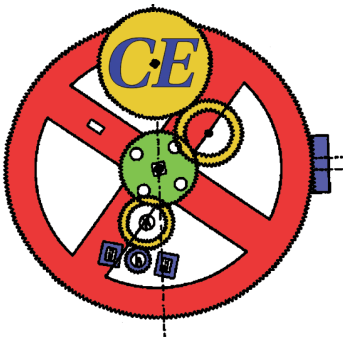


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2012

MSc THESIS

Dynamic loading and task migration for streaming applications on a composable system-on-chip

Adriaan van Buuren

CE-MS-2012-11

Abstract

Multiprocessor System on Chips (MPSoCs) offer more and more processing capability to embedded systems. As a result, an increasing number of applications run on one system, sharing MPSoC resources. Some of these applications are streaming applications and may have real-time demands, e.g., guaranteed throughput, hence their temporal behavior has to be verified at design-time. Moreover, applications may start and stop at run-time, creating a set of use-cases. The interference at shared resources introduces fluctuations in the temporal behavior of applications that may invalidate the results of the design-time verification. The CompSoC platform eliminates this interference by virtualizing shared resources, enabling independent, per-application design and verification. However, currently CompSoC requires that the entire source code to be compiled and linked into one executable. This code may belong to different applications and the developers of these applications have to hence share their source code, which is not always possible due to IP protection issues. Furthermore, resources may be wasted because a core's memory must be sufficient for all applications that ever run there and not only for the most demanding use-case.
In this thesis we augment CompSoC with dynamic loading and task migration mechanisms that allow separation of executables per application, enable memory reuse between use-cases, and, in general, increase flexibility, making possible, e.g., workload balancing. We implement this mechanism at application level, hence in a separate system application. This ensures that loading and migration do not interfere with running applications. More precisely, applications have independent temporal behavior from the moment they start until they are paused or stopped. Moreover, the dynamic loading and task migration processes execute in bounded time, thus are predictable, which makes them suitable for streaming applications. We implemented the dynamic loading and task migration on an FPGA prototype of CompSoC. Experiments indicate that the temporal behavior of a running application is not influenced by dynamic loading and task migration of other applications. We investigate the performance of the dynamic loading process for a jpeg decoder and an image rotation application and we detail the performance overhead involved in task migration. Furthermore, our implementation does not affect the critical operating system execution and the memory footprint overhead is only 4KB.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Dynamic loading and task migration for streaming applications on a composable system-on-chip

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Adriaan van Buuren
born in Rotterdam, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Dynamic loading and task migration for streaming applications on a composable system-on-chip

by Adriaan van Buuren

Abstract

Multiprocessor System on Chips (MPSoCs) offer more and more processing capability to embedded systems. As a result, an increasing number of applications run on one system, sharing MPSoC resources. Some of these applications are streaming applications and may have real-time demands, e.g., guaranteed throughput, hence their temporal behavior has to be verified at design-time. Moreover, applications may start and stop at run-time, creating a set of use-cases. The interference at shared resources introduces fluctuations in the temporal behavior of applications that may invalidate the results of the design-time verification. The CompSoC platform eliminates this interference by virtualizing shared resources, enabling independent, per-application design and verification. However, currently CompSoC requires that the entire source code to be compiled and linked into one executable. This code may belong to different applications and the developers of these applications have to hence share their source code, which is not always possible due to IP protection issues. Furthermore, resources may be wasted because a core's memory must be sufficient for all applications that ever run there and not only for the most demanding use-case.

In this thesis we augment CompSoC with dynamic loading and task migration mechanisms that allow separation of executables per application, enable memory reuse between use-cases, and, in general, increase flexibility, making possible, e.g., workload balancing. We implement this mechanism at application level, hence in a separate system application. This ensures that loading and migration do not interfere with running applications. More precisely, applications have independent temporal behavior from the moment they start until they are paused or stopped. Moreover, the dynamic loading and task migration processes execute in bounded time, thus are predictable, which makes them suitable for streaming applications. We implemented the dynamic loading and task migration on an FPGA prototype of CompSoC. Experiments indicate that the temporal behavior of a running application is not influenced by dynamic loading and task migration of other applications. We investigate the performance of the dynamic loading process for a jpeg decoder and an image rotation application and we detail the performance overhead involved in task migration. Furthermore, our implementation does not affect the critical operating system execution and the memory footprint overhead is only 4KB.

Laboratory : Computer Engineering
Codenumber : CE-MS-2012-11

Committee Members :

Advisor:	Dr. A.M. Molnos, CE, TU Delft
Chairperson:	Associate Prof.Dr. S.D. Cotofana, CE, TU Delft
Member:	Associate Prof.Dr.ir. J.S.S.M. Wong, CE, TU Delft
Member:	Prof.dr. K.G.W. Goossens, TU Eindhoven

Contents

List of Figures

List of Tables

List of Acronyms

IMEM Instruction Memory

DMEM Data Memory

CMEM Communication Memory

DDR Double Data Rate Synchronous Dynamic Random Access Memory

DMA Direct Memory Access

FIFO First In First Out

SA System Application

ST System task

RR Round Robin

TDM Time Division Multiplexing

MPSoC Multiprocessor System on Chip

NoC Network on Chip

MMU Memory Management Unit

ELF Executable and Linkable Format

PIC Position Independent Code

GOT Global Offset Table

PLT Procedure Linkage Table

CCB CompOSe Control Block

ACB Application Control Block

TCB Task Control Block

FCB FIFO Control Block

API Application Programming Interface

SDF Synchronous Data Flow

VLD Variable Length Decoding

IDCT Inverse Discrete Cosine Transform

CC Color Conversion

CAVLC Context-Adaptive Variable-Length Coding

LWC Local Write Counter

LRC Local Read Counter

RWC Remote Write Counter

RRC Remote Read Counter

LFC Local Finish Counter

RFC Remote Finish Counter

VFCU Voltage Frequency Control Unit

OS Operating System

ISA Instruction Set Architecture

LUT Lookup Table

FPGA Field-Programmable Gate Array

Acknowledgements

In the first place I want to thank my parents for their support and motivation as they have always been there for me. Next I would like to thank my friends for all the great times I had during my study and the help and motivation they have given me. Furthermore, I would like to thank my girlfriend for the love and motivation she has given me throughout the years.

For all the help during my thesis I would like to thank the CompSoC group, most of all Anca Molnos who has been a great help and where I always could walk by for questions. I also want to thank her for the patience and the useful feedback during the writing of this thesis.

I hope the CompSoC platform will have a bright future and my work will contribute to further research on this topic.

Adriaan van Buuren
Delft, The Netherlands
October 29, 2012

1 Introduction

Embedded systems are used in many products, ranging from telephones to televisions. These products would ideally require an inexpensive and small computation system to execute specific functionality. The growth of the processing capabilities allows these systems to be extended with more and more functionality. We can see this for example with televisions, which nowadays not only decode the incoming signal, but also capture and store video, and many have a network connection to stream videos from the internet.

A category of applications running on many embedded systems are streaming applications. These applications may have real-time requirements, such as minimum throughput for video decoding. Typically these applications are parallelized and executed on multiple processor cores [1] to fulfill their computation demands. This also reduces the power consumption, as multiple cores consume less power compared to a single core with the same processing power, given the same computation capability. For these reasons Multiprocessor System on Chips (MPSoCs) have been introduced. Such MPSoCs are complex and usually run a real-time Operating System (OS). Temporal verification of the application, the underlying OS, and MPSoC hardware should be done to make sure that real-time requirements are met. Furthermore not all applications that have to run on the MPSoCs actually execute concurrently. Applications may start and stop at run-time, creating a set of use-cases.

The wide range of products that embedded systems are used for and the fast growth of functionality of such systems, require their manufacturers to achieve a tight time-to-market at low design cost. These can only be achieved if the design of hardware blocks, OS modules, and applications [2] can be reused.

Reusing applications is not easy. When a single real-time application executes alone on an MPSoC, it is possible to verify its temporal behavior [3]. However, when multiple real-time applications share MPSoCs resources, each of the applications is dependent on the availability of these resources. When a resource is in use by one application, other applications need to wait, which means that one application potentially influences the temporal behavior of another application. Variation in temporal behavior could lead to situations in which the real-time requirements of an application are not met. To ensure that such requirements are always met, the behavior of these applications should be reverified when they are combined on a single MPSoC platform. Furthermore, if an application is updated, the other applications have to be also reverified. In this manner, the complexity of temporal verification grows exponentially with the number of concurrent applications, therefore a component-based development process is required [4]. Ideally, the temporal behavior of an application is independent of the behavior of other applications that may run concurrently. To provide this, composable systems have been introduced [5, 6, 7].

Currently applications and platforms are increasingly dynamic, in that the compu-

tation capacity, e.g., the clock frequency of the processors, as well as the application demands, e.g., computation requirements per quality-of-service level, may vary at run-time. To deal with this dynamicity and execute applications efficiently, e.g., provide a good power-performance trade-off, platform flexibility is require to adapt to the operating conditions.

CompSOC [6, 8] has the potential to offer easy application reuse. We hence assume CompSOC as a starting point for our work and we address several challenges related to system flexibility and adaptivity, as described in the next section.

1.1 Problem statement and goals

One current limitation of the CompSoC platform is that the entire source code, implementing all functionality that should ever run on a processor core, has to be compiled and linked into one executable. This source code may belong to different applications and should also include the OS functionality. The first problem that arises here is that the developers and/or vendors of applications have to share their source code. This is not always acceptable, because application vendors/developers may not want to make their intellectual property (IP) public to ensure future market advantages.

The second problem here is that already scarce MPSoC resources may be wasted because the memory available to a processor core must be sufficient to fit for all applications that ever run there and not only for the most demanding use-case. This may lead to over-dimensioning of the system and hence an increase in cost.

A third problem is that tasks are statically bound to processor cores. This limits the ability to adapt at run-time to operating conditions, such as the presence of hotspots (processors that are generating a lot of heat by continuously running on maximum capacity) to increase the reliability.

Furthermore, several requirements on the infrastructure that implements run-time adaptivity stem from the characteristics of embedded systems. First, the adaptation mechanism should not disrupt the execution of applications that are not involved in the reconfiguration. This is necessary to preserve real-time behavior of applications. Second, to put the premises of real-time adaptivity, the reconfiguration process has to have tight/useful temporal bounds, or in other words, it should be predictable. Third, the costs involved in reconfiguration should be as small as possible. Here costs include application performance, hardware area and software binary overheads.

1.2 Contributions

To reuse application designs and separate the creation of executables, we introduce dynamic loading and task migration. Dynamic loading also enables us to reuse allocated memory if an application is stopped. This means that the worse-case memory footprint is not the combined footprint of all applications, but only the footprint of the use-case that requires the largest amount of memory.

Furthermore we introduce a migration protocol that removes the need to bind a task to a processor statically. By moving tasks around, we can use otherwise unused

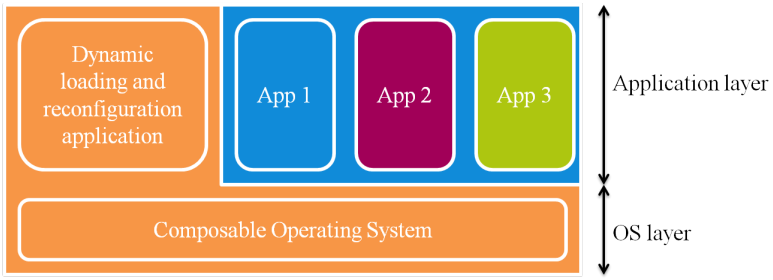


Figure 1.1: Layers in a composable MPSoC platform.

resources, and potentially reduce the power consumption of a tile and move tasks away from tiles that are producing too much heat.

Dynamic loading and task migration implementations have been proposed on a variety of MPSoCs [9, 10]. The unique feature we introduce in this thesis is that we provide independent functional and temporal behavior for each application, from the moment that it is started until it is suspended or stopped. To the best of our knowledge this is not offered by related approaches.

To offer realize this application independence, the dynamic loading and task migration is done in a System Application (SA). This application has one System Task (ST) mapped on each processor tile of the MPSoC and it executes at the application level (see Figure 1.1). CompSoC, the light-weight operating system running on each CompSoC processor, guarantees independent functional and temporal behavior of applications. Hence all running applications maintain this property, including the SA. However, the SA offers services to all other applications, which creates dependencies on loading and migration of these, hence our implementation offers composability only after applications have been loaded and started.

Due to the temporal and functional independence of running applications, design and integration will be easier compared to a non-composable system since verification can still be done per application. We can still verify applications independent, because compared to the current composable system we only need to add to the verification process the differences in start time. Although these variations in start time are dependent on other applications, applications can still be verified in isolation.

The verification needs to be extended as well for task migration since the start of the migration and variation in migration time need to be taken into account. The start time and migration time are predictable, hence its effects can be taken into account and the applications can be verified separately from other applications.

We also provide a distributed solution. When looking at existing migration techniques on non-composable MPSoCs, there is a central control element [10, 11]. Our protocol is designed with one ST per tile, and different STs can execute in parallel, hence our approach does not require a centralized memory model.

Next to offering independent execution time, the use of the SA minimizes overhead in the OS. By offering the dynamic loading and migration in the application level, the time-critical part where the operating system executes is not extended. Another

advantage of doing the dynamic loading and migration in the SA is that it allows the user to configure the system to have a good balance between the overhead in processing capacity reserved for the SA and the response time of the SA. The scheduler of the OS divides the processing time in time slots, for which the system designer can control what is executed in those slots. Hence, the user is able to decide how much time is reserved for the SA. By reserving more time for the SA the dynamic loading and task migration can have a faster response time, but other applications cannot run during this time and a larger overhead is introduced.

1.3 Related work

In this section we first discuss approaches to realize composable systems and then existing dynamic loading and task migration implementations.

Because most applications have real-time requirements, interference between them is undesired. For that purpose composability has been introduced [5]. Work on composable platforms is done in [6], [12], and [13]. In [13] the authors explain the need for composability and propose a resource manager to shift the burden of design-time analysis to run-time monitoring and intervention when necessary.

The authors of [6] present an MPSoC that eliminates the interference between applications through resource reservations. This system is extended in [12] to offer a temporally composable system on hardware and software level. There has been much effort into this system to offer more functionality and performance to show that it is possible to have an efficient and easy to use composable platform. Research on this hardware platform has been towards power management [14] and memory organization [15], and on the software level to maximize design reuse [16]. We extend this system with dynamic loading and task migration mechanisms, which, to the best of our knowledge have not been implemented on a composable platform and guarantee independent functional and temporal behavior of an application from the start of its execution until the application is suspended or stopped.

Dynamic loading and task migration are not new concepts concerning computing devices. In the domain of workstations with multiple processors dynamic loading and task migration has already been implemented and researched, and are used by many workload balancing algorithms integrated in the operating systems of those workstations.

However, these concepts are relatively new to embedded systems. General purpose dynamic loading and task migration approaches do not directly apply to embedded systems due to the constraints on cost and performance in these systems.

Furthermore, task migration mechanisms have been introduced in embedded systems, focusing on workload balancing while minimizing overhead. Dynamic loading is often assumed to be included in the task migration mechanisms and therefore in most work not the topic of research. Here we distinguish the work on homogeneous MPSoCs and heterogeneous MPSoCs systems.

Heterogeneous systems require a more complex mechanism since the architecture of the processing elements differ. In [17] a migration initiation technique is proposed to minimize overhead and reaction time on a heterogeneous MPSoC. It utilizes migration points on the MPSoC, following existing approaches that target larger computing systems

[18]. A migration point is a point within a task where its execution can be interrupted to reduce complexity of the migration.

Other work on heterogeneous task migration is presented in [11]. The work shows that classic task migration are not suitable for an Network on Chip (NoC) environment and thus it presents a migration technique tailored for a NoC emulation platform.

On homogeneous MPSoCs, task migration has also been presented. A feasibility study is done in [10], where the focus is on an MPSoC with a non-uniform memory architecture, an on-chip bus instead of a NoC, and one OS per core, thus having quite some similarities to the composable platform used in this work. The authors claim to be the first to present a detailed implementation of a migration mechanism in the MPSoC domain.

In [9] an adaptive MPSoC framework is described that uses a message passing programming model. This model enables the user to specify migration policies to for better workload balancing. The architecture is very similar to CompSoC as it has decentralized control, homogeneous array of processing elements, distributed memory and a scalable, NoC-based communication network. This work shows the restrictions of the dynamic loading process. It discusses the lack of a Memory Management Unit and thus virtual memory. To be able to reuse executables, the authors explain the use of Position Independent Code (PIC). In further work the authors evaluate the impact of the task migration in the MPSoC. Furthermore the authors note that the overhead of PIC for both memory and performance remain under the 5% [19], which is much better than the 39% performance loss introduced by virtual memory [16].

1.4 Overview

The summary of this thesis is as follows. In Chapter 2 we explain the terminology necessary to understand the dynamic loading protocol and task migration protocol. In Chapter 3 the design of our protocols is presented and we discuss the decisions and the restrictions of our protocols. In Chapter 4 we explain how the protocol works in detail and how it is implemented. In Chapter 5 we present experiments suggesting independent temporal and functional behavior of applications from the start until the execution of the application is suspended, and indicating the performance of the implementation. Chapter 6 contains the conclusion and future work ideas.

2 Background

Most of the applications that run on embedded systems have real-time requirements. This introduces the need for temporal verification, which is addressed by the concepts of predictability and composability. These concepts will be explained in the first section. In the next section, we will give an overview of the resources in an embedded system to explain how dynamic loading and task migration work in the following sections. In the last section we will give an overview of the platform we use to implement the dynamic loading and task migration.

2.1 Embedded systems requirements and terminology

We use a predictable and composable system to provide independent execution time to applications. In this section we explain these concepts and why they are so important for applications running on embedded systems. In the last part we will explain what we mean by independent execution time and how it maintains most of the advantages of a predictable and composable system.

2.1.1 Predictability

In embedded systems, many applications are streaming applications. These applications have real-time requirements, meaning they should be able to produce a certain throughput. For example, a video decoder needs a minimum execution time so it can process all the frames, since it is undesirable to skip frames. Every frame should be processed before a certain time, thus has a deadline. To be able to process the streaming data before a deadline, we need to be able to predict the time the processing of data takes, so it is possible to ensure the deadlines are met. An embedded system on which we are able to ensure deadlines are met is called a predictable system.

Next to undesirable behavior, missing deadlines could also lead to faulty behavior. For example, making the deadline too soon could lead to too much data in a buffer. Since buffer back-pressure is not always available, this could lead to overwriting data in the buffer or buffer overflow.

Furthermore, embedded systems with real-time requirements are often overdimensioned. When the processing capacity or memory required by an application is not constant, the worst-case execution time and the best-case might differ a lot. Because deadline misses should be minimized, the processing capacity is extended to finish the worst-case execution before the deadline. This introduces an overdimensioning of the system because the processing will often be finished before the deadline and not use the entire processing capacity.

To make sure of functional correct behavior of real-time applications and to reduce overdimensioning because of the limitations in resources in embedded systems, we require the system to be predictable.

2.1.2 Composability

More and more applications are able to run concurrently due to the growth of processing power in embedded systems. These applications share the resources of the Multiprocessor System on Chip (MPSoC), which means that the set of applications running on the system can influence the temporal behavior of an application. Where predictability provides lower bounds on the performance of applications, the application might not have the same temporal behavior when combined on a system with multiple applications.

The influence on temporal behavior can be split in response time, start time and execution time. Response time is the time an application requires access to a resource until the request is served. This can be influenced by other applications blocking a resource. The start time is the time the application is able to start using the resource. The execution time is the entire time the resource is used by the application. Composability provides complete independence between applications that are sharing resources, hence it provides the same execution time, response time and start time for access to resources such as the processor, memory, or the network [8].

This independence improves the verification of the real-time requirements of applications, as the applications can be verified in isolation. Verification of an application can be done by searching through all the possible states to see if they are temporally correct. Because the problem space of verification grow exponential by adding states, adding the states of another application creates a problem space that is impossible to verify due to lack of computational power. A composable system leaves out the need for the integration verification, as applications can be verified in isolation, simplifying the verification process.

2.1.3 Independent execution time

The definition of composability stated in [8] is focused on independent use of hardware resources. This concept holds also for shared applications, or OS functionality. If an application has a dynamic loading or task migration request, this should be handled. If this this functionality is not offered independently per application, it is not composable. This does not have to influence the independence on response time, execution time or start time of resources used by an application.

Therefore, if an application is started, and it is completely isolated and cannot affect any other application's functional or temporal behavior, nor does it affect the functional or temporal behavior of an application, we say that the system provides independent execution time.

Independent execution time still allows the applications to be verified in isolation, since there is no interference at shared resources at the time an application is executing. Next, our protocols are predictable which only introduces the need for verification with variation in start time of an application. The same holds for suspending and stopping an application.

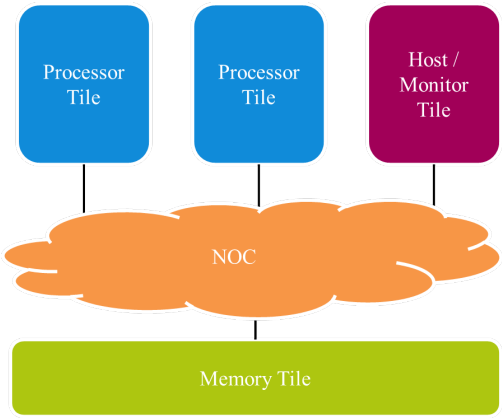


Figure 2.1: CompSOC platform instance.

2.2 Resources

To explain dynamic loading and task migration we will provide an overview of the resources available in an embedded MPSoC.

A MPSoC is a combination of processing elements, memory and an interconnect on a chip. A tile is the processor core combined with local memory, that can be connected to other tiles and a larger memory through the Network on Chip (NoC), as visible in Figure 2.1. The resources of an embedded system can be split up in processing capacity, memory and the communication infrastructure.

Multiple processors provide the processing capacity. We make a distinction between two types of systems, heterogeneous and homogeneous. We focus our work on a homogeneous system, which means that all the processing tiles are the same.

To preserve power by changing the frequency, the processors are all connected to their own Voltage Frequency Control Unit (VFCU). This could introduce a difference in the processing capacity per processor at a certain time.

The memory in embedded systems should be fast. By keeping local memories small, we are able to maintain fast access times. The first local memory is the Instruction Memory (IMEM). This is a read-only memory that contains all the instructions that need to be executed on the system. The second local memory is the Data Memory (DMEM), where the static and dynamic data is stored that a program uses.

An application consists of tasks that have a computation function that is executed on the processor. In most applications, these tasks need to communicate with tasks on other processors or need to access the larger remote memories. For this purpose local shared memory, Communication Memory (CMEM), is introduced. With the use of Direct Memory Access (DMA) modules, that takes care of the data transfers, we preserve the fast access time for the shared local memory [15]. To maintain composability, the DMA modules are not shared between applications.

The remote memories are very often slow, but have larger capacities. In the used system we have access to Double Data Rate Synchronous Dynamic Random Access Memory (DDR), which is not situated on the chip and therefor has a large access time,

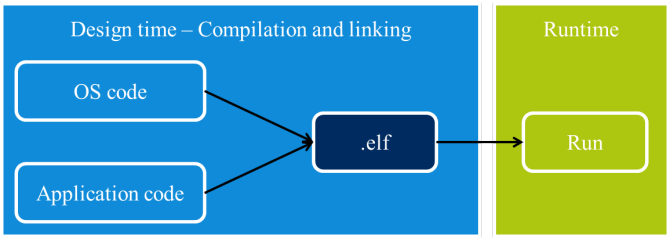


Figure 2.2: Static loading.

but compared to the size of the local memories (up to 256KB per memory) the DDR is very large (2GB).

To connect everything on the chip, a NoC is used. This creates the possibility to create direct connections between all elements on the chip, and with that can provide independent data transfers between the tiles. It also allows easy reconfiguration, simplifying the design of the system.

2.3 Dynamic linking and loading

To execute an application on a MPSoC, the application must be compiled and linked into an executable file, after which the instructions and data need to be loaded in the memory of the system. Dynamic loading is introduced to be able to reuse the executables, but it also introduces more flexibility with these executable files. Dynamic loading enables that the set of applications that is running on the system can be changed, without having to create new executable files. We call the set of running applications on a system and the mapping of the tasks of the applications on the processing tiles a use-case. It is fairly normal for a system to not continuously run the same set of applications all the time. This means that we are able to free resources that are not in use by an application and use it for other applications, increasing our system performance.

The process of loading exists of loading the instructions that a processor can execute into memory, and loading the static data that these instructions refer to. These instructions and static data should all be mapped to physical memory addresses. All this information is combined in the executable file. The Executable and Linkable Format (ELF) is a well-known file format for executable files, which we use as well. An ELF file contains all the instructions and static data for an application, with memory address information, so an Operating System (OS) knows where to put all this information in memory.

To know where the instructions and data are in an ELF file, the file is split up into sections that contain information on where they should be mapped to in memory. There are for example data sections available such as .data for initialized variables and the .bss section for all data that needs to be initialized with the value 0. We are also able to create our own sections to be able to place functionality on specific addresses. For every section the mapping to a physical address, the size and the start address of the section is stored in the ELF file. With this information, the instructions can be loaded on their predefined memory address and we can run an application.

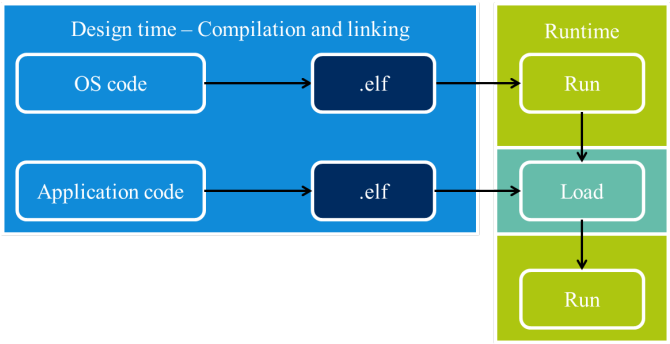


Figure 2.3: Dynamic loading.

The time at which we load the instructions and data into memory can variable. We distinguish static and dynamic loading. The main difference is that in dynamic loading, the instructions and data are loaded at runtime, whereas in static loading the instructions are loaded at boot time. Dynamic loading introduces the need for resolving addresses at runtime. In static loading, a single executable is created and loaded into memory (see Figure 2.2). This means that all the references are resolved at design time and the addresses of functions and data do not change at runtime.

When the instructions and data are dynamically loaded, we can create several executable files that are loaded into memory while the OS is running (see Figure 2.3). The executable can be loaded in any place in memory, but references to data and to external functions need to be resolved. The best way to solve this is dynamic linking and the use of virtual memory, but these options are not always available in embedded systems due to their high cost.

Another option is to build the executable such that it uses Position Independent Code (PIC). This means that the instructions and data can be loaded in any place in memory by using indirect addressing for data references, and relative addressing for jumps in the instructions. By adding the PIC flag to the compiler, the data references in the executable file become indirect, as opposed to static data references. Indirect addressing enables us to load the data anywhere in memory because a Global Offset Table (GOT) is used for the indirect addressing of the data. By using a register in the processor to store the address of the start of the GOT, the code can be compiled with a reference to the start of the table plus an offset to the address of the data. Because we can easily change the contents of this table and the content of the register pointing to the start of this table, the data can be mapped to any physical address. The advantage of static references is that it is faster than indirect addressing because indirect addressing requires an extra step where the address of the data is retrieved from a table.

Sometimes we also require calls to functions outside of the application executable. The addresses of these functions file are not always known at design time. This requires dynamic linking, which is the process of retrieving these addresses at runtime. A library with the instructions of the external function is loaded into memory when needed and the dynamic linker finds the addresses and creates the correct references.

2.4 Task migration

For the purpose of workload balancing, power management or resource management, it might be necessary to move a task from one place in memory to another, and even to another tile. Task migration introduces the flexibility to move task around for these purposes, as we can change the mapping of tasks of an application on processing tiles, while maintaining the state of a task. By switching between use-cases we can optimize our applications dynamically for the system. When for example an application has a very computational intensive first task that only does work at the start of the application, it could improve performance if other tasks are moved to the tile that the computational intensive task is running to make use of the processing power that was needed by the first task, but not used any more after it finished.

Task migration involves replication of a task, or recreating the task and then moving the entire state of a task. Because the state of a task exists of data in local and remote memory, migration pointes have been used to have a clean state in which can be migrated. The recreation of the task is done by dynamically loading it.

2.5 Platform

In this thesis we use an instance of the CompSoC [6][20], a composable system-on-chip platform. This system, together with the OS CompOSe [12], offers predictability and composability to all applications. In this section we will fist describe the hardware platform CompSoC. Then we will describe the application model and CompOSe that supports this application model. As last part we will describe the existing setup for dynamic loading.

2.5.1 CompSoC

CompSoC is a system build up from asymmetric processor tiles. This means that the Instruction Set Architecture (ISA) is the same for every tile, but every tile runs its own instance of CompOSe, and every processing tile has its own VFCU to control the frequency. Figure 2.1 shows an instance of the system. There can be any number of processor tiles, connected through a NoC. The NoC also connects the tiles to a host/monitor. The host/monitor is a special tile that configures the NoC and processes debug information. Next to the memory on the tiles, there is also one large DDR available, which is currently only composable if it is used by a single application.

Figure 2.4 is a visual representation of a single processor tile. Each tile runs its own instance of the operating system CompOSe [12]. A processor tile consists of various building blocks, including the processor, memory and a VFCU to vary the frequency of the tile. The processor is a Xilinx Microblaze that can interact with local and shared memory. The local memory is split up into IMEM and DMEM, whereas the shared memory is split up into CMEM out and CMEM in. The CMEM in is directly connected through the NoC while the CMEM Out is connected to the NoC via a DMA module. This setup of shared memory with DMA modules allows fast local access time and it

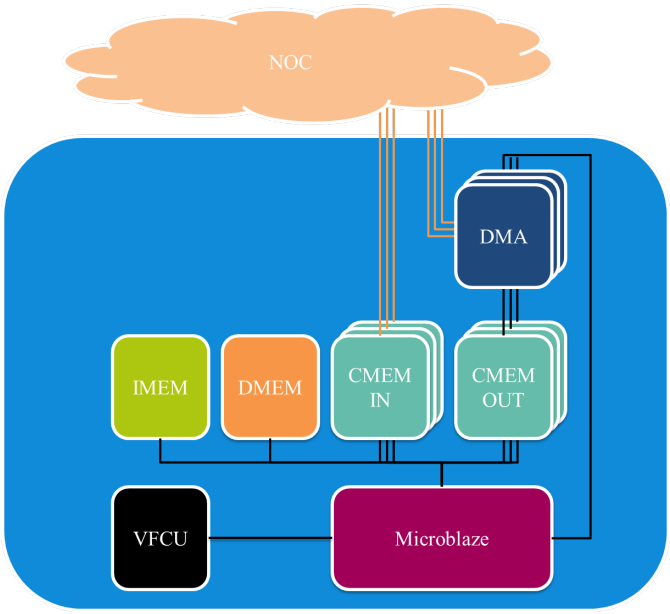


Figure 2.4: CompSoC tile.

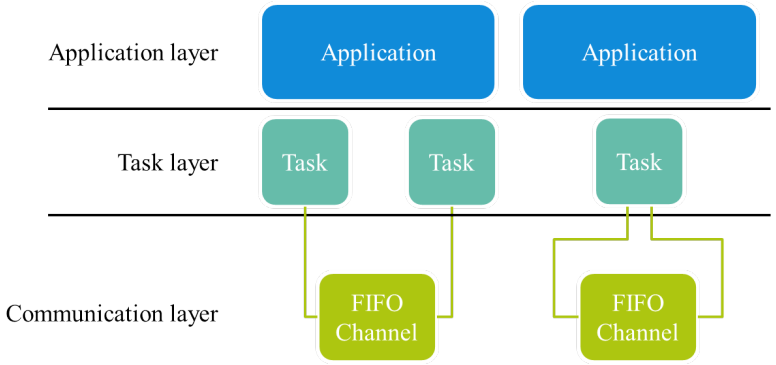


Figure 2.5: Application model.

gives very good data availability and maximizes performance for remote data transfers [15].

2.5.2 Application model

Our application model consists of different abstraction layers as shown in Figure 2.5. At the application level, power management is done and the tasks are scheduled. These applications are programs like video decoders and can consist of several tasks. Examples of the dataflow graphs of tasks used in this thesis can be found in Appendix C.

In the task layer, the input data of a task is retrieved, then the instructions are executed by calling the task computation function and last output data is produced. This process is iterative, so after the output data is produced, the task is idle until the task is scheduled again and the process is repeated.

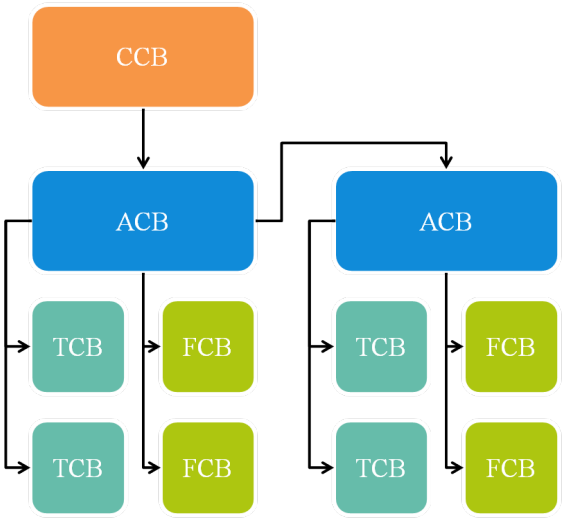


Figure 2.6: CompOSe Control Blocks.

The input data that instructions of the computation function consumes and processes and the output data that the computation function produces comes from the communication layer through First In First Out (FIFO) channels. This means that the data produced first in time, will be consumed first in time. A FIFO is connected to a task via a port to distinguish the FIFOs channels. It is possible to have self FIFO channels, which means that one task consumes and produces data on a FIFO channel. Self FIFO channels are used to save data between iterations of the task. This could mean that for the first iteration, a task will need an initial token to begin the execution.

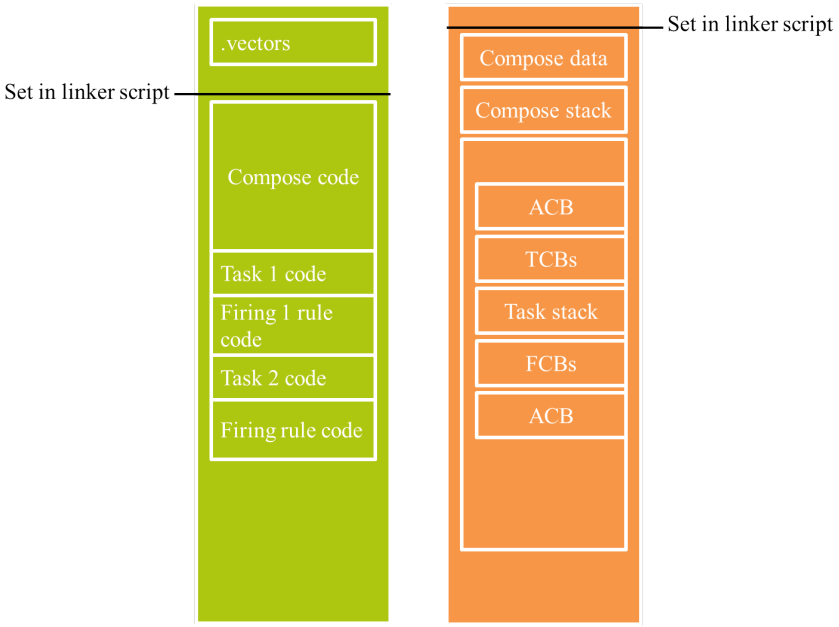
Sometimes it might change per iteration of a task what FIFO channel is consumed or produced on. To keep track of the data that needs to be consumed and produced, the task has a FIFO rate table with producing and consuming rates of the ports the FIFO channels are connected to.

To schedule a task, the FIFO rate tables are used by the task scheduler to check if data is available on all FIFO channels that the task consumes from this iteration. Then the scheduler checks if there is space on the FIFO channels to produce data. When a task is scheduled, it will first consume the FIFO channel data. Then it will execute the task function and write away data to the FIFO channels. To update the producing and consuming rate tables, the firing rule function is called. Then the iteration is finished and the application layer can schedule a new task.

2.5.3 CompOSe

CompOSe [12] is the OS that runs on every processor tile and implements the previously described application model (Figure 2.5). CompOSe uses control blocks to store information.

There is a control block for the OS, for the applications, tasks and FIFO channels. These control blocks are linked as shown in Figure 2.6. The CompOSe Control Block (CCB) contains information about the OS such as the list of applications, the list of



(a) IMEM (b) DMEM

Figure 2.7: Local memory structure.

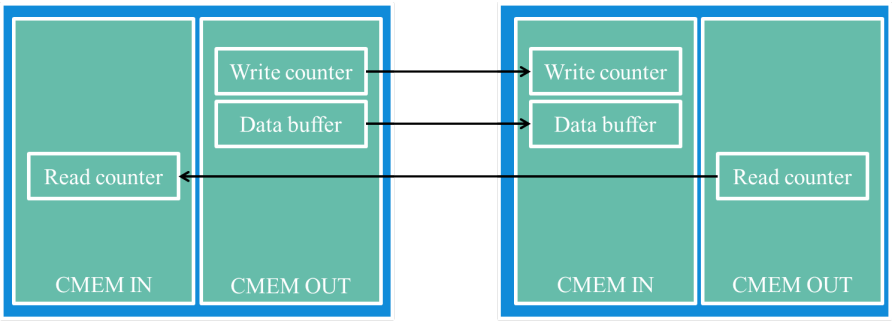
available DMA modules, and the application scheduler information. All the control blocks are stored on the heap of CompOSe as shown in Figure 2.7b.

The application information is stored in an Application Control Block (ACB) and contains a pointer to the next application in the list, the task scheduler information, and power management information. The ACB also contains a list of Task Control Blocks (TCBs) and FIFO Control Blocks (FCBs).

The TCB contains all the information about a task, such as the pointer to the task computation function, the stack size and stack pointer, timing information, and power management variables. The task computation function of a task consists of instructions that are stored in IMEM. In the existing system, all code is compiled together and therefore the CompOSe instructions, task instructions and firing rule instructions are all in the same section of the IMEM (see Figure 2.7a). The start of this section is defined in the linker script of the executable that is created for a tile. All the static data that is used by the task computation function is combined with the static data used by CompOSe in DMEM. The address of this data is also set in the linker script of the same executable. The stack of a task is also in DMEM, but allocated on the heap of CompOSe.

For the communication to other tasks, the TCB contains a list of FCBs of FIFO channels it is consuming tokens from and the FCBs of the FIFO channels the task is producing tokens to. The TCB also contains the firing rule information including the firing rule function and FIFO rate tables. These tables contains information on what FIFO channels the task needs to consume or produce tokens the current iteration.

The FCB contains the information of the communication channel. When a FIFO channel has the producing task and the consuming task on the same tile, there is one



(a) Produces task FIFO data (b) Consumer task FIFO data

Figure 2.8: Shared memory structure.

FCB containing all the pointer information. This information includes all the pointers to the counters and data as explained in Appendix B. It also includes information on the amount of tokens the tasks consume or produce and counters to keep track of the amount of data and space in the data buffer.

When the producer and consumer of a FIFO channel are mapped on different tiles, one FCB of the FIFO channel is at the consumer tile and the other FCB is at the producer tile. At the consumer tile the FCB is only used to store the information about the task that is consuming tokens and the amount of tokens that the tasks consumes per iteration. The FCB on the tile with the produces is only used to store information about the producer and the amount tokens that is produced every iteration.

To store the FIFO channel data CMEM is used. CompOSe uses the C-HEAP protocol [21] that requires a data buffer, a write counter and a read counter. Appendix B contains a more elaborate explanation of the protocol and its counters.

In CompOSe the FIFO channels are managed by pushing data instead of pulling data. That means that all the data and counters that has to be send to a remote tile, should be in CMEM Out. In case the FCB is at the producer tile, the write counter and a data buffer is stored in CMEM Out. As shown in Figure 2.8, write counter and the data buffer on the tile with the consumer end up in CMEM In.

CompOSe implements a two level scheduling scheme that allows applications to run independently. The two schedulers are an application scheduler and a task scheduler. The application scheduler is part of the OS and schedules the applications. Throughout this thesis we will assume it is a Time Division Multiplexing (TDM) scheduler since this is the default policy that ensures composability. The time in which CompOSe is running is called the OS time slot. In this time slot CompOSe also saves and loads the context of applications, and does application scheduling. We call the time the application is running the application time slot.

The task scheduling is done in the application time slot. The default configuration, and assumed in this thesis, is the Round Robin (RR) scheduler. Figure 2.9 shows the default configuration and time slot distribution. It also shows the state that the tasks go through if they are scheduled.

The task scheduler will always schedule the idle task when no other task can execute. To check if a task should be scheduled, the FIFO rate tables are checked. The FIFO rate

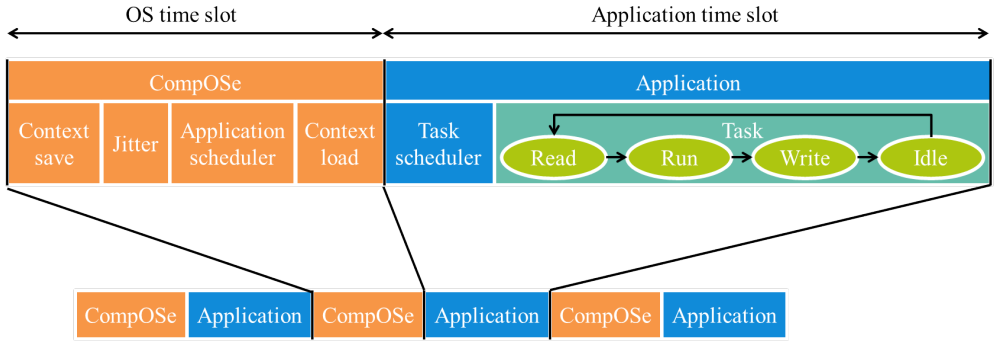


Figure 2.9: CompOSe application scheduler.

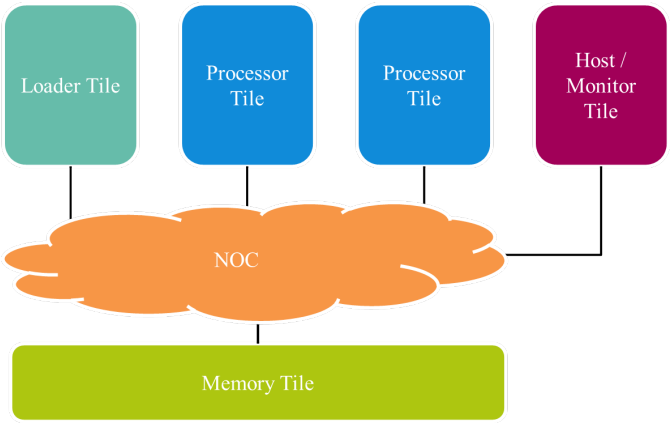


Figure 2.10: CompSOC platform overview of the existing dynamic loading project.

tables define the rates at which a task is producing tokens on the FIFO channel and at which rate a task is consuming tokens from the FIFO channel.

If the FIFO rate table defines that a task consumes from a FIFO channel, the scheduler checks if there is data available on that channel. If the FIFO rate table defines that a task produces tokens on a FIFO channel, the scheduler checks if there is space available on that channel. If all input FIFO channels have data and all output FIFO channels have space, the task is scheduled.

After a task is scheduled, it goes through the Read, Execute, Write and Idle state as shown in Figure 2.9. The task will start consuming data from the input FIFO channels after which it executes the instructions that the TCB points to. The produced data is then written to the FIFO channels and the task ends up in the idle state.

At the end of the task iteration, a function called the firing rule is executed. The firing rule updates the FIFO rate table according to the current CSDF cycle.

2.5.4 Existing dynamic loading support

CompSoC provides an initial implementation of dynamic loading. This implementation is based on a loader tile that runs a function, which distributes loading information to

an application called the System Application (SA) on the tiles that run CompOSe. The extension of CompSoC with the loader tile is visualized in Figure 2.10.

The loader tile writes data to the FIFO channel buffers and updates the counters in the other tiles via a DMA module. On the other tiles, the SA calls the CompOSe API functions to add ACBs, TCBs and FCBs.

The information that is send over the FIFO channels consists of parameters that were necessary to call the CompOSe API function. This information was packed in new structures containing the application, task and FIFO channel information. Since these were of variable size, the FIFO channel buffers needed to be as large as the largest structure.

But not all information was send over the FIFO channels. The IMEM is not writable from the processor and thus for loading the instructions into IMEM, a new DMA module was introduced that is connected to the IMEM to be able to pull the instructions from the CMEM of the loader. Another DMA module was introduced connected to DMEM, because the data sections were quite large so directly loading the static data sections into DMEM saves CMEM and time because it does not require pulling the static data in CMEM of the processing tiles and copying the data from CMEM to DMEM.

The instructions and data needed to be pulled from the CMEM of the loader tile to the IMEM and DMEM. Because the CMEM is restricted in size, the ELF files were stored on a flash card. The processor on the loader tile parsed the ELF files at runtime and copied only the needed information to the CMEM.

Parsing the ELF files at runtime introduced some requirements for the ELF file structure. Specific naming for sections was required to load the task code and without variations in the linker script for every tile, only be one ELF file can be loaded since static data references were made. The last step was to send an instruction and descriptor over the FIFO channel from the loader tile to the processing tile to let the processing tile know where it should copy the instructions and data to.

This setup has as mayor disadvantage that there the static data could not be loaded anywhere in memory. If static data is used, all the tasks that are loaded on a tile should be in one executable, or if multiple executables were created, the linker scripts should be adjusted to let the mapping of static data to physical memory addresses correspond to where the data is loaded. If a different set of tasks is mapped on a tile, these predefined addresses can overlap, which requires changes to the linker script, hence a new executable.

Another disadvantage is that the loader tile was not able to handle multiple requests. The steps of loading an application were done sequentially. Next, parsing the ELF files took quite some time, which made the loading process slow. Furthermore, for this solution descriptors for loading information were introduced that are very similar to the control blocks in CompOSe. By not using the CompOSe control blocks the system was made more complex than necessary.

2.6 Summary

This chapter contains the background information required for understanding our implementation of dynamic loading and task migration. We start with the explanation of the concepts of predictability, composability and independent execution time. A system

is predictable is all interference between applications is bounded in time. A composable system guarantees that applications are completely isolated and cannot affect each other's functional and temporal behavior. A system that guarantees independent execution time, guarantees independent temporal and functional behavior of applications from the start of their execution until the application is suspended.

In the next section we provide an overview of the resources used in embedded systems. This includes the explanation of the memory infrastructure and the communication infrastructure. After this we give an brief explanation of the concepts of dynamic loading and task migration.

In the last section of this chapter we describe CompSoC, the used platform in this thesis, and CompOSe, the OS running on the platform. We also describe the application model here. It consists of applications that have tasks. These task have an computation function, a firing rule function and FIFO channels to transfer data to other tasks. Every iteration of a task it can consume and produce on FIFO channels and executes its computation function. The tasks contains FIFO rate tables to keep track of the FIFO channels the task consumes from and produces on. In every iteration these tables are updated with the firing rule function.

3 Concept and design

In Section 1.1 we state the following goals:

1. Maximize design reuse.
2. Minimize the amount of needed resources, in particular reduce worst-case memory usage.
3. Maximizing runtime flexibility by enabling use-case switching
4. Provide independent execution time to applications executing on the platform
5. Minimize the extension of OS execution time
6. Minimize the time needed to load an application
7. Minimize reaction time of the migration protocol
8. Minimize the time needed to migrate a task

To satisfy the first three goals, we introduce a dynamic loading protocol and a task migration protocol. Goal 1 is satisfied by dynamic loading as it enables us to create the executables for the applications separately and reuse those executables on different systems. Next, dynamic loading enables free up resources for new applications, satisfying goal 2. By stopping and loading new applications we also can change the set of running applications, hence partly satisfy goal 3. The introduction of task migration satisfies goal 3 completely as it allows us to maintain the state of a task while changing the mapping of tasks on tiles, hence providing even more runtime flexibility.

The most important goal that we want to satisfy is to provide independent execution time (goal 4) since it has the similar advantages as composability as explained in Section 2.1.3. Next, the Operating System (OS) is light-weight and does not need many resources, most of all in processing time. Therefore the he addition of dynamic loading and task migration support should not extend the execution time of the OS (goal 5).

In the rest of the goals we set the requirements for the performance of the dynamic loading and task migration. For dynamic loading we want to load an application as fast as possible, hence minimize the time needed to load an application (goal 6). The goals for task migration are split in reaction time (goal 7) and migration time (goal 8). The reaction time is the time needed to start migration after a request for migration is done. This delay is introduced by the use of migration points that are discussed in Section 3.1.4. The migration time is the time needed to migrate a task. The reason for making a distinction between the reaction time and migration time is that the reaction time depends on the execution of the task that is migrated, while the migration time is not influenced by the execution of the task that is migrated.

To enable dynamic loading and task migration we introduce the System Application (SA). In the first sections we describe how the SA works and what design decisions have

been made to satisfy our goals. In the following section we describe the protocols at a high level.

3.1 Design decisions

A SA is introduced to enable dynamic loading and task migration. The SA is an application that runs under CompOSe and since CompOSe offers functional and temporal independence of all applications, the SA is an independent application as shown in Figure 1.1. This also means that the SA has no influence on the applications that have been loaded. This way we provide independent execution time, satisfying goal 4.

The execution time OS slot is time-critical, because it is executed between all the application slots. We minimize the execution time of the time-critical OS slot (goal 5), since all dynamic loading and task migration processing is done in the application layer. Another advantage is that if dynamic loading or migration is not needed, this application can simply be removed and its resources can be used for other applications.

The SA has a task per tile, the System task (ST), that does the dynamic loading and task migration. The ST processes every iteration a ST instruction. A ST instruction also contains a pointer to a descriptor that contains loading information. We will explain the instructions and descriptors in more detail in Section 3.1.3.

Figure 3.1 shows the global setup of CompOSe with the SA. First the monitor tile initializes the board, and stores all the ST instructions and descriptors that contain loading information in the DDR. The use of DDR will be explained in Section 3.1.2. Then it loads the Executable and Linkable Format (ELF) files into DDR. These ELF files contain the computation functions of the tasks and the static data the computation function uses. The SA can put the computation function and the static data at different addresses, which introduces some requirements to the compilation and linking of the ELF files. We explain this in Section 3.1.1 and Appendix A. The next step in Figure 3.1 is that the monitor sends the start address of the ST instructions to the processing tile so the ST knows where it should pull the first ST instruction from. After synchronization of the tiles, a CompOSe instance is created and the ST is added and scheduled.

When the ST is scheduled, it pulls instructions together with a pointer to a descriptor with loading information from DDR. After the ST has processed the instruction, it pulls the descriptor and then calls the necessary CompOSe Application Programming Interface (API) functions.

In the following sections we will explain the design decisions in more detail.

3.1.1 Position independent code

To maximize design reuse, we create separate executables. The ST must be able to store the task instructions and static data in these executables anywhere in Instruction Memory (IMEM) and Data Memory (DMEM). For dynamic loading, references to data and functions outside of the task code (for example CompOSe API functions) are the major issue when mapping the task instructions and static data to a physical addresses. This can be solved by the use of virtual memory, which is supported on our platform [16], but the performance analysis indicates that the introduced overhead is too large.

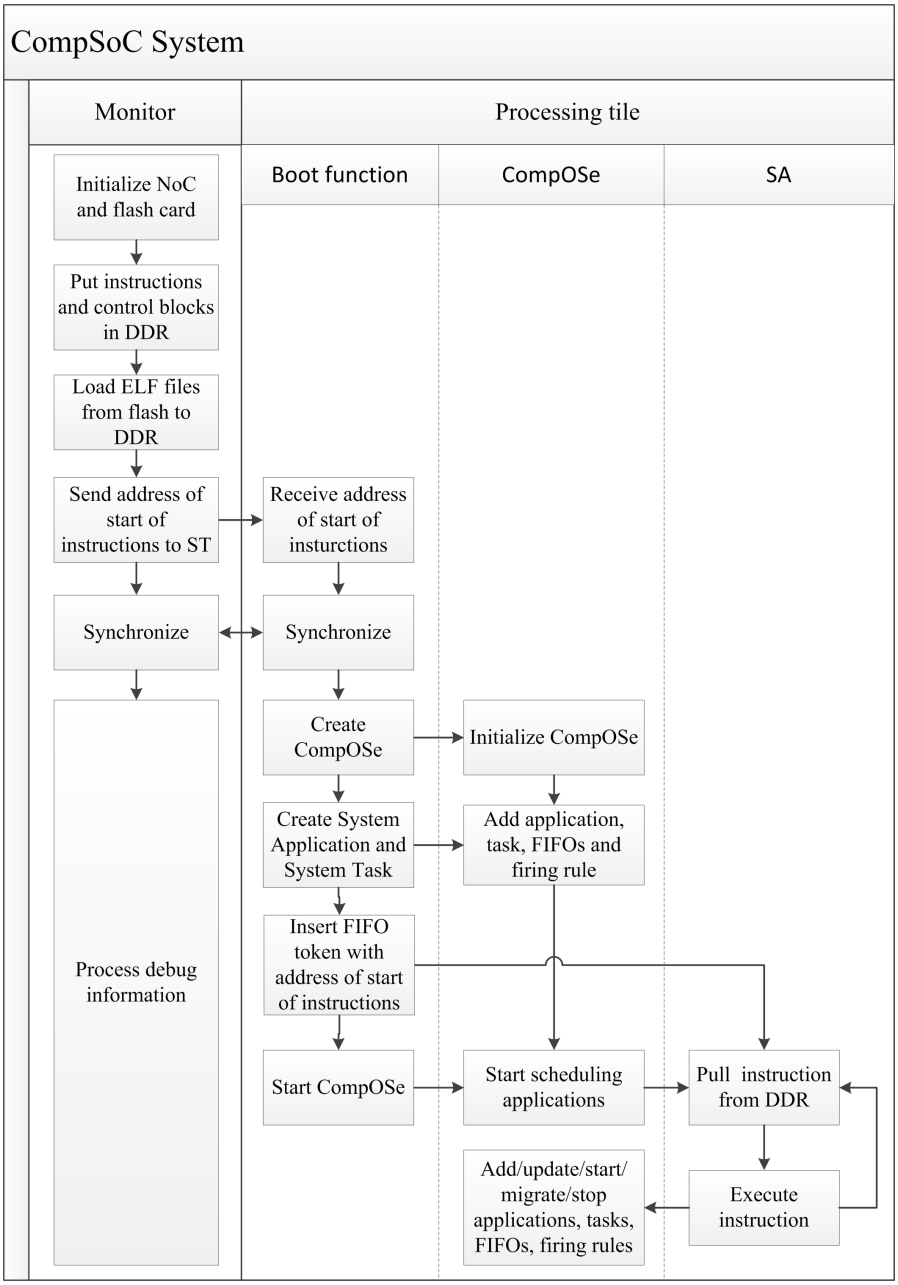


Figure 3.1: Support for dynamic loading and task migration concept.

To make use of CompOSe API functions, we introduce indirect function calls. We create a table in the executable that is compiled for every tile. This table contains all the addresses of the CompOSe API functions that are required by the applications. In the linker script we define the memory address of this table.

From the application executables we refer to this table. A header file is added to the task source code that contains the address and the layout of the function table. The

file also contains definitions that redefine the function calls to first retrieve the function address from the table, and then call the function on that address. This allows us to reuse current applications without having to remove the CompOSe API functions from the source code or keep track of the addresses of all these functions to create static references in the task source code.

When we use static references to the physical addresses of data, we are not able to load the data anywhere in the DMEM. If static references are used, the static data should be placed on the same address as defined in the linker script. If two tasks use the same memory range, the executables cannot be reused, and changes to the linker script are required to allocate memory that is not used by other tasks. Since our goal is to maximize design reuse, we require the static data to be position independent. To change the references to static data, we use Position Independent Code (PIC) flag in the MicroBlaze gcc compiler. Similar to the indirect function calls, a table is used so the instructions are compiled to use indirect addressing of the static data.

3.1.2 Distributed setup

We converted the existing dynamic loading to a more distributed setup where the ST fetches the loading information from a place in memory. We removed the central processing tile that was introduced in the existing dynamic loading on CompSoC and decided to store the loading information in DDR. We use the DDR because of its size. Note however that there is no fundamental limitation on where the data is stored and thus both local and shared memory can be used.

The distributed setup allows the ST to load an entire application on a tile independent of the other tiles, but if an application is mapped over several processing tiles, we do require synchronization between the STs on the tiles the tasks are mapped on. This introduces the need for communication channels between all the STs, exponentially growing when adding more processing tiles with STs. Since we only transfer 32 bit values over these communication channels to synchronize, adding these communication channels will neither gives much memory overhead, nor requires a lot of bandwidth on the Network on Chip (NoC). Therefore we expect that this solution is scalable enough to provide extensions of the number of processing tiles for the platform in the near future.

3.1.3 Application descriptors

The information that the ST fetches from DDR is divided up in an instruction and a descriptor pointer (see Figure 3.2). This is generalized for everything the ST does and therefore allows a readable source code in the monitor for storing these instructions into DDR.

Next to this control sequence, the monitor loads ELF files with task instructions and static data, and the descriptors with ST instruction specific information.

In the existing dynamic loading project, new structures were introduced to store loading information. We use the CompOSe control blocks to define our loading information. The descriptors are corresponding with the CompOSe control blocks so we have uniformity throughout the system at the cost of some overhead. This overhead is introduced since not all control block information is used for dynamic loading or task migration,

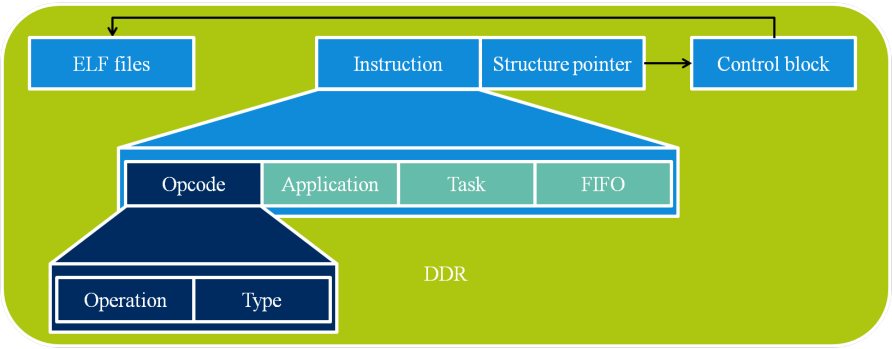


Figure 3.2: The loading information setup in memory.

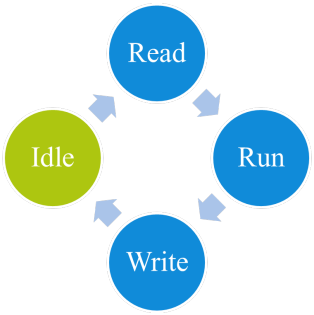


Figure 3.3: The states of a task and the migration point (the idle state).

while the entire control block is pulled by the ST. On the other hand, some extra information is added to the control blocks for dynamic loading such as ELF information for tasks that is not used by CompOSe.

3.1.4 Migration points

As with many existing migration mechanisms, we looked at the use of migration points [9, 10, 11, 17], because we want minimize the time it takes to migrate a task. Migration points speed up the migration process since we can define a state in which it is easier to save and restore the state of the task.

Since the applications in our system are based on dataflow graphs, we have well defined states (Figure 3.3) in which a task can be. Our model also assumes that no static data is used, because this data should be passed to the next iteration of a task by a self First In First Out (FIFO) channel. So when the task is in the idle state, it will have no data on the stack or heap, and the task does not make any changes to the FIFO channel data. For this reason, we choose the idle state as the migration point. This restricts the migration to only the point where a task has finished an iteration. The limitation of this choice is that if the task never finishes an iteration it cannot be suspended and thus not migrated.

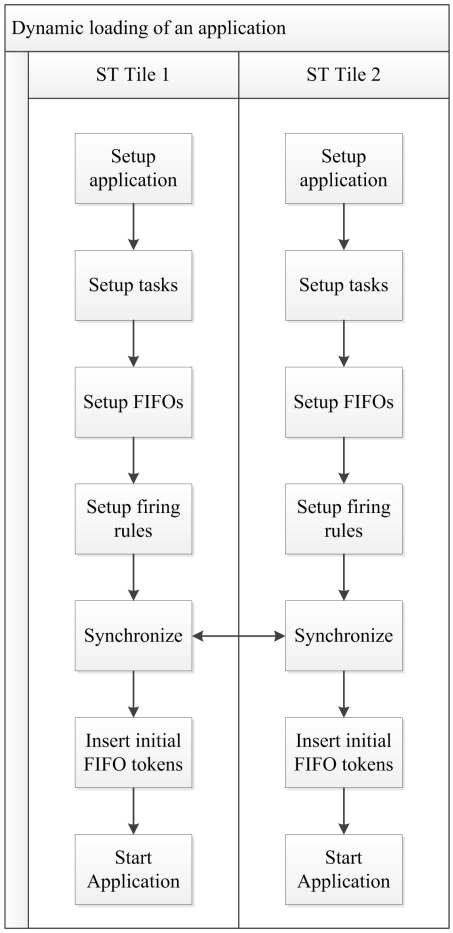


Figure 3.4: Dynamic loading protocol

3.2 Dynamic loading

To load an application, a CompOSe control block structure as in Figure 2.6 needs to be set up. To setup this structure, we follow the process shown in Figure 3.4. After CompOSe is started and the ST is scheduled as shown in Figure 3.1, the process of loading can start. For every part of the protocol as shown in Figure 3.4 an ST instruction is pulled from DDR. First the Application Control Block (ACB) is created, then for every task a Task Control Block (TCB) is created and the instructions and data are loaded in IMEM and DMEM. Next for every FIFO channel a FIFO Control Block (FCB) is created. For setting up firing rules, the instructions of the firing rule function are loaded into IMEM and the firing rule is added to the TCB the firing rule belongs to.

If the tasks of an application are mapped on several tiles, we need to synchronize before inserting tokens in the FIFO channels since a FIFO channel might not have been initialized. After the initial tokens are inserted in the FIFO channels, the application is started by adding it to the Time Division Multiplexing (TDM) table of the application scheduler. The different steps and the information needed to load an entire application

are described in more detail in Section 4.4.

Since we setup the control blocks the same way as in the existing system, we can use the implemented CompOSe API functions. For tasks we created new functionality to load the instructions and static data from the ELF files in DDR, and for firing rules only the instructions. As mentioned we use the CompOSe control blocks as descriptors for loading information. This introduces the need for some extensions to the TCB for loading. The extra information that is added to the TCB consists of the location of the instructions and static data in the ELF files.

Another extension to the TCB and ACB is a flag to let the task scheduler and application scheduler know if a task or application should be scheduled or not. This flag is checked by the application scheduler and enables us to start or suspend a task or application.

3.3 Task migration

Task migration is a complete new feature where more decisions needed to be made than for dynamic loading. We migrate a task from the source tile to a destination tile. In this section we explain in short the phases as shown in Figure 3.5.

First of all, we require the task to be in the idle state as mentioned in Section 3.1.4. We wait on all tiles for the task to be suspended by synchronizing between the STs. This is done because task mapped on the other tiles might produce FIFO channel data for the task that is required to finish its iteration.

Next, the ST suspends the entire application. Although it is only required to suspend all tasks connected through a FIFO channel to the task that is migrated, it would require a more complex set of ST instructions, because we need to check what tasks are connected and suspend these one by one. This also implies one ST instruction for every FIFO channel for the next step instead of one ST instruction per tile the application is running on.

For migration we also need to be sure that there is no data in-flight in the NoC, so we wait for the FIFO channels to in a quiescent state. This is done by a handshake protocol similar to the C-HEAP protocol used in the FIFO channels. We introduce an extra value in the FCB where after the last data transfer the write counter or read counter is send to. This means that we use only the existing connections on the NoC for this part of the protocol. Further details of this setup can be found in Section 4.5.2.

When the FIFO channels are quiescent, the ST recreates the task on the destination tile by dynamic loading the task. To make sure the FIFO channels are initialize, the STs synchronizes before it copies all the data from the FIFO channels to the destination tile. Next, the ST removes the task from the source tile. Because the task is in the idle state and thus has no private data, the FIFO data is all that needs to be copied. Copying this data from the source tile to the destination tile does require an extra connection on the NoC between the source and destination tile that is not used for the FIFO channel.

If the task structure has been removed on the source tile and setup on the destination tile, we make sure that all data is copied by synchronizing the STs again, so the application can be resumed. We only need to resume the application and not a specific

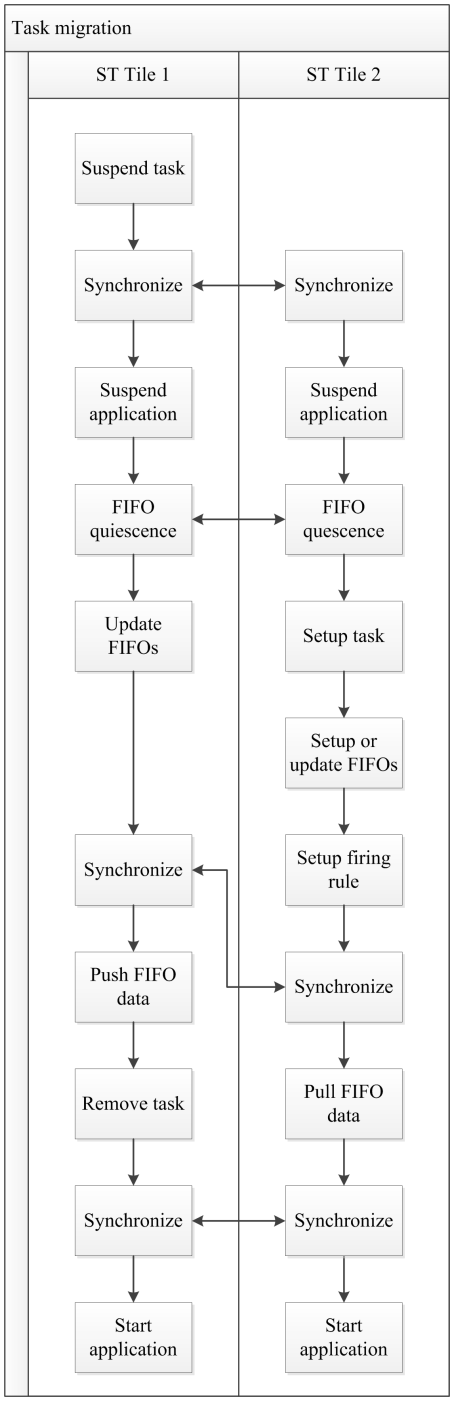


Figure 3.5: Migration of a task from tile 1 to tile 2

task, since the task is setup on the destination tile with the schedule flag enabled and on the source tile this task has been removed.

3.4 Summary

In this chapter we introduce the design for our dynamic loading and task migration protocol. Our protocols allow creation of separate application executable that can be reused on various instances of the CompSoC platform. Dynamic loading and task migration also enable use-case switching at runtime. This means we can reuse the executable of an application when changing the mapping of their tasks to processing tiles, or when changing the set of applications running concurrently.

To guarantee independent temporal and functional behavior of applications, we implemented the functionality in the SA. This minimizes the overhead in the execution time of the OS since we control dynamic loading and task migration in the application time slot, hence do not increase the execution time of the OS. The SA has one ST per tile that pulls an instruction from DDR. This ST instruction contains a opcode for the ST and the location of a descriptor that contains loading information.

Before the start of the system, CompOSe and the SA are compiled and linked in an executable for every processing tile. Another executable is compiled and linked for the monitor. The platform is initialized by the monitor. The monitor stores all the ST instructions, descriptors and the application executables in DDR. For every ST, the start address of the ST instructions is send to the processing tile that they need to be initialized and then all the processing tiles synchronize with the monitor. A processing tile starts with receiving the start address for the ST and initializes CompOSe and the SA and ST.

To facilitate this setup, we explain the use of PIC for the application executables in Section 3.1.1 so that we can store the contents of the executable in various addresses in IMEM and DMEM. Furthermore, we explain the setup of the ST instructions and descriptors that contain the loading information.

For task migration we introduce the use of migration points that reduces the complexity of task migration. We use the idle status of a task as migration point, which requires a task to reach this state before it can be migrated.

In the last two sections we described the steps for dynamic loading and migration. For dynamic loading we go through the stages of setup application, setup tasks, setup FIFO channels, setup firing rules, synchronize, insert initial FIFO tokens and start application.

The migration of a task from a source tile to a destination tile is slightly more complex. The ST first waits for the task to be idle, then synchronizes and stops the entire application. Next, the ST makes sure the FIFO channels are in a quiescent state. Then task is dynamically loaded on the destination tile, so the task is set up, FIFO channels are set up and the firing rule is added. After a new synchronization between the STs to wait for the FIFO channels to be initialized, the FIFO data is transferred, the task is removed and the STs synchronize again before resuming the application.

4 Implementation

In this section we present the implementation details of the dynamic loading and migration protocol. We first describe the additions in software and hardware to the existing CompSoC platform. Then we will explain the changes in local memory structure to facilitate the loading of the task computation function instructions and static data into memory at runtime. In the last two sections we will describe the protocols of dynamic loading and task migration.

4.1 Hardware additions

For dynamic loading, we need to store instructions and static data in the local memory of the processor tiles. We use the DDR to store the Executable and Linkable Format (ELF) files with the instructions. From DDR the instructions need to be copied to Instruction Memory (IMEM), and the static data is copied into Data Memory (DMEM).

We aim to follow as much as possible the existing hardware structure; hence we do not need to make a lot of additions the hardware. The only addition to the hardware are two Direct Memory Access (DMA) modules. One is attached to IMEM to be able to load instructions from a remote location. We need to load the instructions from a remote location since it is not possible to write to the IMEM from the MicroBlaze

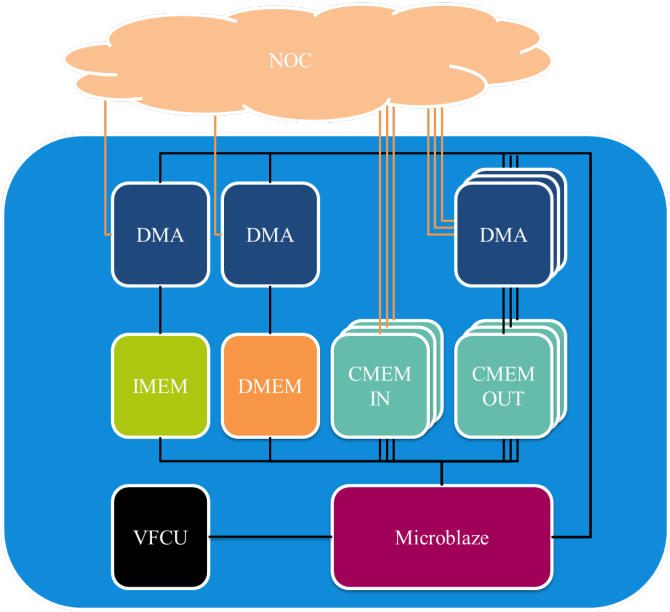


Figure 4.1: CompSOC tile.

processor directly.

The other DMA module is attached to DMEM to be able to load the data from a remote location directly. The DMA module connected to the DMEM is not necessarily required, since we are able to first pull the data into Communication Memory (CMEM) and then move the data into DMEM with the processor. But this would require a large buffer in CMEM and time to copy the static data from CMEM to DMEM. Adding the DMA module for DMEM therefore improves performance as we do not need to pull all the data into CMEM and then copy it to DMEM, but can pull the data directly into DMEM.

A tile with the new DMA modules is shown in Figure 4.1.

4.2 Initialization of processor tiles

Dynamic loading enables the separate creation of executables. First we will explain the executable that is used to boot the tile. This executable contains instructions and static data of CompOSe and the System Application (SA). Next we will discuss the creation of the executables for applications.

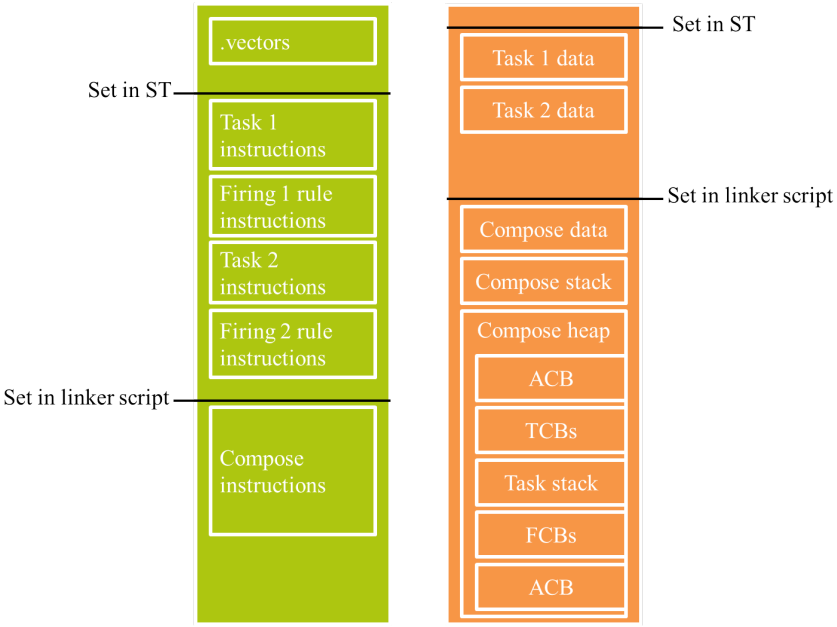
To start executing the computation function of tasks of applications on a tile, we need to have an initial system that is loaded on the platform. This requires an executable per tile with a boot function that initializes the Operating System (OS) and the SA and adds a System task (ST) on the tile to be able to load other applications. The ST is the same for every tile, so to remove duplicates, we created one instance of the source code of the ST that is included in all the boot functions per tile.

While the ST is the same for all the tiles, the SA is dependent on the setup of the tile. The SA is dependent on the following:

- Every ST uses First In First Out (FIFO) channels to synchronize with other STs. The SA has a CMEM and DMA module allocated on each tile for this purpose. This CMEM and DMA module is also used to pull the instructions from the DDR so requires a connection to the DDR.
- CompOSe is initialized with a Time Division Multiplexing (TDM) table for the application scheduler. We can vary the amount of slots allocated for the SA per tile and thus the time that is reserved for dynamic loading and task migration.
- The SA currently has one ST per tile, that is scheduled with a Round Robin (RR) policy, but we are able to use different scheduling policies per tile.

The ST will pull per iteration an instruction from the DDR. This means we have at design time already decided what the SA will execute.

The instructions for the ST are pushed to the DDR by the host/monitor. The host/monitor will also have to put the executables of the applications in DDR. The specific requirements for these files can be found in Appendix A. The main extensions to these files are the use of Position Independent Code (PIC), and if one executable contains several tasks, these tasks should be divided into several independent sections.



(a) IMEM (b) DMEM

Figure 4.2: Local memory structure.

4.3 Local memory

For dynamic loading of a task, the instructions of a task should be loaded into IMEM and the data into DMEM. This means that there should be space in these memories and therefore the data layout in these memories is adjusted. Where previously the task instructions and static data were included in the same object file as CompOSe, we now define a address range in memory that is not used by CompOSe to be able to load the instructions and data there. Figure 4.2 shows the data range in IMEM and DMEM that is allocated for the task instructions and the static data. The linker script for the executable that contains CompOSe is changed to start at a higher address, such that other applications can use the memory until that address. In the ST we define the start address where the instructions of a task need to be loaded. This is not at the start of IMEM, because the first part of IMEM is used for interrupt tables.

This setup also requires the task code to be position independent. The first requirement for position independent code is no static references to data. The instructions of the task already used relative addressing, but the data was statically referenced. To solve this we use PIC support in the Microblaze gcc compiler, which introduces an extra level of indirection in addressing data. The compiler produces a table with addresses (the Global Offset Table (GOT)) that is referenced by the instructions. The compiler reserves a register where a pointer to the GOT is stored and uses an offset from the start of the table to find the address in the table. Figure 4.3 visualizes this indirect addressing.

The second requirement for loading instructions anywhere in IMEM is to have relative jumps to functions in the loaded task instructions and no calls to functions outside of the

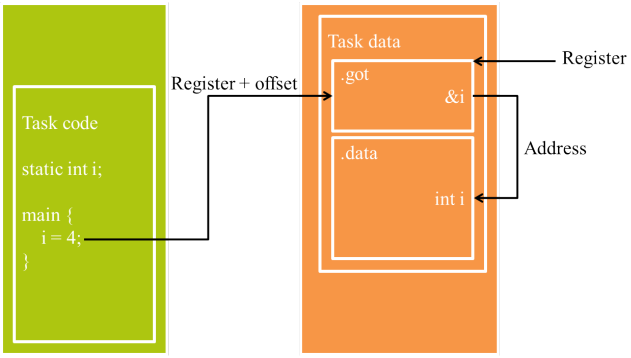


Figure 4.3: The indirect addressing of data.

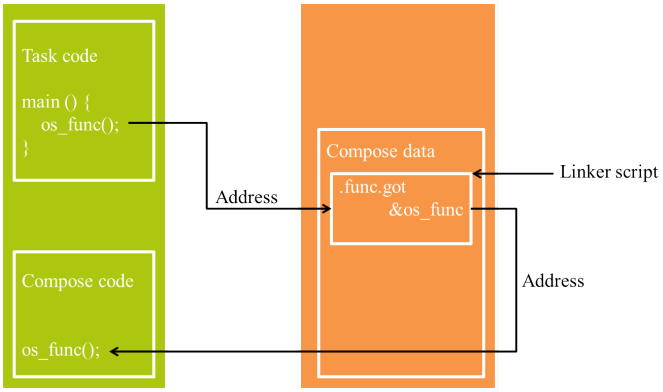


Figure 4.4: The indirect addressing of OS functionality.

task instructions. Because we want to call CompOSe functions, we created an extra level of indirection here, because there is no support for shared libraries and dynamic linking in the MicroBlaze compiler. This indirection is visualized in Figure 4.4. We create a function table that we compile with CompOSe. We define the place of the function table in the linker script and include this information with the source code of the applications that will be loaded. In the application source code we also include a file that redefines the CompOSe API function calls to an indirect function call through the table.

4.4 Dynamic loading protocol

The protocol for dynamically loading a task is visualized in Figure 3.4. As shown, the SA adds the entire application nearly completely in parallel per tile. Every ST will first setup the Application Control Block (ACB), then load the instructions and data into the local memory per task and create the Task Control Blocks (TCBs). If all the tasks are added, the FIFO channels are set up. The FIFO channel counters are initialized from the input FIFO, hence we need to synchronize the STs to make sure the FIFO channel is not initialized after we insert the FIFO token. After the initial tokens have been inserted in the FIFO channels, the application is started. The protocol will be explained step by

step in more detail in the following subsections.

4.4.1 Adding an application

Adding an application consists of calling the CompOSe Application Programming Interface (API) function to add an application. This function requires the following parameters:

- Application ID
- Number of tasks
- Number of FIFO channels
- The time slot the task scheduler is using (OS slot or application slot)

Because this information is already in the ACB, we use the ACB as descriptor to store loading information without making any changes to it.

4.4.2 Adding a task

When the ACB has been created, the tasks can be added. Adding a task requires next to creating a control block, the loading of instructions of the computation function in IMEM, and, if the instructions refer to static data, the loading the static data in DMEM. In this section we first describe the information that the ST needs to load a task. Then we describe every steps of the ST involved in adding a task.

To store the instructions of a task we use ELF files. These files include sections for instructions and data, and contain information about the start and length of the sections. How to create an ELF file and how to get the needed information can be found in Appendix A.

To load the instructions of a task the following information is needed from the ELF file:

- For executing the task function we need the instructions. These should be in a section in the ELF file, of which the ST needs to know where the section starts in DDR and the size of the section. With this information the ST pulls the instructions directly into IMEM with the attached DMA module.
- The start of the task function might not correspond with the start of the section that is loaded. Therefore we need the offset from the start of the section to the start function of the task. This way we can correctly point to the start of the task function from the TCB.

When static data needs to be loaded we also need information on the data sections. The following information is required to load the static data:

- The ST needs to know where the .got sections start in DDR and the size the .got sections. This range includes the .got.plt section which is used to mark the start of the GOT and the .got section that is used to store the GOT.

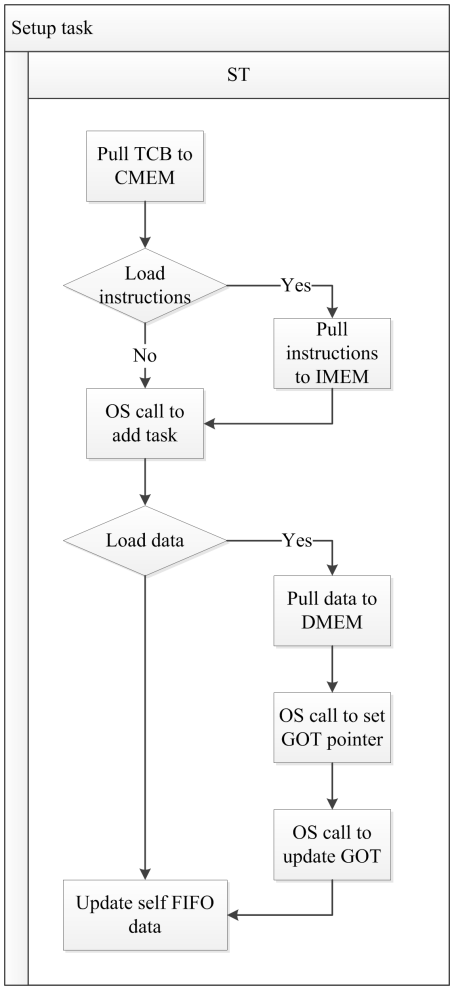


Figure 4.5: Loading a task

- The ST needs to know where the data sections start in DDR and the size of all of the static data sections. Note that the data sections might not contain any bits in the ELF file, but in CMEM these bits must be allocated. In our implementation the data sections are a copy of the ELF file, which requires the data sections that have no bits in the ELF to be at the end of the data sections. The size of the data sections should include the bits required for these sections.

- To update the GOT, the SA needs to know the address where the static data should be loaded according to the ELF file. We use a small example to explain how the GOT is updated. If the ST loads the data at address 200, while the first data section in the ELF is mapped to 100, the addresses of in the GOT will be 100 off. When we update the addresses in the GOT with the difference, we have the correct addresses.

- Because we need to update the GOT, we require the size of the GOT and offset of

the GOT from the start of the .got sections

- There is one special section, the .bss section that requires all data to be initialized to 0. To be able to initialize them with 0, the ST needs to know the size of the .bss section and the offset from the start of the data sections.

Since we use the CompOSe control blocks to store our loading information, the TCB structure needs to be expanded with the information described above.

Figure 4.5 shows the implemented steps for adding a task. When an instruction to add a task has been pulled from DDR, the ST pulls the descriptor into CMEM with all the required information to setup a TCB and load the instructions and static data.

If task instructions need to be loaded the ST uses the DMA module that is attached to IMEM to pull the instructions from DDR. It can then add the task control block with the CompOSe function for adding tasks. We also offer support to load the instructions for several task computation functions at once. This is done by loading the instructions in the first task. If for the following tasks the start of the section with task instructions and the size is the same, the task computation function offset value will be used to set the pointer to the task computation function in the previously loaded instructions.

When data needs to be pulled into DMEM, the attached DMA module will pull the data sections from DDR, set the GOT pointer and correct the content of the GOT. Because we have no memory allocation mechanism for both IMEM and DMEM, the ST keeps track of the free memory space with a pointer that is stored in the self FIFO channel of the ST.

4.4.3 Adding a FIFO channel

Loading a FIFO channel is quite straightforward as no information is needed compared to the setting up FIFO channels before the start of the OS. The ST hence only needs the following parameters of the CompOSe API function to add the FIFO channel:

- Producing and consuming task
- Producing and consuming port
- Pointers to local and remote buffer
- Pointers to the counters
- FIFO channel size
- Token size
- The used DMA module

For more information about the protocol used for the FIFO channels, see Appendix B.

4.4.4 Adding a firing rule

A firing rule is a function and thus its instructions need to be loaded in IMEM. This requires a pointer to the start of the firing rule function instructions in DDR and the size of the instructions, so we can pull the instructions with the IMEM DMA module. The

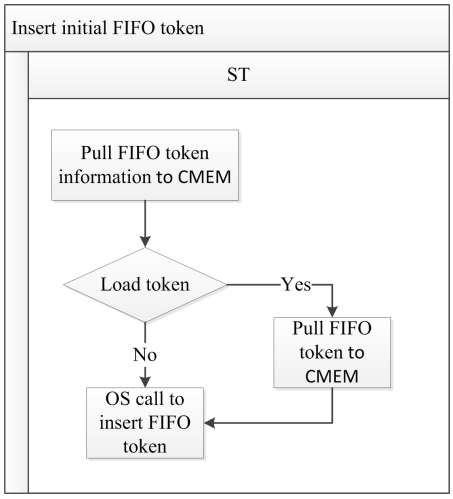


Figure 4.6: Inserting an initial FIFO token

same restrictions apply for creating the firing rule functions as for the task functions, however, these restrictions are easily fulfilled by typical firing rule functions.

Because the firing rule is part of an task, the start and size of the instructions of the firing rule function are added to the TCB.

4.4.5 Inserting FIFO channel tokens

Some FIFO channels require initial tokens. This token can be NULL, which means that only the counters need to be increased.

In the existing static loading initial FIFO tokens were created and referred in the same function, which left out the need for a control block with information on the initial token. This is why we introduce a new descriptor instead of using a control block of CompOSe. This descriptor contains the following information:

- Token rate
- Token size
- The address of the token in DDR

After the ST pulls an insert FIFO token instruction from DDR, it checks if the pointer to the descriptor is NULL to see if a token needs to be pulled from DDR into CMEM. If the descriptor exists, the ST pulls it from DDR, and then pulls the initial FIFO token from DDR into CMEM. Because the initial FIFO token is now in local memory, the CompOSe function for inserting an FIFO token can be called and the token is inserted in the FIFO.

4.5 Task migration protocol

The protocol for migrating a task is visualized in Figure 3.5. We will explain in the following sections how a task is migrated from a source tile to a destination tile.

4.5.1 Suspend entire application

To start the migration of a task we require it to be in a migration point, to make it easier to restore its state. The migration point is the end of an iteration (see Figure 3.3). In this point there is no data on the stack, so we only need to copy the FIFO channel data, because we require for migration that the task has all the static data in a self FIFO. The idle state is reached when the FIFO tokens have been produced and if necessary send to the remote data buffer.

If a task is never in the idle state when the SA is scheduled, we will not be able to start the migration. This can happen if the task is scheduled again directly after it has finished an iteration and is never in the idle state when the execution is preempted by the OS to schedule other applications such as the SA. Therefore we should make sure the task is not scheduled any more.

To make sure the task is in the idle state when the ST executes, we introduce a schedule flag. This flag is set by the ST and checked by the task scheduler of the application of the task that is migrated. If the task was not in the idle state already, the task will continue to run until it reached the idle state. After that it will not be scheduled anymore and in the first time slot where the SA is scheduled, the migration process can start.

A task that is migrated may communicate with other tasks through FIFO channels. To make sure that the state of its FIFO channels is not updated while the task is migrated, the protocol suspends the entire application. However, the protocol has to wait with suspending the entire application until the task is suspended, because it might be dependent on other tasks for the production of FIFO tokens. This is implemented by the synchronization operation.

We need to suspend the tasks that are connected through a FIFO channel because we need to migrate the FIFO channels of the task as well. If a task keeps consuming or producing on one of the FIFO channels that is used by the task that is migrated, we are not sure that the FIFO channel is up to date on the destination tile.

To unschedule an application we introduce a schedule flag in the ACB. This flag is checked by the application scheduler. If the flag is not set, the application is not scheduled anymore and thus we are sure that there will be no data produced over the FIFO channels.

4.5.2 FIFO channel quiescence

Although there is no data produced nor consumed any more by the tasks, FIFO channel data can still be in flight in the Network on Chip (NoC). This could mean that after migration FIFO tokens or counters can arrive at the source tile, and thus the migrated task will miss a token in the data buffer or can have incorrect counter values. To make sure we have no FIFO data in flight, there are several options that we will discuss in the following sections. After that we will propose our solution for this problem.

Wait for a period to pass.

Since the NoC is configured with guaranteed throughput with a defined maximum latency, we can calculate the period of time needed to make sure there is no data in flight.

Though this is an easy solution, it has its disadvantages:

- This is a platform dependent solution and therefore not portable.
- There is no feedback on the fact that the data transfer is finished, which makes it hard to debug if a calculation of the period is incorrect.

Send a finish token through the FIFO channel.

When a task has produced the last token on a FIFO channel, a unique token could be send that identifies the end of the data transfer. This solution also has its problems:

- Deadlock situations can occur when a task is consuming tokens over multiple FIFO channels.
- The token that defines the end of the data transfer cannot be send if there is no space in the buffer.
- If these unique tokens use the C-HEAP protocol, the read and write counters should be updated as well. These updated read and write counters need to be send over the network to make sure the unique token arrived, thus introducing more new data transfers to finish data transfers.

Set a finish flag

We can have a similar solution to the finish token over the FIFO channel, by sending a unique token to a reserved location. For this we require a pointer to the other end of the FIFO channel where a data field is in which we can send the unique token to, to let the other side know we have finished data transfers. When the ST also sends an unique token from other FIFO back to the FIFO that send the unique token, we know that there are no data transfers. This way we are not dependent on the data buffer. The disadvantages of this solution are:

- Extra memory would be required in CMEM to store this unique token and the FCB should be extended with a pointer to this value.
- We have to make sure the unique token does not arrive before the last data.

Proposed solution

The proposed solution is based upon the third option, where we introduce a Local Finish Counter (LFC) and a Remote Finish Counter (RFC) as shown in Figure 4.7. These values are initialized with a unique value that the Local Write Counter (LWC) and Local Read Counter (LRC) cannot reach. We distinguish the producer tile and the consumer tile. The producer tile is the tile on which the task that produces tokens for the FIFO channel is mapped. The consumer tile is the tile on which the task that consumes tokens for

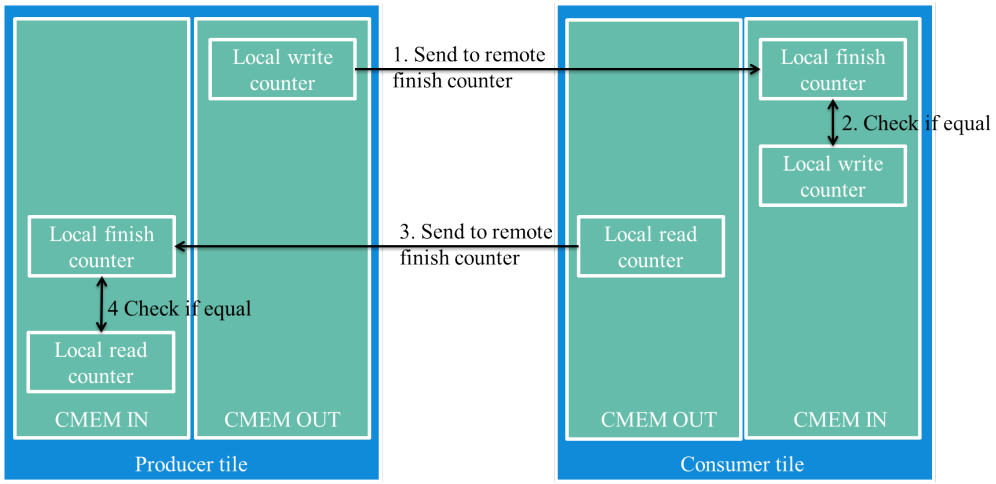


Figure 4.7: Handshake protocol for FIFO quiescence.

the FIFO channel is mapped. This strategy begins by sending the LWC at the producer tile to the RFC, which corresponds to LFC at the consumer tile (1). At the consumer tile, the ST compares the LFC and LWC (2). If these counters are equal, the ST on the consumer tile knows that the data transfers have finished.

To let the ST on the producer tile know that the data transfers have finished, it sends the LRC at the consumer tile to the LFC at the producer tile (3). The ST at the producer tile compares the LFC to the LRC (4) and if the counters are equal, it also knows that the FIFO channel does not have any data in flight on the NoC.

This strategy combined with the use of the idle state as migration point, guarantees that there is no data in the output FIFO buffer. Because the task is in the idle state, we have been through the write state and thus the DMA module has been given the command to push the data and the write counters. Next, because we use the FIFO channel's DMA module to send the LFC and LRC over the same connection on the NoC, we cannot send these counters before the data and we will always be sure that the data is in the data buffer of the input FIFO.

4.5.3 Dynamically load the task at the destination tile

To setup a task on the destination tile, the task is dynamically loaded in the same way are described in Section 4.4. In case a control block for the application or FIFO channels already exists it can be updated instead of created.

4.5.4 Copy FIFO data

We have the control blocks for the task and FIFO channels set up, but to resume the task, we also need the FIFO data buffers and the counters of the FIFOs to be copied from the source tile to the destination tile. We have two shared memories where the counters and data can be. The incoming data is in the CMEM In and is directly connected to the

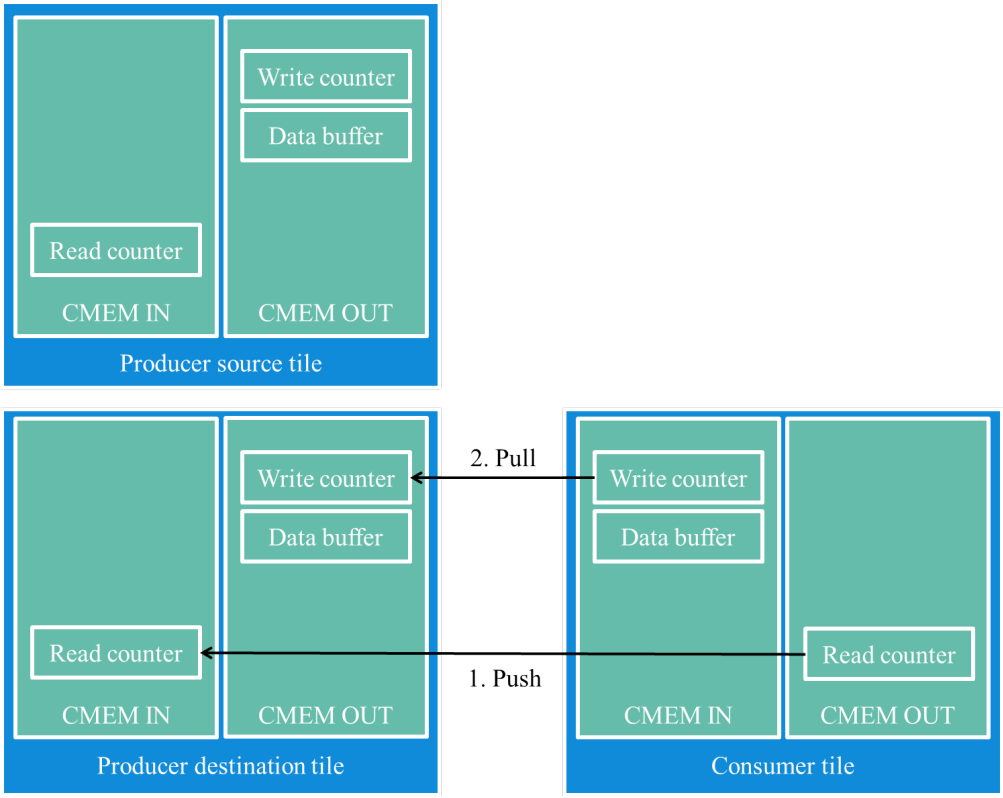


Figure 4.8: The data transfers for migration of a output FIFO.

NoC. For outgoing data we have a CMEM Out, which is connected to the NoC with a DMA module.

What exactly needs to be copied depends if we are migrating the producer task or the consumer task of the FIFO channel. In the first case, we only need to consider the counters because the small data buffer only holds temporary data that is pushed to the data buffer at the consumer tile. We know that the data arrived at the consumer tile, because the data is transferred directly to the input FIFO and we made sure the FIFO channel is in a quiescent state.

For copying the counter values, we make use of the fact that counters are correct at the consumer's side, from where they should be copied as shown in Figure 4.8. The LRC is up-to-date in the CMEM Out at the consumer tile, thus we send a instruction to the ST to push the counter to the LRC in the CMEM In of the destination tile (1). This all goes well if all the pointers to the FIFO channel have been updated to use the addresses of the destination tile. The LWC needs to be received from the CMEM In at the producer tile to the CMEM Out at the consumer tile (2). To do this an instruction is send to the ST at the destination tile.

In case of the migration the consumer task of a FIFO channel, we need to copy the data buffer as well. Figure 4.9 shows this process. The ST first pulls the read counter at the producer tile (1). Then the data buffer is handled. The correct data buffer is only available at the consumer task of the FIFO channel at the source tile, and should

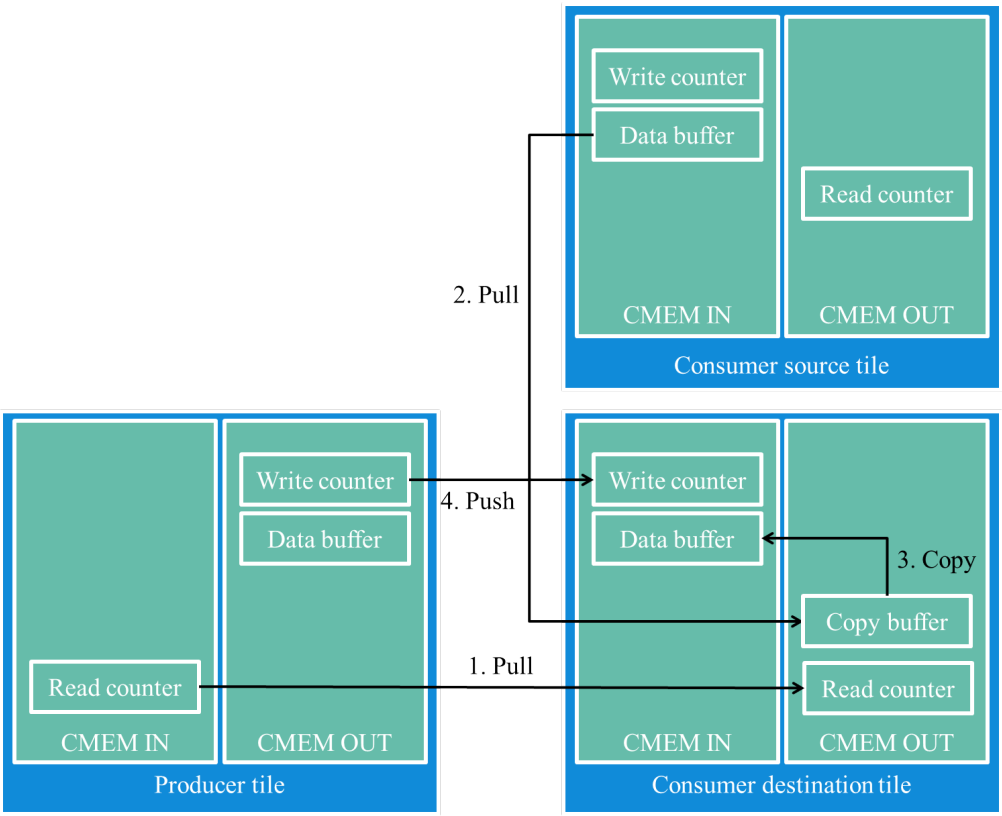


Figure 4.9: The data transfers for migration of a input FIFO.

be copied from there.

Because we cannot transfer the buffer from CMEM In to CMEM In, the data buffer needs to be copied. That is why there should be free space in the CMEM Out to temporary hold the FIFO data buffer. The ST pulls the data into this buffer (2) so it can be copied by the processor from CMEM In to CMEM Out (3). The up-to-date value of the LWC is in CMEM Out at the producer tile, so this LWC is pushed by the ST at the producer tile to the LWC at the consumer tile (4).

To be able to copy the FIFO data, the ST needs some information. To store this information we introduce the a new descriptor. The elements in this structure are:

- The address of the data buffer at the source tile.
- The address of the CMEM Out buffer.
- The id of the DMA module that is used to pull the data to the consumer tile.

For this transfer to succeed, we require a network connection. Because all the counters are send over the FIFO channels, we assume the connections are available. The only extra connection that needs to be taken into account for migration is a connection between the source tile and the destination tile to be able to migrate an input FIFO.

4.5.5 Remove task from source tile

After all state information is copied, it is not necessary to keep the IMEM, DMEM and CMEM allocated for the migrated task in the source tile. It can be that only a task needs to be removed, but when this is the only task left in an application, the resources the application is using could be freed for other applications as well.

The removal of a structure works top-down. So when a task is removed, the used FIFO channels will be removed as well. When an entire application is removed, the tasks and FIFOs of the application are removed as well. This means only one ST instruction is needed when removing an application.

4.5.6 Resume application

For resuming an application the schedule application flag, which we introduced in Section 4.5.1, needs to be set again by the ST. This way the application scheduler will schedule the application in the next slot where the application can use the time slot for execution. When the application was not yet in the TDM table, the application scheduler should also be updated.

4.6 Summary

In this section we have presented the details of how the STs of the SA provides dynamic loading and task migration on the CompSoC platform.

We first described the two added DMA modules needed to load task instructions and static data into IMEM and DMEM. Next, we described how the system is initialized with all the ST instructions, descriptors with loading information and ELF files in DDR, and on the processing tiles CompOSe and the SA. Then the used memory layout that for dynamic loading is described. We allocate at the start of IMEM a range of memory for loading the instructions of a task, and we allocate a range of memory at the start of DMEM for the static data of the tasks. We also explain the indirect function calls to CompOSe API functions and PIC, which enables us to place the instructions and static data at various physical addresses.

In the last part of this chapter the steps of the dynamic loading and task migration protocol are described. For dynamic loading, the control structure is setup by adding an ACB first. Then the TCB is setup per task and the instructions and static data is loaded. Next, the FIFO channels are setup by adding FIFO Control Blocks (FCBs). After that the firing rule function instructions are loaded into IMEM and this function is added to the TCB. To make sure all the data in the control blocks is initialized, the STs that are loading the application are synchronized before proceeding with inserting the initial tokens in the FIFO channels. The last step is to start the application.

For the migration of a task from the source tile to the destination tile, the ST first waits for the task to be in the idle state. To make sure the task is not scheduled again, a flag is added to the TCB. Then the entire application is suspended to make sure the tasks of the application do not produce any data on the FIFO channels. This is done by adding a flag to the ACB and checking for this flag in the application scheduler.

Next, we make sure no FIFO data is in flight on the NoC. This is done by resending the counters through the FIFO channel to a address that was initialized with a value that the counters do not reach. This is done from the producer tile to the consumer tile and back to have a proper handshake protocol.

After the task is loaded by dynamic loading it on the destination tile, the counters and FIFO data are copied. This is done by pushing and pulling the remote counter values that are up-to-date. For copying the FIFO data we require an extra connection from the destination tile to the source tile so the ST can pull this data and copy it to the correct location. When this is done the task on the source tile is removed and the application is resumed.

5 Experiments and results

In alignment with the goals set in Section 1.1, we investigate how our protocols perform at what cost and if we can provide independent execution time. We do that by investigating the functional and temporal behavior of our dynamic loading and task migration protocol, the functional behavior and temporal behavior of applications, and the resources needed to facilitate dynamic loading and task migration.

To investigate performance of the System Application (SA) we use the following metrics:

- The **loading time** is the time it takes to load an application. This includes the time to load all the tasks of an application on the tiles they are mapped to and to create and initialize the required First In First Out (FIFO) channels between these tasks.
- The **reaction time** is the time that is needed for a task to reach the migration point so it can be migrated. This means from the moment the System task (ST) pulls an instruction to suspend a task until the entire application can be suspended.
- The **application stall time** is the time after the reaction time that is required to migrate a task. This is from hte moment we suspend the entire application until the application is resumed.

For this investigation, we created an instance of CompSoC with two processing tiles and a memory tile (DDR) that is implemented on a Xilinx ML605 board. For this platform we define several use-cases described in Section 5.1. In Section 5.2 we explain what we investigate and how our implementation of the dynamic loading and task migration protocols perform.

5.1 Experimental setup

In this section we explain the use-cases utilized throughout our experiments and the details of the platform.

In the experiments we use three applications and the SA. The first application is a H264 decoder. Next, we use a JPEG decoder and last application is a Image Rotation application. A detailed description of the three applications and their tasks are in Appendix C.

With these four applications we create several use-cases. The first use-case is the SA with the H264 decoder, shown in Figure 5.1. These applications are preloaded, so they are be combined in one executable with CompOSe that is loaded on the platform. We do this first of all because we need the SA before the start of CompOSe to load the other

Table 5.1: TDM slot allocation for the four use-cases

TDM table Slots	1	2	3	4	5	6	7	8
Use-case 1	SA	SA	SA	SA	H264	SA	H264	SA
Use-case 2	SA	JPEG	SA	JPEG	H264	JPEG	H264	JPEG
Use-case 3	SA	JPEG	SA	JPEG	H264	JPEG	H264	JPEG
Use-case 4	SA	ImgRot	SA	ImgRot	H264	ImgRot	H264	ImgRot

applications. Second, preloading the H264 decoder allows us to show that the dynamic loading of applications does not affect the temporal behavior the H264 decoder.

The second use-case is the SA, H264 decoder and the JPEG decoder, mapped with only the Variable Length Decoding (VLD) task on tile 1, and with the Inverse Discrete Cosine Transform (IDCT) task and Color Conversion (CC) task on tile 2 as shown in Figure 5.2. Switching from the first use-case to this use-case is done by dynamically loading the JPEG decoder.

The third use-case contains the same set of applications as the second use-case, but with a different mapping of JPEG tasks. The IDCT task is mapped in this use-case on tile 1 as shown in Figure 5.3.

In our last use-case the set of running application is the SA, the H264 decoder and the Image Rotation application as shown in Figure 5.4.

We switch between the use-cases in the order they have been described above. The first use-case switch allows us to show that our dynamic loading protocol does not interfere with the h264 decoder. We also check the temporal and functional behavior of the JPEG decoder to see if this is influenced by the loading.

In our general setup we show that at runtime we can now load applications, increasing runtime flexibility opposed to the existing system. We go even further by introducing task migration. For switching between the second and third use-case we use the task migration protocol. We check that the JPEG application pauses and resumes correctly and that the migration process does not influence the temporal and functional behavior of the H264 decoder.

We investigate the IDCT task for several reasons. First of all, the IDCT is a producer and a consumer of FIFO tokens. This requires the most extensive reconfiguration of the FIFO Control Blocks (FCBs), since a local FIFO channel needs to be converted to a remote FIFO channel, a remote FIFO channel is converted to a local FIFO channel, requiring both the update and creation of new FCBs. Next, FIFO channel tokens need to be copied, so we test all the functions for copying FIFO channel data. The other reason for the choice of migrating the IDCT task is that the IDCT task has no static data. This is a requirement for migration, since we do not offer support for migration of data in Data Memory (DMEM), where the static data is located.

With the last use-case switch, we show that dynamic loading reduces the amount of resource required by the applications, since we can free up memory by removing the JPEG application and use the same resources for the Image Rotation application.

The Time Division Multiplexing (TDM) table slot allocation of the application scheduler for these four use-cases are shown in Table 5.1.

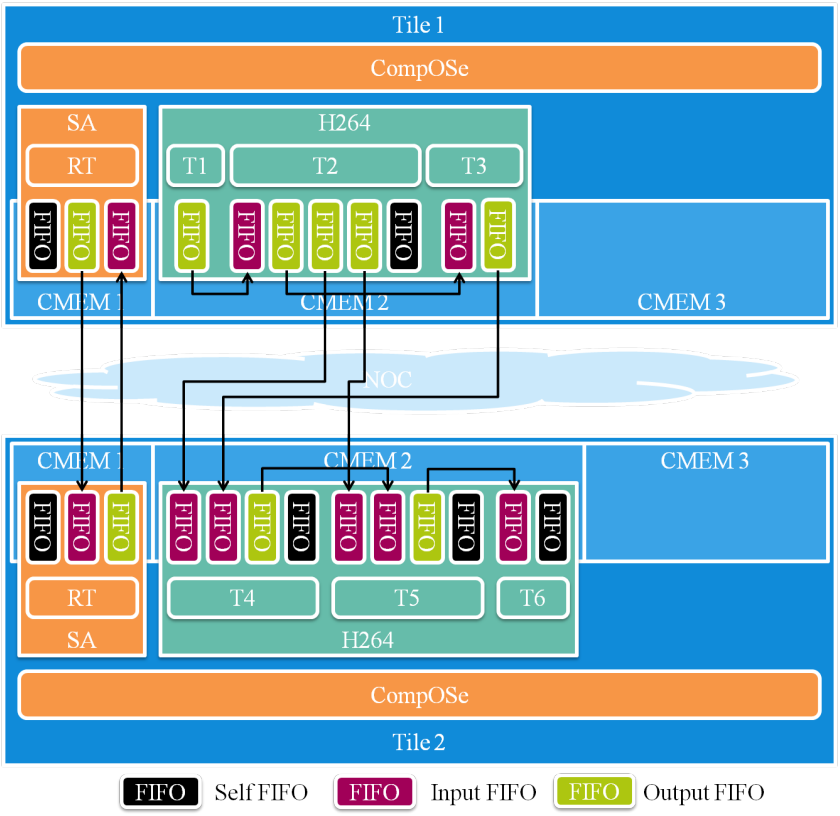


Figure 5.1: The experimental setup before the start CompOSe.

In short, these are the exact steps that are taken in these experiments:

1. Load the setup information and Executable and Linkable Format (ELF) file of the JPEG decoder and Image Rotation application in DDR.
2. Synchronize the tiles with the monitor.
3. Initialize CompOSe.
4. Preload the system application.
5. Preload the H264 decoder.
6. Start CompOSe. The systems is setup as in Figure 5.1.
7. Dynamically load and start the JPEG decoder. The systems now as in Figure 5.2.
8. Suspend the JPEG decoder.
9. Migrate the IDCT task of the JPEG decoder.
10. Resume the JPEG decoder. The systems now as in Figure 5.3.
11. Stop and remove the JPEG decoder.
12. Dynamically load and start the image rotation application. The system is now as in Figure 5.4.

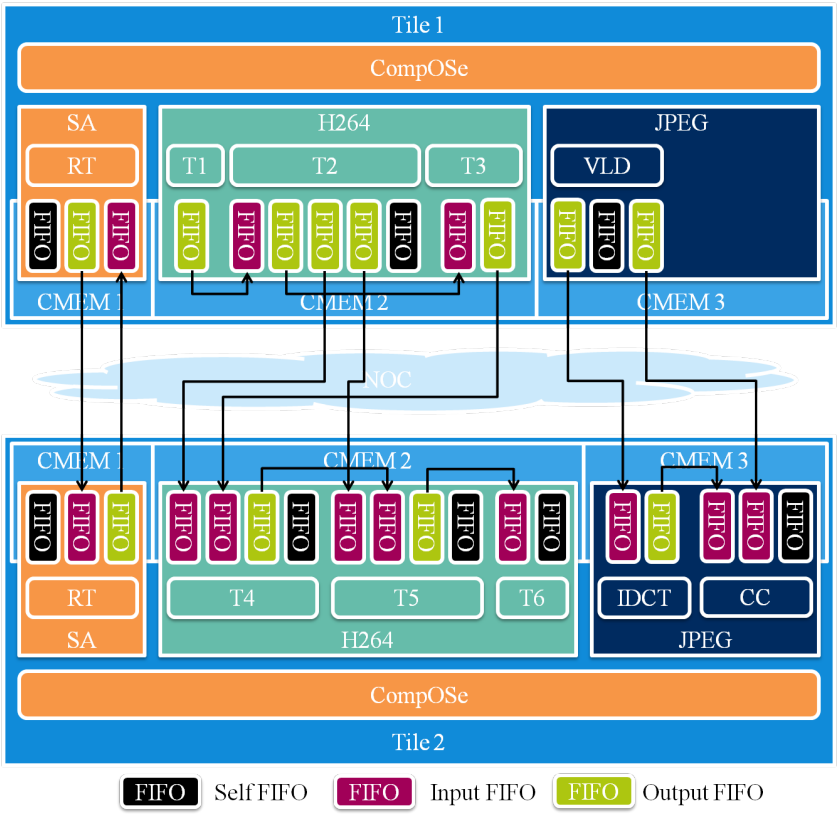


Figure 5.2: The experimental setup after loading the JPEG application.

5.2 Experiments

In the following sections we describe how we use the general setup and what is measured to show that we satisfied the goals.

5.2.1 Functional behavior

The dynamic loading and task migration should not affect the functional behavior of the application that is loaded, nor should it interfere with other applications. In the last task of our applications, the calculated output is checked against an output reference. This task will return debug information if a value does not match the output reference. If the last task does not return incorrect checks, we say that the application is functionally correct.

For all applications, in all of the use-cases, the last task returns no error information, hence we conclude that the applications execute correctly. The fact that the H264 produces no error information shows that loading and migration does not influence the functional behavior of the execution of a running application. The fact that the JPEG decoder and Image Rotation do not produce error information shows that dynamic loading an application does not influences the functional behavior. The correct functional behavior of the JPEG decoder also shows that task migration does not influence the

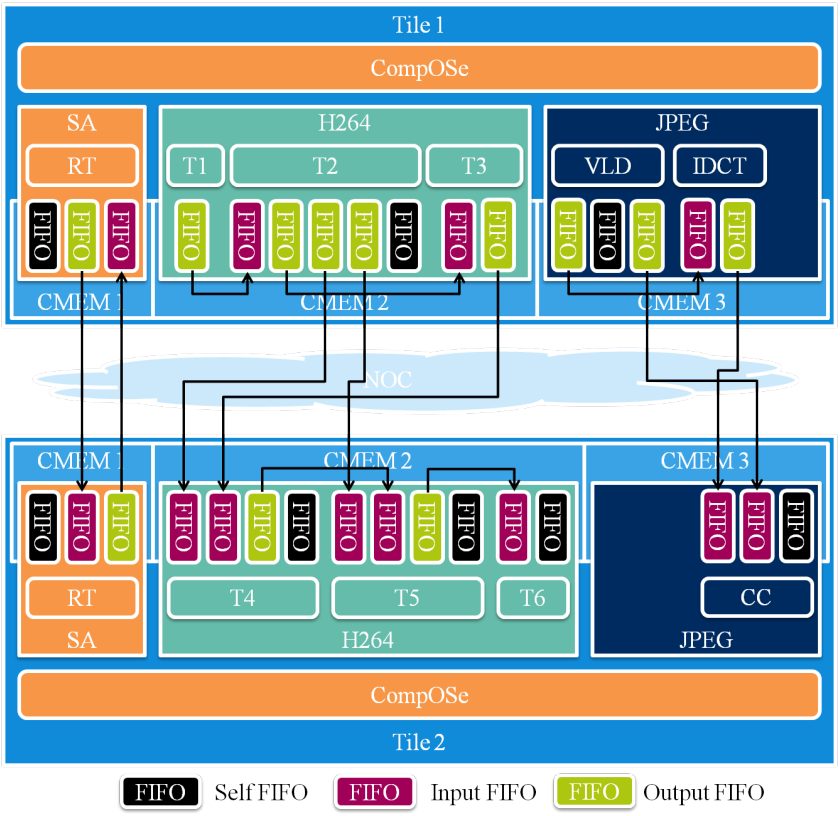


Figure 5.3: The experimental setup after migrating the IDCT task of the JPEG application.

functional behavior of another application that runs on the platform.

5.2.2 Reuse of resources

One of the reasons for dynamic loading is to be able to use the scarce resources of the system more efficient. In our general setup we have use-cases where only two applications and the SA need to run at the same time on our system. Since applications are not bound to static addresses, in our experiment the resources in use by the JPEG application can be freed after it is stopped and reused for the Image Rotation application.

Without dynamic loading these applications needed to be compiled in one executable and mapped on separate resources, hence all applications need to allocate DMEM for the static data before the start of CompOSe. Next, a pair of CMEM In and Communication Memory (CMEM) Out is reserved for every application. Both DMEM and CMEM can be reused with the introduction of dynamic loading. We investigate how much memory can be reused by removing the JPEG decoder and the worst-case memory footprint of our use-cases compared to the memory allocated when a single executable is used. We distinguish in this investigation the allocation of Instruction Memory (IMEM), DMEM and CMEM.

In case of dynamic loading applications, we need the memory space for the use-case

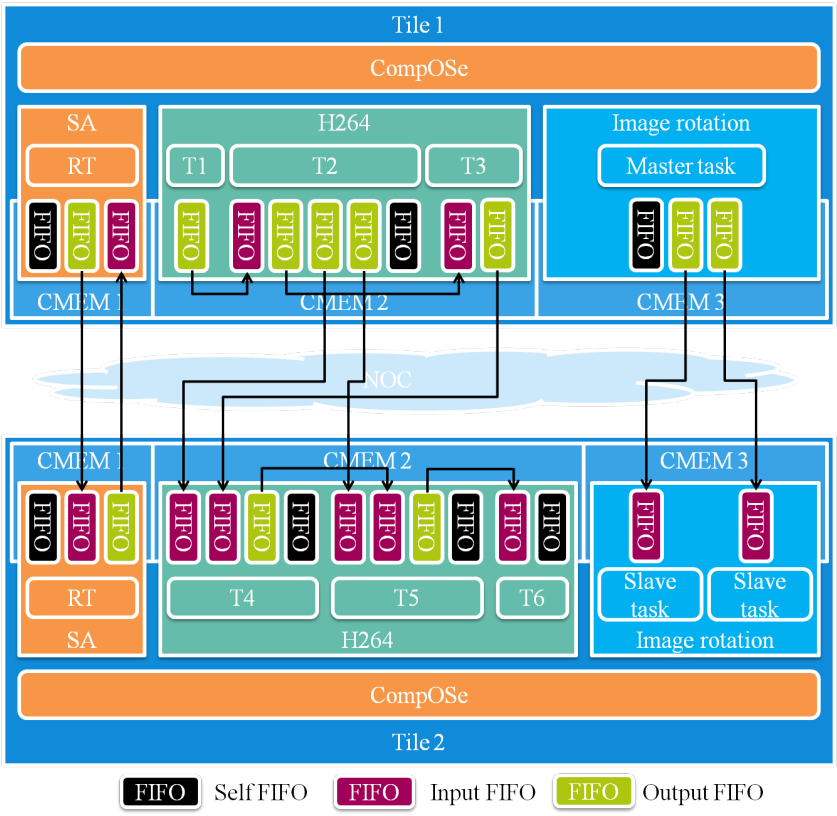


Figure 5.4: The experimental setup after loading the Image Rotation application.

that uses the largest amount of memory. We show the worst-case for IMEM usage on tile 1 in Figure 5.5). In IMEM the base CompOSe instructions are stored with the instructions for the SA and the H264 application. The JPEG application requires space for the instructions and firing rules, a total of 10580 bytes. The image rotation requires 632 bytes. The worst-case is thus 10580, where we would have required 11212 bytes when creating a single executable. In this case the IMEM savings are only 5%, however when more use-cases are introduced, there is potential for a larger impact.

Our worst-case scenario for DMEM is the same since compared to a single executable, because the Image Rotation application does not have any static data. Still, we are able to reuse all the allocated DMEM for the JPEG if other applications are loaded that do require static data. The amount of static data of the JPEG application on tile 1 is 18360 bytes, taking up 28% of the DMEM that is reserved for static data for the loaded applications. This again shows a potential for a larger impact when more use-cases are introduced.

In the existing system, the input images and output references are also compiled as static data and memory is allocated. Although input images might vary in size and the Image Rotation can use the output of the JPEG decoder, we consider the setup in our current experiment where these applications have their own input image. If the input image would be put in DMEM as static data as was done on the existing system, we

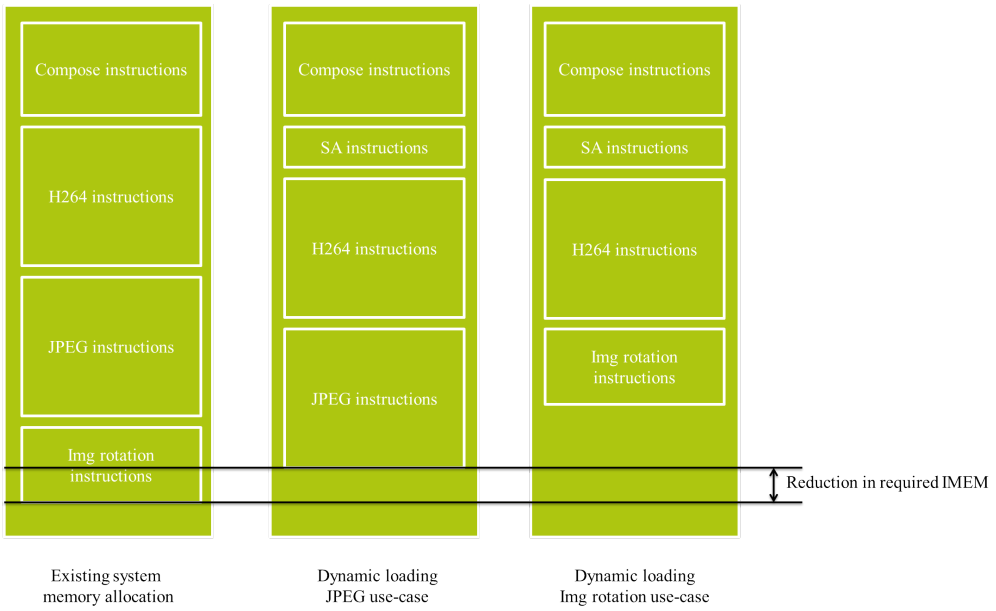


Figure 5.5: The reduction of required IMEM by dynamic loading

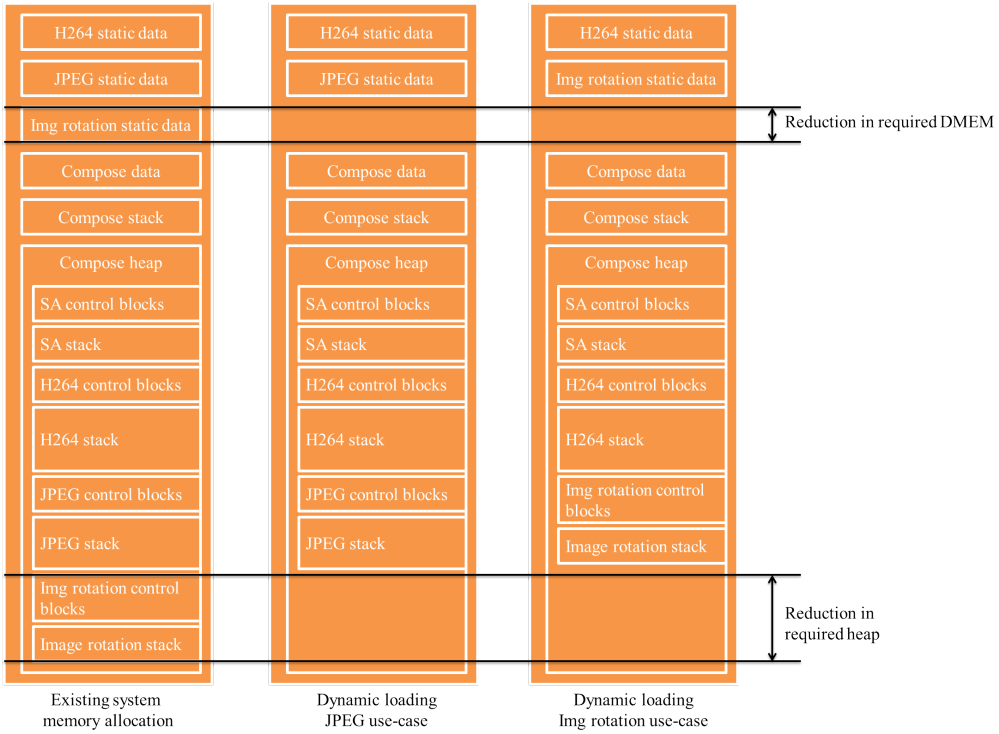


Figure 5.6: The reduction of required DMEM by dynamic loading.

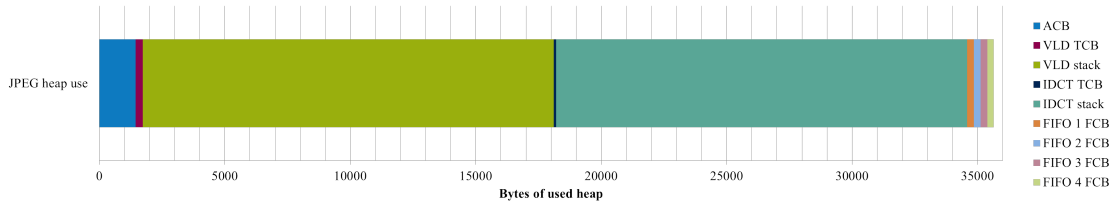


Figure 5.7: The use of heap by the JPEG application on tile 1

save 33% of memory that is reserved for static data of loaded applications on tile 1, since the size of the JPEG input image and static is 25068 bytes, and the size of the Image Rotation input image is 12765 bytes. Our worst-case scenario for a single executable would be 37833 bytes against the worst-case with dynamic loading of 25068 bytes.

We also consider the reuse of heap allocated for applications, since in the existing system all the heap is allocated before the start of CompOSe and not reused at runtime. Both control blocks and the stack of every task are stored on the heap. For the JPEG application with the VLD and IDCT task on tile 1, 35650 bytes are allocated on the heap to store all the control blocks, including stack (see Figure 5.7). In these memory allocation are also allocations included that are linked to the control blocks such as the rate tables for the FIFO channels. The total size of the heap is 151552 bytes, which means we are able to reuse 23.5% of the heap on tile 1 by removing the JPEG application.

The last memory that is allocated for the applications is CMEM. On the existing system, one CMEM is reserved per application. By dynamic loading applications we can reuse the CMEM when switching use-cases. The default amount allocated for CMEM is 8192 bytes for CMEM In and the same for CMEM Out. The actual used amount of CMEM is smaller. For the JPEG application on tile 1 only 1324 bytes are used. The reduction of required CMEM depends on the use-cases, since per CMEM the use-case that requires the largest amount of that CMEM sets the minimum required amount of CMEM.

5.2.3 Independent execution time

We investigate if the execution time of applications is independent of the set of applications running on the system in two cases. First we investigate if the process of dynamically loading an application does not affect the execution time of applications that are running on the system. Second, we investigate if the loaded application is influenced by applications that run on the system.

For the first part, we use the property of the CompSoC platform that ensures applications will always have the same execution time. We do this by setting up one application before the start of CompOSe and check its execution time against a system with the SA enabled and one without the SA.

In our general setup we preload the H264 application, so that it fits this experiment. We first run only the H264 application, so remove the SA (step 4) from our experiment. Removing the SA means that no loading or migration will be performed (all steps from step 7). In a second run, we add the SA, hence we run the complete experiment as

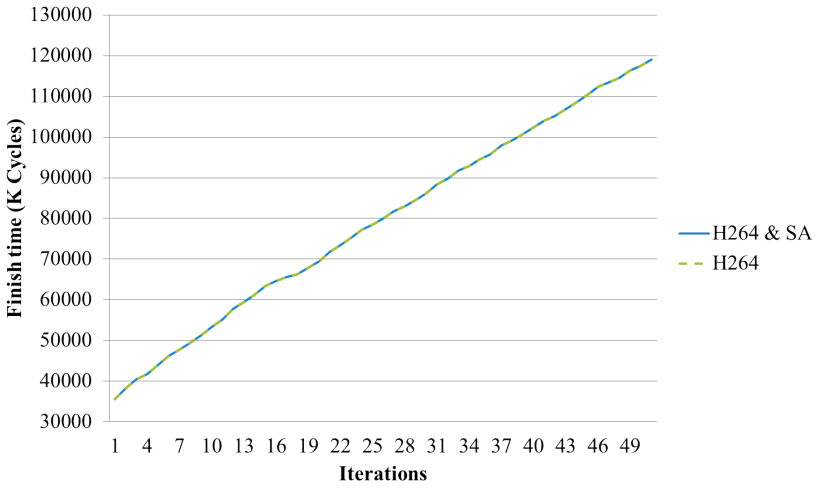


Figure 5.8: H264 - The IDCT task iteration finish time.

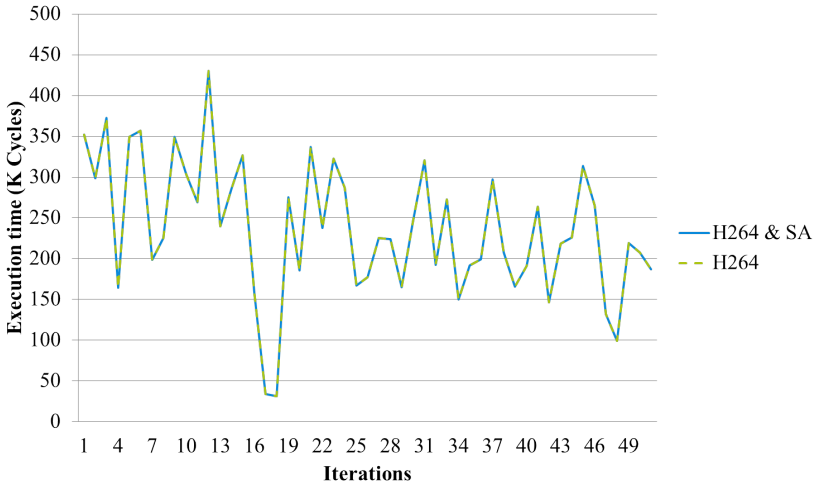


Figure 5.9: H264 - The IDCT task execution time.

described in the general setup.

In these runs we measure the finish time and the execution time of IDCT task of the H264 application and investigate if it is influenced by the SA. Figure 5.8 shows the finish time of the IDCT task iterations, and Figure 5.9 shows the execution time that is needed for the iteration of the IDCT task. Both show no difference between the H264 running alone and the H264 running together with the SA that performs dynamic loading and task migration.

For investigating if an application that is loaded is affected by another application, we schedule only the SA in the first run by removing step 5, and in a second run, we schedule it together with the H264 decoder. We measure the iteration finish time of the VLD task and its execution time per iteration to see if there are differences.

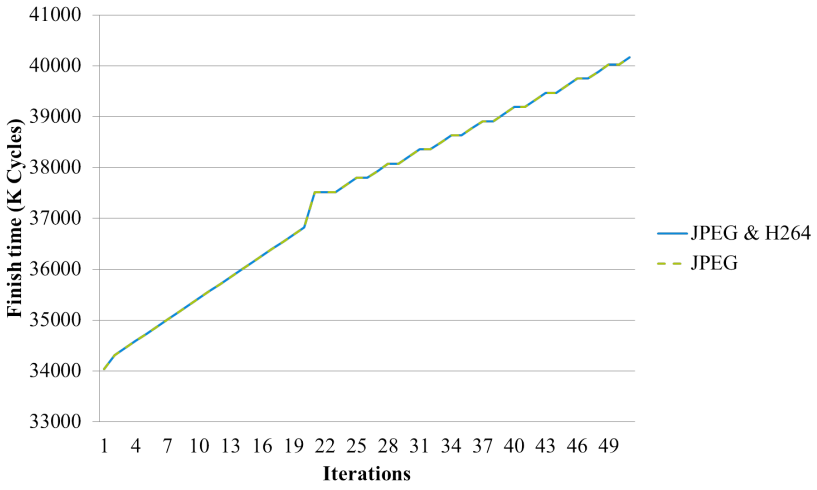


Figure 5.10: JPEG - The VLD task iteration finish time.

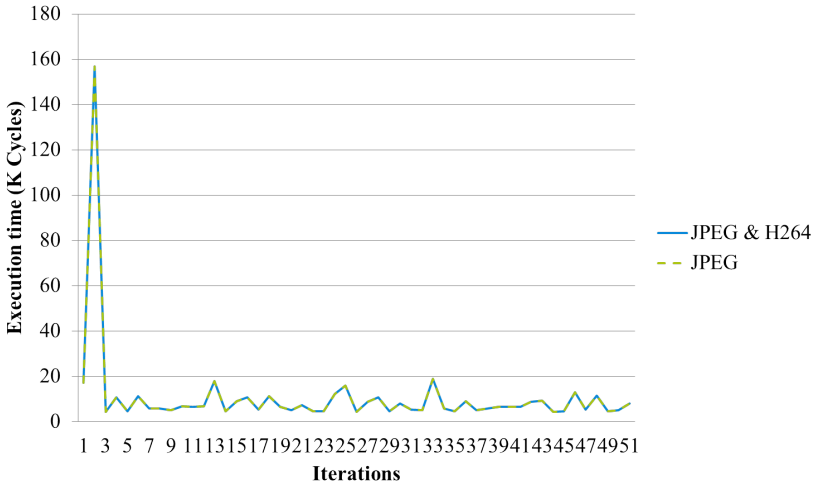


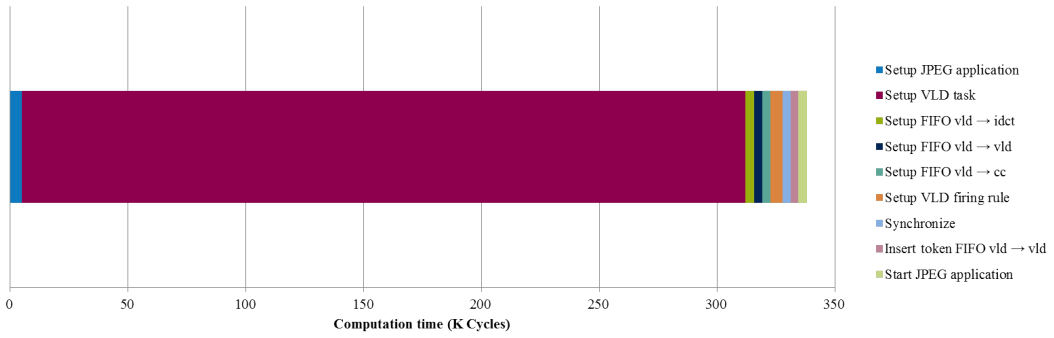
Figure 5.11: JPEG - The VLD task execution time.

Similar to the IDCT task of the H264 application, there are no differences in execution time as shown in Figure 5.10 and Figure 5.11.

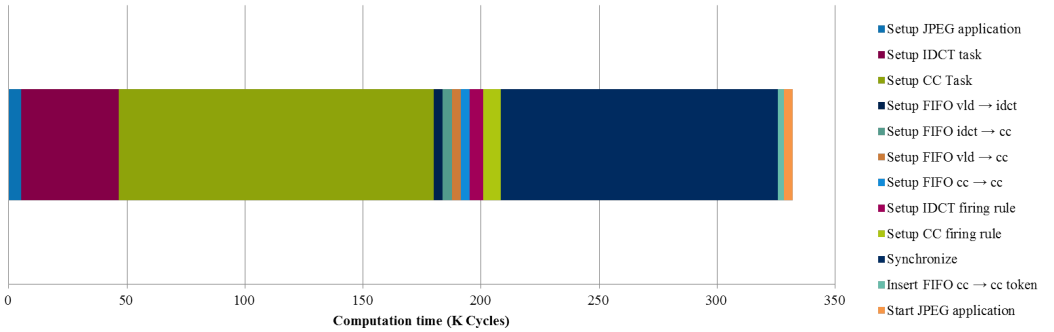
5.2.4 Loading time

The time it takes to load an application can be measured by looking at the execution time of the SA iterations. We measure the execution time for each ST instruction of the loading protocol.

The execution time of the SA iterations can vary much for every part of the protocol, because it is dependent on a different set of system parameters and application parameters. System parameters are for example the Network on Chip (NoC) configuration and the application TDM table configuration. To have a fair comparison, we will use



(a) Tile 1



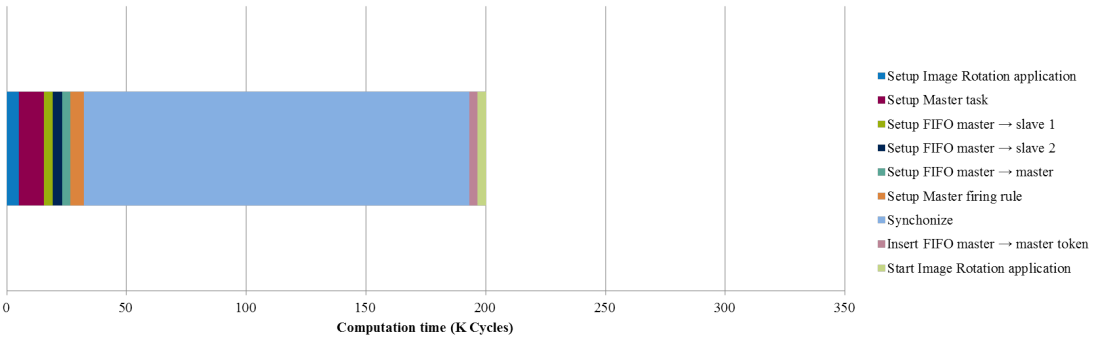
(b) Tile 2

Figure 5.12: JPEG - Load time

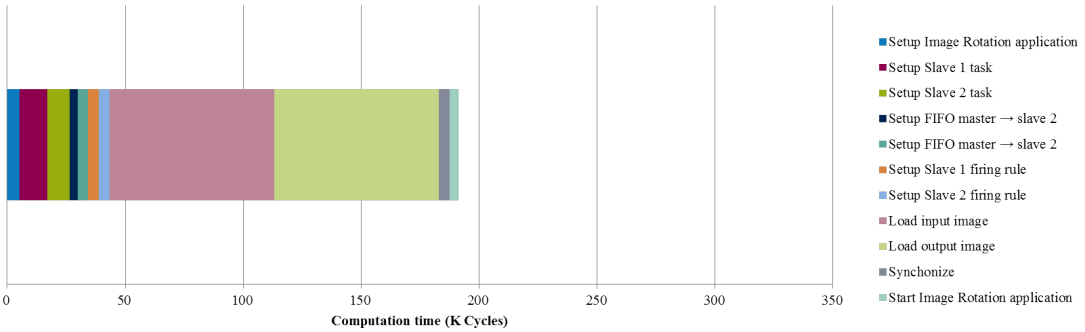
the default bandwidth for all the NoC connections in our design for all our experiments. Next, we use large time slots for the application scheduler to have no interruptions of the loading process. To make a fair comparison between the loading time of the JPEG application and the Image Rotation, we run the platform once with loading the JPEG application from the start of the SA, which is done in the general setup of the experiment and another run where the Image Rotation is loaded directly after the start of the SA. This is achieved by removing the steps 7 to 11 from the general setup.

Figure 5.12a to Figure 5.13b show that the load time varies a lot between the JPEG decoder and the Image Rotation application, of which the largest difference comes from loading task code. Loading the task instructions and static data of the VLD task even takes up 90% of the total time the ST needs for loading. Note that the actual start of an application is always at the start of the first time slot the loaded application is scheduled by the application scheduler, which in this case is the same time for both the JPEG decoder and the Image rotation application due to the large slot size.

With the knowledge that the loading time is dependent most of all on the setup task instruction, we compare the execution time of the SA for this instruction to investigate in how far this is dependent on the size of the instructions and static data. Figure 5.14 shows the different time the SA needs to setup a task, and load the instructions and static data from DDR into IMEM and DMEM. By comparing this to the bytes that



(a) Tile 1



(b) Tile 2

Figure 5.13: Image Rotation - Load time

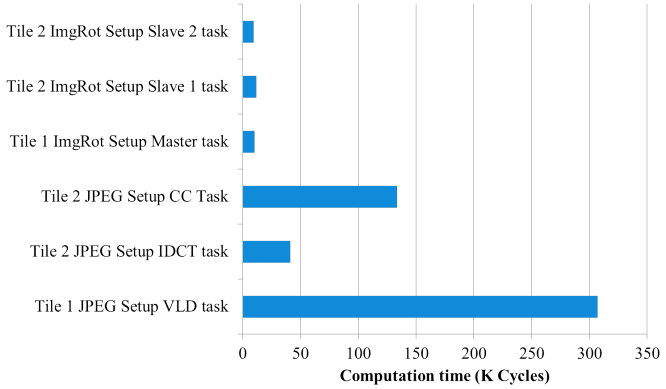


Figure 5.14: Execution times for loading the tasks

need to be loaded as shown in Figure 5.15, we see that the loading time of a task is similar to the instruction and static data sections size.

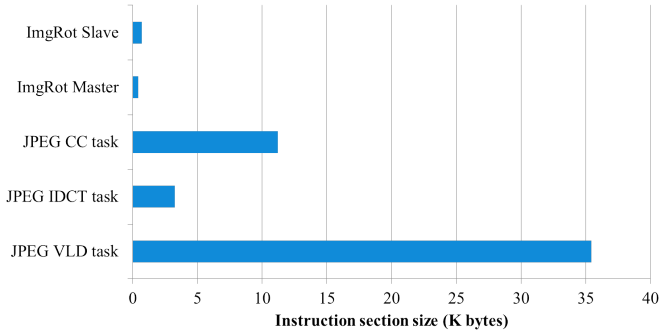


Figure 5.15: The size of the instruction section per task

Table 5.2: TDM slot allocation use for Figure 5.16b

TDM table Slots	1	2	3	4	5	6	7	8
1 slot	SA	JPEG	JPEG	JPEG	H264	JPEG	H264	JPEG
2 slots	SA	SA	JPEG	JPEG	H264	JPEG	H264	JPEG
3 slots	SA	SA	SA	JPEG	H264	JPEG	H264	JPEG
4 slots	SA	SA	SA	SA	H264	JPEG	H264	JPEG

5.2.5 Reaction time

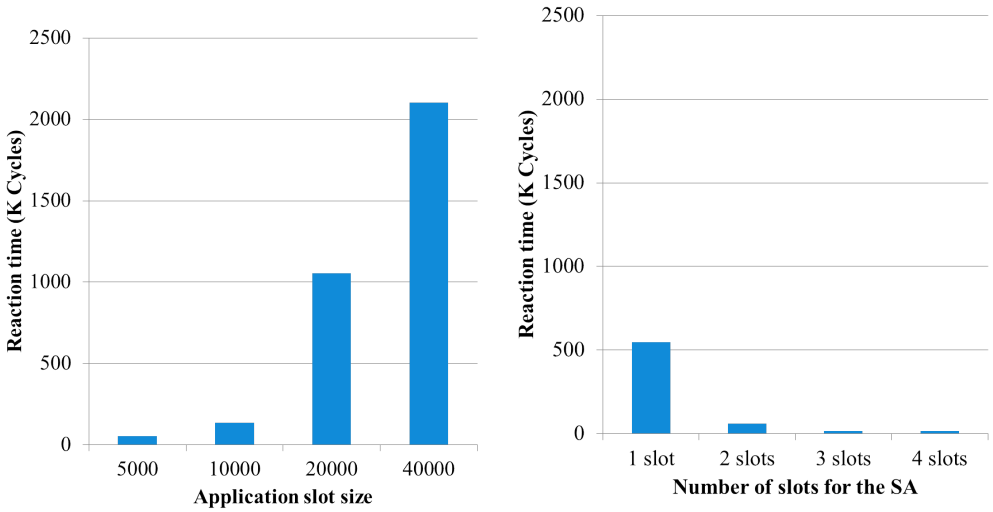
The SA runs in application time and can only execute in the time slots that were assigned to it. A delay is introduced with the use of migration points, because when we want to start migration, we need to wait for the task that is migrated to be in the idle state, which can take some time.

We measure the reaction time, which is the time from the start of the iteration of the SA where the suspend task instruction is processed until the synchronization is done, meaning that the task has reached the migration point. After synchronization the suspension of the entire application can start which we take as the start time of the migration process. We investigate how the reaction time is influenced by allocating a different amount of time slots in the application scheduler to the SA and by changing the application slot size. When we change the application slot size, we also increase the time of the Operating System (OS) slot to keep the percentage of time the SA is executing the same.

Figure 5.16a shows that the reaction time increases if the slot size is larger. In Figure 5.16b we use the default slot size (10000), and show that when more slots are allocated in the application scheduler TDM table, the reaction time will be decreased. The variation in slots can be found in Table 5.2. For the variation in slot size, the TDM table from Table 5.1 have been used.

5.2.6 Application stall time

Our migration protocol completely suspends the application when one of its tasks is migrated. Because our applications are real-time applications, we want to minimize the



(a) Variation in application slot sizes

(b) Variation in application slots allocated to the SA

Figure 5.16: Reaction time for the migration of the IDCT task

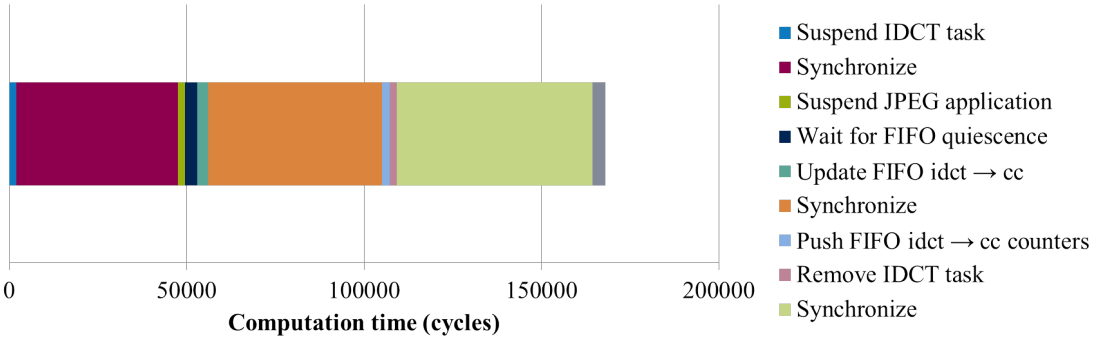
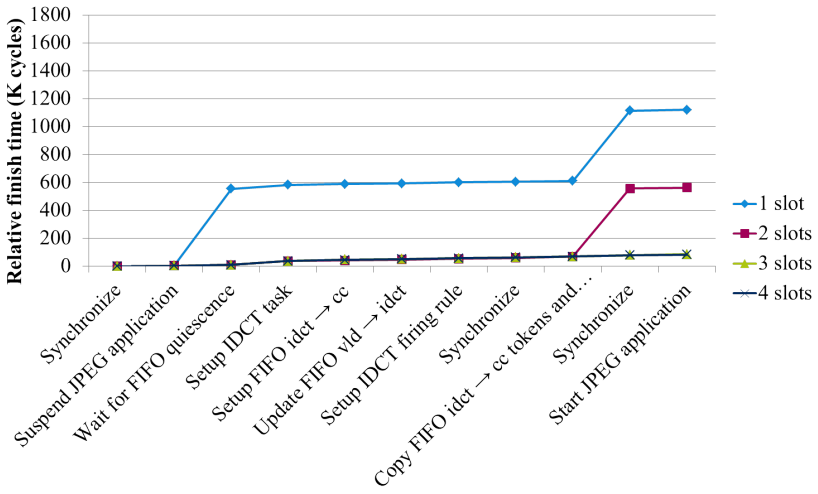


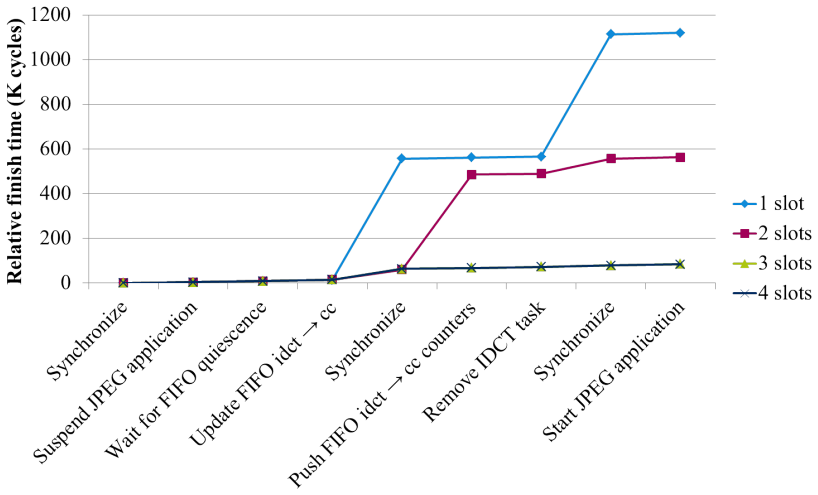
Figure 5.17: The execution time needed by the SA for migration of the IDCT task for the JPEG application on tile 2.

time a task is suspended to provide the required throughput. This is partly dependent on the reaction time, but also on the time it takes to migrate a task. Figure 5.17 shows, similar to the loading time, for every part of the migration protocol how long that part takes.

In Figure 5.17 we see that there are two parts of the protocol that take up the most time. These are all synchronizations, and from Figure 5.19a we see that FIFO quiescence also shows the same behavior. This is because when the ST processes the FIFO quiescence instruction, it also communicates with the remote ST, similar to synchronization. From Figure 5.19 can be derived that the migration time depends most of all on the configuration of the application scheduler due to these synchronization. From Figure 5.18 we see that small slot sizes for the application scheduler speed up the process of migration.



(a) Tile 1



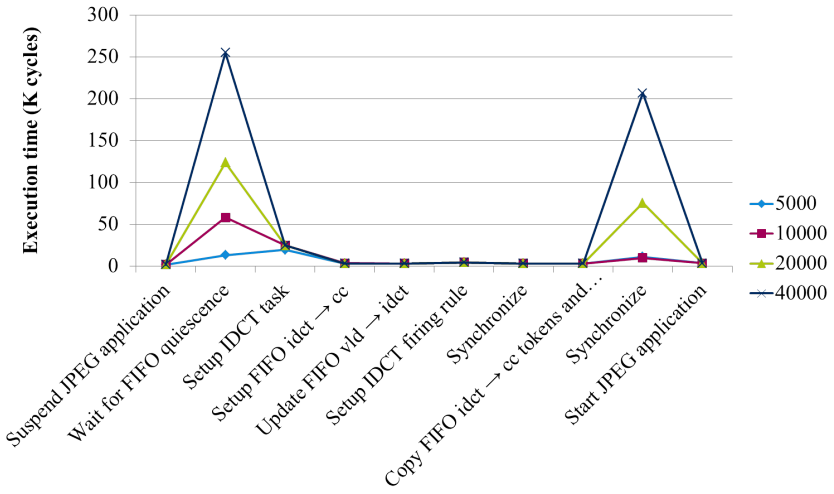
(b) Tile 2

Figure 5.18: Finish time of the iterations of the ST for migrating the IDCT task of the JPEG application for different application slot sizes

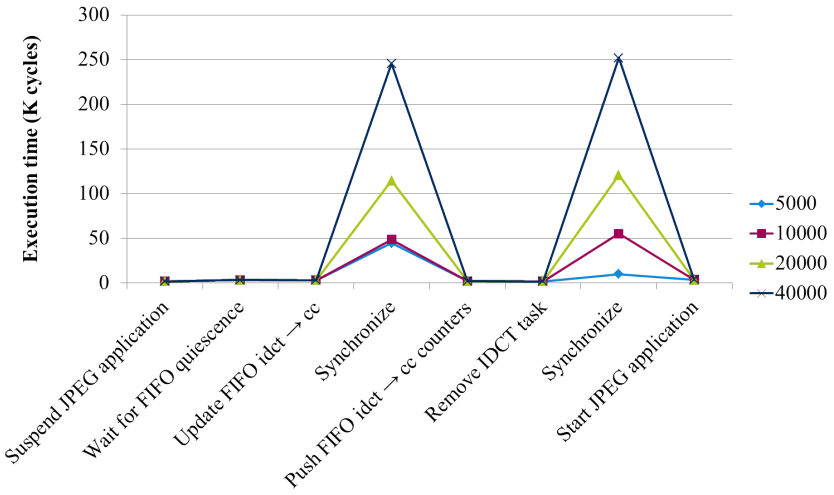
5.2.7 Resources used by the SA

Dynamic loading brings many possibilities, but at a cost. We added two Direct Memory Access (DMA) modules and the SA uses CMEM with a DMA module for retrieving instructions and descriptors, introducing overhead in hardware. Next, we extended CompOSe with functions to support dynamic loading and task migration and added source code for the ST. We investigate the overhead that the introduction of the SA brings by measuring the used hardware blocks, lines of code and used memory.

First we investigate the overhead of the hardware additions. From the synthesis report, we retrieve the number of hardware blocks which are 268 Lookup Tables (LUTs) per DMA module. Compared to the 15720 LUTs used for the platform, the relative



(a) Tile 1



(b) Tile 2

Figure 5.19: Execution time of the iterations of the ST for migrating the IDCT task of the JPEG application for different application slot sizes

overhead is only 0.034%.

Next, we look at the source code size. We have extended CompOSe Application Programming Interface (API) with 15 functions to facilitate both dynamic loading and task migration. These functions extend the source code with 20KB from 233KB to 253KB.

Besides CompOSe additions, we have the ST, where most of the functionality has been implemented. The source code of the system application is 44KB. This source code contains the task computation function, the firing rule function, 14 support functions and three debug functions. This still is a fairly simple and small application compared to the JPEG decoder (166KB) or H264 decoder (372KB).

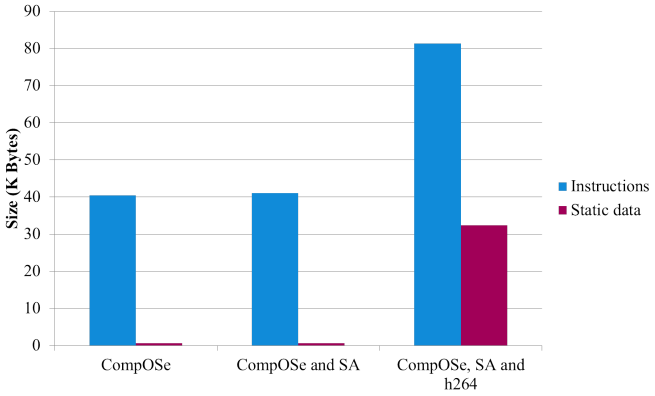


Figure 5.20: The size of the text section and data sections of the executable for tile 1.

Last we investigate the memory footprint for the SA. We split this in IMEM, DMEM and CMEM usage. We start with the instructions. To investigate the used IMEM we compare the text sections of an executable that contains only CompOSe and an executable that contains CompOSe and the SA. We compare the text section because it is the section where the source code is compiled into. Figure 5.20 shows that the SA extends the instructions by 564 bytes, only 1.3% of the total size of the text section.

Figure 5.20 also shows that we do not extend the static data sections, which is expected since we put all the data that is stored between iterations on a self FIFO channel, which is stored on the heap. Although no DMEM is used for allocation of static data, we still do use heap for the control blocks, including stack. The heap allocations for the SA on tile 1 are combined a total of 3096 bytes, which is only 2% of the entire heap.

The last part of memory use of the SA that we investigate is the allocated CMEM. We use the CMEM for storing the counters and data for the remote FIFO channels. Next, we use the CMEM to pull the descriptors from DDR. Because we load all the instructions and static data directly into IMEM and DMEM with the connected DMA modules, the required size of the CMEM is limited to the FIFO channel data buffers and their counters, and the size of the largest descriptor, including the linked data such as the initial FIFO token in case of descriptor for inserting a FIFO token. The FIFO channels require 24 bytes of CMEM per remote ST. In our experiment for tile 1, we have only one remote ST, and the buffer should have a minimum size of 392 bytes, which combines to a minimum requirement for the total amount of CMEM to 416 bytes.

To combine the memory footprint, we take the sum of the instructions, static data, dynamic allocated data (including) stack and required CMEM for the SA and compare it with a system that only contains CompOSe. We have a total of 4076 bytes required for the SA. The complete system without the SA requires 43288 bytes; hence we extend the footprint with 9.4

5.2.8 Qualitative discussion

In this section we briefly discuss the impact of three features of our platform that have not been supported by experiments. First we discuss the overhead in the OS time slot, next we discuss workload balancing. Furthermore we discuss the predictability of the system with dynamic loading and task migration.

Our design aims introduce no overhead in the runtime of CompOSe, since the OS executes between every application slot. The only addition we implemented is a check in the application scheduler for whether an application must be scheduled or not. The time that the OS executes is predefined in the source code, and has not been changed. Therefore there is no difference in the time that the OS executes in the existing platform compared to the platform with dynamic loading and task migration support.

One of the reasons for introducing task migration is enable workload balancing. We can achieve this by moving tasks around which we now offer support for, but have little tasks that can be migrated and only two processing tiles in our setup to experiment with. Although we have only migrated a single task, we see from Figure 5.10 that this migration does speed up the execution. Figure 5.10 shows that the time between the finish of two iterations of the VLD task of the JPEG decoder is reduced after the migration is done around iteration 20.

All the ST iterations are bounded in time, which makes the SA predictable. In all of the iterations, the ST fetches instructions from DDR. Since the NoC has a guaranteed throughput and latency, we know this is bounded by time. For most instructions, the ST fetches the descriptor with loading information which is also bounded in time because it uses the same NoC connection as fetching the instruction. The next step of the ST is to call the API functions of CompOSe, which are also bounded by time.

The only ST instructions where the ST does not follow these steps are synchronization and suspend task. The synchronization is bounded by time since we our FIFO channels over which we do the handshake are using NoC connections with guaranteed throughput as well.

Suspending a task requires waiting for a task to be in the idle state. Because we run predictable applications, we can assume that all the tasks end up eventually in the idle state. To cover all cases it is possible to extend the SA to stop the migration after a certain period of time has passed, hence we state that the iteration of this ST instruction is also bounded in time.

5.3 Summary

In this thesis we investigate if CompSoC extended with dynamic loading and task migration provides independent execution time, and we investigate the overhead and performance of the SA.

For this experiment we use an H264 decoder, JPEG decoder and an image rotation. The H264 decoder will be setup before runtime together with CompOSe and the SA. Then the SA loads the JPEG decoder, migrates the IDCT task and stops the JPEG decoder. Next, the Image Rotation application is loaded by the SA.

We show that there is no change in temporal behavior of the H264 between a run

where the SA is enabled and a run where the SA is not enabled. This shows that the SA does not influence the temporal and functional behavior of an application that is running on the system. Next, we investigate the difference in the temporal behavior of the JPEG decoder that is loaded between a run where the H264 decoder is running and one where the H264 is not set up. The temporal behavior of the JPEG decoder is exactly the same between the two runs, which shows that an application that is loaded on the system is not influence by an application that is running on the system.

In the following sections we investigate the time that it takes the SA to load an application. We show that pulling the instructions and static data of a task has the biggest impact on the load time. For loading the JPEG application on the tile where the VLD task is mapped, the iteration where the task instructions are loaded take up 90% of the load time.

We further investigate the time needed to migrate a task. This is split in the reaction time and loading time. The first is the time that it takes to wait for the task to be idle until the entire application can be suspended, hence the finish time of the first synchronization). Loading time is the time from suspending an application until the application is resumed.

Both the reaction time and migration time are influenced by the synchronization and FIFO quiescence. The time it takes to complete all the other steps is constant when we vary the application slot size and the amount of TDM slots that the SA can execute. By increasing the slot size, we increase the reaction time, but improve the migration time. By increasing the amount of slots for the SA, we reduce both the reaction time and the migration time, but this also reduces the available time for the applications to execute.

Furthermore, we investigate the overhead in software and hardware that the SA brings. The introduced DMA modules, extend the hardware platform by only 0.034%, and the task instructions in IMEM of the ST only extend the executable needed per tile to initialize CompOSe and the SA by 1.3%. When static data and dynamic allocated data is taken into account, the total memory footprint for the SA sums up to 4KB, which is an extention of the memory footprint of CompOSe by 9.42%.

In the last section we discuss that our platform does not extend the execution time of the OS, we discuss the opportunities for workload balancing and we explain that the SA is predictable.

6 Conclusion

In this thesis we propose and implement a dynamic loading and task migration protocol on CompSoC, a composable Multiprocessor System on Chip (MPSoC). CompSoC consists of a set of processing tile, a set of memory tiles interconnected by a network-on-chip. A processing tile contains the processor, and local instruction, data and communication memory, denoted with IMEM, DMEM, and CMEM, respectively. CompOSe, a light-weighted real-time operating system (OS) executes on each processor. An application consists of a set of tasks that communicate through (software) first-in first-out (FIFO) channels and a task scheduler. Internally, CompOSe uses control blocks to store information about the applications, the tasks and the FIFO channels between the tasks. Processor time is divided in slots; the OS schedules applications each slot according to a Time Division Multiplexing (TDM) policy. In the application time slot, the task scheduler arbitrates the processor among tasks.

Dynamic loading enables creation of separate executables per tile, per application; these executables can be easily reused among different CompSoC platform instances and mapped on various processing tiles within one CompSoC platform instance. In this way developers of applications will not be required to share their source code, since they are able to provide one executable per application.

Task migration enables even more flexibility at runtime. Tasks can be moved at runtime from one processing tile to the other, while maintaining the state of an application. This makes workload balancing possible, for example by moving tasks to processing tiles where the full processing capacity is not used.

We implement the dynamic loading and task migration protocol in an System Application (SA). This application is allowed to call the application management library provided by CompOSe. By implementing the protocols in application time, the critical Operating System (OS) time is unaffected. Furthermore, running applications are unaffected by the SA, since CompOSe guarantees independent temporal behavior of all applications, including the SA. This property provides that applications execute undisturbed, meaning that their temporal behavior is independent of other applications, from the moment they start, i.e., after they are loaded, until they are paused or stopped, i.e., when a use-case is switched.

The SA comprises of one System task (ST) running in each processor tile. The ST works by interpreting a set of ST instructions that we defined. These instructions contain an opcode to let the ST know what CompOSe Application Programming Interface (API) function it should call, and a pointer to a descriptor with loading or migration information. These descriptors correspond to the control blocks in CompOSe to have consistency throughout the platform.

In our design we store the ST instructions in a remote, DDR, memory tile, together with the descriptors and Executable and Linkable Format (ELF) files that contain the

instructions and static data for the tasks that are loaded. Every iteration the ST pulls a ST instruction and a descriptor into the Communication Memory (CMEM) to execute a part of the loading or migration protocol.

The dynamic loading process starts to initialize an application by setting up an Application Control Block (ACB). Then the instructions and static data for the tasks are loaded and an Task Control Block (TCB) is set up. After that the First In First Out (FIFO) channels are initialized by creating FIFO Control Blocks (FCBs). This setup allows the SA to load applications independently per tile. When the tasks of an application are mapped on several tiles there is need for synchronization, to make sure all the FIFO channels are initialized before inserting initial FIFO tokens. When the initial FIFO tokens have been inserted in the FIFO channels, the application is started. For this synchronization between the STs, we use FIFO channels. These FIFO channels have a small memory footprint, however this synchronization mechanism does require connections between all the STs, potentially limiting the scalability.

For loading the task instructions and static data, Direct Memory Access (DMA) modules are added to the Instruction Memory (IMEM) and Data Memory (DMEM). This has been done since in our prototype the processor cores cannot directly write instructions into IMEM and it leaves out the need for an indirect copy. The downside of this approach is that the DMA module cannot be used for multiple transfer requests at the same time, which means that we can only service one dynamic loading or task migration request.

The instructions and static data can now be pulled from the remote memory to the IMEM and DMEM, but they may be required to run from different physical addresses that the ones they were compiled and linked to. This introduces some requirements for the executables of the applications. First, all the task instructions need to exclusively use relative branches. This works if the memory layout of a task, i.e. the spatial order in which functions are stored within an address range, of the IMEM is the same as the one in the ELF files. The start of the address range where task instructions are stored in IMEM does not need to be the same as in the ELF file, hence instructions are relocatable.

Furthermore, some CompOSe libraries may need to be accessible by applications. Since the functions in these libraries are outside of the loaded code, the compiler and linker would not be able to statically generate the correct relative branches for the application. To solve this limitation we created indirect function calls. The CompOSe API functions are compiled, linked, and loaded, to a static address range, thus it is known at design-time where these functions are located in IMEM. We created a static table with the addresses of these functions and provided that to applications. In this manner, applications can correctly reference these functions. This table also allows us to limit the set of CompOSe API functions the applications can use, because the addresses of the other API functions are unknown to the applications. This offers a protection mechanism against erroneous or malicious applications that may make undesired calls to OS functions.

Moreover, some applications require static data that is stored in DMEM. It is preferred to have this data in a self FIFO channel, but not all existing applications implement this. In the existing applications the static data is accessed through physical addresses, which means the data should be placed at the physical address that the executable is

linked to. To be able to place static data anywhere in DMEM, we use the support for Position Independent Code (PIC) by the compiler. Enabling the PIC flag introduces indirect addressing for static data. A Global Offset Table (GOT) is used to store the addresses of the data. The instructions are compiled to load the address by a value in a special register plus an offset to the address.

The protocol to migrate a task from the source tile to a destination tile, has several steps. First it has to wait for the task on the source tile to reach a migration point. This is done to simplify the process of restoring the state of a task. Tasks of streaming applications have natural migration points immediately after they end an iteration. This guarantees that the task has no data on the stack and that its next iteration will start from the start of the task function. Since we only allow tasks that have no static data to be migrated, the ST only needs to copy the FIFO channel data to restore the state of the application.

After that the entire application is suspended, such that the FIFO channels will not be updated anymore. After suspending the application the ST follows a protocol to ensure that all FIFO channels are in a quiescent state. Then the task is recreated on the destination tile by dynamically loading it and creating or updating the control blocks for its FIFO channels. When the entire control structure has been created, we move the state of the task by copying all the FIFO channel data. Then the task is removed from the source tile and the application is resumed.

To check the functionality and performance of our implementation, we used an FPGA prototype of CompSoC extended with the proposed dynamic loading and task migration protocols. We use a Xilinx ML605 board to create an instance of CompSoC with two tiles that contain a MicroBlaze processor.

First we investigate if we can provide independent execution time of applications. This property is investigated by running an H264 decoder application together with the SA. The SA loads a JPEG decoder application, migrates one of its tasks and stops it. Then the SA loads an Image Rotation application in the memory space where the JPEG decoder was running. By comparing the computation time and finish time of the iterations of the Inverse Discrete Cosine Transform (IDCT) task of the H264 decoder of the two runs. Both the computation time and finish time are exactly the same, which shows no influence of the execution of the SA on a task running on the system.

We also show that it is possible to reuse the local memory. By introducing dynamic loading we only need memory for the instructions and data of the running applications instead of the memory needed for all the applications. Since the JPEG decoder is much larger than the Image Rotation application, we have in our setup a reduction of only 5% in required IMEM compared to the required IMEM for the existing system. However, we expect this percentage to increase when the number of applications increases.

We managed to keep the SA very small, extending the size of the instructions of the system executable (that includes CompOSe and processor initialization instructions) on a processing tile with only 1.3%. Furthermore, most of the time spend by the SA for loading an application is actually needed to copy the task instructions and static data in the local memory of the tile from the DDR remote tile. For loading the JPEG decoder with the Variable Length Decoding (VLD) task on a tile the ST needs 90% of the time to setup the task and load its instructions and static data.

To investigate the migration process, we look at the reaction time and the migration time. The reaction time is the time between the moment the migration is initiated until the task reaches a migration point. The migration time is the time that is needed to suspend the entire application, do the migration, and resume the application. We show that both the reaction time and the migration time depend on the slot allocation of the TDM table of the application scheduler, and depend on the application slot size. When we allocate more slots in the TDM table to the SA, both response time and migration time improve. Increasing the application slot size on the other hand only improves migration time.

Furthermore, the ST executes in bounded time and it is hence predictable. Exception may make the process of suspending a task, as this requires the task to reach the end of its iteration. However, this is not a problem since our applications are predictable and thus reaching the end of an iteration should happen in bounded time. In all other cases the ST fetches ST instructions and descriptors over NoC connections with guaranteed throughput and call OS functions that are also bounded by time.

Future work

In this thesis we provided an infrastructure to implement dynamic loading and task migration. The work can be further extended with intelligent policies that take into account various operating conditions, e.g., workload and temperature in different points of the SoC, and determine when to migrate tasks.

Another direction for future work could be a resource manager that can process loading and migration requests of applications. This allows applications to request extra tasks to make use of all the available processor capacity of the platform.

A third topic for further research is to investigate the trade-offs involved in implementing one ST per application. This way multiple loading and migration request can be handled, and the ST would not be a point of potential contention any more, which would make the platform with support for dynamic loading fully composable. This requires hardware extensions that have not been the topic of this thesis, such as writable IMEM. Currently, the IMEM and DMEM DMA modules can only service one request at the time, creating dependencies on other migration and loading requests.

Last, we noticed in our experiments that changing the application slot size or the TDM slot table of the application scheduler influences the performance of the migration. All of the variations in reaction and migration time with the TDM slots and application slot size are due to the synchronization mechanism between the STs. Since this synchronization also limits our scalability, a more in-depth research on this topic could improve both performance and scalability.

Bibliography

[1] B. De Sutter, D. Verkest, E. Brockmeyer, E. Delfosse, A. Vandecappelle, and J.-Y. Mignolet, "Design and tool flow of multimedia MPSoC platforms," *Journal of Signal Processing Systems*, vol. 57, pp. 229–247, 2009, 10.1007/s11265-008-0296-1. [Online]. Available: http://dx.doi.org/10.1007/s11265-008-0296-1

[2] G. Martin, "Overview of the MPSoC design challenge," in *Proceedings of the 43rd annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: ACM, 2006, pp. 274–279. [Online]. Available: http://doi.acm.org/10.1145/1146909.1146980

[3] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problemoverview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1347375.1347389

[4] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, pp. 157–177, 2004, 10.1023/B:TIME.0000045316.66276.6e. [Online]. Available: http://dx.doi.org/10.1023/B:TIME.0000045316.66276.6e

[5] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Boston, MA: Kluwer, 1997.

[6] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 2:1–2:24, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1455229.1455231

[7] R. Wilhelm *et al.*, "Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems," *IEEE Trans. on CAD*, 2009.

[8] B. Akesson, A. Molnos, A. Hansson, J. A. Ambrose, and K. Goossens, "Composability and Predictability for Independent Application Development, Verification, and Execution," in *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*, M. Hübner and J. Becker, Eds. Springer, 2011, ch. 2.

[9] G. M. Almeida, G. Sassatelli, P. Benoit, N. Saint-Jean, S. Varyani, L. Torres, and M. Robert, "An adaptive message passing MPSoC framework," 2009. [Online]. Available: http://dx.doi.org/10.1155/2009/242981

[10] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, ser. DATE '06, 2006, pp. 15–20.

[11] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J.-Y. Mignolet, "Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 234–239. [Online]. Available: http://dx.doi.org/10.1109/DATE.2005.91

[12] A. Hansson, M. Ekerhult, A. Molnos, A. Milutinovic, A. Nelson, J. Ambrose, and K. Goossens, "Design and implementation of an operating system for composable processor sharing," *Microprocessors and Microsystems*, vol. 35, no. 2, pp. 246 – 260, 2011, special issue on Network-on-Chip Architectures and Design Methodologies. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933110000608

[13] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and Y. Ha, "Analyzing composability of applications on MPSoC platforms," *Journal of Systems Architecture*, vol. 54, no. 34, pp. 369 – 383, 2008, system and Network on Chip. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762107001269

[14] K. G. W. Goossens, D. She, A. Milutinović, and A. M. Molnos, "Composable dynamic voltage and frequency scaling and power management for dataflow applications," in *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD 2010, Lille, France*, S. Lopez, Ed. USA: IEEE Computer Society, September 2010, pp. 107–114.

[15] J. Ambrose, A. Molnos, A. Nelson, S. Cotofana, K. Goossens, and B. Juurlink, "Composable local memory organisation for streaming applications on embedded MPSoCs," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11. New York, NY, USA: ACM, 2011, pp. 23:1–23:2. [Online]. Available: http://doi.acm.org/10.1145/2016604.2016631

[16] C. Meenderinck, A. Molnos, and K. Goossens, "Composable virtual memory for an embedded SoC," sept. 2012.

[17] V. Nollet, P. Avasare, J.-Y. Mignolet, and D. Verkest, "Low cost task migration initiation in a heterogeneous MPSoC," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 252–253. [Online]. Available: http://dx.doi.org/10.1109/DATE.2005.201

[18] P. Smith and N. C. Hutchinson, "Heterogeneous process migration: The tui system," Vancouver, BC, Canada, Canada, Tech. Rep., 1996.

[19] R. Busseuil, L. Barthe, G. Almeida, L. Ost, F. Bruguier, G. Sassatelli, P. Benoit, M. Robert, and L. Torres, "Open-scale: A scalable, open-source NOC-based MPSoC for design space exploration," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, 30 2011-dec. 2 2011, pp. 357 –362.

[20] A. Molnos, J. Ambrose, A. Nelson, R. Stefan, S. Cotofana, and K. Goossens, "A composable, energy-managed, real-time MPSoC platform," in *Optimization of Electrical and Electronic Equipment (OPTIM), 2010 12th International Conference on*, may 2010, pp. 870 –876.

[21] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Bus, K. Goossens, R. Peset Llopis, and P. Lippens, "C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," *Design Automation for Embedded Systems*, vol. 7, pp. 233–270, 2002, 10.1023/A:1019782306621. [Online]. Available: http://dx.doi.org/10.1023/A:1019782306621

List of definitions

Task instructions Bytes in IMEM that can be executed by the processor

Static data Bytes in DMEM in .data, .rodata .bss or similar sections where an applications stores information

Dynamic data Data on the stack and heap

Iteration The execution cycle of a task (with the states read, run, write, finish)

Dynamic loading Setting up the application during runtime

Preloading Setting up an application before runtime

FIFO channel A communication mechanism between a task that is producing tokens and a task that is consuming tokens

Token An data element in a FIFO channel

Tile A part of the CompSoC platform containing the processor, local and shared memories, DMAs and the VFCM

Processing tile A tile running an instance of CompOSe and able to run several applications

Loader tile A tile dedicated to distribute applications and tasks to worker tiles

Host tile A tile dedicated to setup the NoC, LEDs and other hardware instances of the CompSoC platform

Monitor tile A tile dedicated to process debug information

Host/monitor tile A combined host tile and monitor tile that does host tasks before monitor synchronization and monitor tasks after monitor synchronization

Monitor synchronization (mon sync) The process of all the tiles synchronizing with the monitor to have a predefined and known start time of the system.

Setup time The time before monitor synchronization

Runtime The time after monitor synchronization

System application An application that contains a System Task per processing tile

System task The task that controls dynamic loading and task migration by fetching ST instructions and descriptors

ST instruction The instructions for the ST in DDR that contain an opcode and a pointer to a descriptor.

Descriptor A structure with information needed by the ST to load an application or migrate a task

.got Data segment containing addresses to all the variables and constants

.data Data segment containing statically-allocated global variables and static variables that are initialized.

.rodata Data segment containing constant data.

.bss Data segment containing statically-allocated variables initially represented solely by zero-valued bits

Dynamic loading project generation

A

Dynamic loading enables the utilization of separate executables per application. In this section we will explain the preferred setup for creating loadable executable files. For this we use an instance of the CompSoC platform that we will describe first. The hardware platform and a single executable are loaded onto a Xilinx ML605 board. This executable, that contains CompOSe with the SA and ST for every tile, will be described in the next section. In the last section we will describe the preferred setup for compiling the application executables. The application executables contain the task functions and can be loaded onto the platform by the SA.

A.1 The generation of the hardware platform

The sources to generate a CompSoC instance and the source code for CompOSe and applications can be found in the CompSoC git repository. We will use an instance of the hardware platform with the following requirements:

1. The Double Data Rate Synchronous Dynamic Random Access Memory (DDR) should be accessible from all the tiles.

2. Every processor tile should have a DMA module connected to IMEM and a DMA module connected to DMEM

3. For every ST we require CMEM In and CMEM Out with a DMA module. This DMA module must be connected to DDR, and must be connected to the CMEM In of every other ST for synchronization.

4. The monitor tile must have read access to the flash card interface.

5. The monitor tile must have write access to the DDR to store the loading information

All these requirements have been fulfilled in the xilinx2tile6dma_migration example. We show below per requirement what should be added to the architecture.xml file to recreate this example.

To make the DDR accessible (1), the following IP should be in the architexture.xml of the example:

```
<ip id="shared" type="Memory">
  <parameter id="Type" type="string" value="DDR" />
  <parameter id="size" type="int" value="0x20000000" />
  <port id="pt" type="Target" protocol="MMIO_DTL">
```

```
    <parameter id="width" type="int" value="32" unit="bits" />
    <parameter id="address" type="int" value="0x40000000" />
  </port>
</ip>
```

For adding DMA module to IMEM and DMEM (2) the following ports should be added to every tile:

```
<port id="dma_pi" type="Initiator" protocol="MMIO_DTL">
  <parameter id="width" type="int" value="32" unit="bits" />
</port>
<port id="imem_pi" type="Initiator" protocol="MMIO_DTL">
  <parameter id="width" type="int" value="32" unit="bits" />
</port>
```

The connections to DDR and CMEMs from the DMA modules (3) are added in the communication.xml file.

The monitor tile should have access to the flash card (4) and to DDR (5). We provide the latter via a MMIO bridge that is already available in most of the examples. The access to flash is achieved by adding a parameter to the monitor tile. The entire IP defined in the architecture.xml for the monitor is as follows:

```
<ip id="monitor" type="Host">
  <parameter id="flash_card" type="bool" value="1" />
  <parameter id="mem_size" type="int" value="0x40000" />
  <parameter id="fsl_clock_gate" type="bool" value="1" />
  <port id="pi" type="Initiator" protocol="MMIO_DTL">
    <parameter id="width" type="int" value="32" unit="bits" />
  </port>
</ip>
```

A.2 Loading CompOSe with the SA

When the hardware platform is generated and the project with source code has been created, we have a fully functional platform that can already run CompOSe with several applications. We can run the platform from the following directory (where we use xilinx2tile6dma_migration as example project):

```
$(EXAMPLE_PROJECT)/work/xilinx/SDK/SDK_WORKSPACE/bit
```

To run CompOSe with the SA on Xilinx ML605 board, the bit folder provides a makefile from where everything is compiled and run on the FPGA board. We focus on the SA, together with CompOSe and the monitor, which can be run with the command 'make run-sa_rt'. In this section we explain CompOSe, SA and the used monitor one by one.

First CompOSe, this is the OS that can be found in the bsp folder. This has been extended with some functionality for dynamic loading and migration. It is fully functional

with the existing applications. When we reference to the CompOSe API functions, this is the folder where those functions can be found.

For every tile there is a folder with a main function to start execution on the tile. The folders per tile are sa_rt_mb0 and sa_rt_mb1. These folders contain the setup for CompOSe and the SA. The source code of the SA can be found in the folder 'SystemTask'. The source code in this folder contains the task function and support functions. The task function parses the instructions that it pulls from DDR and calls the support functions. Every support function is build up to first pull a control block from DDR with the loading information. Then it uses this information to call the CompOSe API functions.

The monitor is based on the clkgate monitor and does five things. First it initializes the board, including the network and LEDs. Then it stores the instructions and the loading information in the DDR. Third it loads the ELF files from the flash card and stores the files in the DDR as well. Then it sends the start address of the instructions in DDR to the processing tiles. As last step it does the monitoring exactly the same as the clkgate monitor.

This monitor assumes that the files are already on the flash card. To store the ELF files onto the flash card or load the ELF files directly into DDR without the flash card interface, we provide the option to send the ELF files through the USB interface. The monitor can directly write the ELF file into memory or stores it on the flash card. Compared to loading the ELF from the flash card, this leads to a later start time of CompOSe due to the low speed of the USB and the reaction time of the user to send the file to the USB interface. This could also lead to a later start time of CompOSe than the predefined start time that is achieved with clock gating, which leads to clock gate error messages.

A.3 The creation of the application ELF files for dynamic loading

The tasks of the application require instructions to execute the task function and a firing rule function. To create these instructions, the source code of the tasks and firing rules is compiled and linked into ELF files to be able to retrieve the information for loading easily. Figure A.1 shows the process of loading the instructions from the ELF file into IMEM and DMEM. On the left it shows the ELF file.

The instructions and static data are stored in a consecutive space in memory. The linker has created mapping information which we call the virtual address. This information is stored in the ELF file as well. Note that it is possible that some sections have no data but do take up space in memory. An example is the .bss section, which contains values that are initialized with 0. It is not needed to store these values in the ELF file, but these values do need memory allocated when the instructions are executed.

The virtual address does not have to correspond to the physical addresses at runtime as shown in the right of Figure A.1. It is also not necessary to load all the sections. This address translation is managed by the SA.

For every task and firing rule we load the instructions into IMEM. The source code

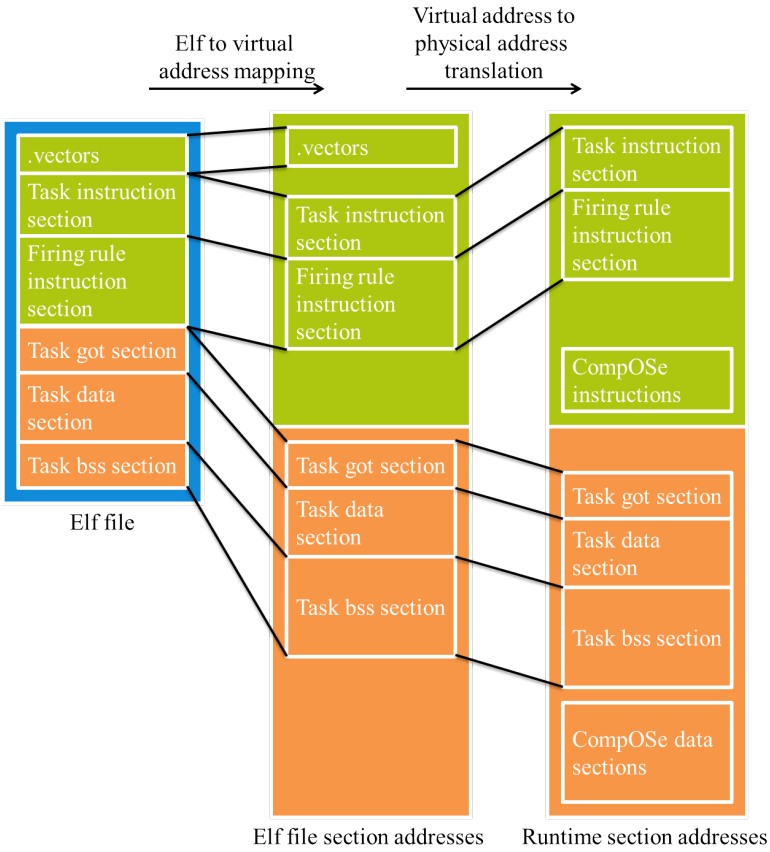


Figure A.1: Memory translation process from ELF to physical memory

of the firing rules of existing applications is fairly simple and thus we will focus on the task function, but the same rules apply to both. To be able to load the instructions on different physical addresses, we require Position Independent Code (PIC). This means that independent on where the instructions are loaded into memory, the code can execute. This means that static addressing for function calls is not allowed.

There are several options for compilation and linking that we will first enumerate and in the next section we describe the preferred setup. Then we discuss the other options we provide.

A.3.1 Options for ELF setup for loading task instructions

In this section we enumerate all the options for the setup of the ELF files. We distinguish three options on what tasks are contained in an ELF file:

a) One ELF file per application

b) One ELF file per task

c) One ELF file per tile

The instructions of the tasks can be compiled and linked in the ELF files as following:

d) One self-contained section per task

e) One self-contained section per task plus a shared text section

f) A shared section with all task functions

The static data can be compiled and linked in the ELF files as following:

g) No static data

h) Static data that is referenced by direct addressing

i) Position independent addressed static data (indirect addressing through a GOT)

The preferred setup is one ELF file per application (a), with every task in a self-contained section (d) and no static data (g). We discuss how to achieve this setup in the next section.

A.3.2 Preferred ELF setup for loading task instructions

In CompOSe we use CSDF graphs to model our tasks. This model implies that there is no static data between iterations (option g). All the data that needs to be saved between iterations can be passed on by self-FIFO channels. All data that is stored in a task iteration is therefore temporary and to store this data the task can use the stack. This would only require us to load instructions, which greatly simplifies the loading process and creation of ELF files. The instructions already are compiled to use relative jumps without the need for extra compilation flags, making them independent of the physical address.

The only problem are calls to the CompOSe API functions. These functions have already been mapped to a physical address that can vary by changes in the CompOSe source code. But we can make use of the fact that the physical address of these functions is known at design time.

We introduce a table in CompOSe, mapped to a specific address, containing the addresses of the CompOSe functions. A header file that must be included in the task function contains the address of this table, the structure of the table and definitions that create indirect function calls through the table from the original function calls. The definitions make it possible to make no further changes to the source code of existing applications. An example of such a definition is the following:

```
#define mon_debug_info(arg1)  _func_table->mon_debug_info(arg1)
```

In this code, _func_table is the pointer to the function table in CompOSe.

To load all the instructions of a task into memory, we first need to know where they should be loaded from, so we require the start address and the size of the instructions. For this we use sections in the ELF file per task (option d). We require this section to be self-contained per task, with the exception of the above explained indirect calls to CompOSe API functions. This means no jumps to code outside of the task section. Due

to memory restrictions we require this section to contain only the task function code. All other compiled sections, such as interrupt tables, initialization code or OS code, should not be loaded with the task function.

The separate sections also allows us to have one ELF per application (option a). Putting the task functions in separate sections requires the extension of the linker script with these sections, and requires the source code to specify in what section the task should end up. As mentioned, per task the instructions should be self-contained and thus one section with instructions should not have jumps to other sections. This does introduce the need for duplication of functions when they are used by separate tasks. For the firing rules we require separate sections as well, since we load the firing rules separately from the task functions.

To put a function in a specific section, the following code should be appended to all the function declarations of the task in the header file, where SECTION_NAME is the name of the section of the task:

```
__attribute__((section(SECTION_NAME)))
```

When this section has been created, the read-elf tool is used to show the offset from the start of the ELF file to the start of the section, and to show the size of the section. The read-elf tool can be found in the Xilinx gcc directory (XILINX/ISE_DS/EDK/gnu/microblaze/lin64/bin). The read-elf tool is used as follows:

```
mb-readelf -S application.elf
```

This will show the section headers with their name, address, offset and size. The offset and size are the values we need to load the instructions. The following is a part of the output of this tool:

```
Section Headers:
 [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
 [ 0]                   NULL            00000000 000000 000000 00      0   0  0
 [ 5] .text             PROGBITS        00000050 000124 0008d0 00 WAX  0   0  4
 [10] .got.plt          PROGBITS        00040000 002a84 00000c 04  WA  0   0  4
```

This output shows what is visualized in the left part of Figure A.1. Every section has a name, an address in virtual memory and a size in virtual memory. The offset value in this output it the offset of the section from the start of the ELF file. For the type we need to distinguish the PROGBITS and NOBITS. The first means that the section is filled with data, the latter means it has no data in the ELF file, but needs memory allocated. We offer no support for the expansion of sections so all the NOBITS sections should be at the end of the virtual address to be able to load sections together.

Because there can be several functions in the source code of a task, the start of the task function might not be at the start of the section. For this we need to find the task start offset, which is the start of the task function to the start from the start of the section. This is done with the object dump tool, which can also be found in the Xilinx mb-gcc directory. Executing the following will give the assembly output with source code of the ELF file:

```
mb-objdump -S application.elf
```

This output will show the address of every instruction. Because the source code is included, a search can be done for the start of the task function. The address that is shown is the physical address the instructions are mapped to, so by subtracting the start of the physical address of the task section we have its offset. The latter can be found in the read-elf tool output or in the output of the object dump tool. Due to the simplicity of the current firing rules, we assume that these only have one function and therefore the start of the firing rule will always correspond to the start of the start of the firing rule section.

To summarize, we have set the following restrictions:

- Static and global variables are not allowed. All data between iterations of a task must be in a self FIFO and all private data should be stored on the stack.
- The task source code must be self-contained. To use CompOSe API functions, a header file must be included that creates indirect function calls. Two tasks using the same function requires duplication of the function.
- For minimizing overhead the task instructions should be compiled in separate sections.

For loading the instructions into memory, we require the following values that can be found with the read-elf and objectdump tools in the Xilinx gcc directory:

- The size of the task section.
- The offset of the start of the task section from the start of the ELF file.
- The offset of the start of the task function from the start of the task section.
- The size of the firing rule section.
- The offset of the start of the firing rule section from the start of the ELF file.

A.4 Extended support for relaxing the restrictions

Some of the existing applications violate the requirement to have no static data, but we still want to be able to load these applications. We will discuss support for the use of static data and for the use of a single section for all the task functions so that functions can be shared amongst tasks.

A.4.1 Static data

Current applications do have static data that require loading sections of the ELF file into DMEM. Because the references to the data are static (option h), these sections need to be loaded on exactly the same place as defined by the ELF file. If multiple applications are loaded, either all applications should be in the same executable (option b), or changes to the linker script must be made to let the physical mapping to DMEM

in the application executable correspond with the location where the data is loaded by the SA.

A less complex solution when running multiple applications that require static data is to use PIC support of the mb-gcc compiler (option i). This introduces a table with addresses of the static data, called the GOT. Code compiled with the -fpic flag will use a reserved register to store a pointer to the .got.plt section and uses an offset from this pointer to find the address of the static data in the .got section. This indirection provides the task to use static data that can be loaded anywhere in memory provided that the GOT is updated accordingly.

To provide PIC the linker script is extended with the sections required for position independent code:

- .got.plt - The location the PIC register points to. The compiled code contains data references with an offset from this location to the address in the .got section.
- .got - The table with addresses to variables and constants in the data sections.
- .rela.dyn - Unused but required section when using the PIC flag.

To load the static data sections into DMEM by the SA the following values are required:

1. The size of the static data sections, including the GOT.
2. The offset of the start of the static data sections including the GOT from the start of the ELF file. We require the .got.plt section to be at this address.
3. The physical address that the .got.plt section is mapped to in the ELF file
4. The size of the .got section
5. The offset of the .got section from the .got.plt section in the mapped address space.
6. The size of the .bss section
7. The offset of the .bss section from the .got.plt section in the mapped address space.

All the static data sections need to be loaded together with the GOT. This requires the size of the all the data sections (1) and where it starts in the elf (2). These static data sections include the .data, .rodata, .bss, and we include the GOT as well in these sections.

To be able to update the GOT, we need to know where the static data should be loaded according to the ELF file (3) and the physical address the data is stored. The latter is known by the SA, so it can calculate the difference. Updating the GOT can only be done if the SA knows where the GOT is located in DMEM, hence we need the start address (5) and size (4) of the GOT.

The last part of loading is to initialize all data in the .bss section to zero. For this we require the size (6) and start (7) of the .bss section.

There will be only one GOT per ELF file. This means that either the tasks functions will share the data section, or there should be only one task per ELF file. The latter has

the preference, but we provide the first by loading the data with one of the tasks. By setting the start of the data 2 to NULL, no data will be loaded. The value in the start of the global offset table 5 will be used as pointer to the global offset table.

A.4.2 Shared functions between tasks

When two tasks share many functions, it generates a lot of overhead if these functions are duplicated. This is why we offer support for sharing a section between tasks (option f). If the start of the task instructions and the size are the same as the previous task, the SA assumes the instructions are shared. The offset to the start of the task function can differ to distinguish the task functions.

To make it easier to retrieve the start of the task computation function, it is advised to put this function in a separate section per task (option e). This way it is possible to find the start of the computation function from the mb-readelf output and leaves out the need for using mb-objdump.

C-HEAP protocol

B

The inter-task FIFO communication follows the C-HEAP protocol [21], which is briefly explained in this appendix.

A FIFO channel is a communication channels that enables tasks to communicate. A FIFO channel has a single producer task and a single consumer task. The producer task writes data tokens on the FIFO channel, which are consumed by the consumer task. The data token that is produced first will also be consumed first. To keep track of the amount of produced and consumed data the C-HEAP protocol introduces two administration counters:

- The write counter is a counter that keeps track of the amount of tokens produced in a FIFO channel.
- The read counter is a counter that keeps track of the amount of tokens consumed from a FIFO channel.

A FIFO channel can provide communication between tasks on the same processing tile, but also on separate tiles. In the first case, the administration counters are stored in the CMEM of the tile, as shown in Figure B.1. In the latter case a copy of the administration counters will be stored in the CMEM of both processing tiles, so that it can be quickly accessed to determine the availability of the data tokens in the FIFO.

There is always only one copy of the FIFO data. Regardless of where the producer and consumer are mapped, the FIFO data could reside in CMEM or in a separate memory tile. In this work we assume that the data is stored in the CMEM of the consumer task as shown in Figure B.2.

In the following sections we describe how the administrative counters of the FIFO channel are stored and updated by producer and consumer.

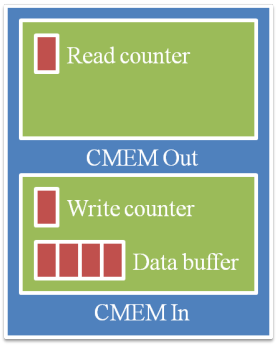


Figure B.1: FIFO data and administrative counters on a single processing tile

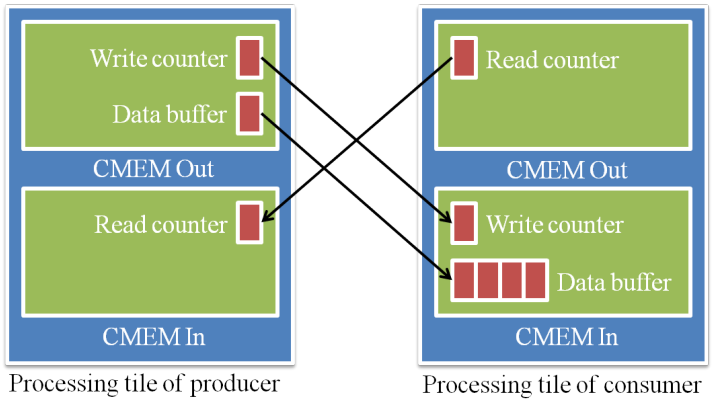


Figure B.2: FIFO data and administrative counters on separate processing tiles

B.1 Producer task

The producer task produces FIFO tokens on the FIFO channel. The FIFO tokens that were created are written to a temporary data buffer in CMEM Out. Then the DMA module can push this data to the remote data buffer at the tile the consumer resides. Because data is produced, the write counter is updated. This is done by first updating the Local Write Counter (LWC) in the CMEM Out and let the DMA module push it to the copy of the LWC on the tile the consumer task is mapped.

The counters at the processing tile of the producing task are mapped as follows on our platform:

- LWC - Counter in CMEM Out.
- Local Read Counter (LRC) - Counter in CMEM In.
- Remote Write Counter (RWC) - Pointer to the copy of the LRC at the tile of the consumer task.
- Data buffer - Buffer in CMEM Out to temporary store a FIFO token to push it to the remote tile.

For our migration protocol, we have extended the FCB to contain two extra pointers. These pointers facilitate the part of the protocol that ensures no data is transferring on the Network on Chip (NoC) as explained in Section 4.5.2. These administrative counters are mapped as follows:

- Local Finish Counter (LFC) - Value in CMEM In for comparing against LRC to check for FIFO quiescence.
- Remote Finish Counter (RFC) - Pointer to the LFC in the CMEM In of the tile the consumer resides where the the LWC is send to for FIFO quiescence.

B.2 Consumer task

The consumer task consumes the data from the data buffer. Since we assume the producer pushes the data to the data buffer at the CMEM In of the consumer, it is directly available for the consumer. It does need to update the read counter for every FIFO token it consumes. The read counter is also pushed to the CMEM In at the tile of the producer task, to update the local copy of the read counter there as well.

The mapping of the counters and data buffer at the tile of the consumer task is as follows:

- LWC - Counter in CMEM In
- LRC - Counter in CMEM Out
- RWC - Pointer to the LRC that resides in the CMEM In of the tile of the producer task.
- Data buffer - Buffer in CMEM In store the FIFO tokens.

Similar to the counters for a producer, extra values are needed at the tile of the consumer for the FIFO quiescence part of the migration protocol:

- LFC - Value in CMEM In for comparing against LWC to check for FIFO quiescence.
- RFC - Pointer to the LFC at the tile of the producer task.

C Applications

In this appendix we show the dataflow graphs of the applications that we use in the experiments and name the function of each task to provide an overview for the reader that wants to work and experiment with the platform.

C.1 Application 1 - H264

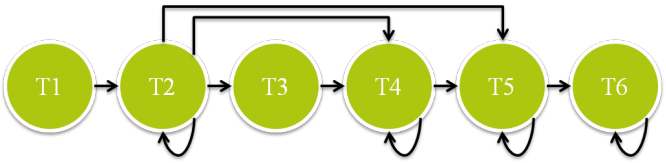


Figure C.1: The dataflow graph of the H264 application

T1. Network abstraction layer unit decoding and parse slice headers

T2. Parser and Context-Adaptive Variable-Length Coding (CAVLC)

T3. IDCT

T4. Intra prediction

T5. Deblocking filter

T6. Cyclic redundancy check

C.2 Application 2 - JPEG

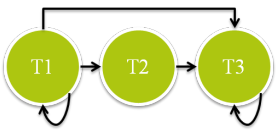


Figure C.2: The dataflow graph of the JPEG application

T1. VLD

T2. IDCT

T3. Color Conversion (CC)

C.3 Application 3 - Image rotation

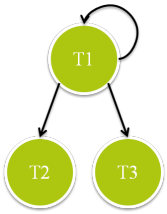


Figure C.3: The dataflow graph of the Image Rotation application

T1. Distribution of slices

T2 & T3. Processing slices