FPGA platform for emulation of composable and predictable MPSoC power management

She, D.

Award date:
2009

Eindhoven University of Technology

MSc Thesis

FPGA Platform for Emulation of Composable and Predictable MPSoC Power Management

Dongrui She

Advisors:
MSc. Aleksandar Milutinovic
Prof.dr. Kees Goossens
Prof.dr. Henk Corporaal

Technische Universiteit
Eindhoven
University of Technology

founded by Philips

September 2, 2009

Abstract

As the IC technology is advancing quickly, Multi-processor System-on-Chip (MPSoC) becomes the trend of embedded System-on-Chip (SoC) design. A typical modern MPSoC for embedded systems usually runs multiple applications, some of which have real-time requirements. To meet the real-time requirements, predictable system is needed. The interference between applications in the MPSoC usually result in exponential increase in design and verification complexity. To avoid such situations, we need composable system, on which the temporal behavior of different applications does not depend on each other. In modern IC design, especially for embedded systems, power consumption is becoming a major constraint, which means power management is essential for MPSoC. Hence we need composable and predictable MPSoC with power management capability.

In this thesis, the design and implementation of an FPGA-based composable and predictable platform for emulating the power management of MPSoC are presented. The platform consists of multiple MicroBlaze processor cores and Æthereal Network on Chip (NoC) is used as the interconnection. The platform enables the sharing of different resources, including the processer, among different applications in a composable way. The support for data-flow application in the platform hardware and software enables the predictable execution of streaming applications. And with the power management hardware and software infrastructure, composable and predictable power management for each application is possible. The support for power management in data-flow application is implemented and demonstrated in the experiments, which saves up to 50% energy compared to the trivial power management policy.

Acknowledgements

The nine months time in the SoC Architectures and Infrastructure research group of the NXP Semiconductors has given me a great opportunity to learn and experiment the research in computer architecture. First, I would like to thank my supervisors at NXP Semiconductors, Kees Gossens and Aleksandar Milutinovic, for their great support and interest in my work. I would also like to thank Henk Corporaal, my supervisor at the university, for providing me with valuable feedback and support. Finally, I would like to thank the other members in the group, especially the students, for the support and the good time I had during my project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A typical chip for an embedded system, i.e. a system-on-chip (SoC) usually integrates different intellectual property (IP) components.

As technology advances rapidly, more complex IPs are integrated to a single chip. Embedded systems, especially the ones in consumer electronics business usually have strict requirements on cost efficiency, power efficiency, and time to market. To fulfill these requirements, modern SoCs tend to integrate multiple programable processing cores such as general purpose processors and DSPs which run software, along with other resources like a external memory controller, resulting in multi-processor system-on-chip (MPSoC).

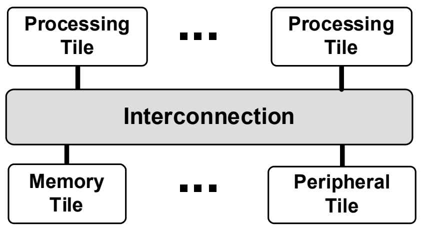


Figure 1.1: A typical architecture of an MPSoC

Fig. 1.1 shows a typical architecture of an MPSoC. The processing tile (PT) consists of a programable processing core, typically a processor or DSP, and local resources that are tightly coupled with the core. An MPSoC has multiple PTs, and other resources such as memories, and these elements communicates with each other via an interconnection infrastructure. To fully utilize the computation power of such a system and achieve cost and power efficiency, multiple applications, some of which have real-time requirements, are mapped on the system. Applications mapped on the system can be independent, and yet they still share some resources.

1.1 Challenges in MPSoC Design

The technology development introduces a lot of challenges in the MPSoC design. In this section we will discuss the key challenges in MPSoC design.

1.1.1 Verification Complexity and Composability

With the increasing integration level in MPSoC, the complexity of verification is growing for both software and hardware. Interference between different components of the system, which may come from different vendors, increases the complexity of design and verification. If this interference is not handled properly, the complexity grows exponentially as more IPs and software are integrated in the system.

To keep verification complexity linear, we want to avoid the unintended interference between components in the system. To be specific, the applications in the system should not influence each other unless we want it. A system with such property is called a *composable* system. In such a system, hardware and software components can be designed and verified independently, therefore the complexity of verification is linear in the number of applications and components.

1.1.2 Predictability

In embedded systems, a lot of applications have real-time requirements, e.g. video player should be able to deliver result at a certain frame rate. To meet such requirements, the behavior of the system should be bounded in temporal domain, i.e. it should be *predictable.*

1.1.3 Power Efficiency

Power consumption has become the major concern in the modern IC design. For embedded systems, power efficiency is even more important as many systems are powered very limited power supply, e.g. batteries, solar panels. ICs in embedded systems need to have power management capability in order to achieve power efficiency.

1.1.4 Simulation-based Verification

The design of an IC has to be verified before it enters the fabrication process. Typical verification methods include formal verification and simulation-based verification.

Current IC verification practices depend heavily on simulation. For the verification of MPSoC, it usually requires hardware-software co-simulation. Software-based simulators have limited simulation speed, and the situation gets worse as the number of processor cores keeps increasing. This results in a situation where the verification process is becoming a dominant part of development in terms of both time and money.

By using Field Programable Gate Array (FPGA) for emulation of the SoC, we can achieve a speed that is very close to real system, and yet the platform has the flexibility similar to software simulation.

1.2 Problem Description

Hansson et al. [23] introduce CoMPSoC, a predictable and composable MPSoC template, which removes all interference between applications through resource reservations. In [19], the design of CompOSe, an RTOS that enables composable sharing of processor between applications is presented. These works give a good template for designing a composable and predictable MPSoC. However, the power management is not addressed in these works.

The goal in this thesis is to design an MPSoC platform that extends CoMPSoC and CompOSe with power management capability, and define an FPGA-based emulation platform based on it. The follows have to be done:

- Define a hardware and software infrastructure for power management in a composable and predictable MPSoC platform, which is targeting streaming applications.
- Design and implement an FPGA-based platform for the emulation of composable and predictable power management on the MPSoC platform.

1.3 Requirements

What we want to achieve in this thesis is to design and implement a platform with the following properties:

- The MPSoC platform is a scalable platform targeting streaming applications.
- Multiple applications, which may have real-time requirements, run on the same processor in a composable way.
- The emulation platform has the hardware infrastructure to support the emulation of power management, and together with the software system, it is possible to perform composable and predictable power management.
- The speed of the emulation is sufficient to run actual applications.
- The observability of the emulation platform is sufficient for verification and experiments for the applications.

1.4 Contributions

In the work of this thesis, we achieve the follows:

- Hardware and software infrastructure for power management in a composable and predictable MPSoC platform based on the CoMPSoC template.
- Composable two-level scheduling on each processor with task budgeting interface and slack management.
- Composable and predictable power management for data-flow applications.
- Improve inter-processor communication by DMA controller on processing tile.
- Monitor for gathering trace data of the FPGA emulation.

1.5 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2, the concepts of different aspects of the platform are introduced, as well as the the proposed architecture and methodology. Then we start to describe the details in the design of the platform, the hardware system design is in Chapter 3, followed by the software system design in Chapter 4. Next the experiments we have done are shown in Chapter 5. Related work is presented in Chapter 6 and the thesis ends with conclusions and future work in Chapter 7.

Chapter 2
Basic Concepts and Methodology

The design of a power efficient, composable and predictable MPSoC involves satisfaction of trade-offs in a lot of different aspects. This chapter gives the description of the important concepts. Section 2.1 gives the definition of composability and explains its necessity in modern MPSoC design. Another orthogonal aspect, the real-time requirement of a system and the predictability are discussed in Section 2.2. Thereafter, in Section 2.3, the power model of CMOS circuit and the basic concepts of power management are introduced. Next in Section 2.4, the interconnection of MPSoC is discussed and the network-on-chip is proposed as the solution. The programming model for streaming application in this platform, the data-flow graph model, is introduced in Section 2.5. Then we discuss the importance of FPGA emulation in modern MPSoC verification in Section 2.6. Finally, the architecture of a platform is proposed in Section 2.7, as well as the design flow.

2.1 Composability

An application is a set of communicating tasks, which provides a certain function for the system.

With the advance in technology, it is common to have a number of independent applications running on one system instead of physically independent systems, e.g. a smart-phone normally has the functionalities which used to be provided separately by a cell-phone, a digital camera and a PDA.

The integration of different applications in one system means that the level of resource sharing is increased. A system is composable if there is no interference between applications, which means the behavior of an application is not influenced by the presence or absence of other applications. In such systems, resources allocated to an application can be seen as a virtualized platform that only runs this application. Thus the application can be designed and verified separately. Fig. 2.1 gives an example of such a system, three applications run at the same platform, and each of them has a virtual platform that only runs the application. To achieve composability, both hardware and software resources have to be shared in a composable way. In this work we focus on the composability in the temporal domain, similar to [29].

Traditionally, composability is achieved by avoiding sharing resources, e.g. in the automotive industry, each function is offered by a separate processor and dedicated resources, such systems are common in both automotive and aerospace industry. Apparently, this solution is too expensive, and not energy efficient for most embedded systems, especially for the ones in consumer electronic business. For such systems, resource sharing is necessary, and it brings up the problem of composable scheduling for hardware and software resources.

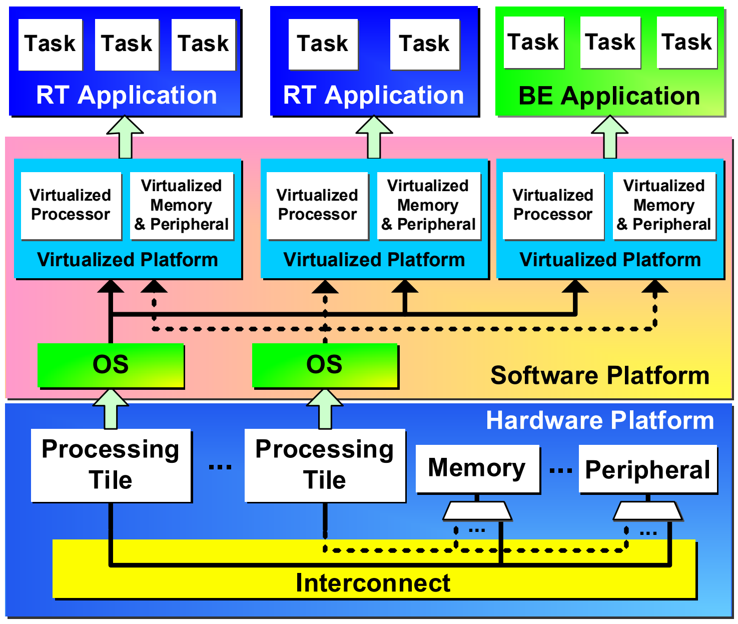


Figure 2.1: Composable system

2.2 Real-time Applications and Predictability

An application is said to be real-time if the correctness of its operation depends not only upon its functional correctness, but also upon the time in which it is performed.

The real-time requirement of an application is usually defined by deadline. Applications can have different kinds of real-time requirements, namely, hard real-time (HRT), firm real-time (FRT) and soft real-time (SRT) [13].

The main difference of different real-time requirements is the effect of deadline misses. For hard real-time applications, the misses of deadline can lead to critical failure and therefore are unacceptable, e.g. the flight control system of an airplane. Firm real-time applications have similar real-time requirements as hard real-time applications, except that deadline misses make the result useless but do not cause critical failure, e.g. an audio playback device. Soft real-time on the other hand, allows deadline misses as long as they are not too often. Most multimedia applications have soft real-time requirements, which are often specified in probability terms.

In this work we call an application a real-time (RT) application if it has such requirement, otherwise we call it a best-effort (BE) application. The real-time requirement is defined in terms of throughput and latency requirements.

2.2.1 Predictability

A system is predictable if its behavior is bounded in the temporal domain, i.e. it provides guaranteed *useful* lower bounds on performance [8]. It is essential for a system running real-time application to be predictable in order to meet the real-time requirements. To achieve predictability, both the hardware and the software should be able to provide predictable service to the applications.

2.3 Power Management

In this section, the power model of CMOS circuit is introduced, and the techniques for power management for such circuits are discussed.

2.3.1 Power Consumption Model of CMOS digital circuit

In a CMOS digital circuit, the power dissipation can be calculated by (2.1), which consists of two parts, dynamic power dissipation and static power dissipation.

$$P = P_{dynamic} + P_{static} = \alpha C V_{DD}^2 f + I_{leak} V_{DD} + I_{SC} V_{DD} \tag{2.1}$$

Dynamic Power Consumption

The dynamic power consumption of CMOS circuit can be modeled by the charging and discharging of capacitance, and it is determined by (2.2),

$$P_{dynamic} = \alpha C V_{DD}^2 f \tag{2.2}$$

where α is the activity rate of the circuit, i.e. the probability that a power consuming transition happens in the system in every clock cycle. C is the capacitance of the circuit, V_{DD} is the supply voltage and f is the frequency of the circuit.

Static Power Consumption

The static power consumption is determined by leakage current, the short circuit current and the supply voltage, as (2.3).

$$P_{static} = I_{leak} V_{DD} + I_{SC} V_{DD} \tag{2.3}$$

In traditional IC design, static power consumption is considered insignificant. However as IC process technology continues advancing towards smaller feature size, static power is becoming increasingly important. It is believed that static power may become the dominant factor in IC power consumption, especially for the deep sub-micron technologies below $65nm$.

This work focuses on the power management of processors in active mode, in which the dynamic power consumption is the dominant factor.

For static power consumption, a lot of technologies in circuit design and IC process provide choices to reduce the static power, but they are beyond the scope of this work, so we do not go into the details of them.

2.3.2 Power Management Techniques

According to (2.2), to reduce the dynamic power consumption, we should reduce the follows: C, α,f and V_{DD}.

C and α are mostly determined by the circuit design and the type of the application. Obviously, the best way to reduce the dynamic power consumption is to lower the supply voltage V_{DD} as it can result in quadratic power reduction. However, reducing the supply voltage increases the circuit delay, which can be estimated by (2.4), where τ is the propagation delay of a CMOS transistor, V_T is the threshold voltage and V_G is the input gate voltage [14].

$$\tau \propto \frac{V_{DD}}{(V_G - V_T)^2} \tag{2.4}$$

From (2.2) and (2.4), we can see there is a trade-off between power consumption and the delay. The curve in Fig. 2.2 is the typical power-delay curve, the exact shape and value depend on the process and circuit design. The delay of a circuit determines the maximum frequency it can run at. For a digital computing system such as a processor, this is a trade-off between power consumption and computational performance.

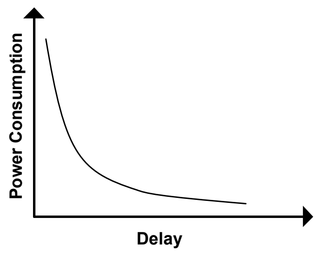


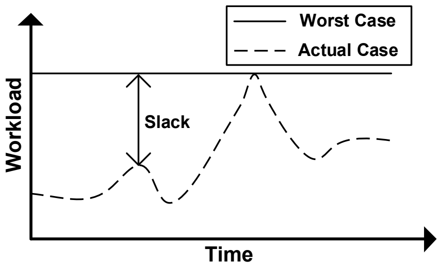
Figure 2.2: Power-delay curve

Another technology for saving dynamic power in digital system is clock gating, which disables the clock when the circuit is idle but the idle time is not long enough to shut down the circuit. It changes the f in (2.2) to 0 and saves the dynamic power.

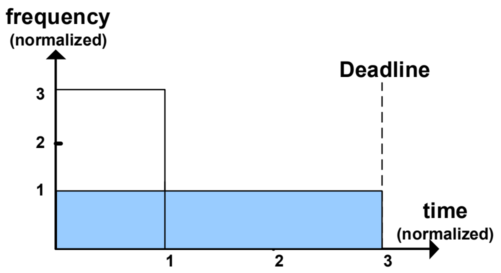
For static power consumption, most of the techniques of reducing static power consumption are at circuit level. One of the techniques is adaptive body biasing (ABB) [44], which can reduce the leakage current. Another technology that is also popular is power gating, which completely shuts down part of the circuit and is able to save a lot of energy, but the overhead is high and it is useful only when the power-off time is long enough.

2.3.3 Architectural Level Power Management

As discussed in Section 2.3.2, the most effective way to save dynamic power is to lower the supply voltage. The trade-off between power consumption and performance becomes the most important problem. Deciding the frequency and supply voltage statically at design time clearly limits the possibility of power saving for a processor based system because the workload of a processor varies according to the software it runs. For real-time applications, the static frequency-voltage point is determined by the worst case workload which usually results in significant over-estimation in the actual execution. At this point, one can see that dynamic adjust of the frequency and supply voltage based on runtime information would be a good solution to this problem for microprocessor, i.e. the dynamic voltage and frequency scaling (DVFS) technique [26, 41].



(a) Slack Example



(b) Use slack to set a lower frequency

Figure 2.3: An examples of DVFS

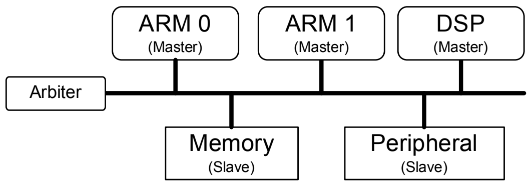
The basic idea of DVFS is to dynamically adjust the processor's frequency so it runs at a lower supply voltage. For hard or firm real-time applications the problem here is to make sure the task still finishes in time. Assuming that the workload of a task is defined by the number of cycles it needed to finish, and there is is a deadline for this task, a number of cycles are allocated to this job according to the worst case workload and the deadline. In reality the actual workload often varies, the difference between the worst-case and actual-case workload is called slack, as Fig. 2.3a shows. If the processor is able to detect slack generated at runtime, it can use it to lower the frequency. The task produces the correct result at the right time as long as the task gets the required number of cycles before the deadline. Fig. 2.3b gives an example of DVFS using slack, the job is given 3 time units to finish its work but at the maximal frequency 3 it only needs 1 time unit, so it can lower the frequency to 1 which allows a lower supply voltage, and can fully utilize the time assigned to it. It has been proven that the just-in-time policy minimizes dynamic power in DVFS [26], i.e. the processor finishes the task just before the deadline. In DVFS for best-effort or soft real-time applications, similar technique can be used, and there are more possibilities since the timing constraints for these applications is not so strict as hard/firm real-time applications. Per-core DVFS is an attractive option for MPSoC power management, but it requires careful consideration of the trade-off between the overhead and the energy saving [28].

For static power consumption, power gating is a practical choice at architectural level. Some studies also suggest use ABB at the architectural level in order to reduce static power consumption [35]. At application level, most of these techniques do not show significant effect at runtime, and some of them can be simulated by the same infrastructure as the DVFS simulation, e.g. the power gating.

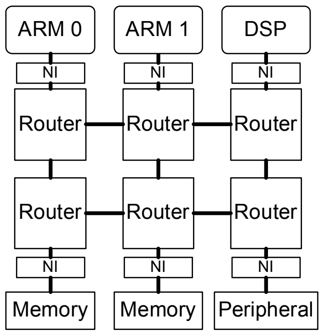
In this work we assume the techniques described above are used, and we focus on per-core dynamic power management at the architectural level using DVFS on an MPSoC.

2.4 Network on Chip

When multiple applications are integrated on the same system, resource sharing between applications is inevitable as different elements in the system need to communicate and resources need to be reused.



(a) Shared Bus



(b) Network-on-Chip

Figure 2.4: Examples of interconnection based on shared bus and network-on-chip

In traditional SoC architecture, shared buses such as AHB [4] and PLB [24] are widely used for solving this problem. In such an architecture, IPs that need to access resources are masters on the bus, and the resources are slaves on the bus, as shown in Fig. 2.4a. For shared buses, at any given moment, only one master has the control of the bus and is able to access one of the slaves. Concurrent requests are scheduled by the bus arbiter.

The shared bus is simple and cheap, but it is not scalable at either architectural or physical level. At the architectural level, a shared bus is not scalable as it is shared by all masters, which becomes a central bottleneck, causing severe performance losses when number of masters increases. At physical level, the structure of shared bus results in long global wires, which causes large delay and easily becomes the critical path in the system implementation.

Networks on chip (NoC) [18] are the solution for this problem. A typical NoC consists of a number of routers and links between them, see Fig. 2.4b. IPs are connected to the network through a network interface (NI). Data is routed through routers in the network. The data transfer in a NoC uses only local wires between routers, which provides scalability at physical level. Additionally, NoC makes it easier for IPs to run at their own clock domain compared to shared bus. It enables globally asynchronous locally synchronous (GALS) system design [32], which eases the design and provide more possibility for reducing power consumption. At the architectural level, NoC handles concurrent transactions and can be easily pipelined, therefore it does not become the central bottleneck. A NoC can provide high bandwidth for multiple IPs in the system by properly allocating resources.

NoCs achieve a better trade-off than traditional interconnection architectures [3, 11]. Crossbar and multi-layered or hierarchical bus structure have been used to improve the scalability of shared buses, but due to the nature of bus, these interconnection cannot completely solve the problem, and a NoC would be the most reasonable choice of interconnection for large scale MPSoC [3].

As stated earlier in this chapter, to build a composable and predictable system, composable and predictable services should be provided by each component in the system. For NoC this is not trivial as its structure is more complex. The Æthereal [20] architecture addresses this issue by supporting time-division multiple access (TDMA) of the network components across different connections, i.e. virtual wires. With the TDMA time slots allocated to match the application requirements, Quality of Service (QoS) guarantees is achieved, and the guaranteed service connections are composable [21, 22]. The Ætheral is chosen to be the interconnection of the platform in this work. More details are given later in Chapter 3.

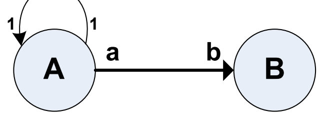
2.5 Data-flow Graph Programming Model

Data-flow graphs are often used for modeling DSP applications and designing streaming multimedia applications [34]. The platform in this work supports different types of data-flow graph model, and we focus on the synchronous data-flow graph [33].

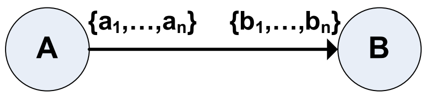
A synchronous data-flow (SDF) graph models tasks as nodes in the graph, called *actors*. The communication channels between tasks are represented by *directed* edges in the graph. The actor execution in SDF graph is called *firing*, incoming edges to an actor represent the input data needed by the actor in one firing, while outgoing edges represent the output data produced by the actor in one firing. Fig. 2.5a is an example of SDF graph with two actors and one communication channel.

It is clear that the edges in the SDF graph represent the data dependence between actors. The communications in the SDF graph is done in first-in-first-out (FIFO) channels, the amount of data transferred is measured in *tokens*. The number of tokens an actor produces or consumes in one firing is called *rate*, as a and b in Fig. 2.5a, for actor A and actor B, respectively.

In SDF graph, whether an actor is allowed to fire or not, depends only on the status of the FIFOs connected to it. If all incoming FIFOs have enough data and all outgoing FIFOs have enough free space, the actor is allowed to fire once, and the tokens are consumed according to the rate of the FIFOs. A self-edge in SDF graph is used to model *auto-concurrency*, i.e. then maximum number of simultaneous executions the actor, e.g. actor A in Fig. 2.5a cannot start a new execution until the last one finishes. The SDF graph is monotonic, i.e. starting early does not increase the completion time of the successors of an actor, which is important in scheduling and power management.



(a) Synchronous data-flow graph



(b) Cyclo-static data-flow graph

Figure 2.5: Data-flow graph

There are different kinds of data-flow graphs besides SDF graph. The Cyclo-static data-flow (CSDF) graph model [10] is a similar computation model as SDF graph. The major difference is that instead of having constant value in SDF, the rate of a FIFO in the graph varies according to a pre-defined sequence, see Fig. 2.5b. This modification makes it easier to model applications. CSDF graph can be converted to SDF graph [39], so the analysis techniques for SDF graph still apply. Kahn process network (KPN) proposed by Kahn in [27] is even more expressive, as it allows data-dependency on the rate and actor behavior, on the other hand it makes the analysis more difficult.

In this work we choose the data-flow graph as the programming model for streaming applications. Note that the platform in this work has built-in support for the scheduling and communication of the data-flow graph model, but it does not mean only applications using this model can run on the platform.

2.6 FPGA-based Hardware Emulation and Simulation

As mention in Chapter 1, IC verification practice usually depends heavily on simulation. Software-based simulation has limited simulation speed. The situation gets worse as the number of processor cores in the design keeps increasing. The requirement for hardware-software co-simulation puts even more pressure on the simulator. A typical trade-off in software simulator is to trade speed for abstraction-level. However, even for simple simulators, the speed is quite limited [6, 12].

To correctly capture the behavior of an MPSoC, the simulation has to be detailed, which leads to slow simulation. This results in a situation where the verification process is becoming a dominant part of development, in both time and money.

FPGA-based emulation and simulation platforms are able to perform fast simulation way at low cost. Comparing to a software simulator, an FPGA platform provides a much higher speed, usually more than 1/10 of the real system speed, sometimes even at full speed. Moreover, the FPGA platform is very close to the real IC environment, one can test the design without really producing the chip. With state-of-the-art FPGA technology, the observability of such platforms is comparable to software simulators. The Berkeley Emulation Engine (BEE) [15], and the Research Accelerator for Multiple Processors (RAMP) based on BEE [46] are interesting examples of such FPGA platforms.

The speed of FPGA emulation is a motivation of this work.

2.7 Proposed System Architecture and Design Flow

In this work, we extend the CoMPSoC template in [23], and propose a tiled MPSoC architecture. They system consists of a number of tiles, as Fig. 2.6 shows. There are different types of tiles in the system, e.g. processing tile and memory tile. These tiles are connected by the Æthereal NoC which allows predictable and composable communication between different tiles.

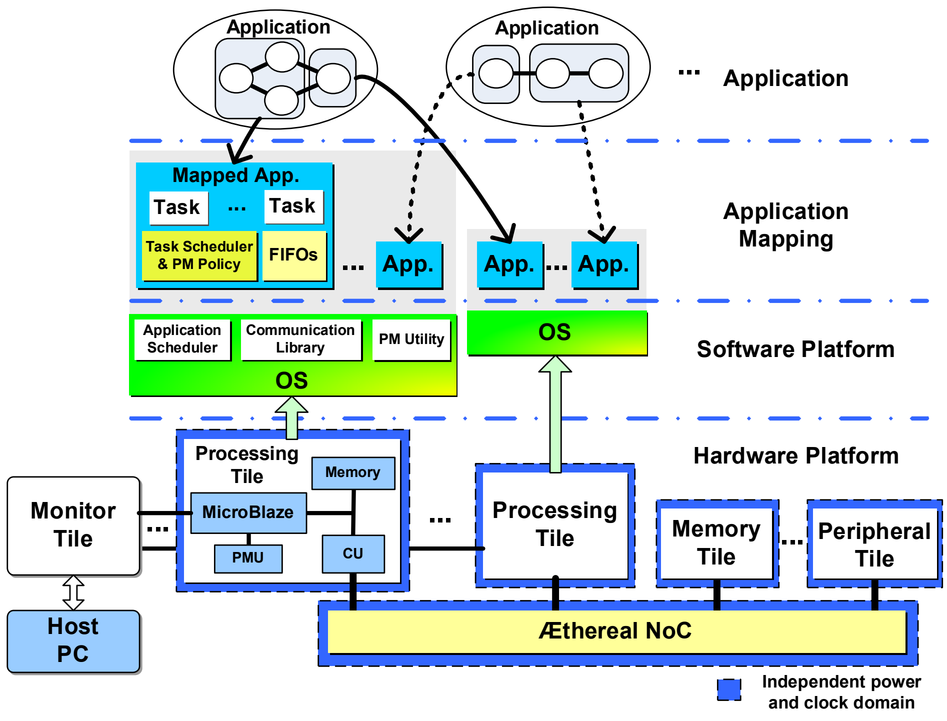


Figure 2.6: Proposed architecture

We focus on the hardware and software design of the processing tile, on which the software applications are running. A processing tile has a MicroBlaze processor with some local memories, a power management unit and a communication unit. On the processor, an operating system (OS), which is an extension of the work in [19], manages the processing tile resources and schedules application in a composable manner. We assume the streaming applications are mapped on the platform using data-flow graph model, which is directly supported by the OS. An application can be mapped to one or multiple processing tiles.

As the system is designed in the globally asynchronous locally synchronous (GALS) style, each processing tile performs the power management independently. This enables the per-tile power management. In this work we design and implement a power management unit (PMU) on each processing tile, which allows the processor to perform DVFS on the tile.

Apart from the processing tiles and resource tiles, a monitor tile is introduced to guarantee the observability of the system. The monitor tile collects data at runtime from each processing tile, and sends the data to the host PC for analysis.

For power management, we assume the PMU on each processing tile is able to generate different frequencies for the tile independently. Although it is difficult or even impossible to implement complete DVFS on an FPGA due to the missing of on-chip voltage scaling infrastructure, we do manage to create a power management unit that enables the processing tile to emulate the behavior of DVFS.

On such a platform, the design flow is simple and effective, see Fig. 2.7. As the components in the system provide composable services, it is possible to design the applications separately. Each application has its own task set, scheduling algorithm and power management policy. The application is mapped on the virtual platform and verified independently, as the composability of the system guarantees that integration of applications does not affect the behavior of an individual application. However, in the current implementation, the composable design flow is still not complete yet, the mapping of different applications are still performed together.

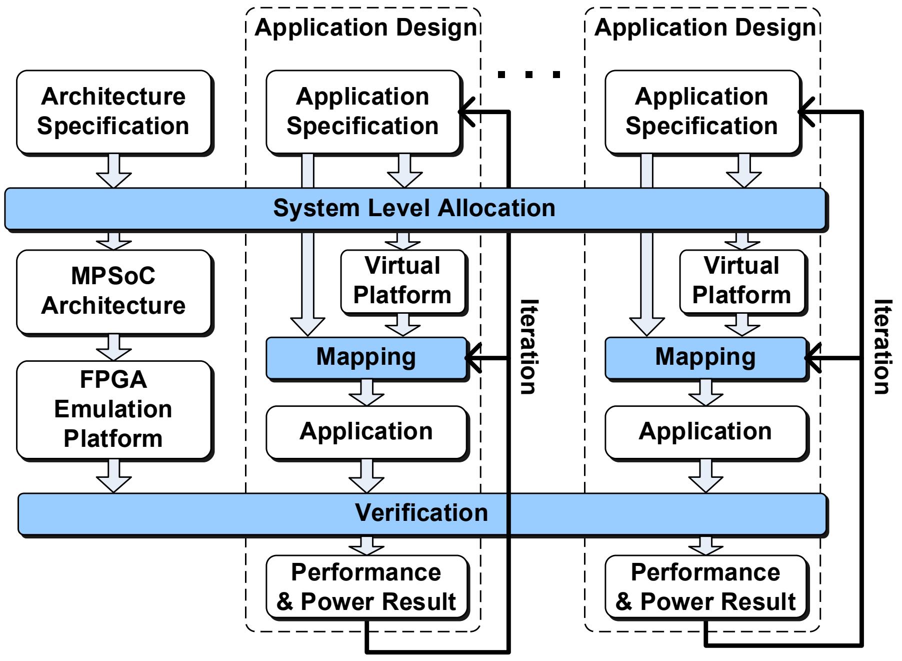


Figure 2.7: Design Flow of the System

2.8 Summary

The basic concepts of the platform are introduced in this chapter. The composability and predictability are two important properties in MPSoC design for embedded system. The power consumption is becoming the major problem in modern IC design, which is also the most important motivation of this work. We discuss the power model and different power management techniques in this chapter. Then the necessity of developing FPGA-based emulation platform is presented. Finally, we show the proposed system architecture, as well as the design flow for such platform. In the remainder of this thesis, the design of this platform is explained in detail.

Chapter 3
Hardware Platform Design

In this chapter, the design and implementation of the hardware modules in the platform are presented. The hardware platform architecture is shown in Fig. 3.1. The system consists of different kinds of tiles and a NoC that connects them.

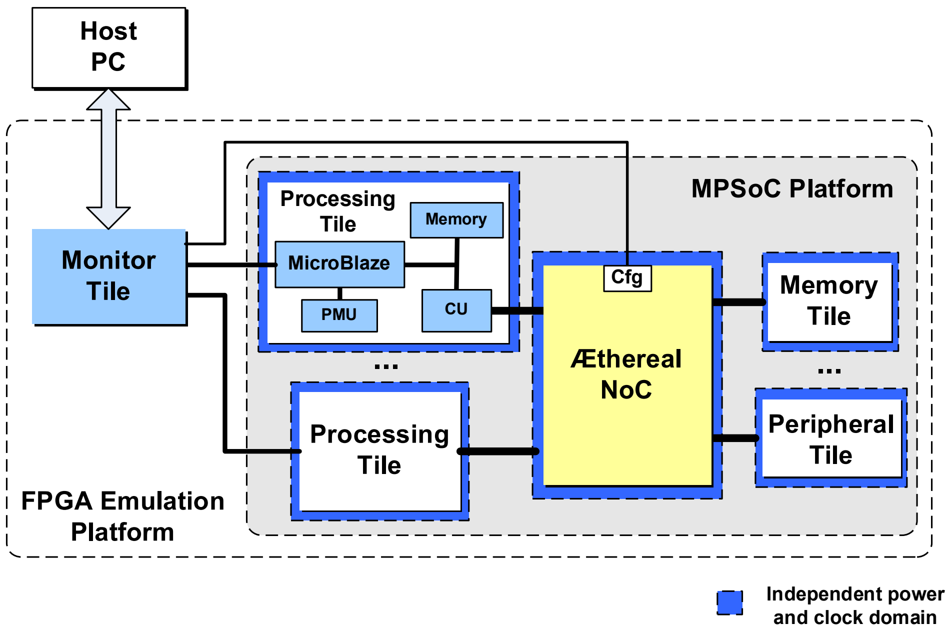


Figure 3.1: Hardware platform architecture

Interconnection of the platform is a key element in a tiled architecture, so we first introduce the Æthereal NoC in Section 3.1. Next, the most important tile in the system, the processing tile is described in Section 3.2, including the power management and communication infrastructure. Then the hardware part of the monitor, which plays an important role in the emulation system, is discussed in Section 3.3, and the system configuration, which is implemented on the monitor tile, is introduced in Section 3.4. Finally, we put everything together to a complete hardware platform in Section 3.5.

3.1 NoC Based Interconnection

The Æthereal NoC is used as the interconnection between different tiles in the system. In this section, we briefly introduce the architecture of the Æthereal NoC, as well as the communication in Æthereal.

3.1.1 Æthereal NoC

The Æthereal NoC consist of two kinds of components, the routers that transfer data among each other and network interface (NI) that are responsible for connecting the IPs to the router network [20, 22].

The NI consists of two parts, the NI *kernel* and the NI *shell*, as shown in Fig. 3.2. The NI shell is used as the bridge between the IP and the NI kernel, which allows the IP to use different protocols to access the NoC, e.g. DTL [40]. The core functions of NI, e.g. the packetization and depacketization, are provided by the NI kernel. In this work, the DTL shell is used to connecting different tiles.

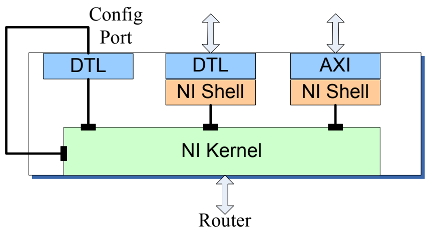


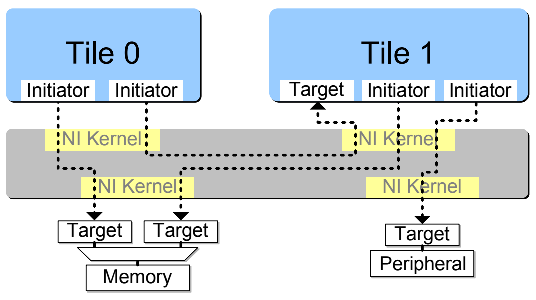
Figure 3.2: Æthereal Network Interface

3.1.2 Connection-based Communication

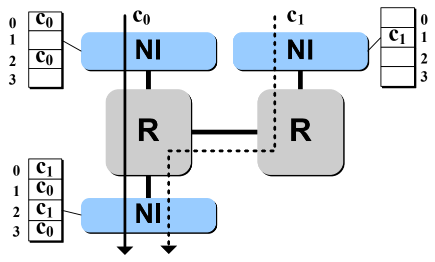
In the Æthereal NoC, communication between nodes of the network is based on connections, which are established by resource reservation as Fig. 3.3a illustrates.

The links between components are resources that are shared with different IPs. Contention on the links has to be resolved properly so the interconnection is composable and predictable. Æthereal solves this problem by time division multiplexing (TDM) scheduling of the links, which results in a contention-free routing [22].

Connections have requirements for the quality of service(QoS), which are characterized by the bandwidth and latency requirements. There is a slot table on each NI. Some slots in the slot table are reserved for a connection base on its QoS requirements. An IP is only allowed to send data to the NoC when the current time slot is reserved for the connection it is using. The slot tables in different NIs are synchronized, so it is possible to allocate the slots such that no contention on the links even happens. A simple example is given in Fig. 3.3b. Two connections c_0 and c_1 need to reach the NI at the bottom and each cycle a datum is transferred through a link. c_0 is allowed to send data at time slot 0 and 2, and c_1 is allowed data at time slot 1, so there is no contention at the left router and the destination NI.



(a) Connections in Æthereal NoC



(b) Contention-free routing

Figure 3.3: Interconnection based on Æthereal NoC

In Æthereal, connections without QoS requirements, i.e. the best-effort connections use the time slots that are not allocated, or allocated but not used.

With this contention-free routing scheme, the interference between any two connections with guaranteed service is eliminated, so the interconnection is composable. And since the routing is based on TDM, the temporal behavior could be easily bounded, i.e. the NoC is also predictable.

In this work, different applications use the composable connections provided by Æthereal to access the remote resources.

3.2 Processing Tile Design

The processing tile is the central element in the system. Apart from running task code, a processing tile also needs to provide the following things:

- Hardware infrastructure that allows software on the processor to perform power management.
- Hardware infrastructure that connects the tile to the NoC, allows software to access remote resource

The architecture of a processing tile is in Fig. 3.4. There are three main components in a processing tile: a MicroBlaze processor core, a power management unit and a communication unit.

The PMU is connected to the processor via FIFO interface which provides clock domain crossing. The communication unit is connected to the local data bus, which could be accessed by the processor using load/store instructions.

To ease the analysis of predictability, we decide not to use the cache in MicroBlaze. Moreover, all communication with off-tile resources is performed by the communication unit. As the platform is targeting streaming applications, such design is efficient. Other configurations of the core are determined according to the application requirement.

Besides the main components, an instruction memory and data memory are also essential parts of the tile, they are connected to the processor via local memory bus (LMB), and the communication unit accesses the data memory using DTL

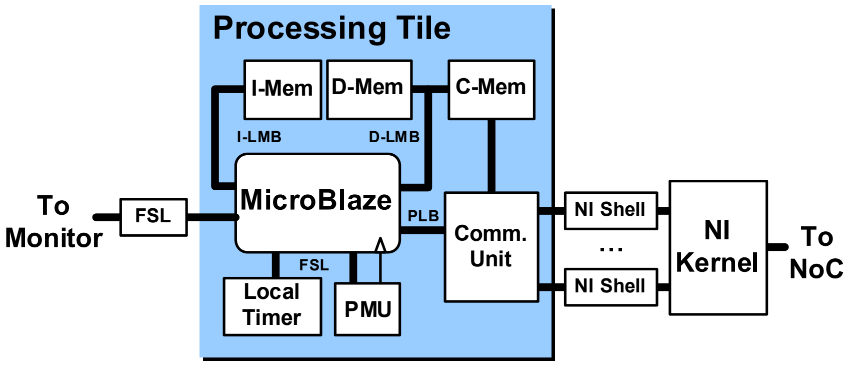


Figure 3.4: Architecture of the processing tile

protocol [40]. The local timer, a counter that counts the local clock, i.e. the output clock of the PMU is also provided, which can be useful for the software. In addition a processing tile may have some local peripherals, depending on the requirements of the applications, e.g. a computation accelerator.

The remainder of this section presents the design and integration of the components in detail.

3.2.1 The MicroBlaze Processor Core

MicroBlaze is a 32-bit processor soft-core provided by Xilinx [49]. The instruction set architecture (ISA) of MicroBlaze is a typical reduced instruction set computer (RISC) ISA.

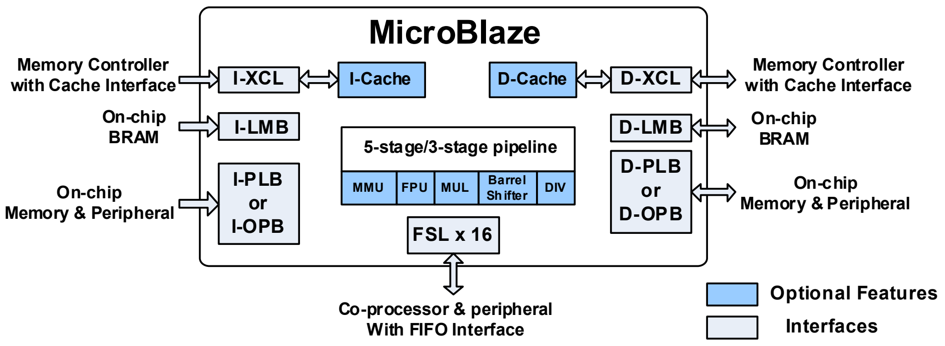


Figure 3.5: MicroBlaze core architecture

The MicroBlaze core is highly configurable, which enables users to keep only needed things in the system. The following parts of MicroBlaze core are configurable,

- Pipeline: 5 stages by default, 3 if optimized for area;
- Arithmetic unit: optional barrel shifter, hardware multiplier, divider, floating point unit (FPU), etc;

- Memory system: configurable memory management unit (MMU), cache and cache-links;

In addition, the MicroBlaze core has different interfaces for communicating with others IPs, including other processors [49]:

- Processor Local Bus (PLB) or On-chip Peripheral Bus (OPB): shared bus for on-chip IPs, available on both data and instruction side. It is part of the CoreConnect of IBM [24].

- Local Memory Bus (LMB): a single-master bus for accessing Block RAM(BRAM) on the FPGA. It is simple and provides zero-wait-state access to local memory. Available on both data and instruction side.

- Fast Simplex Link (FSL) [48]: a simple FIFO protocol. The interface is directly connected to the register file of MicroBlaze and a set of instructions is used to control it. Typically, it is used for co-processor and high speed peripheral.

- Xilinx Cache Link (XCL): a dedicated FIFO interface based on the protocol of FSL. It provides low latency access for the cache controller and reduces the traffic on the bus. Available on both data and instruction side.

In comparison to other synthesizable processor cores, e.g. the LEON and Open-RISC, MicroBlaze achieves a good trade-off between performance and resource utilization on FPGA [36]. By using different configurations, MicroBlaze can be used in a very wide range of cases, from doing simple finite state machine (FSM) duty, to running operating systems that require a full featured processor, e.g. Linux. In the area-optimized configuration, a MicroBlaze core utilizes only about 600 slices in a Virtex2-Pro FPGA [1].

The simplicity of this core makes it easy to analyze. Without complex features such as multiple issue, dynamic branch prediction, and out-of-order execution, it is easy to predict the behavior of a MicroBlaze core, provided the peripherals are also predictable, and this is very important for our system.

Although a simple single-issue in-order core seems to be quite limited in performance, the flexible interfaces of MicroBlaze allow users to extend the processor in different ways, e.g. connecting a computation engine via the FSL interface can achieve similar performance as the multimedia extension in the ISA of some cores. So it is even possible to use MicroBlaze in situations where high computational performance is required.

Configuration for the Processing Tile

In this work, the MicroBlaze can be configured according to the requirement of the application. Typically we have a MicroBlaze processor with 5-stage pipeline, a 32-bit integer multiplier and a barrel shifter. Such configuration consumes about 1200 slices on a Virtex2-Pro FPGA.

3.2.2 Power Management Infrastructure

Power management unit consists of two sub-modules, the system timer and the frequency generator, as shown in Fig. 3.6.

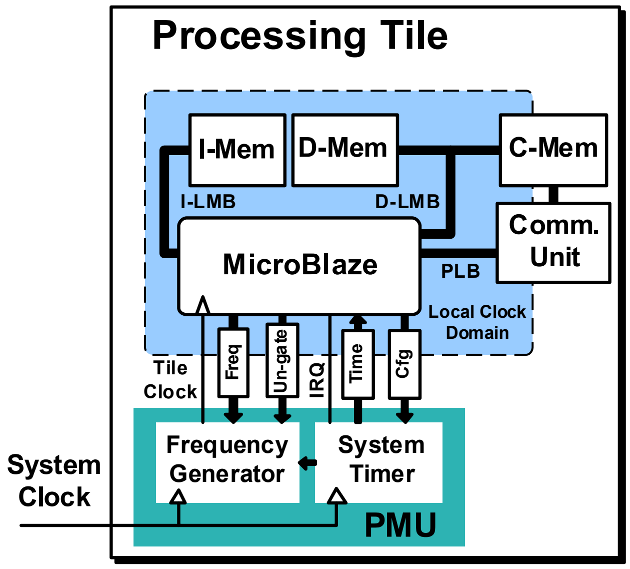


Figure 3.6: Power Management Unit

System Timer

The main purpose of the system timer is to provide a time reference. Here we define *wall time* as the absolute time, which can be measured by number of cycles of a clock. The system timer provides the following services for the system:

- Provide wall time reference to the frequency generator. The system timer is readonly in this case. The purpose is to enable the frequency to work in a predictable way.

- Provides wall time reference to the processor. In this case, the system is a programable timer, which is writable. It provides wall time reference to the processor in two ways: by generating an interrupt request signal (IRQ), and by allowing the processor to read the timer value.

The system timer is implemented by a simple programable counter which counts the global clock signal. In this work it is usually set to a down counter. When the timer value reaches zero, an interrupt request signal is generated if the interrupt of the timer is enabled.

Since the processor may change the operating frequency, it is very important that the system timer is always running at the maximum frequency, which enables it to give the wall time reference to the processor and the frequency generator.

Frequency Generator

Frequency generator is designed for two purposes:

- For setting output frequency to $\frac{n}{d}f_{in}$ at a given wall time, where n and d are programable parameters. This functionality allows the processing tile to set its own operating point.
- For gating the output clock, i.e. set output frequency to 0, and un-gate it at a required moment. This functionality could be used to stop the processor in order to achieve constant execution time for certain programs, and to emulate the idle state, e.g. sleep mode or power gating. As the *halt* instruction is not available in MicroBlaze's ISA, this capability is very useful.

On existing FPGAs, there is no infrastructure for implementing complete DVFS. In particular there is no on-chip regulator available to change the supply voltage at runtime. So the PMU should include a module for emulating DVFS. For emulation, the absence of on-chip voltage regulator is not a hard limit, as the voltage level itself does not have direct impact on the behavior on the application.

For frequency scaling, a few options are available. The partial reconfiguration of the digital clock manager (DCM) on the Xilinx FPGA implements a real frequency scaling. However it introduce a lot of extra overhead such as the relocking of the phase locked loop (PLL), which limits flexibility of the frequency generator, so it is not suitable for our system. Alternatively, the frequency can be generated by fine-grained clock gating, i.e. deciding whether to enable the clock for each cycle. Although no actual frequency scaling is implemented, the processor, as well as the software running on it, get the same number of clock cycles as the ideally scaled clock, but the exact time of the clock pulse is different. The algorithm of generating the frequency is given in Algorithm 1. With this algorithm, the clock pulse is distributed in the most uniform way, i.e. closest to the ideally scaled clock. And it does not cost extra hardware compared to implementations that give all the clock pulses in the beginning of the end of a period.

Algorithm 1: Clock-Generation

```
   Input: $ClkIn$, $N$, $D$,
   Output: $ClkOut$, where $f_{ClkOut} = \frac{N}{D} \cdot f_{ClkIn}$
1  $acc \longleftarrow 0$;
2  for each cycle of $ClkIn$ do
3      $acc \longleftarrow acc + N$;
4      if $acc \geq D$ then
5          $acc \longleftarrow acc - D$;
6          $ClkOut \longleftarrow ClkIn$;
7      else
8          $ClkOut \longleftarrow 0$;
9      end
10 end
```

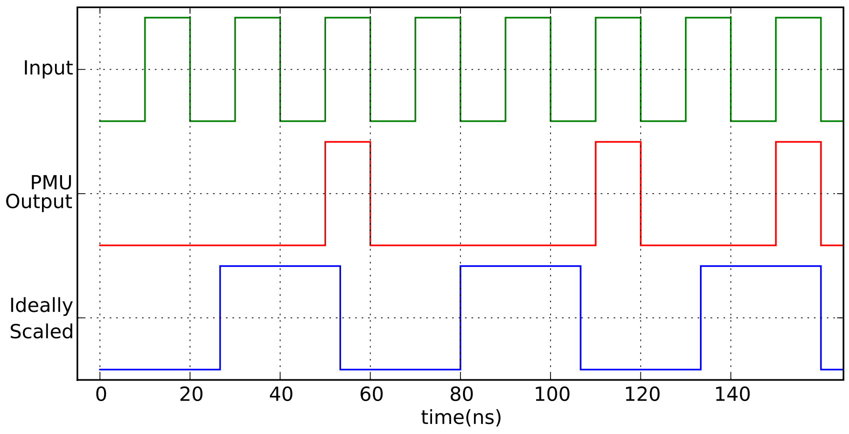


Figure 3.7: Example of $\frac{3}{8}$ Output Frequency

In the implementation, to ease the calculation of the software and allow the processor to set the frequency using only one command word, d is fixed to 16, so the output frequency is $\frac{n}{16} f_{in}$, where n is a programmable parameter.

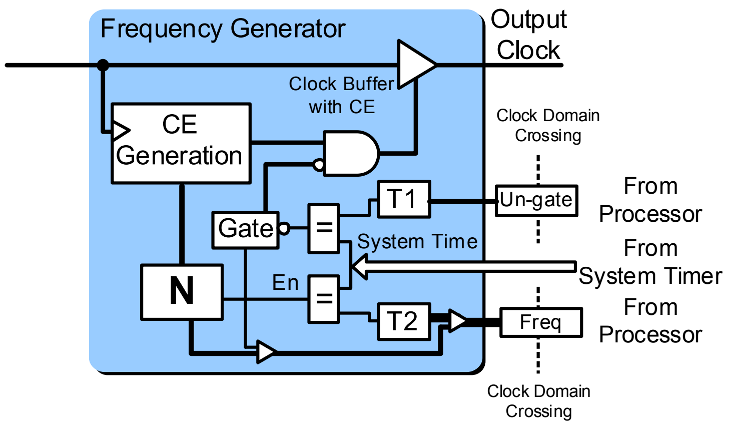


Figure 3.8: Architecture of the frequency generator

Fig. 3.8 gives the architecture of the frequency generator. The clock enable (CE) generation module uses Algorithm 1 to generate the CE signal. The processor can program the frequency parameter N through a FIFO. Note that the processor may run at a different frequency, therefore the latency of the FIFO is not constant. To eliminate the uncertainty caused by the variable frequency and clock domain crossing, a programmable time value register $T2$ is introduced. The parameter N does not affect the CE generation until the system timer reaches the value set in $T2$, as shown in Fig. 3.9. The algorithm in Algorithm 1 works in a periodic way, of which the period is 16. In order to keep the frequency scaling factor valid for the period before the frequency switch, the frequency switch is only allowed when the system timer value is multiple of 16, i.e. the value of $T2$ is aligned with 16.

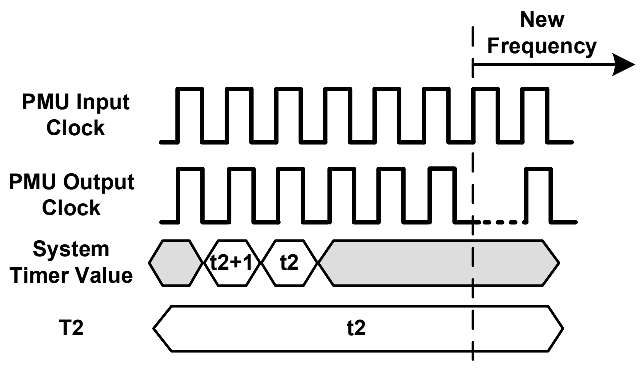
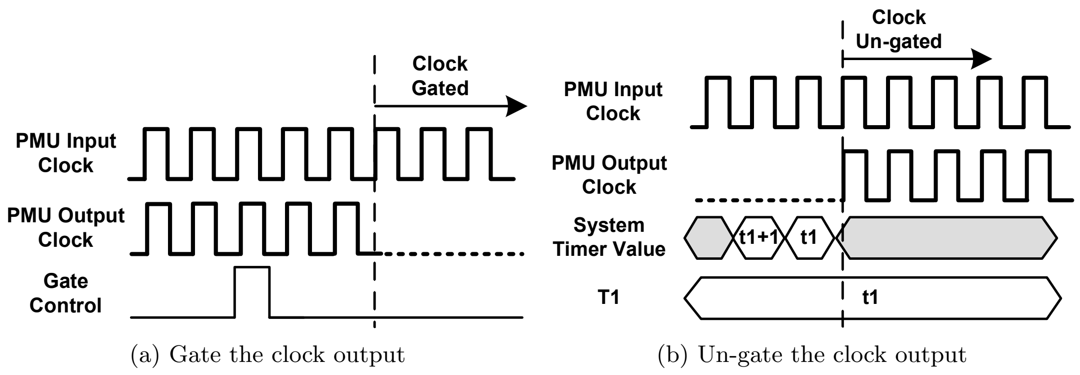


Figure 3.9: Frequency switch in the frequency generator

When a gate command is sent to the frequency generator, it sets the *Gate* register which disables the output clock. When the system time reaches the value of register $T1$, the *gate* register will be reset and the output clock will continue. The timing of gate and un-gate of the clock is shown in Fig. 3.10.



(a) Gate the clock output

(b) Un-gate the clock output

Figure 3.10: Gate and un-gate of the clock

PMU Interface to Processor

There are three input ports and one output port in the PMU interface. In principle they can be mapped to any interface that is accessible for the processor, including a bus. But in order to build a predictable system, it is important to make sure that these ports are mapped to a predictable interface. In our design, we map these ports to the FSL of MicroBlaze, which is fast and more importantly, completely predictable.

As shown in Fig. 3.6, a FIFO from the processor to the system timer is used for configuring the timer, which is also used for clock domain crossing. The timer value is sent to the processor via a synchronizer, which is implemented by two flip-flops.

The processor can perform the following operations on the system timer:

1. Start and stop the timer.

2. Read and/or set the timer value.

3. Enable or disable the timer interrupt.

Note that due to the clock domain crossing, reading of the system timer value is not cycle accurate, and cannot be used directly used as the cycle-level wall time reference. The processor has to compensate for the latency according to the frequency it is running at, or use the time reference in a coarse granularity.

As shown in Fig. 3.6, two FIFOs from the processor to the frequency generator are used for configuring the frequency generator. The processor can perform the following operations on the system timer:

1. Set new frequency and its effective time. Done by sending a 32-bit word via the *Freq* FIFO, the lower 4 bits are the frequency parameter N and the rest are the effective time, which is aligned to 16.

2. Gate the clock output. Done by sending a command word via the *Freq* FIFO with the control bit [48] set to 1. In this case the content of the command is ignored.

3. Set un-gate time. Done by sending the effective time via the *Un-gate* FIFO.

In a system with DVFS, the transition between different frequencies may introduce a state in which the clock is unstable and cannot be used to drive the circuit. Moreover, the frequency switch latency at different frequencies are different, due to the variable latency of the frequency generation and the asynchronous communication. Here we introduce a method to perform predictable frequency switch using the PMU described before. The typical timing of using all three operations to switch frequency is shown in Fig. 3.11. The processor sets the new frequency and un-gate time, then gate the clock. The frequency switching happens when the processor is not running. When the processor comes back, it is already running at the new frequency. The switching in Fig. 3.11 can be used to get the WCET of the frequency switching by setting $T1$ to the proper value.

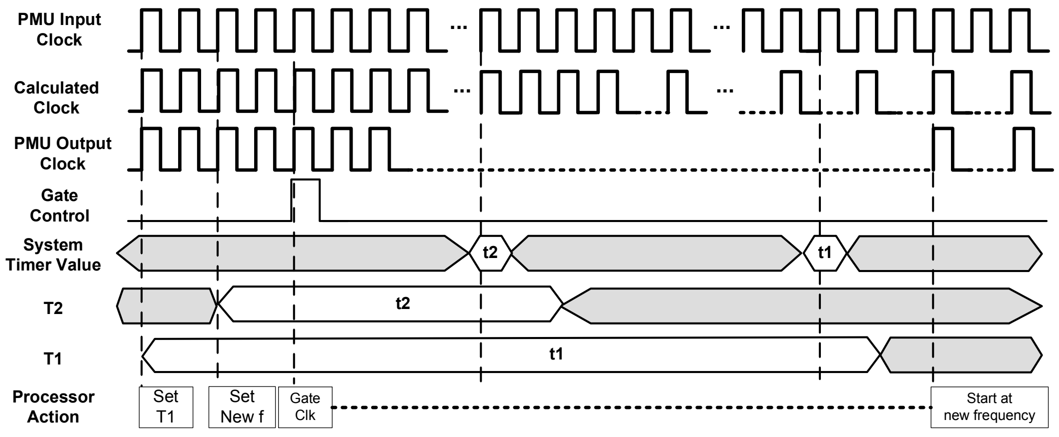


Figure 3.11: Frequency switching with gate and un-gate

3.2.3 Tile Communication Infrastructure

As described in section 3.1.1, the NoC provides connections between nodes in the network. A connection allows the tile processor to access remote resources. Here we assume the connection is between a tile and a memory or peripheral with memory-mapped I/O (MMIO), and both ends of the connection use DTL-MMIO protocol [40]. In such a connection, the processor is the initiator and the remote device is the target. Note that the remote resource for one processor may be the local resource of another processor.

Trade-offs in Designing the Interface to the NoC

To connect the MicroBlaze processor to the NoC, the following options are available:

- Map the remote memory into the processor's memory space, so the processor accesses data on remote location by load/store instructions.

- Use a communication unit that supports Direct Memory Access (DMA), which requires explicit control over the transactions over the NoC.

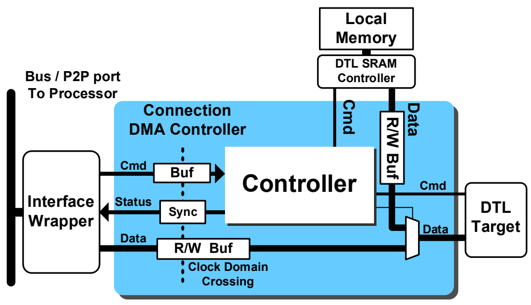
The memory-mapped solution is simple and straight forward. It provides transparent access to remote resources to the processor. The major drawback is that the processor is exposed to the large latency cause by the NoC and/or remote resource, e.g. the access of off-chip DRAM can easily cost hundreds of cycles. For simple processor cores like MicroBlaze, this causes loss in not only performance, but also predictability. Large latency of memory access results in long stalling of the processor pipeline, and consequently significant increases of worst-case interrupt response time, which makes it difficult to derive a useful bond for the system behavior and the predictability of the processor is jeopardized.

The use of DMA controller is a solution to this problem as the use of a DMA controller only requires local accesses. The DMA controller supports block transaction and runs in parallel with the processor, hence there is a potential for getting a better performance than the memory-mapped communication. The downside is that the communication is no longer transparent to the software. For applications that do not decouple communication from computation, it may be very difficult to adapt the remote access in the software code to the communication unit. A well known example of system with similar solution is the CELL processor, on the synergistic processing elements (SPE) in CELL, all remote access through the element interconnect bus (EIB) is performed by a DMA controller [2].

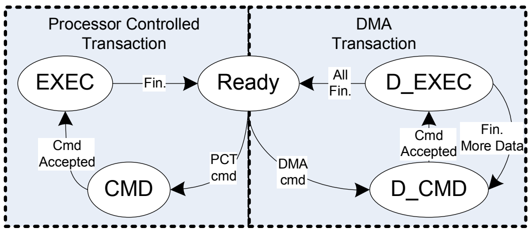
In this work, we choose the second option for the processing tile, based on the following consideration,

- The MicroBlaze does not support block transaction on the data bus, which limits the performance. A communication unit with DMA support gives better performance than a direct connection.

- The data-flow programming model used in this work provides a clean separation between computation and communication, therefore it is easy to use such module in the application. More details can be found in Chapter 4.



(a) Structure of the Connection DMA Controller



(b) Finite State Machine of the DMA Controller

Figure 3.12: Connection DMA controller

A communication unit is designed for the processing tile. In this communication unit, each outgoing NoC connection on the tile has a DMA controller, called connection DMA controller (CDMAC). In the remainder of this section, we discuss the design of the CDMAC, and the management of multiple connection on the processing tile.

Connection DMA Controller

In our design, the connection DMA controller supports the following two kinds of transactions:

1. Processor controlled transaction: in this type of transaction, data is transfer between the processor and the remote location. The processor writes to and reads from the data buffer inside the controller, the block size in this case is limited by the DTL protocol. This kind of operation is useful in transferring small data unit as it provides lower latency than the DMA transaction.

2. DMA transaction: in this type of transaction, the controller moves data between the on-tile memory and remote location, while the processor is doing other things. It supports larger block size than processor controlled transaction. The DMA transaction is efficient in transferring large amount of data, which is chopped into multiple DTL transactions.

The structure of the CDMAC is in Fig. 3.12a. The control logic is implemented as a finite state machine (FSM), as shown in Fig. 3.12b. Note that there are two parts in the FSM, the first part that controls the processor controlled transaction, and the other part that controls the DMA transaction. Either part is optional, for example if the processor controlled transaction is not necessary, it is possible to reduce the FSM, as well as the corresponding hardware in the DMA controller.

Listing 3.1 is the pseudo code of the use of the CDMAC. Note that the programming of CDMAC in a DMA transaction should be atomic, otherwise the behavior of the CDMAC might be incorrect.

Listing 3.1: Transaction on CDMAC (Blocking)

```
/* conn_id is the id of the connection being used */

while(ConnectionIsBusy(conn_id)==true)
```

```
    NOP

if transaction is DMA transaction
    SetSourceAddress(source_addr, conn_id)
    SetDestinationAddress(destination_addr, conn_id)
    if transaction is read
        SendCommand(dma_read, conn_id)
    else
        SendCommand(dma_write, conn_id)
else
    if transaction is read
        SetSourceAddress(source_addr, conn_id)
        SendCommand(pct_read, conn_id)
        ReadBuffer()
    else
        SetDestinationAddress(destination_addr, conn_id)
        SendCommand(pct_write, conn_id)
        WriteBuffer(output_data)
```

Multi-connection Management

For multiple connections, if there is any incoming connection to the local data memory, a scheduler is needed in order to share the memory port between incoming connections and outgoing connections.

If there are multiple outgoing connections, two options are available:

- Extend the DMA controller to support multiple outgoing connections.
- Each outgoing connection that needs to be composable uses a DMA controller of its own.

The first option probably results in a complex DMA controller. As we want the controller to provide composable and predictable service for multiple applications on the processor, the scheduler inside the controller has to be composable and predictable. Moreover, if both incoming and outgoing connections exist, we need a scheduler for the memory port anyway. In Æthereal, such a scheduler is implemented in the initiator bus [22].

Since the scheduler at the memory port cannot be avoided, the second option is a more reasonable choice. In this solution, we keep the DMA controller simple and small. For outgoing connections, each of them has a connection DMA controller.

The resulting architecture is shown in Fig. 3.13. Here a separate communication memory (C-Mem) is used for two reasons: first, the memory map in inter-processor communication is easier if there is any incoming connection. Second, if single port memory is preferred for the local memory, such a separate memory for communication reduces the performance loss of the processor caused by sharing memory port with the NoC connections. If there is no incoming connection, it is possible for the communication unit to use the other port of the processor data memory (D-Mem).

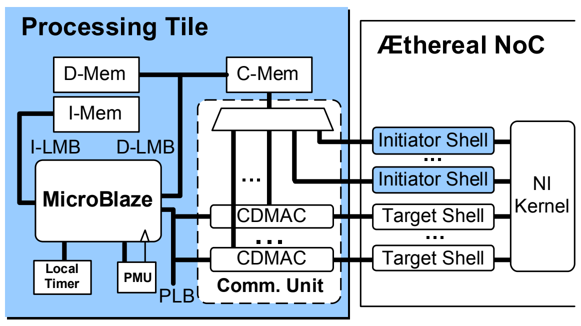
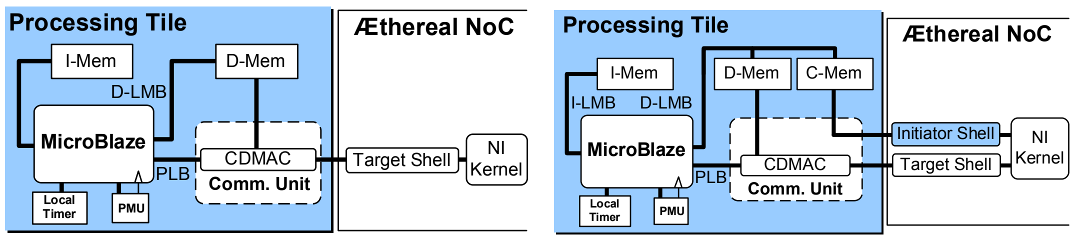


Figure 3.13: Tile Communication Infrastructure for Multi-connection

In the current implementation, because the composable and predictable memory controller is not available yet, each composable connection has a separate communication memory. Fig. 3.14 shows two examples. The tile in Fig. 3.14a has only one outgoing connection, the CDMAC of this connection is connect to D-Mem of the tile. In Fig. 3.14b, the tile has one incoming connection and one outgoing connection. The outgoing connection is connected to D-Mem while the incoming connection is connected to a C-Mem.



(a) One outgoing connection (b) One incoming and one outgoing connection

Figure 3.14: Communication unit examples

3.2.4 Clock Domains on the Processing Tile

There are multiple clock domains on a processing tile. We assume that the NoC is running at a constant frequency f_{noc}.

The processor and local buses use the clock signal generated by PMU, of which the frequency is variable, denoted as f_{tile}. The input clock of the PMU is assumed to have a constant frequency f_{sys}. In the design of this thesis, we assume that $f_{sys} = f_{noc} = f_{max}$ where f_{max} is the maximum frequency in the system, and the clock of the NoC is synchronous with the input clock of the PMU. Note that this is not a restriction but a design choice, it's possible to have a different clock for the NoC.

The operating frequency of the connection DMA controller can be either f_{noc} or f_{tile}. In this work, we chose to run the DMA controller at f_{noc}. The motivation is to

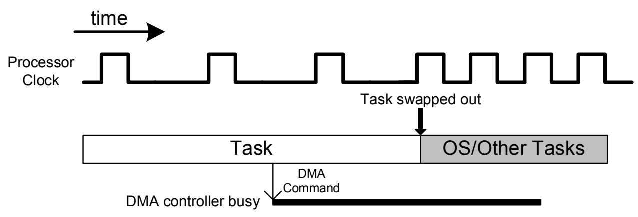


Figure 3.15: Overlap of DMA transaction and task execution

avoid the unintended interference between tasks. When a DMA controller receives a command, it executes it and waits until it is completed, even if the task that gives the command is swapped out, as shown in Fig. 3.15. The new task (or OS) swapped in is probably running at a different frequency, which means f_{tile} is changed. If the DMA controller is running at f_{tile}, interference between tasks is introduced, which is unintended and hard to detect and cope with at design time. If the two tasks belong to different applications, the interference is between applications, and consequently the composability of the system is compromised.

The standard processing tile architecture is shown in Fig. 3.16. There are two clock domains on the tile, PMU and communication module running at f_{noc} and the other parts of the tile running at f_{tile}. The clock domain boundary runs through FIFOs between PMU and the processor, and the FIFOs inside the CDMAC in the communication unit, see Fig 3.6 and Fig. 3.12a. Another clock domain crossing in the processing tile is within the C-Mem and/or D-Mem, which is true dual port memory that allow separate clock on each port.

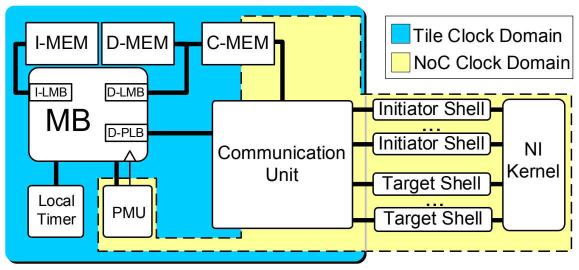


Figure 3.16: Tile Architecture and Clock Domains

3.3 Monitor Tile Design

The main purpose of the monitor tile is to gather trace data from the processing tiles in the system. It monitors the processing tiles by recording the events of interest. There are two ways to do that, to actively detect the events of interest happen on the processing tiles and record them, or to let the processing tiles send the messages to the monitor when events of interest happen, i.e. the monitor collects information passively.

In this work, most events of interest are high level events, e.g. task switching in the OS. It is difficult for the monitor to detect such events for each processing tile, so we decide to use the passive method and let the processing tiles send messages to the monitor when events of interest happen. The FSL of the monitor is used for this propose. The trace data from the processing tiles is stored in memory and sent to the host PC via the serial port when enough data is collected. The architecture of the monitor tile is shown in Fig. 3.17.

The processing tiles use non-blocking FSL instructions [49] to send messages to the monitor tile, hence if the monitor tile fails to empty the FIFOs in time, the processing tiles do not stall. Here the FIFO size is set to 64 words, which is enough for the message from one tile in a time slice, which usually has a length of over 15000 cycles. The load of the monitor is light, thus no data loss is expected during the execution.

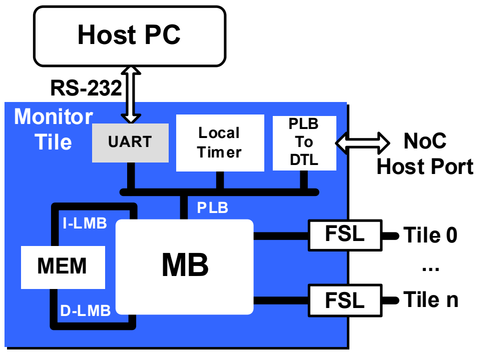


Figure 3.17: Monitor Tile Architecture

3.4 System Configuration

The system has to be configured before the start of any application. The most important thing is setting up the NoC connections for the tiles in the system. And before the configuration is finished, it is important to make sure that the tiles in the system do not try to start running, otherwise their behavior can be incorrect.

In this work the system configuration is performed the monitor tile described in Section 3.3. The monitor tile configures the NoC via a PLB-to-DTL converter. The FSL channel from monitor to processing tile is used to synchronize and configure the tiles. The typical use-case is to synchronize the start of the processing tile, so they start safely after the NoC is configured and the difference between tiles is small (< 100 cycles).

3.5 Integration

The platform is designed in a modular way, so the integration is straightforward. The complete platform is shown in Fig. 3.18. Each processing tile has its own clock and power domain, and we assume that the rest of the platform is in one clock and power domain, in which the frequency is always the maximum frequency of the system f_{max}.

Note that the Æthereal NoC itself does not have power management infrastructure yet, so in this work we do not go into the power management of the NoC.

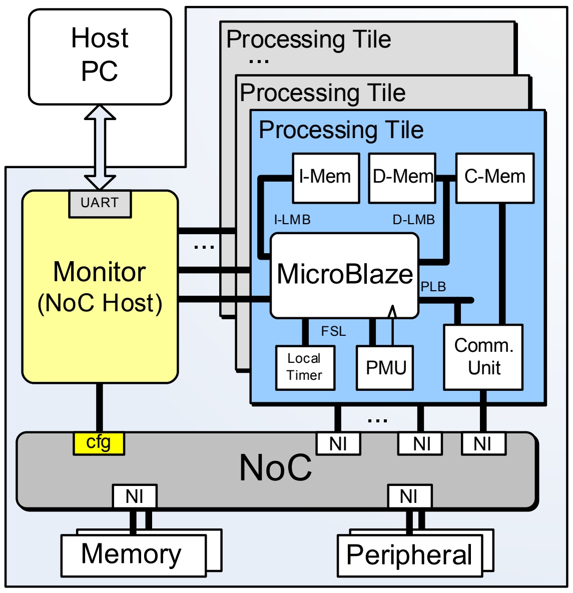


Figure 3.18: Hardware Platform Architecture

3.6 Summary

In this chapter, the design of hardware of the platform is presented. The system has a tiled architecture with the Æthereal NoC as the interconnection. All the basic hardware components in the system provide composable and predictable services. The infrastructure for emulating DVFS and the monitoring support hardware enable the platform to host emulations for power management experiments, which is the main purpose of the platform. The result of this chapter provides a solid foundation for further developments and experiments.

Chapter 4
Software Platform Design

Software is crucial to a processor based system. This chapter describes the software design for the platform, for both the processing tile and the monitor.

CompOSe [19] provides a good starting point for the software system on the processing tile. CompOSe provides composable scheduling which is useful for this work. However, there are several limitations in CompOSe. The most important one is no support for dynamic power management in CompOSe. The lack of progress awareness of tasks and the ability to perform DVFS, makes it impossible to perform efficient power management. Also the limited communication library in CompOSe is too simple for mapping real applications on a multi-processor platform since it only supports single rate static FIFO. In this work we improve the design and implementation of CompOSe, and add components to support power management on a MPSoC platform, as well as the support for monitoring in the FPGA emulation.

An application can be mapped to multiple processing tiles, as discussed in Section 2.7. The hierarchy of the software system on the processing tile is shown in Fig. 4.1.

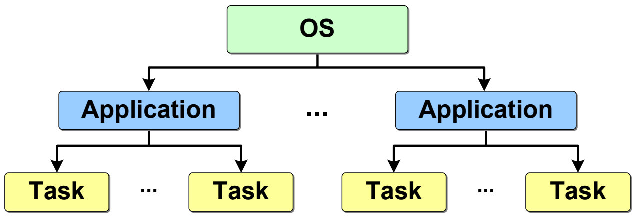


Figure 4.1: Hierarchy in the operating system on a processor

The structure of the OS is in Fig. 4.2. The OS has three main functions: scheduling, inter-task communication, and power management. In the remainder of this chapter we start with the two-level scheduling of the OS, as well as the slack management, is discussed in Section 4.1. Then, in Section 4.2, the inter-task communication service provided by the system is described. Thereafter, the support for power management is shown in Section 4.3. Apart from the application execution, the monitoring of the system activity is also important. It is covered in Section 4.4. At the end, a complete software system is presented in Section 4.5.

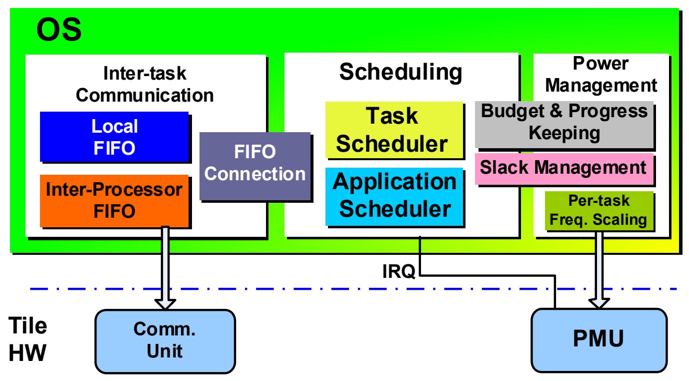


Figure 4.2: Operating system structure on processing tile

4.1 Scheduling on the Processing Tile

In the OS, processor time is divided into time slices of equal length. For each time slice, a task is selected to execute. In this section we talk about how the scheduling is done.

4.1.1 Time Slices on the Processor

The length of a time slice is defined in *wall time*, and in this work, it is controlled by the system timer in the PMU. As specified in Section 3.2.2, the frequency of the input clock of the system timer is the maximum frequency in the system f_{max}, which is constant.

Part of the system time slice is used by the task execution, and part of it is used by the OS, in which most of the OS services are done, as Fig. 4.3 shows. We call the first part the *task time slice* and the second part the *OS time slice*. The task time slice is the time unit for the scheduling in the software system. In each task time a task is selected to execute. The scheduling is preemptive, i.e. at the end of the task time slice, the task that is still running is swapped out. The OS services in the OS time slice run at the maximum frequency f_{max}, while the application is allowed to determine the operating frequency in the task time slice.

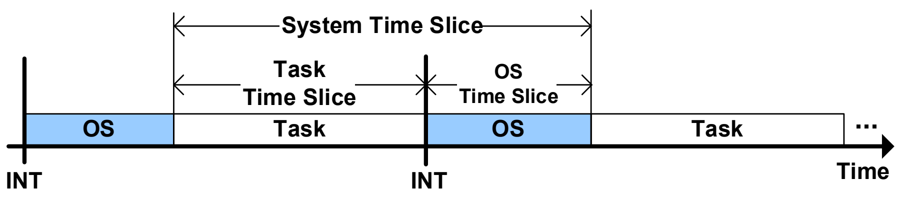


Figure 4.3: Time slices in the operating system

Predictability and Composability of the Time Slices

It is crucial that the system time slice has a constant length. To achieve that, both the task time slice and the OS time slice have to be predictable. In this section we discuss the factors that can affect the predictability of the time slices, and the solutions.

First we look at the interrupt response time and the predictability of the task time slice. At the beginning of each task time slice, the system timer is set to the value of the length of task time slice. When the timer value reaches zero, an interrupt request signal (IRQ) is sent to the processor. That is how a lower bound of the task time slice is achieved. However, the exact length of the task time slice depends on the interrupt response time of the processor and it is not constant in wall time. In our system the interrupt response time depends on the following factors:

1. The instruction being executed when the IRQ is generated. When an IRQ is generated, the instruction in the execution stage of the MicroBlaze's pipeline continues executing until it finishes [49]. This determines the response latency in processor cycles. This latency depends on the instruction type and the memory controller. Assuming no division instruction is used and all memory accesses are local, this latency is less than 5 cycles.

2. Some times the IRQ cannot be served because the processor disables the interrupt temporally, e.g. in this system if there is any sharing of NoC connections the interrupt is disabled when programming the connection DMA controller. This introduces a latency of roughly 30 cycles in the worst case.

3. The frequency of the current task time slice. It determines the response time in wall time together with the response latency in cycles.

In the system, the worst case latency is around 30 cycles. Compared to the length of the time slice, this overhead is small, so the length of the task time slice in wall time has an useful upper bound.

Another thing that affects the length of the system time slice is the execution time of the OS service in the OS time slice, which depends on the the application and task being scheduled.

In this work, we hide these variations in the OS time slice. The length of the OS time slice is the sum of the following:

1. The worst case interrupt response time.

2. WCET of the OS routines.

3. The worst case frequency switching time.

Note that as described in Section 3.2.2, the frequency switch can only happen when the system timer value is multiple of 16, so the length of the OS time slice in cycles should also be aligned to 16.

Fig. 4.4 shows how the variations are hidden in the OS time slice. When the IRQ is generated and the interrupt service routine starts, the processor frequency is switched to the f_{OS} which equals to f_{max}. The OS saves the task context and starts the services, and the system timer continue counting. When the OS has finished everything and ready to enter the task time slice, it gates the clock using PMU and set the un-gate time to $(T_{max} - T_{OS})$, where T_{max} is the maximum value of the system timer and T_{OS} is the length of the OS time slices. When the clock is un-gated, the timer is loaded with the task time slice length, the processor frequency is set to the task frequency and a new task time slice starts. By using hand-written assembly code, and assuming the code is running a local memory with one-cycle latency, we ensure that the code executed after the un-gate is always the same. For frequency switch, the predictability is guaranteed by the switching method in Fig. 3.11. Hence the time between the interrupt and the start of the next task time slice is a constant.

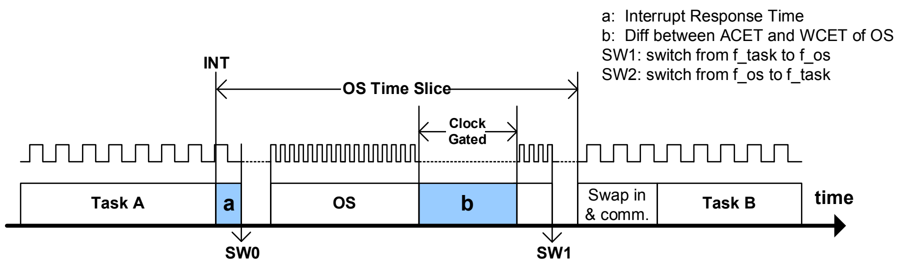

a: Interrupt Response Time
b: Diff between ACET and WCET of OS
SW1: switch from f_task to f_os
SW2: switch from f_os to f_task
INT
OS Time Slice
Clock Gated
Task A
a
OS
b
Swap in & comm.
Task B
time
SW0
SW1

Figure 4.4: Use OS time slice to hide the variation

As all the variations are hidden in the OS time slice, the length of the system time slice is always constant. For the task time slice, the length is predictable, and more importantly, the variation of the length of a task time does not depend on tasks running in other task time slices.

4.1.2 Two-level Scheduling

In this work, we use a scheduler that is similar to the one in [19].

In each task time slice, the scheduler selects a task to run. A preemptive two-level scheduling is employed. The two levels are *system level*, where an application is scheduled, and *application level*, where a task is scheduled. The scheduling is done in two steps, first select an application, then a task in the application is selected to execute in that time slice, as shown in Fig. 4.5.

In the operating system, a task can be a independent thread that scheduled by the OS, or it can be a task in a data-flow application.

A data-flow application is defined as a set of tasks with the communication channels bteween them. We assume there is no dependence between applications. Each task in such an application is an actor in the data-flow graph, which has a number of FIFOs connected to it. The task is allowed to start an iteration when all its input FIFOs have enough data and all its output FIFOs have enough free space, called the firing rules. The checking for input data and output space is called firing rules checking. In this work, the OS is aware of the firing rules, this is required for managing slack and hence real-time frequency scaling and power management [37].

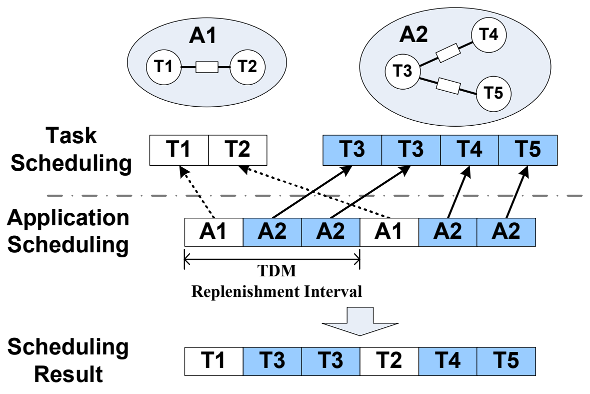


Figure 4.5: An Example of Two-level Scheduling

Composable System Level Scheduling

The application scheduling is based on Time Division Multiplexing (TDM) scheduling. The OS has a list of application scheduling order. Each entry of the list is a application ID, and in each time slice the corresponding application is selected, then the scheduler goes into the task scheduling and selects a task from that application. the length of this list is called TDM replenishment interval.

The length of the system time slice is constant even if the adjacent time slices are assigned to the same application. So once the TDM scheduling order is set, the temporal behavior of an applications in the system does not depend on any other application, i.e. the application scheduling is composable.

Application Level Scheduling

An application consists of a set of tasks, and the dependence between these tasks. Each application defines its own scheduling policy. Once the application is scheduled by the system, the scheduling process is passed to the task scheduler of the application, which selects a task for the given time slice.

In this work, two kinds of task schedulers are available, TDM and round-robin.

Task States in Data-flow Application

As stated earlier in this section, the firing rule checking of the data-flow tasks is handled by the OS, thus the state transition of these tasks is also controlled by the OS.

In data-flow graph, whether a task is allowed to fire or not depends only on the status of the FIFOs connected to it. With such a model, the state transition of a task depends on status of the FIFOs connected to it. A task can be in one of the three states, *waiting*, *running* and *finished*, as shown in Fig. 4.6.

Note that in our system, the OS only checks the status of the FIFOs when it tries to schedule a task, so the state that tokens are ready and the task is waiting to be scheduled does not exist in this system. When a running task is preempted, it stays in the running state as it is ready to run again at any time. When a task finishes an iteration, it enters the finished state, in which the OS resets the task stack. The finished state is not a stable state, once the task is reset, it goes in the waiting state unconditionally and tries to start the next iteration.

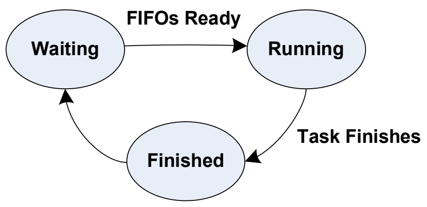


Figure 4.6: Different task states in data-flow application

State	Description
Waiting	Task is waiting for tokens
Running	Task is running, or started but preempted
Finished	Task has finished and is waiting to be reset

Table 4.1: SDF graph task states

4.1.3 Slack Management

Slack Definition

Slack is the idle processor time. Ekerhult [19] defines two kinds of slack based on how the slack is generated:

- *Internal slack*, generated when a task finishes before the end of the task time slice. In this case part of the time slice is slack.
- *External slack*, generated when a task time slice is assigned to an application but it has no eligible task to execute. In this case the whole time slice is slack.

In this work, the OS keeps the budget and progress information of a task, which enables us to extend the definition of slack.

With budget and progress information, there are two kinds of slack in the system:

- Input data or output space is not ready for the selected task. This type of slack is external slack.
- The selected task does not use up the budget in the last execution, i.e. the task execution time is less than its WCET.

The first type of slack is static and in theory can be calculated at design time using the data-flow graph model.

The second type of slack can be detected by proper budget and progress information keeping of the tasks.

Slack Management

When a slack time slice is generated, there are two possibilities to handle it:

- Find a task in the same application that can accept the time slice.
- Give the time slice to other application, the recipient of the slack is not composable in this case.

Fig. 4.7 shows the different levels of slack management. When a time slice is detected as slack, the system first runs the application level slack management routine. If again only the idle task is scheduled, the system runs the system level slack management routine, which tries to schedue another application.

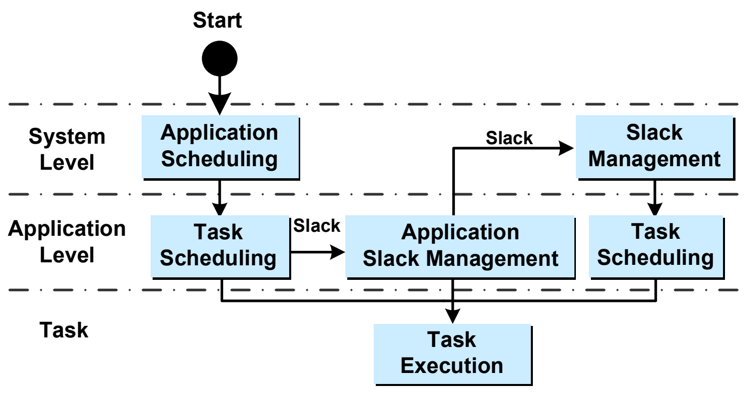


Figure 4.7: Two-level Scheduling with Slack Management

For application level slack management, a interface similar to the task scheduling interface is given. An application defines its own slack management policy. This slack management function can also be seen as part of the task scheduler, but a separate interface makes it more flexible to have different combinations of task scheduling and slack management policies.

For system level slack management, i.e. the slack exchange between applications, we use the similar scheme as in [19]. A slack-matrix is used to define the possibility of slack exchange between applications. The row and column index of the matrix is the ID of the application that gives the slack and the application that receives the slack. Note that if an application can accept slack from other applications, it is no longer composable. In addition it is not predictable if the application is not *monotonic*.

The slack can be used for two purposes, improving the throughput of the application, or reducing the power consumption of the application for a given throughput. This work focuses on the second one.

4.1.4 Complete Scheduling Flow

Listing 4.1 shows the pseudo code of the complete scheduling flow.

Listing 4.1: OS scheduling flow

```
/* App_order is an array that contains the scheduling order */
i = -1;
L = the length of App_order[]
for each interrupt
    i = (i + 1) % L
    app = find_app(App_order[i])
    task = app->task_scheduler()
    if task == idle_task
        task = app->slack_management
        if task == idle_task
            app = sys_slack_management
            task = app->task_scheduler()

    CopyInputData(task)
    Restore_context(task)
    /* Start of interruptible region */
    Run(task)
    CopyOutputData(task)
    WaitForInterrupt()
```

4.2 Inter-task Communication

In this section the support for inter-task communication in the OS is discussed. It is based on C-HEAP protocol [38] and it provides support for both local and inter-processor communication.

The task state in the data-flow graph model is also discussed in this section, as it only depends on the status of the FIFOs connected to the task.

4.2.1 C-HEAP Protocol

The C-HEAP protocol is used as the inter-task communication protocol of the system. It is a FIFO-based communication protocol, in which each FIFO is a logical communication channel which has one producer and one consumer, as shown in Fig. 4.8a. The unit element in the communication is called a *token*. In each transaction, the producer (consumer) writes (reads) a number of tokens, the number it writes (reads) is called the *produce rate* (*consume rate*), or *token rate*. The producer and consumer of the same FIFO can have different token rates.

Fig. 4.8b demonstrates a write transaction and a read transaction. Listing 4.2 shows the pseudo code for a blocking write transaction. The read transaction is performed in a similar way. Note that the write counter and the read counter can only written by one actor, and the writing and reading of the FIFO are controlled by these two counters, therefore no mutual exclusion primitive is required for the synchronization.

Listing 4.2: Write Transaction(Blocking)

```
/* wr_cnt and rd_cnt are in shared memory (remote or local) *
 * wc and rc are their local copies.      */
do{
    rc = rd_cnt;
    space = CalculateSpace(rc, wc);
}while(space < produce_rate);

WriteFIFO();
wc = UpdateWriteCounter();
wr_cnt = wc;
```

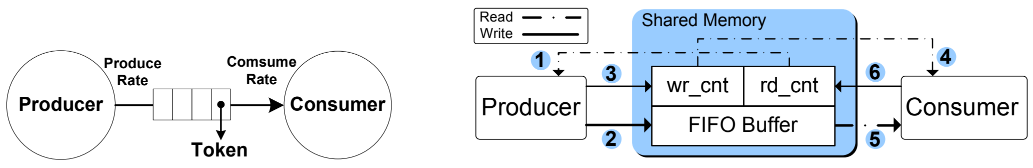
In the C-HEAP implementation, three primitives are used for the write transaction:

1. *claimSpace*: check if there is enough free space in the FIFO.
2. *writeFIFO*: copy data from output buffer to the FIFO.
3. *releaseData*: update the write counter so the consumer can read the data.

Similarly, three primitives are used for the read transaction:

1. *claimData*: check if there is enough data in the FIFO.
2. *readFIFO*: copy data from the FIFO to input buffer.
3. *releaseSpace*: update the read counter so the producer can write to the free space.

As described in Section 2.5, the platform in this work uses data-flow as the primary programming model, C-HEAP fits this model very well.



(a) Model of FIFO (b) C-HEAP transactions

Figure 4.8: Model of FIFO and communication protocol

4.2.2 Communication Flow in the Operating System

The communication flow for applications using data-flow graph model is shown in Fig. 4.9. Noted that here only the case for one time slice is shown, but since the interrupt is enable after the read FIFO operations, there can be more than one time slice. The communication is part of the task execution, so the time for it is also in the task time slice. To decouple communication from computation, the control of the communication is done by the OS. A task is allowed to start a new firing when a time slice is allocated to it and the firing rule checking succeeds. Before starting to run the task code, the OS copies data from all input FIFOs to a local buffer, and after the task code finishes, the OS copies data from local buffer to output FIFOs.

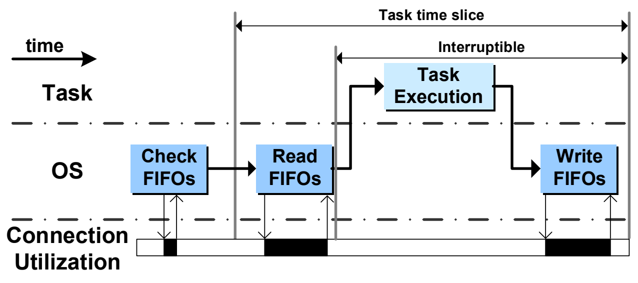


Figure 4.9: Communication Flow for Data-flow Model

Listing 4.3: Communication flow for data-flow task

```
/* T is the task to be scheduled in this slice */

for each FIFO in T
    fail = CheckForToken()
if fail == 1
    return firing rule checking fail

for each input FIFO in T://Read input data
    readFIFO()
    releaseSpace()

Restore_contex(T)
Run(T)

for each output FIFO in T://Write output data
    writeFIFO()
    releaseData()
```

For the applications using data-flow graph model, only communication inside the application is allowed, so all FIFOs are between tasks in the same application. The calculation of the task budget does not need to consider the waiting time for remote resources, as it should be captured by the data-flow model.

Inter-task communication inside the task execution code (Task Execution in Fig. 4.9) is still possible, which enables the possibility of running applications that do not use data-flow model. But for such applications, budgeting becomes a complex problem, hence we do not consider the slack and power management for such applications.

Dynamic Configuration of Firing Rules

To support more flexible programming model such as the cyclo-static data-flow (CSDF) graph, the OS allows the application to dynamically change the firing rules. For each FIFO, two parameters can be configured:

- Token rate for next firing. This is the basic support for the dynamic firing rule. The task, i.e. the actor is allowed to change the number of tokens written (read) to (from) a FIFO in the next iteration.

- Pointer to local buffer. This is useful when the granularity of the variation is very fine, the amount of data (free space) used in one firing is hard or even impossible to be determined before hand, as shown in Fig. 4.10. A typical example of this case is the variable length decoding (VLD) in the multimedia stream decoding. In theory this can be solved by setting the token size to 1 bit, but obviously it is not realistic for most application.

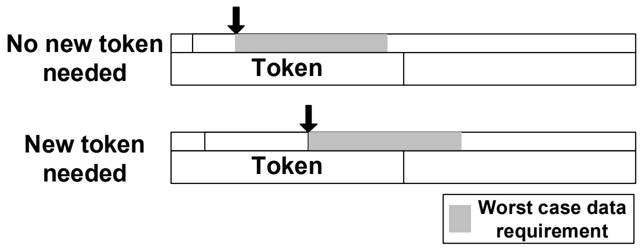


Figure 4.10: Example of Variable Buffer

Memory Mapping of FIFOs

For a non-local FIFO, as described earlier in this section, three things need to be shared by both the producer tile and the consumer tile, namely, the write counter, the read counter and the FIFO storage. There are three different ways to map these shared resources in the system, as listed in Table 4.2:

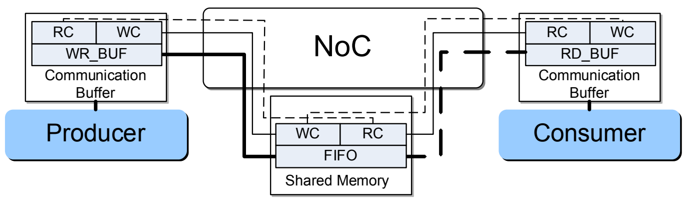
FIFO storage	Counters
Shared Memory	Shared Memory
Shared Memory	Local Memory
Local Memory	Local Memory

Table 4.2: Different memory mapping of FIFOs

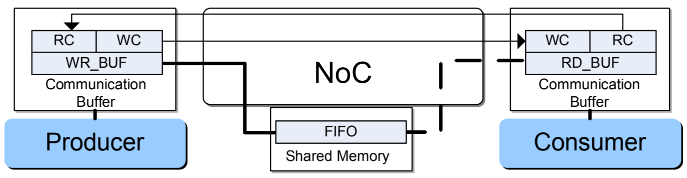
- All shared resources on shared memory, as Fig. 4.11a shows. This mapping is simple, but the OS has to perform a remote read when checking the firing rules, which affects the predictability of OS service.

- Only the FIFO storage is mapped to the shared memory, while the counter values are directly written to the tile's local memory, as shown in Fig. 4.11b. No remote read is needed in firing rules checking. This requires more connections on the NoC in order to enable external access to the tile's local memory.

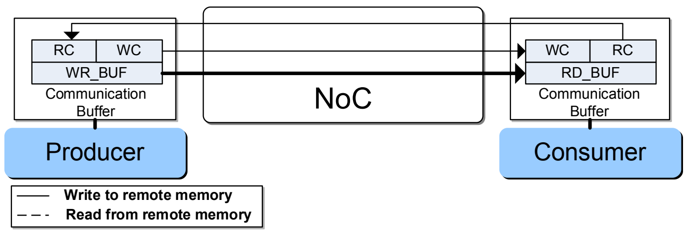
- No shared memory is used at all, everything is mapped to the local memory. As shown in Fig. 4.11c, the FIFO storage locates in the consumer's local memory, possibly combined with the consumer's read buffer. which eliminates remote read. However, the capacity of the FIFO storage is limited by the size of the communication buffer on the consumer tile.



(a) FIFO and counters on shared memory



(b) Only FIFO storage on shared memory



(c) No shared memory

Figure 4.11: Three different memory mappings of a FIFO

The choice of the memory mapping of a FIFO depends on the system architecture and the requirement of the FIFO. Ideally, the first mapping should be avoided, and the third one should be chosen whenever it is possible.

Note that in the second option, the counter and the FIFO data are transferred via different connections. In this work, a write transaction of a tile is posted, i.e. the finish of a write transaction just means the data is sent to the NoC, but the actual time that data reaches the remote resource depends on the QoS of the NoC connection. In Æthereal, the order of the finish time of transactions on different connections is not preserved. This may cause inconsistency in the write transaction of the C-HEAP protocol, the write of the write counter may arrive at the consumer tile before the data is written to the FIFO storage and the consumer may read the data before it is ready. There are two possible solutions for this problem: use tagged write [40] for the last write command, or perform a read from the destination after the write transaction. The first option is easier for software but requires hardware support that is not available in the NoC yet, which leaves the second one to be the only option.

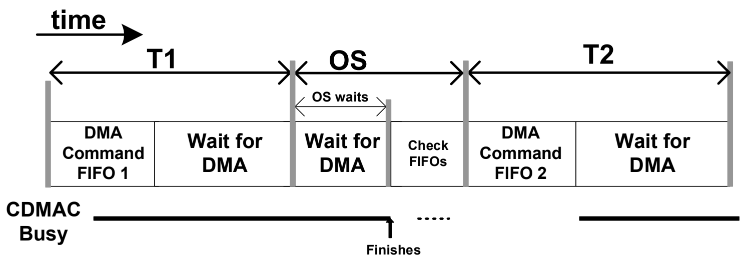
Connection Management in Inter-processor Communication

For FIFOs of which the producer and consumer are not on the same tile, inter-processor communication is required. In the Æthereal NoC, all communication is based on connections, as discussed in Section 3.1.1. And as discussed in Section 3.2.3, a CDMAC is used for each connection. Ideally, each FIFO in the operating system is mapped to one of such connections. In this case the task connected to a FIFO does not need to worry about any conflict on the use of the CDMAC.

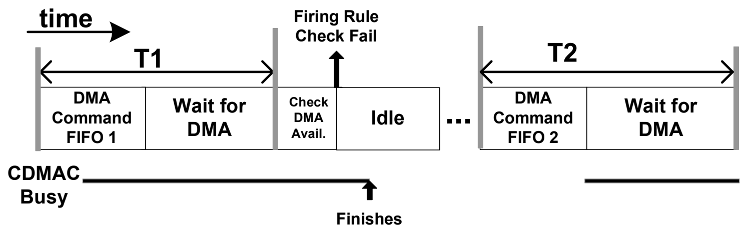
However, for FIFOs that use the same remote memory, there is a possibility of using the same CDMAC for inter-processor communication, which save some hardware resource. In such case, two questions are brought up:

- Assuming the QoS of the connection satisfies the requirements of all FIFOs, can different FIFOs share the same connection?
- If different FIFOs share the same connection, how to manage the potential conflict on the use of the CDMAC?

The connection of the NoC is connected to the processor through the CDMAC, which can transfer data in parallel with the software execution on the processor. With the existence of such a module, sharing a connection with different FIFOs may cause conflicts on the access to the DMA controller.



(a) Blocking connection sharing



(b) Non-blocking connection sharing

Figure 4.12: Connection sharing example

For FIFOs of the same task, connection sharing may limit the communication performance, but it does not cause resource conflicts. However, for FIFOs of different tasks, since the DMA controller can run in the background, connection sharing may cause conflicts.

The conflict can be resolved in two ways, namely, *blocking* and *non-blocking*. An example of two tasks $T1$ with $FIFO1$ and $T2$ with $FIFO2$ sharing a connection is given in Fig. 4.12. The blocking solution hides the interference in the OS time budget, but it may result in a significant increase of the WCET of the OS service, which is unacceptable for predictable systems. On the other hand, the non-blocking version, which adds the connection availability to the firing rule, introduces interference between tasks, and it is unacceptable if the two tasks belong to different applications that have to be composable.

Ideally, each FIFO should has its own CDMAC and connection. In this work, to save resource, we decided to allow the sharing of connections inside an application in the non-blocking way.

4.3 Power Management

In this section we discuss the support for power management in the data-flow application. As stated in Section 2.3.3, the power management requires the budgeting and progress information of the task, so we start with the budgeting for data-flow tasks in Section 4.3.1, then the power management is discussed in Section 4.3.2.

4.3.1 Processor Time Budgeting

For each task that needs budgeting, two kinds of information have to be kept:

- *Task Budget* B_{task}: the amount of wall time assigned to the task for an iteration (firing). A conservative power management should guarantee that the task can finish the current iteration before it runs out of budget.
- *Worst case work* (WCW): the number of processor cycles required to finish an iteration. Assuming the worst case execution time ($WCET$) at the maximum frequency in wall time is known, then $WCW = WCET \cdot f_{max}$.

Task budget B_{task} is defined in wall time. The number of cycles at maximum frequency in a time period of length B_{task} guarantees the finishing of the task iteration in the worst case. In the implementation, B_{task} is converted to number of task time slices T_s, as it is the time unit in the scheduling. So the unit of the B_{task} is the length of the task time slice in wall time, i.e. T_s.

WCW is defined by the number of processor cycles needed for one iteration in the worst case. We assume it does not change when the frequency changes. Similar to the budget, we also want to transform it to a simple metric that does not need fraction. The WCW is updated after each time slice the task uses, therefore the unit of the WCW should be the minimum number of processor cycles a task can get in one time slice.

Assuming there are N uniform frequency steps on the tile $F = \{\frac{f_{max}}{N}, \ldots, \frac{N \cdot f_{max}}{N}\}$ where f_{max} is the maximum frequency. Then, the minimum number a task can get in a time slice would be $T_s \cdot \frac{f_{max}}{N}$. Therefore the WCW is converted to a metric of which the unit is $T_s \cdot \frac{f_{max}}{N}$. cycles.

Assuming resources used by the task are either local resources which run at the same frequency as the processor, or remote resources which have frequency-independent performance, and keeping in mind that a task in data-flow application is never blocked by any other task once it starts running, the workload left at the beginning of the ith time slice in an iteration, $w[i]$, is estimated by (4.1), and $w[0] = WCW$.

$$w[i] = \begin{cases} 0 & Task\ finished \\ w[i-1] - N(\frac{f[i-1]}{f_{max}}) & Otherwise \end{cases} \quad where\ f[i] \in F \tag{4.1}$$

The budget left at the beginning of the ith time slice assigned to the task in an iteration, $B[i]$ is defined in (4.2), and $B[0] = B_{task}$.

$$B[i] = B[i-1] - 1 \tag{4.2}$$

If the *WCW* of a task is known, the budget for this task B_{task} can be easily calculated. It is the minimal number of time slices that gives at least *WCW* cycles, i.e. $(B_{task} \cdot f_{max}) \geq WCW$. A simple way of assigning the budget for a task is defined in (4.3).

$$B_{task} = \lceil \frac{WCW}{T_s \cdot f_{max}} \rceil \tag{4.3}$$

Note that the processor has different frequencies, and as discussed in Section 4.1.2, the length of the task time slice in the system is defined in wall time, therefore the number of cycles in each time slice is variable. With different frequencies, the number of slices used by the task might vary. Since the OS is using preemptive scheduling, the overhead of the task switching can invalidate the simple linear relation between *WCW* and *WCET*. In the remainder of this section, a proper budgeting method for different frequencies is given, which takes the variable overhead of task switches into account.

Budgeting for Multi-frequency

The total number of cycles needed by a task in one iteration is defined by (4.4), where n is the number of time slots needed by the task, C_{sw} is the overhead in each time slice, including the task switching overhead. C_{exe} is the number of cycles needed by the task communication.

$$C_{task} = \lceil n \rceil \cdot C_{sw} + C_{exe} \tag{4.4}$$

Let C_{slice} be the number of cycles in one time slice at the maximum frequency, n is the number of time slices needed by the task in one iteration, it is determined by (4.5).

$$n = \frac{C_{exe}}{C_{slice} - C_{sw}} \tag{4.5}$$

Let $d = f_{max}/f_{scaled}$, the number of cycles in each time slice at f_{scaled} is $\frac{C_{slice}}{d}$. The correctness of the task execution requires that $C_{slice} > d_{max} \cdot C_{sw}$, where $d_{max} = f_{max}/f_{min}$. If this condition is not met, the task is not able to execute at the minimal frequency.

The number of slices needed by the task changes to n', as shown in Fig. 4.13. n' is determined by (4.6). Consequently, the number of cycles needed by the task is defined by (4.7).

$$n' = \frac{C_{exe}}{\frac{C_{slice}}{d} - C_{sw}} = \frac{d \cdot C_{exe}}{C_{slice} - d \cdot C_{sw}} \tag{4.6}$$

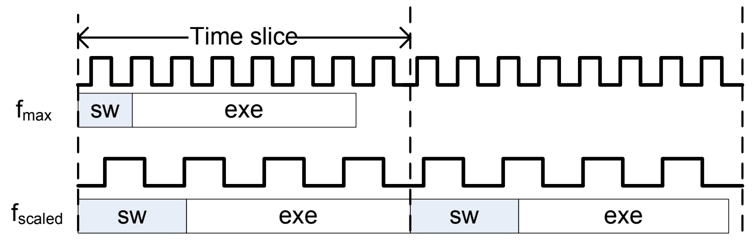


Figure 4.13: Number of time slices changes

$$C'_{task} = \lceil n' \rceil \cdot C_{sw} + C_{exe} \tag{4.7}$$

Obviously, we have $n' \geq n$ and $C'_{task} \geq C_{task}$,

$$\frac{n'}{n} = \frac{\frac{d \cdot C_{exe}}{C_{slice} - d \cdot C_{sw}}}{\frac{C_{exe}}{C_{slice} - C_{sw}}} = \frac{d \cdot (C_{slice} - C_{sw})}{C_{slice} - d \cdot C_{sw}} \tag{4.8}$$

In (4.8) we can see that $\frac{n'}{n} > d$, thus we have $\lceil n' \rceil \geq \lceil d \cdot n \rceil$. Let $x = \frac{C_{slice}}{C_{sw}}$. To run the system correctly, it has to be guaranteed that $x > d_{max}$. And for a real system, it is reasonable to assume that $x > 2d_{max}$, then we have (4.9).

$$\frac{n'}{n} = d \cdot \frac{x-1}{x-d} = d \cdot (1 + \frac{d-1}{x-d}) < 2d_{max} = d_{max} + c \tag{4.9}$$

In a real system, x should be a fairly big positive number, and a tighter bound for c can be derived. And the worst case work for a task for multiple frequencies is determined by (4.10), and a budget is assigned to the task.

$$WCW = (\lceil n \rceil + d_{max} + c) \cdot C_{sw} + C_{exe} \tag{4.10}$$

4.3.2 Power Management

As described in 3.2.2, sixteen different frequency steps are available. In the implementation of the OS, eight frequency steps are available, they are $\{\frac{1}{8} \cdot f, \ldots, \frac{8}{8} \cdot f\}$. The reason is that most SoCs in the real world only have no more than 5 frequency steps. Note that this is not a hard limit, changing the available number of steps to 16 only requires very little effort.

In the power management of this platform, we make the following two assumptions:

1. The number of cycles needed by a task in one iteration, i.e. WCW, is independent of the operating frequency. This assumption holds with the budgeting scheme described in Section 4.3.1 and assuming all resources either run at the processor frequency or have frequency-independent performance. This implies that the the DVFS does not increase *WCW*.

2. Once a task starts executing, it does not need to wait for any resource other than the processor time. This guarantees that receiving slack does not result in a worse execution time than the *WCET*.

Based on the assumption that the number of cycles needed for task execution is independent of the processor frequency, for a task that needs t_0 time to finish at frequency f_0, we have,

$$t_x = \frac{1}{\alpha} \cdot t_0, \text{ where } \alpha = \frac{f_x}{f_0} \tag{4.11}$$

This assumption holds if the task only uses local resources that runs at f_x.

$$f_{ideal} = \frac{cycles\ needed}{cycles\ allocated} \cdot f_{max} \tag{4.12}$$

Processor should be set to the lowest possible frequency that is greater than or equal to ideal frequency, and it is conservative according to the assumptions.

Based on the budgeting scheme described in Section 4.3.1 and the slack management, there are two possibilities for a task to get more processor cycles than it needs:

- If there is internal slack caused by the rounding of task budget, e.g. a task with a WCW of half a time slice would be given a budget of 1 time slice. We called it *over-provision slack*, which is already included in the task budget.
- Extra time slice(s) received through slack management, i.e. the *received slack* S_r.

Let $w[i]$ be the workload left at the beginning of the ith time slice defined in (4.1), N be the number of frequency steps, and B_l be the budget left for the task in the current iteration, the frequency for the ith time slice $f[i]$ can be determined by (4.13).

$$f[i] = \lceil \frac{w[i]}{N \cdot (B_l + S_r)} \rceil \cdot f_{max} \tag{4.13}$$

For many circuits, there is a minimal voltage V_{min}, below which the circuits do not function correctly, therefore the frequency should not scale below the maximum frequency at V_{min}, as it does not save more energy. The OS supports the emulation of this effect by setting a f_{min} parameter.

Note that the assumption of (4.11) is very important for the predictability of the power management. Obviously, it is not accurate if a task needs to access resource outside the tile, e.g. the shared memory.

Workload Decomposition

In principle, the workload of a task can be decomposed into two parts: *on-tile* workload W_{on} and *off-tile* workload W_{off}. Assume the workload is measured in number of cycles, the on-tile workload remains the same in processor cycles when the frequency scales down. The off-tile workload, on the other hand, depends on the frequency of the remote resources, i.e. it is defined in number of cycles at maximum frequency. As stated in Section. 3.2.4 and Section. 3.5, all the communication and remote resources are running at maximum frequency. So when the processor frequency scales down, i.e. $f = f_{max}/N$, the workload of a task in processor cycles is determined by (4.14).

$$W' = W'_{on} + W'_{off} = W_{on} + \frac{W_{off}}{N} \leq W_{on} + W_{off} \tag{4.14}$$

We can see that although the assumption that the workload is constant in number of processor cycles is not accurate, it is a conservative assumption since it does not lead to deadline misses in power management. However it leaves possibility for further optimization. *Workload decomposition* based on memory access intensity is introduced for accurate modeling and using this effect [17, 31]. In this work we do not go into the details, but the decomposition of communication from computation in the OS does provide the opportunity to use such techniques.

Estimating the Energy Consumption

In this work, we focus on the energy consumption of the processing tile. Since the rest of the system is running at maximum frequency, its energy consumption is considered linear to time.

Similar to [37], we assume a linear dependency between the voltage and the frequency, i.e. $V_{DD} = a \cdot f$, where a is a constant. According to (2.2), the energy consumption is determined by (4.15).

$$E = Pt = \alpha C V_{DD}^2 ft = \alpha C a f^2 ft \tag{4.15}$$

As discussed in this section, the number of cycles needed for a given task, i.e. the workload w, is consider as a constant. The energy consumption of a task is determined by (4.16), where f_i is the ith time slice the task used, and t_i is the time the task actually used in the ith time slice.

$$E = \alpha C a f^2 ft = \beta \sum f_i^3 t_i \tag{4.16}$$

4.4 Monitor Support

The monitoring is import for serving the main purpose of this platform as it guarantees the observability of the system. As discussed in Section 3.3, the processing tile is responsible for sending message to the monitor tile. To avoid the interference with task execution, the monitoring information is sent during the OS time slice.

The monitoring messages are sent to the monitor in data packets, which consists of a number of 32-bit words. The structure of the packet is shown in Fig. 4.14. Table 4.3 shows the different types of monitor packets and their payload sizes. Each tile can be configured to send any combination of different types of monitoring packets.

The packet is sent to the monitoring tile via the FSL using non-blocking instructions. The monitor tile keeps polling each FSL port and parse the message. When enough data is collected, it sends all the data to the host PC for analysis.

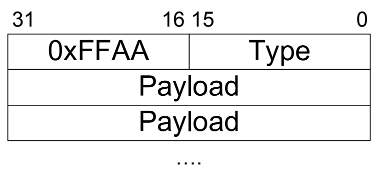


Figure 4.14: Monitoring packet

Type	ID	Payload size	Description
Execution	0x06	1 word	Scheduling information
Task Progress	0x07	3 words	Task progress and budgeting
FIFO read	0x02	3 words	FIFO read operation
FIFO write	0x03	3 words	FIFO write operation
Execution time	0x05	2 words	Execution time of the last iteration

Table 4.3: Monitoring packet types

4.5 Integration

Different components of the operating system are organized in the data structure shown in Fig. 4.15. The system is organized in a hierarchical structure. The top level is the *processor control block* (PCB), where the system-wide information is stored. PCB has a list of *application control blocks* (ACB). Each ACB has a list of *task control blocks* (TCB) and a list of *FIFO control blocks* (FCB). As stated in Section 4.2.2, only tasks that belong to the same application are allowed to communicate with each other, so each FCB belongs to only one ACB. The FCB of a local FIFO is connected to two TCBs in the same application, while the FCB of a inter-processor FIFO is connected to one TCB in the same application.

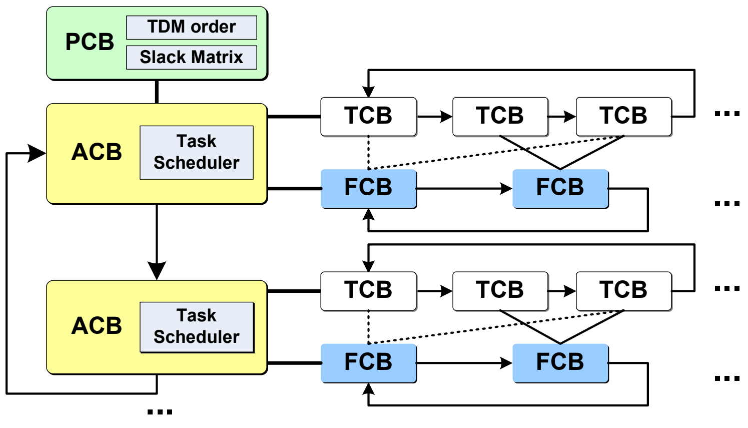


Figure 4.15: Data structure of the system

The control flow of the operating system kernel loop is given in Fig. 4.16. The task update handler is used to let the application perform certain operations after each time slice, e.g. custom budget keeping. This flow summarizes the flows of scheduling, communication and power management discussed in the previous sections.

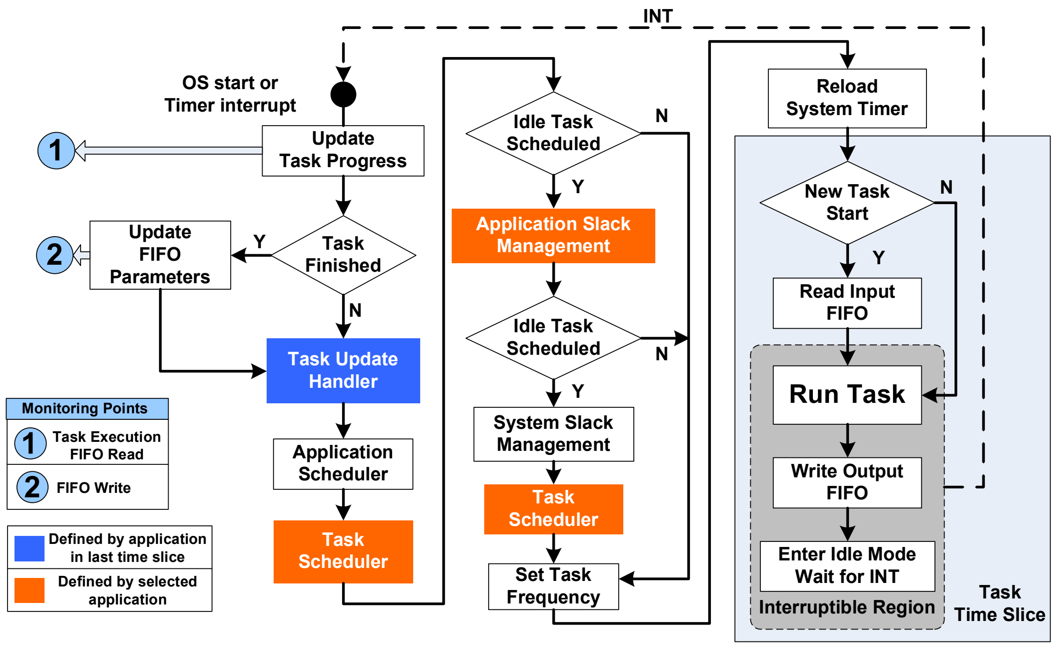


Figure 4.16: Kernel loop of the OS

4.5.1 Predictability

The predictability of the time slices in the OS is discussed in Section 4.1.1. The OS handles the system timer in a way that guarantees the predictability of the OS time slice and the task time slice. However, the predictability of the OS services still needs to be taken care of.

Predictability of the OS Services

To provide a predictable service to the applications, the OS services have to be predictable. As shown in Fig. 4.16, most parts of the OS services in the kernel loop can be easily bounded. However, the following thing should be noted as the OS is highly configurable:

- The number of FIFOs linked to a task should be bounded by a reasonable number, otherwise the time for firing rules checking is not predictable.
- The routines defined by the applications, including the task state update handler, and the task scheduler, should have a bounded execution time. Preferably, the user should only choose the scheduler from a safe, fixed set of schedulers, e.g. the TDM and Round-robin scheduler. And the task update handler should avoid the complex operations that can result in long execution time.

As the OS services is predictable, a budget can be assigned to it based on the WCET of the OS, i.e. the length of the OS time slice. In the implementation, with fixed set of schedulers and task update handlers the WCET of the OS can be measured by experiments using system timer and/or simulation. The typical number of cycles needed by the OS services is between 1500 and 3000. The length of the OS slice length is a configurable parameter in the system, with a default value of 4000 cycles, which is safe for most cases.

4.6 Summary

The design of software part of the platform is discussed in this chapter. The focus is on the OS of the processing tile. The two-level scheduling in the OS provides composability between applications, as well as a flexible scheduling interface for task scheduling inside the application. The system supports the data-flow graph programming model, which is an efficient model for streaming application. The built-in inter-task communication, C-HEAP protocol, can fit in the data-flow model very well. To achieve the primary goal of this work, the power management, the OS must have the ability to keep track of the task budget and progress, and set the processor operating point at appropriate times. Also, the support for monitoring in the emulation is discussed. The result of this chapter, together with the result of Chapter 3, gives the complete template of an MPSoC emulation system.

Chapter 5
Experiments and Results

As the design of the platform is completed, we are ready to run experiments on it. In order to carry out experiments, the experimental system has to be built, so we start with the system configuration, as well as the FPGA implementation flow in Section 5.1. Then, in Section 5.2, the applications used in the experiments are introduced. Finally, the experiments and the result analysis are in Section 5.3. Four different experiments are done here, including tests of execution time and frequency relation, predictability and composability. In the end, a power management experiment is also given to show the power management capability of this platform. These experiment will show that the platform meets the requirements of this work.

5.1 Experimental System Setup

The experiments in this work are done on the Xilinx University Program Virtex-II Pro Development System [1], the FPGA chip on this platform is *XC2VP30*, with $30,816$ Logic Cells, 136 18-bit multipliers, $2,448$ Kb of block RAM , and two PowerPC Processors. Due to the limitation of the FPGA platform, especially the limited on-chip memory, we have different configurations for the different experiments. Table 5.1 shows two different system configurations. The first system is a system with two processing tiles with the same architecture shown in Fig. 3.14a. The architecture of this system is similar to the one in Fig. 5.1, but without the PowerPC, VGA and DDR.

Parameter	Dual-tile	Dual-tile+VGA
Monitor memory size	32KB	32KB
Memory left for trace	96KB	48KB
Tile I-MEM size	32KB	32KB
Shared memory size	16KB	32KB
Tile D-MEM size	32KB	32KB

Table 5.1: Dual tile system configuration

In order to test the application and try the real-life application, a system with frame-buffer and VGA output is also built. A PowerPC processor on the FPGA is used to control the platform IO. The architecture of this system is shown in Fig. 5.1.

This platform is useful for the demonstrations. However, due to the limitations of the FPGA platform, the existence of the IO components limits the configuration of the MPSoC platform, thus it also limits the experiments on this platform.

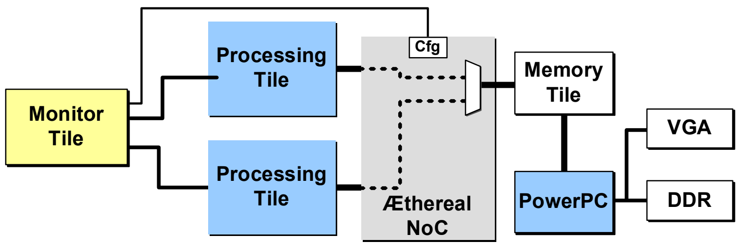
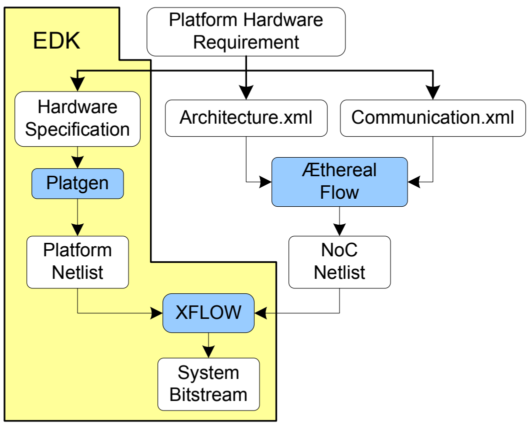


Figure 5.1: Dual tile system with VGA output

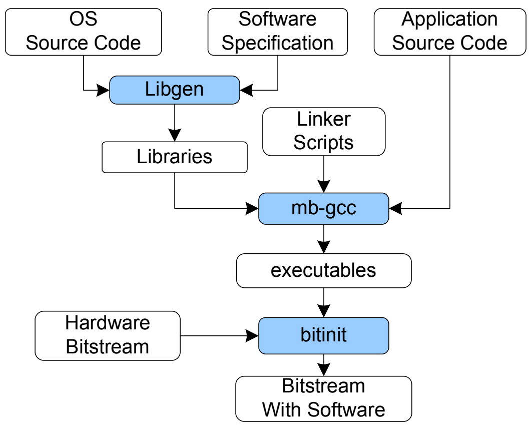
5.1.1 Implementation on FPGA Platform

To implement the hardware of the emulation platform on the FPGA, two different tool flows are used, namely the Xilinx EDK [47] and the Æthereal NoC tool flow [22]. The XML files used to specify the NoC are given in Appendix A. Fig. 5.2a shows the hardware flow of the FPGA emulation platform. The specifications of the hardware platform and the NoC are created according to the system requirement. The Æthereal flow produces the netlist of the NoC (.edf), which is imported to the EDK as a peripheral. The EDK then generates the bitstream (.bit) which is used to configure the FPGA.

For the software, this work uses the software tool-chain in EDK. The flow is shown in Fig. 5.2b. Note that the executables are embedded in the bitstream for FPGA configuration. This is not flexible for reconfiguration of the system; however we leave the dynamic configuration of the software for the future work.



(a) Hardware flow



(b) Software flow

Figure 5.2: FPGA tool flow

5.2 Test Applications

Two different kinds of applications are mapped on the platform: the synthetic application and the JPEG decoder. In this section we will described them.

5.2.1 Synthetic Applications

The synthetic application template is used to emulate the behavior of different applications. A synthetic application consists of two kinds of components, synthetic tasks and FIFOs between the tasks.

The FIFOs used in the synthetic application are almost the same as FIFOs used in normal applications, except that the content of the FIFOs storage can be abandoned. The behavior of different kinds of FIFOs is emulated by setting different token and FIFO sizes, and changing the token rate of the FIFOs.

Parameter	Description	Allowable Value
exe_time	Initial execution time	Positive integer
exe_time_dist	Execution time distribution	User defined function
nbr_fifo	Number of FIFOs	Positive integer
fid_list	FIFO ID list	Array of integer
t_rate	Token rate list	Array of integer
token_rate_dist	Token rate distribution	User defined function

Table 5.2: Parameters for a synthetic task

The synthetic task is useful for emulating of the behavior of different types of tasks. Since the communication is already decoupled from computation in the OS, the implementation of a synthetic task is fairly simple: consume a certain number of processor cycles, and change the token rate. The behavior of a synthetic task is defined by the parameters in Table. 5.2, and the control flow of a task is shown in Fig. 5.3.

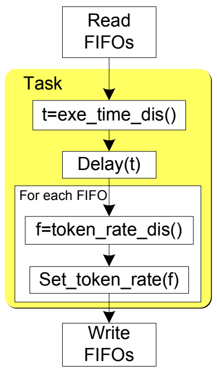


Figure 5.3: Flowchart of the synthetic task

The workload distribution function generates the number of cycles consumed by a task in each iteration. Different workload distributions are implemented, including constant, random and pulse width modulation (PWM). In random distribution, the workload is generated by a pseudo random number generator (PRNG), with a maximum value. In the PWM distribution, the workload is periodic, which is described by two pairs of parameters, (*Period*, *Duty-Cycle*) and (*High*, *Low*). In the first *Duty-Cycle* iterations of a *period*, the workload is *High*, and it is *Low* in the rest of iterations in the period, as shown in Fig. 5.4.

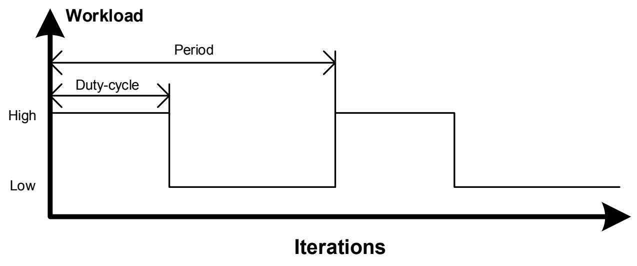


Figure 5.4: PWM workload distribution

5.2.2 JPEG Decoder

To demonstrate the ability of running actual applications on the platform, a JPEG decoder is ported to the platform.

The JPEG decoder is divided into three pipeline stages, the variable length decoding (VLD), the inverse discrete cosine transformation (IDCT) and the color-space conversion (CC). Each stage is mapped to a separate task. An iteration is defined as the processing of one Minimal Coding Unit (MCU) of an JPEG picture. For each task, there is a special iteration called initialization iteration, in which the parameters of the input picture are set in the task state structure, e.g. the picture size, sub-sampling factor.

Fig. 5.5 shows the execution time for each task in the JPEG decoder for decoding three images similar to Fig. 5.5a, the troughs in IDCT and CC are the execution time of the initialization iterations. Based on this information, on the dual processing tile system, the VLD task is mapped on one tile and the other two are mapped to the second tile, as shown in Fig. 5.6.

Note that the producer of the input FIFO and the consumer of the output FIFO are not on any of the processing tiles, but they still need to use the C-HEAP protocol.

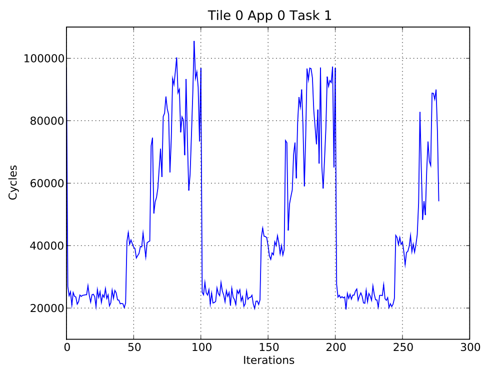
On the platform with VGA output, the input and output of the JPEG decoder is controlled by a PowerPC in order to let the I/O fit in the C-HEAP protocol. By using large buffers and a fast processor (the PPC's frequency is twice as high as the processing tile's maximum frequency), stalling for I/O never happens on the processing tile, so for the tasks on the processing tile, the I/O is infinitely fast.

On the platform without PowerPC, the input FIFO is pre-filled, and the output data is not actually written to the output space.

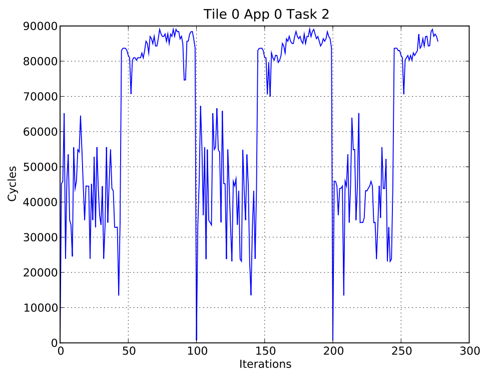
5.3 Experiments and Results

Four different experiments are performed. First one is the test of the impact of communication on the task execution time. Second one is the experiment of ap-

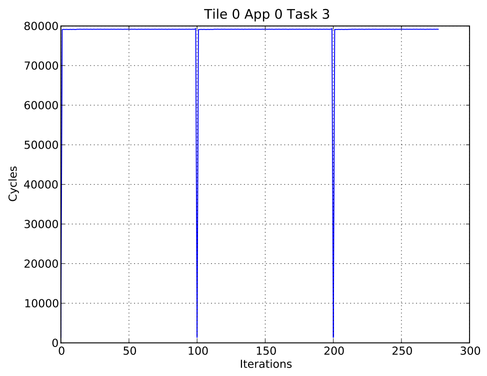
(a) Input image



(b) VLD task execution time



(c) IDCT task execution time



(d) CC task execution time

Figure 5.5: Execution time of the tasks in the JPEG decoder

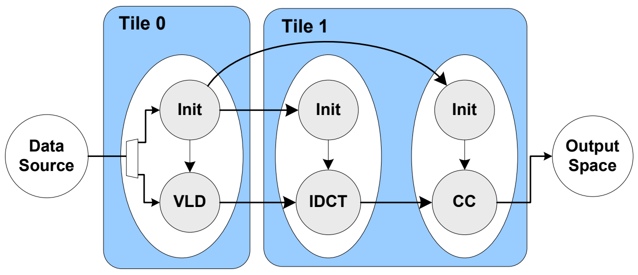


Figure 5.6: Mapping of the JPEG decoder

plication and predictability. The third experiment tests the composability of the system. Finally a power management experiment is performed to demonstrate the power management for data-flow application.

5.3.1 Impact of Inter-processor Communication

In Section 4.3, one of the assumptions is that the workload in processor cycles is independent of the processor frequency. Based on this assumption, a linear relation between the execution time (in wall time) and the processor frequency can be derived. However, as discussed in Section 4.3.2, the assumption may not be valid for workload of different characteristic, but it should be conservative. In this section, an experiment is carried out in order to investigate this assumption.

The setup is a single synthetic task with one single-rate input FIFO and one single-rate output FIFO running on a processing tile. The producer of the input FIFO and the consumer of the output FIFO are on the other processing tile, on which a stand-alone program produce token and read data when ever it is possible. With such setup, the task under test never needs to wait for tokens. Table 5.3 shows different configurations of the experiment, the task delay parameter is constant in all configurations and the task time slice is set to a value that guarantees the task to finish one iteration within one time slice.

FIFO token size (Word)	Workload (Cycle)	Communication in workload (Percentage)
0	12228	0%
16	13387	8.7%
64	14056	13.0%
128	14975	18.3%
256	16809	27.3%
512	20479	40.3%
1000	27502	55.5%

Table 5.3: Different workload of the task

In this experiment, each type of workload is tested at all possible frequency steps, and the execution time of the task is measured by the system timer on the processing tile.

Results and Analysis

Table 5.4 shows the execution times at different frequencies measured by the system timer on the processing tile. The numbers are averaged over 100 iterations and the unit here is cycle.

Fig. 5.7 shows the normalized throughput at different frequencies, where a value of 1 represents the throughput for each configuration at the minimal frequency, i.e. 6.25MHz. We can see that when there is no inter-processor communication, the throughput increases linearly as the frequency increases. When there is inter-processor communication, the curve is below the one without communication. And as the task communicates more, the difference between linear projection and the throughput increases, as expected.

Token Size (Word)	50 MHz	43.75 MHz	37.5 MHz	31.25 MHz	25 MHz	18.75 MHz	12.5 MHz	6.25 MHz
0	12228	13974	16303	19563	24454	32604	48906	97810
16	13387	15266	17771	21287	26544	35334	52884	105615
64	14056	15936	18444	21952	27205	35974	53524	106255
128	14975	16851	19364	22873	28133	36905	54473	107023
256	16809	18689	21200	24704	29974	38753	56265	108943
512	20479	22359	24871	28378	33637	42417	59935	112655
1000	27502	29380	31886	35392	40647	49429	66986	119695

Table 5.4: Execution times at different frequencies

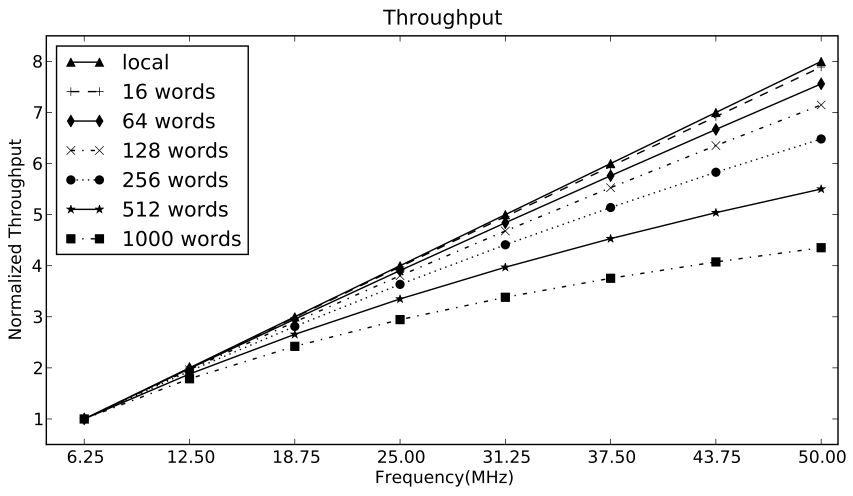


Figure 5.7: Throughput at different frequencies

The conclusion is that the linear assumption in Section 4.3 is conservative, which means the number of processor cycles does not exceed the calculated WCW. Thus we can use it as the basis for the power management.

5.3.2 Application Slack Management and Predictability Test

In this section, experiments are carried out to test the application slack management and predictability of the system. A synthetic application is used in this experiment. The setup of the test application is shown in Fig. 5.8. The task slice length in this experiment is 1/1000 s

The application consists of four tasks, connected as a pipeline. Each FIFO in the task graph is single rate, with token size of 40 Bytes and FIFO size of 8 tokens. The configurations of the synthetic tasks are shown in Table 5.5.

TDM is used for task scheduling, and the scheduling order is as follow:

- Processing Tile 0: {T1, T1, T1, T1, T2, T2 }.

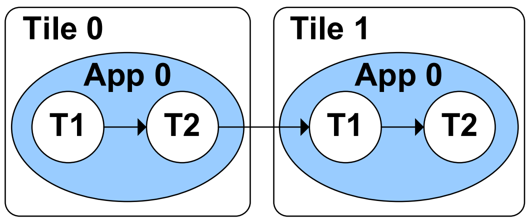


Figure 5.8: Slack Management Test Application

Task	Random Max	WCW (1/8 slice)	Budget Slice	Number of FIFOs
Tile 0 T1	47000	32	4	1
Tile 0 T2	16384	16	2	2
Tile 1 T1	47000	32	4	2
Tile 1 T2	16384	16	2	1

Table 5.5: Configurations of the synthetic tasks

- Processing Tile 1: {T1, T1, T1, T1, T2, T2 }.

Two types of application slack management policies are tested in this experiment.

1. *Self*: tries to give the slack slice to the task which generated it (the owner).
2. *Next-eligible* (NE): tries to find an eligible task starting from the task next to the owner in the application's TCB list. The owner is tested when all other tasks cannot run.

Fig. 5.9 shows an example of the scheduling without any slack management, and Fig. 5.10 shows the result of using the two slack management policies. In Fig. 5.10a the next-eligible policy significantly reduces the number of idle slices. In Fig. 5.10b the self policy only uses the idle slices from 43 to 45.

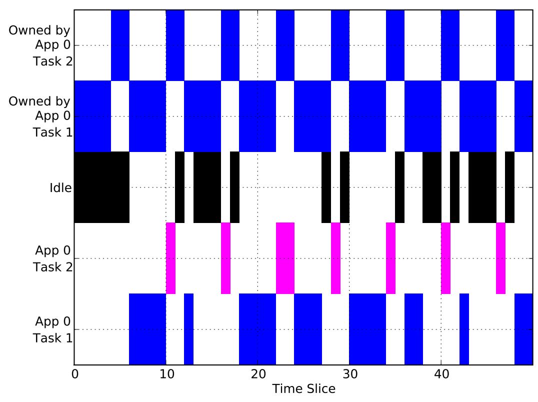
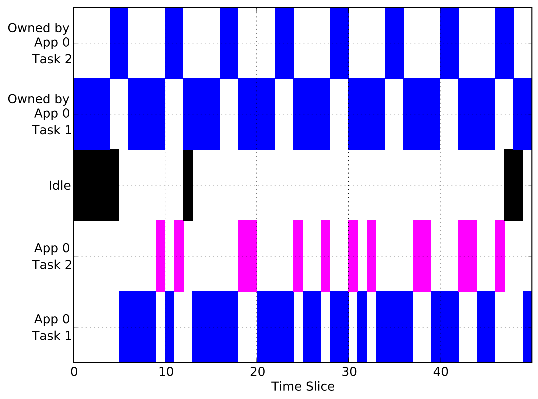
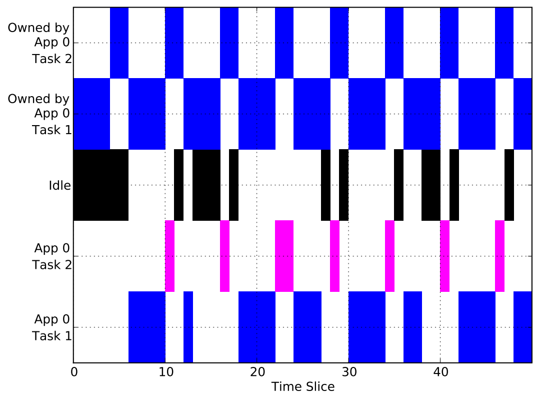


Figure 5.9: Scheduling without slack management



(a) Next-eligible policy



(b) Self policy

Figure 5.10: Scheduling with slack management

There are two ways to use the slack. First, the slack can be used to get a higher throughput. Second, the slack can be used to enable the processor to run at a lower frequency to save energy. In this experiment, both options are tested.

Results and Analysis

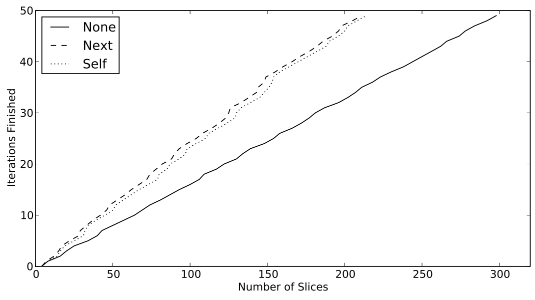
Table 5.6 shows the result of running 80 iterations. From this table we can see that when DVFS is disabled, both policies significantly improve the throughput of the application, thus the utilization of the processor is high. However, for the power management, the next-eligible policy performs much better than the self policy.

Task	Fixed TDM	NE No DVFS	Self No DVFS	NE DVFS	Self DVFS
Tile 0 T1 Used Slices	234	234	236	363	320
Tile 0 T1 Owned Slices	320	228	236	320	320
Tile 0 T2 Used Slices	106	106	106	116	108
Tile 0 T2 Owned Slices	159	112	118	159	159
Tile 0 Utilization	0.7098	1.0	0.9661	1.0	0.8935
Tile 1 T1 Used Slices	212	212	220	295	215
Tile 1 T1 Owned Slices	324	228	248	322	324
Tile 1 T2 Used Slices	106	106	106	121	106
Tile 1 T2 Owned Slices	161	114	124	160	161
Tile 1 Utilization	0.6557	0.9298	0.8763	0.8631	0.6619

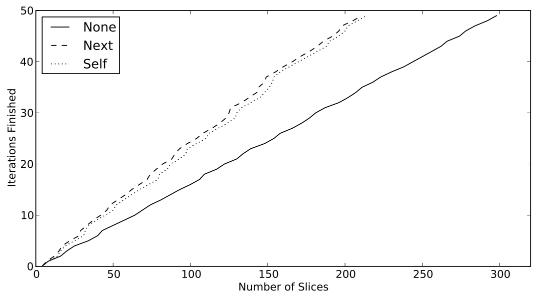
Table 5.6: Processor utilization

Fig. 5.11 shows the accumulated finishing time for 50 iterations, i.e. the throughput of each task without DVFS. We can see that with slack management, the time used for finishing 50 iterations becomes shorter, i.e. the throughput increases. Moreover, the finishing time with slack management is always earlier then the one without slack management, i.e. the slack management is conservative.

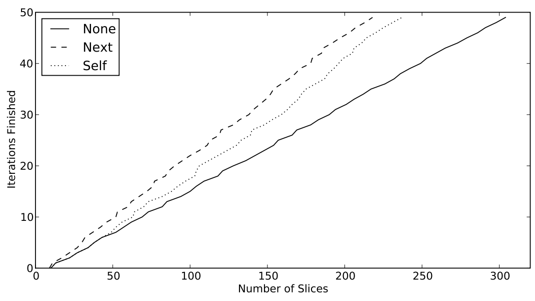
Fig. 5.12 shows the throughput of each task with DVFS. Here we can see that



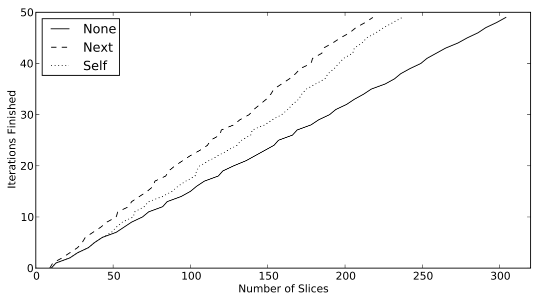
(a) Tile 0 Task 1



(b) Tile0 Task 2



(c) Tile 1 Task 1



(d) Tile1 Task 2

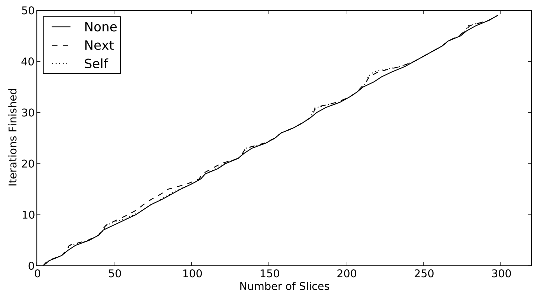
Figure 5.11: Throughput without DVFS

the slack management is still conservative. But in contrast to Fig. 5.11, there is no considerable difference in the finishing time although the slack management is enabled.

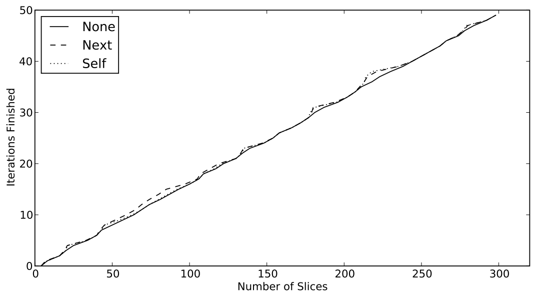
Fig. 5.13 shows the frequency traces of each task for the next-eligible policy, and Fig. 5.14 shows the ones of the self policy. Here the frequency trace is subsampled for each task, i.e. only the time slices used by the task are included. The self policy cannot do much on the second tile, as the pace of the second tile is mostly determined by the first tile. Since the slack management with DVFS does not increase the throughput, in the self policy, T1 on the second tile can not start before it gets tokens from T2 on the first tile, which runs at the similar speed as the fixed TDM scheduling. So the result of the tasks on the second tile is similar to the result of fixed TDM scheduling.

The slack accumulation of the tasks are also different with different goal for slack management In this experiment, the budget is assigned based on the WCET of the task, which means the actual execution time of the task is always smaller or equals to the budget. Consequently, when no DVFS is performed, the slack of a task accumulates over time, while in the situation with DVFS, the slack is consumed by tasks running at low frequency and the slack accumulates slower or even does not accumulate. An example of the slack accumulation is shown in Fig. 5.15, the slack accumulates quickly when DVFS is disabled, and it does not accumulate when DVFS is enabled.

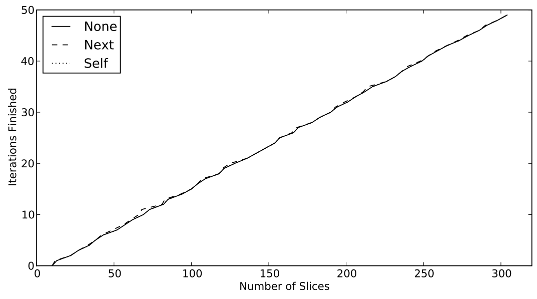
In this experiment, the application slack management is tested. Two different policies are used and the two different goals of slack management, throughput and energy with the two policies are observed. The system produces the expected results,



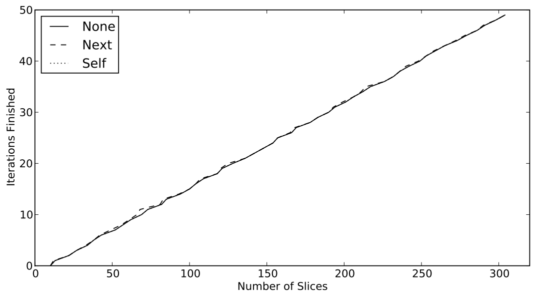
(a) Tile 0 Task 1



(b) Tile0 Task 2

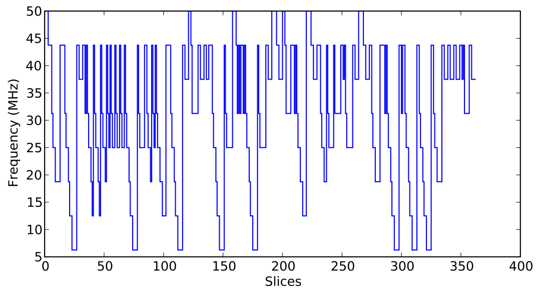


(c) Tile 1 Task 1

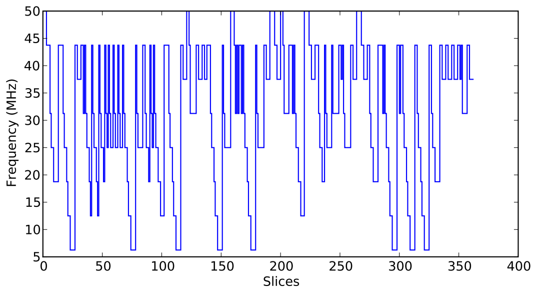


(d) Tile1 Task 2

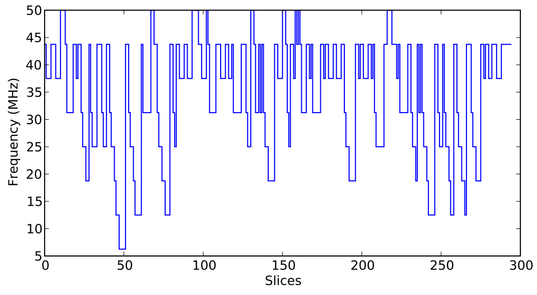
Figure 5.12: Throughput with DVFS



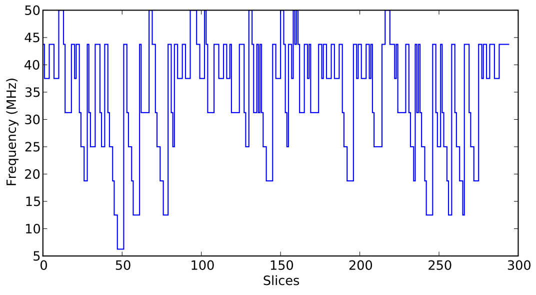
(a) Tile 0 Task 1



(b) Tile0 Task 2

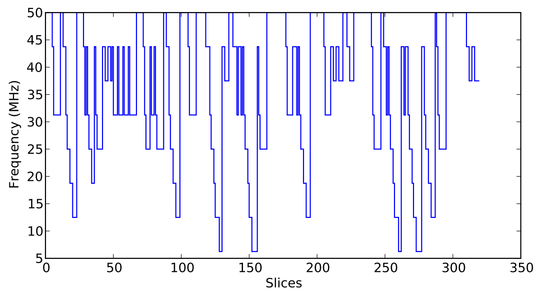


(c) Tile 1 Task 1

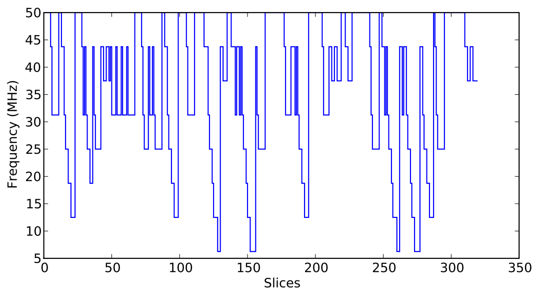


(d) Tile1 Task 2

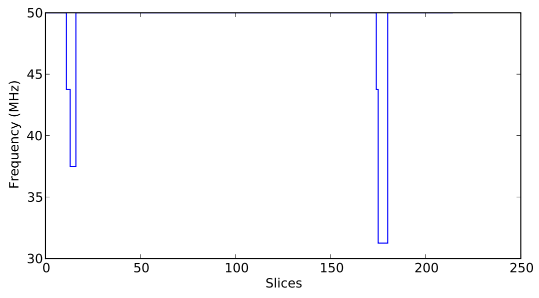
Figure 5.13: Frequency of each task with next-eligible policy (subsampled)



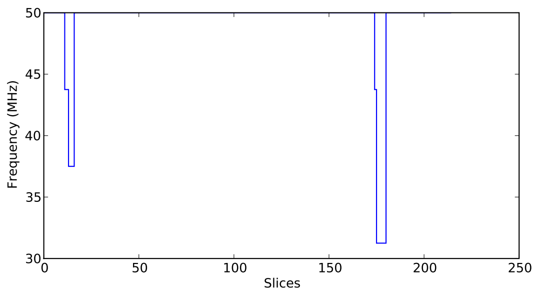
(a) Tile 0 Task 1



(b) Tile0 Task 2

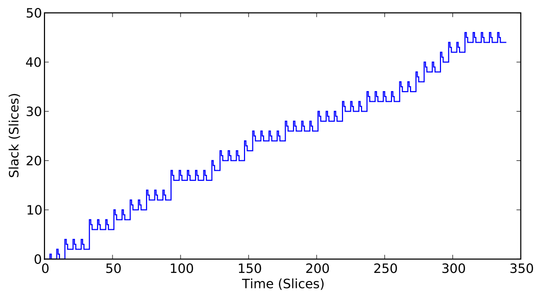


(c) Tile 1 Task 1

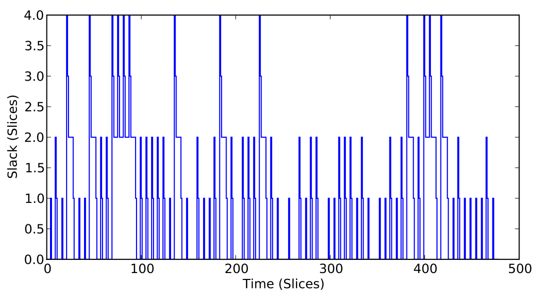


(d) Tile1 Task 2

Figure 5.14: Frequency of each task with self policy (subsampled)



(a) Without DVFS



(b) With DVFS

Figure 5.15: Accumulated slack for Tile 0 T2 with next-eligible policy

from which we can conclude that the slack management in the TDM scheduling is conservative, whether power management is enabled or not.

5.3.3 Composability Test

In this section, experiments are carried out to verify the composability of the platform. The setup of the test applications is shown in Fig. 5.16. The JPEG decoder is mapped to two processing tiles, and on each processing tile, there is an additional synthetic application which consists of three tasks. Since the implementation of a composable memory controller is not available yet, in this experiment only one application, the JPEG decoder is mapped on multiple processing tiles. The task slice length in this experiment is 1/3000 s.

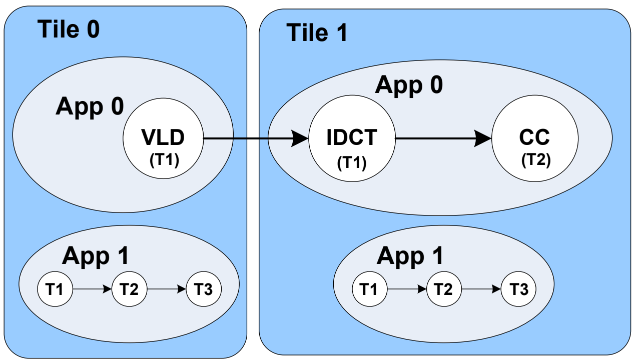


Figure 5.16: Composability test applications

To test the composability, we fix the settings and input of the JPEG decoder, and change the settings of the synthetic applications. The JPEG decoder uses round-robin on both tiles, and the synthetic application has different test cases as in Table 5.7. Here cases of different schedulers, with and without slack management (SM) and DVFS are tested.

Case	Synthetic Application Task Scheduler
1	Round Robin
2	TDM
3	TDM + SM
4	TDM + SM + DVFS

Table 5.7: Composability test case

The workload and budget for the synthetic application in the TDM scheduling are shown in Table 5.8.

In TDM scheduling, both synthetic applications use the following scheduling order:

- {T1, T1, T1, T1, T2, T2, T3, T3 }.

Task	PWM Value (High, low)	PWM Period (Period, duty cycle)	WCW (1/8 slice)	Budget Slice
Tile 0 T1	$(15000, 500)$	$(5, 1)$	32	4
Tile 0 T2	$(8000, 500)$	$(5, 3)$	16	2
Tile 0 T3	$(8000, 500)$	$(10, 3)$	16	2
Tile 1 T1	$(15000, 500)$	$(5, 2)$	32	4
Tile 1 T2	$(8000, 500)$	$(5, 2)$	16	2
Tile 1 T3	$(8000, 500)$	$(5, 2)$	16	2

Table 5.8: Task workloads in composability test

The application scheduling order for the two tiles is as follows:

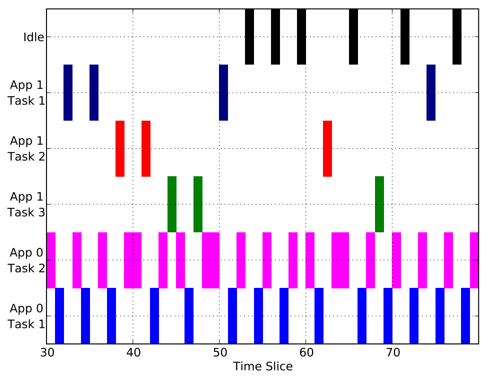
- Processing Tile 0: {A0, A1 }.
- Processing Tile 1: { A0, A0, A1 }.

This work focuses on the composability in the temporal domain. In this experiment, we change the configuration of the synthetic application according to Table 5.7, and measure the following parameters for the JPEG decoder:

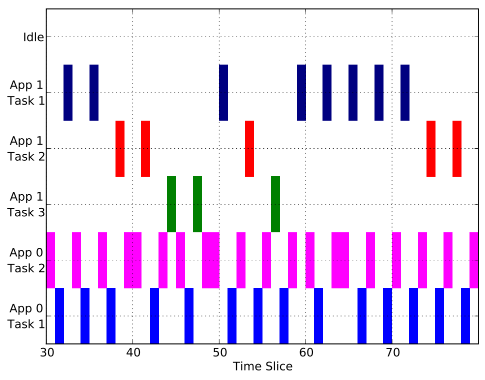
- Trace of the scheduling order on each of the processing tiles.
- Execution time of each iteration.
- Finishing time of each iteration.

Results and Analysis

Fig. 5.17 shows an example of the scheduling order. Application 1 uses different scheduling policies, namely, without slack management in Fig. 5.17a and with slack management in Fig. 5.17b. The idle slices in Fig. 5.17a are used in Fig. 5.17b, which increases the throughput of the application. The scheduling order in application 0 remains the same when different scheduling policies are used in application 1.



(a) App 1 without slack management



(b) App 1 with slack management

Figure 5.17: Scheduling of tile 1

The execution time of the JPEG decoder measured by the system timer on the processing tile for different test cases is exactly the same. Fig. 5.18 and Fig. 5.19 show the execution time of the sink tasks (VLD on tile 0 and CC on tile 1) on the two tiles. Case 1, 2, 3 and 4 correspond to the four cases in Table 5.7. In different test cases, the throughput of the synthetic application changes a lot while the behavior of the JPEG stays the same.

To see the impact of execution time in wall time, the finishing of each iteration of the application is time-stamped by the monitor. This measurement is not accurate at cycle level as there is interference between different processing tiles and different applications on the monitor side. However the relative error due to the time-stamping is less than 0.5%, so we can conclude that the execution time in wall time is also constant for all test cases.

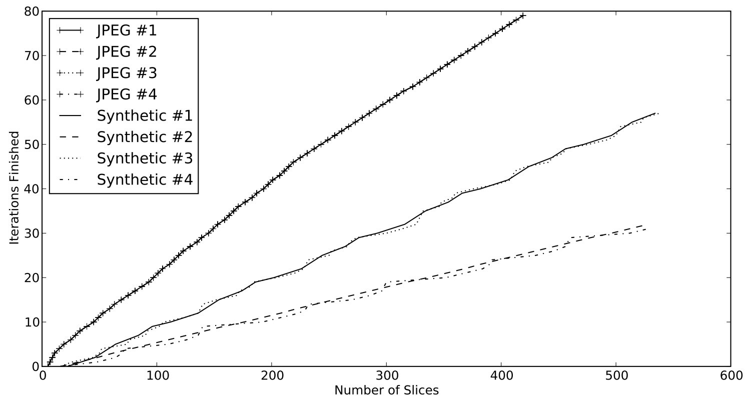


Figure 5.18: Throughput in different test cases on tile 0

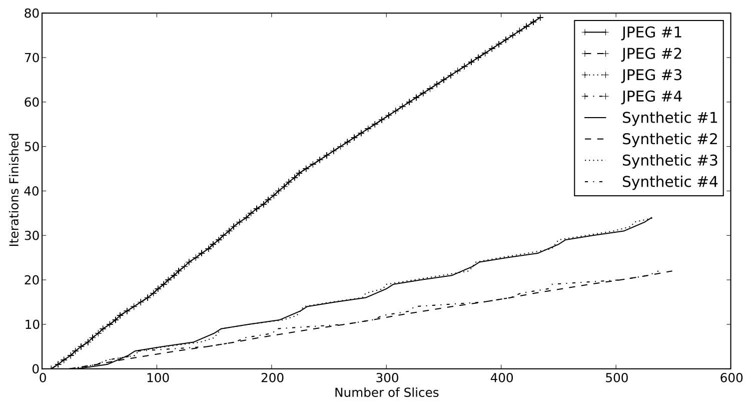


Figure 5.19: Throughput of different test cases on tile 1

In this experiment, the behavior of the JPEG decoder remains the same when the other applications change the behavior, which shows that the platform is composable in the temporal domain.

5.3.4 Power Management Case Study

In this section, an example of power management in the platform is given. In this experiment only one application is running on the dual processing tile platform. The test application is a synthetic application with the task graph is shown in Fig. 5.20. All FIFOs have the same token size, 10 words, and all token rates are 1. The sizes of the FIFOs are given in Table 5.9. The task time slice length in this experiment is $1ms$, i.e. 50000 cycles.

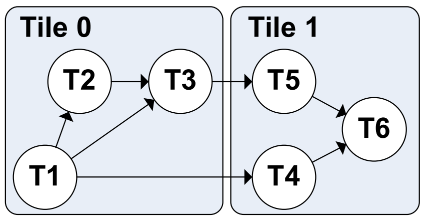


Figure 5.20: Task graph of the synthetic application

FIFO	Size (tokens)
FIFO 0	8
FIFO 1	4
FIFO 2	4
FIFO 3	8
FIFO 4	8
IPC FIFO 0	8
IPC FIFO 1	8

Table 5.9: FIFO size in the synthetic application

In this experiment, two types of workload distributions are used, PWM and random. Table 5.10 gives the workload and budget information of the synthetic tasks. The energy consumption of each task is estimated by the model in Section 4.3.2.

Task	PWM Value (High, low)	PWM Period (Period, duty cycle)	Random Max	WCW (1/8 slice)	Budget Slice
T1	$(24000, 5000)$	$(8, 1)$	16384	16	2
T2	$(36000, 5000)$	$(5, 3)$	47000	24	3
T3	$(47000, 5000)$	$(3, 2)$	37500	32	4
T4	$(24000, 5000)$	$(8, 1)$	16384	16	2
T5	$(47000, 5000)$	$(8, 1)$	47000	32	4
T6	$(36000, 5000)$	$(8, 1)$	37500	24	3

Table 5.10: Workload of the synthetic tasks

TDM scheduling with progress information is used on both tiles. Based on the task graph and the workload information, the scheduling order is as follows,

- Processing Tile 0: $\{T1, T1, T2, T2, T2, T3, T3, T3, T3\}$.
- Processing Tile 1: $\{T4, T4, T5, T5, T5, T5, T6, T6, T6\}$.

For slack management, the next-eligible policy described in Section 5.3.2 is used.

Results and Analysis

For each type of workload, the following cases are tested:

- Both tiles use fixed TDM scheduling, no slack management (SM) and no DVFS.
- Both tiles use TDM scheduling with slack management, but only one tile uses DVFS.
- Both tiles use TDM scheduling with slack management and DVFS.

In the experiment, a test case with a minimal frequency at $0.5 \cdot f_{max}$ (25MHz) instead of $0.125 \cdot f_{max}$ (6.25MHz) is also included to see the effect of the minimal voltage describe in Section 4.3.2. In this case, the following five frequency steps are available: 50MHz, 43.75MHz, 37.5MHz, 31.25MHz and 25MHz.

When calculating the energy consumption, the unit is the energy consumption of a full task time slice running at f_{max}. And according to (4.15), for a given period of time t, the energy consumption $E = \alpha Caf^2ft = \beta f^3 t$ where β is a constant for a certain circuit and application. Here we let the length of the task time slice be the time unit and denote it as T, and let the maximum frequency f_{max} be the frequency unit. Here we define the unit of energy consumption to be the energy consumption of running at maximum frequency for one task time slice, then the energy consumption in one task time slice is determined by (5.1). Where t_x is the actual time used in that time slice. For a task that does not finish in the slice, we have $t_x = T$.

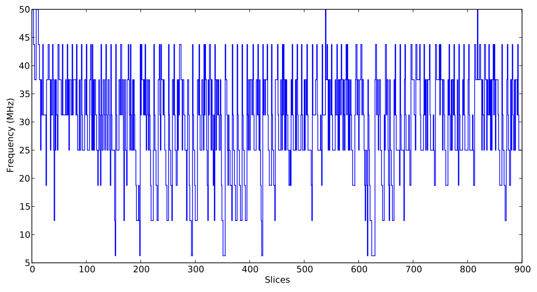
$$E_x = (\frac{f_x}{f_{max}})^3 \frac{t_x}{T} \tag{5.1}$$

In this experiment, the average energy consumption for each task in one iteration in 100 iterations is calculated. Since the OS gates the clock when the idle task is selected, in this section, the idle slices in the system are considered not consuming any energy. For each task time slice there is a corresponding OS time slice which runs at f_{max}, and it consumes energy as well. Here we also calculate the overhead of the OS time slice, assuming the execution time of the OS services is constant, then the overhead is linear to the number of time slices used. In this experiment the OS overhead is set to 0.08 per slice, which equals to 4000 cycles (the default OS time slice length) at this task time slice length.

Table 5.11 shows the result of random workload. The maximum power saving is 49.93% in comparison to the fixed scheduling without any power management, which happens when both tiles enable DVFS with minimal frequency restriction. The number drops to 43.55% when the 8% OS overhead. If we assume the system always runs at maximal frequency instead of halt when it is idle, the maximal saving here is 74%. Note here only the energy consumption of the processing tile is considered; for the system wide energy consumption, the energy consumption of the other resources has to be considered. The comparison between different policies is shown in Fig. 5.25a. Limiting the minimal frequency has positive impact on the energy saving in this case, although the difference is not significant. Fig. 5.21 and Fig. 5.22 shows the frequency of the processor with and without the limit of minimal frequency. The minimal frequency restriction forces the processors to run at a more uniform frequency, so it consumes less energy.

Tile0 Tile1	Fixed Fixed	SM+DVFS SM	SM+DVFS SM+DVFS	SM+DVFS+min SM+DVFS+min
T1	0.723	0.380	0.380	0.373
T2	1.233	0.579	0.579	0.574
T3	2.579	1.182	1.182	1.174
T4	0.638	0.639	0.331	0.334
T5	2.694	2.695	1.352	1.321
T6	1.394	1.394	0.913	0.862
Sum	9.2629	6.8690	4.7372	4.6376
Norm. Energy	1.0000	0.7416	0.5114	0.5007
With OS overhead	1.0000	0.7767	0.5783	0.5645
Total slices	1808	1806	1817	1757
Used slices	1233	1528	1757	1724
Utilization	0.682	0.846	0.967	0.981

Table 5.11: Energy consumption for random workload


(a) $f_{min} = 6.25\text{MHz}$

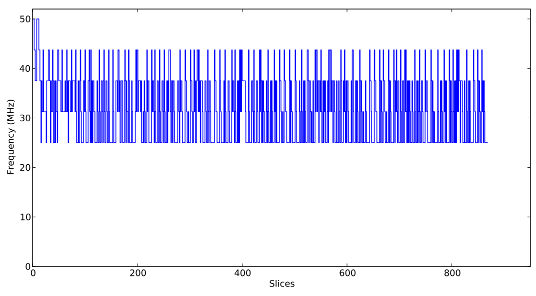
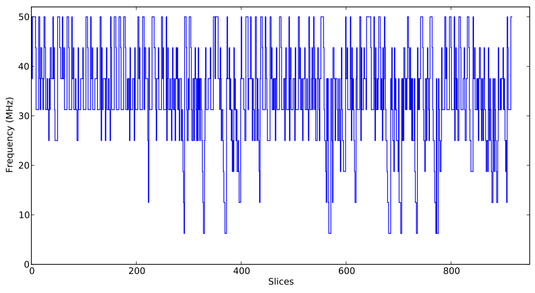
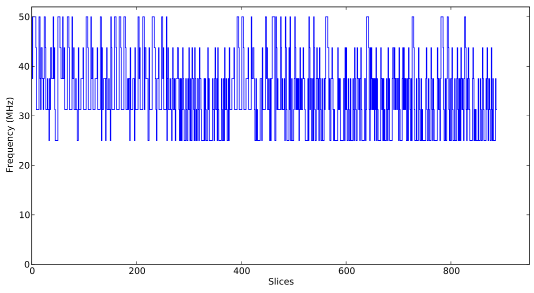

(b) $f_{min} = 0.5 \cdot f_{max} = 25\text{MHz}$

Figure 5.21: Frequency of Tile 0 with random workload

Table 5.12 shows the result of PWM workload. The maximal power saving is 47.98% in comparison to the fixed scheduling, which happens when both tiles enable DVFS with minimal frequency restriction.. The number drops to 41.57% when the 8% OS overhead. If we assume the system always runs at maximal frequency instead of halt when it's idle, the maximal saving here is 68%. Again, here only the processing tile is considered. Limiting the minimal frequency in this case also saves even more energy then the case without the restriction. Fig. 5.23 and Fig. 5.24 shows the frequency of the processor with and without the limit of minimal frequency, which is similar to the case with random workload. The comparison of different policies is shown in Fig. 5.25b.

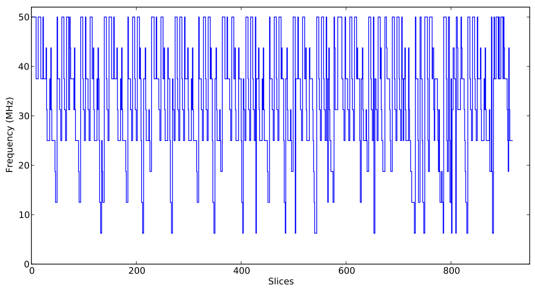


(a) $f_{min} = 6.25$MHz

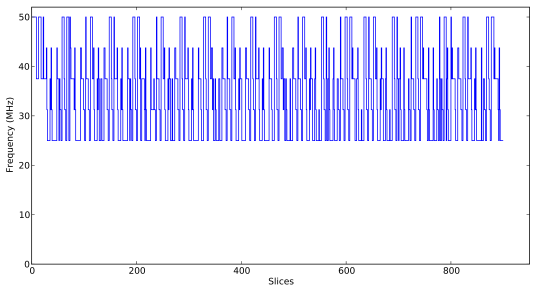


(b) $f_{min} = 0.5 \cdot f_{max} = 25$MHz

Figure 5.22: Frequency of Tile 1 with random workload

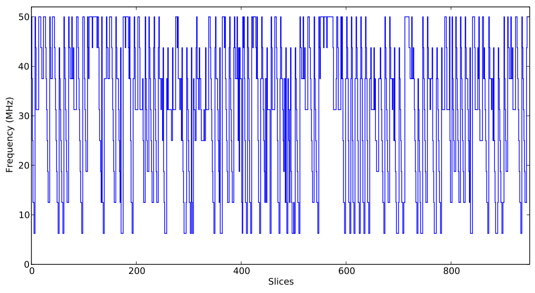


(a) $f_{min} = 6.25$MHz

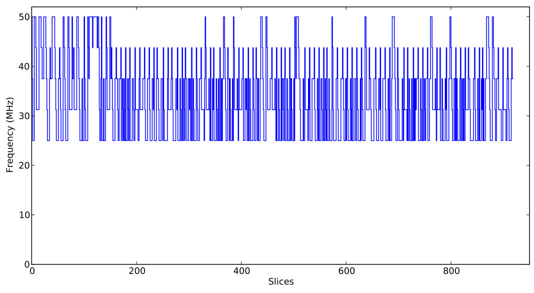


(b) $f_{min} = 0.5 \cdot f_{max} = 25$MHz

Figure 5.23: Frequency of Tile 0 with PWM workload



(a) $f_{min} = 6.25$MHz

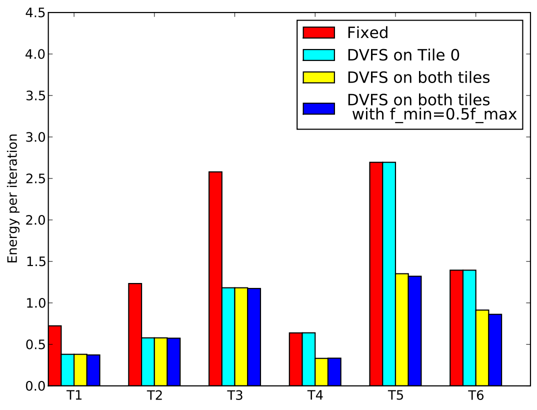


(b) $f_{min} = 0.5 \cdot f_{max} = 25$MHz

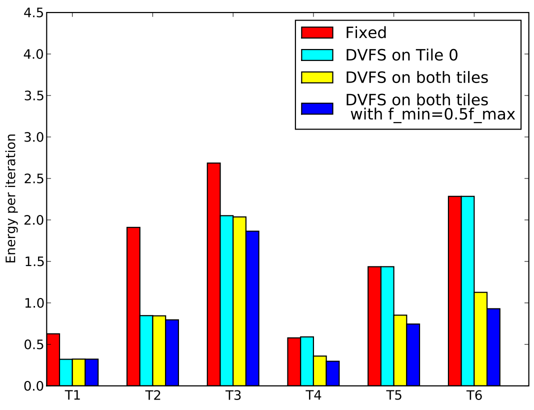
Figure 5.24: Frequency of Tile 1 with PWM workload

Tile0 Tile1	Fixed Fixed	SM+DVFS SM	SM+DVFS SM+DVFS	SM+DVFS+min SM+DVFS+min
T1	0.627	0.320	0.323	0.322
T2	1.910	0.847	0.844	0.796
T3	2.685	2.051	2.036	1.864
T4	0.578	0.590	0.359	0.296
T5	1.436	1.436	0.852	0.745
T6	2.284	2.284	1.127	0.930
Sum	9.5208	7.5283	5.5410	4.9524
Normalized	1.0000	0.7907	0.5820	0.5202
With OS overhead	1.0000	0.8182	0.6494	0.5843
Total slices	1807	1805	1976	1819
Used slices	1233	1435	1820	1757
Utilization	0.655	0.805	0.921	0.966

Table 5.12: Energy consumption for PWM workload



(a) Energy consumption for Random Workload



(b) Energy consumption for PWM Workload

Figure 5.25: Energy consumption of the test application

The result of this experiment shows that the platform has the ability to perform power management in real-time application. A few interesting points are discovered in the experiment. However, the detail study of power management policies is beyond the scope of this work, we leave it as a future work.

5.4 Summary

In this chapter, the setup up of the experimental platform, the test application and the experiments are described. This chapter shows that the design and implementation of the platform meets the requirements of this work – a composable and predictable MPSoC emulation platform with power management capability.

Chapter 6
Related Work

In this chapter, the related work is discussed. We start with MPSoC platforms similar to this work in Section 6.1. Then we take a look at the power management techniques in MPSoC design in Section 6.2. Finally, the FPGA-based emulation platforms for MPSoC design is discussed in Section 6.3.

6.1 Composable and Predictable MPSoC Platform

Composable MPSoC architecture templates are presented in [23, 30].

The time-triggered platform proposed in [30] is a multi-core platform targeting the automotive industry. It includes error correction and redundancy, and requires logical synchronicity and a global notion of time.

CoMPSoC is a template for composable and predictable MPSoC, where each application is given its own reconfigurable virtual platform [23].

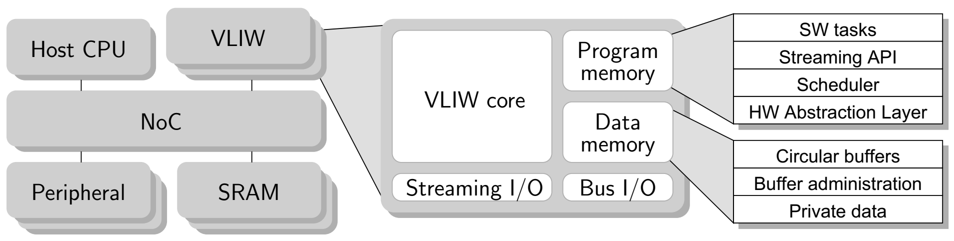


Figure 6.1: CoMPSoC architecture template, from [23]

In CoMPSoC, composability is provided by eliminating interference between applications through resource reservations, thus the cycle-level behavior of an application is independent of all other applications without placing any restriction on the applications. Predictability is achieved by using hardware resource budget enforcement to give a lower bound on resource availability. The template is given in Fig. 6.1, an instance with three VLIW processor cores is implemented on FPGA platform to demonstrate it. Due to the limitation of the VLIW cores, the processing elements are not shared between applications in CoMPSoC.

The work of this thesis uses a template similar to CoMPSoC, and extends it to support sharing processors between application. Using an operating system based on the work in [19], the processor is shared between applications in a composable way. Besides the basic composable and predictable resource sharing, the MPSoC platform in this work is able to perform per-tile composable and predictable power management.

6.2 Power Management in MPSoC

Vasanth Venkatachalam and Michael Franz [45] give an overview of different power management techniques for microprocessors. As mentioned in Section 2.3, the most promising way to save dynamic power is to lower the supply voltage, and dynamic voltage and frequency scaling (DVFS) is the obvious and the most promising solution.

A lot of studies have been done on the DVFS for MPSoC. For multi-processor system, obviously a chip-wide power management policy is not efficient, and per-core DVFS has been shown to offer larger energy savings [25, 43].

In [28], a trade-off between the benefit of per-core DVFS and the cost of on-chip switching regulators is carefully considered, and the result suggests that the on-chip regulator can provide fine grained DVFS, but the overhead of the regulators for multiple power domains can be expensive.

Beign et al. [7] introduces a scheme that uses *voltage hopping*, a technique similar to DC-DC converter, to generate an effective voltage and frequency for each unit in a GALS system. With such technique, each unit of the system has its own power and clock domain, and it is possible to perform fine grained DVFS for each unit. The power efficiency of the complete system is evaluated close to 95%, and the area overhead of the power supply unit is only 5% of the unit area.

To completely avoid the expensive on-chip regulator, [42] proposes *thread motion*, a scheme that uses cores with a number of fixed V-F levels and rapid thread migration between cores at different V-F levels to achieve a lower *effective voltage* for the thread execution, and increase the throughput at a given power budget. Thread motion avoids the DVFS module and enables the fine grained power management (at *ns* level). Currently, it is limited to homogeneous multi-processor platform only, but it is an interesting option for power management in MPSoC.

In this work we assume the existence of DVFS module for each processing tile, i.e. for each core, and per-core DVFS is performed. The power management is targeting streaming applications using data-flow model, which is similar to the work in [37].

6.3 FPGA-based Hardware Emulation

FPGA emulation is a widely used technique in multiprocessor system design, Rapid Prototype engine for Multiprocessors (RPM) [50] is an early example.

As power consumption is becoming the dominant factor in current IC design, there are raising attentions on using FPGA to emulate the power behavior of MPSoC [5, 9].

The work of [9] based on BEE [15], is a platform for power estimation of MPSoC with the following feature:

- Two LEON3 cores, AHB bus with snooping.

- Component-based power model, trace fine details in the system.
- The OS is linux, with knowledge of the power model.

Fig. 6.2 shows the architecture of the platform.

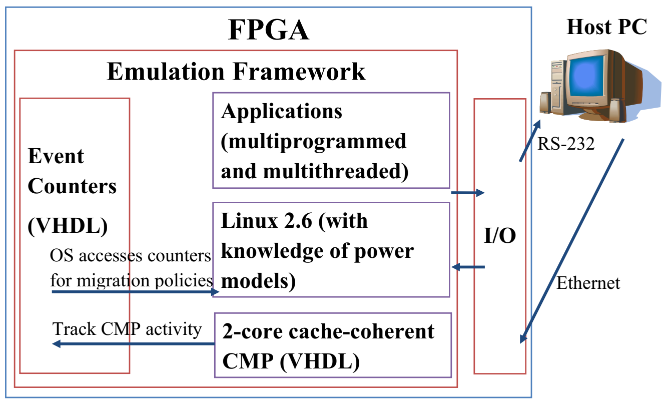

FPGA
Host PC
Emulation Framework
Event Counters (VHDL)
Applications (multiprogrammed and multithreaded)
RS-232
Linux 2.6 (with knowledge of power models)
I/O
OS accesses counters for migration policies
Ethernet
Track CMP activity
2-core cache-coherent CMP (VHDL)

Figure 6.2: FPGA based Power Evaluation Platform, from [9]

Most of such platforms are just for emulating the power-related behavior of the system. The platform in this work, however, can support the actual power management emulation. Also, the use of NoC brings the scalability of the platform to a new level.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

As technology advances rapidly, the use of MPSoC in embedded systems is becoming more and more common. In such systems, usually multiple applications are running on the same platform, which requires resource sharing in the MPSoC. The real-time constraints of the applications requires predictability of the MPSoC. To reduce the complexity of design and verification of multi-application systems, composability is required. And for embedded system, the power consumption is a major design constraint. Thus, we need composable and predictable power management on MPSoC.

In this thesis the design and implementation of a FPGA-based composable and predictable MPSoC emulation platform with power management capability is presented.

The MPSoC platform is an extension of CoMPSoC [23] platform template and the operating system is based on CompOSe RTOS [19]. A general processor core MicroBlaze is used on the processing tile in the platform. And with an composable RTOS running on the it, applications can share the processor in a composable and predictable way.

Inter-task communication in the system is improved in both hardware and software. In hardware, the design of a communication unit is presented. In software, the implementation C-HEAP protocol and the support of data-flow graph model are improved to cope with the real application.

Hardware and software infrastructure for composable and predictable per-tile power management of the processing tile are designed and implemented. In hardware, the power management unit (PMU) enables the simulation of DVFS on each processing tile. In software, a power management system is presented such that each applications on this platform are able to composable and predictable power management.

The platform is tested and demonstrated in a number of experiment. These experiments shows that the platform is composable and predictable, and composable and predictable power management is possible on such a platform. So the primary goal of this thesis is achieved.

7.2 Future Work

In this section, a number of improvements and useful features are presented as the possible future research areas.

7.2.1 Complete and More Accurate Power Model

The power model used in this work is simple and only considered the power consumption of the processing tile. A complete and more accurate power model for the system would be very useful for the research on power management.

7.2.2 Supporting Power Management for Different Types of Applications

The platform can perform power management on data-flow application with budgeting information. However, it is not limited to this type of application. The support for more general applications would be very useful.

7.2.3 Dynamic configuration and reconfiguration

The support for dynamic reconfiguration of the platform is useful, such as adding or deleting applications and tasks, dynamic reconfiguration of the NoC. In particular, the dynamic application loading is very interesting. Since the on-chip memory is limited in size, applications with large code size usually do not fit in the local memory completely. Possible solution including cache, and application loading via the NoC.

7.2.4 Cache Handling

Whether to use cache in the system is a design choice. In this work the cache is not used so that the analysis of predictability is simple. However, a lot of legacy applications require a lot of changes due to the missing of cache. It would be interesting to see how the cache can fit in such a composable and predictable system.

7.2.5 Supporting Fine-grained Power Management

Related work in Section 6.2 shows a trend towards fine-grained power management, which can exploit the dynamics in the application behavior. In this work, the decision point of the power management is the same as the scheduling point, which limits the granularity of the power management, as the task switching overhead can be high.

7.2.6 Improving Monitor Capability

The monitor in this system is simple, most of the work is done in software on the monitoring tile. However, this limits the system in two ways. First, the accuracy of the monitoring is not at cycle level when the time-stamping is done at the monitoring tile. Second, the scalability is limited by the number of FSL ports and the response time of the monitor. Improving the monitoring infrastructure is important for the emulation.

Bibliography

[1] Xilinx resource page[online]. `http://www.xilinx.com`.

[2] T.W. Ainsworth and T.M. Pinkston. Characterizing the cell EIB On-Chip network. *Micro, IEEE*, 27(5):6–14, 2007.

[3] F. Angiolini, P. Meloni, S. Carta, L. Benini, and L. Raffo. Contrasting a NoC and a traditional interconnect fabric with layout awareness. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, 2006.

[4] ARM Limited. *AMBA Specification (Rev 2.0).*

[5] David Atienza, Pablo G. Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, and Jose M. Mendias. A fast HW/SW FPGA-based thermal emulation framework for multi-processor system-on-chip. In *Proceedings of the 43rd annual conference on Design automation*, pages 618–623, San Francisco, CA, USA, 2006. ACM.

[6] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[7] E. Beign, F. Clermidy, S. Miermont, and P. Vivet. Dynamic voltage and frequency scaling architecture for units integration within a GALS NoC. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip (nocs 2008) - Volume 00*, pages 129–138. IEEE Computer Society, 2008.

[8] Marco Bekooij, Orlando Moreira, Peter Poplavko, Bart Mesman, Milan Pastrnak, and Jef van Meerbergen. Predictable embedded multiprocessor system design. In *Software and Compilers for Embedded Systems*, pages 77–91. 2004.

[9] Abhishek Bhattacharjee, Gilberto Contreras, and Margaret Martonosi. Full-system chip multiprocessor power evaluations using FPGA-based emulation. In *Proceeding of the thirteenth international symposium on Low power electronics and design*, pages 335–340, Bangalore, India, 2008. ACM.

[10] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258 vol.5, 1995.

[11] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. Cost considerations in network on chip. *Integr. VLSI J.*, 38(1):19–42, 2004.

[12] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 83–94, 2000.

[13] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems.* Springer, 1st edition, 1997.

[14] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen. Low-power CMOS digital design. *Solid-State Circuits, IEEE Journal of*, 27(4):473–484, 1992.

[15] Chen Chang, Kimmo Kuusilinna, Brian Richards, and Robert W. Brodersen. Implementation of BEE: a real-time large-scale hardware emulation engine. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 91–99, Monterey, California, USA, 2003. ACM.

[16] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT processor. *Micro, IEEE*, 29(2):6–16, 2009.

[17] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 174–179, Newport Beach, California, USA, 2004. ACM.

[18] William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th annual Design Automation Conference*, pages 684–689, Las Vegas, Nevada, United States, 2001. ACM.

[19] Marcus Ekerhult. *CompOSe: Design and implementation of a composable and slack-aware operationg system targeting a Multi-Processor System-on-Chip in the signal processing domain.* Master's thesis, Technical University of Lund, 2008.

[20] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *Design & Test of Computers, IEEE*, 22(5):414–421, 2005.

[21] Kees Goossens, John Dielissen, Om Prakash Gangwal, Santiago Gonzalez Pestana, Andrei Radulescu, and Edwin Rijpkema. A design flow for Application-Specific networks on chip with guaranteed performance to accelerate SOC design and verification. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, pages 1182–1187. IEEE Computer Society, 2005.

[22] Andreas Hansson. *A Composable and Predictable On-Chip Interconnect.* Phd's thesis, Eindhoven University of Eindhoven, 2009.

[23] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: a template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):1–24, 2009.

[24] IBM. *128-Bit Processor Local Bus Architecture Specifications Version 4.6.*

[25] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient Multi-Core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–358. IEEE Computer Society, 2006.

[26] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 1998 international symposium on Low power electronics and design*, pages 197–202, Monterey, California, United States, 1998. ACM.

[27] G Kahn. The semantics of a simple language for parallel programming. In JL Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 475, 471. North-Holland, 1974.

[28] Wonyoung Kim, M.S. Gupta, Gu-Yeon Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 123–134, 2008.

[29] H. Kopetz, C. El Salloum, B. Huber, R. Obermaisser, and C. Paukovits. Composability in the time-triggered system-on-chip architecture. In *SOC Conference, 2008 IEEE International*, pages 87–90, 2008.

[30] Hermann Kopetz, Roman Obermaisser, Christian El Salloum, and Bernhard Huber. Automotive software development for a Multi-Core System-on-a-Chip. In *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, page 2. IEEE Computer Society, 2007.

[31] R. Kotla, A. Devgan, S. Ghiasi, T. Keller, and F. Rawson. Characterizing the impact of different memory-intensity levels. In *Workload Characterization, 2004. WWC-7. 2004 IEEE International Workshop on*, pages 3–10, 2004.

[32] M. Krstic, E. Grass, F.K. Gurkaynak, and P. Vivet. Globally asynchronous, locally synchronous circuits: Overview and outlook. *Design & Test of Computers, IEEE*, 24(5):430–441, 2007.

[33] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[34] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[35] S.M. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, pages 721–725, 2002.

[36] Daniel Mattsson and Marcus Christensson. *Evaluation of synthesizable CPU cores.* Master's thesis, Chalmer's University of Technology, 2004.

[37] Anca Molnos and Kees Goossens. Conservative dynamic energy management for Real-Time dataflow applications mapped on multiple processors. In *Proc. Euromicro Symposium on Digital System Design (DSD)*, August 2009.

[38] Andr Nieuwland, Jeffrey Kang, Om Prakash Gangwal, Ramanathan Sethuraman, Natalino Bus, Kees Goossens, Rafael Peset Llopis, and Paul 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, October 2002.

[39] T.M. Parks, J.L. Pino, and E.A. Lee. A comparison of synchronous and cycle-static dataflow. In *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, volume 1, pages 204–210 vol.1, 1995.

[40] Philips Semiconductors. *Device Transaction Level (DTL) Protocol Specification. Version 4.2.*

[41] Johan Pouwelse, Koen Langendoen, and Henk 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, Rome, Italy, 2001. ACM.

[42] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: fine-grained power management for multi-core systems. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 302–313, Austin, TX, USA, 2009. ACM.

[43] G. Semeraro, G. Magklis, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, and M.L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 29–40, 2002.

[44] J.W. Tschanz, J.T. Kao, S.G. Narendra, R. Nair, D.A. Antoniadis, A.P. Chandrakasan, and V. De. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *Solid-State Circuits, IEEE Journal of*, 37(11):1396–1402, 2002.

[45] Vasanth Venkatachalam and Michael Franz. Power reduction techniques for microprocessor systems. *ACM Comput. Surv.*, 37(3):195–237, 2005.

[46] J. Wawrzynek, D. Patterson, M. Oskin, Shin-Lien Lu, C. Kozyrakis, J.C. Hoe, D. Chiou, and K. Asanovic. RAMP: research accelerator for multiple processors. *Micro, IEEE*, 27(2):46–57, 2007.

[47] Xilinx, Inc. *EDK Concepts, Tools, and Techniques.*

[48] Xilinx, Inc. *Fast Simplex Link(FSL) Bus.*

[49] Xilinx, Inc. *MicroBlaze Processor Reference Guide.*

[50] Koray ner, Luiz A. Barroso, Sasan Iman, Jaeheon Jeong, Krishnan Ramamurthy, and Michel Dubois. The design of RPM: an FPGA-based multiprocessor emulator. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 60–66, Monterey, California, United States, 1995. ACM.

Appendix A NoC Specification XML Files

A.1 Architecture Specification

```
<!DOCTYPE architecture SYSTEM ”../../etc/architecturegrm.dtd”>
<architecture id=”plb2dtl”>

  <parameter id=”clk” type=”int” value=”50” />
  <parameter id=”slotsize” type=”int” value=”3” />
  <parameter id=”wordsize” type=”int” value=”4” />
  <parameter id=”pckhdr” type=”int” value=”1” />
  <parameter id=”cmdsize” type=”int” value=”2” />
  <parameter id=”maxfc” type=”int” value=”31” />
  <parameter id=”maxpcklen” type=”int” value=”8” />
  <parameter id=”riqueue” type=”int” value=”8” />
  <parameter id=”niiqueue” type=”int” value=”8” />
  <parameter id=”nioqueue” type=”int” value=”8” />
  <parameter id=”link_pipeline_stages” type=”int” value=”0” />

  <!-- Next comes the actual IP blocks and their interfaces. -->
  <ip id=”monitor” type=”Host”>
    <port id=”pi” type=”Initiator” protocol=”MMIO_DTL”>
      <parameter id=”width” type=”int” value=”32” unit=”bits” />
      <parameter id=”address” type=”int” value=”0xb0000000” />
    </port>
  </ip>

  <ip id=”mb0” type=”IP”>
    <port id=”pi” type=”Initiator” protocol=”MMIO_DTL”>
      <parameter id=”width” type=”int” value=”32” unit=”bits” />
    </port>
  </ip>

  <ip id=”mb1” type=”IP”>
    <port id=”pi” type=”Initiator” protocol=”MMIO_DTL”>
      <parameter id=”width” type=”int” value=”32” unit=”bits” />
    </port>
  </ip>

  <ip id=”ram” type=”IP”>
    <port id=”pt” type=”Target” protocol=”MMIO_DTL”>
      <parameter id=”width” type=”int” value=”32” unit=”bits” />
      <parameter id=”delay” type=”string” value=”1” />
    </port>
  </ip>
</architecture>
```

A.2 Communication Specification

```
<!DOCTYPE communication SYSTEM "../../etc/communicationgrm.dtd">
<communication>
   <application id="application">

     <connection id="0" qos="GT">
        <initiator ip="monitor" port="pi" />
        <target ip="ram" port="pt" />
        <read bw="0.01" />
      </connection>

      <connection id="1" qos="GT">
        <initiator ip="mb0" port="pi" />
        <target ip="ram" port="pt" />
        <read bw="4" burstsize='128'/>

      </connection>

      <connection id="2" qos="GT">
        <initiator ip="mb1" port="pi" />
        <target ip="ram" port="pt" />

        <read bw="4" burstsize='128'/>
      </connection>

   </application>
</communication>
```