c. Due to the monotonicity of dataflow execution, the average throughput derived for a graph following a WCSTS provides the lowest average throughput bound for the same graph following an STS. The schedule of actors in a WCSTS of a HSDFG is derived as follows:

$$\forall k \geq K(G) : s(i, k + N(G)) = s(i,k) + N(G) \cdot \mu(G) \tag{2.6}$$

where $K : G \to \mathbb{N}$ is the number of graph iterations until the periodic phase of execution.

Static-Periodic Schedule (SPS)

In an SPS, actors fire at periodic intervals. In [76] this is formally characterised for all actors $i \in V$ and iterations $k \in \mathbb{N}$ as:

$$\forall k : s(i,k) = s(i,0) + T \cdot k \tag{2.7}$$

with T being the period of the SPS schedule. The start time of every actor iteration $s(i,k)$ occurs at intervals of length T from the starting time of the actor's first iteration $s(i,0)$. The per-edge admissibility constraint for an SPS is more strict than Equation 2.1:

$$s(j,k) + T \cdot d(i,j) \geq s(i,k) + t(i) \tag{2.8}$$

For a single edge in isolation, constraint Equation 2.8 ensures that the starting time of the consuming actor $s(j,k)$ cannot be earlier than the starting time of the producing actor $s(i,k)$ in addition to the worst-case time it takes for the producing actor to complete $t(i)$. The token being consumed in graph iteration k is produced $d(i,j)$ iterations earlier, as the initial tokens on the edge are consumed first. As actors fire with a period of T, the token being consumed was produced at the latest $T \cdot d(i,j)$ earlier than the finishing time of current iteration of the producing actor $s(i,k) + t(i)$.

2.2 CompSOC: Predictable and Composable Hardware

The CompSOC hardware is a composable NoC centric MPSoC, providing the ability to derive temporal bounds on execution and communication [44]. CompSOC is a tile-based architecture, consisting of a combination of computation, interconnect and memory tiles, as illustrated in Figure 2.2a. This illustration presents an example CompSOC platform with three processing tiles, one memory tile and one Thin-Film Transistor (TFT) display. An example set of logical point-to-point Æthereal NoC connections is also shown to demonstrate how the tiles are able to communicate over the NoC. In Section 2.2.1, we focus on the contribution of a composable processing tile enabling the creation of a composable CompSOC hardware platform. Much work has been previously carried out on the composable Æthereal NoC and Raptor Synchronous Dynamic Random Access Memory (SDRAM) controller, as described in Section 2.2.2.

The CompSOC platform has an automatic generation toolflow [35], enabling the platform architecture, communication and application configuration to be specified at relatively high abstraction levels in Extensible Markup Language (XML). Additional peripheral tiles can be added as needed, but care needs to be taken to ensure that the peripherals maintain the predictable and composable nature of the CompSOC platform.

2.2.1 Processing Tile

The CompSOC processing tile enables predictable execution of software, enabling worst-case timing bounds to be derived for executables, such as POSe actors. Using the CoMik microkernel, the CompSOC processing tile enables the physical processor to be temporally divided into multiple composable virtual processors. The CompSOC tile uses the general purpose MicroBlaze processor unlike other temporally isolating processor sharing techniques, such as Precision-Timed Systems (PRET) [65] that use a custom processor with multiple independent hardware threads. The latter techniques have an upper limit on concurrent composable computation, as they achieve temporal isolation by allocating dedicated hardware processor resources.

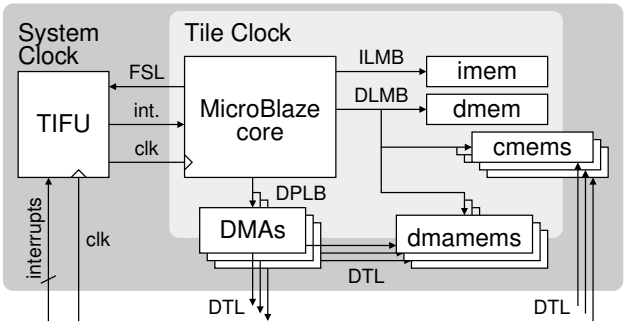


Figure 2.10: CompSOC processing tile

The CompSOC processor tile can be configured in multiple ways. Figure 2.10 illustrates an example configuration consisting of a single MicroBlaze processing core, some local memory, one or more DMAs, and a TIFU. The MicroBlaze core is a soft core that we configure to have a single 5-stage in-order pipeline with no branch prediction. The in-order pipeline simplifies the core's predictability, while disabling the branch prediction enables CoMik to maintain composability between applications, as the branch predictor's state, which is influenced by applications executing in their partitions, would be carried over between partitions. While the MicroBlaze processor has support for hardware caches and virtual memory through the use of a Memory Management Unit (MMU), these features are not implemented in the CompSOC platform. Caches are not implemented to simplify predictability. While an MMU would be a desirable feature for memory composability in the spacial domain, due to its ability to isolate memory regions, it is not implemented due to the associated timing performance overheads of the MMU. An Memory Protection Unit (MPU) would be a suitable alternative that provides memory region isolation without the MMU's overhead of address translation. While an MPU would be a desirable feature, it is orthogonal to composable and predictable execution and is therefore not currently implemented in the CompSOC platform. The CompSOC platform does use the MicroBlaze processor's hardware stack protection and the DMA drivers have software memory region protection.

CoMik divides the processor into TDM time slices that are allocated to virtual processors, as described in Section 2.3. Each virtual processor can perform DVFS, receive interrupts and set timed interrupt events independently of other virtual processors. To achieve this, we contribute the hardware Timer-centric Interrupt and Frequency Unit (TIFU) that regulates CoMik's virtual processor TDM schedule and enables composable and programmable timed DVFS and interrupt events. The TIFU is described in detail in Section 2.2.4.

Communication and Memory Architecture

CoMik is used to create virtual platforms consisting of multiple virtual processors that can be located on multiple physical processing tiles. A virtual processor on one tile that belongs to the same virtual platform as a virtual processor on another tile must be able to communicate without interfering with any other virtual platform. Interference may occur on shared hardware resources along the communication path. In the CompSOC processing tile, these are the DMAs and local memories. In this section, we contribute an overview of various processing tile communication and memory architecture configurations, as illustrated in Figure 2.11. Of these memory architectures we contribute two composable tile configurations and explain the difference and trade-offs between the two.

The CompSOC tile configuration of the MicroBlaze core has one instruction bus and two data buses. One of each type is a single cycle latency Local Memory Bus (LMB) (for a 32 bit word load/store) and the remaining data bus is multi-cycle latency Processor Local Bus (PLB) (also for a 32 bit word load/store). Figure 2.11a presents a basic memory architecture that uses one single port memory on each LMB. A bridge on the PLB allows the MicroBlaze to perform Memory Mapped Input/Output (MMIO) transactions across the Device Transaction Level (DTL) Æthereal NoC. MMIO transactions carried out across the PLB are blocking, meaning that the computation stalls until the communication finishes. As the transactions are communicated across the NoC to a remote memory location, the exact duration of the stall depends on the configuration of the NoC and the arbitration of the remote memory. A single word transaction can stall the computation on the processor for hundreds of cycles. CoMik's virtualisation scheme, as described in Section 2.3.3, uses interrupts to context switch virtual processors, but it is not possible to interrupt the multi-cycle PLB transaction. The worst-case interrupt service latency can therefore be very high, and worse depend on the configuration of the rest of the system. Although Figure 2.11a can be used when the interrupt service latency is less than a predefined jitter bound, we prefer to make the virtual processor performance analysis independent of other resource and therefore do not use this tile configuration.

Parallelising communication and computation using a DMA module enables non-blocking remote MMIO transactions. The DMA is programmed using MMIO transactions on the PLB, allowing multiple DMAs to be connected to the same bus. Once the DMA module has been programmed with the source address, destination address and size of the transaction then the processor may continue computation. Figure 2.11b presents a modification of the CompSOC tile in Figure 2.11a to use a DMA module instead of a bridge to perform communication over the NoC. The DMA module connects to a second port on the dmem, enabling it to access the memory simultaneously with the processor.

The DMA module contains a FIFO transaction queue, allowing multiple outstanding NoC transactions at any time. Each transaction is processed in order, with the amount of data specified by the programmed size of the transaction being moved from the source address to the destination address. For a DMA write transaction, the data is read from the local memory connected to the DMA and written to the NoC accessible destination

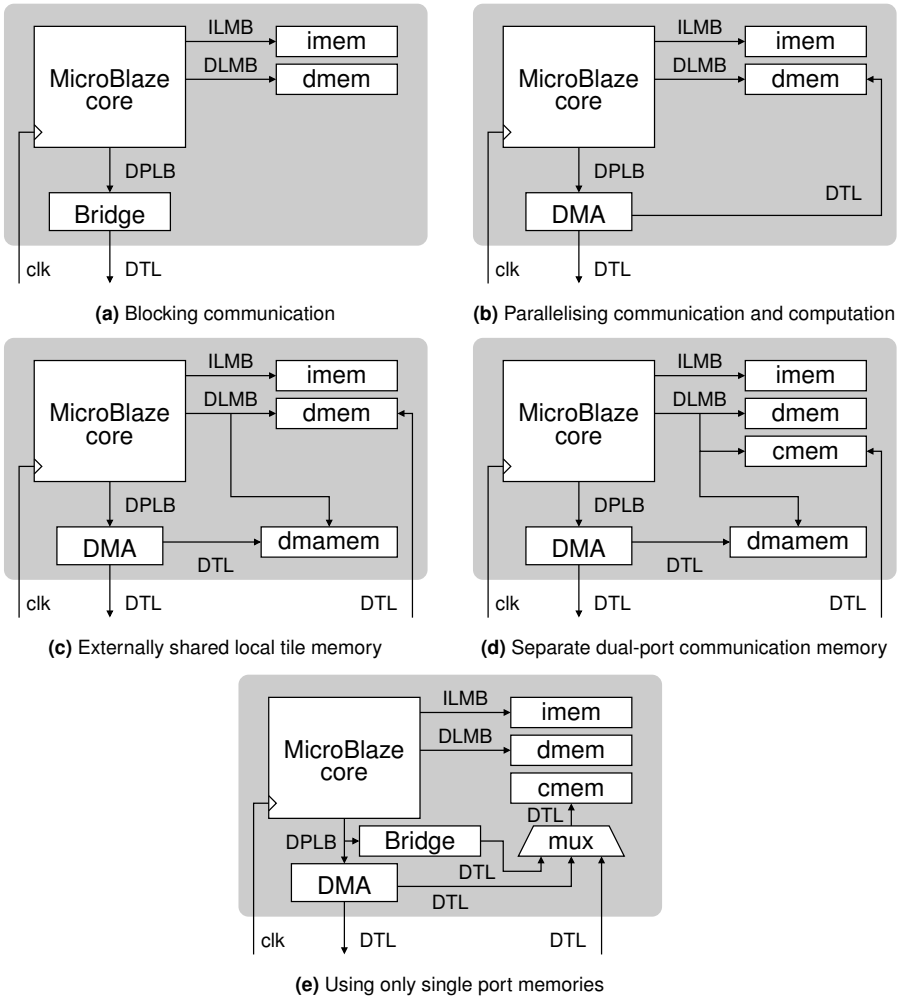


Figure 2.11: CompSOC processing tile communication and memory configurations

address. Similarly, for a read transaction, the data is read from the NoC accessible source and written to the local DMA memory. To ensure application-level composability, each DMA can only be used by a single application.

CompSOC tiles using either the memory architecture from Figure 2.11a or Figure 2.11b can communicate data via a remote shared memory. To communicate data between tiles, it must first be written from a tile's local dmem into a shared memory location and then read into another tile's local dmem. The memory architecture presented in Figure 2.11c reduces the number of transactions required to communicate data between

Mem. Arch.	Compos-able	Parallel Comm. & Comp.	Remote Access to Tile Mem.	Single Port Mem. Only	All Mem. on LMB
Figure 2.11a	✗	✗	✗	✓	✓
Figure 2.11b	✗	✓	✗	✗	✓
Figure 2.11c	✗	✓	✓	✗	✓
Figure 2.11d	✓	✓	✓	✗	✓
Figure 2.11e	✓[1]	✓	✓	✓	✗

[1] When using a composable arbitration scheme, such as TDM or any temporally boundable arbitration scheme with delay blocks.

Table 2.1: Overview of CompSOC memory architecture properties.

tiles by enabling data to be written directly into the tile's local dmem from across the NoC. This is achieved by using a separate dual-port memory for use with the DMA, called the dmamem. This frees up one of the ports of the dmem, allowing it to be connected to the NoC enabling remote MMIO access.

Using a separate dmamem with every DMA enables multiple DMAs to be instantiated as part of the tile. DMAs can therefore become dedicated resources of virtual processors. While access to the DMA resource is therefore composable and the Æthereal NoC connections are composable, the time taken to complete a transaction also depends on contention at the memory controller. Figure 2.11d illustrates a memory architecture that enables multiple dual-port communication memories (cmems) to be instantiated per tile. These cmems can therefore become dedicated resources of virtual platforms, removing the possibility of contention from other virtual platforms. Virtual platforms consisting of virtual processors on multiple physical processors can therefore perform composable inter-tile communication by using dedicated DMAs, dmamems, Æthereal NoC channels and cmems.

Figure 2.10 illustrates a version of the tile architecture from Figure 2.11d that has three cmems, DMAs and dmamems enabling up to three virtual platforms to perform composable communication to and from the tile. The dmamems and cmems use dual-port memory, which has a higher logic overhead than single-port memory. Figure 2.11e presents an alternative composable memory architecture that only uses single-port memories. This is achieved by merging all dma and cmems into a single cmem that is arbitrated using a composable arbitration scheme, such as TDM or any temporally boundable arbitration scheme with delay blocks. MMIO transactions directly from the processor (using a protocol bridge), local DMAs and NoC connections all contend for access to the cmem. In general, access to the cmem is therefore significantly slower than directly accessing the memory. This diminishes the memory access speed benefit of using local scratchpad memories.

The saving achieved by using only single-port memories (higher memory density and

fewer memory controllers) is offset by the addition of a multiplexer and the logic for its arbitration scheme. As such, the use of the memory architecture in Figure 2.11e only makes sense if the number of cmems and dmamems that would be used for the memory architecture in Figure 2.11d are of sufficient size and number that the saving outweighs the additional cost of the multiplexer.

For the CompSOC platform to be composable, the processing-tile must use either the architecture from Figure 2.11d or Figure 2.11e, as listed in Table 2.1. The choice is a trade-off between faster memory access using the single-cycle per word LMB, provided by the memory architecture of Figure 2.11d, and the potentially cheaper in terms of logic, memory architecture of Figure 2.11e. The CompSOC FPGA prototype that is used in this thesis uses the memory architecture from Figure 2.11d as the virtex 6 FPGA that is used for experimentation uses dual-port memory blocks.

2.2.2 Predictable and Composable NoC Interconnect and SDRAM

The Æthereal NoC is the interconnect that joins all of the tile types together. It provides virtual point-to-point connections across its network topology, that have predictable upper bounds on throughput and latency. The point-to-point connections time share the hardware along their path using TDM arbitration. The arbitration is configured in such a way to avoid any contention for resources between the connections. The details of the Æthereal NoC, and how it achieves these composable TDM configurations, are beyond the scope of this thesis. Much research has been carried out on the Æthereal NoC for over a decade [32], with detailed information available in [43, 94, 95].

Composable and predictable memory access is another part of the CompSOC platform where much research has been performed [4, 33, 36, 63]. Whereas it may be possible to provide multiple small scratch pad memories in tiles to avoid memory contention and hence provide composability, this is not practical for larger and potentially off-chip memories, such as SDRAM. Composable arbitration schemes, such as TDM and Credit Controlled Static Priority (CCSP) with delay blocks [5], can be used to arbitrate the memory. Semi static scheduling with patterns or restricted dynamic scheduling are used to make the memory predictable allowing conservative bounds to be given on access latency and throughput. More information on how this is achieved can be found in [63], as the specifics are beyond the scope of this thesis.

2.2.3 Modelling CompSOC Inter-tile C-HEAP Communication

The CompSOC predictable hardware platform enables worst-case temporal bounds to be calculated for a given usage. These worst-case timings are used with dataflow models enabling a worst-case temporal analysis. In this section, we describe how the FIFO communication of tokens along a dataflow edge can be implemented using the C-HEAP communication protocol. We contribute three different inter-tile C-HEAP FIFO configurations with HSDFs timing models, enabling temporal analysis of the communication.

These models are incorporated into the combined application and CompSOC HSDFG, as described in Section 2.5.2.

Edges as C-HEAP FIFOs

To ensure compatibility with the dataflow MOC, all inter-task communication must be carried out using FIFO channels. POSe provides software FIFO channels using the C-HEAP communication protocol [82]. A simple C-HEAP FIFO consists of a data buffer and an administration that is stored in a shared memory that is accessible by both the FIFO's producer and consumer. Data is transferred as fixed size tokens of data. The memory size of the FIFO buffer is therefore determined by the size of the FIFO tokens and the number of tokens that the buffer accommodates.

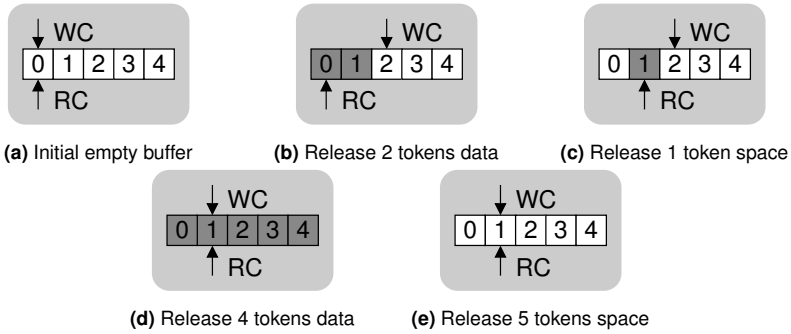


Figure 2.12: C-HEAP circular buffer administration

The token size and FIFO capacity is configurable per FIFO channel. This information is stored in each FIFO's administration along with a read and write counter. The C-HEAP protocol uses an administrated circular buffer to create a FIFO channel, as illustrated in Figure 2.12. The Read Counter (RC) and Write Counter (WC) are used to track the FIFO buffer occupancy. Figure 2.12a illustrates an empty C-HEAP buffer with a capacity of five tokens. Whenever the producing task writes a token into the FIFO buffer, it increments the write counter to release the data. This is shown in Figure 2.12b when two tokens of data are inserted into the FIFO buffer and then released. Similarly, whenever the consuming task reads a token from the FIFO buffer, it increments the RC. Figure 2.12c demonstrates this for the running example by consuming a token from the FIFO buffer and then releasing the space.

The occupancy of the buffer is determined by subtracting the RC from the WC and if the result is negative adding the size of the buffer in terms of tokens, e.g for Figure 2.12c the resultant occupancy is one, so there is no need to add the size of the buffer. Figure 2.12d and Figure 2.12e demonstrate that the situation when both the RC and WC are equal is ambiguous, as the buffer in Figure 2.12d is full yet the buffer in Figure 2.12e is empty. To disambiguate this situation, in implementation, the most significant bit of the RC and WC

is toggled every time the counter wraps around when it reaches the end of the buffer. To calculate the buffer occupancy, the counters are masked to find the RC and WC values and their wrap around bit. As before, the masked RC value is subtracted from the WC value, but now, the size of the buffer is only added to the resultant value if the two wrap around bits differ.

The C-HEAP administration uses two values to track buffer occupancy to avoid the need for any sort of memory locking mechanism in the event that the producer and consumer tried to update the administration at the same time. This might occur whenever the producer and consumer tasks execute on different processors on a multi-processor system. By using two values, the producing task only ever updates the WC and the consuming task only ever updates the RC, therefore negating the need for a memory lock.

A C-HEAP FIFO requires that both the producer and the consumer have access to the memory on which it is mapped, i.e. that the memory is shared. On multi-core platforms with a distributed memory architecture, the accessibility of memories can vary per processor. In addition, the memories themselves can vary in capacity and access speed, i.e. small fast local scratchpad memory vs. large relatively-slow remote SDRAM. In the following sections, we present three rational C-HEAP configurations where the RC, WC and FIFO buffer are located in various distributed memory locations. An overview of some possible configurations is given in Table 2.2

C-HEAP Communication via Remote Shared Memory Only

Figure 2.13a illustrates the situation where the C-HEAP administration is located in shared memory that is remote to both processing tiles. This might be necessary in the situation where both processing tiles do not have any access to a local memory of the other processing tile, but do have access to a shared memory. Both the producer and the consumer task keep local copies of the C-HEAP FIFO's RC and WC. Each task must keep its local copies up-to-date. This is achieved by reading the values from the shared memory. As the producer only updates the WC and the consumer only updates the RC, they both only need to retrieve the value that the other task updates, to keep their local C-HEAP administration up-to-date.

The processing tiles' dmamems are local scratchpad memories. These are generally fast to access but are relatively small. The shared remote memory can be a much larger SDRAM. As such, in Figure 2.13a the full C-HEAP FIFO buffer is located in the shared remote memory, with space reserved in the local dmamems for a single iteration of the task's produced/consumed tokens.

The enumerated transactions required to produce and consume data from a C-HEAP FIFO that is configured as shown in Figure 2.13a are explained as follows:

1. **Producer:** Read RC into dmamem from shared memory then check RC and WC for buffer space.

2. **Producer:** Write token from dmamem to shared memory.

In Section 2.2.3	Data[1]	RC[1]	WC[1]	Properties
Page 29	S	S	S	Supports large data buffer. Slow[2] to access data and counters
✗	P&C[3]	P	P	Relatively small data buffer. Fast to access data and counters from producer, but slow for consumer.
✗	P[4]&C	C	C	Relatively small data buffer. Fast to access data and counters from consumer, but slow for producer.
Page 31	P[4]&C	P&C	P&C	Relatively small data buffer. Fast to access data from consumer. Slow to update counters, but fast to check.
Page 33	S	P&C	P&C	Supports large data buffer. Slow to access data and update counters, but fast to check counters.

[1] C-HEAP components (Data, RC, WC) in shared memory are either:

P Local to producer.
C Local to consumer.
S Not local to producer or consumer.

[2] The CompSOC platform's DMA and Æthereal NoC combination supports posted writes, enabling multiple concurrently outstanding write transactions per connection, but only one outstanding read transaction per connection. It is therefore preferable, from a timing performance point of view, to minimise reading from remote shared memories.

[3&4] It is not necessary for this buffer to reserve the full capacity of the FIFO, but it must be minimally dimensioned to contain the maximum number of tokens that:

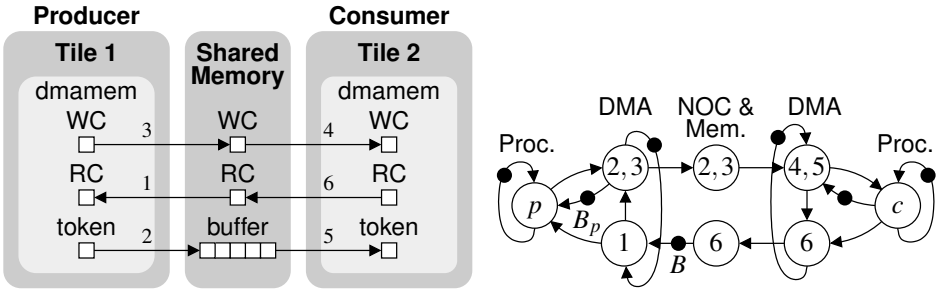
[3] the consuming task requires for a single firing.

[4] the producing task can output in a single firing.

Table 2.2: Overview of CompSOC memory architecture properties.

3. **Producer:** Write updated WC from dmamem to shared memory.
4. **Consumer:** Read WC from shared memory into dmamem then check RC and WC for buffer data.
5. **Consumer:** Read token from shared memory to dmamem.
6. **Consumer:** Write updated RC to shared memory from dmamem.

Figure 2.13b presents an HSDF model of the C-HEAP memory mapping illustrated in Figure 2.13a, enabling temporal analysis of the C-HEAP communication. The transaction enumeration from Figure 2.13a is marked on the actors to show which transaction timings they represent, with some actors representing the timing of multiple transactions.



(a) C-HEAP memory mapping

(b) HSDF model of Figure a

Figure 2.13: Inter-tile C-HEAP FIFO communication via remote shared memory only.

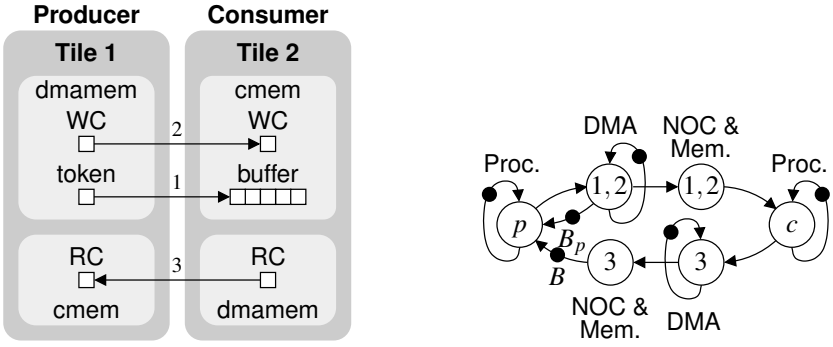
The C-HEAP FIFO is initially empty with its buffer capacity represented by B initial tokens in Figure 2.13b. Transaction 1 is modelled by an actor that represents the time it takes to read the RC from the shared memory and write it into the dmamem. The actor representing the producing task p can subsequently fire if there is enough space in its local output buffer of capacity B_p. In Figure 2.13a, local buffers have sufficient capacity for a single token, making $B_p = 1$ and $B_c = 1$. Transactions 2 & 3 are represented by a single actor that represents the time it takes for the DMA to transfer the produced data and the updated WC onto the NoC. The following actor represents the time it takes for the last word of data from transactions 2 & 3 to cross the NoC and be written into the shared memory.

With the WC updated in shared memory to indicate the presence of data in the buffer, the consuming task performs transactions 4 & 5 that are represented as a single actor in Figure 2.13b, if there is enough space in the local buffer of capacity B_c. Once the data is stored in the consuming task's local dmamem, the actor representing the consuming task c can fire. The task consumes the data and releases the space by performing transaction 6 using the DMA, to update the RC in the shared memory. This is represented in the HSDFG in Figure 2.13b by two actors. The first actor represents the time taken by the DMA to write the updated RC onto the NoC, while the second actor represents the time for the RC to cross the NoC and be written into the shared memory. The producing task is now able to observe this space by performing transaction 1, and the whole process repeats.

C-HEAP Communication Using Local Scratchpad Memories Only

The CompSOC processing tile has small scratchpad memories (cmem) that can be written to from the NoC. Figure 2.14b illustrates a C-HEAP mapping that only uses memories that are local to either the producing or consuming task, enabling faster access to the data and requiring fewer NoC transactions. This comes at the cost of memory capacity as local scratchpad memories are generally relatively small Static Random Access Memorys (SRAMs). In Figure 2.14a, the C-HEAP buffer and administration is mapped in such a

way to only require data to be written across the NoC, taking advantage of the Æthereal NoC's posted write capability. This is achieved by placing the C-HEAP values that are updated by the task on the other tile in the cmem, i.e the RC for the producing task and the WC and buffer for the consuming task. Values that the task updates are located in the local dmamem, enabling the remote copies of the values to be updated using the DMA to write the values to the other tile's cmem, i.e. the WC and token buffer for the producing task and the RC for the consuming task.



(a) C-HEAP memory mapping

(b) HSDF model of Figure a

Figure 2.14: Inter-tile C-HEAP FIFO communication using local memories only.

The enumerated transactions required to produce and consume data from a C-HEAP FIFO that is configured as shown in Figure 2.14a are explained as follows:

1. **Producer:** Check RC and WC for buffer space then write token from dmamem to shared memory, if space is available.

2. **Producer:** Write updated WC from dmamem to cmem local to consumer.

3. **Consumer:** Check RC and WC for buffer space then read token from shared memory to dmamem. Write updated RC to shared memory from dmamem.

We present a HSDF model in Figure 2.14b of the C-HEAP memory mapping that is illustrated in Figure 2.14a. The actors in the HSDFG are marked with the transaction enumeration from Figure 2.14a to show which transaction timings they represent, with some actors representing the timing of multiple transactions.

The C-HEAP FIFO is initially empty with its buffer capacity represented by B initial tokens in Figure 2.14b. The producing task checks for space in the buffer by comparing the local RC and WC. The producing actor can subsequently fire if there is space in the C-HEAP buffer and also enough space in its local output buffer of capacity B_p. Once the producing task has completed, transactions 1 & 2 write the data and updated WC into the consuming tile's cmem. The timing of these transactions is represented by two dataflow

actors. The first actor represents the time taken by the DMA to write the data followed by the WC onto the NoC. The second actor represents the time taken for the last word of data from transactions 1 & 2 to be written across the NoC and into the cmem of the consuming task.

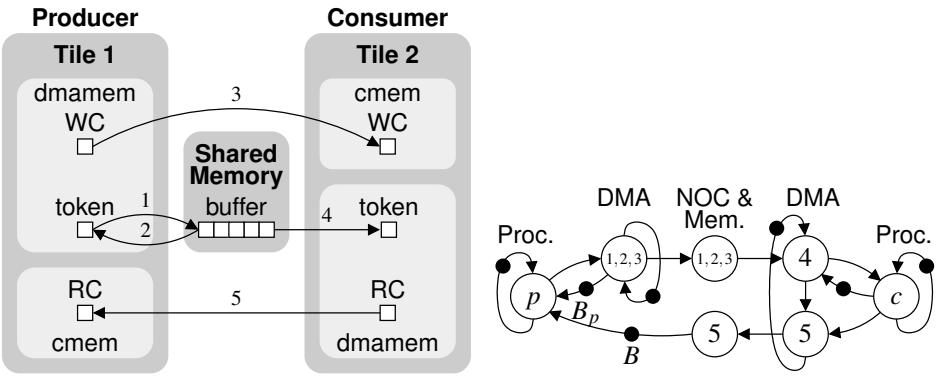
The consuming task is enabled to fire whenever it observes the presence of the data in the buffer by comparing its local RC and WC. After its firing has completed, it updates the local RC and performs transaction 3 to release the space. The timing of transaction 3 is modelled using two dataflow actors. The first actor models the time it takes the DMA to transfer the updated RC onto the NoC. The second actor represents the time it takes for the RC to cross the NoC and be written into the cmem that is local to the producing task. The producing task is now able to observe this space by comparing its local RC and WC, and the whole process repeats.

C-HEAP Communication via Remote Shared Memory with Local Administrations

The C-HEAP memory mapping that is presented in Figure 2.13a uses a relatively slow but potentially large memory to store the C-HEAP FIFO tokens. Its method of operation requires six NoC transactions to be performed to produce and consume one token of data. The scheme also does not take full advantage of the Æthereal NoC's posted write capability, as three of these six transactions are reads. The C-HEAP memory mapping that is presented in Figure 2.14a uses fast to access, but small, local memories to store the C-HEAP FIFO components. Its method of operation requires three NoC transactions to be performed to produce and consume one token of data. All of the three transactions are writes, taking full advantage of the Æthereal NoC's posted write capability. We continue by presenting a hybrid of these two C-HEAP memory mappings that uses the slower potentially larger remote shared memory to store the C-HEAP buffer but uses the faster local scratchpad memories to maintain the C-HEAP administration values. This mapping requires five NoC transactions to be performed to produce and consume one token of C-HEAP data, of which two of the five transactions are reads.

The C-HEAP memory mapping illustrated in Figure 2.15a is similar to the mapping illustrated in Figure 2.14a, except the buffer is moved from the consuming tile's cmem into the shared memory. As such, the consuming tile requires a local buffer in the dmamem to store C-HEAP token data that is read using a DMA from the C-HEAP buffer in the shared memory.

As the C-HEAP administration and buffer reside in different memories, the producing task must make sure that the data is written into buffer memory before updating the WC. If this is not done and the time taken to write the WC into the consumer's cmem is less than the time taken to write the data in to the shared buffer then the consumer could read the data from the buffer before it has been updated, or while it is partially updated [61]. To prevent this, the producing task reads a single word from the shared memory using the same NoC connection before updating the WC. This ensures that the data has been written into the shared memory, as transactions on Æthereal NoC connections are carried



(a) C-HEAP memory mapping

(b) HSDF model of Figure a

Figure 2.15: Inter-tile C-HEAP FIFO communication using local and remote memories.

out using FIFO ordering.

The enumerated transactions required to produce and consume data from a C-HEAP FIFO that is configured as shown in Figure 2.15a are explained as follows:

1. **Producer:** Check RC and WC for buffer space then write token from dmamem to shared memory, if space is available.

2. **Producer:** Read single word from shared memory using the same NoC connection, to ensure that the data has been written to the buffer.

3. **Producer:** Write updated WC from dmamem to cmem local to consumer.

4. **Consumer:** Check RC and WC for buffer space then read token from shared memory to dmamem.

5. **Consumer:** Write updated RC from dmamem to cmem local to producer.

We model the memory mapping from Figure 2.15a as a HSDFG as illustrated in Figure 2.15b. The actors in the HSDFG are marked with the transaction enumeration from Figure 2.14a to show which transaction timings they represent, with some actors representing the timing of multiple transactions.

The C-HEAP FIFO is initially empty with its buffer capacity represented by B initial tokens in Figure 2.15b. The producing task observes the space in the buffer by comparing its local RC and WC. It is enabled to fire if space is available in both the C-HEAP buffer in the shared memory and in its local output buffer of capacity B_p. Once the producing task has finished firing, the produced data is written into the C-HEAP FIFO using transactions

1,2 & 3, to write the data into the C-HEAP buffer in the shared memory, check it has finished being written and update the WC. This is represented by two actors in the dataflow model. The first actor represents the combined time it takes for the DMA to perform transactions 1,2 & 3. The second actor represents the duration of time it takes for the WC to be written across the NoC and into the consuming task's local cmem.

The consuming task observes the presence of data in the C-HEAP buffer by comparing its local RC and WC. If there is enough space in the consuming task's local token buffer of capacity B_c, then the data is read from the C-HEAP buffer by performing transaction 4 with the local DMA. Once the data is present in the local buffer, the consuming task can fire. Upon completion, the space of the consumed data in the C-HEAP FIFO is released by updating the RC and performing transaction 5 using the local DMA to update the RC in the local cmem of the producing task. The timing of transaction 5 is represented by two actors in the dataflow graph. The first actor represents the time taken by the DMA to write the updated RC onto the NoC. The second actor represents the time it takes for the RC to cross the NoC and be written into the cmem of the producing task. The producing task is now able to observe this space by comparing its local RC and WC, and the whole process repeats.

2.2.4 Time-centric Interrupt and Frequency Unit

CoMik requires interrupts and frequency changes to occur in a coordinated manner at precise times, e.g. virtual processor context switches are triggered by an interrupt and the frequency is changed to the CoMik slot frequency, as explained in Section 2.3.3. Achieving this while enabling more complex functionality, such as partition-level DVFS and interrupts, requires considerable coordination between the different hardware modules that provide the functionality. Without hardware support, CoMik would have to provide the coordination effort in software adding complexity, code size and timing overhead [16]. As such, we contribute the TIFU hardware module that provides the functionality required by CoMik using integrated hardware coordination enabling a relatively low software control overhead.

The TIFU module performs the roles of being a time reference, a DVFS module and an interrupt controller. The TIFU follows a modular design, as illustrated in Figure 2.16, integrating a Programmable Interrupt Controller (PIC), a DVFS module and two 64 bit Programmable Interrupt Timers (PITs), for the frequency scaled and unscaled time domains (hence time-centric). A control unit interfaces with the processor and coordinates requests with the TIFU's internal modules, enabling the TIFU to perform functionality in hardware that would otherwise have been in software.

The TIFU forms part of the CompSOC processor-tile, enabling per tile DVFS and interrupt control. As illustrated in Figure 2.10 the TIFU is connected as a slave to the processor via an Fast Simplex Link (FSL) bus. Software running on the processor sends requests to the TIFU using the FSL instructions that form part of the MicroBlaze's Instruction Set Architecture (ISA). Multiple FSL instructions may need to be executed to

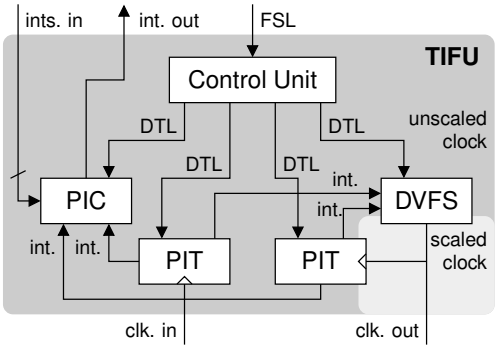


Figure 2.16: TIFU hardware architecture

provide or retrieve the appropriate amount of data associated with the request. To simplify accessing TIFU services, we abstract away from the instruction level by providing driver functions.

2.2.5 Control Unit

The TIFU's control unit interprets the communication from the processor, performing the requested action or returning the requested data. Each FSL transaction has a data length of 32 bits. The TIFU's control unit interprets the most significant 16 bits as the operation and the least significant 16 bits as the operand. The control unit uses the decoded operation to decide which TIFU sub-modules carry out the operation. The decoded operation also instructs the control unit what to do with the data. Depending on the operation, further data may be required and is provided in subsequent FSL transactions. TIFU requests, such as reading the time, requires data to be returned to the processor from the TIFU. The TIFU makes the data available to be read by the processor across the FSL bus. Data larger than 32 bits in size is read using multiple FSL transactions.

2.2.6 PIT Modules

The TIFU is called time-centric as its main utility is the precise timing coordination of hardware DVFS and interrupt events. Due to its DVFS functionality, the TIFU straddles two clock domains. To allow events to be triggered based on time in either domain, two PIT modules are used, to provide a time reference for each domain. Each PIT module contains a 64 bit hardware counter that counts the rising edges of an input clock signal. In the TIFU, one PIT counts the reference clock (which we refer to as the *tile clock*) and one PIT counts the scaled clock (which we refer to as the *partition clock*) produced by the DVFS module. Each counter increments on each rising edge, enabling a single cycle resolution time reference in both clock domains.

A 64 bit counter is used, as it provides sufficient magnitude to allow the PITs to count for 5845 years using single cycle granularity at 100 MHz, without rolling over to zero again. A 32 bit counter will roll over to zero after only 42.9 seconds under the same conditions. While this might be sufficiently long for some use cases, it is safe to assume that 42.9 seconds is too short for many use cases. In comparison to a 32 bit counter, a 64 bit counter does not add much additional complexity to the hardware, but does add an area overhead. Software techniques can be used to effectively extend a 32 bit counter, making it seem that the counter is larger than it actually is, but in comparison to a 64 bit counter, this adds software complexity and overhead.

Each counter can be instructed to start or stop counting. The value of the counter can be read while the counter is running or stopped. The 64 bit counter value needs to be returned in two 32 bit (high and low) pieces via the TIFU's FSL connection with the processor. To ensure consistency between the high and low values, the 64 bit timer is sampled in a single action by reading it into a 64 bit buffer that is subsequently returned via the FSL. The values of the counters are also programmable via the FSL, allowing the context of the timer to be stored and restored, thereby enabling CoMik partitions to maintain their own virtual time reference as part of their context.

In order to regulate the timings of DVFS and interrupt events, the PITs are programmable to produce internal TIFU interrupts at requested times. The counters are programmed with the time value at which the event should occur. This time value is compared with the counter's time value, to decide whether an internal TIFU interrupt should be produced. If the counter's time value is greater than or equal to the event time, an internal interrupt is produced. Each PIT can store concurrent times for DVFS and interrupt events. The requested event times are stored in separate registers. An internal TIFU interrupt is produced for the particular module depending on which event register triggers the event condition.

As described in Section 2.3.3, CoMik requires periodic interrupts to regulate the context switching of its virtual processors. The PIT allows a period to be specified for the internal interrupt that is produced for the PIC module. The period of the interrupt is stored in a register. Whenever an interrupt is triggered for the PIC module, the value of the period is added to the time that triggered the event and stored back in the PIC interrupt time register, to produce an interrupt sometime in the future. CoMik's virtual-processor context-switch interrupt is always produced using the TIFU's tile clock PIT. CoMik's partitions can specify periodic and non-periodic interrupts using the TIFU's partition clock PIT.

2.2.7 DVFS Module

The DVFS module allows the TIFU to scale the voltage and frequency of the CompSOC processor-tile. On an Application Specific Integrated Circuit (ASIC) implemented system, the DVFS module scales to the requested frequency and selects the lowest voltage that reliably supports that frequency. The CompSOC platform is prototyped on FPGA. While

many FPGA chips support voltage and frequency scaling to some degree, using their capability to support multiple voltage and frequency islands adds significant complexity to the design.

The FPGA prototyped TIFU provides frequency scaling through clock division of the reference clock signal. This is achieved by specifying the desired frequency as a fraction of the reference frequency in the form of an integer numerator and denominator value. The scaled clock frequency is derived from the reference clock frequency by propagating a numerator amount of reference clock pulses in every denominator amount of reference clock pulses, and blocking the rest. This is achieved on the FPGA by using a clock buffer component with a clock enable pin, such as the BUFG component on Xilinx FPGAs. The clock enable pin allows reference clock pulses to propagate through the buffer module when it is raised, and blocks the reference clock pulses when it is low. By counting the number of rising edges on the reference clock, the clock enable signal is raised for a numerator amount of cycles every denominator amount of cycles. The clock pulse width of the scaled clock remains the same as the reference clock, but on average the scaled clock has a frequency that is a fraction (numerator over denominator) of the reference clock.

For any given duration, frequency scaling alone only reduces the dynamic energy usage of a component. By reducing the frequency it is possible to run components reliably at lower voltages, decreasing both static and dynamic energy usage for the same duration. In general, there is no advantage to be gained by running at a higher than necessary voltage for the frequency in use. As such, voltage is scaled dynamically with the frequency, hence DVFS. The FPGA prototyped TIFU does not physically scale the voltage on the FPGA. Instead, the power model that is described in Section 3.1, assumes that the voltage has been scaled to the lowest value that reliably supports each frequency.

New frequencies can be specified to occur immediately or at a specified time. Time triggered frequency changes are achieved by programming the time at which the frequency should take place on either of the TIFU's PITs. The DVFS module contains registers for each interrupt specifying the numerator and denominator that should be used to scale the reference clock, in the event that the interrupt is raised. For example, to change to the maximum frequency at a specific point in the future based on the unscaled tile clock PIT, equal numerator and denominator values are programmed into the registers on the DVFS module associated with the tile clock PIT and subsequently the PIT is programmed with the time to commit the DVFS change.

The TIFU's control unit enables compound jobs consisting of multiple operations, such as clock gating the partition clock until a time in the future, to be achieved. This is useful for the clock gating example, as the processor that controls the TIFU will be unable to program a time to ungate the partition clock after it has been gated. Continuing with the clock gating example, once the control unit has received all the information required to perform the clock gating operation from the processor, it proceeds to request that the DVFS module halts the clock immediately, by setting the scaling numerator value to zero. After this has been performed, the control unit programs the future DVFS change using

the DVFS and PIT modules as described previously.

CoMik's processor virtualisation scheme requires that the frequency is scaled to a specified level whenever the interrupt is generated that signals the virtual processor context switch. The frequency level is programmable, and is stored in a register in the TIFU's DVFS module. Whenever the tile clock PIT raises an internal interrupt for the PIC module, the DVFS module switches to the specified frequency that is stored in the register. A separate register is provided by the TIFU's DVFS module to allow the partition clock PIT to trigger a frequency change, whenever it raises an internal interrupt for the PIC module.

2.2.8 PIC Module

The PIC module takes multiple interrupts as inputs and produces a single interrupt as an output. The PIC maps the status of each incoming interrupt signal to a bit in a status register that is accessible via the TIFU's FSL connection. In the CompSOC plaform, once the processor receives the interrupt produced by the TIFU, it reads the interrupt status register from the TIFU to find out which interrupt was raised.

CoMik's temporally isolated processor virtualisation also requires that the interrupts are virtualised in a temporally isolated manner. If the interrupts are not virtualised, then any interrupt could interfere with the timing of any virtual processor, whether it was intended for that virtual processor or not. To prevent this, the PIC is programmable, via its FSL connection, with a mask that is the same length as the interrupt status register. The mask instructs the PIC on which of its incoming interrupts should cause its output interrupt to be raised. When reading the interrupt status register, the mask is also used to mask out the interrupt bits that the PIC is not currently responding to. CoMik associates each virtual processor with its own interrupt mask. CoMik reprograms the PIC's interrupt mask as part of restoring the virtual processor's context.

For a more detailed description and analysis of composable virtualisation of interrupts, we refer the reader to [16].

2.3 CoMik: Predictable and Composable Virtualisation

The processor, like any other hardware resource in a system, becomes a point of contention when it is shared among multiple applications. In a mixed time-criticality system, where real-time and non real-time applications share the same hardware resources, the inter-application interference due to resource contention can make it difficult or impossible to provide real-time guarantees. Temporal composability is a solution to this problem, enabling resources to be shared among applications in a manner that prevents inter-application interference. This has been demonstrated for hardware components such as the Æthereal NoC [32, 43, 94, 95] and the Raptor memory controller [4, 33, 36, 63].

Section 2.3 contains an abridged and updated version of publication [79].

In this section, we present the CoMik microkernel that enables composable virtualisation of the processor resource. It divides the processor's time creating multiple virtual processors that can be used as dedicated resources by partitions. These virtual processors are cycle-accurately temporally-isolated, meaning that activity on concurrent virtual processors that do not belong to the same partition, cannot affect each other's timing by even a single cycle. A partition can therefore be temporally verified in isolation as the presence/absence of concurrent partitions does not affect the partition's timing.

CoMik's virtual processors provide an interface similar to that of the physical processor, as illustrated in Figure 2.2b. They require a pointer to a main function and optionally pointers to partition-level interrupt and exception handlers. As with a physical processor, the pointer to the main function points to the start of the partition's instructions that represent its functionality. Similarly, the pointer to the interrupt handler points to instructions that execute whenever a partition receives an interrupt, and the pointer to the exception handler points to instructions that execute whenever an exception is raised.

On multi-processor platforms such as the CompSOC platform, CoMik can create multiple virtual processors per physical processor. CoMik operates in a distributed manner per physical processor. Each virtual processor's utilisation of the underlying physical processor is configured using its slot allocation in the virtual processor TDM scheduling table.

Partitions consist of either a guest OS, or an application without an OS, as illustrated in Figure 2.2b. The timing of a partition is designated as being either *guaranteed* or *best-effort*. Virtual processors allocated to guaranteed partitions only use the TDM slots allocated to them, whereas the virtual processors allocated to best-effort partitions may use otherwise unused TDM slots in addition to their TDM allocation. By only using their allocated TDM slots, guaranteed partitions ensure that the behaviour of other partitions do not affect their timing.

In what follows, we present how the data required by CoMik to function is organised, in Section 2.3.1 (An example of the C code used to configure CoMik is presented in Appendix B). After that, we present how CoMik schedules virtual processors and maintains cycle-accurate temporal isolation between them, in Sections 2.3.2 and 2.3.3 respectively. We further describe how CoMik enables each partition to composably receive and handle interrupts, exceptions and use partition-level critical regions, in Section 2.3.4. CoMik's memory allocation scheme is presented in Section 2.3.5, describing how each partition can perform temporally isolated dynamic memory allocation. Finally, partition-level independent power management is described in Section 2.3.6.

2.3.1 CoMik Organisation

CoMik structures the information that it requires to function in a hierarchical manner, as illustrated in Figure 2.17. Information is grouped into control blocks, with global information stored in a CoMik Control Block (CCB) and virtual processor specific information stored in a Partition Control Block (PCB). Platform hardware information

necessary for driver functions, such as for the TIFU and DMAs, are also stored in control blocks.

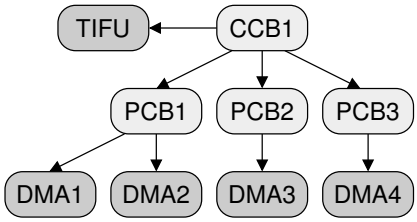


Figure 2.17: Example illustrating CoMik's Hierarchy of Control Blocks.

Only one CCB exists per processor containing information such as the heap and stack locations and sizes, the virtual processor TDM schedule, the CoMik slot and virtual processor slot durations, and pointers to the PCBs and hardware control blocks. Each PCB contains the information for a single virtual processor. This includes the virtual processor's heap and stack locations and sizes, operating frequency, pointers to its main function, interrupt handler and exception handler, and pointers to control blocks of dedicated hardware resources.

Temporal and energy accounting information is also stored within the CCBs and PCB as is explained in more detail in Chapter 3.

2.3.2 Virtual Processor and Partition Scheduling

As multiple virtual processors must share the same physical processor resource, TDM arbitration is used to decide which virtual processor is scheduled. TDM arbitration not only ensures a level of service but also when the service will be delivered. It is also relatively simple to formally analyse.

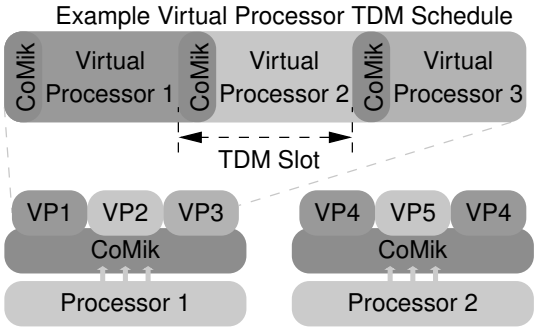


Figure 2.18: Temporal processor virtualisation.

Figure 2.18 illustrates how virtual processors are scheduled following a TDM schedule. In the diagram, physical processor 1 is virtualised as processors 1-3 and physical processor 2 is virtualised as processors 4 and 5. The TDM schedule for processor 1 is illustrated, showing that each virtual processor has one slot in the TDM table. At the start of each TDM slot, CoMik switches context from the previously scheduled virtual processor to the next virtual processor, ensuring cycle-accurate temporal isolation between them. This is explained in more detail in Section 2.3.3. Figure 2.18 also illustrates how a virtual processor may have multiple TDM slots, as shown for virtual processor 4 on physical processor 2. The slots do not need to be consecutive and may have any distribution within the TDM schedule.

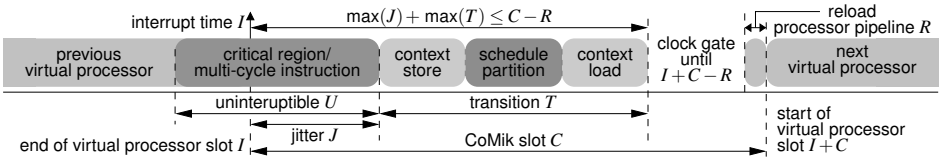


Figure 2.19: Cycle-accurate temporally-isolated virtual processor context switch.

If a slot in the TDM table has not been allocated to a virtual processor, or the slot is allocated to a virtual processor that will be clock gated for the entirety of the slot, the slot is deemed to be unused. These slots can be used by virtual processors that belong to partitions with best-effort timing. A round-robin arbitration scheme is used to decide which best-effort virtual processor gets the slot. Virtual processors that are allocated to partitions that are guaranteed to be temporally isolated cycle-accurately from other partitions cannot use the otherwise unused slots, as the availability of these slots depends on the presence/absence of other partitions and their timing.

CoMik does not perform any scheduling within partitions. Virtual processors are scheduled with cycle-accurate guaranteed or best-effort timing, but CoMik is agnostic to what a partition executes on them. This provides a clear separation of concerns. For instance, a guest Real-Time OS (RTOS) is free to use any partition-level scheduling scheme, but the timeliness of whatever processes/tasks/threads that it schedules is solely the responsibility of the partition. Care must be taken at design time when dimensioning the TDM table. For instance, shorter TDM slots allow for a higher throughput of virtual processor context switches, enabling lower virtual processor response times, but proportionally increases context switching overhead.

In practice, the default CoMik slot and partition slot lengths are `0x1000` and `0x10000` cycles, of the TIFU's reference clock, respectively. These are values that are not dimensioned for any particular purpose. With these numbers, 5.9% of the physical processor's time is spent in the CoMik slot as context switching overhead. The proportion of processor time spent in the CoMik slot can be reduced by increasing the partition slot length. Multiple virtual processors share the same physical processor, and are allocated slots in the TDM scheduling table. Increasing the partition slot length also increases the time for

a single TDM table period, which increases the worst-case response time of individual slots, and hence also of the virtual processors.

2.3.3 Cycle-accurate Temporal Isolation

Partitioning and cycle-accurate temporal isolation simplifies the temporal verification of real-time applications that share resources. This is partially achieved through scheduling, as explained in Section 2.3.2. CoMik's TDM scheduling scheme is regulated by a periodic interrupt that signifies a virtual processor context swap. Critical regions and multi-cycle instructions prevent the interrupt from being handled immediately. CoMik ensures that this jitter does not permeate to the next scheduled virtual processor, providing cycle-accurate temporal isolation.

Figure 2.19 illustrates how CoMik swaps virtual processor contexts. In this example, the scheduling interrupt arrives at time I, but cannot be handled immediately, as the processor is uninterruptible for a duration of U. This causes a jitter of time J. Duration U is variable, depending on the critical region or multi-cycle instruction. After the jitter, control passes to CoMik's interrupt routine that performs the virtual processor context switch. The context of the previous partition is stored. This entails storing the state of the physical processor's registers, etc., on the stack of the partition. CoMik then schedules the next virtual processor as described in Section 2.3.2, before restoring its context. In Figure 2.19, this transition from one virtual processor context to the other takes time T. Duration T is variable, due to variation in scheduling time.

If the following virtual processor started immediately after its context is loaded, its precise start time would depend on the jitter J and the transition time T. To provide complete cycle-accurate isolation, CoMik ensures that the resumption time of the virtual processor is independent of this variation. We achieve this by splitting the TDM slot into a fixed duration CoMik slot and virtual processor slot, as illustrated in Figure 2.18. The CoMik slot starts at the time the context change interrupt is raised and lasts for a fixed duration, C in Figure 2.19. The virtual processor slot starts precisely at time $I+C$. We achieve this by clock gating the physical processor after the next virtual processor's context has been loaded. The processor is ungated at time $I+C-R$, which is a constant R cycles earlier than the start time of the virtual processor slot $I+C$ to allow the processor pipeline to return to the state it was in when the virtual processor was swapped out. For the MicroBlaze processor, R is 2 cycles to account for the instruction fetch and decode stages of the pipeline, enabling the virtual processor slot to start where it left off, with the instruction at the execution stage of the pipeline.

To ensure that the virtual processor slot starts on time, the jitter and the context transition time must be less than or equal to the time at which the physical processor ungates, $max(J)+max(T) \leq C-R$ as is illustrated in Figure 2.19. A definitive upper bound $max(T)$ can be derived for the duration of the transition time. No inherent upper bound for the interrupt jitter $max(J)$ exists, as there is no inherent limit to the duration of an uninterruptible critical region U. The jitter bound $max(J)$ is therefore a design

decision that restricts the maximum length of partition-level critical regions. How this is enforced is explained in Section 2.3.4. The jitter bound should last minimally long enough to accommodate the processor's longest multi-cycle instruction (32 cycles for the integer division `idiv` instruction on the MicroBlaze processor). Increasing the duration of the jitter bound $max(J)$ also increases the necessary duration of the CoMik slot C. A trade-off therefore exists between accommodating a longer worst-case critical region and decreasing the CoMik slot overhead.

2.3.4 Partition-level Interrupts, Critical Regions and Exceptions

Partition-level (intra-partition) interrupt and exception handlers can be specified, allowing partitions to use independent event handling policies. Events are handled hierarchically, with control passing first to CoMik's event handlers, allowing them to decide whether the event is intended for a partition-level handler or CoMik.

Partitions are able to set partition-level timed interrupts and receive off-tile interrupts, e.g. to coordinate partition-level inter-tile communication. Interrupts can only be received while the partition is scheduled. Timed interrupts are stored as part of the partition's context and cannot occur while the partition is not scheduled. To prevent partitions from receiving off-tile interrupts intended for another partition, each partition has a mask that CoMik sets on the PIC to mask them out. Any timed or off-tile interrupts that arrive when the partition's virtual processor is not scheduled, are raised when the virtual processor is scheduled next.

Partitions can enable and disable their sensitivity to interrupts, creating partition-level critical regions. CoMik provides interrupt enabling and disabling functions that sets a mask on the PIC to prevent interrupts from being raised. To maintain cycle-accurate temporal isolation, partition-level critical regions should not be longer than the virtual processor context switch jitter bound $max(J)$, as explained in Section 2.3.3. The interrupt disabling mask is therefore set with an expiry time that is equal to the jitter bound minus the duration of the longest multi-cycle instruction. This prevents partitions with longer than dimensioned for critical regions from interfering with the timing of other partitions.

Having to forcibly interrupt a partition's critical region is a partition-level malfunction. CoMik raises a partition-level exception when the partition is scheduled next, allowing the partition's exception handler to decide how to handle the situation.

Depending on the physical processor, other exceptions can be raised, e.g. a stack pointer boundary exception. In the case that a partition decides to stop execution due to an exception or otherwise, it is simply removed from the TDM schedule and also from the best-effort list, if it was on it.

2.3.5 Memory Allocation

Partitions are allocated dedicated memory regions. A hardware MPU can be used to prevent partitions from accessing memory regions allocated to other partitions, but it

is orthgonal to achieving cycle-accurate temporal isolation, for instance the CompSOC platform prototype that we use for the experimentation in this thesis does not have an MPU. Without an MPU, malfunctioning partitions could overwrite information in memory allocated to other partitions, causing inter-partition temporal interference. The use of an MPU is therefore a trade-off between the area, cost and speed overhead of using one, and lower reliability without.

CoMik's heap and stack memory is statically allocated at design time. Memory for each partition is dynamically allocated on CoMik's heap, based on the partition's memory requirements. As with executing directly on the physical processor, each partition sets aside a heap and stack region within its memory allocation. Partition-level dynamic memory requests are allocated on the partition's heap. The time taken to allocate the memory is therefore independent of the memory requests of other partitions.

2.3.6 Partition-level Power Management

CoMik provides partitions with the ability to perform partition-level power management. Partitions are able to make DVFS decisions that change the frequency of the physical processor. The partition's frequency is stored and restored as part of the virtual processor's context, allowing every partition to perform independent power management.

Lowering the frequency increases the duration of the partition's critical regions. The CoMik slot is therefore assigned a constant frequency level at design time. Using Figure 2.19 for reference, the virtual processor context switch interrupt I causes the frequency to switch to the CoMik slot frequency. The rest of any partition-level critical region that executes after the interrupt I is performed at this frequency. The jitter bound $max(J)$, worst-case virtual processor context transition time $max(T)$, and the CoMik slot length C, are therefore independent of the partition frequency but should be dimensioned appropriately for the given CoMik slot frequency.

A partition can also clock gate itself for a duration of time, executing no instructions while it is gated. TDM slots of clock gated partitions are therefore made available to best-effort partitions. The virtual processor scheduler checks if the next scheduled virtual processor is clock gated, is due to ungate, has a raised partition-level interrupt, or is due to receive a timed partition-level interrupt, during its slot. If it will be gated for the duration of its slot, the slot is designated as *unused*, allowing a best-effort application to be scheduled, as described in Section 2.3.2.

If there are no best effort applications available to use the slot, then the idle partition is scheduled. This partition consists of an infinite loop (a single instruction that branches immediately to itself) that is clock gated. The idle partition is scheduled instead of the scheduled clock gated partition because loading and unloading the partition's instructions into the processor pipeline would cause some of the instructions to execute. The exact number of instructions depends on the specific processor pipeline. Scheduling the idle partition therefore prevents the clock gated partition's instructions from advancing, while still clock gating the slot. Clock gating for a duration longer than a virtual processor slot

length therefore causes the partition to have a longer worst case response time to off-tile interrupts. Any off-tile partition-level interrupts that are raised during the partition's unused slot are handled when the partition is scheduled next, as described in Section 2.3.4.

CoMik provides the composability part of composable and predictable power management. In the rest of this section we show how applications are modelled, executed and analysed using the dataflow MOC, but without the use of DVFS. In Chapter 4 we explain how these models are used to derive static VFS levels for use at run-time that are guaranteed to meet the applications timing requirements. This technique has no run-time computational cost, but is unable to consume slack in the schedule caused by dynamic variations in task execution time. In Chapter 5 we present a run-time power management technique that can make use of dynamic slack to perform conservative DVFS.

2.3.7 Temporally Modelling CoMik's Virtualisation

We proceed to explain how the temporal effects of CoMik's TDM arbitration on applications in a virtual platform are modelled. Using a HSDF latency-rate server abstraction of CoMik's TDM table, we describe how CoMik's timing affects the worst-case execution time of a computational task.

TDM Latency Rate Server

It is shown in [108] how the timing of TDM arbitration can be conservatively modelled as a latency-rate server that can be represented as a dataflow model, which is illustrated in Figure 2.20. A latency rate server is modelled as a HSDFG using two actors; a latency L actor and an inverse rate R^{-1} actor. The R^{-1} actor represents the inverse of the conservatively sustainable rate of the TDM table while the L actor represents the latency before the rate R is conservative. Auto-concurrency allows the L actor to fire multiple times concurrently, whereas auto-concurrency is prevented for the R^{-1} actor as it has a self-edge with a single initial token. The duration of the latency L actor is equal to the TDM table's worst-case response time minus the inverse of the conservative rate R^{-1}.

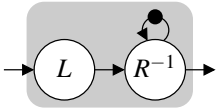


Figure 2.20: HSDFG representation of a latency-rate server

The sustained rate R of a TDM table is the number of cycles of service received S in a single table length divided by the number of cycles for a single table length T:

$$R = \frac{S}{T} \tag{2.9}$$

The latency L component of the latency-rate server is calculated as follows:

$$L = \overline{r} - R^{-1} \tag{2.10}$$

$$\overline{r} = T - S + 1 \tag{2.11}$$

where $\overline{r}$ is the TDM table's worst case response time that is calculated using Equation 2.11.

As an example, we continue by explaining how the timing of a virtual platform that is allocated two out of four slots in a CoMik TDM table, configured to have a CoMik slot of 1000 cycles and a virtual processor slot of 10000 cycles, is modelled as a latency-rate server. By applying Equation 2.9 to the properties of the table, the value of sustainable rate R of the table is calculated as follows:

$$\begin{aligned} T &= 4 \times (10000 + 1000) = 44000 \text{ cycles} \\ S &= \quad 2 \times 10000 \quad = 20000 \text{ cycles} \end{aligned}$$

$$R = \quad \frac{20000}{44000} \quad = \frac{5}{11} \quad \text{service cycles per cycle}$$

meaning that this virtual processor receives a sustainable rate R of five service cycles for every eleven cycles of the physical processor. Using Equation 2.11 and Equation 2.10 the latency L after which the rate R is conservatively sustainable is derived as follows:

$$\begin{aligned} \overline{r} &= 44000 - 20000 + 1 = 24001 \quad \text{cycles} \\ L &= \quad 24001 - 2.2 \quad = 23998.8 \text{ cycles} \end{aligned}$$

The HSDF model is annotated with the actor timings of 23999 cycles for the L actor. The duration of the virtualised actor timing R_v^{-1} depends on the duration of the computational task v that it represents, which is calculated as follows in Equation 2.12:

$$R_v^{-1} = t(v) \times R^{-1} \tag{2.12}$$

where $t(v)$ is the worst-case work of actor v and R^{-1} is the rate component of the latency-rate server that represents the timing of TDM table of the virtual processor. For example, If computational task v requires a worst-case work $t(v)$ of 500 cycles, then R_v^{-1} is calculated as follows:

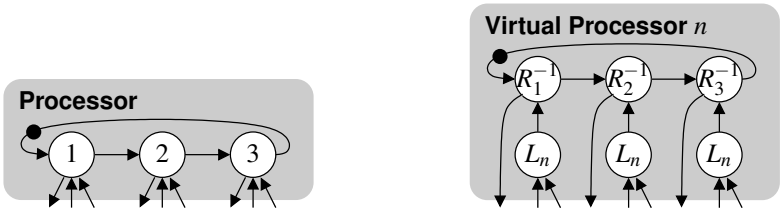
$$R_v^{-1} = 500 \times \frac{11}{5} = 1100 \text{ cycles}$$

and the R_v^{-1} actor is therefore annotated with the timing of 1100 cycles.

Latency-rate produces a simple TDM abstraction that can be overly pessimistic. A more complicated but less pessimistic model could be used instead [62].

HSDF Application Virtualisation

HSDF applications are composed of multiple tasks, and multiple of these tasks may be mapped onto a single CoMik virtual processor. Like the physical hardware processor, the virtual processor is a single resource, requiring task arbitration. On a virtual processor using POSe, tasks are scheduled following a SOS enabling the arbitration to be modelled as an HSDFG, such as the one illustrated in Figure 2.21a. Each actor cannot fire until it has at least one token on each of its incoming edges. Even if an actor is scheduled following the SOS, it might not be able to fire due to inter-tile data dependencies.



(a) Physical hardware processor **(b)** CoMik virtual processor

Figure 2.21: HSDFGs of a task SOS on a physical and virtual processor

CoMik's TDM table length and virtual processor slot allocation are configured per core. The CompSOC platform is also a Globally Asynchronous Locally Synchronous (GALS) system, and as such, the clock on each processing tile, and therefore CoMik's TDM tables, cannot be assumed to be synchronised. All this, coupled with dynamic variations in task execution time, mean that data can arrive and enable a task to fire at any moment within CoMik's TDM table.

Figure 2.21b illustrates how the HSDF latency-rate server is incorporated into the SOS, to bound the possibility that each task could experience the worst-case response time of the virtual processor's allocation in CoMik's TDM table. Each task actor v is split into a latency and rate actor. The incoming edges of the actors are connected to their respective latency actors that represent the worst-case duration before the task is processed at the sustainable rate of the virtual processor, once the task is enabled to fire. The outgoing edges from the task actor v are connected to the their respective rate actors. The rate actor of a latency-rate server is constrained by a self-edge, as illustrated in Figure 2.20. This is not necessary in Figure 2.21b as the control edges that govern the task execution SOS prevents auto-concurrency of the rate actors.

The latency actor L_n is annotated with the latency of the latency-rate server that conservatively models the timing of the virtual processor n, to which task v is mapped. This is calculated using Equation 2.10. The rate actor R_v^{-1} of each task v, is annotated with the number of cycles that it takes the task to perform its work at the virtual processor's sustainable rate, as calculated using Equation 2.12.

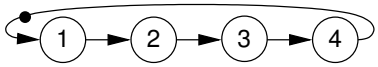
Timing Effects of CoMik TDM scheduling and a GALS System

CoMik operates in a distributed manner with the TDM table on each core potentially dimensioned in completely different ways, e.g. number of slots, slot length, frequency. By performing a latency-rate server abstraction of the TDM table on each core it is possible the timing behaviour of, and the interactions between all of the cores, even in a GALS system. The latency-rate abstraction can be quite pessimistic, as it assumes that any data communicated between cores always experiences the receiving core's worst-case response time, i.e it always arrives at the worst possible moment in the TDM schedule. If it is possible to symmetrically dimension all of the CoMik TDM tables (i.e. CoMik and virtual processor slot lengths, number of slots in the table, and allocate the application the exact same slots on each core) then the latency-rate pessimism can be reduced.

Figure 2.22a illustrates an HSDFG application that could be executed by POSe on CoMik's virtual processors. On a single core without CoMik virtualisation, it executes following a SPS as presented in Figure 2.22b. For a three core platform, if tasks one to four are executed on cores one, two, three and two, respectively, following the same SPS the same application would execute as illustrated in Figure 2.22c. Adding CoMik virtualisation to all of the cores, with symmetrically dimensioned an synchronised TDM tables results in the execution timing presented in Figure 2.22d. The schedule from Figure 2.22c is unchanged, except it is "sliced" and shifted apart by the symmetrical TDM schedules, due to the allocation's of other partitions, in much the same way as a Damien Hirst artwork[1]. It is therefore possible to bound the schedule using a latency-rate server abstraction, by modelling the duration of the tasks at the sustainable rate of the TDM schedule after a single delay of the latency of the latency-rate abstraction of the TDM schedule, rather than experiencing the latency every time there is a communication between cores.

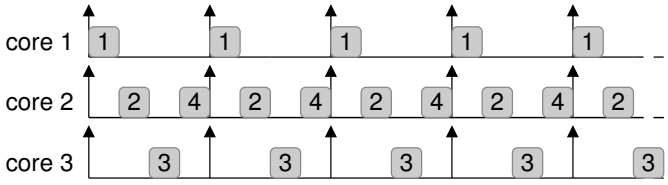
Unlike for the latency-rate abstraction in general, a GALS system provides a complication for this technique as it requires that all of the TDM tables are symmetrically dimensioned and synchronised. Due to the nature of a GALS system, it might not be possible to completely synchronise the TDM tables. Figure 2.22e illustrates what happens to the execution of the application from Figure 2.22d when the TDM tables are not synchronised. The schedule from Figure 2.22c is not so neatly partitioned as previously. If the difference in TDM synchronisation can be bounded, then the execution can still be conservatively bounded. As previously, the duration of the tasks are modelled using the sustainable rate of the TDM table, but a single initial delay of the TDM schedule's latency minus the synchronisation variation bound is used instead. Each inter-core communication is also assumed to experience the delay of the variation bound, which in general is less pessimistic than assuming that each inter-core communication arrives at the worst-case moment in the TDM schedule.

[1] Damien Hirst, "Some Comfort Gained from the Acceptance of the Inherent Lies in Everything", 1996

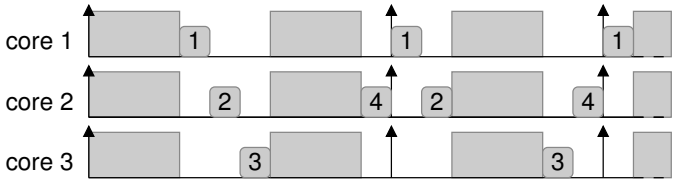


(a) HSDF application.

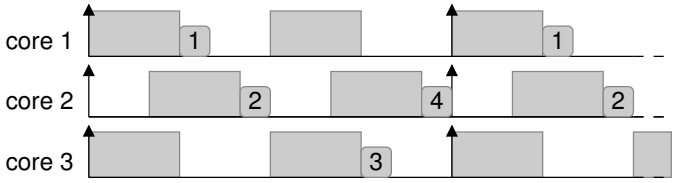
(b) Single core SPS.



(c) Multi-core SPS with synchronous starting time.



(d) Multi-core CoMik virtualisation of the SPS with synchronous starting time.



(e) Bounded synchronous CoMik TDM start time, with SPS executing as STS.

Figure 2.22: The affect of CoMik virtualisation on application graph throughput.

2.4 POSe: Dataflow Execution Library

We contribute the POSe library that simplifies creating new applications, or modifying legacy applications, to fit the dataflow MOC. Using the C programming language, the POSe library provides a dataflow modelling framework and MOE, that allows applications to be structured and executed in a manner that can be represented and analysed as a dataflow graph (An example of the C code used to configure POSe is presented in Appendix B). Using this framework enables applications to be structured with clearly defined computational tasks, that are representable as dataflow actors, that perform inter-task communication by transferring tokens over FIFO channels.

2.4.1 POSe Organisation

The basic building blocks of dataflow graphs are actors and the edges that describe their communication. Similarly, the basic building blocks of POSe's dataflow applications are computational tasks and the FIFO channels that they use to communicate. Internally, POSe maintains the properties and relationship of these building blocks in C structs that are known as control blocks. Applications, tasks and FIFOs are managed using Application Control Blocks (ACBs), Task Control Blocks (TCBs) and FIFO Control Blocks (FCBs), respectively. The end user does not interact with POSe's control blocks directly, but instead uses the provided Application Programming Interface (API) functions to access the information they contain.

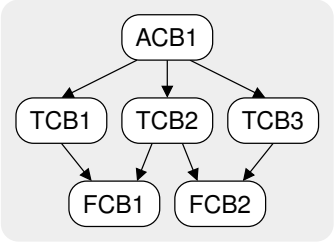


Figure 2.23: Example illustrating POSe's Hierarchy of Control Blocks.

POSe dynamically allocates the memory for its control blocks on the virtual processor's heap. Each control block is created and initialised by calling its initialisation function. Once initialised, the control blocks form a hierarchy that describes the structure of the application's dataflow graph. To create this structure the control blocks are initialised from the top of the hierarchy down.

2.4.2 Applications

The ACB must be created and initialised before all of the other POSe blocks as their initialisation depends on its existence. The ACB is initialised with the number of tasks

that the application contains and a pointer to the task scheduling policy. The number of tasks is used to dynamically allocate space for an array of pointers to the task TCBs.

The scheduling policy is a function that decides which task to execute next. POSe executes dataflow actors in a non-preemptive manner (preemption of the virtual processor is still possible), meaning that once scheduled, actors are able to execute to completion without having their execution preempted by another task. In POSe, scheduling is performed in a cooperative manner, with the scheduler being invoked after the completion of each task. For the application to be analysable as a dataflow graph, the task scheduler must also be analysable using dataflow. SOS is one such example of a dataflow analysable scheduling policy.

2.4.3 Actors as Tasks

POSe's dataflow applications consist of computational tasks that are C functions that take no arguments. Each task is represented by a single TCB in the application hierarchy. The example illustrated in Figure 2.23 is of an application that has three tasks. A pointer to the initialised TCB is placed in the ACB's TCB array.

Each task's TCB is initialised with an Identification (ID), a stack size and a pointer to the task's function. The stack that is used by the task's computation is allocated dynamically on the heap using the specified stack size. The pointer to the task's function is stored in the TCB so that the application's task scheduler can execute the function whenever the task is scheduled.

Tasks consume data for computation via incoming FIFO channels and produce data on outgoing FIFO channels. After the TCB has been initialised another function call is used to specify how many incoming and outgoing FIFOs the task uses. This information is used to dynamically allocate an array of incoming and an array of outgoing FIFO FCB pointers.

Tasks may retain state between execution iterations, but to enable task code reuse and to fit with the dataflow MOC, this must be accomplished using a self FIFO. This is a FIFO that has the task as both the producer and consumer. The task consumes its state information from the self FIFO at the start of its iteration and produces its updated state information into the self FIFO at the end of its iteration.

2.4.4 Dataflow MOE

The dataflow MOC has implied actions, such as the transfer of data between tasks, that must be carried out explicitly when executed on the CompSOC platform or other general hardware. To enable dataflow applications to be executed and analysed as dataflow graphs, we contribute the POSe MOE.

Figure 2.24, illustrates POSe's dataflow task MOE that models the explicit actions required to execute a dataflow application task on a processor. The POSe MOE is a one-to-one replacement of each task actor in a dataflow application. Figure 2.24 illustrates

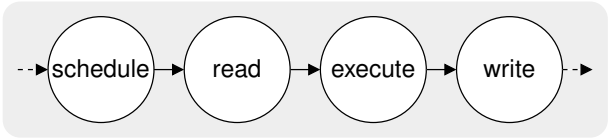


Figure 2.24: POSe dataflow task MOE

the dataflow model of the explicit actions that are required to execute a single dataflow actor on a physical hardware platform.

Dataflow actors can fire as soon as there are sufficient tokens on all of their incoming edges. The "schedule" actor in Figure 2.24 represents the action of checking the task's firing rule to find out if there is sufficient data in its incoming FIFOs and enough space in its outgoing FIFOs to complete a single execution iteration.

If there is sufficient data and space to pass the task's firing rule, the POSe MOE progresses to the "read" actor. Some of the task's data could be located in memories remote to the processor tile. The read actor represents the action of reading the necessary remote data into local memory for the task to be able to execute to completion without the need to fetch more data. In the CompSOC platform, data from local tile memories is accessible at a constant rate of 32 bits per cycle, simplifying the predictability of task execution time.

After all the necessary data is located in local memory, the POSes MOE proceeds to execute the task, as represented by the "execute" actor in Figure 2.24. The task consumes the data on its incoming FIFO channels, performs computation and then produces data on its outgoing FIFO channels. Some of the output data may need to be written to remote memory locations. The "write" actor represents this action.

Replacing each task actor in an application graph with the POSe MOE creates an explicit dataflow representation of the implicit actions that are required to execute a dataflow application on a physical hardware platform. The POSe MOE does not model resource constraints, e.g. the availability of the processor to execute a task. The POSe MOE should therefore be used in combination with dataflow modelling of resource constraints to form a single analysable dataflow model of the application when mapped on the hardware platform.

2.4.5 Dataflow MOE Implementation

The POSe MOE is an explicit sequential execution of functionality that is implicit to dataflow graphs. It describes the actions to be performed by the processor to execute individual tasks in the manner of a dataflow graph. The MOE's actions and order are the same for all tasks in the application. As such, the POSe OS provides a generic task wrapper to implement its dataflow MOE.

The generic MOE wrapper function takes the task's TCB as an argument allowing it to perform the specific actions to execute that task. The task must therefore be scheduled

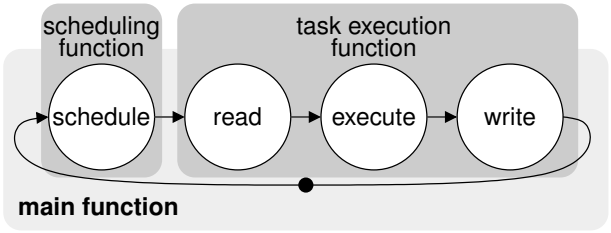


Figure 2.25: POSe dataflow task MOE implementation

before the task wrapper is called. Each application must specify a scheduling function. Before a task can be executed, the scheduling function is called to arbitrate which task is next. After a task is scheduled the task wrapper is called with its TCB as an argument. POSe's main function is an infinite loop that repeatedly calls the scheduling function followed by the task execution function, as illustrated in Figure 2.25.

Scheduling Function

POSe executes dataflow applications in a non-preemptive manner. As such, only one task of the dataflow application can execute on the processor at a time. Each application must therefore provide a scheduling function that arbitrates which task should execute next. The scheduling function takes a single void pointer as an argument, effectively allowing as many arguments as necessary to be passed to the function depending on how it interprets the data at the void pointer location. A scheduling algorithm selects a task, following its specific selection criteria. Following selection, it checks that the task's firing rules are met, i.e. that there is enough data in incoming FIFOs and space in outgoing FIFOs for a single task iteration. Depending on the scheduling algorithm, if the firing rules are not met the scheduler can either proceed to select another task or block by re-checking the firing rules. Upon selecting a task that can fire, the scheduling function returns the ID of the task to be executed.

The scheduling algorithm contained within the function can be as simple or as complicated as the programming language allows. In order to for the application to be analysable as a dataflow graph, it must also be possible to express the scheduling algorithm as part of the application's dataflow graph, e.g. SOS is incorporated into an application's dataflow graph in Section 2.1.1. Depending on the complexity of the algorithm this might not even be possible.

Task Execution Function

Once a task has been scheduled, the POSe task execution function is called to execute the task in a manner that is dataflow analysable. The task execution function is a generic wrapper around the tasks computational function. It takes the task's TCB as an argument,

enabling it to perform the specific actions necessary to execute the scheduled task.

Following POSe's MOE, the task execution function iterates through all of the task's incoming and outgoing FIFO FCBs, that are listed in the task's TCB, to check for the availability of sufficient data and space. If the FIFO's data is located in a remote memory location, it is also read into local memory.

POSe's scheduling function polls the task's FIFOs until sufficient data and space is available to execute the task, then executes the task's computational function. The task execution function finds the pointer to the task's computation function specified in the task's TCB. When the task's computational function completes, the execution thread returns to the task execution function.

A single task iteration consumes a single token of data from each of the task's incoming FIFOs and produces a single token of data on each of the task's outgoing FIFOs. After the task's computation function returns, the task execution function once again iterates over all of the task's FIFO FCBs to update the state of each FIFOs. The task execution function also pushes data produced from the task computation function to FIFO buffers located in remote memory, as necessary.

Power Management

Power management may be performed in either the user specified scheduling or task execution function. In Section 3.2 we explain how POSe maintains accounts of run-time temporal and energy information enabling power management functions to make DVFS decisions while still meeting an application's real-time requirements.

2.4.6 Multi-core Applications

POSe can be used to execute a single application on multiple processors. This might be necessary to meet real-time requirements, or in a resource constrained platform where all of the application's task instructions and data might not fit in the memory of a single core. POSe enables individual tasks from dataflow applications to be mapped and executed on different processors.

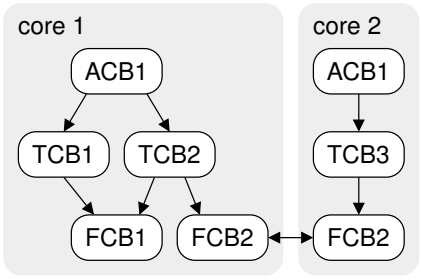


Figure 2.26: POSe's multi-core control block organisation

POSe keeps track of an application's tasks and FIFOs on a multi-core platform using control blocks, in the same manner as described in Section 2.4.1. POSe maintains a local ACB per core, that keeps track of the application's local tasks, as illustrated in Figure 2.26. In this example, two of the application's three tasks are mapped onto core 1 and the remaining task is mapped onto core 2. The task's TCBs, instructions and data are only located on the tile to which the task is mapped.

Tasks that are located on different cores must be able to communicate, in order to form part of a single application. This is achievable using POSe's C-HEAP FIFO channels, allowing a task on one core to be the producer, and a task on another core to be the consumer. POSe maintains a single FCB per core for each FIFO used by tasks located on that core. This means that an FCB is maintained on each core for FIFOs used to communicate between two cores. These FCBs contain copies of the FIFO's C-HEAP administration, with information on the FIFO buffer's address, token size and capacity. The scheme that is used to maintain the coherency of the read and write counter variables, that are used to indicate the buffer occupancy, depends on how the C-HEAP FIFO is mapped onto the shared memories, as described in Section 2.2.3.

2.5 Dataflow Modelling of Application and Platform

In the previous sections of this chapter we have described the techniques and components necessary to enable the execution of dataflow analysable applications. In this section, we contribute a technique that translates a dataflow application with a platform mapping, to a dataflow graph that combines the timing of the application and the software/hardware delays of the CompSOC platform, enabling temporal analysis of the application.

In this section, we explain how the original application dataflow graph is modified, by adding dataflow actors and edges, to take into account delays associated with inter-processor communication, POSe's dataflow execution framework and CoMik's virtualisation of the processor using TDM arbitration. We contribute an algorithm that describes how these modifications are applied to achieve the final dataflow analysable application.

2.5.1 Mapped Dataflow Application

The starting point of our technique is a mapped dataflow application. In order to execute the application on the CompSOC platform, each task is mapped on a processor. The resultant inter-tile FIFO communication must also be mapped onto physical hardware components, such as DMAs.

The hardware mapping of real-time applications affects their ability to meet their timing requirements. Much work has been carried out on automated application mapping [18, 53] and we acknowledge the importance of application mapping in meeting real-time requirements and therefore the availability of scheduling slack to perform power

management. The work in this thesis assumes as a starting point that the application is already mapped to the platform and that the tasks are already scheduled following a per core dead-lock free SOS.

An example mapping, for the dataflow application that is illustrated in Figure 2.27a, is presented in Figure 2.27b. Each task is assigned to a processor, with tasks 1 and 3 assigned to the processor on tile 1 and tasks 2 and 4 assigned to the processor on tile 2. Tasks are scheduled following a SOS per tile with additional control edges added to the dataflow graph to model the processor resource constraint, as described in Section 2.1.1.

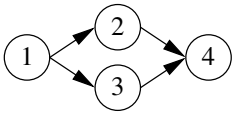
The mapping causes the edges between tasks 1 and 2 and tasks 3 and 4 to span multiple tiles. POSe implements dataflow edges as finite capacity C-HEAP FIFOs. This is modelled as described in Section 2.1.1 by adding an additional edge in the opposite direction of each edge that spans multiple tiles. The finite capacity of each FIFO is represented by the number of initial tokens on the additional edge, with each token representing the availability of space for a single token in the C-HEAP FIFO. The DMA and memory resources required for inter-tile communication also form part of the application mapping.

2.5.2 Incorporating Inter-tile C-HEAP Communication

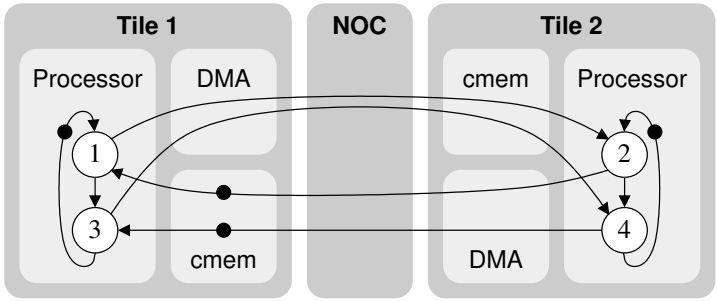
The application mapping illustrated in Figure 2.27b has two inter-tile C-HEAP FIFO communication channels. The capacity of the C-HEAP FIFO is modelled using an additional reverse edge with the capacity of the FIFO represented by the number of initial tokens on this edge, as described in Section 2.1.1. The precise timing and modelling of the physical implementation of the FIFO depends on how the components of C-HEAP FIFO are mapped. Section 2.2.3 explains how three C-HEAP configurations are modelled as HSDFGs. We proceed to explain how these models are incorporated into a combined application and CompSOC platform HSDFG.

Algorithm 2.1 Incorporate Inter-Tile C-HEAP

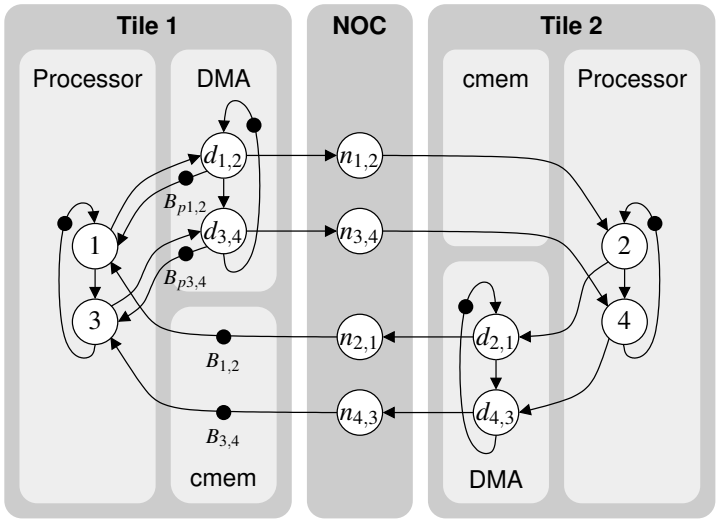
Require: input HSDFG G
  for all C-HEAP edge pairs $\{(p,c),(c,p)\} \subseteq E$ **do**
    $G \leftarrow G \setminus \{(p,c),(c,p)\}$
    $G \leftarrow G \cup \text{getCHeapHSDFG}(\{(p,c),(c,p)\})$
  end for
  for all processors P **do**
    $V_p \leftarrow \text{getActors}(G,P)$
    for all DMAs D local to P **do**
      $V_d \leftarrow \text{getActors}(G,D)$
      $G \leftarrow \text{createSOS}(G,V_d)$
      $G \leftarrow \text{orderSOSactors}(G,V_p,V_d)$
    end for
  end for



(a) Example HSDFG application

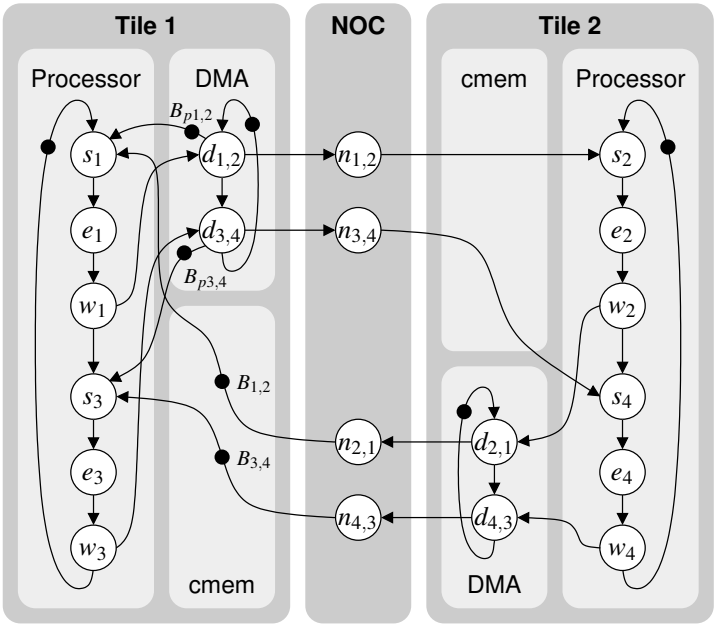


(b) Application and resource constraints mapped onto CompSOC hardware resources

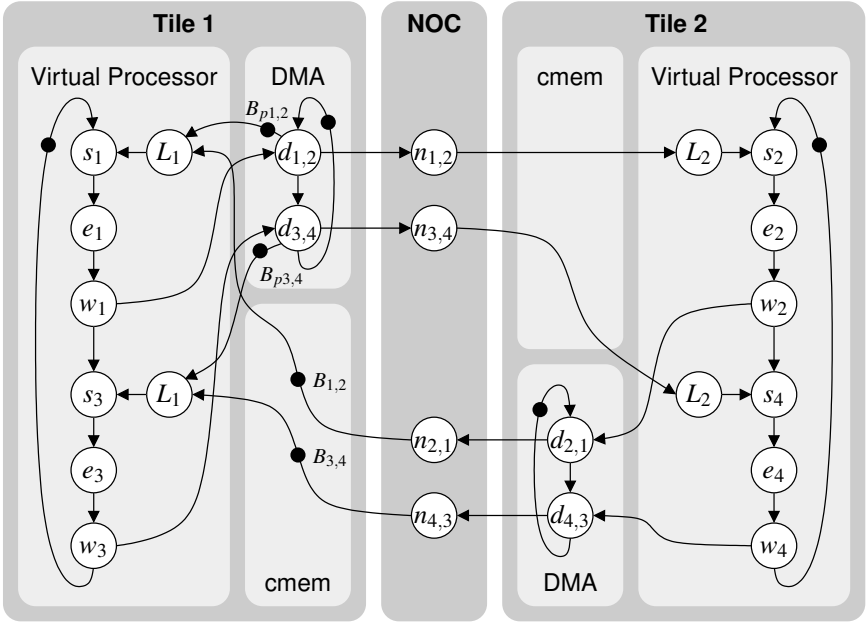


(c) Incorporating hardware timing in the application's dataflow graph

Figure 2.27: Combined application and CompSOC platform dataflow modelling



(d) Task dataflow actors replaced by POSe dataflow MOE



(e) Latency-rate abstraction of CoMik TDM table

Figure 2.27: Combined application and CompSOC platform dataflow modelling

Algorithm 2.1 describes the steps necessary to incorporate the C-HEAP models from Section 2.2.3 into the mapped application HSDFG from Figure 2.27b. The algorithm iterates over all of the C-HEAP edge pairs, that represent the forward data path and the reverse space path of a single C-HEAP FIFO, replacing them with the appropriate C-HEAP HSDFG. In Algorithm 2.1, $\{(p,c),(c,p)\} \subseteq E$ is a set containing the edge pair, from producer p to consumer c and back again, that represent a single C-HEAP FIFO. The algorithm first removes the edges belonging to the C-HEAP edge pair from the mapped graph G then merges the detailed HSDFG representation of the C-HEAP FIFO with graph G. getCHeapHSDFG: $E \times E \rightarrow G$ returns the appropriate HSDFG representation of the C-HEAP FIFO for the C-HEAP edge pair.

Some of the C-HEAP connections might use the same DMA, requiring that a cycle of resource constraint edges are used to ensure that all the DMA actors fire sequentially. Algorithm 2.1 iterates over all the DMAs of all the processors, finding all the actors that belong to each DMA, before forming a single SOS per DMA and ordering the actors relative to the SOS of the task actors on the local processor. The function getActors: $G \times H \rightarrow V$ takes a graph G and a hardware resource H and returns the set of actors V from G that represent the timing of H. In Algorithm 2.1, this is used to find the set of actors V_p that represent the processor P and the set of actors V_d that represent the DMA D. The function createSOS: $G \times V \rightarrow G$ takes an HSDFG G and a subset of actors V and returns a graph G with a cycle of resource constraint edges to the actors V in G, forming a SOS. The ordering of the transactions that are carried out on each DMA depends on the order of task execution on the local processor that programs the DMA. The function orderSOSactors: $G \times V \times V \rightarrow G$ takes the graph G and two subsets of actors V, ordering the SOS of the second set of actors relative to the first. This is used in the algorithm to order the SOS of the DMA actors V_d relative to the ordering of the task actors on the local processor V_p.

Figure 2.27c is what results when Algorithm 2.1 is applied to the mapped application graph from Figure 2.27b. The two inter-tile C-HEAP edge pairs are replaced with the C-HEAP HSDFG that models the C-HEAP communication using local scratchpad memories only, as described in Section 2.2.3. Both C-HEAP FIFO channels use the same DMA on each processing tile. Following Algorithm 2.1 the actors modelling DMA transactions are added to a SOS per DMA and ordered relative to the task execution order of the local processor.

2.5.3 Incorporating the POSe MOE

With the inter-tile communication modelled in Figure 2.27c, we proceed to explain how the POSe MOE is incorporated into the application and CompSOC platform HSDFG. The original HSDFG of the application from Figure 2.9, models the application tasks and their communication. These are the same actors modelling processor tasks in Figure 2.27c. Dataflow applications are executed on the CompSOC platform using the POSe OS, with the timing overhead modelled using the POSe MOE HSDFG, as described in Section 2.4.4.

Algorithm 2.2 describes how the POSe MOE is incorporated into the combined application and CompSOC platform HSDFG.

Algorithm 2.2 Incorporate POSe MOE

```
Require: input HSDFG G
  for all processors P do
    V_p ← getActors(G, P)
    for all actors v ∈ V_p do
      G ←substitute(G, v,POSeTaskMOE(v))
    end for
  end for
```

Algorithm 2.2 iterates over all actors in the HSDFG that model processor tasks, substituting each of these actors with the POSe MOE. The function substitute: $G \times V \times G \rightarrow G$ takes a graph G with an actor V to be substituted with a graph G and returns the graph G with the completed substitution. The incoming edges of the substituted actor are transferred to the first actor in the actor order of the replacement graph, and the outgoing edges to the last actor. The function POSeTaskMOE: $V \rightarrow G$ takes an actor and returns the POSe MOE graph for that actor.

As described in Section 2.4.4, the POSe task MOE consists of four actors; schedule s_n, read r_n, execute e_n and write w_n, for task ID n. The resultant graph, when Algorithm 2.2 is applied to the graph from Figure 2.27c, is illustrated in Figure 2.27d. The POSe MOE read actors are omitted in this instance as the tasks in the application graph example do not read any data from remote locations.

2.5.4 Incorporating CoMik TDM Timing

The combined application and CompSOC platform modelling described in the previous sections explain how to model the application executing using the POSe OS directly on the CompSOC hardware, as illustrated in Figure 2.27d. We proceed to explain how CoMik virtualisation can be taken into account in the combined application and CompSOC platform HSDFG.

CoMik virtualises a single processor into multiple virtual processors using TDM arbitration, i.e. the virtual processors time share the physical hardware processor. Section 2.3.7 describes how the timing of CoMik's virtualisation is modelled as a latency-rate server that can be incorporated into the combined application and platform HSDFG. Algorithm 2.3 describes the steps necessary to incorporate CoMik's latency-rate server abstraction into the application HSDFG, e.g. translating Figure 2.27d into Figure 2.27e.

Algorithm 2.3 iterates over all of the virtual processors in the graph updating the timing annotation of the actors that model the timing of tasks executing on the physical processor to the timing of the tasks on the virtual processor. The function updateLRtiming: $G \times V \rightarrow G$ takes the HSDFG G and an actor v and updates its annotated timing to correspond with

Algorithm 2.3 Incorporate CoMik TDM Timing

Require: input HSDFG G
  for all processors P **do**
    $V_p \leftarrow$ getActors(G,P)
    for all actors $v \in V_p$ **do**
      $G \leftarrow$updateLRtiming(G,v)
    end for
    for all edges $(i,j) \in G$ **do**
      if $i \notin V_p$ **and** $j \in V_p$ **then**
        $G \leftarrow$substitute$(G,j,$CoMikLRserver$(j))$
        $V_p \leftarrow V_p \setminus j$
      end if
    end for
  end for

its execution time on the virtual processor, returning the updated HSDFG G. The latency actors, of CoMik's latency-rate server abstraction, are appended to the incoming inter-tile communication edges. Algorithm 2.3 iterates over all the edges in the graph (i,j) for every processor P. For the set of actors V_p belonging to P, if the producing actor i is not in V_p and the consuming actor j is in V_p then the edge (i,j) is an incoming inter-tile communication edge. The function CoMikLRserver: $V \rightarrow G$ takes an actor and returns the latency-rate server HSDFG for that actor, which the algorithm then substitutes for j in graph G. The actor j is then removed from the set of actors V_p as the substitution only needs to occur once per actor.

2.5.5 Combined Application and CompSOC Platform HSDFG

Each of the algorithms presented in the previous sections take an HSDFG and modifies it to produce an updated HSDFG that incorporates more of the detail of the application timing on the CompSOC platform. Algorithm 2.4 presents how each algorithm is simply invoked sequentially to produce a combined application and CompSOC platform HSDFG, such as Figure 2.27e, from a mapped application HSDFG, such as Figure 2.27b.

Algorithm 2.4 Combined Application and Platform HSDFG

Require: mapped application HSDFG G
  $G \leftarrow$ incorporateCHeap(G)
  $G \leftarrow$ incorporatePOSe(G)
  $G \leftarrow$ incorporateCoMik(G)

Algorithm 2.4 first invokes the function incorporateCHeap: $G \rightarrow G$ that uses Algorithm 2.1 to incorporate the timing of inter-tile communication using the C-HEAP

communication protocol. Algorithm 2.4 then invokes the incorporatePOSe: $G \rightarrow G$ function that uses Algorithm 2.2 to incorporate the timing overhead of the POSe MOE. Finally, Algorithm 2.4 invokes the function incorporateCoMik: $G \rightarrow G$ that uses Algorithm 2.3 to incorporate the timing of CoMik's TDM virtualisation into the application graph. A combined application and CompSOC HSDFG is the end result.

2.6 Related Work

Interference between applications is a particular problem for safety-critical applications, such as those found in the automotive [83] and aeronautical [52, 88, 92, 110] industries. The strictest standards are found in the avionics industry, and ARINC specification 653 [10], is an avionics industry standard for the implementation of temporal and spatial partitioning [52, 110]. The standard also specifies requirements for interfaces, libraries and programming languages, enabling interoperability between ARINC 653 compliant systems [88]. LynxOS-178 [66], VxWorks 653 [109], INTEGRITY [40] and PikeOS [96] are commercially available ARINC 653 compliant RTOSs.

2.6.1 Composable and Predictable Systems

Two well known composable and predictable real-time system approaches are Time Triggered Architecture (TTA) [57] and PRET [27]. TTAs have been researched since 1979 [58], and is a well established technique used when designing safety critical embedded systems, e.g. in the avionics and automotive industry. Computation in a TTA is triggered by the tick of a clock (known as a global clock) in much the same way as synchronised hardware. Timing isolation is achieved by only permitting components to interact at pre-computed points in time, as governed by the global clock. TTA architectures therefore use non work-conserving inter- and intra-application scheduling, whereas POSe's intra-application scheduling is work conserving, i.e. an earlier task finish leads to an earlier start of subsequent tasks.

Timing repeatability to reduce the complexity of verifying real-time systems, is the focus of the PRET programming model [64] and platform [65]. Timing isolation and repeatability is achieved in PRET by dedicating processor resources, e.g. independent hardware threads. The PRET methodology needs ISA support for precise timing control, such as a deadline instruction, and requires processor modification to supports such instructions if they do not exist. With the CompSOC platform's deterministic hardware timing, and our strict notion of timing composability, where applications do not interfere by even a single cycle, the CompSOC platform also has repeatable timing (see Section 6.1). CoMik time shares the processor instead of dedicating processor resources, which lowers average performance, but does not have a hardware limitation on scalability. The TIFU enables precision timed actions to be performed, such as halt until deadline, without requiring modification to the processor.

2.6.2 Virtualisation

Outside of safety critical domains, temporal and spatial partitioning is used for embedded system virtualisation [41,45,77]. The OKL4 microvisor [46] is a virtualising microkernel, that is developed for use on mobile phones. It enables mixed time-criticality applications to execute on virtual machines as if they were running directly on the hardware platform. OKL4 uses thread-level partitions, with time slices allocated per-thread. Threads are scheduled following a priority based pre-emptive schedule. As with ARINC 653, precisely when the partition receives service depends on the presence/absence of other partitions. OKL4 permits inter-partition communication, enabling timing interference between the communicating partitions.

CoMik combines partition-level cycle-accurate temporal isolation with a virtualised processor interface. Partitions are allocated one or more dedicated virtual processors on one or more shared physical processors. A limitation of CoMik's cycle-accurate isolation is that guaranteed partitions may not receive information from any other partitions, but they may send information in a non-blocking manner to best-effort partitions. Best-effort partitions may communicate freely, but can experience inter-partition timing interference due to the communication.

2.6.3 Formal Abstraction

Real-time calculus [99] is another formalisation method that can be used abstract real-time systems to provide timing guarantees. We use dataflow as our formalism method for the same reasons as given in [14]. The main advantage of dataflow is that it permits cyclic data dependencies and can hence model back pressure. Our application abstractions are able to use this back pressure to model FIFO occupancy Section 2.1.1. Real-time calculus cannot capture back pressure, but can model the timing effect of buffer capacities in some specific cases [100, 101].

2.7 Summary

In this chapter, we have presented the CompSOC platform for executing mixed time-criticality applications. We show how the dataflow modelling paradigm is used to model real-time applications enabling worst-case timing analysis to be performed to provide guarantees. Dataflow is a restrictive MOC, and as such we describe the POSe OS that enables applications to be structured and executed in a dataflow analysable manner. As a mixed time-criticality platform, multiple applications with various timing requirements may execute on the same processor. We present the CoMik microkernel that composably virtualises the hardware processor into multiple virtual processors. Applications executing on these virtual processors do not interfere with each other's timing by even a single cycle. We describe how to create a combined application and CompSOC platform dataflow graph that enables temporal analysis of the application taking into account the timing overhead

of the practical implementation. The power management techniques presented in the rest of this thesis use the CompSOC platform's temporal analysability to perform DVFS while still meeting real-time application requirements.

CHAPTER 3

Composable Time, Energy and Power Accounting

Composability enables applications to execute independently of one another while using the same physical resources. In this chapter, we describe how the CompSOC platform enables each application to maintain independent temporal, energy and power performance budgets for composable power-management of real-time applications.

Power management of embedded systems is a trade-off between an increase in power consumption or an increase in execution time (due to a reduction in operating frequency). Due to the relation of time, energy and power, a reduction in power may even lead to an overall increase in energy consumption for the amount of work performed. The exact relationship between power and frequency depends on many factors including the type of transistors used, feature size, and how the transistor is specifically dimensioned for implementation. As many of the details are beyond the scope of this thesis, we first present a general overview.

3.1 DVFS Power Model

The DVFS mechanism enables trading a decrease in operating frequency and voltage for a decrease in power consumption. This trade-off is (usually) monotonic, meaning that a reduction in operating frequency can only lead to a power consumption that is less than or equal to what it was before. When the DVFS mechanism is applied to a processor, an application can save power by simply lowering its operating frequency without needing

to be aware of how much power it will actually save. Unfortunately, without a power model, it is impossible to say if this reduction in power consumption would also translate to a reduction in energy consumption for the amount of work performed. The problem of reducing power/energy consumption is further complicated on multi-core systems running real-time applications with inter-core data dependencies, due to the complex trade-offs available when using multiple DVFS islands. This is the topic covered in Chapter 4 and Chapter 5. To be able to make informed DVFS trade-off decisions, a power model is necessary.

The power consumption of a System on Chip (SoC) is complex to model as it is a collection of many smaller components with various interactions and power behaviours. Creating detailed and accurate power models of electronic components is beyond the scope of this thesis, but an understanding of the power modelling is necessary to demonstrate the applicability of our techniques to physical systems. The power/energy reduction techniques presented in this thesis do not rely on a specific power model and are suitable for use with any power model that can be expressed as a convex function of monotonically decreasing power consumption as a consequence of decreasing operating frequency (and hence voltage). While our techniques do not rely upon a specific power model, it is important that they are applicable to current technologies and some foreseeable future technologies. In the rest of this section, we present a basic overview of power modelling and the problems faced by decreasing feature sizes.

3.1.1 Power Modelling Overview

SoCs are mainly composed of transistor based logic and interconnecting paths. Power is primarily consumed performing logic functions and hence in the transistors. Much work has therefore been carried out to model the power performance at the transistor level. The alpha-power law [17,91] is a commonly applied Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) power consumption model. For instance, it is applied in [50,72] to model the power consumption of Complementary Metal-Oxide-Semiconductor (CMOS) circuits. This is modelled in [50,72] as the sum of the circuit's dynamic switching power consumption P_{dynamic}, its static leakage power consumption P_{static} and in [50] a constant power consumption P_{on}:

$$P_{\text{circuit}} = P_{\text{dynamic}} + P_{\text{static}} + P_{\text{on}} \tag{3.1}$$

The dynamic power of the processor P_{dynamic} caused by gate switching is calculated as follows:

$$P_{\text{dynamic}} = \alpha C V_{dd}^2 f \tag{3.2}$$

where α is the circuit's average switching activity, C is the circuit's total capacitance, V_{dd} is the supply voltage and f is the operating frequency. When performing DVFS, both

V_{dd} and f are scaled. For a requested f, a V_{dd} should be chosen that enables the lowest power consumption. This can be achieved at run-time by performing a table lookup of predetermined levels.

The static power consumption is more difficult to model, as it is a result of multiple complex leakage phenomena in Field-Effect Transistors (FETs). Six different types of MOSFET leakage current are described in [90]. The CMOS circuit power models described in [50, 72] model sub-threshold leakage, reverse bias junction leakage and in [72] also gate-oxide tunnelling leakage. The static processor power leakage P_{static} is modelled simply as the sum of the power leakages from all gates in the circuit in [71]:

$$P_{\text{static}} = \sum_{\forall \text{gates}} (|V_{bs}|I_j + V_{dd}I_{sub} + V_{dd}I_{gate}) \tag{3.3}$$

where V_{bs} is the body bias voltage, I_j is the reverse bias junction current, I_{sub} is the sub-threshold leakage current and I_{gate} is the gate-oxide tunnelling current.

The constant power P_{on} is used to model the rest of the circuit's power consumption that is not covered by P_{static} and P_{dynamic}, such as the power consumption due to logic and clocking paths.

Temperature has an affect on threshold voltage and on electron and hole mobility. It therefore also affects power consumption and switching duration. The technique presented in Chapter 4 optimises frequency levels off-line and is therefore unable to account for run-time temperature variations. To ensure that power consumption is not underestimated, the worst-case power consumption for any temperature for each frequency should be used.

3.1.2 DVFS Hardware Support

A hardware actuator is required to change the physical voltage and frequency levels of a power island. Multiple power islands can exist per SoC with each having independent voltage and frequency levels. The CompSOC platform is designed to have a power island per tile, enabling each processor to independently perform DVFS. Clock Domain Crossings (CDCs) are used to enable communication across clock domain boundaries. There are multiple methods to implement CDCs. One common method is to use dual ported SRAM memories at the clock domain boundary, with one port in each of the clock domains.

In the case of the CompSOC platform, the TIFU provides the hardware actuator that sets the voltage and frequency levels. For practical reasons, the TIFU in the FPGA prototyped CompSOC platform does not actually perform voltage and frequency scaling of FPGA regions. As the FPGA is being used to prototype an ASIC platform, any measured power savings would not be indicative of those achievable on an implementation of the ASIC platform. Instead, the timing effects of frequency scaling are achieved using clock division, as explained in Section 2.2.4 and the platform's power consumption is calculated from a model, as described in Section 3.1.4.

CoMik enables independent voltage and frequency levels between virtual processors, requiring relatively fast DVFS changing to maintain composability with a low temporal overhead. Hardware techniques presented in [73] and [104] enable voltage and frequency scaling in hundreds of nanoseconds, translating to frequency changes in tens of cycles at a frequency of 100 MHz. The technique presented in [73] achieves DVFS transitions of approximately 200 ns using 130 nm technology, in simulation. A physical implementation of the technique presented in [104] was achieved in a 65 nm technology test-chip, and performs DVFS transitions in approximately 100 ns. They achieve this using a voltage-hopping technique that approximates the frequency between discrete frequency-levels using dithering patterns to produce a linear interpolation [104]. The power consumed is also a linear interpolation, making the frequency-power model a piecewise linear curve.

3.1.3 Decreasing Feature Sizes and DVFS

As Moore's law continues, transistor sizes are ever decreasing [1]. For the MOSFET, this causes many of the leakage currents to increase [72]. This has led to the development of FETs with a different structure to the MOSFET, to minimise leakage currents. At the time of writing, the FinFET and the Fully Depleted Silicon-on-Insulator (FD-SOI) transistor are vying to replace the MOSFET at smaller feature sizes [67]. The specific benefits of each are beyond the scope of this thesis and are still being debated. Importantly, for the power management techniques proposed in this thesis, both technologies support DVFS [55, 67].

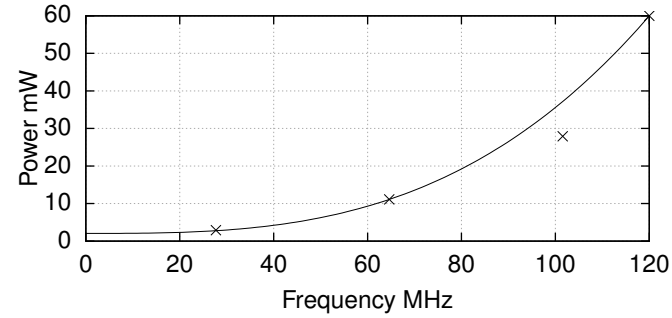
At present, FD-SOI appears set to become the technology of choice for future SoCs from STMicroelectronics [67]. This is at least partially down to its suitability for power-management using DVFS, due to its wide voltage scaling range [28]. This makes FD-SOI suitable for low-power applications, such as mobile devices. A 2.6 GHz ARM Cortex A9 processor implemented in 28 nm FD-SOI technology that exhibits a wide voltage scaling range is described in [49], demonstrating that FD-SOI is a feasible technology for low-power mobile processors.

3.1.4 Convex Frequency-Power Model for CompSOC DVFS

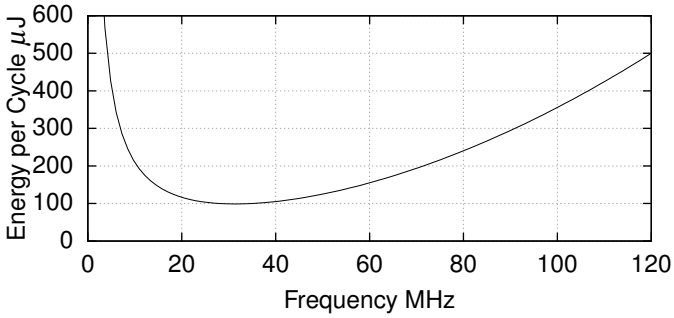
The power-management techniques described in Chapter 4 and Chapter 5 require a convex power model for use with the disciplined convex programming technique. Our techniques calculate frequency scaling levels to obtain the lowest power consumption. It is assumed that some form of table exists that translates the desired frequency to the appropriate supply V_{dd} and bias V_{bs} voltages required to support the frequency. Independently scaling V_{dd} and V_{bs} produces a frequency-power trade-off region [72], rather than a single convex frequency-power trade-off curve.

A method is presented in [6] that demonstrates for simulated 28 nm FD-SOI technology how discrete points can be selected within the frequency-power trade-off region and linearly interpolated, using voltage-frequency hopping techniques [104], to form a

piecewise convex frequency-power trade-off curve. The disciplined convex programming method used by our technique in Chapter 4 requires a continuous convex frequency-power model. Piecewise convex models can be conservatively fitted using a continuous function to enable their usage with our technique.



(a) Relationship between power and frequency.



(b) Energy consumption per cycle of work.

Figure 3.1: Continuous models suitable for convex optimisation.

The techniques presented in this thesis work with any convex frequency-power model that can be expressed using disciplined convex programming [39]. In order to demonstrate our techniques, we use the continuous convex frequency-power model as illustrated in Figure 3.1a. Based on the power modelling equations presented in Section 3.1.1, Figure 3.1a presents a continuous third-order polynomial function that is curve-fitted to the discrete convex operating points from [104]. The points in [104] are on a normalised frequency and power scale, from zero to one. To be able to give an idea of the sort of savings that could be expected in realistic system, we multiply the normalised scales with appropriate values.

We multiply the frequency scale by 120, as the FPGA prototyped CompSOC platforms used in this thesis, have a maximum frequency of 120 MHz. As the MicroBlaze processors are soft cores (that can be configured in many different ways), there is no definitive

appropriate value to multiply the power scale. For this purpose we use a measured power value of 60 mW for an ARM Cortex-A8 processor running the Dhrystone benchmark at 125 MHz. This measurement was carried out by Texas Instruments for the OMAP3530 chipset [98]. Scaling the operating points from [104] and fitting the polynomial frequency-power function $P(f)$ produces the following model:

$$P(f) = 3.353 \times 10^{-5} f^3 + 2.065 \tag{3.4}$$

where f is the operating frequency in MHz and the produced power $P(f)$ is in mW. This power model is illustrated in Figure 3.1a. *We do not claim that this power model is accurate for any particular processor, nor is it necessary that this particular power model is used for the functioning of our techniques. The power model given by Equation 3.4 is used throughout this thesis for illustrative purposes only.* While the ARM Cortex-A8 is a high-performance mobile processor and the MicroBlaze is a simpler soft-core, the measured power value for the ARM Cortex-A8 is useful to put the power model in the right area of approximation for power consumption by contemporary processors.

The convex optimisation techniques that we present in the following chapters can also be used to minimise the amount of energy consumed for the amount of work performed. The convex power function presented in Figure 3.1a translates to a convex energy per cycle-of-work function, presented in Figure 3.1b, making it suitable for use with our convex optimisation based technique. From Figure 3.1a and Figure 3.1b can be seen that while lowering the frequency translates to a reduction in power, it does not always translate to a reduction in energy for the amount of work performed. As the frequency is reduced the duration of an individual cycle increases until eventually the static power consumption causes further frequency reduction to cause an increase in energy required to perform a single cycle.

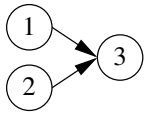
3.2 POSe Accounting

For real-time applications, trading a reduction of power consumption for a reduction in performance through DVFS must be done within the constraints of the application's timing requirements. Performing run-time DVFS therefore requires information about the application's current timing performance, e.g. the run-time power-management schemes described in Chapter 5 use this run-time information to conservatively perform DVFS. In this section, we describe how POSe provides the ability to monitor an application's timing performance at the task and application levels.

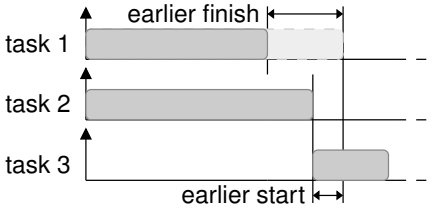
3.2.1 Task-level Timing

POSe tasks can be annotated with a worst-case work that is stored in the task's associated TCB. The task's worst-case work is measured in cycles of work that the task executes in

its worst-case. The task's WCET therefore depends on the task's worst-case work and the frequency at which the task is executed. The TIFU hardware timer, that counts the scaled clock signal, counts the number of cycles of work that the task performs at run time. The difference between the task's measured cycles of work and the task's worst-case work is observed temporal slack.



(a) Actor 3 has two data dependencies.



(b) Earlier start of task 3 is less than or equal to the earlier finish of task 1 (or task 2 if it was early).

Figure 3.2: Task-level timing slack observation.

Task-level slack that is observed in this manner is not always useful to perform conservative frequency scaling. Tasks can have multiple data dependencies, and an earlier finishing time of one of the preceding tasks does not guarantee an earlier starting time of the dependent task[1]. Figure 3.2 illustrates an example of an HSDF application in which task 3 depends on data from tasks 1 and 2. In this example, task 1 finishes earlier than its worst-case execution time. Task 3 cannot start immediately making its earlier start time less than the earlier finish time of task 1. The timing slack that is available to perform conservative frequency scaling is therefore equal to the earlier starting time of task 3 and not the earlier finishing time of task 1.

3.2.2 Application-level Timing

Real-time requirements are commonly specified on larger granularity than the task level. For instance, a frames per second timing requirement for a video decoder application may require multiple graph iterations to complete a single video frame. POSe applications can be mapped across multiple cores. POSe's accounting operates in a distributed manner, i.e. POSe on each core only has the timing information of the part of the application that is mapped locally. If global application timing information is required, it is up to the application's power-management scheme to communicate this information via the regular dataflow communication channels.

POSe accounts application-level timing on the granularity of application graph iterations. Timing requirements can be set as an integer number of graph iterations to be

[1] An earlier enabling of an actor depends on the latest finish of preceding actors. In [80], slack lost due to multi-core execution was not correctly taken into account. Tasks accumulated slack to scale their next iteration regardless of whether this translated to earlier subsequent task starting times.

performed within an amount of time. This information is stored in the application's ACB, with the time stored in terms of the number of cycles of the unscaled clock frequency.

POSe can only measure how many times the locally mapped part of the application graph has finished. Applications can be modelled as HSDF, SDF or CSDF graphs. For SDF and CSDF modelled applications, tasks might have to fire multiple times to complete a single graph iteration. The combined application and CompSOC platform dataflow models, described in Section 2.5, require that application tasks are scheduled following a SOS. The schedule order is specified to POSe as a table of local task IDs. POSe's task scheduler simply schedules the task with the ID corresponding to the next ID in the table, and wraps around to the first table entry after the last table entry has been scheduled. By specifying that the table should contain the SOS of local task executions for a single application graph iteration, the number of local application graph iteration completions are counted by observing the number of times the scheduler reaches the end of the SOS table.

POSe performs the application graph count in a distributed manner. The number of local application graph iterations that have completed can differ per core. This can happen because data is buffered in the FIFO channels between dataflow tasks that are mapped onto different cores. POSe does not provide any explicit method to determine locally how many times the application graph has completed globally. This information can be gained by the application in multiple ways. One way is by having a single "synchronising" task with which the final task in each static order schedule communicates. The synchronising task can fire whenever it has received a single token from the final task in each core's SOS. The synchronising task's iteration count is the number of complete application graph iterations. This information can then be communicated via the regular dataflow communication channels to tasks on the other cores, if required.

While using a synchronising task is relatively simple to implement, it limits scalability. All cores must be able to communicate with the core on which the synchronising task is mapped. Even though the amount of communicated data is relatively small, the communication infrastructure must exist to enable this. In Section 5.2 we describe a distributed power-management technique that uses the local application graph count and the static information of the graph topology and mapping to calculate the minimum number of complete application graph iterations that must have taken place. Each core performs the calculation independently, requiring no explicit synchronisation or communication of the calculated result.

3.3 CoMik Composable Accounting

The CoMik micro-kernel enables composable energy and power accounting per virtual processor. The execution on each virtual processor is cycle-accurately isolated from other concurrent virtual processors that operate on the same physical processor, as explained in Section 2.3. It is therefore possible to maintain independent accounts, and set inde-

pendent budgets per virtual processor. In this section, we explain how CoMik maintains independent time and energy accounts for each virtual processor, before describing how independent composable energy budgets are allocated in Section 3.4.

CoMik enables each virtual processor to composably perform DVFS management. The frequency of the virtual processor can be changed at any time by the software running on the virtual processor. Following the processor power model described in Section 3.1, each frequency level has an associated power consumption, i.e. the rate at which energy is consumed. The energy accounting information is updated whenever a frequency change takes place. Knowing the current frequency level and its associated power level, the time between the previous frequency change and the current frequency change is used to calculate the amount of energy consumed since the last frequency change. A running total of energy consumed by a virtual processor is simply the sum of all of the energy calculations for every frequency change.

CoMik splits the processor time into CoMik slots and virtual processor slots and maintains separate CoMik and virtual processor accounts. A timed interrupt signal is raised whenever the virtual processor's slot has come to an end and the CoMik slot begins. The TIFU simultaneously raises the interrupt and changes the processor frequency to the CoMik slot frequency. As is explained in Section 2.3, it can be a number of cycles until the virtual processor's execution can be interrupted, due to an uninterruptible multi-cycle instruction or critical region. Once interrupted the virtual processor's energy accounting is updated, and its context is stored. For simplicity, the virtual partition's energy accounting only tracks the energy consumed during the virtual processor slot, i.e. before the interrupt occurred. The energy accounting is stored as part of the virtual processor's context enabling accounting and budgeting per virtual processor.

3.3.1 Implementation

CoMik divides the processor's time between executing user code during the virtual processor slots, and executing CoMik code to context swap between virtual processors in the CoMik slot. Each virtual processor keeps an account of its energy consumption in its associated PCB (described in Section 2.3.1). The energy consumed during the CoMik slot is accounted for in CoMik's CCB. The virtual processors and CoMik can receive independent energy or power budgets. A detailed explanation of composable energy and power budgeting is given in Section 3.4.

In both the PCB and the CCB, the energy accounting and budgeting is maintained as a set of three 64 bit signed integer variables:

`ENERGY energy_budget` The energy allocated to the account.
`ENERGY energy_budget_remaining` The currently remaining budgeted energy.
`TIME last_account_update` The last time that the account was updated.

with the type definitions of `ENERGY` and `TIME` used to differentiate the type of data stored in the variables. Energy is stored in units of Joules, with the unit prefix (milli-,

micro-, etc.) determined by the power model used. The `energy_budget` variable stores the total of the energy budget that is allocated to the virtual processor or CoMik account. The `energy_budget_remaining` value is initialised with the `energy_budget` value. As the virtual processor or CoMik consumes its budget, the associated `energy_budget_remaining` is decremented by the consumed amount. Whenever the budgeting information is updated, a time-stamp is stored as the `last_account_update` value. The `last_account_update` time is used as the starting time for future energy consumption calculations. The user code executing on a virtual processor only has access to its own energy accounting information and this access is read-only.

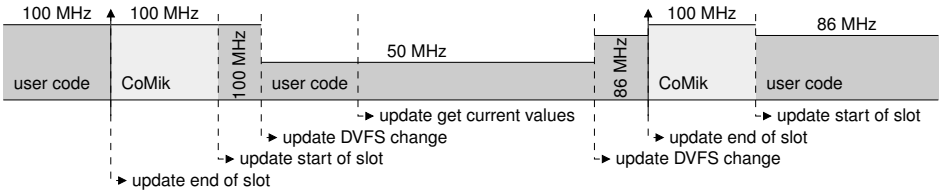


Figure 3.3: CoMik energy accounting update points.

The energy budgets of the virtual processors are updated by CoMik in the CoMik slot, before and after each virtual processor slot, as illustrated in Figure 3.3. After CoMik schedules the virtual processor it sets the `last_account_update`, of the virtual processor's account, equal to the start time of the next virtual processor slot. This is done so that the virtual processor's energy consumption is measured from the start of its next slot and not the entire time since it was last updated at the end of its previous slot. At this point (the end of the CoMik slot), CoMik also updates its own energy account in the CCB with the energy consumed during the CoMik slot, and updates the `last_account_update` value in the CCB.

After the virtual processor slot has ended, the energy accounts are updated to account for the energy consumed between the `last_account_update` and the end of the slot. The `last_account_update` value in the virtual processor's PCB and in CoMik's CCB is set to the time of the end of the virtual processor slot.

During the virtual processor slot, the accounting information is updated whenever a DVFS change is made, or whenever the user code requests the current account values, as illustrated in Figure 3.3. If a DVFS change is made, the power associated with the old frequency level is multiplied by the amount of time that the virtual processor has spent at that level since the `last_account_update`. Similarly, if the current account values are requested, the power associated with the current frequency level is multiplied by the amount of time since the `last_account_update` and the current time. The amount of energy calculated in either case is deducted from the `energy_budget_remaining` and the `last_account_update` value is updated to match.

The user code executing on a virtual processor can access the accounting values of its virtual processor via API get functions, provided as part of CoMik's API. CoMik does

not provide the equivalent set functions, making the energy accounting values read-only from the perspective of the user code.

3.4 Composable Energy Budget Distribution

The CompSOC platform's composable virtual processors ensure that the timing behaviour of code executing on concurrent virtual processors does not interfere. Each partition is assigned an independent timing budget (that represents the partitions real-time requirement) and also an energy and/or power budget. Partitions can independently perform power-management, using their budgets in whatever manner. This means that partitions may consume their budgets at different rates, with some partitions consuming their budgets earlier than others.

The timing budget represents an abstract resource of time to complete a quantity of work. The rate at which a partition consumes this budget effects its timeliness, but has no effect on the timing behaviour of other partitions. The energy and power budgets represent a physical finite platform resource. Once a partition has consumed these budgets, any further consumption uses a physical resource that was not allocated to it, interfering with the ability of the partition, to which the resource was allocated, from consuming it. For example, if two similarly dimensioned partitions on the same processor are given the same energy budget, if one partition always runs at the maximum DVFS level and the other at the minimum DVFS level then the partition running at the maximum DVFS level will run out of energy much sooner than the other partition. If the application in the partition with no energy was to continue running, it would consume energy that was not allocated to it, e.g. from the energy allocated to the partition running at minimum DVFS. The application running at minimum DVFS would therefore not be able to use the entire budget it was allocated. This interference between applications is non-composable.

Even if the partition with no energy was prevented from being scheduled again, energy would still be required to keep its idle slots in a low-power state. Completely voltage gating the processor would not offer a solution, as storing and restoring the processor's context still requires energy. It is therefore necessary for composability that energy is held in reserve to cover this eventuality. How big the reserve needs to be depends on many factors, such as the minimum and maximum power consumption rates. In this section, we describe a method that enables composable energy and/or power budgets per partition, providing a guarantee that a partition will be able to consume the entire budget it was allocated regardless of its power-management strategy or the behaviour of concurrent partitions.

3.4.1 Dividing the Core Energy Budget

We begin by describing how an energy budget that is allocated to a single core is composably divided among multiple virtual partitions. Each partition is assigned an individual energy budget that can be used independently without interfering with the ability of

other partitions to consume their budget. The ability to design and verify in isolation without the need for a final monolithic verification step is central to the composable design paradigm. A specifically tailored energy solution that depends on a behavioural analysis of concurrent partitions therefore does not fit this design concept. A developer may not even know how their own partition's power-management will perform at run-time, if for instance the partition's execution is data dependent and the power-management responds to these variations.

Energy not only needs to be budgeted for the partitions but also for CoMik to execute between partition slots. Enough energy E_{comik} has to be budgeted for CoMik to run for as long as any partition still has energy remaining. Our method allocates each partition an energy budget based on the number of slots that it is allocated in the CoMik TDM table. Given a single energy budget for the core $E_{\text{core}} \in \mathbb{R}^+$, this budget is divided into four types of energy budget allocation; partition slot energy $E_{\text{partition}} \in \mathbb{R}^+$, CoMik slot energy $E_{\text{comik}} \in \mathbb{R}^+$, reserve energy $E_{\text{reserve}} \in \mathbb{R}^+$ and final slot energy $E_{\text{final}} \in \mathbb{R}^+$. The E_{final} budget is another type of reserve budget that ensures that if the scheduled application runs out of energy somewhere during its partition slot there is enough energy to complete the partition slot at whatever DVFS level it is running at.

The partition slot energy budget $E_{\text{partition}}$ is the amount of energy allocated to a single partition slot. The total partition energy budget for the entire core is calculated as $N \cdot S \cdot E_{\text{partition}}$, where $N \in \mathbb{N}$ is the maximum number of TDM table iterations that can be completed within the E_{core} budget and $S \in \mathbb{N}$ is the total number of slots in the TDM table. Similarly, the CoMik slot energy budget is the amount of energy allocated to permit CoMik to execute for the duration of a single CoMik slot in the TDM table. The total CoMik energy budget for the entire core is therefore calculated as $N \cdot S \cdot E_{\text{comik}}$.

The reserve energy budget E_{reserve} is the amount of energy per TDM slot that must be held in reserve to be consumed by the partition slots of partitions that have finished their energy budget, so that other partitions may finish theirs. In the worst-case, this budget will be used by all but one of the partitions, as the last partition to finish its budget cannot affect the composability of other partitions. The total reserve energy budget for the entire core is therefore calculated as $N \cdot (S - A) \cdot E_{\text{reserve}}$, where $A \in \mathbb{N}$ is the smallest TDM slot allocation assigned to a single partition on the core. If this is not known at design time, the worst-case, $A = 1$, can be used for a conservative result.

The final slot reserve energy budget E_{final} is dimensioned to contain enough energy to allow the completion of the partition slot in the eventuality that the partition's energy budget runs out at some point during the slot. This is necessary to ensure that the partition is able to complete its entire energy budget allocation. The partition's energy budget is not monitored continuously during execution. The only guaranteed time that a partition's budget is checked is at the end of each partition slot. A partition with a non-empty energy budget is scheduled regardless of whether it has enough energy to complete the partition slot and may execute at any power-level for the duration of the slot. The final slot energy budget per core is therefore calculated as $V \cdot E_{\text{final}}$, where $V \in \mathbb{N}$ is the number of virtual processors that share the TDM slots. If this is not known at design time, the worst-case,

$V = S$, can be used for a conservative result.

The sum of all four budget types must be less than or equal to the energy budget of the core E_{core}, as presented in the following equation:

$$E_{\text{core}} \geq N \cdot (S \cdot (E_{\text{comik}} + E_{\text{partition}}) + (S - A) \cdot E_{\text{reserve}}) + V \cdot E_{\text{final}} \tag{3.5}$$

Each partition n receives an independent energy budget E_n proportional to the number of slots A_n that it is allocated in CoMik's TDM table, which is calculated as follows:

$$E_n = N \cdot A_n \cdot E_{\text{partition}} \tag{3.6}$$

Even though the budget is calculated per partition slot, the partition receives its energy budget E_n as a single quantity, thereby virtualising the energy source, e.g. the partition can be viewed as having a virtual battery.

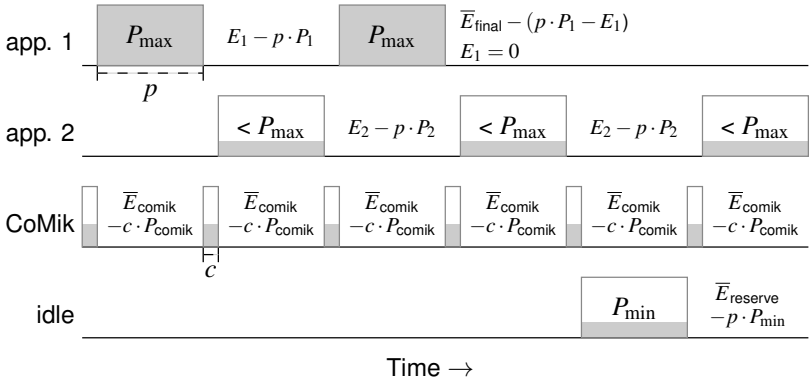


Figure 3.4: Composable energy consumption.

Figure 3.4 illustrates an example of how applications composably consume energy from their energy budgets. Applications illustrated in Figure 3.4 run at a constant power level, but in implementation they can change DVFS settings at any time. In this example, application 1 runs at maximum power $P_{\max}$ and application 2 runs at less than maximum power $< P_{\max}$ causing application 1 to consume its energy budget E_1 before application 2 consumes its energy budget E_2. CoMik is invoked at regular intervals (to perform application context switch, etc.), consuming energy from its energy budget $\overline{E}_{\text{comik}}$ (total tile CoMik energy budget). Application 1 runs out of energy in the middle of its final slot, with the shortfall in its energy budget E_1 being decremented from the $\overline{E}_{\text{final}}$ reserve budget (total tile final slot energy budget). After this, application 1 cannot be scheduled and its slots are idled at the minimum available power level $P_{\min}$. The energy consumed by idled slots is decremented from the $\overline{E}_{\text{reserve}}$ budget (total tile reserve energy budget).

The execution of application 2, and its ability to consume its entire energy budget E_2, is completely unaffected by application 1 finishing its budget earlier.

The virtualised energy source is a dedicated resource of the partition that is guaranteed to be composably isolated from interference due to the behaviour of other partitions. The reserve energy budget E_{reserve} is dimensioned to ensure that slots of partitions that have consumed their budgets can be idled without affecting the budgets of other partitions. In the worst-case, one partition consumes its energy budget at the lowest power level P_{min} and the other consumes its energy budget at the highest power level P_{max}. The partition that consumes its energy budget at P_{max} will finish its budget first and its slots are idled at P_{min} using energy from the E_{reserve} budget. The relationship between the quantities of these budgets is therefore as follows:

$$\frac{E_{\text{partition}}}{P_{\text{min}}} = \frac{E_{\text{partition}}}{P_{\text{max}}} + \frac{E_{\text{reserve}}}{P_{\text{min}}} \tag{3.7}$$

which can be rewritten to find E_{reserve} as follows:

$$E_{\text{reserve}} = E_{\text{partition}} \cdot \left(\frac{P_{\text{max}} - P_{\text{min}}}{P_{\text{max}}} \right) \tag{3.8}$$

Calculations made using these equations are composable as they do not depend on application specific information. From Equation 3.8 it is clear that the per slot reserve energy budget is less than or equal to the per slot partition energy budget $E_{\text{partition}}$, with the proportion dependent on the values of P_{max} and P_{min}. The lower the ratio of P_{min} to P_{max} the more energy needs to be held in reserve per slot, as presented in Figure 3.5.

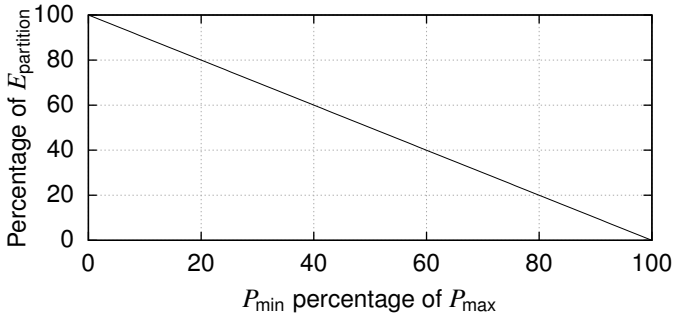


Figure 3.5: E_{reserve} in relation to $E_{\text{partition}}$, P_{min} and P_{max}.

The reserve energy budget is calculated to cover the amount of energy required to idle the partition's slots from the moment its energy budget runs out. The final slot energy budget is therefore only required to hold enough energy to make up the shortfall in the event that the partition runs at P_{max} for the rest of its partition slot after running out of energy. The final slot energy budget E_{final} is therefore calculated as follows:

$$E_{\text{final}} = p \cdot (P_{\max} - P_{\min}) \tag{3.9}$$

where $p \in \mathbb{R}$ is the duration of the partition slot.

The energy that is allocated to the core is divided among the budgets to maximise the number of table iterations N that can be composably completed. By rewriting Equation 3.5, a relationship for N is described as follows:

$$N \leq \frac{E_{\text{core}} - V \cdot E_{\text{final}}}{S \cdot (E_{\text{comik}} + E_{\text{partition}}) + (S - A) \cdot E_{\text{reserve}}} \tag{3.10}$$

As N is an integer number of table iterations, the right side of Equation 3.10 is rounded down:

$$N = \left\lfloor \frac{E_{\text{core}} - V \cdot E_{\text{final}}}{S \cdot (E_{\text{comik}} + E_{\text{partition}}) + (S - A) \cdot E_{\text{reserve}}} \right\rfloor \tag{3.11}$$

Substituting Equation 3.8 and Equation 3.9 into Equation 3.11 produces the following equation:

$$N = \left\lfloor \frac{E_{\text{core}} - V \cdot p \cdot (P_{\max} - P_{\min})}{S \cdot (E_{\text{comik}} + E_{\text{partition}}) + (S - A) \cdot E_{\text{partition}} \cdot \left(\frac{P_{\max} - P_{\min}}{P_{\max}} \right)} \right\rfloor \tag{3.12}$$

The maximum number of table iterations N is achieved for a given core energy budget whenever all the partitions consume their partition energy budget $E_{\text{partition}}$ at $P_{\min}$, making the energy consumed by the partition slot $E_{\text{partition}} = p \cdot P_{\min}$. The CoMik slot is executed at a constant frequency and hence power-level (some periods of clock gating may occur, but it is conservative to assume a constant power-level greater than or equal to the clock gating power-level). The energy consumed by the CoMik slot is therefore $E_{\text{comik}} = c \cdot P_{\text{comik}}$, where $c \in \mathbb{R}$ is the duration of the CoMik slot.

$$N = \left\lfloor \frac{E_{\text{core}} - V \cdot p \cdot (P_{\max} - P_{\min})}{S \cdot (c \cdot P_{\text{comik}} + p \cdot P_{\min}) + (S - A) \cdot p \cdot P_{\min} \cdot \left(\frac{P_{\max} - P_{\min}}{P_{\max}} \right)} \right\rfloor \tag{3.13}$$

The duration of the CoMik slot c bounds the worst-case work that must be performed in the CoMik slot. Lowering the frequency lowers the power-level but also increases the duration of the CoMik slot c. As shown in Figure 3.1b, at lower frequency levels this will cause the total amount of energy to increase for the amount of work performed. The lowest energy consumption by the CoMik slot is therefore achieved at the frequency of the

lowest energy in Figure 3.1b. For a CoMik slot with a worst-case work of W_{comik} cycles, using the power model described in Equation 3.4 with frequency f_{comik} in MHz, E_{comik} is calculated in joules (J) as follows:

$$E_{\text{comik}} = c \cdot P_{\text{comik}} = \frac{W_{\text{comik}}}{f_{\text{comik}} \times 10^6} \cdot (3.353 \times 10^{-8} f_{\text{comik}}^3 + 2.065 \times 10^{-3}) \quad (3.14)$$

By differentiating Equation 3.14 and solving the resultant equation, the value of f that corresponds to the minimum E_{comik} is found to be 31.34 MHz. Using Equation 3.13 and Equation 3.14 with the following attributes:

$$\begin{aligned} f_{\text{comik}} &= 31.34 \text{ MHz} \\ W_{\text{comik}} &= 4096 \text{ cycles} \\ p &= 546.133\ \mu\text{s} \\ S &= 10 \text{ slots} \\ V &= S \text{ slots} \\ A &= 1 \text{ slot} \\ E_{\text{core}} &= 100 \text{ J} \\ P_{\text{min}} &= 2.065 \text{ mW } \textit{from Equation 3.4 with } f = 0 \textit{ MHz} \\ P_{\text{max}} &= 60 \text{ mW } \textit{from Equation 3.4 with } f = 120 \textit{ MHz} \end{aligned} \quad (3.15)$$

we plot N for a sweep of $f_{\text{comik}} = 0 \rightarrow 120$ MHz (f_{comik} in Equation 3.15 is used for subsequent experimentation) producing the graph presented in Figure 3.6 that shows the maximum number of TDM table iterations N that can be achieved for the range of CoMik slot frequencies f_{comik}. This graph shows that the maximum number of iterations are achieved for the given core energy budget E_{core}.

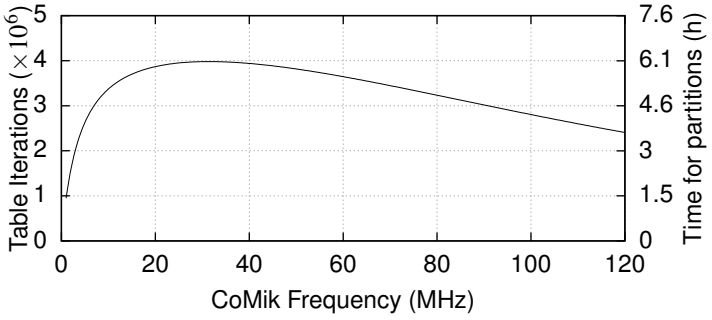


Figure 3.6: Effect of CoMik slot frequency on number of TDM iterations.

While a CoMik slot frequency of 31.34 MHz enables the maximum number of composable TDM table iterations, it does not produce the longest maximum composable

run-time $\overline{R}$, as presented in Figure 3.7 using the same attributes as for Figure 3.6. The maximum composable run-time $\overline{R}$ is the maximum time that the energy budgets can provide for composable execution (i.e. a platform with a maximum composable run-time of one hour, cannot guarantee composable execution after that hour), and is calculated as follows:

$$\overline{R} = N \cdot S \cdot (c + p) \tag{3.16}$$

which is the duration of a single TDM table iteration $S \cdot (c+p)$ multiplied by the maximum number of table iterations N.

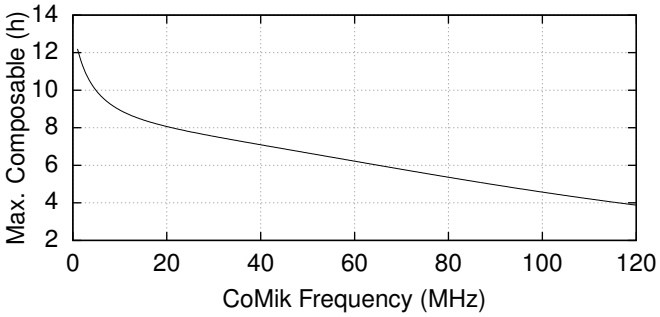


Figure 3.7: Effect of CoMik slot frequency on the maximum composable run-time.

While Figure 3.7 shows that the system will be able to run composably for longer, Figure 3.6 shows that when the CoMik frequency drops below 31.34 MHz, fewer table iterations are achievable for the same core energy budget. This means that even though the maximum composable run-time gets longer as $f_{\text{comik}} \to 0$, the amount of time allocated to partitions to perform work actually decreases, when $f_{\text{comik}} < 31.34$ MHz. Depending on the target application of the system, this could still be a worthwhile trade-off.

The number of table iterations N that is calculated using Equation 3.11 is used to calculate the total energy allocated per core to the partitions $\overline{E}_{\text{partition}}$, the total energy held in reserve $\overline{E}_{\text{reserve}}$, the total energy allocated to CoMik $\overline{E}_{\text{comik}}$, and the total energy held in reserve for a partition's final slot $\overline{E}_{\text{final}}$. These energy budgets are calculated as follows:

$$\overline{E}_{\text{partition}} = N \cdot S \cdot p \cdot P_{\text{min}} \tag{3.17}$$

$$\overline{E}_{\text{reserve}} = N \cdot (S - A) \cdot p \cdot P_{\text{min}} \left(1 - \frac{P_{\text{min}}}{P_{\text{max}}}\right) \tag{3.18}$$

$$\overline{E}_{\text{comik}} = N \cdot S \cdot c \cdot P_{\text{comik}} \tag{3.19}$$

$$\overline{E}_{\text{final}} = V \cdot p \cdot (P_{\text{max}} - P_{\text{min}}) \tag{3.20}$$

$\overline{E}_{\text{partition}}$ Equation 3.17 is the energy allocated to a single partition slot $E_{\text{partition}} = p \cdot P_{\text{min}}$ multiplied by the number of partition slots in the TDM table S and the maximum number

of iterations of the TDM table N. $\overline{E}_{\text{reserve}}$ Equation 3.18 is the energy held in reserve for a single slot E_{reserve}, calculated using Equation 3.8, multiplied by the maximum number of slots in the TDM table that the reserve needs to be held for $S-A$ and the maximum number of iterations of the TDM table N.

Plotting the distribution of E_{core} to the $\overline{E}_{\text{reserve}}$, $\overline{E}_{\text{comik}}$ and $\overline{E}_{\text{partition}}$ budgets with the attributes from Equation 3.15 ($\overline{E}_{\text{final}}$ is a tiny proportion of E_{core} and therefore is not visible in this graph), but with $p = 0 \rightarrow 10$ ms, produces Figure 3.8. This graph shows that increasing the partition slot length increases the ratio of the core budget that the partitions can be composably allocated. The effect is greater for shorter partition slot durations. As $p \rightarrow \infty$, the longer partition slots increase the TDM table duration, which in turn reduces the maximum number of iterations N that can be completed for the given core energy budget. Increasing p also increases the amount of energy kept in reserve to bound the energy consumption of the partition's final slot E_{final}. As $p \rightarrow \infty$, E_{final} increases consuming an ever larger proportion of the core energy budget, eventually reducing the ratio of core energy given to the partitions at longer partition slot durations. While there is an optimal p to provide a maximum ratio of core energy to the partition, it is not necessarily a practical value as longer partition slots increases the time between partition swaps and increases the duration of a single TDM table iteration. If the partition slots are too long relative to the real-time requirements of the applications within the partitions, then the sampling/output of the applications will appear bursty or require buffering, which would require physical memory resources with additional area and power consumption overhead.

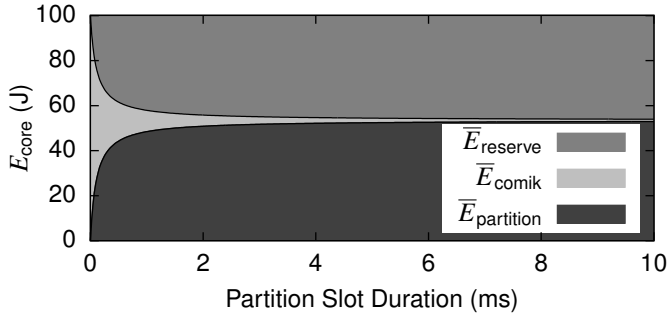


Figure 3.8: Effect of the partition slot duration on E_{core} budget distribution.

Relatively short partition slot lengths p and a shorter TDM table (i.e. fewer number of slots S) are more generally suitable to meeting latency requirements of applications. Whereas shorter partition slot lengths p decrease the ratio of core energy that is allocated to the partitions, fewer slots S in the TDM table increases the proportion of core energy allocated to the partitions. This is presented in Figure 3.9 for the attributes from Equation 3.15 except with $S = 2 \rightarrow 100$. The maximum number of composable partitions is equal to the amount of slots in the TDM table, with 2 being the minimum number of slots

that permits sharing. For a constant partition slot duration p, as the number of slots in the TDM table increases $S \to \infty$ the amount of energy that is allocated to the partitions decreases. In Figure 3.9, this is due to the smallest slot allocation to a virtual processor A remaining at the conservative level of one slot, while the total number of slots in the TDM table increases. This causes more energy to be held in the E_{reserve} budget to cover the possibility that the single slot virtual processor depletes its budget at minimum power, while the rest of the slots deplete their budget at maximum power.

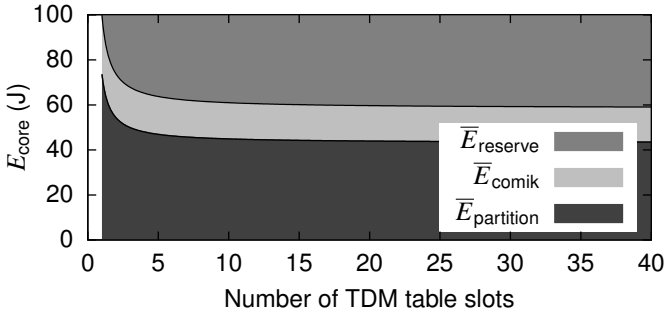


Figure 3.9: Effect of the number of TDM table slots on E_{core} budget distribution.

In Figure 3.10, we present the E_{core} budget distribution for a range of values of the minimum virtual processor slot allocation A. This graph is produced using the values from Equation 3.15, but with $S = 100$ and $A = 1 \to 50$. This range is chosen for A as the smallest virtual processor allocation cannot be greater than 50% of S when the processor is shared, i.e. if one virtual processor has $> 50\%$ of S then another virtual processor is guaranteed to have the smallest allocation, and that it is $< 50\%$ of S. Increasing A enables less energy to be held in the E_{reserve} budget, permitting more energy to be used by the partitions, increasing the $E_{\text{partition}}$ budget, while at the same time increasing the maximum number of composable table iterations N, thereby increasing the amount of energy that needs to be held in the E_{comik} budget.

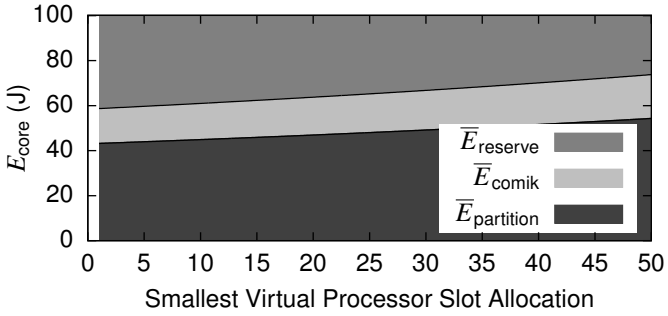


Figure 3.10: Effect of the smallest slot allocation on E_{core} budget distribution.

3.4.2 Guaranteed Composable Run-time

The end result of the energy budget dimensioning is a virtualised composable energy source, which can be thought of as a virtual battery. To achieve composability, energy is held in reserve, reducing the amount of energy that the partitions can composably use. It is possible that none, some or all of the reserve energy budget remains after the partitions have depleted their energy budget. This cannot be composably redistributed to the partitions as the quantity of remaining reserve energy depends on the behaviour of all of the partitions. After the point where the virtual batteries have depleted, the remaining reserve energy can continue to power the system, but without a composability guarantee.

Since each partition may perform power management using DVFS, the amount of time that the system can run for a given energy budget depends on the frequency levels, and hence power levels, that it uses. For an energy budget of 20000 J[1], using the power model from Equation 3.4, at continuous maximum power consumption ($f = 120$ MHz) the budget would be depleted in 3.86 days. At continuous minimum power consumption ($f = 0$ MHz) the budget would be depleted in 112.1 days.

The ratio of the minimum power to the maximum power affects the amount of energy that must be held in reserve, as presented in Figure 3.5. The higher the ratio, the lower the energy reserve that must be held. Increasing the ratio, by lowering the maximum power-level (e.g. by preventing the partitions from scaling above a particular frequency) reduces the amount of energy that must be held in reserve, enabling more energy to be distributed to the partitions' energy budgets, providing a longer composable run-time. Figure 3.11 demonstrates this for the values from Equation 3.15, but with $E_{\text{core}} = 20000$ and scaling P_{max}. In this figure, the max. run-time line is the composable run-time whenever all the partitions consume their budget at minimum power, and the min. run-time line is the composable run-time whenever all the partitions consume their budget at maximum power.

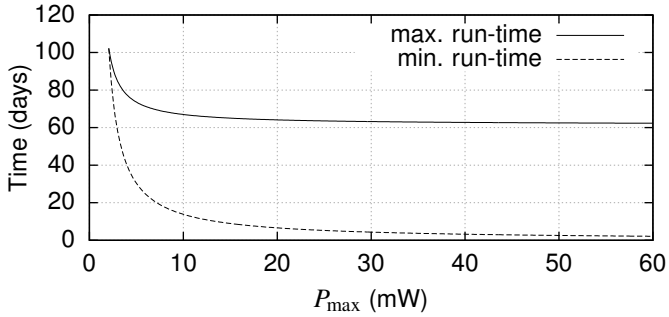


Figure 3.11: Composable run-time keeping P_{min} constant.

[1] approx. 5.56 Wh, which is equivalent to the rated capacity of a current generation smart-phone battery. Battery self-leakage is not taken into consideration in this instance, but there is no fundamental reason preventing it, given an appropriate model.

When P_{max} is capped at 50 mW the minimum composable run-time is approximately 3 days, and at 10 mW the minimum composable run-time is approximately 12 days. While putting a cap on the highest power level (frequency) that a partition can use has benefits, it also reduces the freedom the frequency range that the partition can use to perform DVFS. More importantly, capping the higher end of the frequency range can only make it more difficult for a partition to meet its real-time requirement. Figure 3.12 presents the composable run-time for the same configuration, but this time increasing the ratio of the minimum power to the maximum power by raising the minimum power level while keeping the maximum power level constant at the value given in Equation 3.15.

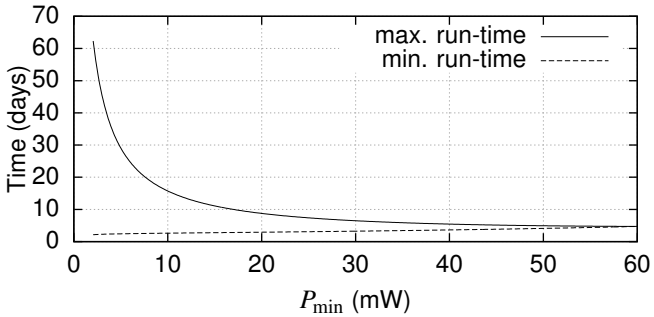


Figure 3.12: Composable run-time keeping P_{max} constant.

As the minimum power P_{min} increases, the amount of energy that needs to be held in reserve decreases allowing more energy to be distributed to the partitions. The minimum composable run-time increases, reflecting the larger partition energy budget. The maximum composable run-time decreases even though the partition energy budget increases because the budget increase does not compensate the increase in minimum power level.

3.4.3 Multi-core Systems

On a single core, care is taken that enough energy is held in reserve to bound the eventuality that some partitions consume their budget quickly (e.g. at maximum power) and other partitions consume it slowly (e.g. at minimum power). This reserve is necessary as it is not necessarily possible, or practical, to completely power down the processor for the fine grained durations of partition slots. On multi-core systems each core gets an individual energy budget E_{core} that is divided into the partition, CoMik and reserve budgets. Depending on the energy allocated to the core, the type of core (in a heterogeneous system) and the configuration of CoMik (e.g. number of TDM slots), each core could have a different composable run time. If it is not possible to fully power down the processor after it has completed, the core must also hold enough energy in reserve to idle the processor until the rest of the processors have composably completed their budgets.

In a homogeneous system, in which the power consumption of every processor is conservatively bounded using the same power model and CoMik is configured with the

same values on each processor (same number of TDM slots, etc.), each core can be assigned the same quantity of energy budget. The maximum and minimum composable run-time of each core will therefore be the same. On systems that are unable to completely power down individual processors enough energy must be held in reserve to permit the partitions on each core to consume their entire energy budget. This is achieved using the energy budgeting equations Equation 3.17–Equation 3.20, by setting the smallest partition slot allocation $A = 0$, thereby ensuring that an energy reserve is maintained for each slot in the TDM table for the eventuality that the partitions on the core finish their energy budgets sooner than the maximum composable run-time.

We leave the dimensioning of energy budgets in heterogeneous multiprocessor systems as future work.

3.4.4 Composable Power Budgets

In the previous sections we have described in detail how a single energy budget (such as a battery) can be composably shared among multiple partitions. Energy in embedded systems can also come from power sources that deliver energy at a rate, such as from energy scavenging components. The maximum power consumption P_{max} of the platform caps the rate at which the partitions can consume an energy budget. Allocating power budgets to partitions that are higher than P_{max} gives the partition freedom to always execute at maximum frequency. If a partition's power budget is lowered below P_{max} this is not possible. A simple solution to ensure that a partition stays within its power budget is to cap the maximum frequency (and hence power) that the partition can use. If the partition is running a real-time application, the maximum frequency cap might cause it to violate its real-time requirements (or at least its worst-case guarantee). There is therefore a fundamental upper-limit on the amount of power that a partition can consume and also a practical lower-limit (which is application dependent) below which the temporal requirements of a real-time application are no longer guaranteed to be met.

A power budget is expressed in units of Watts. To fit with CoMik's energy accounting scheme, the power budget is translated into a replenishing energy budget. As with the energy distribution scheme, we assume that each core is assigned an individual power budget, P_{core}. CoMik replenishes the core energy budget E_{core} every N TDM table iterations, with the amount of energy that corresponds with the available energy, which is calculated as follows:

$$E_{\text{core}} = P_{\text{core}} \cdot N \cdot S \cdot (c + p) \tag{3.21}$$

where S is the number of slots in CoMik's TDM table, c is the CoMik slot duration and p is the partition slot duration. The E_{core} budget is divided between CoMik and the partitions, with the per TDM slot CoMik energy budget E_{comik} calculated using Equation 3.14 and the per TDM slot partition energy budget $E_{\text{partition}}$ as follows:

$$E_{\text{partition}} = \frac{E_{\text{core}}}{N \cdot S} - E_{\text{comik}} \tag{3.22}$$

The E_{core} budget is divided by the number of table iterations N and the number of slots in the table S to calculate the amount of energy that is available to perform a single CoMik and partition slot. The $E_{\text{partition}}$ budget receives the remainder of the budget. This amount might not be enough to execute the entire slot at maximum frequency. To account for this, the maximum frequency that the partition can use is capped at a lower frequency f_{cap} that the energy budget can sustain until the next replenishment, which can be calculated using Equation 3.4 as follows:

$$E_{\text{partition}} = (3.353 \times 10^{-5} f_{\text{cap}}^3 + 2.065) \cdot p \tag{3.23}$$

$$f_{\text{cap}} = \sqrt[3]{\left(\frac{E_{\text{partition}}}{p} - 2.065\right) \cdot \frac{1}{3.353 \times 10^{-5}}} \tag{3.24}$$

where in $E_{\text{partition}}$ is in mJ and f_{cap} is in MHz.

A higher maximum partition frequency is more desirable, to meet the real-time requirements of a larger set of applications. Increasing the amount of energy that is allocated to the partitions slots, by decreasing the frequency of the CoMik slot, increases the maximum partition slot frequency, but has an adverse affect on the partition's ability to meet real-time requirements. Decreasing the frequency of the CoMik slot increases the CoMik slot length c and therefore also the TDM table length $N * S * (c + p)$. This increases the amount of energy in the E_{core} budget for a given P_{core}, as calculated using Equation 3.21, which in turn enables a higher sustainable partition frequency. However, this higher maximum partition frequency does not translate into a higher maximum partition throughput, as the increase in maximum partition frequency does not compensate the loss in throughput caused by the longer CoMik slot that is necessary due to the lower CoMik slot frequency.

3.5 Related Work

The related work on DVFS power modelling is covered in Section 3.1.

The concept of resource containers was described in [13] for servers. They define a resource container as containing all resources that the server uses to perform a particular independent activity. In [105] the resource container concept was applied to energy and time in an energy-aware operating system. The maintained accounts are used to make scheduling decisions. The resources assigned to a container can be limited, with enforcement provided by the operating system. While use of resources can be limited per container they are not guaranteed. Resources assigned to a container can be reduced if they are consumed elsewhere, e.g. by another resource container.

The resource container concept is applied to an embedded system in [68, 69] for use in low-power wireless sensor networks. Using a multiprocessor hardware architecture, they partition the platform into multiple physical power domains. They use a device that they call the Energy Management and Accounting Preprocessor (EMAP), to provide fine grained monitoring and independent accounting of the energy dissipated in the power domains. The EMAP periodically updates the host processor with the energy accounts for the operating system to make scheduling and power management decisions.

Similar to the resource container concept, CoMik maintains composable independent energy and power accounts for virtual processors. This was originally published for an earlier version of the CompSOC platform in [80] (The related work in [80] mainly covers embedded virtualisation, which we cover here in Section 2.6). Our composable energy and power budgeting technique described in [103] and Section 3.4, provides our virtual processors with a guaranteed allocation. From what we can ascertain, no related work exists on energy/power allocation for composable application execution.

3.6 Summary

In this chapter, we have presented the CompSOC platforms composable time, energy and power accounting. We gave a general overview of how the energy and power consumption of electronic systems are commonly modelled, and presented the energy and power models that are used to model the processor power consumption in the CompSOC platform. Following this, we described how POSe provides task or application timing accounting information and CoMik provides energy accounting information to the application to make power-management decisions.

Having described how CompSOC's run-time accounting is performed, we explain how energy and power budgets are composably distributed between virtual processors. Even though each virtual processor has its energy composably accounted for, the manner in which the system's energy is allocated to virtual processors is important to ensure that applications executing on the virtual processor can composably use the energy or power budget that it is allocated. The energy that an application is allocated E_n on a virtual processor n is calculated using Equation 3.6. This calculation is composable as it only depends on the application's configuration, i.e. the number of slots allocated to its virtual processor.

CHAPTER 4

Static Voltage and Frequency Scaling

Voltage and frequency scaling in order to minimise energy and/or power consumption for real-time applications is non-trivial. Reducing the operating frequency increases execution times, potentially violating timing requirements. In this chapter, we apply convex programming to dataflow modelled applications, enabling the derivation of timing guarantees at different frequency levels. Furthermore, the convex model enables the derivation of optimal frequency levels that produce the lowest power consumption.

Convex optimisation is computationally intensive, making it better suited to off-line use. In this chapter, we present an off-line technique that derives static frequency levels that minimise the power consumption of a dataflow modelled application that is annotated with worst-case timings. In Section 2.5, we described how the CompSOC platform and application is modelled as a single dataflow graph. Here the topology and timing of the dataflow graph is translated into a set of constraints forming a convex optimisation problem. In Section 4.1, a detailed explanation of this translation, along with the formulation of objective functions for optimisation, is given. We follow this in Section 4.5 with an experimental evaluation, demonstrating our technique's effectiveness and also its limitations, in Section 4.3 before making some concluding statements.

4.1 Convex Power Optimisation

The DVFS mechanism enables the trade-off between execution speed and power consumption. On real-time systems, execution speed can only be traded within the bounds of the application's temporal requirements. The trade-off between execution speed and power was presented in Section 3.1. In this section, we explain how the timing of the CompSOC platform dataflow model from Section 2.5 is represented as a set of Disciplined Convex Program (DCP) [39] constraints that take DVFS into account [81]. We further describe the minimising objective functions that in combination with these constraints form a convex optimisation problem that can be solved to obtain optimal per-core static frequency levels for power consumption.

4.1.1 SPS Convex Programming Constraint

POSe executes an application's tasks following a STS. An STS schedule varies depending on actor firing duration (task execution time). A WCSTS or SPS is used to conservatively model the STS execution by annotating each actor with their worst-case firing duration. We describe the formalism of these scheduling techniques in Section 2.1.2. An SPS schedule is also conservative in regards to the WCSTS, but is easier to analyse. By modelling the application as an SPS we therefore conservatively bound the timing of the actual application using a STS.

The per edge SPS feasibility constraint is given by Equation 2.8, which we repeat here for readability purposes:

$$s(j,k) + T \cdot d(i,j) \geq s(i,k) + t(i) \qquad \text{(2.8 repeated)}$$

which states that the start time of the consuming actor $s(j,k)$ can only be later than the finishing time of the producing actor $s(i,k)+t(i)$, $d(i,j)$ iterations ago, where $d(i,j)$ is the number of initial tokens on the edge and T is the period of the SPS. We modify this constraint to model the task execution duration when performing VFS as follows:

$$s(j,k) + T \cdot d(i,j) \geq s(i,k) + \frac{t(i)}{f(i)} \qquad (4.1)$$

where $t(i)$ is the worst-case work of actor i measured in cycles and $f(i)$ is the frequency that is used as actor i fires. As the frequency $f(i)$ decreases, the firing duration of actor i increases.

One constraint is added to the convex optimisation for every edge in the application's HSDFG. For a dataflow application executing in POSe on a CoMik virtualised platform, a constraint is added per edge in the combined application and platform HSDFG, as illustrated in Figure 2.27 in Section 2.5.

4.1.2 Power Minimising Objective Functions

With the timing of the combined application and platform graph represented as a set of convex constraints, it is possible to formulate minimising objective functions for power. The convex problem solver minimises the objective function by scaling the function's free variables within the context of the problem's constraints. In this section we present objective functions that are used with the per edge SPS constraints to minimise power consumption while meeting the application's latency and throughput requirements. We continue by describing how this can be achieved for various static frequency configurations.

Reducing Single-Core Power Consumption

The power consumption of the processors in the CompSOC platform are modelled using Equation 3.4, which we repeat here for readability purposes:

$$P(f) = 3.353 \times 10^{-5} f^3 + 2.065 \quad \text{(3.4 repeated)}$$

This power model relates the processor's frequency f in MHz to its power consumption $P(f)$ in nJ. For a single core system, we can formulate Equation 3.4 as the objective function of a convex optimisation problem with f_{core} as the free variable:

$$\text{minimise}(3.353 \times 10^{-5} f_{\text{core}}^3 + 2.065) \tag{4.2}$$

with the following per edge constraints derived from Equation 4.1:

$$s(j,k) + T \cdot d(i,j) \geq s(i,k) + \frac{t(i)}{f_{\text{core}}} \tag{4.3}$$

where each actor in the graph executes at the same frequency f_{platform} in Hz. To ensure that the SPS schedule meets its latency L (maximum time between task firings) and/or throughput constraint R, the period T of the schedule is represented by a free variable and is constrained by the following two constraints:

$$T \leq L \tag{4.4}$$

$$T \leq R^{-1} \tag{4.5}$$

where Equation 4.4 ensures that the period T of the SPS is less than or equal to the latency requirement between actor firings and Equation 4.5 ensures that the period T is less than or equal to the inverse rate requirement of the graph.

The convex problem solver will return the frequency f_{core} that produces the lowest power consumption while permitting the application to meet its timing requirements. All of the platform's clock generators can be set to this frequency, with its associated voltage, at design time.

Reducing Multi-Core Power Consumption

The CompSOC platform is a multi-processor platform that has voltage and frequency islands per processor. These can be configured statically at design time, or dynamically at run-time, enabling the selection of different voltage and frequency levels per processor. We can achieve this by using the following objective function that is formulated from Equation 3.4:

$$\text{minimise} \left(\sum_{\forall c \in \text{cores}} \left(3.353 \times 10^{-5} (f_c \times 10^{-6})^3 + 2.065 \right) + P_{\text{platform}} \right) \tag{4.6}$$

which minimises the sum of the power consumed by all of the processing cores while executing the application, where f_c of core c is measured in Hz, and P_{platform} is a constant representing the power consumed by the rest of the platform other than the cores. As before, we form the constraints for this optimisation from the per-edge constraint given in Equation 2.8.

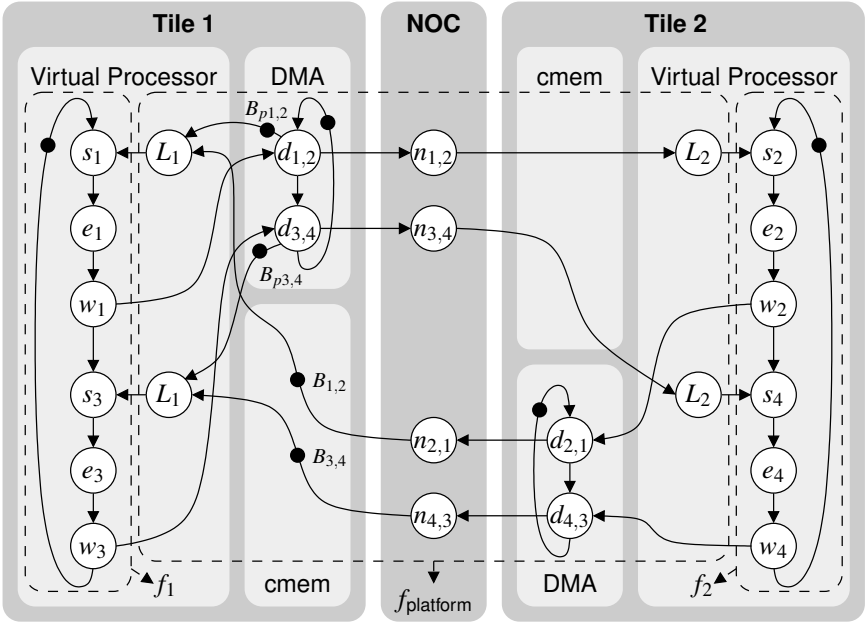


Figure 4.1: Combined application and platform HSDFG with per core DVFS.

Figure 4.1 illustrates an example combined application and dataflow graph. In this example, the rest of the platform has a static frequency of f_{platform} and processing cores 1 and 2 are each assigned independent frequencies f_1 and f_2, respectively. The timer that regulates CoMik's TDM arbitration, measures time using the static f_{platform} as reference.

The latency L_c before the virtual processor is able to service any arrived data therefore depends on f_{platform}. FIFO channels that have a producing actor i with a duration affected by f_{platform}, as illustrated in Figure 4.1, use the constraint given by Equation 4.1, but unlike previously f_{platform} is constant. Other edges with producing actors i that depend on f_c use the following constraint:

$$s(j,k) + T \cdot d(i,j) \geq s(i,k) + \frac{t(i)}{f_c} \tag{4.7}$$

where f_c is a free variable in the optimisation that scales to achieve the objective function given in Equation 4.6. The application's real-time requirements are taken into account as constraints given by Equation 4.4 and Equation 4.5.

Reducing Multi-Core CoMik Virtual Processor Power Consumption

CoMik enables each physical processor in a system to be virtualised into multiple virtual processors, as described in detail in Section 2.3. CoMik achieves this by time sharing the physical processor using TDM scheduling. As power is the rate at which energy is consumed, virtual processors consume energy at a lower rate than the physical processor, which is proportional to the virtual processor's allocation in CoMik's TDM table. Taking this into account produces the following power minimisation function:

$$\text{minimise} \left(\sum_{\forall c \in \text{cores}} \left(\left(3.353 \times 10^{-5} (f_c \times 10^{-6})^3 + 2.065 + \frac{P_{\text{shared}}}{C} \right) \cdot \frac{A_c}{S_c} \right) + P_{\text{dedicated}} \right) \tag{4.8}$$

where C is the number of physical processors in the platform, S_c is the length of CoMik's TDM table on the physical processor that virtual processor c resides and A_c is the number of TDM slots that c is allocated. The power consumed by shared components P_{shared} (e.g. the NoC) is divided evenly between the number of physical processors C. This simple method assumes homogeneous processors, but a more complex division for a heterogeneous platform is possible. For each virtual processor c belonging to the application's VP, its portion of the processor's power and P_{shared} is calculated proportional to its allocation A_c in the TDM table length S_c. The sum of the power consumed by the VP's dedicated components $P_{\text{dedicated}}$ is added to the power consumed by the VP's virtual processors and the VP's share of the power consumed by the rest of the platform.

4.2 Formulation for CVX convex solver

Having described in theory how convex problems are formulated to derive frequencies to achieve reduced energy or power consumption, we proceed to explain how we describe

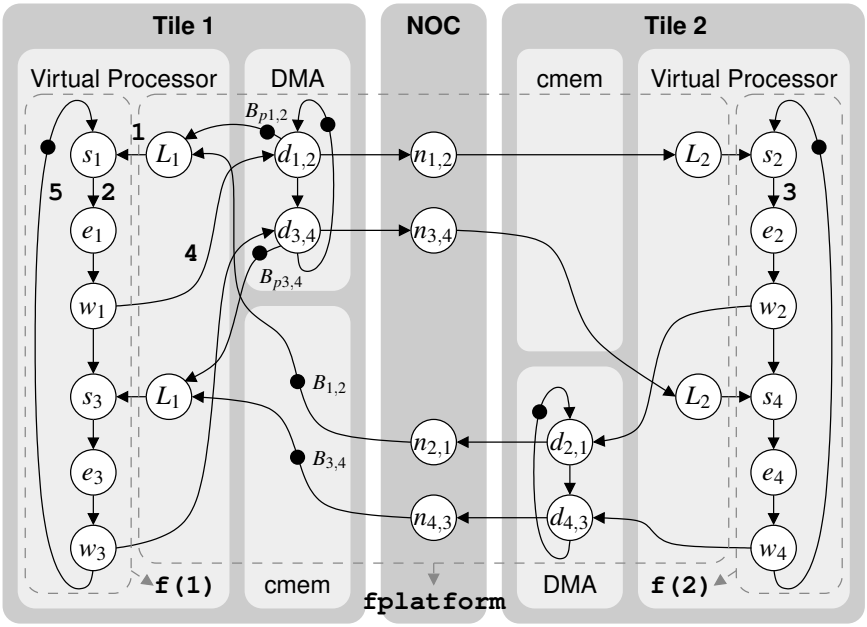


Figure 4.2: Combined application and platform HSDFG with per core DVFS.

these convex problems as DCPs. The CVX convex solver for Matlab takes a DCP as an input and solves it for the objective function within the problem's constraints, returning the values of the free variables that provide a solution (An example of the Matlab code required to perform a convex analysis is presented in Appendix C). In Code 4.1 we present part of the DCP that returns the per core frequencies that minimise the energy consumption of the application illustrated in Figure 4.2.

The actor firing times `s`, core operating frequencies `t` and the SPS period `T` are identified as being variables that the DCP is solved to obtain. The objective function given by Equation 4.8 is set as a minimising objective function. The functions `pow_pos` and `inv_pos` are functions provided by the CVX toolbox that return the convex positive part of the power and inverse functions respectively. Array `A` contains each virtual processor's TDM slot allocation and array `S` contains the TDM table length for each virtual processor, in terms of slots. Two constraints are used to specify that the returned core frequencies `f` have to be ≥ 0 and $\leq$ the maximum possible frequency achievable on the platform `MAX_FREQUENCY`. The application's real-time latency `L` and inverse throughput `inv_pos(R)` requirements are set as constraints that ensure that the period `T` of the resulting SPS meets these requirements.

The inter-actor token dependencies of the HSDFG are represented by one constraint per edge in the graph that is derived from Equation 4.1. In this example, the labels of the actors from Figure 4.2 are integers that identify the location of the actors' attributes in the

```
cvx_begin
  variable s(NUM_ACTORS) % array of actor start times
  variable f(NUM_CORES)  % array of core frequencies
  variable T             % SPS period

  minimise(
    sum((3.353E-5*pow_pos(f*1E-6,3)+2.065+Pshared/NUM_CORES)*A/S
        + Pdedicated)
  )

  f          >= 0
  f          <= MAX_FREQUENCY

  L          >= T % latency requirement
  inv_pos(R) >= T % throughput requirement

  % per edge token constraints
  s(s1)   + T*0 >= s(L1) + t(L1)*inv_pos(fplatform) % edge 1
  s(e1)   + T*0 >= s(s1) + t(s1)*inv_pos(f(1))      % edge 2
  s(e2)   + T*0 >= s(s2) + t(s2)*inv_pos(f(2))      % edge 3
  s(d1_2) + T*0 >= s(w1) + t(w1)*inv_pos(f(1))      % edge 4
  s(s1)   + T*1 >= s(w3) + t(w3)*inv_pos(f(1))      % edge 5
  ... % omitted per edge token constraints
cvx_end
```

Code 4.1: Part of the energy minimising DCP for Figure 4.2.

start time and worst-case work arrays, `s` and `t` respectively. Array `t` contains the constant worst-case work values of the actors. The coefficient of `T` is the number of initial tokens on the edge.

It is not always possible for the CVX solver to find a solution to the DCP within the given constraints, e.g. if the specified real-time requirement is infeasible. If CVX can solve the DCP, the array of `f` contains the per core frequencies and `T` contains the SPS throughput. The array `s` contains admissible SPS actor firing times, which are not used for execution as the application executes following a STS.

The frequencies that are produced using the convex optimisation are in the $\mathbb{R}$ domain. Commonly, it is only possible to perform DVFS to achieve a static set of discrete voltage and frequency levels that were decided at design time. To remain temporally conservative, the closest available frequency level that is greater than the derived optimal frequency is used. By doing this, the resulting frequency levels cannot be said to be optimal for the original objective they were derived for. Due to the monotonicity of dataflow execution, using a higher frequency level cannot cause the application to perform with a lower throughput or longer latency. This results in static slack in the application's schedule that

our technique cannot exploit using the available static frequency levels. We remedy this in Chapter 5 using a run-time technique that observes static and dynamic slack in the application's schedule and performs DVFS.

4.3 Applied in Practice

We continue by demonstrating how our convex optimisation technique is applied in practice to achieve a reduction in application power consumption. We show this for our running example dataflow application that is illustrated, with its mapping, in Figure 4.2. The application is mapped onto two CoMik virtual processors that are mapped onto two physical cores of a CompSOC platform. Both virtual processors are configured with an allocation of five slots out of a ten slot TDM table, with the CoMik and partition slot durations set to 4096 and 65536 cycles at the processor tile's maximum frequency, respectively. The entire CompSOC platform is configured with a frequency of 120 MHz, which serves as the maximum frequency for the processor tiles. Each virtual processor is able to independently perform DVFS, with 16 available frequency levels that linearly scale the maximum frequency, i.e. frequency level one is $120/16 = 7.5$ MHz, up to 120 MHz at frequency level 16.

For static frequency scaling, we want to achieve the lowest possible power consumption while always meeting the application's real-time requirement in the worst case. As described in Section 4.2, the combined application and platform HSDFG can be formulated as a convex program, enabling the derivation of static frequency levels that provide the lowest power consumption while meeting the application's throughput requirements.

To demonstrate the applicability of our model to deriving static low-power frequency levels, we begin by using our model to calculate the power consumption and period of the graph for all 256 possible frequency combinations of the two cores. We achieve this using a similar convex program to that described in Code 4.1, where the frequency combination `f` is specified (i.e. is no longer a free variable in the optimisation) and the objective function is `minimise(T)`. Figure 4.3 shows the results of solving the convex program for the 256 possible frequency combinations. Figure 4.4 shows the power consumption (from the run-time energy accounting) and graph period measured from an FPGA instance of the CompSOC platform. For each frequency combination, the measured graph period from the FPGA platform instance is shorter than the graph period calculated from the model. Our model therefore conservatively bounds the actual throughput of the application.

Many of the 256 points produce the same graph period, but a different power consumption. This occurs due to the nature of this particular application's topology, where one of the core frequencies is responsible for the critical graph cycle. If the frequency on one core is held constant, the frequency of the other core can be set to any level that does not make it part of the critical cycle, while the graph throughput remains the same. Higher frequency levels can therefore increase the power consumption without increasing graph throughput.

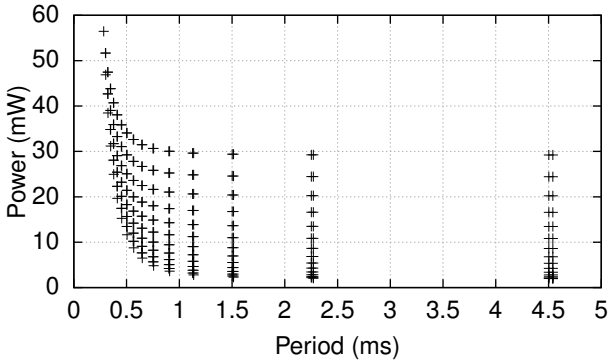


Figure 4.3: Full search using the model.

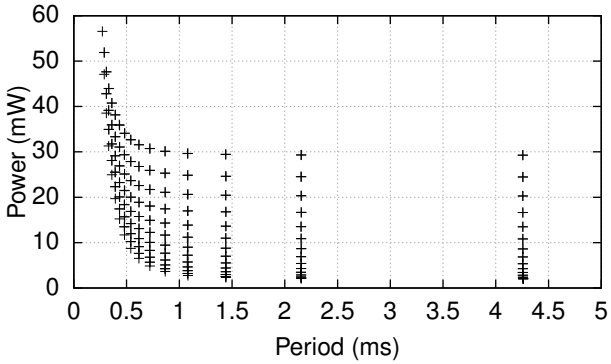


Figure 4.4: FPGA measurements for all frequency combinations.

Figure 4.5 presents the pareto front for the goal of low power-consumption and graph period. This set of frequency combinations produce the lowest power consumption at the particular graph period. As the front is monotonically decreasing for an increase in the graph's period, for a given throughput requirement, the closest point with a period lower than the requirement produces the lowest power option. Performing a full search of the frequency space and deriving a pareto front is not a scalable solution, as the number of possible combinations increases exponentially with the number of cores used by the application.

Our power minimising convex program, enables the derivation of low-power frequency combinations that meet the application's real-time requirement, by solving a single convex program. The calculated frequencies are in the continuous domain, and are therefore rounded up to the nearest available discrete frequency level, to ensure that the graphs period remains timing conservative. The result can no longer be called optimal, as it might not have been necessary to round all of the frequencies up to the nearest discrete

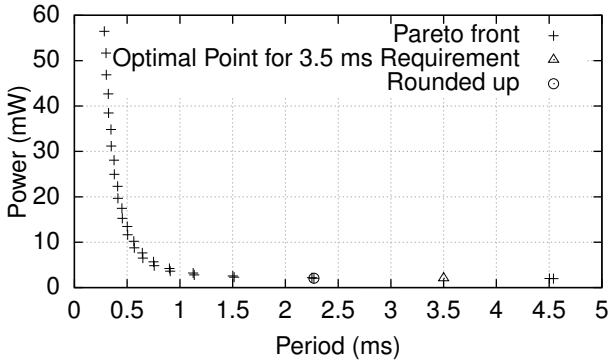


Figure 4.5: Pareto front from the full search using the model.

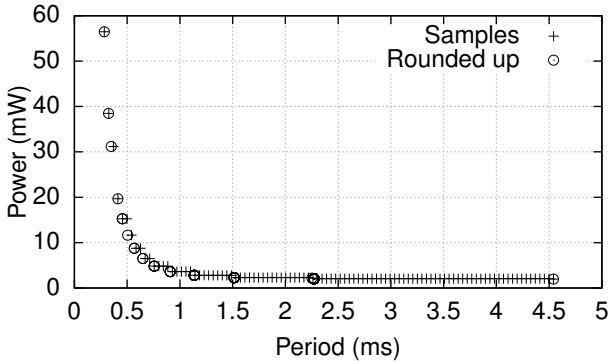


Figure 4.6: Sampling the graph period range at regular intervals.

frequency level, it might have been possible to round some down while still meeting the throughput requirement.

As an example, we use our convex program to find the lowest power frequency combination for a graph period of 3.5 ms (approx. 285.7 graph iterations per second). The resultant combination are the frequencies of 9.73 and 9.65 MHz for core one and two, respectively. The result takes approximately 2 seconds to derive on the ageing dual core laptop that I am writing this thesis on. The graph period and power consumption of this combination is shown as a triangle in Figure 4.5. The frequencies are rounded up to the nearest available frequency level (15 MHz), producing the point shown by a circle in Figure 4.5, which still produces a low power combination while meeting the application's throughput requirement. Figure 4.5 shows the low power frequency combinations that are derived for a range of throughput requirements. The range of period lengths between all of the cores running at maximum frequency and running at minimum frequency, is sampled at regular intervals to produce 100 throughput requirements, indicated by crosses in

Figure 4.6. The power minimising convex program is solved for each of the requirements, producing low power frequency combinations. The resultant frequencies are rounded up to the nearest available discrete frequency level, as indicated by the circles in Figure 4.6. As a result the same rounded up discrete frequency combination can be valid for multiple frequency combinations in the discrete domain.

We have shown that our model conservatively bounds the timing observations as measured on an FPGA instance of the CompSOC platform. Using our model we have demonstrated how a static low power frequency combination is derived using our convex programming technique, that meets the application's real-time requirements.

4.4 Related Work

To present a coherent overview of related power management techniques, we present the related work of static off-line and dynamic on-line power management techniques together in Section 5.4 on Page 136.

4.5 Summary

In this chapter, we have explained how a combined application and CompSOC platform HSDFG can be formulated as a convex optimisation problem that can be solved to find operating frequencies for optimal power consumption. For each optimisation objective, we describe an appropriate objective function that produces either an optimal static frequency for the entire platform, or a single static frequency per core. For each objective function, we also describe how the combined application and platform HSDFG is represented using an optimisation constraint per edge in the graph.

Using the example combined application and platform HSDFG from Figure 4.2, we have presented how the graph is formatted as a DCP to be solved by the CVX convex problem solver for Matlab.

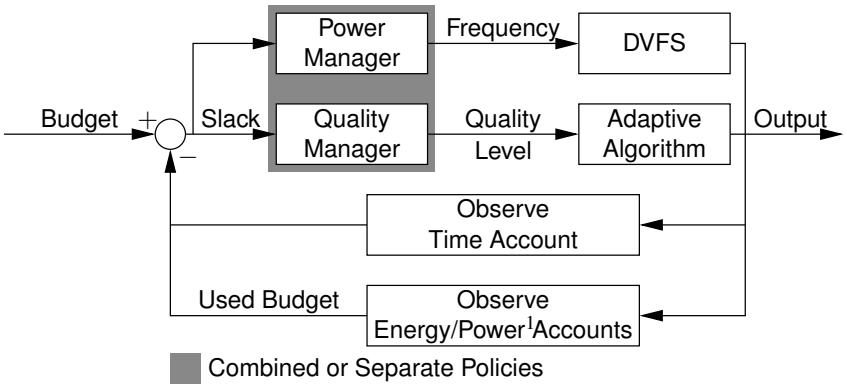
CHAPTER 5

Dynamic Voltage and Frequency Scaling

Dynamic variations in application timing leads to slack (in addition to static slack) at run-time enabling voltage and frequency levels to be lowered. Static voltage and frequency scaling techniques, such as the technique presented in Chapter 4, scale using static slack in the application's worst-case schedule. In this chapter, we propose a novel run-time DVFS technique to take advantage of the dynamically accumulated slack in the actual schedule.

To ensure guaranteed timing conservativeness, run-time power management can only be performed using slack that has been observed in the schedule, as opposed to using a prediction based approach. We propose a closed power management control loop (illustrated in Figure 5.1) that uses the quantity of observed slack as the input for making power management decisions. To achieve this, POSe and CoMik maintain timing and energy accounts as described in Section 3.2 and Section 3.3 respectively. Figure 5.1 also illustrates the use of a quality scaling mechanism in addition to DVFS.

Some adaptive applications can provide a quality scaling mechanism, enabling a reduction in quality in exchange for a reduction in execution time. Applications with adaptive algorithms that can decrease the execution time of tasks for a monotonic decrease in output quality are used to generate more timing slack. Once observed, this slack is used by the power-management policy to scale the application's frequency to a lower level than would otherwise have been possible, while still guaranteeing to meet the application's timing requirement in the worst case.



[1] The power budget is accounted for using an energy account that has a periodic replenishment interval.

Table 5.1: Power/energy management control loop.

The power and quality management policies are created by the application developer and can be combined into a single control policy or used as separate policies. These policies use the observed slack in the application's accounts to decide on the application's operating frequency and quality level, as illustrated in Figure 5.1. Timing and power slack is observed by comparing the application's accounted time and energy over a number of application graph iterations and comparing them with budgeted values, i.e the slack in an account is the budgeted amount minus the used amount. If an account's slack is positive, there is a surplus in the account that can be used to lower the operating frequency and/or increase the quality-level. If an account's slack is negative, there is a deficit in the account, which means that it has used too much of the budgeted resource, e.g. the application's execution takes longer than the budgeted amount of time causing it to miss a deadline. The power- and quality-management policies must ensure that budgets with hard or firm requirements never have negative slack.

In the following section, we explain how a quality-management policy is used to assist in meeting an energy budget in addition to a separate power-management policy. We demonstrate this on a dedicated single core (i.e. non-virtualised) CompSOC platform for an H.263 decoder application that has been modified to enable a trade-off between output quality and execution time. In Section 5.2, we explain how the progress of a dataflow modelled application that is mapped onto multiple processor cores is conservatively estimated in a distributed manner (on each core), enabling temporally conservative DVFS and quality scaling decisions, based solely on locally available run-time information.

5.1 Quality/Power Trade-off Mechanism

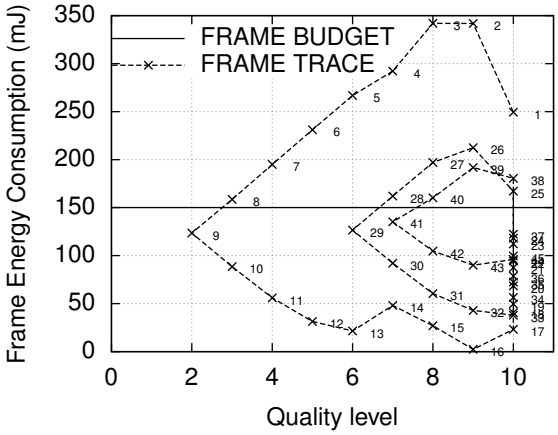


Figure 5.1: Adaptive H.263 decoder per frame trace of quality-level scaling to meet the indicated average energy target.

Adaptive applications [70] may change their execution profile at run-time, enabling, e.g. a slower exact algorithm to be substituted for a faster approximate algorithm at the cost of output quality. For example, in [19, 111], it was shown for an MPEG2 video decoder that output quality may be sacrificed to meet timing constraints by adapting the application.

In this section, we describe a low-complexity technique for application quality scaling, with a relatively small processing and energy overhead. Figure 5.1 shows how an adaptive H.263 decoder [48, 97], using our technique on a single core platform, can scale the quality of selected algorithms in order to meet an average energy-per-frame budget. This is achieved by simply decreasing the quality level whenever the video is consuming more energy per frame than budgeted, and similarly increasing the level when it consumes less. Playing a video on a mobile phone is an example use case that can benefit from such a trade-off. If, for instance, a user wanted to display a video right until the end, but the phone's battery does not contain sufficient charge for this, and also retain enough energy to make a possible emergency phone call. Quality scaling for power reduction enables the user to have the option to watch the entire video at lesser quality, while not exceeding its allocated energy budget. This enables the user to retain some energy in the battery to make their potential emergency phone call.

We demonstrate our technique for an adaptive H.263 decoder application mapped onto a single core of a CompSOC platform instance. We analyse the effectiveness of our

Section 5.1 contains an abridged and updated version of publication [78]

technique for the use case of scaling quality in order to achieve a particular number of decoded frames from an initial energy budget. We show that our technique works with both soft and firm real-time DVFS techniques, and that the timing criticality of the DVFS technique does not significantly affect our technique's ability to trade a decrease in quality for a decrease in energy consumption. From experimentation on a single processing core, we show that our quality-scaling technique is able to extend the number of video frames decoded by up to 45% for the same energy budget, over using the DVFS technique on its own, in exchange for a quality reduction of up to 22dB of Peak Signal to Noise Ratio (PSNR).

5.1.1 Adaptive Application

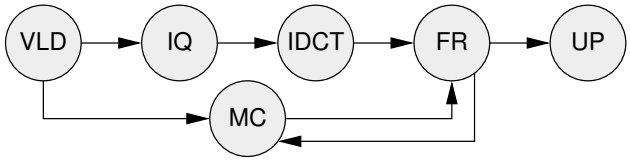


Figure 5.2: H.263 decoder application task graph.

We demonstrate our technique using an adaptive baseline H.263 video decoder application [48]. The H.263 application decodes the input video stream as illustrated in Figure 5.2. The compressed stream first undergoes Variable Length Decoding (VLD). The resultant stream consists of macroblocks, with each macroblock containing frequency encoded YUV information for an 8×8 block of pixels.

Macroblocks belong to either an I-frame or a P-frame. I-Frames contain the information to reconstruct all the pixels of the encoded frame. These frames are reconstituted through Inverse Quantisation (IQ), Inverse Discrete Cosine Transform (IDCT), and Frame Reconstruction (FR). P-frames do not contain the encoded version of the entire frame. Instead these frames contain Motion Compensation (MC) information, that groups pixels with vector translations, allowing the frame to be reconstructed from the previously reconstructed frame. The reconstructed I/P-frames undergo Up Scaling (UP) to fit the allocated display area.

An adaptive H.263 decoder [97] contains parametrised adaptive functions in the application that enables it to decrease the decoder's execution time in exchange for a reduction in the decoder's output quality. The adaptive H.263 decoder used here contains the following two adaptive functions:

1. Parametrised number of decoded AC values in a macroblock.
2. Parametrised up-scaling complexity.

A frequency domain macroblock is encoded using a Discrete Cosine Transform (DCT) [48]. An 8×8 value macroblock of this type consists of a single DC value, that represents

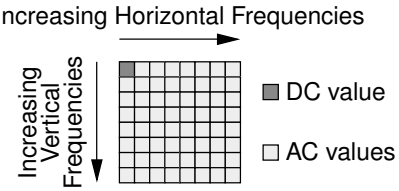


Figure 5.3: 8×8 macroblock.

the average value for the macroblock, and 63 AC values, as illustrated in Figure 5.3. As shown in this 2 dimensional representation of the macroblock, the further in each dimension an AC value is from the DC value, the higher the frequency is that it represents in that dimension. By selectively ignoring AC values, the time taken for the IDCT task of the decoder may be decreased, in exchange for a reduction in reproduction quality of the spatially encoded macroblock. The encoding process places less value on higher frequency information in the macroblock, due to human perception. Similarly the adaptable function in the decoder allows scaling of the amount of processed AC values, ignoring AC values at the higher end of the frequency spectrum first.

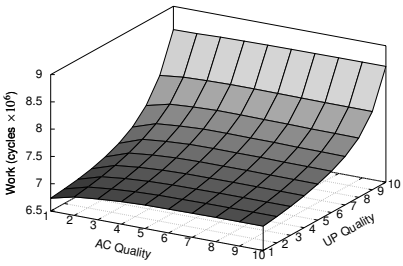
Parametrised up-scaling complexity is achieved by a similarity-threshold parameter passed to a bi-linear interpolation algorithm. If the two pixels under comparison are similar to within the threshold value, then no interpolation takes place. In this eventuality one of the compared values is simply reproduced. If the two pixels are suitably dissimilar, bi-linear interpolation is performed. By relaxing the similarity threshold, a reduction in execution time is achieved at the expense of the reproduction quality of the final image.

Scalable Quality Mechanisms

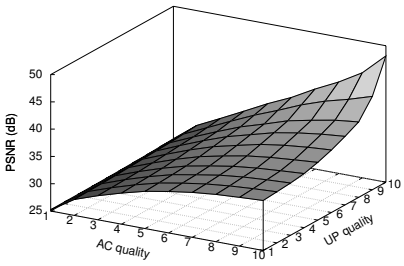
The quality scalable functions of the H.263 have data dependent execution times and quality of output. In this section, we experimentally evaluate the H.263 decoder's scalable quality mechanisms, using an input set of three videos. The H.263 decoder has two scalable quality mechanisms, each having 10 discrete quality levels, giving 100 possible quality combinations that may be requested. For each of these combinations we evaluate both the amount of work that is performed in order to produce a frame of decoded video at a particular requested quality-level and also the image reproduction quality of the adaptive algorithms.

To obtain a measurement of performed work for each of these quality combinations, we decode the input videos with the quality-levels fixed at a single combination for the duration of the decoding. In order to provide a single value per quality-level, we take the average work required to decode a video frame from the first 30 decoded frames. The reproduction quality of the quality scaling mechanisms is also be measured for each of the combinations. The quality is measured as Peak Signal to Noise Ratio (PSNR), of the reproduced frame to the original encoded frame. To provide a single value per

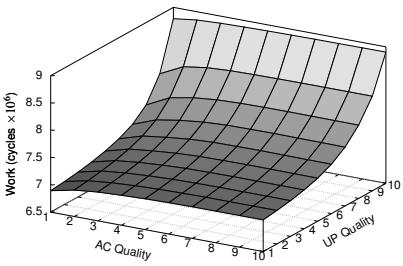
quality-level, we take the average PSNR of the first 30 decoded frames. The results of our evaluation of the H.263's quality scaling mechanism is presented in Figure 5.4 for the three input videos.



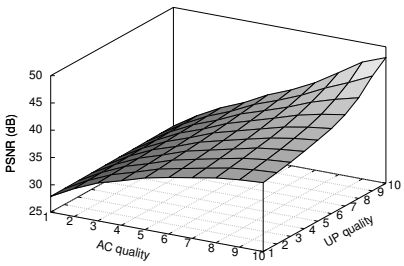
(a) Akiyo average work.



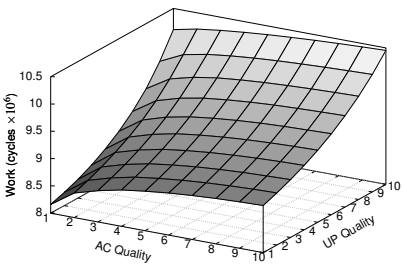
(b) Akiyo average PSNR.



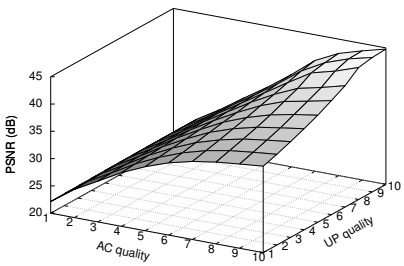
(c) Tree average work.



(d) Tree average PSNR.



(e) Bus average work.



(f) Bus average PSNR.

Figure 5.4: Quality mechanism performance, for the average of 30 decoded frames.

Figures 5.4a, 5.4c and 5.4e show the work to produce a decoded frame at each quality-level combination for the decoded akiyo, tree and bus reference videos respectively. The

resultant surfaces from each video are not exactly the same due to the data dependent nature of the quality scaling mechanism's. Even with the data dependent variation, it is clear that by decreasing either of the two quality scaling mechanisms, there is a monotonic reduction in work that must be performed to decode a frame. This is a useful property of the quality-scaling mechanisms as it ensures that requesting a lower quality-level will not lead to extra work having to be performed.

From Figures 5.4a, 5.4c and 5.4e, it is apparent that the up-scaler quality mechanism produces a larger reduction in work needed to decode a frame, across its range of quality-levels, than the AC-values quality mechanism. This demonstrates that the reduction of work by quality scalable algorithms is algorithm specific. It depends on the physical nature of the algorithm, or the minimum acceptable quality of output that is still useful.

Figures 5.4b, 5.4d and 5.4f show the PSNR against requested quality-level for the decoded akiyo, tree and bus reference videos respectively. As with the surfaces produced by exchanging a reduction in quality for a reduction in work, the resultant surfaces here are also vary due to the data dependent nature of the quality scaling mechanism. Nevertheless, the results from the three input videos show a monotonic decrease in received quality-level as the requested quality-levels are decreased. Reducing the quality-level for the AC-value mechanism shows the largest quality reduction at its lowest quality value, with an $\approx$10 dB difference in comparison to the up-scaler mechanism at its lowest quality value.

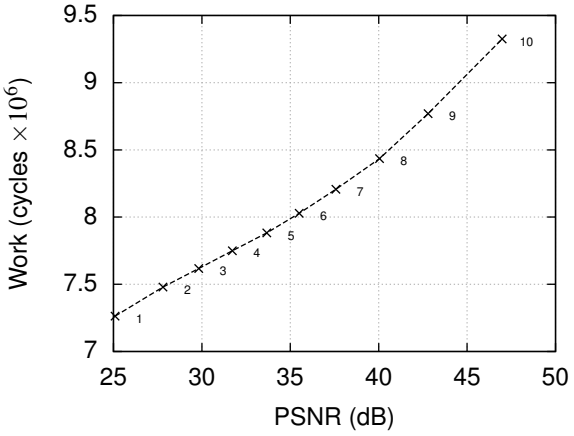


Figure 5.5: Work against PSNR for the 10 selected quality levels.

To simplify the quality scaling to a single mechanism, requiring only one quality-level to be specified, we select a subset of the available 100 quality combinations to create a single monotonic quality to work trade-off, as presented in Figure 5.5. To achieve this, we simply scale both quality mechanisms equally, producing 10 possible quality levels with the desired monotonic trade-off. From Figure 5.5 we can see that on average a work reduction of 2×10^6 cycles is achievable for a quality reduction of 22dB PSNR.

5.1.2 Quality for Power Reduction

We continue by describing how our temporal and power constrained quality scaling technique works in general. We further show how it may be applied in practice using the CompSOC platform and the adaptive H.263 decoder described in Section 5.1.1.

A reduction in quality creates temporal slack that can then be exchanged for a reduction in power consumption through DVFS. These power management decisions for real-time applications must be taken in the context of their timing requirements. We achieve this on the CompSOC platform at run-time, by observing slack in the application's timing budget (schedule) and selecting an appropriate DVFS level that slows the application down but that still enables the application to meet its real-time requirement in the worst case. This can be achieved simply as follows, if a block of code can execute to completion without stalling that takes duration t to execute at frequency f, and it has slack of duration z, then it can conservatively scale to any frequency $\geq f_z$, with f_z calculated as follows:

$$f_z = f \cdot \frac{t}{t+z} \tag{5.1}$$

This is guaranteed to be conservative because the finishing time of the block of code executing at frequency f_z, cannot be later than the finishing time of the code starting z cycles later and executing at frequency f. How and when to perform the scaling depends on the power management policy, which we proceed to explain in the following section.

Applied in General

To make our technique generally applicable for systems that can perform DVFS, we introduce a quality management layer between the application and the system's existing power-management, as illustrated in Figure 5.6. Independent quality- and power-management policies are shown in Table 5.2. Having independent policies enables the technique to be applied to platforms that already have DVFS power-management policies. As they are independent, the quality- and power-management policies may also operate at different granularities, e.g. for the H.263 decoder, the quality may change every frame while the frequency changes every macroblock (this is just an example, not a suggested configuration). This allows our technique to be more general and therefore more widely applicable, than by having a single combined quality- and power-management policy.

Table 5.2 presents policies for four scenarios of time and energy budget under- and overuse, with the action that is taken in each case. If the temporal budget has surplus (i.e. the application is running ahead of schedule) then the power-manager can lower the frequency and voltage conserving power, while still meeting the temporal requirement. If the temporal budget is showing that the application is running a deficit (i.e. the application is running behind schedule), then the power-manager must increase the frequency and voltage to meet the temporal requirement. The increase in the frequency also means an increase in power consumption, and is therefore dependent on the slack

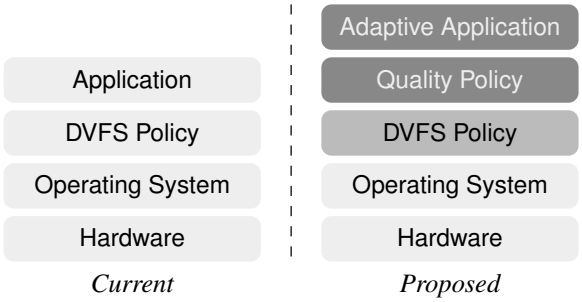


Figure 5.6: System hierarchy.

		Time	
		surplus	*deficit*
Energy	*surplus*	scale frequency[1]	maximum frequency
	deficit	scale frequency[1]	maximum frequency

[1] using conservative frequency derivation from Equation 5.1.

(a) Power-management DVFS policy. (Regardless of energy/power budget constraint)

		Time	
		surplus	*deficit*
Energy	*surplus*	increase quality	decrease quality
	deficit	decrease quality	decrease quality

(b) Quality-management policy

Table 5.2: Quality- and power-management policies, based on run-time temporal/energy/power budget information.

in the energy/power budget. If the temporal and energy/power budgets are concurrently running deficits, then a conflicting situation arises whereby the system needs to increase the frequency in order to meet the temporal budgets constraint, while at the same time needing to lower energy/power consumption in order to meet the energy/power budgets constraint. Which one to prioritise is therefore a policy decision and depends on the requirements of the application. The power-management policy in Table 5.2a, prioritises the application's real-time requirement over its energy/power requirement, i.e. if the time budget is running a deficit, the frequency is increased regardless of the energy/power budget constraint.

Scaling the quality of an adaptive application, such as the adaptive H.263 decoder described in Section 5.1.1, enables a trade-off in quality for a reduction in execution time. Table 5.2b shows a simple example quality-management policy. In this policy the quality is decreased when the temporal or energy/power budgets are running a deficit. Quality may be increased again whenever all the budgets have slack. This directly assists in

meeting the application's time budget by reducing the amount of work that the application has to perform to decode a frame. This reduction in work also assists in meeting the application's energy budget, even without frequency scaling. When used in combination with the power-management policy described in Table 5.2a, the reduction in work due to quality scaling can also translate into a reduction in operating frequency and hence a lower power consumption.

Applied to the CompSOC Platform

We proceed to explain how our quality-management technique is applied to the CompSOC platform. The quality- and power-management control loop that is illustrated in Figure 5.1 is implemented in software as a function at the application-level. We implement the quality- and power-management as independent functions, following the quality- and power-management schemes described in Table 5.2. In our implementation, the quality-manager returns the quality-level as an integer from 1 to 10, with 1 being the lowest quality and 10 being the highest. The adaptive functions take this number as an input and scale their algorithm accordingly. The power-management function derives a frequency to use, following Equation 5.1 calls the TIFU driver function that sets the frequency. The quality- and power-management functions are called by the application. The frequency of calling the management functions is a design decision. For the H.263 decoder, the quality- and power-managers are called on the granularity of video frames.

The H.263 decoder is a real-time application. POSe represents this requirement as a time budget. The H.263 decoder can also be given an energy budget, specifying the maximum amount of energy the application can use to complete the video. This is translated into a target energy consumption per frame, which is effectively a power budget. Since the timing requirement takes priority over the energy requirement in the quality- and power-management policies described in Table 5.2, the energy requirement is specified with a soft criticality.

The quality- and power-managers use POSe's run-time accounting information to observe slack. The relevant budgeting information, provided by the POSe power-management API, consists of the following:

- `time_budget` Amount of time in system time budgeted for n application graph iterations.
- `used_time_budget` Current time in system time used from the temporal budget.
- `energy_budget` Amount of energy budgeted for the entire application execution (i.e. for decoding the entire video).
- `power_budget` Amount of energy budgeted for the current time interval (i.e. video frame duration).
- `used_power_budget` Current energy used from the power budget, for the current time interval (i.e. video frame duration).

where system time is the time measured in cycles of the unscaled reference clock of the TIFU. There is a linear relationship between the progression of system time and wall time.

The H.263 application calls the `control_loop` function, as described in Code 5.1, before every frame. The amount of temporal slack that is available in the budget is calculated by subtracting the `used_time_budget` from the `time_budget`. The `power_budget` is calculated by dividing the `energy_budget` by the number of frames in the video to obtain the power budget constraint, in terms of energy per frame. The amount of slack that is available in the power budget is calculated by subtracting the `used_power_budget` from the `power_budget`.

```
void control_loop(){
  time_slack   = getTimeBudget();
  time_slack  -= getUsedTimeBudget();
  power_slack  = getPowerBudget();
  power_slack -= getUsedPowerBudget();

  app_quality = h263_quality_manager(time_slack,power_slack);

  h263_power_manager(time_slack);
}
```

Code 5.1: Quality- and power-management control loop.

After the `control_loop` function calculates the time and power slacks, it calls the quality- and power-management functions. The `h263_quality_manager`, described in Code 5.2, follows the policy from Table 5.2b. For this policy, if both temporal and power slack are positive, then the quality-level is incremented. If either temporal or power slack is negative, then the quality level is decremented. For all other slack combinations, the quality-level is maintained at its current level. If the quality-level is outside the range of 0 to 10, then the closest available quality level is chosen and returned. The `control_loop` uses the returned value to set the global `app_quality` variable for the adaptive functions.

The `h263_power_manager`, described in Code 5.3, follows the policy from Table 5.2a. In this policy, if time slack is negative the frequency level is set to the maximum available frequency (`MAX_FREQUENCY`), otherwise the frequency is scaled following Equation 5.1. The frequency is scaled depending on how much `time_slack` there is and the amount of time it takes to decode a single frame (`time_to_decode_frame`) at the maximum available frequency. For conservative frequency scaling, the `time_to_decode_frame` must be at least as long as the worst-case time to decode a frame. For soft criticality frequency scaling, a speculative `time_to_decode_frame` can be used, which can be obtained by observing the time taken to decode previous frames. The derived `frequency_scale` is multiplied by the number of available frequencies (`NUM_FREQUENCIES`) and rounded up to the next available frequency to remain con-

```
int h263_quality_manager(time_slack, power_slack){
  /* Quality policy */
  if(time_slack < 0 || power_slack < 0){
    quality_level = getCurrentQualityLevel() - 1;
  }else if(time_slack > 0 && power_slack > 0){
    quality_level = getCurrentQualityLevel() + 1;
  }

  /* stay within available quality range */
  quality_level = max(quality_level,0);
  quality_level = min(quality_level,10);

  return quality_level;
}
```

Code 5.2: Quality-management function.

servative. The result is multiplied by the difference in frequency between two frequency levels, to obtain the `frequency_level` to be requested. If the `frequency_level` is outside the available range of `MIN_FREQUENCY` to `MAX_FREQUENCY`, it is rounded to the nearest available frequency. The `frequency_level` is then set by calling the `setfrequency` function with the required frequency.

Changes in the time taken to decode a frame are observed the next time the control loop is executed. For instance, lowering the quality in this iteration reduces the time taken to decode a frame, which can be observed by the control loop the next time that it is invoked.

5.1.3 Case Study

Having presented how our technique is applied to the adaptive H.263 decoder from Section 5.1.1 for the CompSOC platform, we proceed to evaluate the performance of our technique. We achieve this using an implementation of the adaptive H.263 decoder executing on an FPGA prototyped CompSOC platform. We investigate the relationship between the requested quality level and the output quality of the decoded video frame, measured as a Peak Signal to Noise Ratio (PSNR) in comparison to the reference frame, decoded at the highest quality settings. We also investigate the relationship between the requested quality level and the amount of work required to decode a single frame of video. Following this, we evaluate our quality-scaling technique by providing an in depth analysis of its application to the use case of decoding a particular number of video frames for a given energy budget.

While our experimental results show absolute power and energy estimates, *we do not*

```
void h263_power_manager(time_slack){
  /* Power policy */
  if(time_slack < 0){
    frequency_level = MAX_FREQUENCY;
  }else{
    frequency_scale = time_to_decode_frame/(time_to_decode_frame
        + time_slack);
    frequency_level = ceil(frequency_scale * NUM_FREQUENCIES)*
        MAX_FREQUENCY/NUM_FREQUENCIES;
  }

  /* stay within available frequency range */
  frequency_level = max(frequency_level,MIN_FREQUENCY);
  frequency_level = min(frequency_level,MAX_FREQUENCY);

  setfrequency(frequency_level);
}
```

Code 5.3: Power-management function.

make any claims about the accuracy of our used power model [1]. The power model that we use is *for comparative purposes only*, enabling us to evaluate whether our technique provides an improvement in comparison to the same situation without our technique. Our processor power model is based on the power consumption estimate of the MicroBlaze processor, at 120MHz, for the ml605 board's virtex 6 FPGA. We obtained an estimate of 348mW using the Xilinx "Xpower Analyzer" tool, for a mapped and routed instance of a MicroBlaze processor on the FPGA. What is important for demonstrating the validity of our technique, is that by lowering the processor's operating frequency (and voltage to match) the processor's power consumption decreases. As such, we use a simple linear relationship between frequency and power, but emphasise that our technique also works for other monotonic frequency/power models, such as those with a quadratic or cubic relationship, as may be obtained from the parametrised model described in [51].

We start our experimental evaluation by investigating the effectiveness of our quality- and power-scaling technique to meet a target number of decoded video frames within a given energy budget, e.g. the remaining energy in a battery. We decode a video using the adaptive H.263 decoder for the five configurations described in Table 5.3. The decoder is given an energy budget of 15 J to decode 45 video frames, while maintaining a real-time throughput requirement of 10 frames per second. The depletion of the energy budget during the three different runs can be seen in Figure 5.7. The baseline test is carried out

[1] The experimentation results in this section were published in [78] and used a simple linear approximation of the frequency power trade-off. These results are still valid for the purpose of demonstrating the effectiveness of our technique, but may cause confusion if absolute energy numbers are compared with experimentation elsewhere in this thesis.

Worst-case Frame Timing	Frequency Scaling	Quality Scaling	Annotated as
N/A[1]	✗	✗	NO FS NO Q
✗	speculative	✗	FS NO Q
✗	speculative	✓	FS Q
✓	conservative	✗	WC FS NO Q
✓	conservative	✓	WC FS Q

[1] The time to decode a frame is used by the scaling mechanisms and therefore has no effect when both scaling mechanisms are not in use.

Table 5.3: Experimentation configurations.

without frequency scaling or quality scaling (NO FS NO Q), at the processor's maximum frequency. As can be seen in Figure 5.7 running at the processor's maximum frequency causes the energy budget to deplete at a relatively fast rate, with the energy budget depleted by the 28th frame.

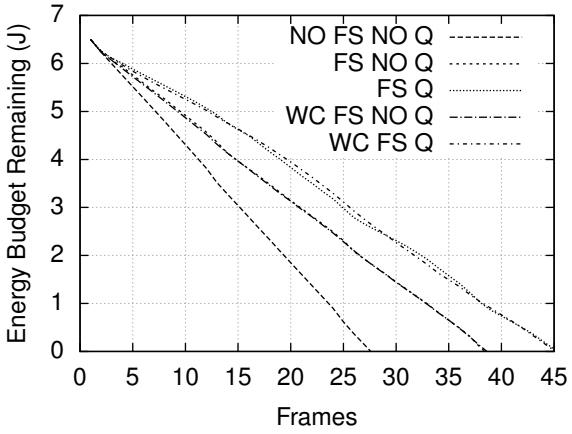


Figure 5.7: Remaining energy budget.

Frequency scaling is performed in accordance with either a conservative or speculative power-management policy. The conservative policy derives a frequency using Equation 5.1 to use until the next invocation of the control loop, for the amount of slack available and the worst-case time to decode a frame. The actual decoding time of video frames cannot use more slack than is available and therefore the H.263 decoder is guaranteed to meet its frame rate.

Deriving an application's worst-case timings can be complicated (even impossible), requiring detailed knowledge of the application's algorithms. While worst-case timings

are necessary to be able to give timing guarantees for hard or firm real-time applications, soft real-time applications, such as the H.263 video decoder, can incur some deadline misses without devaluing the output significantly. We therefore also present experimental evaluations of our quality- and power-scaling technique using speculative frame decoding times. We use a simple speculation method of using the time taken to decode the previous frame when deriving an appropriate frequency to use.

Enabling frequency-scaling using worst-case (WC FS NO Q) and speculative (FS NO Q) frame decoding times enables the processor to run at lower frequencies than the baseline (NO FS NO Q), thereby consuming energy at a lower rate. Figure 5.8 shows that given a per frame work budget suitable to achieve 10 fps on a processor with a maximum frequency of 120 MHz, that the power-management is able to scale the frequency while meeting the real-time requirement. In both the case of the worst-case (WC FS NO Q) and speculative (FS NO Q) frame times, the slack always remains positive, which shows that the 10 fps was met.

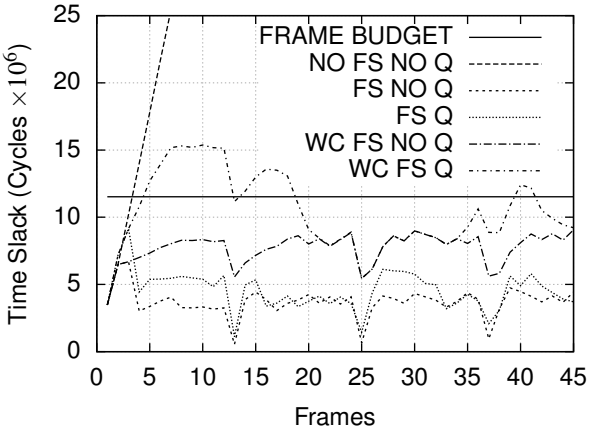


Figure 5.8: Remaining work budget after each frame.

In comparison to the situation with no frequency-scaling (NO FS NO Q) in Figure 5.8, by running continuously at maximum frequency the H.263 decoder under uses its work budget, causing it to continuously accumulate timing slack. In Figure 5.7 it can be seen that by enabling frequency-scaling ((WC) FS NO Q) that the energy budget now stretches to the 37th frame, which is a 32% improvement but is still short of the 45 frame target.

The application's timing slack is accumulative, meaning that the application retains slack between frames. If the conservative policy runs faster than the speculative policy in a particular frame then the conservative policy has more slack to use for frequency scaling in the next frame. Figure 5.9 shows that for this investigation, on average the conservative and speculative policies use similar frequencies, which in turn translates into similar energy consumption, as is apparent from Figure 5.7 where the FS NO Q and WC

FS NO Q lines almost completely overlap.

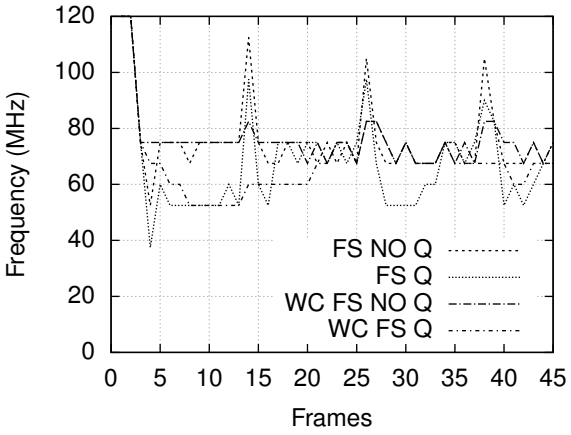
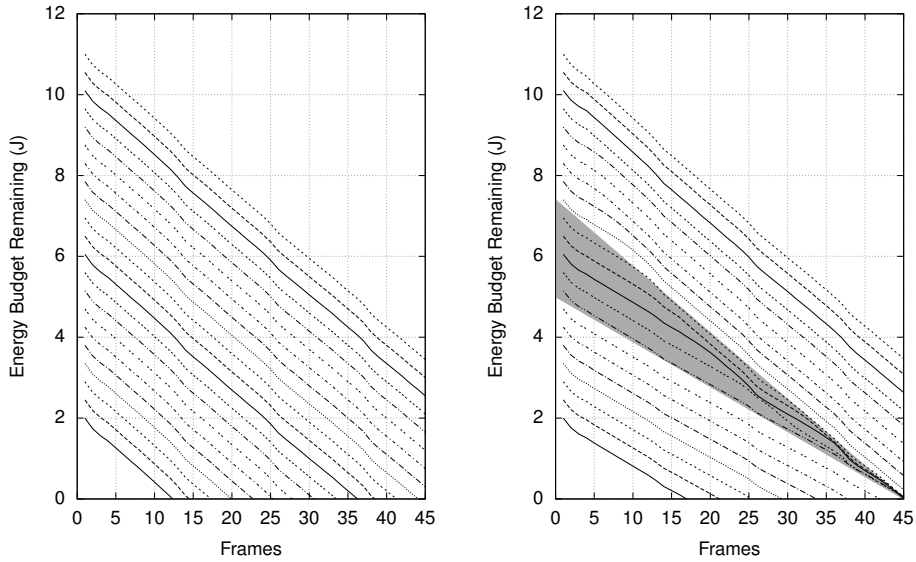


Figure 5.9: Processor frequency-level per frame.

Frequency-scaling alone is not able to reduce the energy consumption rate further without affecting the video's decoded frame rate. By enabling quality-scaling in conjunction with frequency-scaling ((WC) FS Q), the 45 frame target is met within the given energy budget, as shown in Figure 5.7. With the use of quality-scaling and frequency-scaling ((WC) FS Q), the same initial energy budget lasted for 22% more frames than frequency-scaling alone ((WC) FS NO Q), and for 60% more frames than without frequency- and quality-scaling (NO FS NO Q).

We proceed to test the effectiveness of our quality- and power-management technique, for a range of energy budgets, e.g. a range of remaining energy in a battery. For a given energy budget within this range, we run the experiment again with and without quality scaling, for a minimum frame requirement of 45 frames, producing the figures shown in Figure 5.10. Figure 5.10a demonstrates the energy budget depletions for frequency scaling without quality scaling (WC FS NO Q). The power-management policy described in Table 5.2a scales the frequency to meet temporal requirements but does not attempt to meet the energy target for 45 frames. Figure 5.10b demonstrates the battery depletion with both frequency scaling and quality scaling enabled (WC FS Q). In contrast to the power-management policy, the quality-management policy described in Table 5.2b tries to meet energy requirements. This can be seen in Figure 5.10b as the quality-manager tries to make the energy budget last for 45 decode frames. Some starting energy budgets are too low and cannot stretch to 45 frames, even at the lowest quality setting, and other starting energy budgets are so large that the 45 frame target is met without any quality scaling. A funnel shaped region exists in Figure 5.10b where run-time adjustments made by the quality manager are effective to meet the 45 frame requirement.

The total energy budget is divided among the number of frames to create a per frame



(a) Without quality scaling **(b)** With quality scaling

Figure 5.10: Energy budget depletion from various starting points, while decoding H.263 video with a soft 45 frame minimum requirement.

energy budget. Figure 5.1 shows a per frame trace of how the quality manager adjusts the quality level in respect to this budget, for the policy described in Table 5.2b. Whenever there is an energy budget surplus, represented by the trace being below the budget line, the quality is increased one quality-level per frame. Whenever there is a deficit, represented by the trace being above the budget line, the quality-level is decreased by one level per frame. With quality scaling enabled the policy keeps the energy per frame close to the budget. This is sufficient to meet the energy budget's soft real-time requirement.

The outcome of the five different experimental runs are shown in relation to the per frame energy budget in Figure 5.11. Both runs with quality scaling enabled, FS Q and WC FS Q, are shown to keep the energy consumption close to the budgeted amount. The quality levels that they use to achieve this are shown in Figure 5.12. From this graph it can be seen that the quality management with the conservative frame decoding time produces the highest quality for more frames, but also reaches the lowest quality level of the two runs. The quality-management policy does not take the frame decoding time into account, only if the work budget has a surplus or a deficit. Even though the work budget continuously had a surplus when quality scaling was enabled ((WC) FS Q), as can be seen in Figure 5.8, the difference between the speculative and conservative frame decoding times affect the frequency level that the power-manager derives and hence the rate of energy consumption. As per Figure 5.1, this in turn affects the chosen quality level by

the quality management, which also affects the work required to decode a frame, leaving more/less slack and energy for the next quality- and power-management decisions.

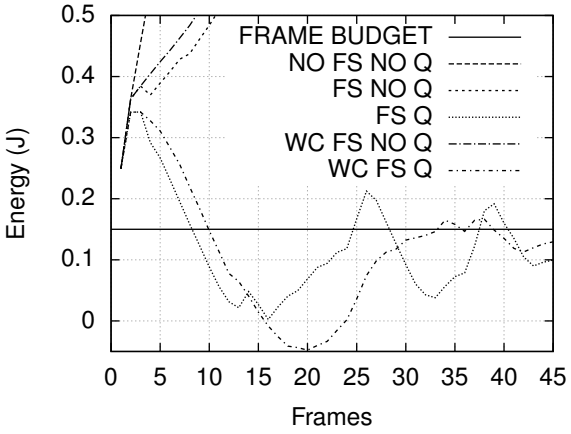


Figure 5.11: Energy budget consumption per frame.

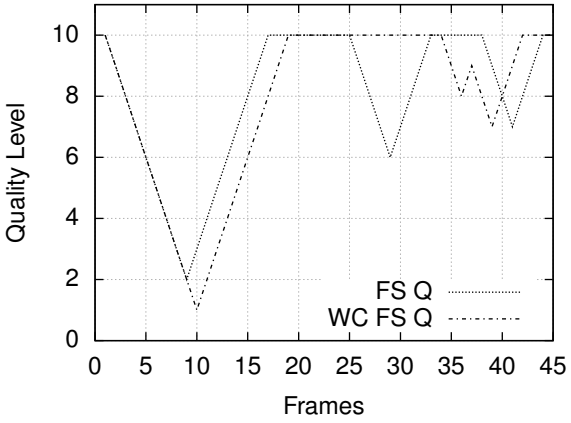


Figure 5.12: Requested quality-level.

A reduction in quality-level translates into a reduction in work that is performed to decode the frame. This in turn translates into an accumulation of extra slack in the work budget, thereby enabling lower frequencies to be used to decode the frame while still meeting the application's throughput requirement. This effect can be seen in Figure 5.9 where the frequency-scaling and quality-scaling combined ((WC) FS Q) is able to run more often at lower frequencies than without quality-scaling ((WC) FS NO Q). This is

achievable while meeting the video decoder's throughput requirement, as can be seen in Figure 5.8. While the choice of a conservative or speculative frame decode time does not have a direct affect on quality management decisions, it does affect the frequency derived by the power-manager which affects the application's energy consumption, and hence indirectly also affects the quality management. From our experimentation, the choice of a conservative (WC) or speculative frame decode time did not cause a significant difference in the ability of our technique to meet the application's energy budget, as can be seen in Figure 5.7. While the difference is small for this example, it is likely that a greater difference could be observed for other applications or methods of speculation.

5.1.4 Conclusion

The energy and power savings that can be made using quality scaling, with adaptive applications, are application and platform dependent. We show how these scaling mechanisms may be applied in a practical context for an H.263 adaptive real-time application executing on an existing MPSOC platform. By using independent power- and quality-managers our technique is able to be integrated more easily onto platforms with existing real-time DVFS power-management techniques.

Through experimentation, using an FPGA prototyped CompSOC platform, we show that quality scaling enables the same level of energy budget to be used to decode more frames than with frequency scaling alone. From our experimentation we show that the same level of energy budget can be used to decode up to 45% more frames when using quality-scaling, but at a cost to image reproduction quality of up to 22 dB PSNR.

While we have successfully demonstrated the applicability of quality scaling adaptive applications, we proceed in the rest of this thesis to explain our static and dynamic power management techniques without the use of quality scaling.

5.2 Distributed Real-time Multi-Core DVFS

Implementing a DVFS control loop, such as illustrated in Figure 5.1, in an application mapped onto multiple cores is non-trivial. As with the application itself, run-time application progress information is distributed across the cores. One solution is to explicitly communicate each core's local progress information to a centralised point that collates this information, derives appropriate conservative frequency-levels and explicitly communicates this information back to the cores. This method is not scalable as each core must be able to communicate with a single point requiring physical communication infrastructure to achieve this.

Conservatively performing DVFS in a distributed manner using locally available application progress information is only possible if the locally available information does not overestimate the progress of the rest of the application. Overestimating the application's progress can lead to a selection of a lower frequency-level causing the application to potentially violate its timing requirement.

In this section, we start by describing how slack in a schedule is conservatively used to scale the period of an SPS schedule without violating its throughput requirement. We follow this with a detailed description of how implicitly communicated global progress information in dataflow modelled applications is used to calculated conservative global slack estimates using only locally available progress information. We finish this section with a description of a conservative method for performing run-time DVFS using the locally observed conservative global slack information.

5.2.1 Conservative Multi-Core Distributed Slack Observation

Our run-time power management control loop requires conservative slack observations in order to derive DVFS levels that reduce power consumption while meeting the application's real-time requirements. We therefore continue by showing how slack may be observed in a conservative manner on a distributed multi-core system.

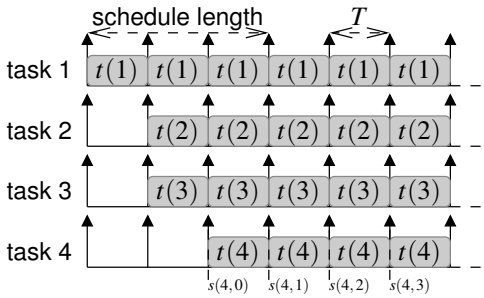


Figure 5.13: Example WCSTS (and SPS) for the HSDFG illustrated in Figure 2.27a

Figure 2.27a illustrates an example WCSTS for our running example mapped dataflow application from Figure 2.27b. The graph executes with period T, enabling it to meet the minimum throughput requirement of T^{-1}. As illustrated in Figure 5.14, dynamic slack in the schedule occurs whenever the tasks execute at less than their WCET. Apart from actual-case task execution, dynamic slack can also occur due to actual-case communication times, due to TDM alignment in the NoC, or between the cores, etc.

As described in Section 2.1.2, a WCSTS can be conservatively modelled as an SPS for analysis purposes, as illustrated in Figure 5.13. Figure 5.15 presents another possible SPS for our running example application where the application's execution period T is equal to the schedule length, providing a less cluttered image for illustrative purposes.

SPS Analysis for STS Execution

Figure 5.16 illustrates the same SPS task start times from Figure 5.15, but with the same varying task execution times from Figure 5.14. Tasks have the same start times $s(i,k)$ but now finish earlier creating gaps or slack in the schedule where no tasks are firing. These

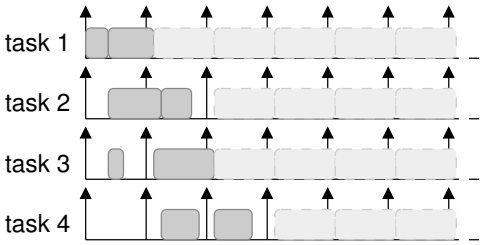


Figure 5.14: Possible STS schedule with task execution times $\leq$ worst-case

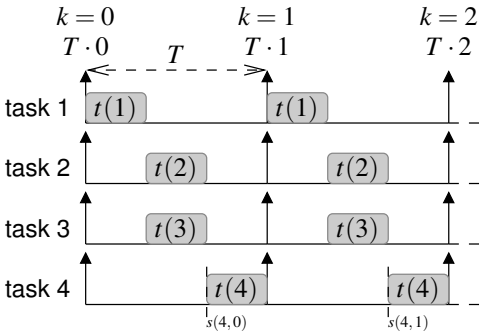


Figure 5.15: Another possible SPS for the HSDFG illustrated in Figure 2.27a with the graph annotated with some worst case execution times $t(v)$

regions where tasks cannot fire due to the restriction of the SPS task start time are marked in Figure 5.16 as unusable slack.

As is illustrated in Figure 5.14, tasks scheduled using a STS fire as soon as there is a token on their incoming edges, enabling the use of the unusable slack from Figure 5.16. Since start times of a HSDFG following a STS are always earlier or the same as the same HSDFG following a SPS, if an SPS is guaranteed to temporally conservative the STS is also guaranteed to be temporally conservative. As such, we give temporal guarantees as if the HSDFG is scheduled as an SPS while actually scheduling the HSDFG following a STS.

We clarify this further with the example illustrated in Figure 5.17. In this example, the graph has an inverse throughput requirement T that must be met. After one application graph iteration following a STS, the graph has finished Z time units earlier than its requirement and therefore has Z time units of slack. The slack is used to derive a suitable frequency level to permit the SPS of the second graph iteration to finish while satisfying the throughput requirement.

The frequency level that was derived for the SPS is then applied to the second iteration of the graph, but scheduled following a STS, the result of which is illustrated in Figure 5.18. The second iteration of the graph also meets the requirement with some slack to spare,

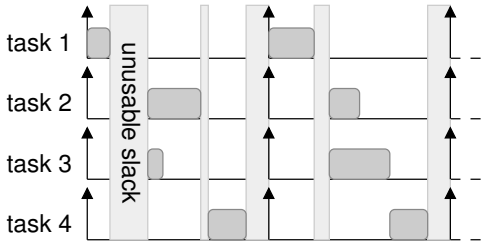


Figure 5.16: Same SPS from Figure 5.15 with task execution times shorter than the worst case

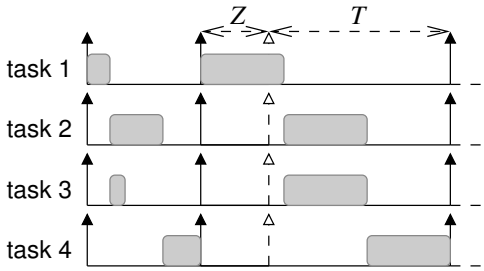


Figure 5.17: Slack Z from an STS iteration used to perform DVFS on the following SPS iteration

allowing the following iteration of the graph to scale, and so on.

Local STS Application-level Slack Observation

The aim of our distributed power management technique is to ascertain conservative global progress using locally observable progress information. We observe local application-level progress as slack. The further an application is ahead of its timing requirement the more slack it has.

The calculation of slack requires a frame of reference to measure against. The application's throughput requirement is translated into a set of deadlines that a set number of application iterations must complete by. In a multi-core system, only some of the application's tasks are executed locally. POSe executes the local tasks following an SOS. Each SOS iteration fires the tasks for a single application graph iteration. Depending on the number of iterations that the application's timing requirement is for, after the same number SOS iterations the application can measure its current time against its deadline to calculate its locally observed timing slack.

Figure 5.19 illustrates the timing of an STS of our running example application. Ideally, after the end of the application graph iteration the end of an application iteration the application level slack Z is measured and frequency scaling is performed, as illustrated

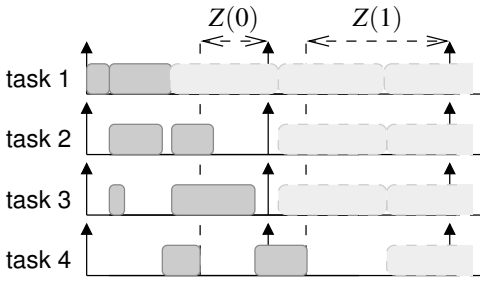


Figure 5.18: Guaranteed conservative second iteration SPS frequency level applied to second iteration STS

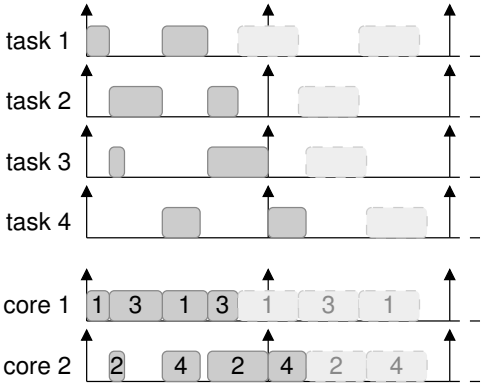


Figure 5.19: Example STS of HSDFG from Figure 2.27b shown per task and per core

in Figure 5.20.

To be able to conservatively perform DVFS to reduce power consumption, we require a method to conservatively translate the locally observed progress information into global application progress information. Using this global progress information, we also require a method to convert this information into a frequency level to be set locally that is guaranteed not to violate the application's throughput requirement.

Observing Global Application Progress Implicitly

The combined application and platform CompSOC HSDFGs, as illustrated in Figure 2.27, are fully connected directed graphs. This means that every dataflow actor has a path to and from every other actor in the graph. Due to dataflow monotonicity, an earlier finishing of any of the actors can not lead to a later enabling of subsequent actors in the graph, but can lead to an earlier enabling. A task finishing earlier on one processor can therefore enable a task on another processor to fire earlier. Due to the finite capacity of the FIFOs

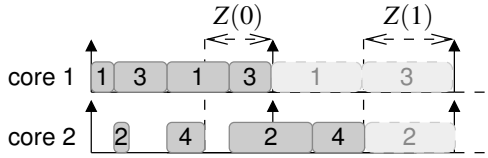


Figure 5.20: Frequency scaling per SOS iteration of the STS from Figure 5.19

connecting the tasks (see Section 2.1), an earlier finish by a producing/consuming task can enable the consuming/producing task earlier by releasing its data/space sooner.

Figure 5.21 illustrates this effect for our running example application from Figure 2.27b, where the inter-core FIFOs between tasks 1 and 2, and tasks 3 and 4 have a buffer capacity of one token. Both iterations of task 2 are stalled until task 1 completes and releases its output data. The third iteration of task 1 is stalled waiting on task 2 to release the space in the buffer so it can fire. A greater capacity buffer would allow task 1 to proceed, allowing the execution of the application on that core to proceed further without the need to stall for space. This leads us to Theorem 5.1.

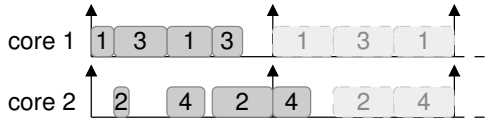


Figure 5.21: STS of HSDFG with finite inter-core FIFO capacities of one token.

Theorem 5.1. *Given an HSDF graph with bounded FIFO capacities, there is a worst case bound on how many iterations one task i may be ahead/behind another task j in the graph, given that the available data and space ($d(i,j)+d(j,i)$) on their connecting edges is known.*

Proof. A FIFO of finite capacity between a producer task i and a consumer task j can be represented in an HSDFG by edges (i,j) and (j,i) [107]. Initial data in the FIFO is represented by initial tokens $d(i,j)$ and the initial space in the FIFO by initial tokens $d(j,i)$, as illustrated in Figure 2.8b. The total buffer capacity of the FIFO is therefore the sum of initial data tokens and available space tokens, i.e. $d(i,j)+d(j,i)$.

Given Equation 2.1, the producer task i may be up to $d(i,j)$ iterations ahead of consumer task j, and similarly j may be up to $d(j,i)$ iterations ahead of producer task i. The combined CompSOC and application dataflow model is a fully connected graph, meaning that for every actor in the graph there is a path to every other actor. A task may be connected to another task via multiple routes through the graph. Due to the cumulative constraint of Equation 2.1 applied to edges along the path, the consumer cannot be further ahead than the shortest path (in terms of tokens) between the producer and the consumer and similarly the producer cannot be further ahead than the shortest path

between the consumer and the producer, i.e. the path between producer and consumer with least data/space constrains other paths between producer and consumer with more data/space. □

After an SOS has finished, while it is not necessarily possible to tell, using local information, exactly how many iterations of an SOS has completed on another core, it is possible to conservatively know how many SOS iterations the other core is maximally behind. This information can be translated into a conservative estimate of global slack.

Conservatively Estimating Global Slack

Having calculated the application's local slack Y_c (on core c) it is possible to conservatively derive a global slack quantity. We know from Theorem 5.1 that the FIFO buffer capacity constrains how far ahead one task can be ahead or behind a task that it communicates with. Due to the fully connected nature of CompSOC's combined application and platform HSDFGs there is a path between each task and every other task in the application. To find the most number of iterations B_c that the local SOS on core c can be ahead of an SOS on another core we take the maximum of the shortest paths between the SOS on core c and the SOSs on all other cores. The shortest path between an SOS on core c and an SOS on another core is the shortest path (in terms of tokens) between any task on core c and any task on the other core.

After an SOS has completed n iterations, all of the tasks belonging to that SOS have completed n iterations. Using B_c we know that all tasks on the path have at least completed $n - B_c$ iterations. This means that all of the application SOSs have at least started the $(n-B)$th iteration. Assuming all other SOSs still have to complete $B_c + 1$ SOS iterations before they have also completed n iterations is therefore a conservative assumption. Using the application's worst-case period T, we can conservatively estimate the global slack Z_c from the observed local slack Y_c as follows:

$$Z_c = \frac{Y_c - (B_c + 1) \cdot T}{N} \tag{5.2}$$

where N is the number of application graph iterations between power management invocations. As the frequency of other cores is not known locally, the application's worst case period T must bound the possibility that other cores are executing at the lowest available frequency level $f_{\min}$. T is derived off-line using a convex program of the combined application and platform HSDFG to minimise T while the frequencies of all of the cores are set to $f_{\min}$.

Equation 5.2 therefore ensures that each core locally calculates a global slack value Z_c that is less than or equal to the actual global slack value.

Conservatively Estimating Global Slack in a GALS System

It is not always possible to guarantee the synchronised start time of the SOSs on multiple cores, e.g. in a GALS system. Temporally bounded variations V in the start times are conservatively taken into account by assuming that all the cores started at the latest time allowed by the bound. For systems with this variation, it is taken into account in the conservative slack calculation Z as follows:

$$Z_c = \frac{Y_c - (B_c + 1) \cdot T - V}{N} \tag{5.3}$$

How to derive the bound V on the variation in starting time is beyond the scope of this thesis. Further reading on bounded clock variation in GALS systems can be found in [26, 84]

5.2.2 Globally Conservative DVFS

Once slack has been observed, it is then possible to use this slack to reduce the operating frequency thereby reducing power consumption. Care must be taken that any reduction in operating frequency does not violate the applications real-time requirements. In this section, we present how conservative run-time DVFS can be achieved, using mapping-specific off-line derived tables that specify how much observed slack is required to conservatively use a particular frequency. We explain how these tables are derived and how they are applied at run time.

Full Search

Using the combined application and CompSOC platform dataflow graph a per core frequency lookup table is derived with appropriate per-core frequency-levels for the amount of observed slack. For an application that is mapped onto a relatively small number of cores with relatively few frequency-levels, it is possible to calculate the minimum application SPS period and power consumption for each frequency combination, an example of which is presented in Figure 4.3. The range of pareto optimal frequency levels for power consumption to SPS period are selected for use at run-time. The pareto front of Figure 4.3 is presented in Figure 4.5. From this range of selected points, a table with the minimum required SPS period for each frequency-level is created. By subtracting the minimum period of the graph from all of the graph periods, the amount of slack necessary to conservatively execute a single iteration of the graph for the particular frequency combination is calculated. An example the resultant frequency-slack table is presented in Table 5.4.

Frequency (MHz)	Required Slack (s)
120	0
90	0.01
60	0.03
30	0.06

Table 5.4: Example frequency-slack table.

Sampling Using Convex Analysis

The number of points in the full search solution space grows exponentially with the number of cores that the application is mapped onto. For a platform with C cores and F possible frequency-levels, F^C points are calculated. For relatively large designs that are mapped onto many cores, this approach will eventually become infeasible as the processing of all the points will take too long.

The number of calculated points can be reduced by formulating the combined application and CompSOC platform model as a convex optimisation problem, as described in Section 4.1, and solving it for a range of SPS period constraints. An example of this technique is presented in Figure 4.6. The derived frequencies are optimal for the given minimisation objective, but are in the $\mathbb{R}$ domain and must be rounded up to the nearest available discrete frequency-level. After rounding, the frequency values are no longer optimal for the minimising objective. Given that the un-rounded frequency values were derived for the application's SPS schedule, and hence its worst-case performance, the application might not execute at its worst-case or even close to its worst-case for most of its execution, and therefore the optimal frequency values are unlikely to be continuously optimal during the application's execution anyway. Our run-time power management technique is slack conserving, meaning that any observed application-level slack that is not used is still available for use later.

A frequency lookup table is created per core from the sampled SPS period and frequency data. As before, the table consists of the minimum SPS period that each frequency-level conservatively supports. The accuracy of the table depends on the number and distribution of the sampled SPS periods in the range, e.g. for any sampling distribution, adding additional sampling points (the original points remain where they are) can only increase the accuracy of the result.

Selecting a Conservative Frequency Level

The period of an SPS can be shorter than its schedule length, i.e. the time before the last actor in the first SPS iteration finishes firing can be longer than its period. We refer to the length of the SPS as the application graph's latency. If the frequencies are statically set, then the graph's latency is observed once at the start of the execution with actors firing with period of the SPS thereafter. If DVFS is performed, the SPS becomes longer

in duration for lower frequencies and shorter in duration for higher frequencies. When lowering the frequency of an SPS a latency may be observed for the first iteration at the lower frequency. This is because the worst-case latency of the graph at the lower frequency is greater than the worst-case latency of the graph at the higher frequency.

The worst-case latency of the graph can also be derived using a minimising convex program in combination with the CompSOC platform's combined application and dataflow graph. This is achieved using a DCP, as presented in Code 5.4. The minimum worst-case latency L_{min} occurs when all the cores execute at the maximum frequency L_{max} and the maximum worst-case latency occurs when all the cores execute at minimum frequency. L_{min} is therefore derived using Code 5.4 by setting all the core frequencies `f` equal to the maximum frequency, and similarly L_{max} is derived by setting all the core frequencies `f` to the minimum frequency.

```
cvx_begin
  variable s(NUM_ACTORS) % array of actor start times
  variable T             % SPS period

  minimise(sum(s+t))

  s          >= 0

  % per edge token constraints
  s(s1)   + T*0 >= s(L1) + t(L1)*inv_pos(fplatform) % edge 1
  s(e1)   + T*0 >= s(s1) + t(s1)*inv_pos(f(1))      % edge 2
  s(e2)   + T*0 >= s(s2) + t(s2)*inv_pos(f(2))      % edge 3
  s(d1_2) + T*0 >= s(w1) + t(w1)*inv_pos(f(1))      % edge 4
  s(s1)   + T*1 >= s(w3) + t(w3)*inv_pos(f(1))      % edge 5
  ... % omitted per edge token constraints
cvx_end
```

Code 5.4: Part of the maximum schedule latency derivation DCP for Figure 4.2.

In order to transition to a lower frequency, the amount of slack that is observed must be enough to bound the increased latency of the graph at the lower frequency. To remain at the same frequency, or transition to a higher frequency, the amount of slack observed only needs to bound the period of the SPS at that frequency level. The frequency tables that are used for scaling therefore have two slack values; one for transitioning to a frequency when a higher frequency is currently in use, and the other for transitioning to a frequency when the current frequency is less than or equal to it.

This frequency-slack table can be calculated from the previously presented frequency-slack table (Table 5.4) during initialisation. For runtime power management that takes place every N application graph iterations, the resultant slack table that is used at run-time is derived as presented in Table 5.5, where S is the slack value from the original table, and

f is the current frequency when performing table lookup.

Frequency (MHz)	Required Slack (s) ($f_{\text{new}} \geq f$)	Required Slack (s) ($f_{\text{new}} < f$)
f_{new}	$S \times N$	$S \times N + L_{\max} - L_{\min}$

Table 5.5: Run-time frequency-slack table derivation using Table 5.4 as input.

Using this table, it is possible to perform run-time DVFS that is guaranteed to be temporally conservative.

5.3 Distributed Power Management Applied in Practice

Having explained how our technique works in theory, we proceed by demonstrating our technique applied in practice. To do this, we use the same running example application, platform and configuration from Section 4.3. Using our run-time power management technique, the frequency of the application's processors are scaled to meet the application's real-time requirement, while lowering the application's power consumption.

The advantage of using a dynamic power management technique, is that it uses both static slack and slack generated by dynamic timing variations, to reduce power consumption. While our static power management technique from Chapter 4 selects frequency levels to minimise power consumption, due to the availability of a limited set of discrete frequency levels, the derived frequency levels might not consume all of the available static slack, even in the worst-case. To demonstrate this, we use our running example application that has constant worst-case task execution times. Using our static power management technique from Chapter 4, we derive frequencies that provide the lowest power consumption, while ensuring that the application still meets a throughput of 1000 graph iterations per second. Figure 5.22 shows the graph finishing times from using static VFS with the derived frequencies, in comparison with executing the graph at maximum frequency and the application's throughput requirement. Even though our static power management method scales the throughput of the application while always meeting its throughput requirement, due to the finite set of available discrete frequency levels, there is still some static slack available that the application can use to perform DVFS.

Unlike our static power management technique, described in Chapter 4, our dynamic technique is able to consume slack that occurs dynamically at run-time. This ability comes at a cost, as the power management control loop is executed at run-time, increasing the amount of computation that must be performed within the application's timing requirements. Our technique also runs in a distributed manner, which means that the control loop is executed on each of the application's processors. Our power management implementation (which is not necessarily the most timing efficient implementation) for the MicroBlaze processor costs approximately 5000 cycles of computation per invocation. Depending on the timing of the graph, it might not make sense to invoke the power

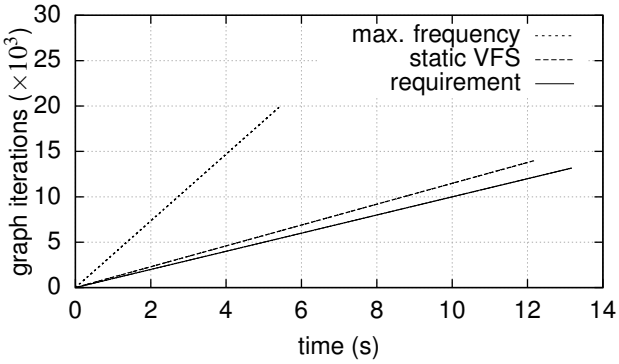


Figure 5.22: Graph iteration finishing times.

management function every graph iteration. Invoking the run-time power management after every so many graph iterations, allows the cost of the power management control to be amortised over the number of iterations.

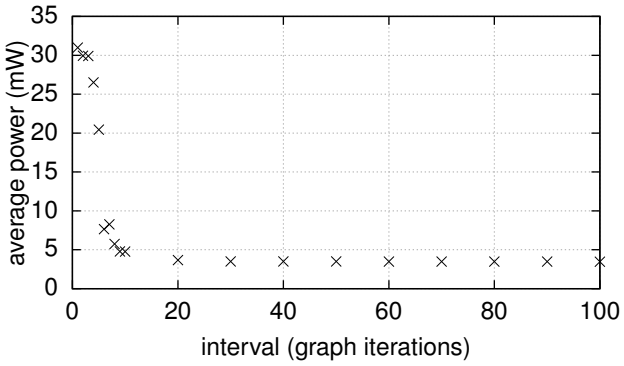


Figure 5.23: Average power consumption with variable power management interval.

Using our power minimising convex program of the combined application and platform HSDFG, we generate a conservative frequency-slack table to be used at run-time. Figure 5.23 presents the average application power consumption, for performing power management at various application graph intervals. At the run-time power management interval of one graph iteration the application's average power consumption is 31 mW, which compares quite poorly to the 3.64 mW achieved by our static power management technique for the same application. The high power consumption is in part caused by the computational cost of performing run-time power management, and in part caused by the pessimistic conservative assumption of the progress of the application on other

cores, as described in Section 5.2.1. As the number of graph iterations between power manager invocations increases, the average power consumption (in general) decreases, until it eventually plateaus at approximately 3.47 mW. The power consumption is only marginally better than the 3.64 mW achieved by the static technique. This is because the only dynamic variation in execution time is due to the invocation of the power manager and communication via the NoC, and also because the computation required by the power manager is a relatively big amount, in comparison to the application.

Many applications (like the H.263 decoder in Chapter 6) have dynamic variations in task execution time. We demonstrate the effects of a shorter than worst-case task execution time, for the running example application, by changing the task execution times of the tasks on each core. Figure 5.24 presents graph iteration finishing times whenever all of the tasks constantly execute at 50% of their worst-case work than was used for the off-line analysis. The static VFS technique is unable to take this reduction of execution time into account and therefore completes graph iterations at a higher rate than when the tasks execute at 100% of their worst-case work. This can be seen when the static VFS finishing times in Figure 5.24 are compared with those in Figure 5.22.

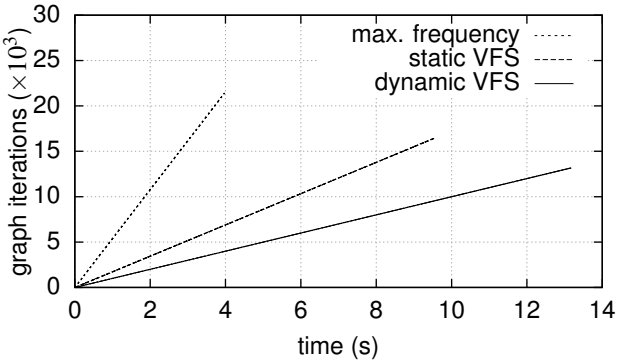


Figure 5.24: Graph iteration finishing times using 50% task worst-case work.

Our run-time power management technique observes application level slack (on the granularity of application graph iterations) and uses this as the input of the power management control loop. Figure 5.25 shows the slack observed by the power manager for a power management invocation every 100 graph iterations. The observed slack settles after a number of graph iterations into a stable but fluctuating quantity of between approximately 165-190 ms. The power manager performs a lookup of the frequency-slack table to find the frequency that provides the lowest power while being guaranteed to meet the application's throughput requirement. Performing the power management every 100 graph iterations means that the application must have enough slack for the chosen frequency to temporally bound the eventuality that the 100 graph iterations all execute at their worst-case timing. Figure 5.26 shows that the amount of slack required when

performing power management every 20 graph iterations is much less. Approximately fluctuating between 32-43 ms. If it is necessary to buffer the output of the graph until its deadline, then performing power management at smaller intervals is desirable to minimise the required buffering. Figure 5.23 shows that smaller power management intervals consume more power on average. The power management interval is therefore a trade-off between average power consumption and output buffer capacity. As this thesis is primarily concerned with power consumption, we leave any further analysis of this trade-off as future work.

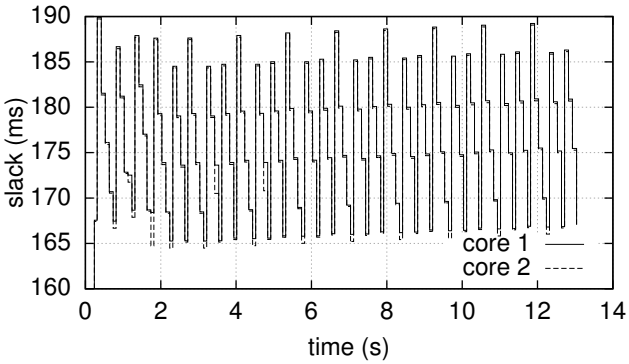


Figure 5.25: Observed slack using 50% task worst-case work and DVFS every 100 graph iterations.

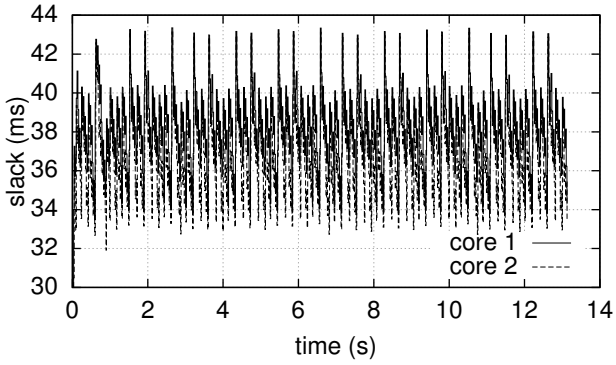


Figure 5.26: Observed slack using 50% task worst-case work and DVFS every 20 graph iterations.

The fluctuations in the application's slack levels are due to the limited set of discrete frequency levels. The power management oscillates between discrete frequency levels,

effectively interpolating the frequencies, which can be seen in Figure 5.27. The oscillating/interpolation occurs naturally as a result of the control loop without being explicitly induced by the power manager. This also translates into an oscillating energy consumption per graph iteration, as presented in Figure 5.28. The fluctuations present in this graph are mostly due to the frequency levels presented in Figure 5.28, but also the variations in the time taken for a single graph iteration due to the variable time taken for data to cross the NoC. Our run-time power management technique reduces the average power consumption to 2.51 mW when the application tasks execute at 50% of their worst-case work, which is a significant decrease when compared with our static power management technique's 3.64 mW.

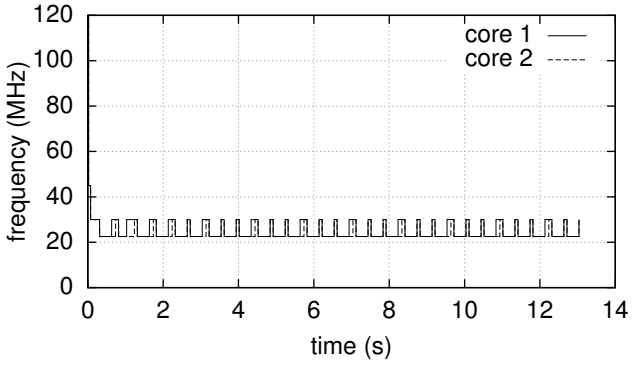


Figure 5.27: Core frequencies using 50% task worst-case work with DVFS.

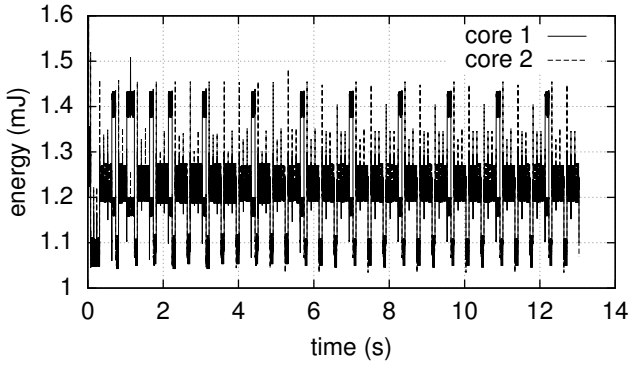


Figure 5.28: Graph iteration finishing times using 50% task worst-case work.

We have shown that our distributed run-time power management technique is able to use slack that is observable at run-time to decrease the application's average power

consumption without missing any deadlines. The results presented in this section show how our techniques work for a simple application that has tasks with constant execution times. In Chapter 6 we demonstrate our static and dynamic techniques applied to an H.263 decoder application that has data dependent task execution times.

5.4 Related Work

Much work has been carried out on the topic of power management using DVFS over approximately the last two decades [20, 37]. In this thesis, we make a distinction that is not always made in literature between VFS where the voltage and frequency are set to as static level and is a design time power management decision and DVFS where the voltage and frequency can be dynamically changed at run-time. In some related work, Dynamic Power Management [24] (DPM, this abbreviation does not appear in the glossary of this thesis to avoid confusion) is considered as separate to DVFS for gating the voltage and frequency to a part of the system for a duration of time. Whether the physical mechanism is separate to the hardware DVFS controller or not, we consider turning off portions of a circuit (processor) to be logically just another level in the DVFS mechanism where the frequency is zero and the voltage is the level of the particular power down state. In this thesis, the term dynamic power management is used to distinguish dynamic run-time power management (Chapter 5) from static off-line power management (Chapter 4). In this section, we focus on related work that help to contextualise the power management techniques presented in Chapters 4 and 5.

DVFS is a commonly used mechanism in many smart phones, tablets and laptops. Using a notion of load [15] (the specific definition of load can vary), the processor's performance is increased to meet high load demands and reduced to as the load dissipates. While this technique is sufficient in many instances to provide a subjectively fluid user experience, it is unsuitable for use with real-time applications that require firm or hard timing guarantees. The difficulties faced in performing DVFS with real-time applications was also acknowledged almost two decades ago in [112]. Real-time applications require timing guarantees with formal mathematical timing abstractions used to verify that they are met. Static off-line and dynamic on-line algorithms are proposed in [112] to find an energy optimal schedule for a set of independent real-time tasks (jobs) with deadlines. The algorithms are applied to a single core, with the derived schedule specifying the task that should execute and the DVFS level that it should execute at, ensuring that the tasks meet their deadlines.

An overview of different real-time power management techniques can be found in [7,47]. With the techniques from [112] still featuring highly. A notable omission from [47] is the acknowledgement that the work in [112] only applies to single core architectures. With the rise of multiprocessor architectures, by the time that [7] was published, the authors highlighted multiple [8, 59, 60, 89] multiprocessor power management techniques.

More recently, a multi-core power management was proposed in [29] for tasks with

arbitrary arrival times, that extended their single core technique from [30], by translating the multi-core problem into a single core problem using a "parallelism" factor and deriving a single global DVFS level for low energy consumption.

5.4.1 Power Management for Graph Based Applications

The most relevant multiprocessor power management technique highlighted in [7] to the work in this thesis is [89], as it considers precedence constraints between the tasks on the multiprocessor. These precedence constraints form a group of tasks (application) into a graph topology.

Graph structured applications have (data) dependencies between tasks. These dependencies restrict the ordering of the tasks and hence also the possible schedules [89]. In [24, 25], an off-line power management technique is proposed to reduce the energy consumption of periodically scheduled KPN applications. Their techniques use heuristics to achieve a static schedule. Another heuristic is then used to decide on a trade-off point between static frequency levels that all the cores will use and the amount of time that can be used to power down the cores.

Off-line and on-line power management techniques are proposed in [23] for dynamic dataflow applications modelled using Finite State Machine Scenario Aware Dataflow (FSM-SADF). Dynamism in the application is captured as a set of SDF scenarios (the Scenario Aware Dataflow (SADF) part) with each of these scenarios representing a state in the Finite State Machine (FSM). A combined FSM-SADF analysis is used to identify critical cycles in the scenarios and hence the application's throughput for different DVFS levels. The heuristic then prunes this space to find frequencies per application scenario that provides the lowest energy consumption. The technique lacks implementation details, such as how the actors execute as an FSM-SADF at run-time, and only the analysis technique was implemented. Without implementation details, the analysis is therefore purely theoretical. This is a common occurrence with power management publications. One of the claims in [23] is that static dataflow methods are not suitable to model the dynamic behaviour of applications such as an H.263 video decoder, rendering them unsuitable for use in conservative run-time DVFS techniques. We demonstrate that this assumption is incorrect in Chapter 6 by applying our techniques that use static dataflow analysis to an H.263 decoder application. Our technique uses dynamic variations in the application's execution time to perform power management while meeting the application's timing requirement.

5.4.2 Distributed Queue Occupancy Power Management

Queue occupancy power management techniques use the occupancy of task output queues as an application progress indicator. This information is then used to perform DVFS. These techniques commonly claim applicability to GALS systems where queues are used between clock domains.

A distributed queue occupancy power management technique is presented [54]. Each tile uses a formula to predict future queue occupancy and explicitly communicates this information to each other core. Using the collective information from the other processors, each processor derives a locally applicable DVFS level. An interesting observation in [54] is that local DVFS using only locally available information can lead to the unbalanced situation where one core scales to a low DVFS level based on local observations, but this local scaling has a global impact and affects the local observations made by other cores. As a consequence, while one core scales to a low DVFS level, other cores have to maintain a high DVFS level to meet timing requirements. This is used as a justification for their explicitly communicated co-ordination approach.

A run-time closed control-loop power-manager is applied to dataflow applications in [9]. The applications that the technique is applied to must comply with a particular structure, with a single identifiable producer and consumer task. Buffer occupancy is used as the control loop's feedback mechanism. Unlike [54], there is no explicitly communicated global progress information. Instead, each tile tries to maintain a static occupancy in its output buffer. While the exact levels of these static occupancies are important to prevent the sort of unbalanced DVFS described in [54], no technique is described in [9] on how to find these levels.

5.4.3 Implementation for DVFS Verification

Many real-time DVFS publications focus on the mathematical nature of power or energy minimisation, but neglect to verify if their technique is implementable, not just as a model, but in practice. This is not a trivial step, and can reveal that assumptions made while modelling do not hold in reality. The power management publications [85–87] are exceptions, taking a more implementation driven approach.

Using the physical hardware platform described in [12], the energy priority scheduling technique in [86] implements an enhancement of the theoretical technique from [112]. The power consumption results they present are physical measurements from their hardware platform. While we acknowledge that this is not always possible for practical reasons such as time or cost, it is important that power management techniques are more than just a theoretical exercise.

5.4.4 This Work in Context

The work in Chapters 4 and 5 of this thesis describe a distributed multiprocessor static and dynamic power management techniques for (virtualised) dataflow applications that can be analysed as HSDF graphs. Our techniques do not perform task scheduling or mapping. Given a static schedule and mapping, our static power management technique uses a convex program to derive temporally conservative low power static per core DVFS levels. Our technique can also be used to derive a single global DVFS level if that is required.

Our dynamic power management technique has much in common with buffer occupancy power management techniques. Actors representing tasks cannot fire unless there is sufficient data in incoming buffers and space in outgoing buffers. The progress of dataflow applications is therefore intrinsically linked to buffer occupancy. We extend our design time convex programming technique to derive a frequency-slack lookup table per core, that enables a temporally conservative distributed DVFS decisions to be made. Our technique does not explicitly communicate progress information between processor tiles, but uses the fully connected nature of the combined application and CompSOC platform HSDFG to conservatively infer global application progress from local application progress. The combination of our design time derived frequency-slack lookup table and distributed conservative global progress observation ensures that our technique does not exhibit the unbalanced multi-core power management described in [54].

We implement our techniques on an FPGA prototype of the CompSOC platform. This ensures that our techniques are actually implementable in practice and that real-time requirements are met. We acknowledge that a limitation of our FPGA prototyping approach is that it is not possible to simply measure power and energy consumption, as what would be measured relates to the FPGA implementation of the prototype and not the prototyped platform itself. Ideally, we would implement an instance of our CompSOC multi-core platform in silicon and perform power and energy measurements, but time, financial budgets and current research objectives have not made this a reality so far.

5.5 Summary

In this chapter, we have presented run-time DVFS techniques enabling dynamic variations in the application's task execution times to be used to lower the voltage and frequency-level of the processor, reducing power consumption. Using an adaptive H.263 application as an example, we describe how output quality can be used as an accompanying scaling mechanism to DVFS. We show how the quality-level of the application can be regulated by a separate quality-manager, reducing the execution time of adaptive tasks. This reduction in execution time can be used to achieve real-time requirements, or to reduce power consumption when applied with a power-manager that scales the frequency-level based on the amount of observed slack.

We follow this by describing how applications that are mapped across multiple processors are conservatively scaled using limited application graph topology information at run-time, i.e. the number of cores on which the application is mapped, the maximum buffer capacity of the graph in terms of tokens, and using only locally observed progress information. If the period of the SPS of the application graph meets the throughput requirement of the application, the scaling is performed conservatively. This can be verified off-line using the static analysis techniques described in Chapter 4.

Using the combined application and CompSOC platform dataflow graph, we further describe two methods how application and mapping specific per core frequency lookup

tables can be derived off-line. One presented technique is to perform a full search of the solution space and select pareto optimal points to create the table. The other technique uses the convex optimisation technique from Section 4.1 to derive optimal core frequencies for a range of application SPS periods.

CHAPTER 6

Case Study

It is intended that our techniques are applicable not just in theory, but also in practice. In this chapter, we apply our power management techniques to an H.263 decoder application running on an FPGA prototype of a CompSOC platform instance. A photo of a similar experimental setup to the one used in this chapter is presented in Figure 6.1. Unlike the running example application Figure 2.27a that is used throughout this thesis, the H.263 decoder has data-dependent task execution times, i.e. some video frames take longer to decode than others. For instance, I-frames take significantly longer to process than P-frames. Our power management technique is not application specific and our run-time power management is implemented in a simple look-up table manner, with no run-time learning capability. While this keeps the computational cost of our technique low, it cannot anticipate sudden workload increases, even if they occur at predictable intervals, yet our technique adequately deals with this scenario. We choose the H.263 decoder to demonstrate our power management techniques as it is a soft real-time application, with an easy to comprehend timing requirement in frames per second. Throughout this chapter, we treat the H.263 decoder as if it has a firm real-time requirement. Using the H.263 decoder, we show that our technique can reduce power consumption of a real application with data dependent execution times while firmly meeting the application's real-time requirements.

We guarantee that the H.263 decoder applications will meet its deadlines regardless of concurrently executing applications by executing the application on CoMik's virtual

processors. The H.263 is therefore able to perform power management without its timing being affected by, or affecting the timing of, concurrent applications. In the following section we demonstrate CoMik's composable virtualisation that enables each virtual processor to execute code that is cycle-accurately isolated from code belonging to other applications on other virtual processors. In Section 6.3 we proceed to demonstrate our power management techniques applied to multiple mappings of an H.263 decoder application, running on a CoMik virtualised multi-core platform.

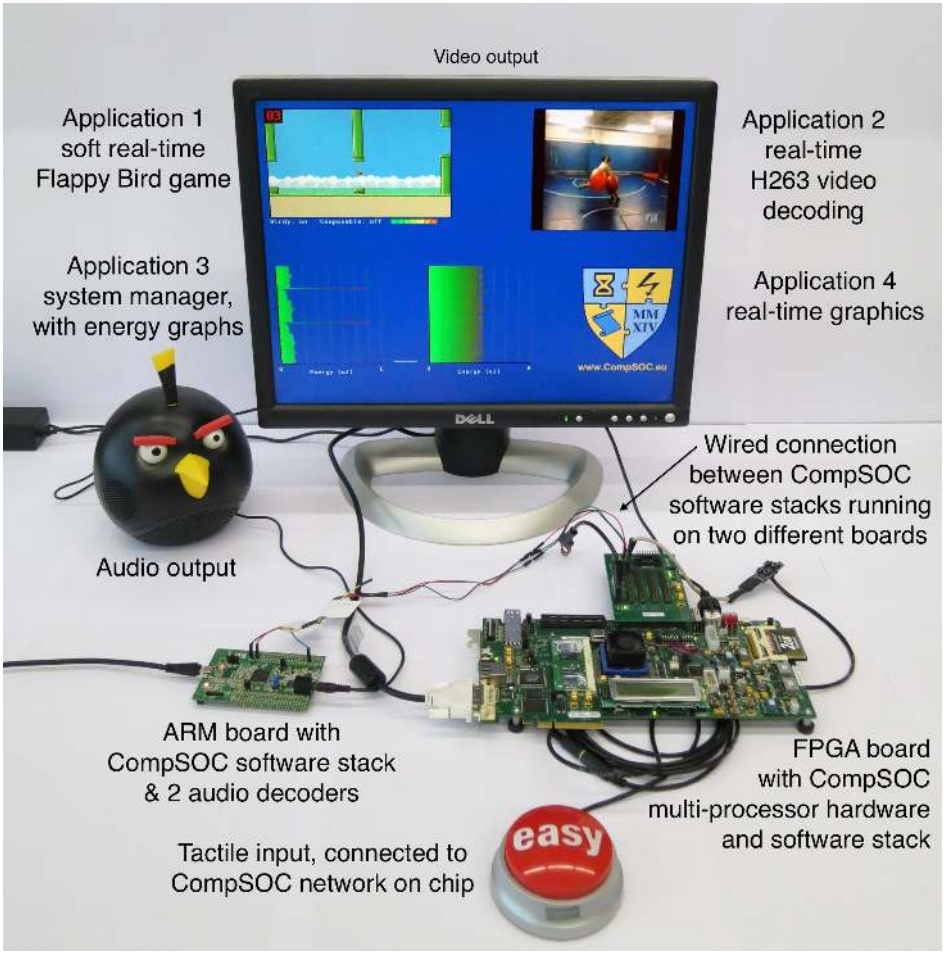


Figure 6.1: CompSOC platform demonstrator from our DATE 2014 university booth.

6.1 CoMik's Composable Virtualisation in Action

We continue by experimentally demonstrating CoMik's cycle-accurate temporal isolation and predictability of its virtual processors. We demonstrate this for an FPGA prototyped instance of the CompSOC hardware platform, as described in Section 2.2. The clock frequency is set to have an upper bound of 120 MHz. We configure CoMik to use a CoMik slot duration of 4096 cycles and a virtual processor slot duration of 65536 cycles, making a virtual processor TDM scheduling slot 69632 cycles.

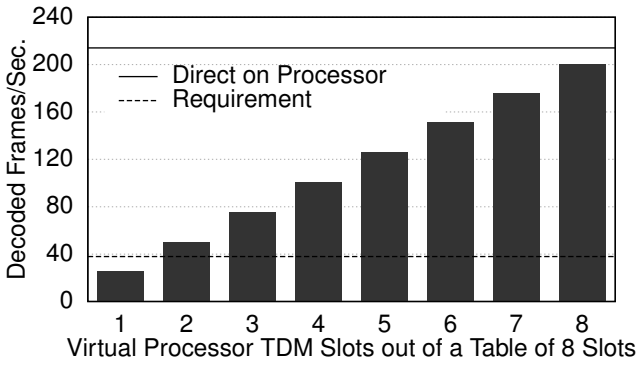


Figure 6.2: MP3 decoder mapped onto a single virtual processor that has its performance scaled using its TDM slot allocation.

To be suitable to run real-time applications, CoMik's virtual processors are not only composable, but predictable also. Figure 6.2 presents the timings of an MP3 decoder (not modelled as a dataflow application) executing directly on the physical processor and as a guaranteed partition on a virtual processor that is configured to have between one and eight slots in an eight slot TDM table. From Figure 6.2 it can be seen that there is a predictable linear relationship between the number of slots allocated to the virtual processor and the MP3 frame decoding rate. Due to the CoMik slot, the virtual processor that uses all eight slots does not decode the MP3 frames as quickly as the physical processor directly. A virtual processor with a minimum of two slots out of eight is sufficient to meet the MP3 decoder's requirement of 38 decoded frames per second, allowing the remaining slots to be used for other partitions.

We continue by demonstrating the timing of concurrently executing applications. We do this by executing a soft real-time MP3 decoder as a best-effort partition and a firm real-time application that generates "ticks" in response to periodic interrupts that are virtualised for that application, as a guaranteed partition. The tick generating application clock gates between ticks, leaving any full slots between ticks unused. Each application

Section 6.1 contains experimentation from publication [79]

is allocated a single virtual processor on the same physical processor with each virtual processor allocated a single slot in a two slot TDM table.

Figure 6.3 presents the resultant timing of the two applications. The tick is produced at regular intervals, except when the time of the tick does not occur when the tick application is scheduled. In this instance the tick is produced in the partition's next scheduled virtual processor slot. The TDM slots that the tick partition leaves unused due to clock gating, are given to the best-effort MP3 decoder partition. The MP3 decoder therefore finishes decoding its frame earlier, enabling its power management scheme to temporarily clock gate the processor while still meeting its throughput requirement.

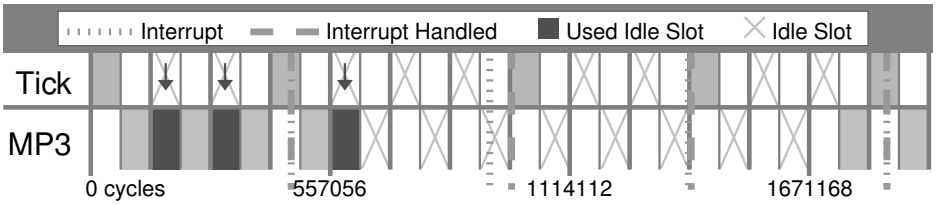


Figure 6.3: A best-effort MP3 decoder partition that is cycle-accurately isolated from a guaranteed tick application responding to PIT interrupt "ticks".

We continue our experimentation by demonstrating that the timing of the tick partition that is guaranteed to be cycle-accurately isolated, does not change by a single cycle whenever another partition is added to the system. To achieve this, we add an additional best-effort partition to be executed concurrently. Its virtual processor is not allocated a TDM slot, so it can only make use of unused slots.

Figure 6.4 presents the difference between the timings of the original partitions with the added best-effort partition, and the timings that were used for Figure 6.3. The MP3 decoder partition shows timing variation between runs, as it must share the unused slots with the added best-effort partition. The tick application has exactly the same timing with or without the additional best-effort partition, demonstrating that it is cycle-accurately composable.

We have demonstrated that CoMik cycle accurately isolates applications executing on dedicated virtual processors. Applications can choose to give up this isolation in order to receive otherwise unused slots from other applications. In the rest of this chapter, we focus on applications executing on CoMik's virtual processors that have guaranteed slot allocations.

6.2 CompSOC HSDF Model Evaluation

Having described in the previous sections how real-time dataflow streaming applications that are mapped onto a CompSOC platform are formalised as an HSDFG for timing analysis, we proceed to demonstrate the accuracy of our modelling technique. To do this,

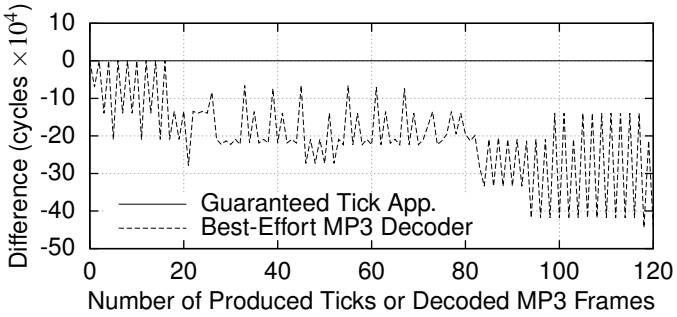


Figure 6.4: Timing difference when run with and without an additional Best-Effort Application.

we execute applications on a FPGA prototyped four core CompSOC platform and compare their actual timings with those predicted by timing analysis of the application's associated HSDFG. For this purpose, we use both the example application from Figure 2.27a (which we will henceforth refer to as the synthetic application), and an H.263 decoder application with which we decode various video streams. In this section, we will show:

- Our technique applied to both a synthetic and H.263 decoder application.
- The tightness of our technique using applications with tasks that constantly execute with WCETs.
- That our technique conservatively bounds the timings of GALS systems, ranging from 100% synchronous Fully Aligned (FA) clock and symmetric TDM table, to systems where the alignment of TDM tables is unknown and all inter-core communications are conservatively assumed to arrive with Worst-Case Arrival (WCA).
- That the tightness of the bounds predicted by our technique depends on the amount of knowledge of the system, i.e. the alignment of the TDM tables.
- That our technique can correctly predict trends, e.g. whether an improvement in throughput is expected when changing mapping.

The CompSOC platform that we use for our experiments has four homogeneous processor tiles executing at 120 MHz, with local instruction, data, communication and DMA memories. Each tile has multiple DMAs and each application is allocated a single DMA per tile.

6.2.1 Synthetic Application

We start our experimentation by comparing the actual graph iteration finishing times of the synthetic application, as measured on the FPGA prototype of the CompSOC platform, with the predicted latency and throughput of its HSDF model from Figure 2.27e. The synthetic application is structured following Figure 2.27a with each actor representing a

task and each edge a C-HEAP FIFO between the tasks. Each FIFO has a buffer capacity of two tokens. For the purposes of ascertaining the accuracy of the model for worst-case analysis, each task has a constant execution time and therefore always executes at its worst-case. Each CoMik instance is configured to have a TDM table length of ten slots with five slots allocated to the synthetic application's VP. Each TDM slot comprises of a CoMik slot and a partition slot, with durations of 4,096 and 65,536 cycles, respectively.

The finishing times of the application's graph iterations, as measured from the FPGA instance of the CompSOC platform, are presented in Figure 6.5. The synthetic application can be seen to make progress (complete graph iterations) whenever its VP is scheduled for five out of the ten iterations of the TDM table, creating the impression of "steps". The CoMik overhead between its five allocated slots also prevents application progress, but is so small that it is unnoticeable in the graph.

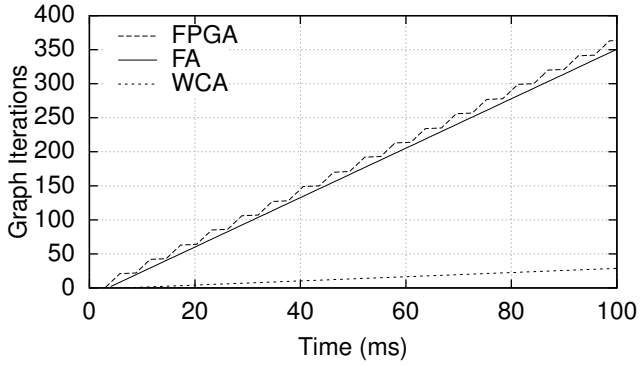


Figure 6.5: Synthetic application graph iteration finishing times.

The predicted timings of the application's HSDF model are also presented in Figure 6.5, and the predicted latency and rate can be seen to conservatively bound the synthetic application's actual finishing times. The effect on timing performance of the VPs' TDM allocation is captured by a latency rate abstraction, as described in Section 2.3. The rate of execution of each VP is calculated as 8 virtual cycles per 17 cycles of the physical processor. The timings of actors $s_{\{1,2,3,4\}}$, $e_{\{1,2,3,4\}}$, $w_{\{1,2,3,4\}}$, from Figure 2.27e, are modified to reflect the virtual rates of execution.

The VP's rate of execution is sustainable after a latency of 368639 cycles, as derived by Equation 2.10 in Section 2.3. In Section 2.3.7, we described how knowledge about the alignment and dimensioning of the TDM table is used to reduce pessimism in the model. In the most pessimistic case, it is assumed that data communicated between cores always arrives at the WCA time, i.e. that the data arrives just too late to be noticed and has to wait the worst-case amount of time before the application is scheduled again. In this case, the latencies L_1 and L_2 from Figure 2.27e are annotated with the latency of the CoMik TDM table for that tile. Alternatively, if the TDM tables are

symmetrically dimensioned and FA (100% synchronous), the latency of the VP's latency rate abstraction is conservatively taken into account as a single offset before the rate of the VP is sustainable, as described in Section 2.3.7, with zero delay being assigned to L_1 and L_2. The results presented in Figure 6.5 for the FPGA implemented CompSOC platform has cycle-accurate FA CoMik TDM tables on each core. The HSDF model's prediction achieves a conservative bound for both the WCA and FA cases. The FA method achieves a tighter bound (predicting throughput to within 1.65% of the actual case) than the WCA method (predicting throughput to within 91.68%). This is as expected because the tiles were fully aligned and not having or using this knowledge results in a pessimistic prediction. In some GALS systems, it may not be possible to bound the TDM table alignment, or it may be desirable to use a non-symmetric CoMik TDM configuration, and for these instances, the WCA method still gives a guaranteed conservative timing bound.

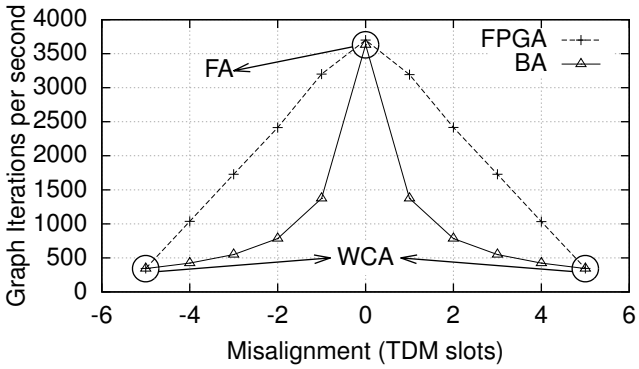


Figure 6.6: Synthetic application throughput for the range of TDM table alignments.

If it is possible to bound the alignment of the TDM tables, it is possible to get a less pessimistic conservative timing bound. Figure 6.6 presents the synthetic application's measured throughput, from the CompSOC FPGA implementation, for the complete range of TDM table alignments. It also presents the throughput predicted by the HSDF model when annotated with the TDM table's alignment bound. The Bounded Alignment (BA) annotation method uses a single VP latency offset minus the alignment bound, and annotates latencies L_1 and L_2 with the alignment bound. If the misalignment is zero, then the BA annotation matches the FA case, while if the misalignment is equal to the VP latency, then the BA annotation matches the WCA case. From Figure 6.6 it can be seen that the predicted throughput is quite tight (to within 1.86%) when there is no misalignment, but becomes less tight as the TDM tables are more misaligned. This happens because the model assumes that every iteration of the graph is affected by the misalignment. But since multiple graph iterations can finish within a single slot, as can be seen in Figure 6.5, this is not always the case in actuality, e.g. If the application slots that are scheduled concurrently multiple iterations of the graph can finish in this time

without being affected by the misalignment. As the tables become more unaligned, more iterations of the graph are affected by the misalignment and the accuracy of the model increases again. From this, it can be seen that the WCA annotation can be tightly accurate (to within 0.4% in Figure 6.6) when slot tables are completely unaligned in the actual case, while conservatively bounding all other alignments.

6.2.2 H.263 Decoder

We proceed to apply our technique to an H.263 decoder (without quality scaling) that has data dependent task execution times (see Section 5.1.1 for details). Figure 6.7 presents the frame finishing times from multiple decoding runs and the predictions from the FA and WCA modelling cases. Three different videos (akiyo, bus and tree [2]) are used as input for the H.263 decoder. To demonstrate the accuracy of our technique for worst-case analysis, the wcet run executes the H.263's tasks with the constant worst-case task timings measured from the runs of the three videos. The H.263 decoder is mapped onto all four cores of the FPGA prototyped CompSOC platform, with FA symmetric CoMik TDM tables. Each table has three slots, of which two are allocated to the H.263 decoder's VPs. Each TDM slot is configured to have a CoMik slot duration of 4,096 cycles and a partition slot duration of 196,608 cycles. From Figure 6.7, it can be seen that the FA and WCA annotated HSDF model predictions conservatively bound the three video runs and the wcet run, although it is hard to see from Figure 6.7 that the FA annotated model conservatively bounds the wcet runs timing because the FA prediction is so tight that the wcet and FA lines appear to overlap at this scale. As the three videos (akiyo, tree and bus) generally have better than worst-case task execution times, their achieved throughput is higher than for the wcet execution and therefore also the HSDF model predictions. This is a limitation of worst-case analysis in general and is not specific to our technique. What is important, is that our technique conservatively bounds the application's worst-case execution, and it achieves this.

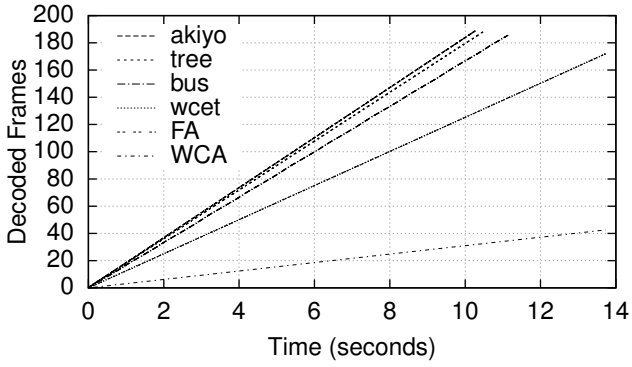


Figure 6.7: H.263 frame decoding times (wcet and FA overlap).

For our final experiment, we compare the accuracy of the H.263 decoder's HSDF model throughput prediction with the throughput achieved by the H.263 decoder executing with constant task WCET on the FPGA instance of the CompSOC platform (wcet), for the H.263 decoder mapped onto one to four cores. Apart from the application mapping the platform is configured the same as for the previous H.263 decoder experiment. From Figure 6.8, it can be seen that the FA annotated model that matches the platform's alignment achieves a tight conservative bound for the four mappings. The WCA also achieves a conservative bound, which is less tight but also bounds the application's throughput if the TDM table's alignment was unknown. While the WCA bound is conservative for all table alignments, Figure 6.8 shows that for a platform with a bounded and relatively small variation in TDM, the WCA bound becomes less accurate as the number of cores increases. This is due to the pessimistic assumption that every inter-core communication arrives at the worst-case time in the TDM table, and therefore incurs a Worst-Case Response Time (WCRT). Both the FA and WCA HSDF annotations correctly show that the two core mapping of the H.263 decoder does not offer an advantage for application throughput over the single core mapping. The FA predicted throughput also correctly shows that the three and four core mappings perform better for application throughput than the single core mapping, whereas the WCA predicts throughputs worse than the single core mapping for the three and four core mappings. Both FA and WCA correctly predicts an improvement in graph throughput moving from a two core to three core mapping, and from a three core to four core mapping.

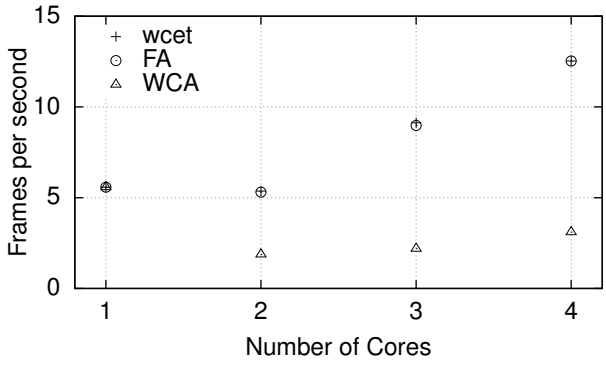


Figure 6.8: H.263 frame rate per mapping.

From our experimentation, we can conclude that our technique conservatively and accurately models application worst-case timings when mapped onto multiple cores of the CompSOC platform. The accuracy of our technique is better if the alignment of symmetrically dimensioned CoMik TDM tables on the cores can be bounded. Regardless of the TDM table's alignment or dimensioning, our technique is still able to produce conservative timing predictions for application graph latency and throughput.

6.3 Power Management of an H.263 Decoder

We have shown in previous sections that CoMik offers composable and predictable virtualisation of the CompSOC platform, that POSe provides predictable dataflow execution and that the application timing can be formally analysed using a combined application and HSDF. In this section, we proceed to demonstrate our power management technique applied to an H.263 decoder application that executes on our CompSOC FPGA prototype. Unlike our running example application, the H.263 decoder has data dependent task execution times. We demonstrate how this affects our power management technique by decoding three different video clips (bus, tree and akiyo), which each producing different timing behaviour from the decoder. We add a fourth video to this mix (wcet) that has computational tasks with constant execution times that are equal to the worst case times of the computational tasks of the three video clips. In addition to this, we also demonstrate the effect of mapping the H.263 over one to four cores on our power management techniques. We show that both our static and dynamic techniques are useful for power reduction, and that our dynamic technique is able to use dynamic variations in task execution time to reduce power consumption further than our static technique.

Mapping	VLD	IQ	IDCT	MC + FR	UP
1	1	1	1	1	1
2	1	1	1	2	1
3	1	1	3	2	3
4	1	1	3	2	4

Table 6.1: H.263 decoder processing core task mappings.

Table 6.1 describes each of the four mappings that we use during our experimentation. These mappings are generated by an automated tool that only ensures that the application's memory requirements are met. We do not claim that these mappings are in anyway optimal for low power execution or throughput performance. In doing so, we show that our analysis and power management techniques are able to analyse and reduce the power consumption of whatever mapping the designer wants to use.

The H.263 decoder has widely varying task execution times that not only depend on the contents of the video frame (i.e. whether there is a lot of motion), but also on the type of frame. As described in Section 5.1.1, H.263 video streams consist of I-frames and P-frames, with I-frames generally taking longer to decode than P-frames. This can be seen in Figure 6.9 for the single core mapping of the H.263 decoder when it decodes the tree video at maximum frequency (MAX). Every twelve frames there is a "spike" where an I-frame takes longer to decode. The H.263 decoder's tasks therefore execute at less than their worst case execution time for most iterations of the application graph. This dynamic variation in task execution time provides the dynamic slack that our run-time power management technique uses in addition to the application's static slack. The H.263 decoder requires 99 graph iterations to decode a single frame and we invoke the

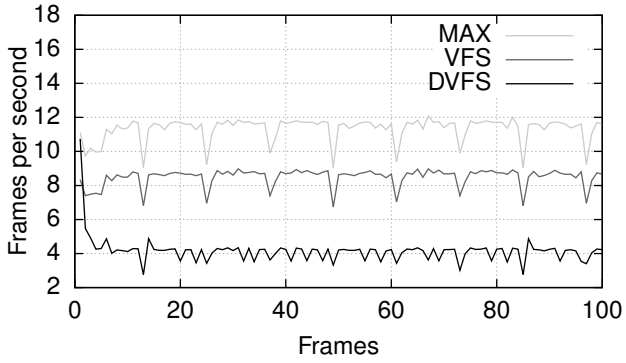


Figure 6.9: Per frame decoding rates for the tree video (1 core).

power manager once per frame. Performing conservative run-time power management on the H.263 decoder with its widely varying task execution times shows that our power management control loop able to deal with sudden changes in the amount of available slack without violating the application's timing requirement.

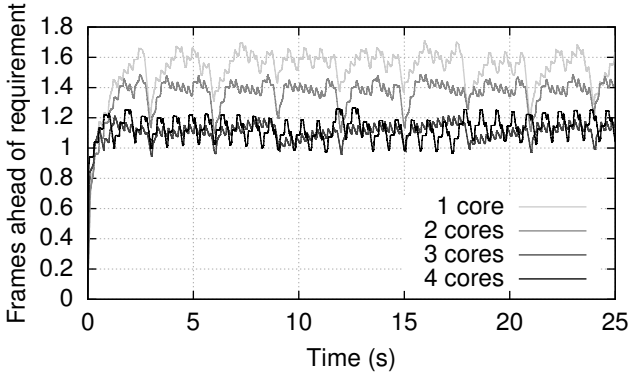


Figure 6.10: Frames in advance of requirement to guarantee conservative execution.

Our dynamic power management technique uses slack in the schedule to perform DVFS while meeting the applications requirement. Our technique ensures that enough slack is present to bound the application's worst case execution at the lower frequency before transitioning to that level. This means that the application is always running ahead of schedule. Depending on the nature of the application, it might be necessary to buffer this output until the required time. In Figure 6.9 the dynamic power management video maintains an average of four frames per second. Since some frames decode faster than necessary, some must decode slower to maintain the four frames per second average.

From Figure 6.10 we can see that our technique executes the application approximately 1-2 frames in advance to bound the possibility of worst case execution at lower frequency levels. This is a feasible cost as the frames can be buffered in the Dynamic Random Access Memory (DRAM). The amount in advance that the application needs to run depends on the graph's worst case execution and the frequency level used, e.g. in Figure 6.10, the 3 and 4 core examples have a shorter worst-case graph period than the 1 and 2 core examples and therefore require less buffering to conservatively run at the same frequency. We leave a detailed investigation on output buffering of applications using our technique as future work.

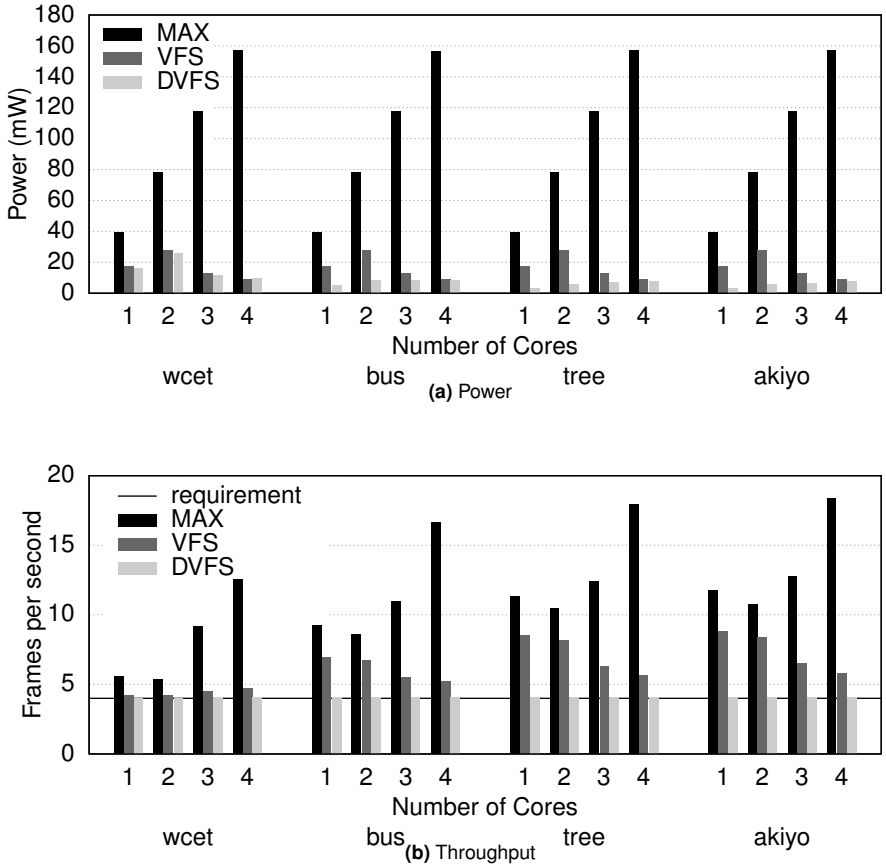


Figure 6.11: Reported values from the CompSOC FPGA prototype for the H.263 decoder application decoding four videos (*wcet, bus, tree, akiyo*) and mappings (*1-4 cores*).

Figure 6.11 presents the power and frame rate (throughput) of the H.263 decoder for all of the various mappings and videos, as resulting from execution on the CompSOC FPGA

prototype platform. The H.263 decoder is allocated two out of three virtual processor slots on each physical core, with a maximum frequency of 120 MHz, CoMik slots of 4096 cycles and virtual processor slots of 196608 cycles. A firm frame rate requirement of four frames per second was used to ensure that the H.263 decoder could meet the frame rate requirement at maximum frequency for all videos and mappings.

Figure 6.11a shows the average power consumed by each video for each of the four mappings. These power numbers only include the power consumption of the processors and not of the infrastructure required by a multi-core system, such as a NoC. Using more cores may require additional infrastructure at a cost of increased power consumption. In this investigation, we make the simplifying assumption that the platform is a four core system with the same infrastructure cost in terms of power for each mapping, which can therefore be removed from the power consumption comparison. The application only accounts for the energy that it consumes and therefore does not account for virtual processor slots that it does not use, or for time on cores to which it is not mapped.

When executing at maximum frequency (MAX) on all cores, the average power consumed is simply the number of cores multiplied by the power consumption of a single core. To apply our static power management (VFS) technique (as described in Chapter 4), we use a power minimising convex program of the combined application and platform HSDFG of the four mappings. We use the H.263 decoder's worst case task timings, to ensure that the analysis is temporally conservative for all of the decoded videos. This can be seen in Figure 6.11b where none of the videos decoded at a frame rate less than the required four frames per second. By using static per core frequency levels per mapping, our static power management technique lowers the power consumption by the same amount for each mapping, regardless of the decoded video, as can be seen in Figure 6.11a. The static frequencies used are dimensioned for the worst case, which means that the three videos with dynamic variation in their task execution times (bus, tree, akiyo) execute at a higher than necessary frame rate, as presented in Figure 6.11b.

By applying our run-time power management technique (as described in Chapter 5), the throughput of the videos with dynamic task execution time variation are lowered further than with our static power management technique while still meeting the timing requirement, as can be seen in Figure 6.11b (DVFS), or in isolation in Figure 6.12.

The lower frequencies used by our dynamic power management also translates into a further power reduction in comparison to our static technique. This can be seen in Figure 6.11a by comparing the power consumption using our dynamic technique with that of our static technique, and can be seen more clearly in Figure 6.13 where the power consumption of these techniques are shown in isolation.

The power consumption using our dynamic power management technique while decoding the wcet video is comparable with our static power management technique, which is as expected as there is little dynamic variation (there is still some due to variations in inter-core communication times) in the application's execution time to enable further reduction using DVFS. The power consumption due to our dynamic technique while decoding the wcet is actually greater than that of our static technique. This can be

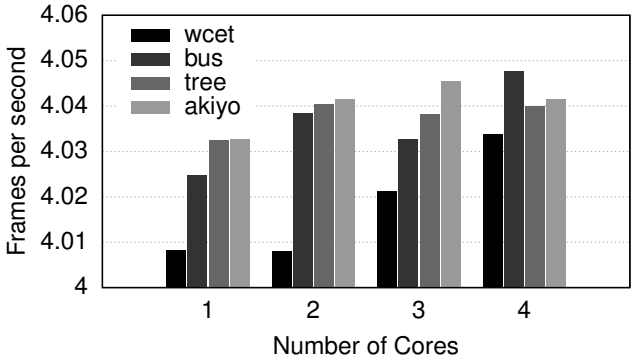


Figure 6.12: Frame rates achieved for the range of mappings and decoded videos using DVFS.

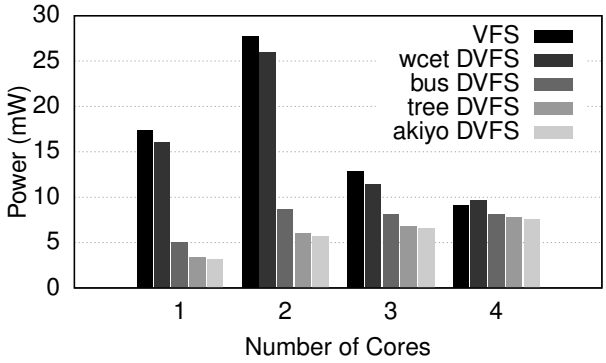


Figure 6.13: Power consumption due to our static and dynamic techniques.

explained by the application's lack of dynamic variation, coupled with the additional power consumption of the run-time power management function executing on all four cores.

For the three videos with dynamic task execution times, our dynamic technique lowers the power consumption for all four mappings, but unlike our static technique where the four core mapping produced the lowest power consumption, the single core mapping produces the lowest power consumption using our dynamic technique. For the single core mapping, earlier finishing times by the tasks always translates into a earlier finishing times at the application level. This is not necessarily the case for multi-core mappings where a task finishing earlier may have little or no affect on the application timing. Unlike for the single core mapping, tasks can block waiting on data from another core, diminishing accumulated slack.

While we do not have enough information to form a general rule (nor are we proposing that one is likely), we can say that in this case for the H.263 decoder that the four core mapping is better for low power consumption when performing static power management. Considering the single core option as the lowest power configuration when executing at maximum frequency, the four core mapping using static power management offers a 76.8% power reduction for a 15.8-50.9% decrease in throughput, for the range of videos tested. We can also say that in this case the single core mapping is generally better (not for the wcet video) for low power consumption when performing dynamic power management. In comparison to a single core mapping at maximum frequency, the single core mapping using dynamic power management offers a 59.2-92% power reduction for a 28.2-65.5% decrease in throughput.

If one design had to be chosen to be the low power design, there is no clear "winner". The four core static power managed solution (obviously) requires four cores, which although it is beneficial in terms of power consumption, is expensive in terms of area. The single core dynamic power managed solution is cheaper in terms of area, but is not guaranteed to have a lower power than the four core static power managed solution, if there is not sufficiently available dynamic slack. This leads us to the conclusion that while all our power management techniques lower the H.263 decoder's power consumption, it is not possible to objectively state that dynamic power management is in some way better than static power management, or that one of the designs provides the lowest power consumption for all decoded videos. As system design is a multidimensional trade-off, our power management techniques provide options at system design time, but can also reduce the power consumption for applications where the platform design is finalised and the mapping is known.

6.4 Summary

In this chapter, we demonstrated the composability and predictability of our power management techniques in combination with the CompSOC platform. We did this by implementing our power management techniques on an FPGA prototyped CompSOC platform instance. Each processor in the multi-core CompSOC platform used the CoMik microkernel enabling each processor to be divided into multiple virtual processors. Through experimentation using multiple applications mapped to separate virtual processors, we demonstrated that the applications were composably isolated by showing that the applications did not interfere with each others timing by even a single cycle.

We followed this by demonstrating through experimentation that the CompSOC platform with the CoMik virtualisation layer and POSe is predictable, by comparing timing output from applications executing on the FPGA prototyped CompSOC platform with output from a worst-case timing analysis using its combined application and platform HSDF. As with any worst-case analysis technique, the accuracy of our model depends on the variation of the task execution times at run-time. Our experimentation showed that for

certain platform configurations, and an application with constant task execution times, our model is accurate to within 1.86% of the measured time from the FPGA prototype.

Having demonstrated the important properties of our CompSOC platform, we applied our power management techniques to an H.263 decoder application, for a range of decoded videos and mappings. For this particular application and set of configurations, we show that our static technique can lower the power consumption by 76.8% and our dynamic technique can lower the power consumption by 59.2-92%.

CHAPTER 7

Conclusions and Future Work

Consumer demand for portable devices is on the increase. These devices are commonly powered by a battery that the user needs to keep charged. While this might seem like a minor inconvenience, battery life is currently a key driver in smartphone purchase. Apart from portable devices, there is a general trend for smart devices that simplify and automate the user experience for reasons of efficiency and safety. This continual demand for ever greater functionality has led to mixed criticality systems. In this thesis, we focus on the problem of mixed time-criticality where multiple applications of various time-criticalities share the same resources.

Timing verification of real-time applications performing power management is complicated, as power reduction is achieved by reducing computational performance, i.e. power management affects the timing behaviour of the application. Power management in mixed time-criticality systems is more complicated still, as power management of shared resources can affect the temporal behaviour of concurrent applications.

In Chapter 2, we present the composable and predictable CompSOC platform that consists of a MPSoC hardware platform and software stack. The CompSOC platform has been designed from the ground up to enable resources to be composably and predictably shared, with all shared hardware resources composably arbitrated. The CoMik microkernel composably arbitrates the processor providing a virtualised hardware interface. The physical processor is virtualised along with its DVFS mechanism (with assistance from the TIFU) enabling each virtual processor to perform independent power management

while not interfering with other virtual processors by even a single cycle.

We further describe the POSe OS that enables the execution of dataflow applications. To be able to derive timing guarantees we explain how the timing behaviours of (virtualised) dataflow applications are formalised as an independent combined application and CompSOC platform HSDFG. The CompSOC platform therefore supports the execution and verification of mixed time-criticality applications by cycle-accurately isolating virtualised hardware. Applications that are mapped onto the CompSOC platform can therefore be verified independently.

In Chapter 3, we explain how CoMik and POSe maintain distributed composable energy, power and time accounts. This information is provided at the application-level enabling the application to make progress based decisions, e.g. when performing power management. We also describe how a single energy or power resource is composably allocated among concurrent applications, enabling each application to completely use the budget it receives regardless of the behaviour of other applications, i.e. that each application has a virtual battery/power-supply. From a design perspective, we show that system dimensioning choices, such as the number of slots in CoMik's TDM table, affect the amount of energy that the system needs to hold in reserve to ensure composable use of energy/power budgets.

Applications are free to use their virtual battery however they see fit. In Chapters 4 and 5, we present temporally conservative power management techniques for real-time applications. Our power management techniques enable virtualised real-time applications that execute on the CompSOC platform to lower their power consumption without violating their timing requirements. We present a static power management method in Chapter 4, that uses an off-line power minimising convex program, of the application's combined application and CompSOC platform HSDFG, to derive low-power static VFS levels for use with the virtual processors on which it is mapped.

While our static technique lowers the application's power consumption without any modification to the application, it is unable to use run-time slack that is caused by dynamic variations in task execution time. In Chapter 5, we present a dynamic power management technique that uses distributed per-core run-time control loops to monitor local application progress, in terms of slack. The local application progress is modified using knowledge of the application's graph topology to achieve a conservative estimate of global application progress to select a low-power conservative DVFS operating point using a table lookup. The table is created off-line using a power minimising convex program to derive frequencies that can be conservatively transitioned to, for the amount of slack observed.

From our work in Chapters 4 and 5 we learned that our static HSDF dataflow model of the application and platform can be applied conservatively as part of static and dynamic power management techniques. Our investigation into using adaptive applications to provide a quality for power trade-off mechanism showed that this was possible, but unfortunately there are currently not that many suitable adaptive applications. Maybe this will change in the future.

We present experimental analyses of our techniques in Chapters 4 and 5, using a synthetic application that we use as a running example throughout this thesis. In Chapter 6 we present a case study analysis of our power management techniques applied to an H.263 decoder application that has data dependent task execution times. In doing this, we demonstrate that our techniques are not only valid in theory but are also implementable. We perform our analysis on an FPGA prototype of a four core CompSOC platform instance. We demonstrate both our static and dynamic power management techniques applied to a virtualised H.263 decoder for multiple video inputs and mappings.

From our case study analysis, we show that both our static and dynamic power management techniques can perform better than each other in certain circumstances. Our dynamic power management technique adds a power management control loop to the application's execution that is additional computation that needs to be performed within the application's timing requirement. For applications with little dynamic variation in task execution time, our static technique provides the lowest power consumption. Our case study analysis shows that our dynamic technique provides the lowest power consumption for applications with enough dynamic variation in task execution time to compensate for the additional computation required by the control loop used by our dynamic technique.

In this thesis, we have presented the CompSOC platform that enables applications to independently perform power management without violating the timing requirements of concurrent applications. We explain and demonstrate how virtualised real-time dataflow applications can perform power management without violating their timing requirements. To verify application timings, we explain how dataflow applications are formalised as a combined application and CompSOC platform HSDFG, enabling applications to be verified in isolation. Applications with mixed time-criticalities can therefore be executed on the CompSOC platform and perform independent power management, while the timing of concurrent real-time dataflow applications can still be verified in isolation.

7.0.1 Future Work

Some things to do while we wait on other engineers to invent the hoverboard...

- *An automated design and analysis flow.* The CompSOC platform already has a well developed flow for MPSoC platform generation. This could be extended to apply the analysis work in this thesis to automatically analyse applications and their mapping to generate the frequency-slack tables for use at run-time.

- *Mapping and scheduling.* The work in this thesis assumes that the application already has a mapping and a per-core SOS as a starting point. We could extend our technique with an algorithm to find a mapping and schedule that works well with our techniques to lower the power consumption.

- *clock (voltage) gating.* The power management techniques in this thesis only used DVFS to lower the voltage and frequency, but not to switch off the processor

completely. We could extend our technique to use a mixture of our current DVFS approach with powering down the processor.

- *Remapping for core shutdown.* Completely powering off a core that is under-utilised could create greater power savings than using DVFS, but requires that the virtual processors that are running on the core are remapped. It would have to be investigated how/if this could be done composably.

Bibliography

[1] International Technology Roadmap for Semiconductors (ITRS) - Process Integration, Devices, and Structures, 2011. Updated 2012. `http://www.itrs.net/reports.html`.

[2] Benchmark Videos. `https://media.xiph.org/video/derf/`, 2014.

[3] The Future of Wearable Tech. *PSFK Labs*, 2014. `http://www.slideshare.net/PSFK/psfk-future-of-wearable-technology-report`.

[4] B. Akesson and K. Goossens. *Memory Controllers for Real-Time Embedded Systems*. Embedded Systems Series. Springer, first edition edition, 2011.

[5] B. Akesson, A. Hansson, and K. Goossens. Composable resource sharing based on latency-rate servers. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD'09. 12th Euromicro Conference on*, pages 547–555. IEEE, 2009.

[6] Y. Akgul, D. Puschini, S. Lesecq, E. Beigne, P. Benoit, and L. Torres. Methodology for Power Mode selection in FD-SOI circuits with DVFS and Dynamic Body Biasing. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2013 23rd International Workshop on*, pages 199–206. IEEE, 2013.

[7] S. Albers. Energy-efficient algorithms. *Communications of the ACM*, 53(5):86–96, 2010.

[8] S. Albers, F. Müller, and S. Schmelzer. Speed scaling on parallel processors. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 289–298, New York, NY, USA, 2007. ACM.

[9] A. Alimonda, S. Carta, A. Acquaviva, A. Pisano, and L. Benini. A feedback-based approach to dvfs in data-flow applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(11):1691–1704, 2009.

[10] ARINC. 653 Avionics Application Software Standard Interface. `http://www.arinc.com`.

[11] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and linearity*, volume 3. Wiley New York, 1992.

[12] J.-D. Bakker, K. Langendoen, and H. Sips. Lart: Flexible, low-power building blocks for wearable computers. In *Distributed Computing Systems Workshop, 2001 International Conference on*, pages 255–259. IEEE, 2001.

[13] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, volume 99, pages 45–58, 1999.

[14] M. Bekooij, A. Moonen, and J. van Meerbergen. Predictable and composable multiprocessor system design: A constructive approach. In *Bits&Chips Symposium on Embedded Systems and Software*, 2007.

[15] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(3):299–316, 2000.

[16] A. Beyranvand Nejad. *Composable Virtual Platforms for Mixed-Criticality Embedded Systems*. PhD thesis, Delft University of Technology, Nov. 2014.

[17] K. A. Bowman, B. L. Austin, J. C. Eble, X. Tang, and J. D. Meindl. A physical alpha-power law MOSFET model. In *Proceedings of the 1999 international symposium on Low power electronics and design*, pages 218–222. ACM, 1999.

[18] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

[19] R. J. Bril, C. Hentschel, E. F. Steffens, M. Gabrani, G. van Loo, and J. Gelissen. Multimedia qos in consumer terminals. In *Signal Processing Systems, 2001 IEEE Workshop on*, pages 332–343. IEEE, 2001.

[20] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 1, pages 288–297. IEEE, 1995.

[21] A. Burns and R. Davis. Mixed criticality systems: A review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

[22] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer, 2011.

[23] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Throughput-constrained dvfs for scenario-aware dataflow graphs. In *In 19th Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 175–184. IEEE, 2013.

[24] P. de Langen. *Energy Reduction Techniques for Caches and Multiprocessors*. Oct. 2009.

[25] P. de Langen and B. Juurlink. Leakage-aware multiprocessor scheduling. *Journal of Signal Processing Systems*, 57(1):73–88, 2009.

[26] D. Dolev, M. Függer, C. Lenzen, M. Perner, and U. Schmid. HEX: scaling honeycombs is easier than scaling clock trees. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, pages 164–175. ACM, 2013.

[27] S. A. Edwards and E. A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th annual Design Automation Conference*, pages 264–265. ACM, 2007.

[28] O. Faynot, F. Andrieu, O. Weber, C. Fenouillet-Beranger, P. Perreau, J. Mazurier, T. Benoist, O. Rozeau, T. Poiroux, M. Vinet, L. Grenouillet, J.-P. Noel, N. Posseme, S. Barnola, F. Martin, C. Lapeyre, M. Casse, X. Garros, M. A. Jaud, O. Thomas, G. Cibrario, L. Tosti, L. Brevard, C. Tabone, P. Gaud, S. Barraud, T. Ernst, and S. Deleonibus. Planar fully depleted soi technology: A powerful architecture for the 20nm node and beyond. In *Electron Devices Meeting (IEDM), 2010 IEEE International*, pages 3.2.1–3.2.4, 2010.

[29] M. E. Gerards, J. L. Hurink, P. K. Holzenspies, J. Kuper, and G. J. Smit. Analytic clock frequency selection for global dvfs. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 512–519. IEEE, 2014.

[30] M. E. Gerards and J. Kuper. Optimal dpm and dvfs for frame-based real-time systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):41, 2013.

[31] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. 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.

[32] K. Goossens and A. Hansson. The Æthereal network on chip after ten years: Goals, evolution, lessons, and future. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 306–311. IEEE, 2010.

[33] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 525–530. EDA Consortium, 2013.

[34] S. Goossens, B. Akesson, M. Koedam, A. B. Nejad, A. Nelson, and K. Goossens. The CompSOC design flow for virtual execution platforms. In *Proceedings of the 10th FPGAworld Conference*, page 7. ACM, 2013.

[35] S. Goossens, B. Akesson, M. Koedam, A. B. Nejad, A. Nelson, and K. Goossens. The CompSOC design flow for virtual execution platforms. In *Proceedings of the 10th FPGAworld Conference*, page 7. ACM, 2013.

[36] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens. A reconfigurable real-time sdram controller for mixed time-criticality systems. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–10. IEEE, 2013.

[37] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25. ACM, 1995.

[38] R. Govindarajan and G. R. Gao. A novel framework for multi-rate scheduling in dsp applications. In *Application-Specific Array Processors, 1993. Proceedings., International Conference on*, pages 77–88. IEEE, 1993.

[39] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx`, Mar. 2014.

[40] Green Hills Software. INTEGRITY. `http://www.ghs.com`.

[41] Z. Gu and Q. Zhao. A state-of-the-art survey on real-time issues in embedded systems virtualization. 2012.

[42] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[43] A. Hansson and K. Goossens. *On-Chip Interconnect with aelite*. Embedded Systems. Springer, Dordrecht, 2011.

[44] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2, 2009.

[45] G. Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 11–16. ACM, 2008.

[46] G. Heiser and B. Leslie. The okl4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010.

[47] S. Irani and K. R. Pruhs. Algorithmic problems in power management. *ACM SIGACT News*, 36(2):63–76, 2005.

[48] ITU-T. *Recommendation H.263*, 2005. `http://www.itu.int/rec/T-REC-H.263-200501-I`.

[49] D. Jacquet, G. Cesana, P. Flatresse, F. Arnaud, P. Menut, F. Hasbani, T. Di Gilio, C. Lecocq, T. Roy, A. Chhabra, C. Grover, O. Minez, J. Uginet, G. Durieu, F. Nyer, C. Adobati, R. Wilson, and D. Casalotto. 2.6GHz ultra-wide voltage range energy efficient dual A9 in 28nm UTBB FD-SOI. In *VLSI Technology (VLSIT), 2013 Symposium on*, pages C44–C45, 2013.

[50] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 275–280, 2004.

[51] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41st annual Design Automation Conference*, pages 275–280. ACM, 2004.

[52] R. John. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, 1999.

[53] R. Jordans, F. Siyoum, S. Stuijk, A. Kumar, and H. Corporaal. An automated flow to map throughput constrained applications to a mpsoc. In *OASIcs-OpenAccess Series in Informatics*, volume 18. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

[54] P. Juang, Q. Wu, L.-S. Peh, M. Martonosi, and D. W. Clark. Coordinated, distributed, formal energy management of chip multiprocessors. In *Low Power Electronics and Design, 2005. ISLPED'05. Proceedings of the 2005 International Symposium on*, pages 127–130. IEEE, 2005.

[55] A. Keshavarzi, D. Somasekhar, M. Rashed, S. Ahmed, K. Maitra, R. Miller, A. Knorr, J. Cho, R. Augur, S. Banna, C.-H. Shaw, A. Halliyal, U. Schroeder, A. Wei, J. Egley, K. Korablev, S. Luning, M.-R. Lin, S. Venkatesan, S. Kengeri, and G. Bartlett. Architecting advanced technologies for 14nm and beyond with 3d finfet transistors for the future soc applications. In *Electron Devices Meeting (IEDM), 2011 IEEE International*, pages 4.1.1–4.1.4, 2011.

[56] S. Khan and E. Marzec. Wearables – Tech Trends. *Deloitte University Press*, 2014. http://dupress.com/articles/2014-tech-trends-wearables/.

[57] H. Kopetz. Internet of things. In *Real-Time Systems*, pages 307–323. Springer, 2011.

[58] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[59] T.-W. Lam, L.-K. Lee, I. K. To, and P. W. Wong. Energy efficient deadline scheduling in two processor systems. In *Algorithms and Computation*, pages 476–487. Springer, 2007.

[60] T.-W. Lam, L.-K. Lee, I. K. To, and P. W. Wong. Competitive non-migratory scheduling for flow time and energy. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 256–264. ACM, 2008.

[61] E. Larsson, B. Vermeulen, and K. Goossens. A distributed architecture to check global properties for post-silicon debug. In *Test Symposium (ETS), 2010 15th IEEE European*, pages 182–187. IEEE, 2010.

[62] A. Lele, O. Moreira, and P. J. Cuijpers. A new data flow analysis model for tdm. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 237–246. ACM, 2012.

[63] Y. Li, B. Akesson, and K. Goossens. Dynamic command scheduling for real-time memory controllers. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[64] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, pages 137–146, New York, NY, USA, 2008. ACM.

[65] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proc. Int'l Conference on Computer Design (ICCD)*, Oct. 2012.

[66] LynuxWorks. LynxOS-178. http://www.lynuxworks.com.

[67] P. Magarshack, P. Flatresse, and G. Cesana. Utbb fd-soi: A process/design symbiosis for breakthrough energy-efficiency. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 952–957, San Jose, CA, USA, 2013. EDA Consortium.

[68] D. McIntire, K. Ho, B. Yip, A. Singh, W. Wu, and W. J. Kaiser. The low power energy aware processing (leap) embedded networked sensor system. In *Proceedings of the 5th international conference on Information processing in sensor networks*, pages 449–457. ACM, 2006.

[69] D. McIntire, T. Stathopoulos, S. Reddy, T. Schmidt, and W. J. Kaiser. Energy-efficient sensing with the low power, energy aware processing (leap) architecture. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(2):27, 2012.

[70] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, July 2004.

[71] M. Meijer. On-chip fully-digital power supply control. *Master's thesis, TU Eindhoven*, 2005.

[72] M. Meijer and J. P. de Gyvez. Technological boundaries of voltage and frequency scaling for power performance tuning. In *Adaptive Techniques for Dynamic Processor Optimization*, pages 25–47. Springer, 2008.

[73] M. Meijer, J. Pineda de Gyvez, and R. Otten. On-chip digital power supply control for system-on-chip applications. In *Proceedings of the 2005 international symposium on Low power electronics and design*, pages 311–314. ACM, 2005.

[74] G. Meyer and S. Deix. Research and innovation for automated driving in germany and europe. In *Road Vehicle Automation*, pages 71–81. Springer, 2014.

[75] A. K. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE*, pages 75–84. IEEE, 2001.

[76] O. M. Moreira and M. J. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007.

[77] T. Nakajima, Y. Kinebuchi, H. Shimada, A. Courbot, and T.-H. Lin. Temporal and spatial isolation in a virtualization layer for multi-core processor based information appliances. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pages 645–652. IEEE Press, 2011.

[78] A. Nelson, B. Akesson, A. Molnos, S. te Pas, and K. Goossens. Power versus quality trade-offs for adaptive real-time applications. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2012 IEEE 10th Symposium on*, pages 75–84. IEEE, 2012.

[79] A. Nelson, A. Beyranvand Nejad, A. Molnos, M. Koedam, and K. Goossens. CoMik: A Predictable and Cycle-Accurately Composable Real-Time Microkernel. *Design, Automation & Test in Europe*, 2014.

[80] A. Nelson, A. Molnos, and K. Goossens. Composable power management with energy and power budgets per application. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 396–403. IEEE, 2011.

[81] A. Nelson, O. Moreira, A. Molnos, S. Stuijk, B. T. Nguyen, and K. Goossens. Power minimisation for real-time dataflow applications. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pages 117–124. IEEE, 2011.

[82] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. P. Llopis, and P. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3):233–270, 2002.

[83] G. Pelz, P. Oehler, E. Fourgeau, and C. Grimm. Automotive system design and autosar. In *Advances in Design and Specification Languages for SoCs*, pages 293–305. Springer, 2005.

[84] M. Perner, M. Sigl, U. Schmid, and C. Lenzen. Byzantine self-stabilizing clock distribution with hex: Implementation, simulation, clock multiplication. In *DEPEND 2013, The Sixth International Conference on Dependability*, pages 6–15, 2013.

[85] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 251–259. ACM, 2001.

[86] J. Pouwelse, K. Langendoen, and H. Sips. Energy priority scheduling for variable voltage processors. In *Low Power Electronics and Design, International Symposium on, 2001.*, pages 28–33. IEEE, 2001.

[87] J. Pouwelse, K. Langendoen, and H. J. Sips. Application-directed voltage scaling. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11(5):812–826, 2003.

[88] P. J. Prisaznuk. Arinc 653 role in integrated modular avionics (ima). In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1–E. IEEE, 2008.

[89] K. Pruhs, R. van Stee, and P. Uthaisombut. Speed scaling of tasks with precedence constraints. *Theory of Computing Systems*, 43(1):67–80, 2008.

[90] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. *Proceedings of the IEEE*, 91(2):305–327, 2003.

[91] T. Sakurai and A. R. Newton. Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. *Solid-State Circuits, IEEE Journal of*, 25(2):584–594, 1990.

[92] S. Samolej. ARINC Specification 653 Based Real-Time Software Engineering. *e-Informatica*, 5(1):39–49, 2011.

[93] A. Smith. Smartphone ownership–2013 update. *Pew Research Center: Washington DC*, 2013. `http://pewinternet.org/Reports/2013/Smartphone-Ownership-2013.aspx`.

[94] R. Stefan. *Resource Allocation in Time-division-multiplexed Networks on Chip*. PhD thesis, Delft University of Technology, Apr. 2012.

[95] R. Stefan, A. Molnos, and K. Goossens. daelite: A tdm noc supporting qos, multicast, and fast connection set-up. 2012.

[96] SYSGO. PikeOS. `http://www.sysgo.com`.

[97] S. te Pas. Quality versus energy trade-off for real-time applications on a composable MPSoC. Master's thesis, Department of Electrical Engineering, Eindhoven University of Technology, 2012.

[98] Texas Instruments. OMAP3530 Power Estimation Spreadsheet. `http://processors.wiki.ti.com`, 2010.

[99] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, pages 101–104. IEEE, 2000.

[100] L. Thiele and N. Stoimenov. Modular performance analysis of cyclic dataflow graphs. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 127–136. ACM, 2009.

[101] W. Tong, O. Moreira, R. Nas, and K. van Berkel. Hard-real-time scheduling on a weakly programmable multi-core processor with application to multi-standard channel decoding. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 151–160. IEEE, 2012.

[102] L. Ulrich. Top ten tech cars. *Spectrum, IEEE*, 51(4):38–47, 2014.

[103] D. van Nijnatten. Budgeting energy in a composable microkernel. Master's thesis, Department of Electrical Engineering, Eindhoven University of Technology, 2013.

[104] P. Vivet, E. Beigne, H. Lebreton, and N.-E. Zergainoh. On line power optimization of data flow multi-core architecture based on vdd-hopping for local DVFS. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*, pages 94–104. Springer, 2011.

[105] M. Waitz. Accounting and control of power consumption in energy-aware operating systems. Master's thesis, Department of Computer Science 4, University of Erlangen, 2003.

[106] M. Weiser. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991.

[107] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit. Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, pages 10–15, New York, NY, USA, 2006. ACM.

[108] M. H. Wiggers, M. J. Bekooij, and G. J. Smit. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, pages 11–22. ACM, 2007.

[109] Wind River. VxWorks 653. `http://www.windriver.com`.

[110] J. Windsor and K. Hjortnaes. Time and space partitioning in spacecraft avionics. In *Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference on*, pages 13–20. IEEE, 2009.

[111] C. C. Wüst, L. Steffens, W. F. Verhaegh, R. J. Bril, and C. Hentschel. Qos control strategies for high-quality video processing. *Real-Time Systems*, 30(1-2):7–29, 2005.

[112] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 374–382. IEEE, 1995.

[113] K. Zickuhr. Tablet ownership 2013. *Pew Research Center: Washington DC*, 2013. `http://pewinternet.org/Reports/2013/Tablet-Ownership-2013.aspx`.

APPENDIX A

Glossary

This chapter provides a guide to the language used in this thesis. Section A.1 contains a list of abbreviations.

A.1 Abbreviations

This list of abbreviations explains the most commonly used abbreviations in this thesis, along with the page number on which they are first used.

ACB Application Control Block 51
API Application Programming Interface 51
ASIC Application Specific Integrated Circuit 37
BA Bounded Alignment 147
CCB CoMik Control Block 40
CCSP Credit Controlled Static Priority 27
CDC Clock Domain Crossing 69
CMOS Complementary Metal-Oxide-Semiconductor 68
CoMik Composable and Predictable Microkernel 7
POSe Predictable Operating System 7

CompSOC Composable and Predictable System-on-Chip 7
CSDF Cyclo-Static Dataflow 14
CSDFG Cyclo-Static Dataflow Graph 14
DCP Disciplined Convex Program 92
DCT Discrete Cosine Transform 106
DMA Direct Memory Access 7
DRAM Dynamic Random Access Memory 152
DTL Device Transaction Level 24
DVFS Dynamic Voltage and Frequency Scaling 4
FA Fully Aligned 145
FCB FIFO Control Block 51
FD-SOI Fully Depleted Silicon-on-Insulator 70
FET Field-Effect Transistor 69
FIFO First In First Out 9
FPGA Field Programmable Gate Array 9
FR Frame Reconstruction 106
FSL Fast Simplex Link 35
FSM Finite State Machine 137
FSM-SADF Finite State Machine Scenario Aware Dataflow 137
GALS Globally Asynchronous Locally Synchronous 48
HSDF Homogeneous Synchronous Dataflow 14
HSDFG Homogeneous Synchronous Dataflow Graph 9
ID Identification 52
IDCT Inverse Discrete Cosine Transform 106
IQ Inverse Quantisation 106
ISA Instruction Set Architecture 35
KPN Kahn Process Network 14
LMB Local Memory Bus 24
MC Motion Compensation 106
MCM Maximum Cycle Mean 21
MMIO Memory Mapped Input/Output 24
MMU Memory Management Unit 23

MOC Model of Computation 12
MOE Model of Execution 7
MOSFET Metal-Oxide-Semiconductor Field-Effect Transistor 68
MPSoC Multiprocessor System on Chip 17
MPU Memory Protection Unit 23
NoC Network on Chip 16
PCB Partition Control Block 40
PIC Programmable Interrupt Controller 35
PIT Programmable Interrupt Timer 35
PLB Processor Local Bus 24
PRET Precision-Timed Systems 22
PSNR Peak Signal to Noise Ratio 106
OS Operating System 4
RC Read Counter 28
SADF Scenario Aware Dataflow 137
SDF Synchronous Dataflow 14
SDFG Synchronous Dataflow Graph 14
SDRAM Synchronous Dynamic Random Access Memory 22
SoC System on Chip 68
SOS Static-Order Schedule 18
SPS Static-Periodic Schedule 15
SRAM Static Random Access Memory 31
STS Self-Timed Schedule 15
TCB Task Control Block 51
TDM Time Division Multiplexed 6
TIFU Timer-centric Interrupt and Frequency Unit 8
TFT Thin-Film Transistor 22
TTA Time Triggered Architecture 63
UP Up Scaling 106
VFS Voltage and Frequency Scaling 9
VLD Variable Length Decoding 106
VP Virtual Platform 5

WC Write Counter 28
WCA Worst-Case Arrival 145
WCET Worst-Case Execution Time 9
WCRT Worst-Case Response Time 149
WCSTS Worst-Case Self-Timed Schedule 18
WLAN Wireless Local Area Network 2
XML Extensible Markup Language 22

A.2 Lists of Symbols

In this section, we present an overview of the commonly used symbols throughout this thesis. In the following sections, the symbols are organised per topic.

A.2.1 Dataflow Graphs

Table A.1 presents the commonly used symbols for dataflow graph analysis.

Symbol	Description	Page
μ	Dataflow graph Maximum Cycle Mean (MCM)	Page 21
C	Set of dataflow graph cycles	Page 21
d	Initial tokens	Page 19
E	Set of dataflow communication channels	Page 18
G	Tuple (V,E,t,d) representing dataflow graph	Page 18
K	Number of graph iterations until the periodic phase of execution under a Self-Timed Schedule (STS)	Page 21
N	Dataflow graph cyclicity	Page 21
s	Task start time	Page 19
t	Task execution time	Page 19
T	Schedule period under a Static-Periodic Schedule (SPS)	Page 21
V	Set of dataflow actors	Page 18

Table A.1: List of common dataflow symbols.

A.2.2 Latency-Rate Servers

Table A.2 presents the commonly used symbols for latency-rate server analysis.

Symbol	Description	Page
L	Latency component	Page 46
$\overline{r}$	Worst-case response	Page 47
R	Rate component	Page 46
S	Service duration	Page 46
T	Time frame in which the service is provided	Page 46

Table A.2: List of commonly used latency-rate server symbols.

A.2.3 Composable Accounting

Symbol	Description	Page
A	Smallest TDM allocation assigned to a partition	Page 85
c	CoMik slot duration	Page 81
E_{comik}	Energy allocated for a single CoMik slot	Page 78
$\overline{E}_{\text{comik}}$	Total energy allocated for virtual processor final slots on a processor	Page 79
E_{core}	Energy allocated to a single physical processor	Page 78
E_{final}	Energy allocated to bound the energy consumption of the virtual processor's final slot	Page 78
$\overline{E}_{\text{final}}$	Total energy allocated to bound the energy consumption of virtual processor final slots on a processor	Page 79
E_n	Energy allocated to virtual processor n	Page 79
$E_{\text{partition}}$	Energy allocated for a single partition slot	Page 78
$\overline{E}_{\text{partition}}$	Total energy allocated for the partition slots on a processor	Page 83
E_{reserve}	Energy held in reserve to bound the energy consumption of a single partition slot	Page 78
$\overline{E}_{\text{reserve}}$	Total energy held in reserve to bound the energy consumption of partition slots on a processor	Page 83
f	Frequency	Page 68
f_{cap}	Cap on maximum frequency	Page 89
f_{comik}	CoMik frequency	Page 82
N	Number of TDM table iterations	Page 78
p	CoMik partition slot duration	Page 81
P_{comik}	Power consumption of CoMik	Page 81
P_{core}	Power budget of a processor n	Page 88
P_{max}	Maximum power consumption	Page 79
P_{min}	Minimum power consumption	Page 79

Continued on next page

Table A.3 – continued from previous page

Symbol	Description	Page
P_{platform}	Power consumption of the rest of the platform (NoC, SDRAM, etc.)	Page 94
$\overline{R}$	Maximum composable run time	Page 83
S	Number of service slots in a TDM table	Page 78
V	Number of virtual processors	Page 78
W_{comik}	CoMik work in cycles	Page 82

Table A.3: List of symbols.

A.2.4 Conservative Slack Estimation

Symbol	Description	Page
B_c	Maximum number of application graph iterations that the SOS on core c can be ahead of an SOS on another core	Page 127
N	Number of application graph iterations between power management function invocations	Page 127
T	Schedule period under a Static-Periodic Schedule (SPS)	Page 122
Y_c	Local observed slack on core c	Page 127
Z_c	Conservative global slack estimation on core c	Page 127

Table A.4: List of symbols.

APPENDIX B

Example CoMik and POSe Application Configuration

In Code B.1, we present an example of the C code used to configure CoMik and POSe to execute an H.263 decoder application. This example configuration is for a single tile mapping.

```
1  #include "comik.h"
2  #include "pose.h"
3
4  #include "h263_headers.h"
5  #include "h263_firing-rules.h"
6  #include "h263_memmap.h"
7
8  #include "vld.h"
9  #include "dQuant.h"
10 #include "idct.h"
11 #include "AddBlock.h"
12 #include "upscale.h"
13
14 /*
15  * App constants
16  */
17 #define h263_ID_APP1    1
18 #define NBR_LOCAL_TASK_h263 5
19
20 /* Create CoMik TDM schedule */
21 int tdm_table[2] = { 1, 0 };
```

```
22 TABLE app_tdm = { 0, sizeof(tdm_table) / sizeof(int), tdm_table };
23
24 /* Create H.263 SO schedule */
25 int h263_task_table_so_app1[5] = { 1, 2, 3, 4, 5 };
26 TABLE h263_task_so_app1 = { 0, sizeof(h263_task_table_so_app1) / sizeof(
       int), h263_task_table_so_app1 };
27
28 void app1_entry()
29 {
30   os_initialise_libpose();
31
32   /*
33    * Application h263
34    */
35   os_add_application(h263_ID_APP1, NBR_LOCAL_TASK_h263,
          os_task_scheduler_so, (void*)&h263_task_so_app1);
36
37   /*
38    * Add Tasks of Application H.263
39    */
40   os_add_task(1, 0, h263_VLD_app1);
41   os_set_task_fifos(1, 4, 1);
42   os_add_task(2, 0, h263_DQuant_app1);
43   os_set_task_fifos(2, 2, 2);
44   os_add_task(3, 0, h263_idct_app1);
45   os_set_task_fifos(3, 1, 2);
46   os_add_task(4, 0, h263_AddBlock_app1);
47   os_set_task_fifos(4, 2, 3);
48   os_add_task(5, 0, h263_Upscale_app1);
49
50   os_set_task_fifos(5, 0, 1);
51
52   /*
53    * Add FIFOs of Application H.263
54    */
55   os_add_fifo(1, 1, 2, 0, 0, INT_PTR(NULL), INT_PTR(NULL), INT_PTR(NULL),
          INT_PTR(NULL), VOID_PTR(NULL), VOID_PTR(NULL), 1, sizeof(
         VLD2Dquant),        1, 1, 0, 0);
56   os_add_fifo(2, 1, 3, 1, 0, INT_PTR(NULL), INT_PTR(NULL), INT_PTR(NULL),
          INT_PTR(NULL), VOID_PTR(NULL), VOID_PTR(NULL), 1, sizeof(VLD2Idct)
         ,          1, 1, 0, 0);
57   os_add_fifo(3, 1, 4, 2, 0, INT_PTR(NULL), INT_PTR(NULL), INT_PTR(NULL),
          INT_PTR(NULL), VOID_PTR(NULL), VOID_PTR(NULL), 1, sizeof(
         VLD2AddBlock),      1, 1, 0, 0);
58   os_add_fifo(4, 2, 3, 0, 1, INT_PTR(NULL), INT_PTR(NULL), INT_PTR(NULL),
          INT_PTR(NULL), VOID_PTR(NULL), VOID_PTR(NULL), 1, sizeof(
         DQuant2Idct),       1, 1, 0, 0);
59   os_add_fifo(5, 3, 4, 0, 1, INT_PTR(NULL), INT_PTR(NULL), INT_PTR(NULL),
          INT_PTR(NULL), VOID_PTR(NULL), VOID_PTR(NULL), 1, sizeof(
         Idct2AddBlock),     1, 1, 0, 0);
60   os_add_fifo(6, 4, 5, 0, 0, INT_PTR(NULL), INT_PTR(NULL), INT_PTR(NULL),
          INT_PTR(NULL), VOID_PTR(NULL), VOID_PTR(NULL), 1, sizeof(
         AddBlock2Up),       1, 1, 0, 0);
```

```
61   os_add_fifo(7, 1, 1, 3, 0, INT_PTR(NULL), INT_PTR(NULL), INT_PTR(NULL),
          INT_PTR(NULL), VOID_PTR(NULL), VOID_PTR(NULL), 1, sizeof(
         VLDDataLocal),      1, 1, 0, 0);
62   os_add_fifo(8, 2, 2, 1, 1, INT_PTR(NULL), INT_PTR(NULL), INT_PTR(NULL),
          INT_PTR(NULL), VOID_PTR(NULL), VOID_PTR(NULL), 1, sizeof(
         DQuantDataLocal),   1, 1, 0, 0);
63   os_add_fifo(9, 4, 4, 1, 2, INT_PTR(NULL), INT_PTR(NULL), INT_PTR(NULL),
          INT_PTR(NULL), VOID_PTR(NULL), VOID_PTR(NULL), 1, sizeof(
         AddBlockDataLocal), 1, 1, 0, 0);
64
65   /*
66    * Insert initial tokens where necessary
67    */
68   os_insert_fifo_token(7, 1, 1, NULL);
69   os_insert_fifo_token(8, 2, 1, NULL);
70   os_insert_fifo_token(9, 4, 1, NULL);
71
72   /*
73    * Set Tasks Firing Rules Parameters of Application H.263
74    */
75   os_set_task_firing_rule(1, firing_rule_h263_VLD,      1);
76   os_set_task_firing_rule(2, firing_rule_h263_DQuant,   1);
77   os_set_task_firing_rule(3, firing_rule_h263_idct,     1);
78   os_set_task_firing_rule(4, firing_rule_h263_AddBlock, 1);
79   os_set_task_firing_rule(5, firing_rule_h263_Upscale,  1);
80
81   os_set_app_throughput_requirement(h263_ID_APP1, 0.1, 99); // 10 fps
82
83   os_application_start();
84 }
85
86 int main()
87 {
88   mk_mon_sync();
89
90   /* Setup CoMik */
91   mk_init_comik(0, 0, &app_tdm);
92
93   /* Setup a virtual processor (partition) */
94   mk_init_partition(h263_ID_APP1, 22*1024, 138*1024, app1_entry, NULL,
         NULL);
95
96   /*
97    * Set the microkernel and virtual processor slot durations
98    */
99   mk_set_mk_slot_duration(0x1000);
100  mk_set_partition_slot_duration(0x100000);
101
102  mk_start();
103 }
```

Code B.1: Example CoMik and POSe initialisation C code for a single-core H.263 decoder.

APPENDIX C

Example HSDFG Convex Analysis Script

In Code C.1, we present a Matlab script that is used in conjunction with the CVX toolbox to perform convex analysis of a combined application and platform HSDFG. This script is used to find either the minimum period of the application's SPS (`mint`) or the application's minimum power consumption (`minp`).

```
1  function [cvx_optval,f,sSc,sr,se,sw,sLc,sd,sn,B,iL,V,T,energy] = static(
       mode,varargin)
2
3  if ischar(mode)
4    mode = lower(mode);
5  else
6    error('mode must be a string');
7  end
8
9  Lat   = [0,0];
10 R_inv = [1,1];
11 V     = 0;
12 iL    = 0;
13
14 switch mode
15 case 'mint'
16   c = varargin{1};
17   p = varargin{2};
18   S = varargin{3};
19   A = varargin{4};
```

```
  V = varargin{5};
  tR = (A*p)/(S*(c+p));
  R = [tR,tR];
  iL = (S*(c+p)) - (A*p) +1 -(1/tR);
  R_inv = 1./R;
  Lat = Lat*0+V;
case 'minp'
  c = varargin{1};
  p = varargin{2};
  S = varargin{3};
  A = varargin{4};
  V = varargin{5};
  T = varargin{6};
  tR = (A*p)/(S*(c+p));
  R = [tR,tR];
  iL = (S*(c+p)) - (A*p) +1 -(1/tR);
  R_inv = 1./R;
  Lat = Lat*0+V;
otherwise
  error('Unrecognised mode');
end

s = [1152,1200,1060,1173];
r = [5484,5520,824,855];
e = [4162,8062,6112,2862];
w = [1336,975,1644,1283];
L = [0,0,0,0];
d = sparse([1,3,2,4],[2,4,1,3],[15,15,15,15]);
n = sparse([1,3,2,4],[2,4,1,3],[73,73,73,73]);

corder = sparse([1,3,2,4],[2,4,1,3],1:4);
scheds    = {[1,3],[2,4]};
dmascheds = {{[1,2],[3,4]},{[2,1],[4,3]}};

B = sparse([1,3],[2,4],[2,2],nnz(d),nnz(d));

if length(scheds) > 0
  for i = 1:length(scheds)
    sched = scheds{i};
    for j = 1:length(sched)
      mapping(sched(j)) = i;
      s(sched(j)) = s(sched(j)) * R_inv(i);
      r(sched(j)) = r(sched(j)) * R_inv(i);
      e(sched(j)) = e(sched(j)) * R_inv(i);
      w(sched(j)) = w(sched(j)) * R_inv(i);

      L(sched(j)) = Lat(i);
    end
  end
end

fmax=120;
Prest=(120^3)*3.353e-5 + 2.065;
```

```
73
74 tic;
75 cvx_begin quiet
76   variable sSc(size(s,2))
77   variable sr(size(r,2))
78   variable se(size(e,2))
79   variable sw(size(w,2))
80   variable sLc(size(L,2))
81   variable sd(nnz(corder))
82   variable sn(nnz(corder))
83
84   switch mode
85   case 'mint'
86     variable T
87     f = ones(length(scheds),1)*fmax;
88     minimise(T)
89   case 'minp'
90     variable f(length(scheds))
91
92     if length(scheds) > 0
93       P = '';
94       for i = 1:length(scheds)
95         sched = scheds{i};
96         if ~strcmp(P,'')
97           P = strcat(P,' + ');
98         end
99         P = strcat(P,sprintf('(pow_pos(f(%d),3)*3.353e-5 + 2.065 + (Prest
               /2))*R(%d)',i,i));
100      end
101    end
102
103    minimise(P)
104
105    f >= (fmax/16)*0.9999999
106    f <= fmax
107  otherwise
108    error('Unrecognised mode');
109  end
110
111  % static order schedule
112  if length(scheds) > 0
113    for i = 1:length(scheds)
114      sched = scheds{i};
115      for j = 1:length(sched)
116        p = sched(j);
117        if j == length(sched)
118          c = sched(1);
119          D = 1;
120        else
121          c = sched(j+1);
122          D = 0;
123        end
124
```

```
125         sr(p)        >= sSc(p) + s(p) * inv_pos(f(i)) * fmax
126         se(p)        >= sr(p)  + r(p) * inv_pos(f(i)) * fmax
127         sw(p)        >= se(p)  + e(p) * inv_pos(f(i)) * fmax
128         sSc(c) + D*T >= sw(p)  + w(p) * inv_pos(f(i)) * fmax
129
130         sSc(p)       >= sLc(p) + L(i)
131       end
132     end
133   end
134
135   % inter core communication
136   if length(dmascheds) > 0
137     for i = 1:length(dmascheds)
138       sched = dmascheds{i};
139       for j = 1:length(sched)
140         comm = sched{j};
141         cn   = corder(comm(1),comm(2));
142         if j == length(sched)
143           ncomm = sched{1};
144           ncn   = corder(ncomm(1),ncomm(2));
145           D = 1;
146         else
147           ncomm = sched{j+1};
148           ncn   = corder(ncomm(1),ncomm(2));
149           D = 0;
150         end
151
152         sd(cn) >= sw(comm(1)) + w(comm(1)) * inv_pos(f(mapping(comm(1))))
                * fmax
153         sn(cn) >= sd(cn) + d(comm(1),comm(2))
154         sLc(comm(2)) + B(comm(2),comm(1))*T >= sn(cn) + n(comm(1))
155         sLc(comm(1)) + T >= sd(cn) + d(comm(1),comm(2))
156
157         sd(ncn) + D*T >= sd(cn) + d(comm(1),comm(2))
158       end
159     end
160   end
161 cvx_end
162 toc;
163
164 f=ceil(f.*16./fmax)*fmax/16;
165
166 energy = sum(sum((((f).^3)*3.353e-5 + 2.065 + Prest/2))*R);
167
168 if ~strcmp(cvx_status,'Solved')
169   cvx_optval = -1;
170   energy = -1;
171 end
```

Code C.1: Example Matlab code to perform a convex analysis (to minimise the period or power consumption) of a combined application and platform HSDFG.

APPENDIX D

Curriculum Vitae

Andrew Nelson was born in Craigavon, Northern Ireland in 1983. He received his M.Sc. degree in Embedded Systems at Eindhoven University of Technology, the Netherlands in 2009. After this, he moved to Delft University of Technology to perform research towards achieving a Ph.D. Since 2014, he is employed as a Researcher at Eindhoven University of Technology. His research interests include real-time (including mixed time-criticality) low-power multi-core embedded-systems and the timing analyses thereof.

APPENDIX E

Publications

Journal Articles

[1] ***Andrew Nelson***, Kees Goossens, Benny Akesson, "Dataflow Formalisation of Real-Time Streaming Applications on a Composable and Predictable Multi-Processor SOC", Under submission to SYSARC: Special Issue on High-performance and Real-time Embedded Systems.

[2] Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, Ashkan Beyranvand Nejad, ***Andrew Nelson***, Shubhendu Sinha, "Virtual execution platforms for mixed-time-criticality systems: The CompSOC architecture and design flow", ACM SIGBED, vol. 1, 2013.

[3] Andreas Hansson, Marcus Ekerhult, Anca Molnos, Aleksandar Milutinovic, ***Andrew Nelson***, Jude Ambrose, Kees Goossens, "Design and implementation of an operating system for composable processor sharing", Microprocessors and Microsystems, Elsevier, vol. 35, no. 2, pp. 246-260, 2011.

Conference Papers

[1] Benny Akesson, Anna Minaeva, Premysl Sucha, ***Andrew Nelson***, Zdenek Hanzalek, "An Efficient Configuration Methodology for Time-Division Multiplexed Resources", Work in progress.

[2] Shubhendu Sinha, Martijn Koedam, Rob van Wijk, ***Andrew Nelson***, Ashkan Beyranvand Nejad, Marc Geilen, Kees Goossens, "Composable and Predictable Dynamic Loading for Time-Critical Partitioned Systems", Accepted to appear at DSD, 2014.

[3] ***Andrew Nelson***, Ashkan Beyranvand Nejad, Anca Molnos, Martijn Koedam, Kees Goossens, "CoMik: A Predictable and Cycle-Accurately Composable Real-Time Microkernel", Design, Automation & Test in Europe, 2014.

[4] Sven Goossens, Benny Akesson, Martijn Koedam, Ashkan Beyranvand Nejad, ***Andrew Nelson***, Kees Goossens, "The CompSOC design flow for virtual execution platforms", Proceedings of the 10th FPGAworld Conference, pp. 7, 2013.

[5] ***Andrew Nelson***, Benny Akesson, Anca Molnos, Sjoerd te Pas, Kees Goossens, "Power versus quality trade-offs for adaptive real-time applications", Embedded Systems for Real-time Multimedia (ESTIMedia), 2012 IEEE 10th Symposium on, pp. 75-84, 2012.

[6] ***Andrew Nelson***, Anca Molnos, Ashkan Beyranvand Nejad, Davit Mirzoyan, Sorin Cotofana, Kees Goossens, "Embedded Computer Architecture Laboratory: A Hands-on Experience Programming Embedded Systems with Resource and Energy Constraints", Workshop on Embedded and Cyber-Physical Systems Education, Published, 2012.

[7] ***Andrew Nelson***, Orlando Moreira, Anca Molnos, Sander Stuijk, Ba Thang Nguyen, Kees Goossens, "Power minimisation for real-time dataflow applications", Digital System Design (DSD), 2011 14th Euromicro Conference on, pp. 117-124, 2011.

[8] ***Andrew Nelson***, Anca Molnos, Kees Goossens, "Composable power management with energy and power budgets per application", Embedded Computer Systems (SAMOS), 2011 International Conference on, pp. 396-403, 2011.

[9] Jude Ambrose, Anca Molnos, ***Andrew Nelson***, Sorin Cotofana, Kees Goossens, Ben Juurlink, "Composable local memory organisation for streaming applications on embedded MPSoCs", Proceedings of the 8th ACM International Conference on Computing Frontiers, pp. 23, 2011.

[10] ***Andrew Nelson***, Andreas Hansson, Henk Corporaal, Kees Goossens, "Conservative application-level performance analysis through simulation of MPSoCs", Embedded

Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on, pp. 51-60, 2010.

[11] Anca Molnos, Jude Angelo Ambrose, ***Andrew Nelson***, Radu Stefan, Sorin Cotofana, Kees Goossens, "A composable, energy-managed, real-time MPSoC platform", Optimization of Electrical and Electronic Equipment (OPTIM), 2010 12th International Conference on, pp. 870-876, 2010.

Workshop Paper **(without published proceedings)**

[1] Benny Akesson, Sander Stuijk, Anca Molnos, Martijn Koedam, Radu Stefan, ***Andrew Nelson***, Ashkan Beyranvand Nejad, Kees Goossens, "Virtual platforms for mixed time-criticality applications: The CoMPSoC architecture and SDF3 design flow", Proceedings of workshop on Quo Vadis, Virtual Platforms, 2012.

APPENDIX F

Samenvatting

Composable en Voorspelbaar Vermogen Beheer

De functionaliteit van embedded systems neemt voortdurend toe. De rekenkracht van de embedded systems groeit om aan deze vraag te kunnen blijven voldoen, terwijl embedded multiprocessor systems steeds normaler worden. De beperkingen van embedded systems zijn niet altijd gerelateerd aan de grootte van de chip, maar zijn veelvuldig een gevolg van beperkingen in energie en/of vermogen. Hoewel het mogelijk is om een krachtiger MPSoC te gebruiken, is het niet altijd mogelijk om een energie- of vermogen bron te verschaffen, die de vraag aankan binnen de volume- en gewichtseisen van het apparaat. Door “power management” middels DVFS is het apparaat in staat om op minder dan zijn maximum spanning en frequentie te draaien, wat ruimte laat voor een groot rekenvermogen indien nodig, met behoud van vermogen op andere momenten.

Embedded systems hebben vaak een real-time functionaliteit. Een real-time applicatie heeft een bijbehorend formeel model om te controleren of het voldoet aan de timing vereisten. Dit formele model wordt gebruikt om een “worst-case” tijd analyse uit te voeren om er voor te zorgen dat de applicatie aan de vereisten voldoet. Deze modellen bevatten de worst-case timing van de berekening en communicatie van de applicatie. Met timing veranderingen als gevolg van power management moet ook rekening worden gehouden, wat het verificatie proces bemoeilijkt.

De drive voor steeds verdergaande functionaliteit heeft geleid tot "mixed time-criticality systems", waarin meerdere applicaties met verschillende timing criticalities dezelfde hard-

ware delen. Dit compliceert het verificatie proces verder, aangezien rekening moet worden gehouden met de verstoring van de timing door de gedeelde resources. Een monolithische verificatie actie is dan ook traditioneel vereist na systeem integratie en deze moet opnieuw worden uitgevoerd als er aanpassingen zijn gedaan die de timing van applicaties beïnvloeden.

De probleemstelling die we willen oplossen in dit proefschrift is om *real-time applicaties zelfstandig te laten executeren en power management uit te laten voeren zonder hun timing vereisten te overtreden of de timing verificatie van gelijktijdig uitvoerende applicaties ongeldig te maken*.

Om dit probleem op te lossen, introduceren we de Composable and Predictable Microkernel (CoMik) om processors composable en voorspelbaar te virtualiseren. Deze virtuele processors kunnen elkaars timing met geen enkele cyclus verstoren, wanneer ze worden gebruikt in combinatie met composable en voorspelbare geheugen controllers en interconnect (zoals verschaft door het CompSOC platform). Als hetgeen dat draait op de virtuele processors (bv. een OS of een applicatie rechtstreeks) een real-time vereiste heeft, kan het, onafhankelijk van hetgeen dat draait op gelijktijdige virtuele processors en virtuele resources, worden geverifieerd.

Om formeel analyseerbare applicatie uitvoering mogelijk te maken, dragen we het Predictable Operating System (POSe) bij dat dataflow applicaties in staat stelt om te worden uitgevoerd op een (gevirtualiseerde) processor. Onze bijdrage is een gecombineerde applicatie en platform dataflow graaf, inclusief een algoritme om dit proces te automatiseren. Wanneer het wordt toegelicht met de worst-case timing, wordt de gecombineerde applicatie en systeem graaf gebruikt om te verifiëren dat de applicatie zijn timing vereiste haalt.

Als de applicatie beter presteert dan vereist (bv. wanneer de input of platform gedrag beter zijn dan "worst case"), kan de prestatie naar beneden worden bijgesteld door middel van DVFS om een vermindering in het energieverbruik te realiseren. Wij contribueren een off-line convex optimalisatie, die een gecombineerd applicatie- en platform dataflow model gebruikt om statische run-time frequentie niveaus af te leiden en een laag energieverbruik te bereiken. De off-line techniek kan statische “slack” (speling) benutten in het schema, maar geen dynamische run-time slack als gevolg van variatie in de taak uitvoering tijden. Voordat de dynamische slack kan worden gebruikt, moet het eerst kunnen worden waargenomen. Voor dit doel verschaft CoMik onafhankelijke vermogen, energie en tijd verantwoordingen per virtuele processor. Dit houdt in dat elke virtuele processor individuele vermogen en energie budgetten toegewezen kan krijgen en aan POSe applicaties kunnen timing budgetten worden toegewezen. Onze bijdrage is een beschrijving en een model voor de verdeling van de energie- en vermogen budgetten tussen meerdere virtuele processors, zodat wat er draait op de virtuele processor composable onafhankelijk power management kan uitvoeren zonder het vermogen te beïnvloeden van de andere virtuele processors om hun hele budget allocatie te gebruiken.

Met behulp van CoMik’s verantwoording infrastructuur, laten we ook zien hoe de kwaliteit van de applicaties dynamisch kan worden afgestemd om te helpen voldoen aan

de timing, energie of vermogen vereisten. Verder contribueren wij een gedistribueerd dynamisch power-management beleid dat het mogelijk maakt om dataflow applicaties, die worden afgebeeld op meerdere (virtuele) processors, observaties te laten maken over gedistribueerde dynamische slack en lokale power-management beslissingen.

We laten de toepasbaarheid van de gepresenteerde technieken zien op een geïmplementeerd Field Programmable Gate Array (FPGA) prototype van een CompSOC hardware platform instantie, met behulp van een H.263 decoder als een case studie applicatie. We tonen aan dat onze technieken niet alleen in theorie werken, maar ook dat ze toepasbaar en geïmplementeerd zijn.