Beiträge aus der Informationstechnik

Mobile Nachrichtenübertragung
Nr. 97

Mattis Hasler

Towards Efficient Resource Allocation for Embedded Systems

VOGT

Dresden 2023

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Bibliographic Information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at http://dnb.dnb.de.

Zugl.: Dresden, Techn. Univ., Diss., 2022

Die vorliegende Arbeit stimmt mit dem Original der Dissertation „Towards Efficient Resource Allocation for Embedded Systems“ von Mattis Hasler überein.

Gesetzt vom Autor

ISBN 978-3-95947-061-2

Jörg Vogt Verlag
Niederwaldstr. 36
01277 Dresden
Germany

Phone: +49-(0)351-31403921
Telefax: +49-(0)351-31403918
e-mail: info@vogtverlag.de
Internet :   www.vogtverlag.de

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Elektrotechnik und Informationstechnik

Towards Efficient Resource Allocation for Embedded Systems

Mattis Hasler

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

Vorsitzender
Prof. Dr. Mikolajick

Gutachter
Prof. Dr. Gerhard Fettweis
Prof. Dr. Ulrich Rückert

Eingereicht am: 1.2.2022
Verteidigt am: 9.6.2022

Mattis Hasler
Towards Efficient Resource Allocation for Embedded Systems
Dissertation

Technische Universität Dresden
Vodafone Chair Mobile Communications Systems
Institute of Communication Technology
Faculty of Electrical and Computer Engineering
01062 Dresden, Germany

I would like to thank Gerhard and Emil for the guidance and supervision. I’d also thank everybody that considers himself my family for believing in me to finish this. Further, Robert shall be thanked for one or two good ideas “over a cup of coffee”.

Inhaltsverzeichnis

Abstract

The main topic is the dynamic resource allocation in embedded systems, especially the allocation of computing time and network traffic on an multi processor system on chip (MPSoC). The idea is to dynamically schedule a mobile communication signal processing pipeline on the chip to improve hardware resource efficiency while not dramatically improve resource consumption because of dynamic scheduling overhead. Both software and hardware modules are examined for resource consumption hotspots and optimized to remove them. Since signal processing can usually be described with the help of static data flow (SDF) graphs, the dynamic handling of those is optimized to improve resource consumption over the commonly used static scheduling approach. A hybrid dynamic scheduler is presented that combines benefits from both processing networks and task graph scheduling. It allows the scheduler to optimally balance parallelization of computation and addition of dynamic scheduling overhead. The resulting dynamically created schedule reduces resource consumption by about 50%, with a runtime increase of only 20% compared to a static schedule. Additionally, a distributed dynamic SDF scheduler is proposed that splits the scheduling into different parts, which are then connected to a scheduling pipeline to incorporate multiple parallel working processors. Each scheduling stage is reworked into a load-balanced cluster to increase the number of parallel scheduling jobs further. This way, the still existing dynamic scheduling bottleneck of a centralized scheduler is widened, allowing handling 7x more processors with the pipelined, clustered dynamic scheduler for a typical signal processing application.

The presented dynamic scheduling system assumes the presence of three different communication modes between the processing cores. When emulated on top of the commonly used remote direct memory access (RDMA) protocol, performance issues are encountered. Firstly, RDMA can neatly be used for single-shot point-to-point data transfers, like used in task graph scheduling. Process networks usually make use of high-volume and high-bandwidth data streams. A first in first out (FIFO) communication solution is presented that implements a cyclic buffer on both sender and receiver to serve this need. The buffer handling and data transfer between them are done purely in hardware to remove software overhead from the application. The implementation improves the multi-user access to area-efficient single port on-chip memory modules. It achieves 0.8 of the theoretically possible bandwidth, usually only achieved with area expensive dual-port memories. The third communication mode defines a lightweight message passing (MP) implementation that is truly connectionless. It is needed for efficient inter-process communication of the distributed and clustered scheduling system and the worker processing units' tight coupling. A hardware flow control assures that an arbitrary number of senders can spontaneously start sending messages to the same receiver. Yet, all messages are guaranteed to be correctly received while eliminating the need for connection establishment and keeping a low message delay.

The work focuses on the hardware-software codesign optimization to increase the uncompromised resource efficiency of dynamic SDF graph scheduling. Special attention is paid to the inter-level dependencies in developing a distributed scheduling system, which relies on the availability of specific hardware-accelerated communication methods.

Kurzfassung

Das Hauptthema ist die dynamische Ressourcenverwaltung in eingebetteten Systemen, insbesondere die Verwaltung von Rechenzeit und Netzwerkverkehr auf einem MPSoC. Die Idee besteht darin, eine Pipeline für die Verarbeitung von Mobiler Kommunikation auf dem Chip dynamisch zu schedulen, um die Effizienz der Hardwareressourcen zu verbessern, ohne den Ressourcenverbrauch des dynamischen Schedulings dramatisch zu erhöhen. Sowohl Software- als auch Hardwaremodule werden auf Hotspots im Ressourcenverbrauch untersucht und optimiert, um diese zu entfernen. Da Applikationen im Bereich der Signalverarbeitung normalerweise mit Hilfe von SDF-Diagrammen beschrieben werden können, wird deren dynamisches Scheduling optimiert, um den Ressourcenverbrauch gegenüber dem üblicherweise verwendeten statischen Scheduling zu verbessern. Es wird ein hybrider dynamischer Scheduler vorgestellt, der die Vorteile von Processing-Networks und der Planung von Task-Graphen kombiniert. Es ermöglicht dem Scheduler, ein Gleichgewicht zwischen der Parallelisierung der Berechnung und der Zunahme des dynamischen Scheduling-Aufands optimal abzuwägen. Der resultierende dynamisch erstellte Schedule reduziert den Ressourcenverbrauch um etwa 50%, wobei die Laufzeit im Vergleich zu einem statischen Schedule nur um 20% erhöht wird. Zusätzlich wird ein verteilter dynamischer SDFScheduler vorgeschlagen, der das Scheduling in verschiedene Teile zerlegt, die dann zu einer Pipeline verbunden werden, um mehrere parallele Prozessoren einzubeziehen. Jeder Scheduling-Teil wird zu einem Cluster mit Load-Balancing erweitert, um die Anzahl der parallel laufenden Scheduling-Jobs weiter zu erhöhen. Auf diese Weise wird dem vorhandene Engpass bei dem dynamischen Scheduling eines zentralisierten Schedulers

entgegengewirkt, sodass 7x mehr Prozessoren mit dem Pipelined-Clustered-Dynamic-Scheduler für eine typische Signalverarbeitungsanwendung verwendet werden können.

Das neue dynamische Scheduling-System setzt das Vorhandensein von drei verschiedenen Kommunikationsmodi zwischen den Verarbeitungskernen voraus. Bei der Emulation auf Basis des häufig verwendeten RDMA-Protokolls treten Leistungsprobleme auf. Sehr gut kann RDMA für einmalige Punkt-zu-Punkt-Datenübertragungen verwendet werden, wie sie bei der Ausführung von Task-Graphen verwendet werden. Process-Networks verwenden normalerweise Datenströme mit hohem Volumen und hoher Bandbreite. Es wird eine FIFO basierte Kommunikationslösung vorgestellt, die einen zyklischen Puffer sowohl im Sender als auch im Empfänger implementiert, um diesen Bedarf zu decken. Die Pufferbehandlung und die Datenübertragung zwischen ihnen erfolgen ausschließlich in Hardware, um den Software-Overhead aus der Anwendung zu entfernen. Die Implementierung verbessert die Zugriffsverwaltung mehrerer Nutzer auf flächen-effiziente Single-Port Speichermodule. Es werden 0,8 der theoretisch möglichen Bandbreite, die normalerweise nur mit flächenmäßig teureren Dual-Port-Speichern erreicht wird. Der dritte Kommunikationsmodus definiert eine einfache MP-Implementierung, die ohne einen Verbindungszustand auskommt. Dieser Modus wird für eine effiziente prozessübergreifende Kommunikation des verteilten Scheduling-Systems und der engen Ansteuerung der restlichen Prozessoren benötigt. Eine Flusskontrolle in Hardware stellt sicher, dass eine große Anzahl von Sendern Nachrichten an denselben Empfänger senden kann. Dabei wird garantiert, dass alle Nachrichten korrekt empfangen werden, ohne dass eine Verbindung hergestellt werden muss und die Nachrichtenlaufzeit gering bleibt.

Die Arbeit konzentriert sich auf die Optimierung des Codesigns von Hardware und Software, um die kompromisslose Ressourceneffizienz der dynamischen SDF-Graphen-Planung zu erhöhen. Besonderes Augenmerk wird auf die Abhängigkeiten zwischen den Ebenen eines verteilten Scheduling-Systems gelegt, das auf der Verfügbarkeit spezifischer hardwarebeschleunigter Kommunikationsmethoden beruht.

1 Introduction

1.1 Motivation

With every iteration of mobile communication standards, the complexity of the digital signal processing increases. In addition, the dynamic range of the processing complexity increases as well. That means, a base station has to be able to handle a very inhomogeneous set of connections in terms of required processing demands. In the fifth-generation (5G) the baseband digital signal processing covers a dynamic range of six orders of magnitude and —as far as we know today— this trend will continue in future standards. Because the timeframe for doing the signal processing stays constant a need for much higher processing power is needed. With the increase of clock frequencies being both more and more difficult to do and power consuming, parallelizing computation seems a promising alternative. Building a specialized application specific integrated circuit (ASIC) implementation for a problem like done in [45, 85, 1] to exploit parallelism will always result in a poor efficiency with regard to utilized hardware, because it has to be dimensioned for the worst (i.e. most demanding) case, leaving a significant fraction of hardware unused in the average case. However, the 3rd generation partership project (3GPP) defines a mobile communication channel to be chopped into transmission time intervals (TTIs), which makes the data a stream of basically independent data packets. The processing of each packet can easily be modeled as an static data flow (SDF) graph, allowing to process it on a general purpose multi processor system on chip (MPSoC). By dynamically assigning resources, hardware can be used more efficiently, saving costs at production as well as operation of base stations and terminals alike.

Running SDF graphs on MPSoCs is usually done using static scheduling, which is itself not efficient in terms of hardware utilization, at least in some situations. A typical signal processing SDF graph has a sequential start and end with a parallelizable hotspot somewhere in the middle. To fully parallelize the hotspot the schedule has to allocate many processors, which will be idle most of the time —except of the brief hostspot phase— which makes them poorly utilized, thus the schedule inefficient. With dynamic scheduling, however, processors can be freed immediately after the hotspot, allowing the next graph to compute it's hotspot fully parallel while still finishing the first graph sequentially.

Dynamic scheduling and hardware allocation has the potential to exploit parallel processing resources to stitch the needed computation pipeline together on the fly. The processing pipeline for each TTI describes exactly the needed resources so that it will only occupy needed resources. The goal of doing this is to optimize resource utilization. It is expected that the dynamic resource allocation has a negative impact on the processing time on a single graph due to the dynamic scheduler. The scheduling will introduce an amount of overhead effort[1]that has to be processed alongside the payload computation. A dynamic system may trade increased parallelism to speed up computation and additional overhead to decreasing efficiency. The ratio of parallelizing speedup and overhead slowdown effect is situation dependent and has to be considered in the live system.

The efficient utilization of the available resources allows to save power consumption e.g. by switching off unused computation units. Exploiting the parallelism of implemented algorithms allows lower clock frequencies compared to a serial implementation. A lowered clock frequency has a direct impact on the power consumption of a respected system.

In order to apply the dynamic scheduling and parallelization, every layer of the computation stack has to be optimized. Usually, the optimization of software assumes the hardware to be fixed. The algorithm is slimmed, or replaced, to better match the situation. In extreme cases, the programming language may be switched to eliminate unwanted factors like an interpreter layer of an indeterministic garbage collector. But, an attribute like real-time capability and also efficiency depends on the whole stack. When the application has to run on unsuitable hardware or operating system (OS), the upper layer may have difficulty creating an efficient execution profile. The problem can be explained with Amdahl's Law [3]. It describes that the impact, an optimization iteration has on

[1] "effort" is an abstract measure of the work that has to be done to execute a computer (sub)program.

the whole program is dependent on the relative size of the optimized part to the entire program. Parameters are the fraction of the original program f, that can be sped up (i.e. parallelized) and the speed up factor p by which this fraction is sped up. The total speedup then can be calculated by

$$S(p,f) = \frac{1}{(1-f) + \frac{f}{p}}$$

which has an upper bound depending on the fraction f for $p \to \infty$ of:

$$S_{\text{max}}(f) = \frac{1}{1-f}$$

This effect does not only apply within an application but also vertically through the computation stack. The usage of an operation provided by an underlying layer causes an amount of resource consumption. When optimizing the operation's implementation, the overall effect is dependent on the frequency the operation is used. A frequently used operation can represent a significant portion of the total execution time and may be worth optimizing. This kind of operation may be a system call into the OS or activation of a hardware accelerator.

Another type of resource consumption of underlying layers is not dependent on the hosted application, but instead occupies a static amount of resources. For example, a preemptive scheduler consumes a fixed amount of central processing unit (CPU) time by timed context switches, independent of the number or type of applications hosted. Also, mixed resource consumption may occur like a garbage collector that is invoked periodically, thus consuming a fixed amount of time. The amount of CPU time a garbage collector invocation consumes depends on the applications running and their behavior, e.g. how much objects they are allocating/freeing.

Each operation is the combination of operations from lower layers. The resources consumed by an operation are the sum of resources of all operations used by this operation. To optimize an operation identified as a hotspot, it is necessary to regard the current and decent to lower the layers. Examining all operations from all layers and their usage frequency can help find the cause of high resource consumption. It may help to find operations in lower layers that may be easier to optimize than the initial hotspot operation itself but still help to make it more efficient.

1.2 The Multiprocessor System on Chip Architecture

The concept of a MPSoC is nowadays a common one. The class of MPSoC architectures includes significantly different types with specific focuses. One purpose of a significant sub-field of MPSoCs is the ease of integration and usage efficiency. A microcontroller is designed to be easily integrated onto custom embedded printed circuit boards (PCBs). They usually include many communication capabilities removing the need for additional interfacing hardware. It helps keep the PCB design and development simple and cheap and lower power consumption by making additional chips unnecessary. Another famous MPSoC field covers processing platforms for single-board computers like smartphones. These chips resemble more a traditional CPU from desktop personal computers (PCs). Usually equipped with a multicore central processor, the main task is to host a standard OS like Linux. Similar to the microcontroller, additional components are included in the chip to save PCB complexity, space, and power. The additional components may vary from the specific application. For smartphone targeting chips, mobile communication modems and multimedia accelerators are the largest non-general purpose processing kernels.

Usually, an MPSoC is heavily overprovisioned —for one reason or another— in the sense that a significant fraction of the chip area is not or only seldomly used. A microcontroller, for example, often contains a multitude of peripheral interfaces. Most products/projects using a microcontroller only use a small fraction of the available interfaces, thus hardware logic. But still using a single microcontroller is usually more power-efficient than using multiple chips, each with a specific task. It is cheaper too because a microcontroller can be produced in vast quantities due to its fit for different tasks. To fit enough products/projects to justify the large quantities, it contains as many peripheral options as possible. Although most projects will only use a fraction of the chip's vast range of functionality, it assures the efficiency of the chip. The multimedia system on chip (SoC) makes use of the same principle by including certain special case logic like a video coding accelerator or a crypto module. It is beneficial to have an accelerator for a special task that may only be activated very seldomly but works efficiently. Again, the cheapness of logic on an ASIC allows for very special and maybe rarely used units and still generate a benefit for the application.

The MPSoC platform that shall be regarded here is a bit different in its focus, and therefore its architectural concept. While the mentioned MPSoCs get their name from the fact that they integrate not only a CPU on a chip but also a set of peripheral units, the class regarded here focuses on the execution of multiple (sub)-programs at the same time. Of course, a multimedia SoC can — and does— have multiple cores and can run numerous parallel programs. The mode of operation, however, is similar to a desktop processors multiprocessing approach. Multiple processing cores accessing one shared memory resembling something like a complex but only single von-Neumann-computer. The distributed SoC in contrast, features a set of multiple von-Neumann-computers that are, except for a shared communication sub-system isolated from each other. In a way, this kind of chip could be called "Systems-on-Chip" with emphasis on the plural of systems cf. to the single system of most controller SoCs.

Every Von-Neumann-System in such a system of systems must —to adhere to the definition [79]— feature a processing unit, a memory, and a communication unit. Concerning the system of systems, all three components are exclusive to this system and cannot be used by any other system. Calling this internal system a processing element (PE) makes the enclosing system a PE-cluster. The cluster now resembles a network of computers within a chip. A set of PEs representing individual and independent computers connected with a network of particular topology and technology. It allows the PEs to communicate by providing —on the lowest level— a way of sending messages carrying data from any PE to any other.

A PE can be of various types and have variable functionality. The most apparent PE would be a general-purpose computing unit. At the very least, it features a standard processor and local —on-chip— memory allowing the isolated execution of a program binary. Of course, a PE can have a more complex design, e.g. featuring multiple processors of different types. The memory may as well be a caching structure instead of a closely coupled memory that fetches cache lines in case of misses over the network from a remote memory. Instead, or even in addition to the general-purpose CPU a PE may also include special purpose accelerating hardware. But also without a general-purpose CPU, it can be useful being controlled through the networking unit. A CPU-less PE could be, to name a few examples, a DDR-RAM module, an LED-strip, an ethernet port, or and HDMI-controller. To assure the interoperability of these heterogeneous PEs unified access to the connecting network is necessary. For that matter, a communication protocol is defined that defines how PEs can send messages to each other.

From the network's point of view, a message is a set of data of a certain length. The transport of a single message can be described as a series of data chunks transported from router to router. The chunk transporting a certain amount in one cycle is called a flit. Depending on the network protocol, a flit may contain some of the message data and network control information. A series of flits traversing the network composes a message that transports the data intended to be sent by the network user. There are different ways of transporting a message. In a packet-switched network, each flit passes through the network on its own. The receiver has to receive flits individually and recompile the message. In contrast, a circuit-switched network allocates a tunnel through the network from the sender to the receiver. Once established, the message can pass through the tunnel as a whole, i.e. all flits directly one after each other. The receiver is sure that the whole message arrives in a continuous stream, and no reassembling has to be done. After the head flit, which has to carry the destination address to establish the tunnel, all other flits may be carrying almost exclusively data and don't have to include any header. The main problem of circuit switching is that deadlocks may happen in the phase of tunnel establishment. It is possible to avoid deadlocks by construction with a carefully chosen network topology and routing algorithm.

Common choices are ring topologies that may be extended into a forward and a backward ring. Rings are easy to implement, resource inexpensive, and deadlock-free. But the average distance between nodes is relatively high, and in some traffic cases, they are not much better than busses. Another common choice is an orthogonal mesh network yielding a lower average distance and better throughput in random traffic scenarios. A routing simple as X-Y is sufficient to assure deadlock freeness even for circuit-switched message transport. Also, other topologies are possible, like hexagonal or octal meshes or multidimensional torus networks to further increase network performance. They come, however, with more complex routing and increased chip area costs.

With a network available for sending messages between PEs the MPSoC platform must define a communication structure on top to allow the PEs to work with each other. The communication unit most likely implements a remote direct memory access (RDMA) protocol. It allows the PE to copy data from the local memory to a remote memory (e.g. the memory of another PE). This very simple and easy to use protocol stack can be used to model any communication protocol, but only with severe performance degradation. Therefore, a more sophisticated networking unit may be considered to implement other communication protocols like message passing (MP) or data streaming channels.

1.3 Concrete MPSoC Architecture

For the course of this thesis, a specific MPSoC architecture will be defined that served as the basis for all considerations. Its purpose is to build an overall picture that serves as a vessel for the discussions on the addressed hotspots. It will lean on the Tomahawk architecture, which has been developed and implemented in a series of chips for more than ten years. [32, 33, 61, 55] The Tomahawk architecture is a well-examined architecture from which many assumptions and results can be reused to create the models needed for the simulations setup in this work. The regarded tiled MPSoC architecture consists of a set of PEs connected by a network on chip (NoC). Each PE features an CPU that uses three memory ports (one for instruction and two for data) to connect to the local memory system. The memory system connects the CPUs and the networking unit to a local set of memory banks with a cross-bar like access controller. The networking unit provides a full-duplex interface to the NoC router. Each router connects to exactly one PE and four neighboring routers, resulting in an orthogonal mesh NoC. Depending on the focused hotspot, the MPSoC components are modeled in more or less detail.

1.3.1 NoC

There are numerous examples for NoC implementations [9, 80, 33, 61, 6]. In this work, the NoC is supposed to be a performant vessel used to build another communication layer on top. The NoC provides a message transfer mechanism. The NoC will transport messages of arbitrary length to the given destination PE. The NoC uses a circuit-switched routing algorithm, which divides the transmission into two phases. In the first phase, the wormhole is constructed, which may include waiting times due to congestions. The PE is then blocked from moving data into the network. Once the wormhole is completed, the receiving PE can start to read data from the NoC interface. For the length of this transmission, the NoC will not obstruct the data flow. Only the PEs are accountable for delays when they cannot read or write data fast enough. The NoC interface transports $s_{\text{flit}} = 128\,\text{bit}$ of data each cycle. The only exception is the first cycle, where a header of $s_{\text{header}} = 64\,\text{bit}$ is sent, leaving only $s_{\text{headdata}} = s_{\text{flit}} - s_{\text{header}} = 64\,\text{bit}$ bytes of data. As long as no congestions occur, the latency of the data is deterministic. The data spends $d_{\text{if}} = 4\,\text{cyl}$ (hardware unit clock cycle [cyl]) moving through the interfaces and asynchronous boundaries until it reaches the

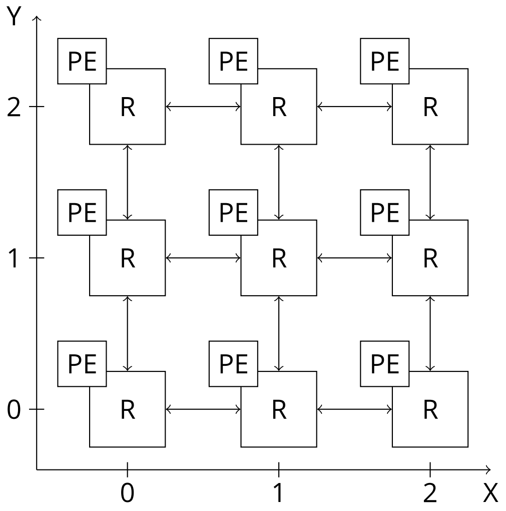

Figure 1.1: NoC topology with a orthogonal connected mesh network of $n = 9$ routers (R). Each router connects a single PE.

NoC router. Each hop to a neighboring router takes another $d_{\text{hop}} = 2\,\text{cyl}$ cycles. And finally, ascending to the destination PE takes another d_{if} cycles. The delay the header flit takes to reach a destination h hops away than becomes $d_{\text{header}}(h) = 2d_{\text{if}} + d_{\text{hop}}h = 8 + 2h$. To get the latency, a message of length s takes to traverse the NoC the number of flits it takes to store the message is added.

$$d_{\text{msg}}(h, s) = d_{\text{header}}(h) + \lceil \frac{s - s_{\text{headdata}}}{s_{\text{flit}}} \rceil \tag{1.1}$$

Arranging a set of n PEs as close as possible to a square (Fig. 1.1) gives an edge length of $\sqrt{n}$. The expected distance of two randomly selected PEs is the sum of the distances in X and Y dimension because of the orthogonal NoC connection pattern. The mean absolute difference of two uniformly distributed variables is $b/3$ with b being the upper bound of the distributions. Applied to the n PE system the average path length is $h = 2\sqrt{n}/3$. The average message latency assuming $n = 25$ PEs and a common signaling message length of $s = 64\,\text{B}$ (Bytes) then becomes:

$$d_{\text{msg}}(n, s) = d_{\text{header}}\left(\frac{2}{3}\sqrt{n}\right) + \left\lceil \frac{s - s_{\text{headdata}}}{s_{\text{flit}}} \right\rceil = 15\text{cyl} + 4\text{cyl} = 19\text{cyl}$$

1.3.2 Processing Core

The PE consists of one or multiple reduced instruction set computer (RISC) processors. For the considerations made in this thesis, it is not important what kind of processor is choosen. However, it is assumed that they may be application specific integrated processors (ASIPs) with an increased data bandwidth to match the bandwidth provided by the NoC. An ASIP processor is a RISC processor that has its instruction set architecture (ISA) extended by a set of commands to help accelerate an application-specific problem. Often the ISA extension comes with the extension of memory ports, essentially transforming the processor into specialized digital signal processor (DSP) [33, 61]. In this platform, we will assume an ASIP with two 128-bit data memory ports, as it was proven in [33, 61] to be a resonable configuration for the targeted applicaiton. It allows optimized algorithms to read-modify-write 16 bytes in a single cycle. This value is important to mention because the rest of the system has to be defined in a way, so that it can keep up with this data rate, e.g. the NoC that has to be able to bring in and take away data fast enough to keep the processor busy. Another issue to keep the processor busy is the connection to the local mem-

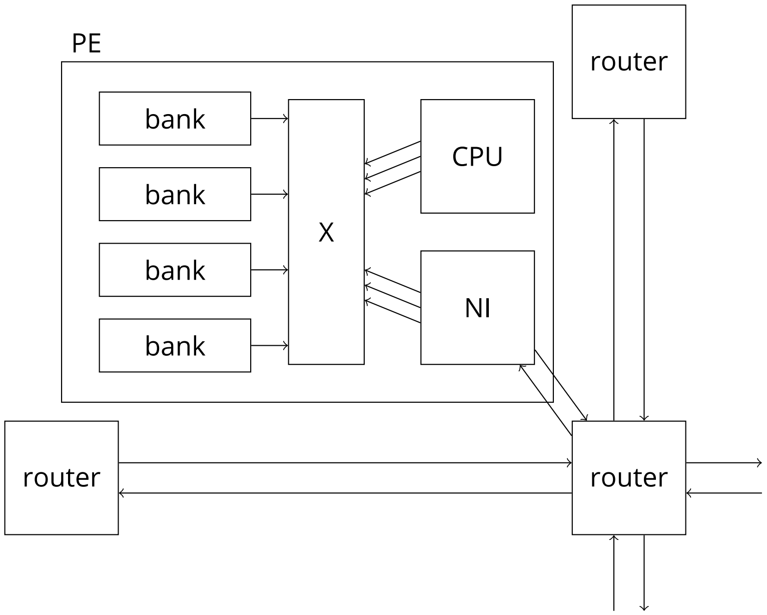

Figure 1.2: Block diagram of the processing element architecture.

ory banks. Both the processor and the networking unit have two memory ports for simultaneously reading and writing (sending and receiving) data and one for control information. In addition to the two data memory ports, a processing core features an instruction memory port. All three memory ports should be able to access the same memory locations (Fig. 1.2) making the physical separation of memory infeasible. Additionally, prior chip production has shown that dual-port memory consumes almost twice the chip area than single-port memory with the same data capacity. Since an MPSoC platform with local memory for the PEs consists mainly of on-chip memory, the storage density of the memory is an important factor. Because of these considerations, a memory system is used that utilizes a set of single port memory banks and connects them to a set of memory masters, allowing them to share access to a continuous memory space transparently.

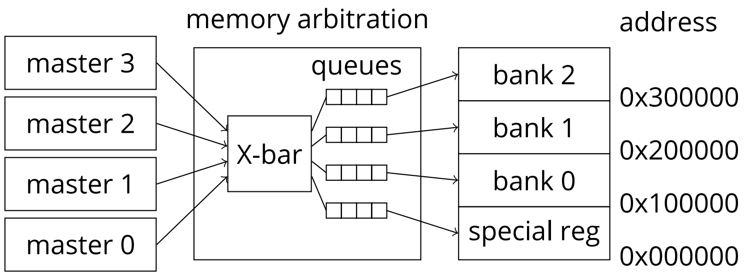

Figure 1.3: Memory system overview. Shown is the access path a master has to a desired location in a memory constructed by linearlly aligned memory banks.

1.3.3 Memory Management

The PE local memory management implements cross-bar like access functionality for a set of memory masters to a set of memory providers like shown in Fig. 1.2. This system was suggested in [83] to allow a most flexible distribution of memory access to serveral memory master (users), focusing on access collision prevention. Depending on the application it promises access performance similar to the usage of dual-port memory bank but with inexpensive single-port memory banks. A bank mapping table allows for each provider (e.g. memory bank) to appear in each master's memory space. Apart from a memory bank, a provider can also be the configuration space of a specialized hardware unit that is controlled with a memory-mapped configuration mechanism. For example, the networking unit's configuration register file is connected to the memory system as a memory provider. With the help of the mapping table access to the networking unit can be granted to or revoked from any master.

In the case that multiple masters request a location from the same provider, an arbitration policy will select one of the requests to be forwarded to the provider, signaling all other requesting masters that their request has been delayed.

The arbitration policy has a request queue for each memory provider. When a master request is routed to a specific provider, it will be appended to the requests queue. The request being in the pole position of the request queue will be granted access. As long as a master keeps requesting the same provider, it will remain in the queue and remains to have access to the provider if being in

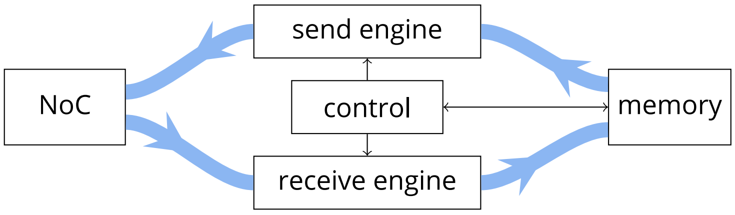

Figure 1.4: Block diagram of networking unit showing the two main parallel dataflows of receiving and sending data.

the pole position. The moment it stops requesting, by changing the address to another provider or clearing the enable bit, it is removed from the queue and has to enqueue at the back. A timeout also removes masters from the queue pole position to prevent master starvation by a single master that never changes location.

1.3.4 Networking Unit

The networking unit provides an automated way of exchanging data with other PEs. It is programmed using a memory-mapped special register file accessible by the masters through a dedicated memory provider. It consists of three basic parts as shown in Fig. 1.4: Two streaming engines (1) moving data from the local memory to the network, (2) moving data from the network to memory, and (3) a controller directing and controlling the streaming engines. Each part governs one memory master port, and the streaming engines each a network port.

The controller is in charge of programming both the send and the receive engine. It can program the send engine to stream a data range from memory to the network. Upon an incoming network message, the controller parses its header and programs the receive engine accordingly. The receive engine will then stream the remainder of the message to the programmed memory range. Both engines are designed to process 128 bit (i.e. one flit) of data each cycle to match the speed of the NoC.

Since the two data engines provide only low-level functionality, the controller is responsible for executing the different communication protocols. The dynamic nature of the targeted signal processing application demands communication modes with different performance focuses. In this networking unit three communication modes are included, each with a different type of communication in mind. The RDMA mode is used for one-time, high-volume, high-throughput, 1-to-1 bulk transfers. It does not need a sophisticated flow control but a simple connection establishment, i.e. the sender needs to know where to write the data in the receivers memory. The RDMA protocol is closely related to the direct memory access (DMA) protocol found in off the shelf desktop PCs, but implemented for a distributed memory architecture. In distributed-memory systems, it is often the only possibility of PEs to communicate with each other. Being separated by a NoC the PEs are otherwise unable to access each other's local memory. To reach a remote memory, the RDMA controller provides two methods. The "put" is used to copy a local range to a remote PE's memory, where the "fetch" method copies from a remote memory to the local one.

The first in first out (FIFO) mode enables constant, high throuhput streams of data without adding a lot of software overhead to the application. Two PEs can communicate through a unidirectional channel reaching from the sender PE to the receiver PE. To use the channel, the sender only writes to a local data buffer. The data will be transported automatically to the receiver into a local buffer, where the receiver can collect it.

For small messages, where response delay is crucial, like in signaling communication (e.g. requesting a service), a MP mode is provied. It provides the capability to quickly and efficiently send small messages to a PE's message box without the need of a connection establishment. The message box is a random access buffer for messages that can be received from multiple senders.

1.4 Representing LTE/5G baseband processing as Static Data Flow

As already mentioned (in Section 1.1) the input of a digital signal processing stage of a mobile communication setup is a stream of more or less independent packets. Each represents the data for one TTI and its processing can be viewed as an isolated problem. The complexity may vary dramatically depending of on several factors like number of antennas, the set of users, their applications and various channel properties. The range of differnt packet configurations already is big for 4G and 5G and is considered to further increase for future mobile communication standards.

For example, the 4G uplink receiver baseband processing of a TTI-packet may be simplified to a simple SDF-graph like shown in Fig. 1.5. Although the basic

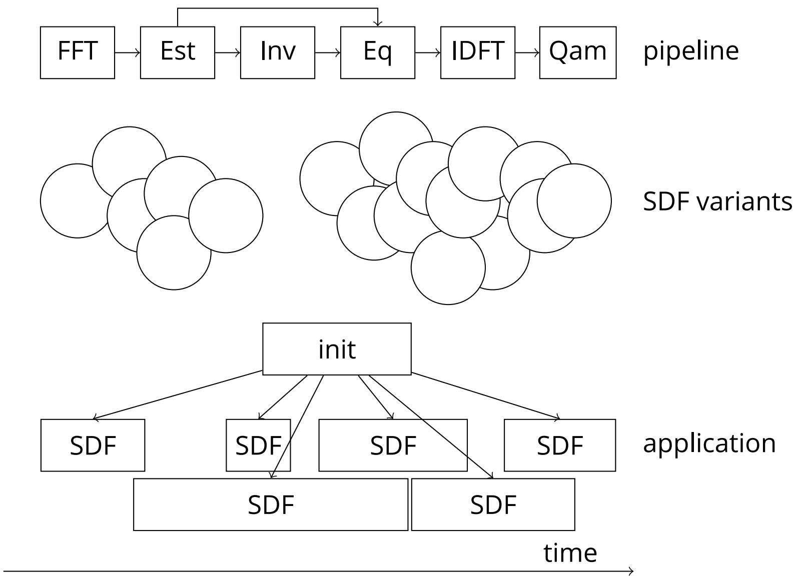

Figure 1.5: LTE uplink pipeline as generic flow chart, two abstract SDF representations resulting from different parameter sets, and as distributed application using different graph instances.

graph may not change depending on the configuration, the complexity of the processing, may still vary. In the simplest case this may manifest in the number and size of tokens transfered on the SDF channels and the firings each SDF actor has to conduct.

An efficient execution needs to employ different strategies to distribute computation to a set of PEs. Alongside different communication modes are needed to support those strategies. Each single firing of an actor may be placed on a different PE because of computation needs. In this case bulk transfers are needed to move the needed data to many different PEs. In contrast simple but high throughput actors may stay on a sinlge PE and be connected with pipelines to assure unobstructed data processing. To quickly react to the ever changing computation needs an MP mode is needed. It allows the efficient implementation of a dynamically generated execution plan on the available PEs.

1.5 Compuation Stack

The computation stack is a set of layers consisting of software and hardware constructs forming a system that is able to do a computation. The efficiency of a system depends on all layers of the computation stack:

- The “algorithm” is a description of a solution to a problem in a computer-executable manner.
- The “language” layer specifies a set of commands and operations to define an executable program.
- A “runtime environment (RTE)” provides a framework with functionality to support the execution of a program with recurring tasks like inter-thread communication. Also, resource allocation can be a task of the RTE especially in distributed computation.
- The “OS” provides security functionality like the isolation of programs to the platform. Isolation of resources implies the allocation of those. In contrast to the RTE, the OS focuses on isolation instead of performance optimization.
- The “drivers” can be included into the OS layer. Since the OS layer is considered optional here, but the drivers are not, they are listed in a separate layer. Drivers provide an abstraction of some functionality from the used hardware. For example, a driver may provide the functionality to send a message to another PE without the application knowing what kind of networking unit is present.
- The “hardware” is the lowest layer, providing the actual manipulation and storage of data. The definition can often be partitioned into units for various tasks like data storing, mathematical operation, moving data, controlling program flow, application-specific accelerated data manipulation, etc..

Generally speaking, each layer provides functionality to the layers above by abstracting and refining functionality provided by the layers below. The costs at which functionality is provided can be measured as the use of two base resources. The two resources that are of interest are the occupation of computing time and memory. While the management of memory is an important topic for the efficiency of a distributed memory system, this work's primary focus will be the computation time. Although the occupation of memory in local memories is omitted, the transfer of data through the network is considered because it can consume a significant amount of time.

Each layer defines a set of operations an upper layer may issue. The issue of an operation is defined by an implementation. An implementation is defined as a set issues of operations of lower layers to produce the desired result. The total resource consumption then is the sum of all issued operations. There are two possibilities to optimize an operation. One is to optimize the implementation to use fewer or cheaper operations of the lower layer. The other possibility is to decent one layer and optimize operations frequently used in the current implementation.

Another way how each layer can affect the execution time is through static resource consumption. The static consumption of resources is independent of the executed program. It may, for example, be a fixed portion in each time slice. For example, in a preemptive scheduling environment, the OS will interrupt the running program and do context switches with a fixed frequency, using a portion of computational resources. Likewise, a communication library decoupled from the CPU delays the progress of the distributed application without directly interfering with the CPUs.

Optimizations of operations can be done on any layer, and each will affect the application's execution time. The effective speedup for the application caused by the optimization can be described with Amdahl's law. With p being the portion of resources consumed by the operation in relation to the total resource usage, and s being the speedup of the optimized operation[2], the effective speedup is described as [3]:

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

An operation may be a candidate for optimization if the product of its resource consumption and the number of issues is high. It may be more beneficial to optimize a low-level operation used in many higher-level operations. Therefore, it accumulates a more significant resource consumption than a complex high-level algorithm issued only once at program startup. In the following, all

[2] Operations are considered sequential. The speedup of an operation is usually achieved by an more efficient implementation. Speedup of the application by parallelization is not directly connected to operation optimizations. An efficient computation stack, however, shrinks parallelization overhead, thus helps application speedup.

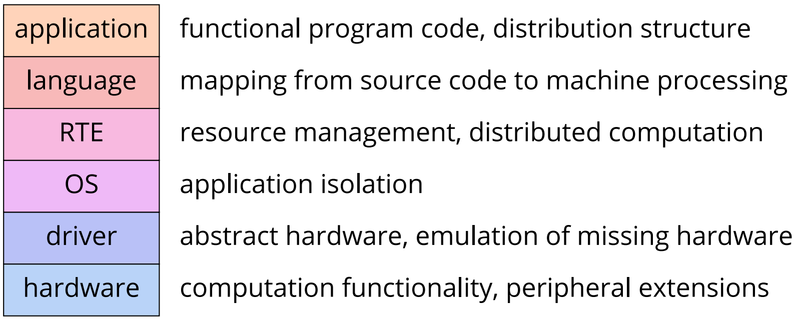
application
functional program code, distribution structure
language
mapping from source code to machine processing
RTE
resource management, distributed computation
OS
application isolation
driver
abstract hardware, emulation of missing hardware
hardware
computation functionality, peripheral extensions

Figure 1.6: Layers of a distributed computing platform.

layers that take part in the execution of a program will be briefly described, and possible hotspots are being analyzed.

1.5.1 The Algorithm and Application Layer

The way an algorithm is defined often also determines the computational effort posed to the platform. Usually, little can be done to improve algorithms' efficiency without going into domain-specific details of the application. Switching the algorithm or parts of it with simplified or heuristic approaches can reduce complexity in exchange for accuracy. However, it is the task of the application developer to decide for an algorithm that consumes as little computational effort as possible while producing results that are sufficiently precise for the intended application.

However, an algorithm can be selected or designed to be adapted to the given platform constraints with regards to the other layers. The targeted environment here is designed for distributed applications. It means that there are multiple independent and isolated PEs connected with a distinct communication fabric. The PEs are expected to be fairly small, with a simple CPU and a small connected memory, so that already a medium-sized application cannot be run on a single PE, but will have to spread over multiple. For example, an application describes as a DSP pipeline does fit the architecture well and may lead to high performance. Each stage can be hosted on another PE, and communication infrastructure provides cheap and performant data transfer between stages.

1.5.2 The Language Layer

Choosing a programming language is important as it can introduce a significant amount of overhead. Several attributes may give hints about overhead, resource consumption, and performance. An “interpreted” language (i.e. script) almost always introduces a lot of overhead through the indirection of operation by a virtual machine. Examples are “python”, “java”, “MATLAB”. The “garbage collector” many languages utilize is activated spontaneously and will consume a significant amount of computing resources. It is not exclusive to interpreted languages, for example, used in “C++” or “go”. Other attributes that also may impact performance are dynamically evaluated types for implementing “polymorphism” and the intensive use of various dynamic objects like “associative arrays” or “dynamic lists”.

Most languages, however, are specifically designed for a certain type of application and environment. Scripting languages like python or javascript mostly focus on simplicity and convenience for writing software. Other languages like “C” and “Rust” are designed explicitly for resource-limited systems and performance craving applications. The for a certain is therefore dependent on the project and usually easily decidable.

1.5.3 The Runtime Environment Layer

The RTE provides an abstraction for an application from the available processing and communication resources. The topology of the distributed application is defined. It is described as a set of processing and data objects and their relation to each other (e.g. read/write access). The RTE’s purpose is to map these objects to the physically available resources, i.e. processors and memories, while incorporating communication cost of object transfers between resources.

The performance issue that can arise in this layer is that every structure requires a certain amount of (computation) resources to be managed. If the application is partitioned into too small parts, or the management operations are too costly, the relative overhead will be significant, making execution inefficent.

On the flip side, the RTE can dynamically utilize available resources and react to changes in the available resources or the application. It allows the RTE to optimize computation efficiency on any given platform size or topology.

1.5.4 The Operating System Layer

The OS layer is, for a platform like the one considered in this work, non-essential. The task is to insert a security layer that isolates the applications and sometimes also the system services from each other. There are various possible ways of how applications could interfere with each other and as many methods to prevent each one. The most trivial inter-application interference is access to another application's memory space. Writing to foreign memory is considered malicious behavior, but even reading could reveal confidential data like cryptographic keys. Apart from security reasons, isolation also has a safety benefit, as the damage a malfunctioning software can cause can be contained easily. On off-the-shelf desktop systems running a variety of very different software simultaneously, the benefit from increased safety and security prevails. In contrast, on an embedded MPSoC, that probably only runs a single piece of software, the overhead introduced by the OS quickly becomes significant. The OS layer may be skipped entirely in this case to keep the performance.

1.5.5 The Driver and Library Layer

The driver and library layer provides an abstraction from the hardware implementation of a set of needed functionality. Depending on the available hardware, it may be necessary to emulate some functionality to fulfill all requirements of an application programming interface (API) exposed to upper layers. For example, a library may implement and MP protocol stack based on an RDMA hardware module if no MP hardware module is present. It allows programs from the higher layers —may it be the OS the RTE or the application directly— be developed against a consistent API, e.g. a MP API, without caring about the available hardware.

Drivers and libraries often contribute a significant amount to the resource consumption of an application. Especially frequently used operations like communication primitives can represent a lot of overhead. The optimization of these operations is often tricky because they mostly consist of the translation of function calls to programming hardware modules through memory-mapped config files. But the large number of issues to these small operations make already small savings in computation worthwhile. However, in some cases it may only be possible to slim the configuration process by changing the hardware module itself, or at least it is the more promising way to go.

1.5.6 The Hardware Layer

The hardware layer is unique as it does not depend on a lower layer's operations but only provides functionality upwards. The most apparent operations are an implementation of a general-purpose ISA and the possibility to store data. Optimization in other layers consists of avoiding issues to lower layers or using cheaper ones, where optimization in the hardware layer works differently. When optimizing at hardware layer, new operations are defined and implemented directly in hardware. The additional operation can be used by other layers instead of implementing the same functionality with a sequence of basic operations. For example, a floating-point unit provides operations that can calculate floating-point arithmetic in very few cycles where a software implementation based on integer operation may take around a hundred cycles to complete [68]. Similarly, larger algorithms may be implemented in hardware to shorted execution time. Not only local calculations but also communication operations may benefit from specialized hardware implementations. The availability of a FIFO channel unit or and hardware MP protocol stack to omit software libraries can help save resource consumption.

1.6 Performance Hotspots Addressed

The previous section stated that to optimize an application decently, the whole system, meaning each layer, must be addressed and optimized towards the given requirements given by the application. Some hotspots will be addressed in the following chapters, and optimization strategies used to improve system performance. There will be no space to cover hotspots in all layers, yet four will be visited that were noticed to be significant in earlier times.

(1) Describing a distributed application is a relatively new problem in computer science. For the most time, a program has been considered sequential with one thread following a track of commands through a program, occasionally and conditionally jumping to create all kinds of decision making. With the advent of shared-memory multiprocessors, this idea has been enhanced by a single consideration: The existence of multiple threads that live in the same memory space. Harnessing the potential of a multi-threaded, memory-shared system is a complex task that poses the danger of unwanted side effects. There are many frameworks, libraries, and OS support to increase safety, security, and ease-of-use of those systems. The functionality that these RTE and OS level solutions

provide always follows two principles: (a) The threads belonging to an application are isolated from each other in memory except for a precisely defined location used to implement the communication. (b) The communication functionality is exposed to the application either as a FIFO-like channel or a message sending service. Access to the whole memory is restricted to a data pipeline allowing serial access to both communication partners or a messaging system working on evenly sized data blocks. In addition to that, most desktop-focused multiprocessing libraries do not consider to apply the same canalization to the computation, chunking the application into inter-dependent processing blocks. With a resource management unit providing isolation and canalization for both data and processing time, optimization can already be done on application level by just choosing the right data and computation types for subprograms.

(2) Although in desktop targeting OSs the structure of chunking the processing time into tasks is usually not provided, some projects offer a task running RTE with the tasks usually drawing from a SDF graph. A problem that arises when implementing such a system is that the management overhead becomes significant relative to the actual computation depending on the task size. The basic relation here is that each task introduces at least a fixed amount of overhead work independent of the task's size. That means having smaller tasks decreases the task to overhead ratio until the management becomes the bottleneck of the system, not allowing the exhaustion of the systems processing power. The main reason is that most systems centralize the management into a single-threaded unit, unable to scale. With a more or less fixed overhead for one task, the system's task throughput is determined by the management unit's speed and is independent of the task size, as long as tasks stay small enough. For bigger tasks, the bottleneck becomes the system's processing power leaving the management unit on idle for some time. Making the management system a distributed application itself would help to scale with the amount of overhead work that needs to be done. The management system could adapt to the task size to occupy exactly as much processing resources as needed to fill the remaining resources with tasks.

(3) The first two hotspots are dealing with software; the third and fourth hotspots address problems in the hardware layer. The main focus here lies in communication technics. The classical computation system does not need specialized communication hardware because any communication protocol can be built based on shared memory. When turning away from shared towards distributed memory, dedicated communication functionality becomes essential. The most common technic even found in desktop systems to relieve the CPU

from long data copy operation is the DMA controller. Although not needed in shared memory systems, it is used to parallelize data transfers and main computation in shared memory systems. In distributed-memory systems, where memories are isolated from each other, the DMA is often the only possibility to exchange data between nodes. The shared memory DMA only needs a single operation, which is copy data from one location to another. In contrast, the distributed memory DMA distinguishes between local and remote memories and defines two operations, one for sending data to and one for pulling data from remote memory, forming the widely used RDMA function set. Although the RDMA allows implementing all communication formats, some may suffer from performance issues. In this work, two chapters are devoted to improving communication in distributed memory systems by implementing specific communication protocols in hardware.

One of the two protocols implements a FIFO channel, used for example, for pipelined signal processing. The flow of data from one MPSoC node doing one processing stage to the next is entirely offloaded to a dedicated hardware unit allowing the CPU to concentrate on the number crunching. This form of communication expects a constant and high amount of data rate.

(4) The second communication protocol provides MP functionality and is vital for distributed application design. It suffers from performance loss when implemented on top of a RDMA stack. When used for synchronization of nodes and requests to service nodes, the main issue a RDMA based implementation has is the message delay. A hardware (HW) implementation can significantly improve the delay by removing unnecessary protocol layer messages. Another problem addressed is extensive memory consumption and the overhead of managing connections states.

1.7 State of the Art

Building a complete dynamic SDF execution system covering every layer from hardware up to the application is a massive task. It has hardly ever been done (best to our knowledge). Also, this thesis struggles to relate everything to everything else. But each of the regarded hotspots does have their field of related work that they can be placed into. At the beginning of each chapter, a more detailed "state-of-the-art" section will cover work related to the topic at hand. Anyway, a quick overview of the most prominent works will be given here, also covering fields not touched in this work but worth mentioning.

There have been several attempts to build an MPSoC with a focus like the one described in Section 1.2 like [33, 61, 40, 65, 14, 24, 54, 6]. Each project has its own set of hardware units featured to support a specific type of use case. The communications features on a MPSoC are often regarded with great detail and versatility reaching from NoCs [9, 80, 5, 6] over DMA controller [63] to FIFO implementations [77, 39]. One step higher on the stack, on the RTE layer, several works are dealing with SDF-like graph processing on multi-core systems [15, 58, 53, 12, 49, 67, 56, 18] and also works that review the hardware and RTE layer together [75, 72]. A technique that could not be covered in this work but worth mentioning and interesting for future work is the clustering of tasks to save overhead effort [22, 29]. A lot of works combine two layers they regard jointly skipping several layers on the computation stack, like Long Term Evolution (LTE) implementation on SDR a platform [8], on a general multicore architecture [66], and on field programmable gate array (FPGA)-based solutions [45, 85, 1]. But most state-of-the-art work assumes standard components in all layers except the one that is improved. There are a multitude of different dataflow models [51, 42, 4, 10, 31] mostly adding different attributes to SDF, new languages to express stream processing [19, 21, 76, 81] and even efforts on compiler techniques to optimize stream processing on multi-processor systems [26].

1.8 Overview of the Work

In this work the key challenges will be highlighted that are most likely to be a deal breaker when designing an efficient dynamic SDF execution system. As already mentioned in Section 1.1 the efficiency of an embedded platform depends on every layer of the computational stack (Section 1.5). Although each layer is important and can have a game-breaking impact on performance, only a few layers will be regarded closely in this work. Some layers can be ignored because they are not mandatory or unlikely to impact performance when used correctly. Using “C” as programming language is unlikely to cause a performance issue on its own, because of its lightweight nature, but it is cumbersome and error-prone to write and thus may be replaced with some modern language to increase development efficiency. Similarly, an OS can be inserted for portability or security reasons. Both cases require extensive investigations to verify the impact on the resource allocation efficiency of the whole system. For that reason, although presenting interesting possibilities to explore, both layers are left out of this work.

What will be discussed in this work are four topics that are expected to impact system performance significantly. On all four topics, a solution matching the problem at hand is, to our best knowledge, not present. First, the topic of distributed application representation is visited in Chapter 2. A hybrid description will be developed that aims to combine the advantages of task graphs and process networks. Additionally, it introduces the issue of balancing the overhead and the payload calculation of a distributed application. This overhead-payload-ratio is revised in the second topic concerning the platform's and application's management instance distribution in Chapter 3. Both topics implicitly assume efficient data transfers within the system. As already mentioned (in Section 1.4) these are bulk transfers, pipelines and efficient message passing.

Where bulk transfers can efficiently be done with widely available RDMA [43, 41, 44, 2] engines, the other two need a closer review. In Chapter 4 the efficient realization of inter-PE data pipelines are investigated. The usual high throughput demands of pipeline connections require close analysis and optimization of local memory access. In Chapter 5 the missing transfer mode ("message passing") is examined. The many-to-one relation of client-server relations demands resource efficiency on the server-side. Additionally, the message delay has a non-neglectable impact on the possible utilization of the server process.