



**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

Zusammenfassung	xii
Kurzfassung	xiii
1 Introduction	1
1.1 Motivation	1
1.2 The Multiprocessor System on Chip Architecture	4
1.3 Concrete MPSoC Architecture	7
1.3.1 NoC	7
1.3.2 Processing Core	9
1.3.3 Memory Management	11
1.3.4 Networking Unit	12
1.4 Representing LTE/5G baseband processing as Static Data Flow	13
1.5 Computation Stack	15
1.5.1 The Algorithm and Application Layer	17
1.5.2 The Language Layer	18
1.5.3 The Runtime Environment Layer	18
1.5.4 The Operating System Layer	19
1.5.5 The Driver and Library Layer	19
1.5.6 The Hardware Layer	20
1.6 Performance Hotspots Addressed	20
1.7 State of the Art	22
1.8 Overview of the Work	23

2	Hybrid SDF Execution	25
2.1	Addressed Performance Hotspot	25
2.2	State of the Art	26
2.3	Static Data Flow Graphs	27
2.4	Runtime Environment	28
2.5	Overhead of Delaying Tasks to a MPSoC	30
2.6	Interpretation of SDF Graphs as Task Graphs	32
2.7	Interpreting SDF Graphs as Process Networks	34
2.8	Hybrid Interpretation	34
2.9	Graph Topology Considerations	35
2.10	Theoretic Impact of Hybrid Interpretation	37
2.11	Simulating Hybrid Execution	40
2.12	Pipeline SDF Graph Example	45
2.13	Random SDF Graphs	46
2.14	LTE-like SDF Graph	50
2.15	Key Learnings	53
3	Distribution of Management	55
3.1	Addressed Performance Hotspot	55
3.2	State of the Art	56
3.3	Revising Deployment Overhead	57
3.4	Distribution of Overhead	58
3.5	Impact of Management Distribution to Resource Utilization	59
3.6	Reconfigurability	64
3.7	Key Learnings	64

4 Sliced FIFO Hardware	67
4.1 Addressed Performance Hotspot	67
4.2 State of the Art	69
4.3 System Environment	71
4.4 Sliced Windowed FIFO buffer	72
4.5 Single FIFO Evaluation	74
4.6 Multiple FIFO Evaluation	77
4.6.1 Traffic Model	78
4.6.2 Evaluation Setup	80
4.6.3 Effective Channel Bandwidth	81
4.6.4 Memory Access Model	83
4.7 Hardware Implementation	86
4.8 Key Learnings	87
5 Message Passing Hardware	91
5.1 Addressed Performance Hotspot	92
5.2 State of the Art	92
5.3 Message Passing Regarded as Queueing	93
5.4 A Remote Direct Memory Access Based Implementation	95
5.5 Hardware Implementation Concept	97
5.6 Evaluation of Performance	99
5.7 Key Learnings	107
6 Summary	109
Abbreviations	113
Symbols	115
Publications	117
Bibliography	127

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 Scheduling 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-Aufwands 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 teuren 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 partnership 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 hotspot 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 its 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 effort¹ 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 $p \rightarrow \infty$ of:

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

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 $S_{\text{flit}} = 128$ bit of data each cycle. The only exception is the first cycle, where a header of $S_{\text{header}} = 64$ bit is sent, leaving only $S_{\text{headdata}} = S_{\text{flit}} - S_{\text{header}} = 64$ bit bytes of data. As long as no congestions occur, the latency of the data is deterministic. The data spends $d_{\text{if}} = 4$ cyl (hardware unit clock cycle [cyl]) moving through the interfaces and asynchronous boundaries until it reaches the

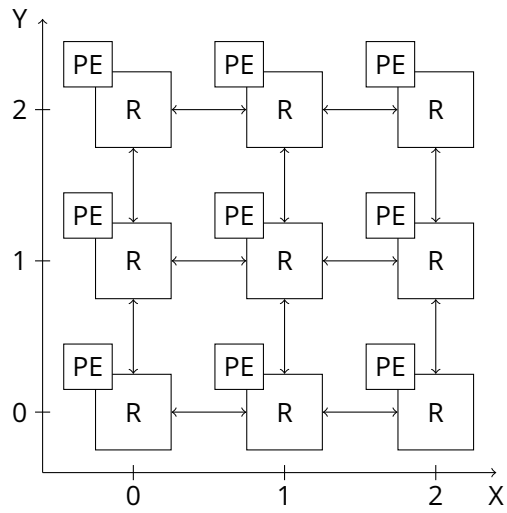


Figure 1.1: NoC topology with a orthogonal connected mesh network of $n = 9$ routers (R). Each router connects a single PE.

NoC router. Each hop to a neighboring router takes another $d_{\text{hop}} = 2$ cycl 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 then 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.

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

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 $s = 64$ B (Bytes) then becomes:

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

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 chosen. 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 reasonable configuration for the targeted application. 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-

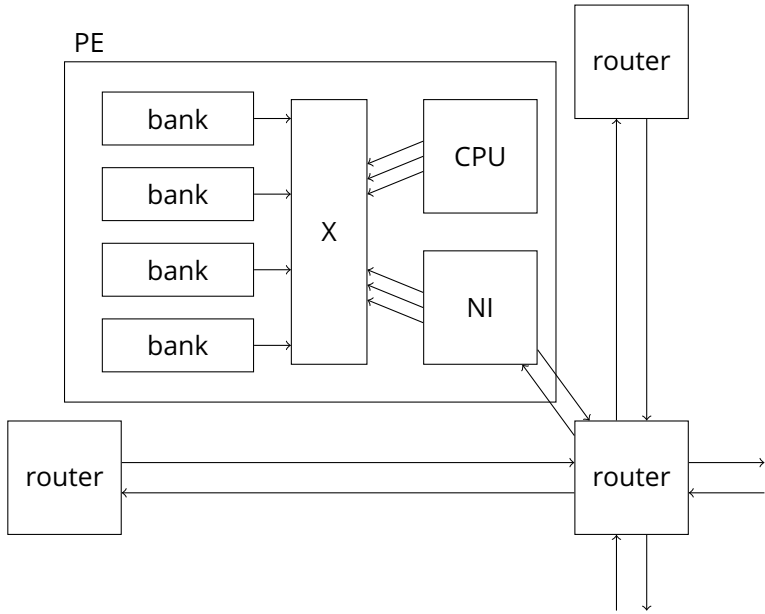


Figure 1.2: Block diagram of the processing element architecture.

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.

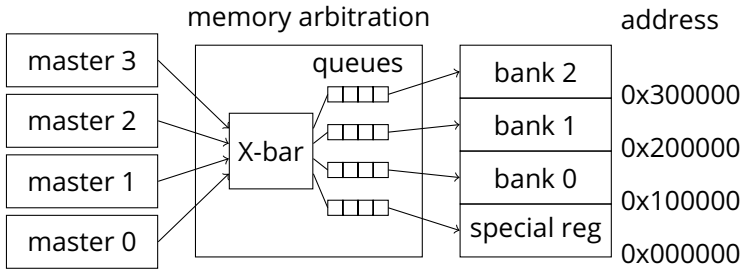


Figure 1.3: Memory system overview. Shown is the access path a master has to a desired location in a memory constructed by linearly aligned memory banks.

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 several 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

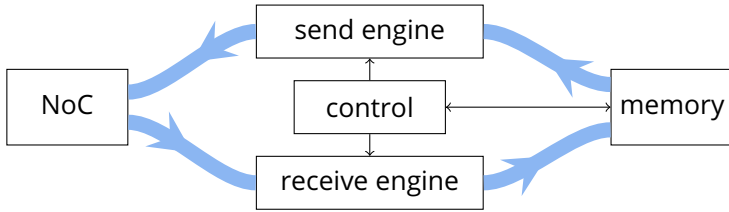


Figure 1.4: Block diagram of networking unit showing the two main parallel dataflows of receiving and sending data.

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 throughput 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 provided. 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 different 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

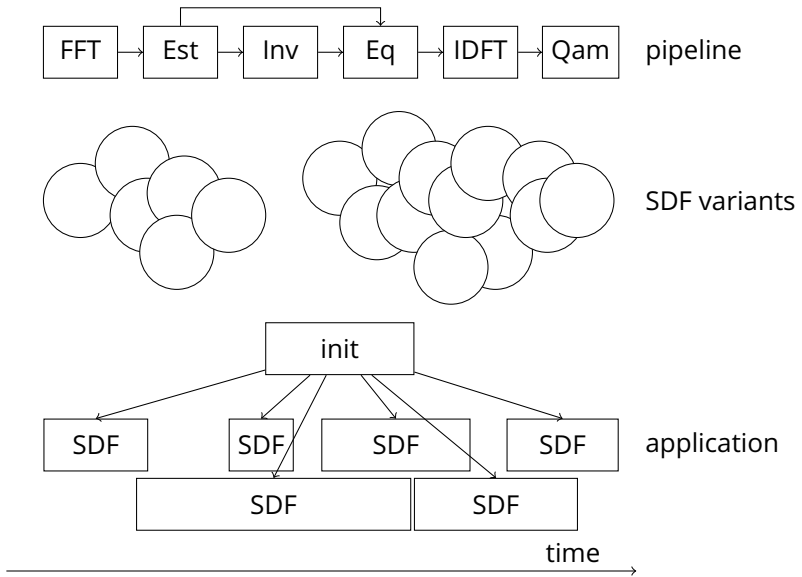


Figure 1.5: LTE uplink pipeline as generic flow chart, two abstract SDF representations resulting from different parameter sets, and as distributed application using different graph instances.

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 transferred 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 single 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 Computation 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 operation², 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

²Operations 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 inefficient.

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 an 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 shorten 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 revisited 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.

2 Hybrid SDF Execution

In this chapter, one performance bottleneck will be addressed. The layer of the runtime environment (RTE) is responsible for deploying an application described in the superjacent layer to the platform defined by the underlying layer. The task is to find the most optimal strategy to interpret the application and assure the correct execution of it. The performance in this regard is considered the amount of computational effort a given system can process in a given period. Two different common approaches are examined in this work and fused to form a new hybrid strategy that can outperform preexisting work.

2.1 Addressed Performance Hotspot

When mapping an application to a hardware platform, an RTE faces multiple optimization problems. As stated before, the application here will be described by an static data flow (SDF) graph. For example, each SDF describes the computational effort for one transmission time interval (TTI) packet. A stream of packets then results in a stream of independent SDF graphs. A performance indicator is the time the system takes to execute all computational effort of a single graph. A more sophisticated performance measure is to determine the frequency at which the system can process graphs.

One usual approach to process an SDF graph is to convert it into a task graph. Each firing of an actor is represented by a task, the channels by data blocks passed between tasks. A bottleneck that such a scheduling system can experience is caused by the scheduling overhead every task scheduling system inherits. With a sufficiently large platform or small tasks, this overhead cannot be

neglected. Eventually, the system's performance is solemnly limited by the number of tasks the scheduler can submit. The workers are left idle for a significant time, making the system inefficient.

Another way of processing an SDF graph is as a process network. Each SDF actor is represented by a process, that are connected with data pipelines to resemble the original graph topology. According to the SDF definition, a process consumes data from the pipelines. When all necessary data is collected, the functional kernel is fired (i.e. executed), produced data is pushed to the outgoing pipelines and the process resumes to the data collection stage. This way scheduling overhead is diminished compared to the task graph where every firing has to be scheduled individually. On the downside a parallel execution of multiple firing of the same actors is not possible.

In the often applied static scheduling, minimal live scheduling cost is combined with the ability to run one actors firings in parallel. The assignment of firings to processors is done offline and only replayed in the live situation. It requires however that the processors chosen in the offline scheduling are available or at least a block of an offline defined number of processors are available. That means, that effectively, a block of processors is occupied over the complete schedule's makespan, even is certain processors are only used at a fraction of the time. This lead to a lower system efficiency or to put it the other way around, demands overprovisioning of the hardware, which was one initial reason to choose a dynamic scheduling approach.

2.2 State of the Art

There are many works targeting stream processing. They do, however, illuminate different levels of the matter and always have a specific focus. Some works define their own language [76, 26, 81, 71] like for example the CAL language [21] which causes the need for a specialized compiler. Universal multiprocessing toolsets like MPI [30] define their data abstraction on a very low level, loading the burden of task prerequisite allocation and data handling to the application. Other works present RTEs for running process fleets without the notion of inter-task data dependencies [15, 81], with the inclusion of hardware resources [75, 72] or with data and stream management [58]. Some works specify an API for defining process networks [58, 72]. The DANBI RTE [58] defines a programming model for irregular programs of kernels communicating through queues. There are some RTEs for stream processing, but all of them focus on a specific

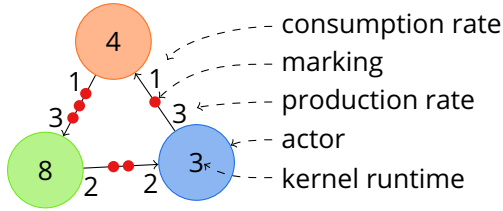


Figure 2.1: Cyclic SDF example graph comprised of three actors with different kernel runtimes, connected by channels of varying data rates and initial markings.

model of computation (MoC)¹, which is usually the one that is presented in that work. It is hard to compare MoCs because each is only supported by a certain RTE. In addition to limiting themselves to a specific MoC there are projects that even define an own language for their RTE like StreamIT [76, 26] or the CAL language [21]. This further decreases the comparability because the compiler contributes a fair amount to the performance of an application. Universal multiprocessing toolsets like MPI [64] define their data abstraction at a lower level. There, an application defines processes, running in parallel, that can communicate directly with messages passed from one (or more) process(es) to one (or more) process(es). In contrast, within the system regarded here, processes interface only with data containers. Cass et al [15] propose an RTE of agents that are executed distributed over multiple instances. It is, however, only designed for the kind of stream processing language Little-JIL [81]. There is an operating system called BORPH [75], which is used to build an integrated HW/SW system where hardware units act like normal software processes [75]. DANBI is an RTE for process networks [58].

2.3 Static Data Flow Graphs

SDF [52, 50] is a well known and well-examined execution model that describes an application as a set of n actors connected by channels. An actor contains a functional kernel that performs a piece of computation each time activated. In Fig. 2.1 the actors are displayed as circles with the computational payload kernel

¹A MoC defines the way an application describes the to be done computation. It may be a sequential description as in traditional programming but also any kind of operator networks

runtime k_i , written inside. The activation—or firing—will happen when all input ports of the actor collected a predefined amount of data-tokens. After the kernel finished the computation, the actor assures to emit a predefined number of data-tokens on every output port. Each of the ports is connected to another with a channel that acts as a transmitter and buffer of data tokens between actors. The number of tokens consumed and produced by the actors is called consumption and production rate, respectively. The number of tokens that are present on a channel at the start is called initial marking. In the figure, channels are drawn as arrows annotated with the production and consumption rate and initial marking. After a certain number of firings c_i of each actor, the SDF arrives at a state with the same number of tokens on each channel as at SDF startup, called a graph iteration. For the graph in the example, the blue actor will fire six times, while orange and green fire two times. A graph will usually be executed a defined number of full iterations—often only one iteration—before destruction. The total time it takes an execution environment to process all desired graph iterations is called the makespan. It is used as a primary performance indicator for an execution environment running a specific graph.

2.4 Runtime Environment

To be able to address the mentioned hotspot and build a graph interpreter that combines the methodology of both, the task graph and the process network interpretation, an RTE was developed that uses a hybrid graph interpreter to exceed performance of other RTEs. The basic idea of the RTE in contrast to traditional multithreading is that the threads are isolated from each other in their memory view, much like processes in a modern operating system (OS). The RTE then defines a set of communication technics to allow inter-thread data transfer. Structuring the communication in such a way allows the RTE to place threads into physically separated memories without the need to synchronize them continuously. Doing so requires significant expenses in additional hardware and also disturbs smooth program execution, affecting system performance. Additionally, it may pose a safety or security risk to do so. By only sending the data that is meant for the neighboring threads, these risks are avoided, and the data transferring fabric's bandwidth is saved. Simultaneously, the execution efficiency is increased on distributed memory systems, which only have to implement the RTE communication primitives. No drawback is created for execution on shared memory systems since the communication primitives can easily

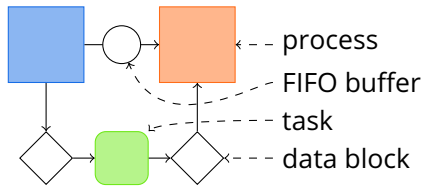


Figure 2.2: RTE example application utilizing all four building blocks. (“task” and “process” are the same to the RTE and only differentiated here to emphasize the flexibility of the RTE.)

be implemented using shared memory locations and still increasing protection against errors based on memory access race conditions.

In contrast to many other RTEs that only define one type for each processing and data to describe a single MoC, an application in the hereby proposed RTE is composed of a network of objects from a set of four types. Fig. 2.2 shows and RTE application utilizing all four objects. The task can best be compared to a traditional thread. Depending on the SDF representation strategy, this can also be called a process. It is instantiated to execute a kernel (e.g. a piece of executable binary). In contrast to the traditional thread, a task is also given a set of input/output (IO) objects that define the initial communication links to other tasks or static data storage. The IO objects can be of two kinds. First is a channel that connects two tasks with a unidirectional FIFO style data pipeline. Accessing the pipeline may block the execution of the task if space or available data in the channel does not suffice the intended operation. The channel allows constant data transfer between simultaneous running tasks. The second option for data access is the data block. A data block is a fixed size RTE managed buffer. It can be assigned to a task in either write or a read mode. Access to it is similar to a channel but without the possibility of blocking. The RTE assures the availability of all data blocks at the task start. A data block is an immutable data structure, meaning that it can be written exactly once (e.g. by one task), after which it can only be read, but multiple times. With this attribute, a block intrinsically works as dependency enforcement between two tasks. Although the second task can only start after the block becomes ready, it already can be scheduled and goes to sleep until the data becomes available. With that, the dependency resolve loop some implementations are suffering from can be shortened.

Any task can create objects of any of the four kinds. That allows hierarchical applications that can spawn or change subgraphs upon change of application

demand. It is especially possible to build the scheduling system as an RTE native application itself. It is also possible to build different SDF interpreters that follow different strategies to process an SDF graph. Three interpreters will be discussed in the following.

Summary of RTE features:

- Isolation of tasks with respect to computation and memory
- RTE managed data containers
- Two communication primitives: data container access and pipelines
- Hierarchical creation of sub graphs

2.5 Overhead of Deploying Tasks to a MPSoC

Before presenting the SDF interpreters, the overhead that an RTE brings to the system for managing the four object types is introduced. The overhead is needed to enable the parallel execution of multiple tasks on an underlying platform. The alternative for the application would be a prescheduled SDF — or trivially a sequential execution—, leaving out all overhead. The overhead is classified into six different stages, each dedicated to a specific part of the tasks-deployment sequence. Together with the kernel stage, the lifecycle of a task is composed of seven stages. Each stage defines the amount of computation effort it introduces. Further, it constraints the order of stages and specifies needed requisites. Depending on the management design, parts or whole stages may be spent on different processing elements (PEs) and run in parallel where possible.

1. **Call** The initiator task spends an amount of effort to set up data structures describing the task and call the RTE application programming interface (API). It thereby hands over control to the RTE.
2. **Control** The whole creation phase is supervised in this stage. It includes creating and maintaining RTE internal data structures, collecting additional information through the stages “place” and “IOaquire”, and deploying the task to the destination PE.
3. **Place** A PE is selected for the new task to be executed on (e.g. destination PE), taking into account the current system state.

4. **IOquire** The IO objects requested for the new task are located and copied to the destination PE. Channels are created and initialized.
5. **Prepare** The destination PE is prepared for the execution of the task's kernel. It may include different things depending on the underlying platform. Usually, it revolves around creating an OS thread and the installation of the allocated IO objects.
6. **Kernel** The kernel function is executed.
7. **Post** Destruction of OS and RTE dependent structures and update of related RTE specific structures like marking blocks as ready and sending "task finish" notification.

Although differentiation of the scheduling overhead is possible in this granularity, in literature, a monolithic centric unit usually controls the scheduling. Such a centralized scheduler is responsible for processing stages 2–4. The other stages obviously cannot be moved from their natural place, which is the initiator PE for stage one and the destination PE for stages 5–7. The centralization of the expensive stages 2–4 in a single unit quickly becomes a bottleneck in a system with sufficiently many PEs or when tasks are sufficiently small compared to the overhead expenses. Because these two factors are crucial for the efficient utilization of the system, they are defined in two ratios. The ratio between overhead and payload effort is described as

$$r_{\text{over}} = \frac{e_{\text{overhead,avg}}}{e_{\text{payload,avg}}}$$

with $e_{\text{overhead,avg}}$ and $e_{\text{payload,avg}}$ being the expected processing effort in overhead stages and the kernel, respectively. The ratio of managing and working PEs is defined as

$$r_{\text{manager}} = \frac{n_{\text{manager}}}{n_{\text{worker}}}$$

with n_{manager} and n_{worker} being the number of PEs assigned to scheduling and to kernel calculation.

Both ratios can be controlled in different ways. To control r_{manager} the number of manager PEs that can be utilized by the RTE must be flexible. As just said this is not possible with a monolithic manager. In Chapter 3 the distribution of the managing subsystem to multiple PEs will be regarded to achieve this

flexibility. For the remainder of this chapter, the focus will be to control r_{over} . Basically, the controlling is possible at two levels in the computation stack. Firstly, the application has a great impact on r_{over} by its decision how to partition the computational effort $e_{payload}$ to a set of SDF actors. While $e_{payload}$ stays constant no matter how and to how many actors the payload effort is distributed. The overhead effort depends on the SDF graph structure (especially the number of actors and number of their firings) $e_{overhead} = f(\text{SDFstructure})$. It is, however, the implementation in the RTE layer that defines $f()$. In the following three different SDF interpreters will be discussed that result in different overhead effort and different parallelization potential for the payload effort.

2.6 Interpretation of SDF Graphs as Task Graphs

The interpretation of an SDF graph as a task graph is the most common found in the literature. It requires an application controller running and emitting a constant stream of tasks at all times. In general, this may be any program that has access to the RTE's API.

The task graph interpretation instantiates a task for each firing of each SDF actor. SDF channels are implemented as a series of RTE data blocks. A production of tokens to a channel is represented by a data block sized to fit the tokens, connected to the corresponding task. Similarly, for the consumption of tokens, the data blocks containing the needed tokens are connected to a task as input object.

The task of the application controller is to create one task for each firing of all actors. It also has to create and assign data buffers for token production and select data buffers for token consumption. In Fig. 2.3 the execution of the cyclic SDF example from Fig. 2.1 is shown. For each actor and each channel, a line of objects is created (e.g. a column). The connections between data and computation indirectly define a dependency network to constrain the tasks' execution order.

pro	con
exploits potential parallelization	high scheduling overhead

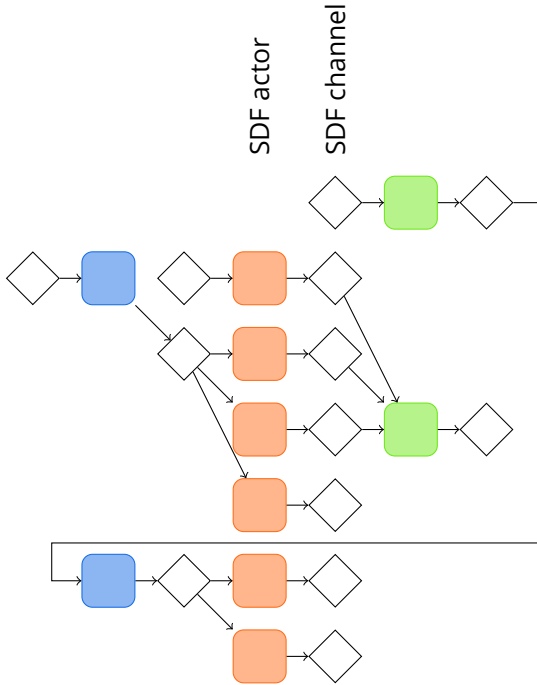


Figure 2.3: Task graph interpretation for the cyclic SDF example (Fig. 2.1). Each SDF actor is represented by a column and owns a set of RTE tasks representing its firings. Likewise the SDF channels are assigned to a column with RTE blocks representing the data flowing through the channel.

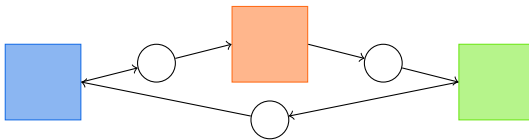


Figure 2.4: Process network interpretation of the cyclic SDF example (Fig. 2.1). Each SDF actor is represented by a RTE process. The connecting RTE channels represent the corresponding SDF channels.

2.7 Interpreting SDF Graphs as Process Networks

Another way of processing an SDF graph —that is also not new to literature— is the process network interpretation. In this interpretation, a single task is spawned for each SDF actor. Like shown in Fig. 2.4 the structure of the application resembles the SDF graph itself. The tasks are connected with RTE channels matching the SDF channels from the model. Each task performs all firings of the actors it represents. It reads the needed token according to the model from available channels. The read request will block the task until the data is ready and transferred to the local memory. When all data is acquired, the task will perform the actor's kernel function. Finally, the task will push the generated data to the output channels, which may again block execution if the space available in a channel is not sufficient. Since the assertion of data availability for local calculation is done through the use of RTE channels, no other management is required while the SDF graph is processed. The whole process minimizes the scheduling overhead to the initial setup of tasks and channels.

pro	con
low scheduling overhead	no intra-actor parallelization

2.8 Hybrid Interpretation

The hybrid graph interpretation allows the coexistence of both SDF actors being interpreted in task and in process mode. A few additional considerations have to be made to allow this hybrid design. The process network is used as a starting point for the hybrid graph. In this graph, the processes belonging to actors that should be interpreted in task mode are replaced by task controllers. A task controller acts as an application controller but only for one actor. Instead of doing the kernel computation by itself, it spawns a series of tasks, one for each firing. It still is connected to neighboring processes over RTE channels.

The task controller collects data blocks from the tasks to produce a sequence of data blocks for each outgoing channel that has to be converted to a continuous data stream for another process. Similarly, an incoming data stream has to be chopped into blocks before passed to the tasks. Both conversions are done on the receiving side of each channel. That means that instead of a data stream, a task controller will only send a series of block handles over the channel. This

way, a homogeneous connection (e.g. controller to controller or process to process) does not have to do any conversion. In case a process receives a handle series, it stitches the data together to a continuous data stream that is then exposed to the computation kernel. The other way around, a task controller will chop a data stream into carefully sized chunks, put them into newly created data blocks, and feed them to the tasks.

Of course, the data stream conversion does introduce a significant amount of overhead that may attenuate the performance speedup achieved by the additional parallelism introduced by the task mode. The selection of actors running in task mode is non-trivial as the resulting performance depends on multiple positive and negative factors. Where additional overhead and data conversion increase the total effort, the parallelization of firings opens the possibility of reducing the makespan despite the increased effort. But the exact impact on the makespan also depends on surrounding actors and graph topological details.

In Fig. 2.5 the execution of the simple SDF example is displayed. The blue actor is selected to be executed in task mode. Therefore it's process from Fig. 2.4 is replaced with a task controller. A line of tasks and data blocks for both input and output to these tasks is created. The controller fills the input blocks with data from the incoming channel while the block handles of the output blocks are fed to the outgoing channel.

pro	con
parallelization is possible	partitioning of graph non-trivial
scheduling overhead reduction possible	

2.9 Graph Topology Considerations

The decision to select an actor for task mode depends on many factors and is not easily decidable until now. In Fig. 2.6 some topological constructs are displayed that might hint for or against a task mode selection. The exaggerated example on the left shows an actor (orange) that may be fired dozens of times for every firing of the blue actor. Additionally, firings of the orange actor do consume more processing time than the neighboring actors. Both facts do emphasize a favoring of task mode for this actor. The multiple simultaneously available firings promise an improvement by parallel execution while the high

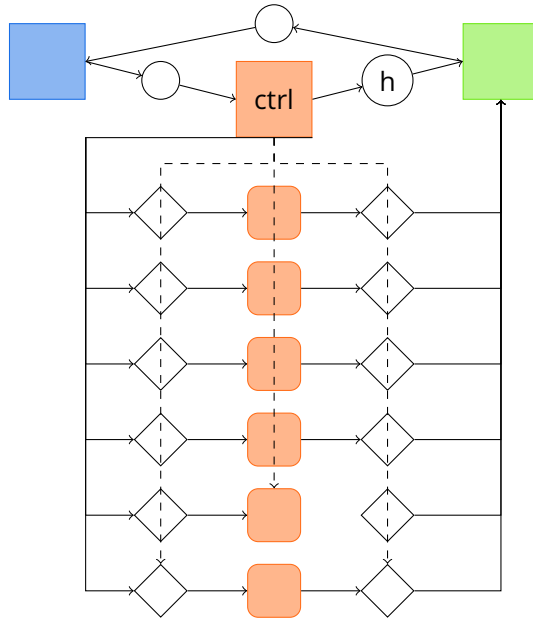


Figure 2.5: Hybrid graph interpretation of the cyclic SDF example (Fig. 2.1). Although all SDF actors are represented by RTE processes, one is making use of a special process controlling the generation of RTE tasks representing individual firings. The channel connecting the controller with the next process only transmits RTE block handles, allowing the following process to stitch data to a continues stream.

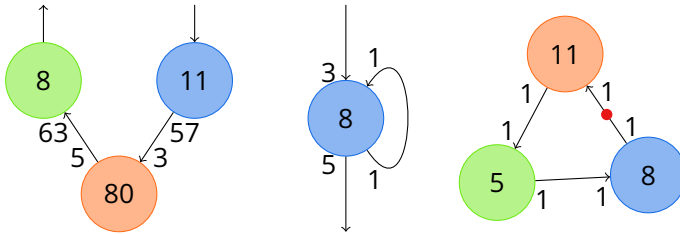


Figure 2.6: Example SDF graph parts that have implications about potential parallelization. The left graph part creates a set of independent red actor firings that favor parallel execution. The middle graph creates a self-dependency of the tasks not allowing any parallelization. Likewise the right graph interlocks all actors in a way that they have to fire strictly sequential.

kernel length ensures that the introduced overhead will not nullify the improvement.

The middle example shows a situation that strongly favors the usage of process mode. By default, an actor in SDF does not have a state that can be transferred between firings. A stateful actor can still be implemented by emitting the state into a channel that loops back to the actor itself. It ensures that the state is present for the next firing. Such a construct implicitly defines that all firings have to be carried out sequentially, which defeats any improvement that could have been achieved in task mode.

The three actors on the right are in a very similar situation. Like the stateful actor, they each depend (indirectly) on the previous firing of themselves. The cyclic channels ensure that each actor's firing is dependent on all previous firings of all actors, including its own. As a result, all firings in the graph have to be executed sequentially. Therefore, all three actors should be run in process mode, if allowed by the RTE multithreaded on the same PE to minimize data transfer and central processing unit (CPU) idle times.

2.10 Theoretic Impact of Hybrid Interpretation

The impact of the different execution strategies on the makespan can be significant. The impact is strongly dependent on the system's and the graph's topology and the RTE implementation, especially the ratio between overhead and

payload computation. Because this ratio is so important it is noteworthy that it can —to a certain extent— be controlled through clustering. As described in [27, 28] clustering results in a smaller task set of bigger tasks and naturally in less scheduling overhead. Seeing that scheduling overhead is the main thing we need to be concerned about in dynamic scheduling, it is assumed that the used graphs are already clustered. The simple SDF example’s execution is simulated based on a simple model to understand the effects of the different execution strategies. In this model, all communication costs are neglected, and task overhead costs are simplified to a single effort of constant time, which is processed strictly sequentially. The number of available PE is infinite to remove any hardware restriction from the simulation. Under these constraints, the example is processed under all three execution strategies.

A sequential execution and a “static” schedule are also included to give an upper and lower boundary on the possible makespan. For these two strategies, all overhead is neglected. The sequential strategy represents the execution on a non-parallel system, where “static” defines the theoretical lower bound for the makespan by placing all firings as early as possible with infinite resources and no overhead assumed.

Fig. 2.7 shows a comparison of makespans with different execution strategies. In the “process network” setup all actors are running in process mode like explained in Section 2.7. The makespan is defined by the sequential execution of the orange actor’s firings. In the next strategy (e.g. “task graph”) all actors are running in task mode, composing the setting from Section 2.6. As explained, each firing is companioned by an overhead effort. All overhead efforts, being executed strictly sequential, delay some firings significantly, leading to a bigger makespan than without the overhead (cf. “static”). Still, in this example, the makespan of “task graph” is smaller than of “process network”.

It is, however, the combination of both systems that achieves the best result. In the execution labeled “hybrid”, only the orange actor runs in task mode because it is the only one that can save more time by parallelization than loose by additional overhead. With the firings of the other actors not even (or only barely) overlapping in time in the “static” schedule, it is evident that running these in task mode will cost more than it will save.

This small example is supposed to show that the hybrid interpretation of an SDF graph actually can have an impact on the execution time. Besides, the hybrid interpretation is a generalization of the execution description, including the other two (task graph and process network) interpretations. And although it cannot compete with the “static” scheduling in terms of pure makespan, it

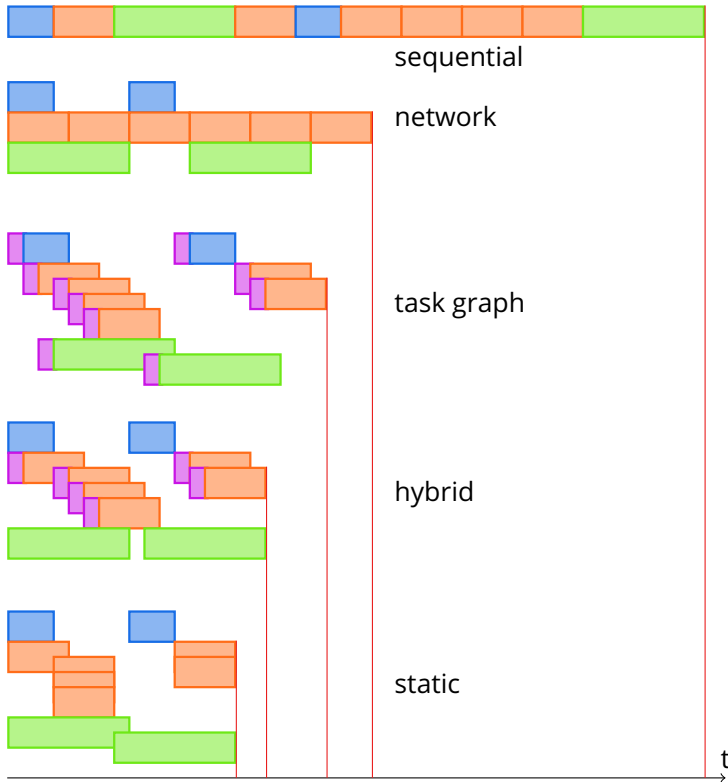


Figure 2.7: Comparison of makespan of different execution strategies for the cyclic SDF example. The firings are colored according to the actor colors in Fig. 2.1. The lavender block represent the overhead effort.

has the advantage that all resources are allocated in task granularity. Where the “static” schedule has to occupy a total of six processing cores for the whole makespan, the dynamic scheduling only occupies the cores when they really need to do work. This finer resource allocation is paid with the overhead of managing the dynamic objects (i.e. tasks and data objects). The application has direct influence over the amount of the overhead through the number of objects it defines. A clustering can reduce the number of dynamic objects to a reasonable level. The number of dynamic objects to manage is multiplied with the amount of overhead the RTE spends for each object.

2.11 Simulating Hybrid Execution

To analyze a small example like the cyclic SDF example, the model used in the previous Section is sufficient. A more sophisticated model is necessary to get a deeper understanding of the coherences of task and process mode. A resource management simulator was built, to model an RTE environment as described in Section 2.4. An overview of the simulation layers is given in Fig. 2.8. Within the simulator, a model is built of two basic object types: resource providers and resource consumers. Objects of these two types interact with each other based on the simulation cycles [scl]. It is not to be confused with clock cycles [cyl] of a processor but rather be thought of as a scheduling time slice.

A resource provider defines a resource with the attributes “capacity” and “parallelism”. The “parallelism” describes how many resource consumers can simultaneously use the resource. It can be thought of as the number of taps on a pipe that consumers can use to share water. The “capacity” defines how much effort a resource can process per cycle. In the water pipe comparison, it describes the amount of water provided by the pipe per time. Depending on the two attributes, a consumer can expect to receive the processing of “capacity” divided by the number of active consumers, where the number of active users is capped by “parallelism” and the processing speed by 1.0. Again in the water pipe comparison: The water running through the pipe is divided by the number of users, which may never exceed the number of taps. Further, each tap can only transport 1 bucket per minute regardless of pipe size.

The “resource consumer” is an active component in the simulation. It can occupy a resource for a given amount of effort. The number of cycles it takes to process the given effort may vary depending on the resource’s attributes, and the number of other consumers also trying to occupy the resource. The

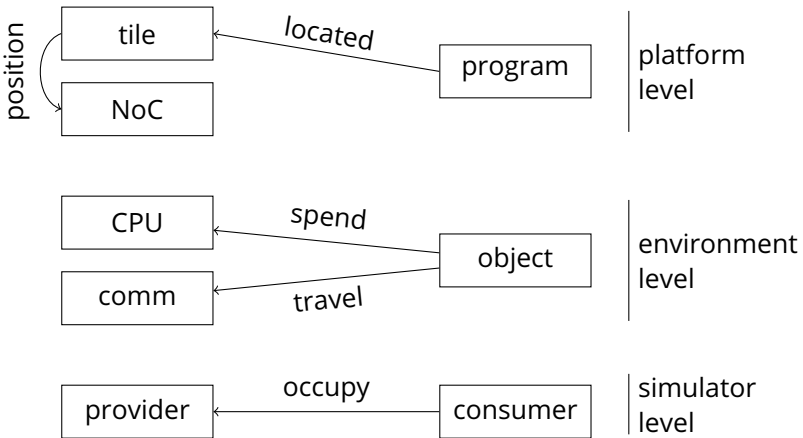


Figure 2.8: Layers of the simulation environment. The upper level is able to describe complex applications to be executed. The middle layer models a hardware platform with the two resources “CPU time” and “data transfer” that can be consumed by “objects”. The lowest level generalizes the system further to only handle abstract resource providers used by abstract consumers.

Table 2.1: Simulation environment parameters controlling execution timing.

parameter	value
number of PEs	12
number of management cores	1
CPU resource parallelism	1
CPU resource capacity	1
NoC resource parallelism	100
NoC resource capacity	100
program stage size	64B

consumer may even have to enter a queue and wait until some other consumers finished before occupying (a share of) the resource.

The RTE is modeled with two resource types. One is a platform tile representing a CPU to run (sub-)programs. Apart from the CPU, a tile also has a position in the network on chip (NoC) of the platform. Typical attributes are a capacity of 1 to 4 to simulate processor with up to four cores and a parallelism, either being equal to the capacity to simulate a simple runtime or infinity to model a sophisticated context switching.

The second resource is the network connecting the tiles. When sending data from one PE to another, this resource is used to model the delay until it reaches the destination. The most straightforward connecting fabric to model is a bus by setting parallelism and capacity to 1, sequencing all transfers. For the multi-processor system on chip (MPSoC) introduced in Section 1.3.1, a NoC is described, that would be very complex to model accurately. However, a NoC resembling connection fabric can be modeled by increasing capacity to allow multiple transfers. Setting an even higher parallelism will cause graceful performance degradation of the NoC when collisions start to happen.

On the consumer side, (sub-)programs are defined to represent every kind of computation effort the RTE is exposed to. They can be thought of as threads that also have a location in the system. The two operations a program can perform within the simulation environment are: (1) Spending CPU time and (2) travel to another location. The program is used to model the RTE tasks as well as the management's different overhead sub-programs.

For the following experiments, a simulation environment is set up with parameters summarized in Tab. 2.1. The platform consists of 11 PEs and a single management core. The CPUs are single-threaded; thus, they do not support

Table 2.2: Consumed processing resource for overhead subprograms in the simulation environment.

operation	simulation cycles [simcycles]
Call	1.50
Control	3.00
Place	1.50
block locate (IOaquire)	1.00
send block (IOsquire)	1.00
receive block (IOaquire)	1.00
Prepare	3.00
Post	0

parallelism. The costs for sending data through the network are calculated from the network path length h and the message size s according to Equation (1.1) with $s_{\text{fit}} = 8 \text{ B}$ and $s_{\text{headdata}} = 4 \text{ B}$ as:

$$d(h, s) = 8 + 2h + \frac{s - 4}{8}$$

The costs of the different overhead operations are summarized in Tab. 2.2. The ratio of values among each other are taken from an own management implementation written in C for an X86 Linux environment.

In its life cycle, a program object can switch locations and occupy CPU resources multiple times. The management subprograms usually have one particular task to do and will travel the system to fulfill it. For example in Fig. 2.9 the journey of subprogram from the "IOaquire" stage is displayed. The subprogram's purpose is to transport a data block to the PE where a task is to be executed soon. Instantiated by the "control" stage, the subprogram will start on an arbitrary PE. Since the current PE is no management PE, the program will first travel to a management unit featuring a block location database. Here, it will occupy the CPU to find the current location of the desired block. It will then travel to the found location and spend CPU time there to initiate a block transfer to the destination PE. After traveling to the destination PE with a transfer of the block size plus its own size, it locks the CPU on the destination PE to model the receiving and inclusion of the block into the local data structure.

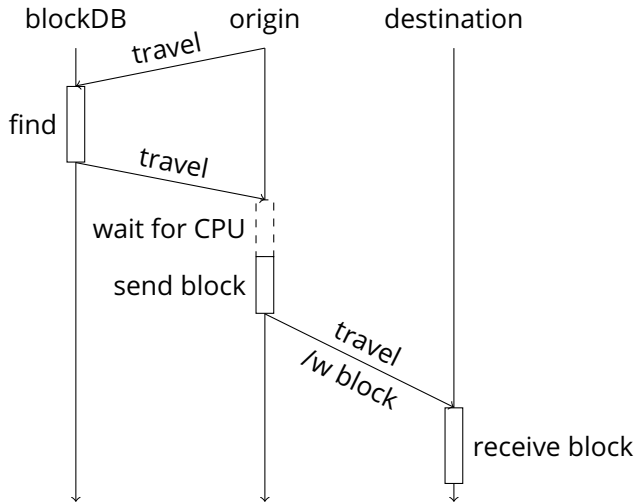


Figure 2.9: Lifecycle of a overhead subprogram running in the simulated RTE fullfilling the purpose of finding and transferring a data block to the destination PE. In its lifetime the subprogram will spend CPU time on multiple PEs. For switching between PEs the NoC has to be traveled by spending time on the NoC resource.

2.12 Pipeline SDF Graph Example

In the cyclic SDF example, both the graph topology and the platform model are synthetic to make explanations easier. With the pipeline SDF example, the platform model is switched to a simulation of a PE network with consideration of network delay and a sophisticated overhead model. The graph also resembles more the typical structure that is known from signal processing applications. It is a pipeline with a topology displayed in Fig. 2.10. Just from examining the graph, it becomes clear that to complete one full iteration the actors must fire, from left to right, 5, 30, and 90 times. The theoretic bounds for the runtime of the sequential and “static schedule” execution can then be derived as 1340 scl (simulation cycles [scl]) and 126 scl, respectively. In Fig. 2.11 the runtimes of one graph iteration is compared between the different setups. Tested are all combinations of process and task mode actors possible.

Neither the task graph configuration (“-”) nor the process network configuration (“1,2,3”) achieves the best results. Configuration “1” puts actor 1 into task mode. This configuration has a slowing down effect compared to “-” because actor 3 is here the limiting factor. For every firing of actor 1, actor 2 can fire six times. Since actor 3 executes sequentially, actor 1 cannot gain anything from potential parallelization. Instead, the additional overhead, especially the communication bridge, hurts performance.

As a contrast, putting actor 2 into task mode (configuration “2”) holds a speed up because executing six actor 2 firings in parallel turns actor 1 into the bottleneck. Combining these two configs yields configuration “1,2”, with both actor 1 and actor 2 in task mode. It intensifies the effect because all firings of each actor can run in parallel, and no communication bridge is needed. In practice, the overhead of spawning the tasks dampens this effect a bit. Adding actor 3 to the task mode set at best does not change the runtime, but mostly prolongs it. The reason for this is the small kernel runtime, which makes it unlikely that the parallelization speedup exceeds the overhead introduced slow down. When going from no task mode actors to just actor 3, the runtime does not change, positive and negative effect canceling each other out. In all other cases, adding actor 3 has a negative effect. In the case that actor 2 is not in task mode, the slowdown is only marginal because actor 2’s firings are finishing sequentially. That allows the overhead to be carried out parallel the firings i.e. outside the critical path. When actor 2 firings finish more clustered —because they run in parallel— the overhead processing manager is overwhelmed and delays the creation of actor

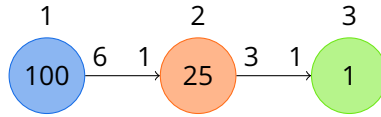


Figure 2.10: The pipeline SDF example graph used to measure impact of different execution strategies in an overseable scope.

3's firings, resulting in a longer runtime. As a result, running actor 1 and actor 2 in task mode yields the best result. Actor 3 is best kept in process mode for mentioned reasons, even if that means to have a communication bridge between actor 2 and actor 3.

Similar to cyclic SDF example, the dynamic scheduling strategy is unable to beat static scheduling in terms of makespan. However, the resource usage is much lower in any case, even the rather slow configurations like, "1,3". The resource consumption is the integral of the parallel processor usage over time. For the the "sequential" execution this is just the sum of all firings. The hybrid configuration have to add the usage of a dedicated management core. Naturally this usage increases with the number of tasks to be managed, so 90 tasks if actor 3 is in task mode and 30 for actor 2. The static schedule occupies 15 cores for the whole makespan of 256 scl resulting in a resource consumption of just over 4000 PE simulation cycles. The 15 cores do not allow a complete parallelization of all firings, thus the longer makespan compared to the theoretical minumum, but occupying 90 cores for a slightly shorter makespan would skyrocket resource consumption out of control.

2.13 Random SDF Graphs

The hybrid strategy is tested with a range of randomly generated graphs to generalize the previous section's findings. The generation is done using the turbine SDF graph generator [11]. Produced graphs have an actor count of 2 to 37, a maximum degree (i.e. edge count) of 6, and a kernel runtime between 1 and 20 simulation cycles. A little more than seven hundred graphs are simulated using the same environment for the pipeline SDF example graph. Apart from the baseline configuration with all actors in process mode, all possible configurations for up to two actors running in task mode are simulated. The most performant configuration is set into relation to the baseline configuration, to

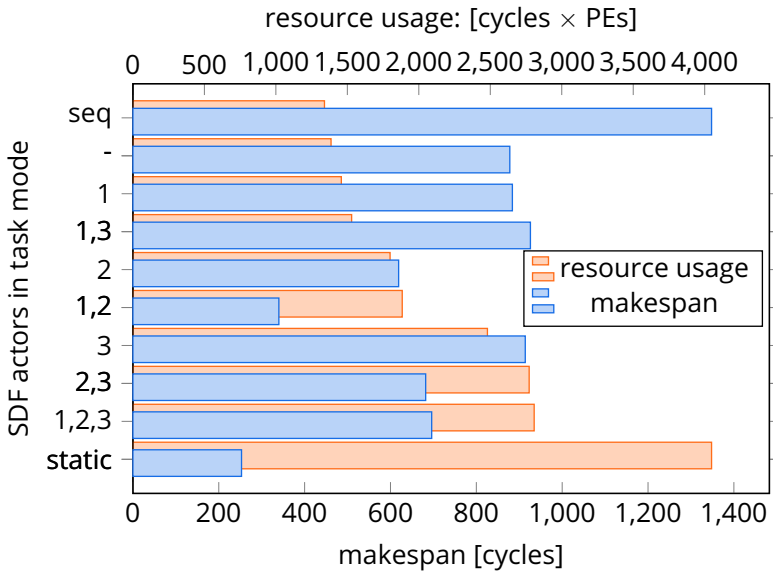


Figure 2.11: Makespan comparison for the pipeline SDF example of Fig. 2.10 with all possible hybrid configurations, which include the pure task and pure process strategy. For comparison the theoretical makespans of an all sequential and an resource unlimited ASAP strategy are shown.

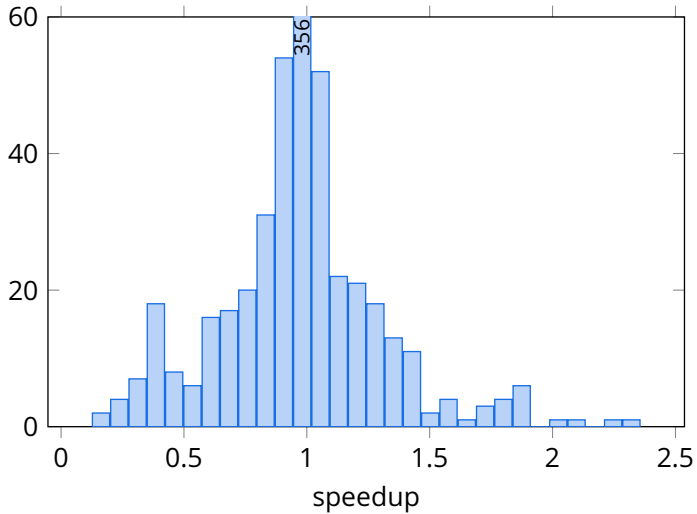


Figure 2.12: Histogram of maximum speedup with two actors in task mode of 700 randomly generated graphs. All configurations featuring at most two task mode actors are simulated to select the best configuration for each graph.

determine the maximal achievable speedup for a graph. In Fig. 2.12 the achievable setup for the 700 graphs is displayed in a histogram. For most of the used graphs, neither a speedup nor a slowdown can be observed. But for a small group of graphs, significant speedups do occur. A speedup of 2x is already considered to be significant because it is directly coupled to the needed (hardware) resources. This means a speedup effectively allows an envisioned chip to be half the size or run at half of the clock frequency, which in the end cuts cost of production and operation in half. The requirement for this to happen is a graph-topological condition that represents a bottleneck in the graphs execution flow, as explained in Section 2.9. The probability of one of these situations to happen is dependent on the parameters used to generate the graphs and the amount of overhead imposed by the RTE implementation. While using a hybrid execution strategy does not hold any benefit in the vast majority of cases, it can bring a significant performance increase when specific situations are present. Because there is no method of detecting these situations, all possible config-

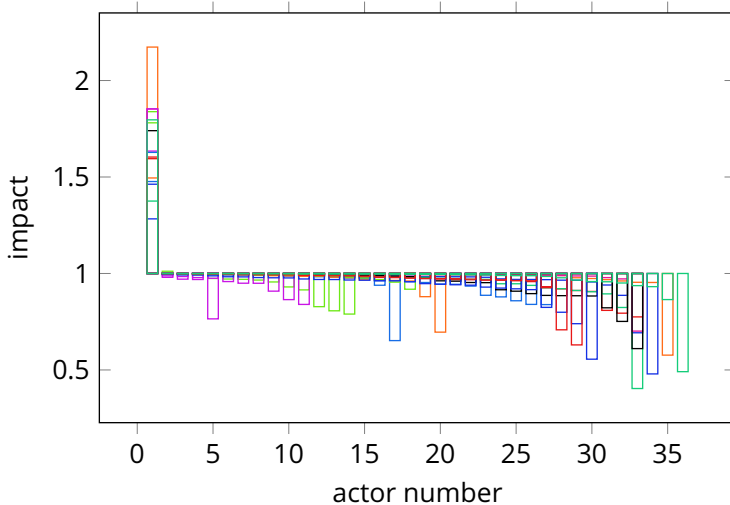


Figure 2.13: Impact an actor has to the speedup when put into task mode. Average speedup of all configurations extended by a task is displayed. The actors are —w.l.o.g.— ordered by their impact to show that usually only one actor is worth running in task mode.

urations have to be tested. Since the total number of different configurations for one graph of size n equals $n_{\text{config}} = 2^n$ it is not possible to simulate every possible configuration.

It seems that performance-limiting bottlenecks are very local and can be mitigated with only two task mode actors. The 16 graphs with the highest speedup are re-simulated with up to four actors in task mode to support this hypothesis. Of those 16 graphs, only one saw a speedup of more than 2%, compared to the 2-task-mode-actor version. In Fig. 2.13, the impact to the speedup for each actor of those 16 graphs are shown. The impact of an actor “a” is the average speedup that is observed by configuration “B” over “A” for all configuration pairs “(A,B)” with “B” being constructed by added actor “a” to the task mode set of “A”. Because the actors numbering in a randomly generated graph is arbitrary, w.l.o.g. the actors can be ordered by the impact.

It shows that all graphs have one particular actor that triggers the speedup where most of the actors do not impact the performance. A reason may be that even if an actor is suitable for task mode, it does not create an impact on

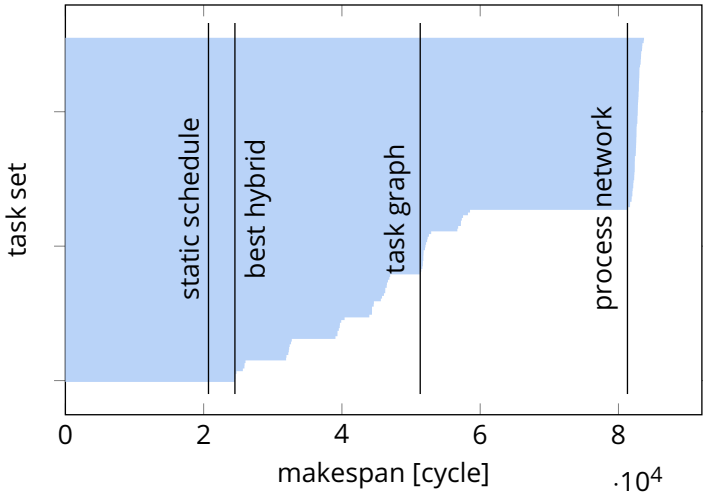


Figure 2.14: Runtime comparison of all 128 possible hybrid configurations for the LTE resembling distributed application graph. The configuration with actors “1,4” and “0,2,3,5” have the best and worst runtimes, respectively. Both full process mode (“no tasks”) and full task mode (“all tasks”) stay within the average.

the makespan if it is not in the critical path. Almost every graph features a few “wrong” actors that can drop performance significantly when selected for task mode. While this shows that these performance increasing situations are happening in random graphs, it does leave the open question about the applicability to real-world problems.

2.14 LTE-like SDF Graph

To apply the findings on the targeted real life example of a mobile signal processing pipeline, the SDF graph introduced in Section 1.4 was simulated using the hybrid scheduling as well as a static schedule for comparison. With the graph consisting of six actors there are $2^6 = 64$ different configuration of selecting actors for task mode. All 64 configurations have been simulated and the resulting makespans are show in Fig. 2.14 sorted by makespan. Marked in the figure are the makespans of relevant configurations. The pure process

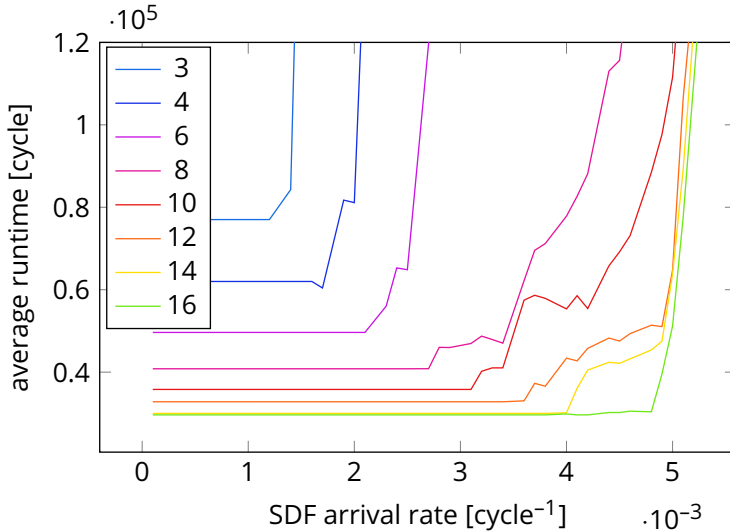


Figure 2.15: Average makespan of LTE resembling SDF graph with variable number of PEs (colors) and graph arrival rate. The dramatic increase of average runtime marks the rate at which the system reaches saturation.

network configuration is in this setup close to the worst possible configuration. While a lot faster, the task graph configuration takes still around twice as long to complete as the best hybrid configuration. The best hybrid configuration is, like suspected in Section 2.13 one with just two actors in the task set, i.e. 1 and 4. With a makespan of $m_{\text{hybrid}} = 24.5 \text{ kscl}$ the hybrid configuration is only around 20% slower than the static schedule with $m_{\text{static}} = 20.7 \text{ kscl}$. However, the static schedule occupies 16 processing cores for the whole make span resulting in a consumption of $16m_{\text{static}} = 331 \text{ kscl}$, while the hybrid schedule consumes not more than an accumulated amount of 133 kscl, a merely 40% of the resources.

While the makespan is an excellent measure to quantify the effectiveness of an SDF graph execution, it only correlates to a certain degree to a more crucial performance indicator valuing the whole system. That is the rate at which a platform can accept newly spawned graphs without saturating its resources. The saturation of resources means that instances of the graph enter the system faster than they leave. From the queueing theory, this is known as a system load

ρ being greater than 1. With ρ being calculated from the task arrival rate μ and task service rate λ as

$$\rho = \frac{\mu}{\lambda}$$

the saturation is clearly dependent on the makespan. It is directly related to the service rate, which would be $\lambda = 1/D$ for a simple first in first out (FIFO) queue. The parallel structure of the system, however, makes λ much harder to determine. Assuming a sequential execution of each graph instance, the service rate is simply the parallelized execution $\lambda = n_{PE}/D_{\text{sequential}}$. This is a good solution when regarding only the graph processing rate because it does not contain any overhead calculation. However, it is not feasible to do sequential TTI processing, because of deadlines that can only be kept if parts are computed in parallel.

When putting the previously used mobile signal processing graph into a simulation scenario with an increasing graph arrival rate the processing capacity can be obtained. In Fig. 2.15, the average individual graph makespan in a test series with increasing arrival rate for different platform sizes are plotted. The drastic increase of makespan marks the saturation frequency. With an increasing number of PEs available, this point moves to higher frequencies showing a higher processing capacity. The second feature seen in this graph is a decrease of the average makespan with an increasing number of PEs. With more PEs available in the system, each graph can run more firings in parallel. This makespan decrease stays in constant relation to the saturation point for smaller PE numbers. The saturation point does not move beyond $5 \times 10^{-3} \text{ scl}^{-1}$ for higher PE numbers, although the average makespan shrinks further. For a PE number higher than 12, the processing power of the PEs is no longer the limiting factor, but the overhead calculating RTE management unit. Unable to create and release new tasks fast enough, the management is now the bottleneck, limiting the platform's size that efficiently can be serviced with the RTE. The size of the system that can be serviced depends on the executed application or, more precisely, the ratio of the sum of all firings—the computation payload—and the overhead work introduced to the RTE management unit by the graph topology—the computation overhead—.

2.15 Key Learnings

The commonly used static scheduling of SDF graphs bears the problem of an inflexibility when deploying applications to hardware platforms. The effective resource consumption is much higher than the theoretically needed because each used processing core has to be occupied for the whole makespan instead of only the time they are needed. The two classic dynamic SDF execution strategies improve on resource consumption, but have the disadvantage to increase the makespan significantly. The task graph strategy dynamically distributes the processing demands to available resources. It exploits the potential parallelization very well at the cost of high and constant overhead effort. In contrast, the process network is limited in its ability to exploit graph given parallelism, and needs as many parallel system threads as there are actors in a graph, but features a small management overhead footprint. For random generated SDF graphs as well as the mobile signal processing distributed application, both strategies only achieve mediocre performance with makespan of 2x to 5x higher than the static schedule. Despite the makespan increase the effectively used resources are roughly cut in half using these dynamic scheduling techniques, since it is much easier to run multiple of those applications in parallel even on relatively small systems.

Proposed is a hybrid execution strategy, that selectively applies the parallel execution of task graphs and the low overhead data pipelines to specific actors to decrease makespan without losing dynamic resource allocation. Simulations show that carefully selecting a single actor for the task mode group yields the best performance in many cases. Similarly, a set of two task mode actors has been identified as the optimal configuration for the mobile signal processing application. This configuration only experiences a makespan increase of 20% over the static schedule but reduces resource consumption by 50%.

Although using only two actors in task mode significantly decreases overhead compared to all actors in task mode, an SDF scheduling RTE can only efficiently manage a small set of PEs without letting the management become the performance bottleneck. The RTE's task management must be parallelized to handle more tasks, serve bigger platforms, and overcome this problem. Balancing "makespan decrease" and "overhead increase" optimizes the makespan of a single graph instance. The next logical step is to balance management and worker resources to optimize system utilization.

3 Distribution of Management

The management of processing resources is a central element in the design of an runtime environment (RTE). Also, the overall system performance often depends on the efficiency and performance of the management subsystem. Already small sets of processing elements (PEs) may outperform classic monolithic task management. In this case, the working PEs perform the given tasks faster than the management can create new ones making the management the system's performance bottleneck. The simplest —and most common— implementations process the whole management effort sequentially in a single-threaded program. Concentrating all sub-problems of the management into a single program —hence the name monolithic— is in contradiction to the targeted platform's distributed character, application, and execution strategy. In this chapter, the management effort needed in an RTE to run distributed applications will be analyzed with regards to distributing and parallelizing it. The goal is to manage bigger systems (i.e. with more PEs) without having the management being the performance bottleneck.

3.1 Addressed Performance Hotspot

The task of an RTE on a distributed memory system is to map the application's tasks to the underlying platform. It assures the execution of all tasks, thus the progress in the application. The application defines the amount and computational effort of the tasks. This effort is supposed to be distributed to the platform by the RTE. Besides, the RTE faces a second set of effort originating from the management overhead that accompanies each task. The total effort an RTE

faces when processing an application therefore is the sum of both: overhead effort and payload effort.

Since the overhead effort is best to be done on a dedicated set of PEs, the system can be regarded as composed of two sub systems. The worker subsystem handles the payload effort while the manager subsystem handles the overhead effort. From the set of all PEs in the system each is assigned to either subsystem creating a partitioning. In order to fully utilize the system the ratio between payload and overhead effort r_{over} has to match the ratio of available processing power for each effort type. The latter is equal—for the sake of simplicity assuming the set of PEs is homogeneous—to the ratio of number of PEs in each subsystem r_{manager} . If the two ratios are not matched, one of the subsystems may be overloaded causing idle times in the respective other, diminishing overall system performance in the end. Assuming a constant and repetitive—thus homogeneous—stream of tasks, the overall system performance is measured as the task arrival rate at which the system remains stable. A system is considered stable when task arrival and task finishing rate match. When the system is unable to finish tasks the same rate as they arrive, tasks will pile up in the system and (slowly) overwhelm it, making the system unstable. Therefore the maximum arrival rate that keeps the system stable defines the “execution capacity”, the maximum rate at which the system can process tasks.

The commonly used monolithic scheduling system limits the management subsystem to a single PE, which fixes $r_{\text{manager}} = 1/n_{\text{PE}}$ restraining the flexibility of the system to equalize the afore mentioned ratios to optimize system performance. A fixed r_{manager} forces a similar value for r_{over} , limiting the partitioning of the application into tasks. Therefore, in this chapter, the distribution of management overhead to multiple parallel working PEs is examined for practicability. The intended goal is to allow a bigger r_{over} , which means relatively more overhead per payload. An application is then able to be partitioned into smaller tasks which may expose more potential parallelism. With more potential parallelism the RTE is able to utilize more parallel worker PEs, which in turn will result in higher system performance. Secondly, with more management PEs also more worker PEs can be deployed without lowering r_{manager} .

3.2 State of the Art

There are a lot of works covering static data flow (SDF) scheduling with all possible attribute variations. The most important attribute is dynamic vs. static

scheduling. Most work on SDF scheduling on multiprocessor systems uses static scheduling. There are, however, exceptions using dynamic SDF scheduling. Pelcat et. al., for example, implemented an Long Term Evolution (LTE) uplink using live scheduling on a 6-core multi processor system on chip (MPSoC) with shared memory [66]. They state that static scheduling is not feasible because of the vast graph variety an LTE modem computation can have on a transmission time interval (TTI) basis. Furthermore, although everything is properly optimized, they experience their single scheduling core to be at 95% utilization, which may become a problem for scaling. Another work that addresses the need for scheduling overhead minimization is [12]. In terms of memory architecture, shared memory is commonly used, only excepted by a few works like [20] and [18] using distributed memory to optimize SDF execution. Among different strategies to optimize SDF performance, clustering is found more often, like in [56].

3.3 Revising Deployment Overhead

The seven stages of an RTE task are explained in Section 2.5. It is stated that all stages but 2–4 must be done on their natural location (e.g. setting up the environment for kernel execution must be done on the destination PE). Stages 2–4 however, are the classical decision-making stages, describing where to put a task and where to get the required data blocks. These can be relocated, and more importantly, parallelized. In Fig. 3.1 the relation of these stages and their dependencies are summarized.

Stage 2 (“Control”) only requires a small piece of memory for its internal data structure and some communication with the stages 3 & 4. It only holds data on the current task and is independent of other tasks. Therefore, it may be executed on any available processor and for different tasks in parallel. Stage 3 (“Place”) is supposed to decide which PE the current task should be placed. To do so, it relies on a database of all working PEs’ state information (e.g. the mapping of current active tasks to PEs). The effort to synchronize this database grows quadratically [86] with the number of locations to synchronize. Therefore, it is necessary to keep the database centralized or at least distributed over only a few locations. Similarly, stage 4 (“IOaquire”), among other things, also queries a database to find data block locations. Like stage 3, it is because of the database synchronization costs that it should be kept centralized.

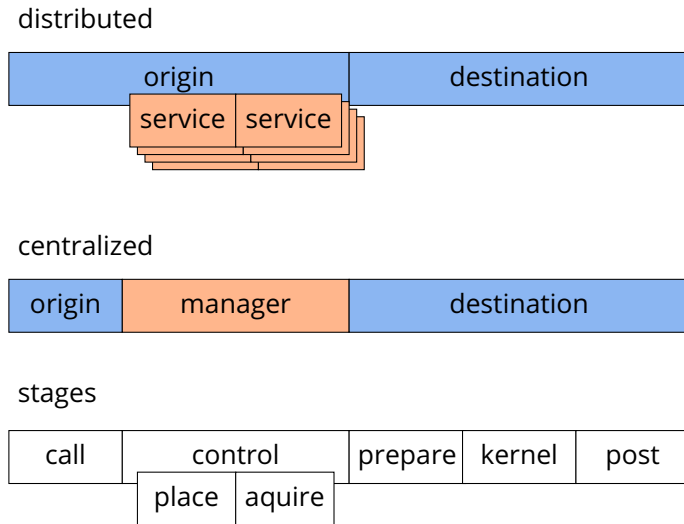


Figure 3.1: Distribution of processing effort of different task stages.

3.4 Distribution of Overhead

Calculating all overhead in a single centralized manager allows the manager to become the system's performance bottleneck. Luckily the task management can cleanly be separated into different stages of which stages 2–4 are to be calculated in a management unit. The stage subprograms have little data dependencies between each other, and only stages 3 & 4 depend on the subprograms of the same stage and other tasks. Generally, this means different stages can be calculated in different PEs, which already turns the management into a pipeline structure. Three dedicated PEs, one for each stage, connected to a pipeline, handle three tasks at once, each in a different stage.

The distribution of management work can be parallelized further by modularizing the management program architecture. In the full modular approach, the "Control" stage will be directly executed on the origin PE. Because it does not have any dependencies on other tasks or any central database, the subprogram's location for this stage is irrelevant. In contrast, stages 3 & 4 do depend on a central database. Thus, their subprograms have to be instantiated as a service in a dedicated PE. These services are queried from stage 2 subprogram

instances in the classic server-client fashion. The client sends a request message, the server answers with a message containing the result.

This architecture softens the border between the management subsystem and the worker subsystem because stage 2 effort is processed on worker PEs. On the other hand, the direct result is that stage 2 is truly and automatically scalable. All stage-2-effort created by task creations is processed directly on the PEs that is causing the effort in the first place. The only remaining problems for a scalable management system are the centralized placing and block-finding services. They cannot be decentralized the same ways it is done for stage 2 because of the data synchronization overhead that does not scale very well. But the synchronization effort can be kept in reasonable boundaries by duplicating the services only a couple of times. Together with a load balancer that distributes the requests to the different service instances, the scalability of the services can be obtained at reasonable costs.

The final distributed management system is displayed in Fig. 3.1. Stage 2 sub-programs, which function as the shell for task creation, can pop up at any worker PE caused by any task. The task shell will eventually turn to both services with a request. Based on the results, it will then issue commands to other worker-PEs to transfer needed data and instantiate the task kernel at the chosen destination PE.

3.5 Impact of Management Distribution to Resource Utilization

The goal of an RTE managing a distributed memory system running a set of distributed applications is to maximize time ratio each PE is working on payload effort to the time it works on overhead effort or is idle. It assures that the throughput of individual SDF instances is maximized. The key performance indicator for the system therefore is again the execution capacity.

To evaluate the distributed management system's scalability, the simulation environment introduced in Section 2.11 is used. This time the number of PEs is increased to show that the system can deal with much higher arrival rates. The application is the same LTE resembling SDF-graph that was used in Section 2.14. By carefully selecting the size of chunks processed in each task a overhead to payload ratio of $r_{over} = 1.0$ has been configured for the following tests. In Fig. 3.2, the execution capacity is plotted as a function of the system size in PEs and the number of service PEs. As a reference, the monolithic scheduling

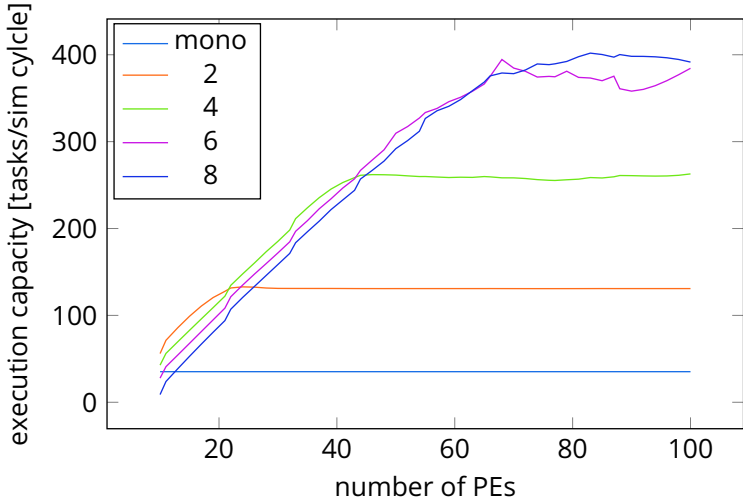


Figure 3.2: Computation capacity (i.e. task rate at which saturation is reached) of distributed overhead management with varying service host numbers and system sizes. The capacity is linear dependent on the number of PEs till the point when the service hosts become the bottleneck, flattening the capacity plot. The execution capacity describes the number of tasks the system can execute each sim cycle. Tasks are created at this rate by spawning the mobile signal processing SDF graph (introduced in Section 1.4) at a appropriate rate. Each simulation is continued long enough to determine if the system can stabilize. The simulations are repeated with varying task rates till the maximum is found the system can tolerate. The service host numbers (color coded) describes the number of processing cores dedicated to the management services of “task placing” and “block localizing”.

approach, known from Section 2.5, is shown. For every management configuration, a constant maximum execution capacity is reached at a particular system size. Above this size, the execution is limited by the management system's performance, thus rendering the system size irrelevant. Below the point, the system performance is determined by the worker subsystem. The system's capacity for processing SDF graphs here is dependent on the number of working PEs. Two conclusions can be drawn, which are reflected in the graph as well. The slope of each configuration below the critical size is equal. That means the capacity increases linearly with the number of PEs in the system. Secondly, the management system occupies a different number of PE in each configuration, thus leaving a different number of PEs for the worker subsystem. With a different number of worker PEs for the same system size, the execution capacity differs as well, which is reflected in the graphs' offset.

Smaller management systems are favored in small setups because they produce a higher capacity, resulting in higher efficiency. In Fig. 3.3 the efficiency of a system is plotted as the execution capacity per PE. The performance loss caused by an oversized management system can be seen here more clearly. With a system size of 10 PEs the efficiency for a 2-PE management is 4x higher than an 8-PE setup. Since both setups are operating in the worker-limited range, the crucial attribute is the worker subsystem's size. In this case, the ratio of 8 to 2 worker PEs under the 2-PE and 8-PE management systems match the efficiency ratio.

Although a small management system is desirable because it does not shrink the worker subsystem as much as bigger configurations, it features a lower performance for larger systems. In Fig. 3.2 is shown, that increasing the management system, increases the number of worker PEs the system can utilize before becoming the bottleneck itself. At the point where both workers and management are balanced in their capacity, the efficiency (cf. Fig. 3.3) is maximized. Further, two things can be observed in this matter. The performance increase gained by a bigger management system declines with a growing management size. Switching from a 4-PE management to a 6-PE management holds an increase of $150 \text{ task scl}^{-1}$. In contrast, increasing further to 8 PEs makes merely little difference. At this point, the network traffic and the synchronization of databases hinders further performance growth.

The results of Fig. 3.3 and Fig. 3.2 are based on the overhead to payload ratio $r_{\text{over}} = 1.0$. Because r_{over} is application dependent, it is necessary to regard it as a variable. The kernel runtime of all SDF actors were artificially scaled to construct a set of different values r_{over} without distorting effects caused by graph topology

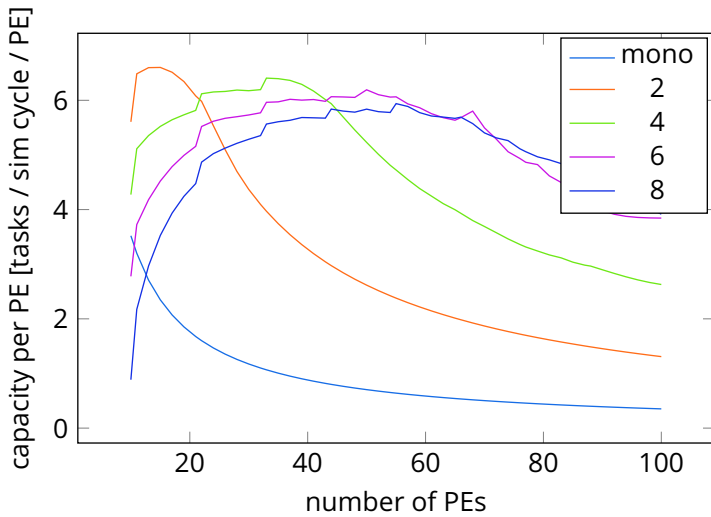


Figure 3.3: The computation efficiency is the computing capacity (as shown in Fig. 3.2) per processing core. A higher number of dedicated management service cores decreases the efficiency because fewer cores are available actually do application processing. Too few service hosts result in the management being the bottleneck, unable to fully utilize the worker cores. The efficiency peaks at the optimal ratio of worker and service cores.

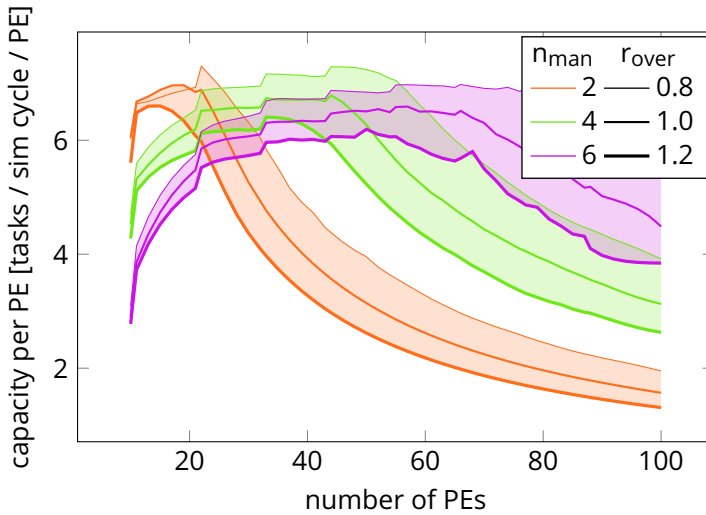


Figure 3.4: Computation efficiency depending on overhead to payload ratio (r_{over}) and manager subsystem size (n_{manager}). Since r_{over} is application dependent the optimal management to worker ratio has to be determined dynamically at runtime.

changes. In Fig. 3.4, the effect of raising or lowering r_{over} is shown. As might have been expected, increasing the relative overhead (by shrinking kernel runtimes) decreases the system's efficiency and thus execution capacity. Decreasing r_{over} , however, increases execution capacity as expected, because fewer overhead has to be processed for the same amount of payload. Also, the peak efficiency is shifted towards larger systems because the same management configuration is able to serve more worker PEs before becoming the performance bottleneck.

3.6 Reconfigurability

The efficiency of the system depends on the well configured ratio between: (1) The size of the worker subsystem. (2) The size of the management subsystem. (3) The overhead effort rate (closely coupled to the task arrival rate). (4) The payload effort rate (closely coupled to the task arrival rate and average task kernel size). When any of those value changes over time, functionality must be provided to change the others accordingly. Both effort rates are always subject to changes when applications enter or leave the system. But even only a change inside an application, like a change in an mobile channel or user can cause drastic changes in both effort rates. Asuming a fixed amount of available PEs, it may be nessecary to migrate PEs from one subsystem to another to match r_{manager} to a changed r_{over} . The reconfigurability can be taken one step further. In a more flexible system, the RTE may be able to allocate/deallocate PEs to not only react to effort ratio changes r_{over} but also changes in effort amount. In any case, to assure a efficiently working system, the ratio matching must be kept.

3.7 Key Lernings

Traditional RTEs for distributed memory systems utilize a centralized managing unit responsible for managing and assigning resources to tasks and processes. Many systems experience a performance limitation by the management unit itself when the working PEs are working faster than the manager. In this case, the ratio between workers and managers (i.e. exactly one manager for the centralized approach) is bigger than the ratio between average task payload and overhead effort. Small tasks are desirable because they allow higher parallelization but decrease the payload-to-overhead ratio. The worker-to-manager ratio must also be lowered by decreasing the number of workers or increasing the number of managers. Because lowering the number of workers does not seem

to be helping the scalability, increasing the number of managers is the only possible solution. Because the management application is not trivially duplicable, it has to be converted into a distributed application.

A task's lifecycle is divided into seven stages, with one being the payload phase and the rest being overhead. Three of those stages are traditionally computed in a centralized manager. Distributing these stages to different PEs already achieves a certain parallelization. Each stage can be duplicated to multiple PEs to increase the parallelization further.

When designing the full system, the crucial point is to match the worker-to-management ratio to the application and RTE dependent payload-to-overhead ratio to maximize the system's efficiency. When not properly matched, either the worker or the management subsystem is not fully utilized, which leads to wasted processing resources. Besides, with the growing system size, the execution capacity growth declines because of increasing overhead caused by data transfer and database synchronization.

All the relations described here are highly dependent on the systems and also the applications parameter. The used simulation environment was configured to resemble the behavior of a Tomahawk MPSoC architecture running a home-brewed RTE for the overhead estimation. An LTE resembling SDF graph was used to model the distributed application with an average kernel runtimes to produce a payload-to-overhead ratio of 1.0. With these settings, a system of 75 PEs achieves the highest execution capacity. However, this value changes when changing the application —or the RTE— to another payload-to-overhead ratio. A smaller ratio of 0.8 lowers the maximal system size to 55, where, with a ratio of 1.2, systems of around 120 PEs are possible.

When dealing with applications that behave dynamic in terms of task frequency and computation complexity, a system must employ a distributed and dynamically sized management subsystem to achieve maximum utilization and thus efficiency.

3 *Distribution of Management*

4 Sliced FIFO Hardware

The two previous chapters' software considerations build upon some assumptions about features of the underlying hardware to assure certain performance on some tasks. An essential function, a distributed memory hardware platform needs to address is the communication mechanisms that allow the tiles to move data between their local memories. In the best case, the data transfer is handled entirely by the hardware relieving the software of overhead spend for communications. The performance indicators for performant communication mechanics are the bandwidth and delay the transmission of data experiences. Although it can be factored into the two attributes, sometimes it is also interesting to explicitly regard the overhead the software sees for initiating a transmission. In this chapter, first in first out (FIFO) type connection will be regarded as often found in process network applications and shown in Fig. 4.1. The connected processes are considered to be on different processing elements (PEs) and connected by the platforms network on chip (NoC) fabric. A hardware solution is presented that exploits the PE's memory system for bandwidth exhaustion, and which is low overhead, and transparent for the using masters.

4.1 Addressed Performance Hotspot

The implementation of a FIFO channel for two communicating processes can be done in various ways, that all may achieve the desired functionality, but with different and differently severe drawbacks. The four performance metrics that are used to rate a solution are *overhead*, *delay*, *bandwidth*, and *chip area*. The *overhead* determines the effort a process has to spend on its main computation unit

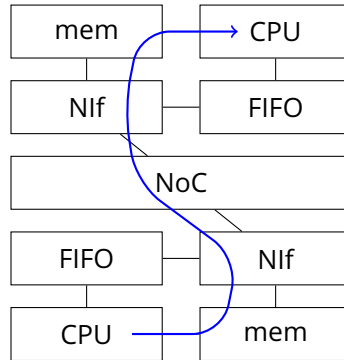


Figure 4.1: Integration of the proposed FIFO controller into an MPSoC. The complete CPU to CPU pipeline (blue arrow) is constructed using two local FIFO instances. The FIFO controller assures efficient parallel access for CPU and Nif to the local memory.

to initiate some data transfer. Since the overhead restrains the process from doing the application functionality, it may lower the application's performance. Therefore, the overhead will be high on a pure software solution, basically because the FIFO functionality is done sequentially to the application. Similarly, implementing a FIFO protocol on top of a simple remote direct memory access (RDMA) unit may pose a significant amount of overhead. The *delay* is a performance indicator measuring how long a transmission takes from the moment the sending process returns from its overhead effort until the receiving process can start working with the data. In that sense, the delay is the time that passed during the transfer but in parallel to the application computation. For example, a shared memory software solution does not have any delay because all FIFO functionality is done in the overhead (e.g. sequential to the application). In contrast, transmitting data over a NoC with a hardware RDMA unit classifies as a delay because it is not interfering with the main CPU. The third measure that defines the performance of a FIFO channel is the *bandwidth* it provides. It states how much data the channel can transport per time unit. For digital systems, it makes sense to fix the time to one clock cycle (*cyc*) and normalize the data to one token (*tok*), which describes the memory access width a master can access with a single request. That leaves the maximum bandwidth any master can achieve per memory port of one token per cycle (1 *tpc*). Another consum-

able to be considered is the *chip area* a solution consumes. While a traditional hardware FIFO unit can be operated with close to no overhead and delay, the area consumption of the internal memory is usually massive. Also, its memory has to be dimensioned at design time and is exclusive for this FIFO unit. Since the on-chip memory often poses the most significant *chip area* share, this kind of single-use memories need to be avoided.

Based on these four performance indicators (i.e. overhead, delay, bandwidth, and chip area), a hardware unit will be proposed to outperform state-of-the-art competitors. A FIFO channel reaching from a PE through the NoC to another PE like shown in Fig. 4.1, can be partitioned into three stages. The first and third stages handle the data transfer from the master to the FIFO unit, while the second stage handles data transfer from one FIFO unit to another through the NoC. This second stage has a relatively simple structure, and its performance mainly depends on the NoC implementation, and therefore will only be regarded in short in this work. The other stages, implementing a FIFO channel within an environment of memory shared to a set of masters, pose some more difficult questions that will be the main focus here.

4.2 State of the Art

A standard model of communication used in MPSoCs is the connection of functional units with data pipelines. They can naturally be used in data stream processing [70, 69] often found in runtime environments (RTEs) designed for MP-SoCs [62, 74] or even directly included into the MPSoC itself [6]. A simple setup of a few computation units coupled together to a pipeline and more sophisticated kahn process networks (KPNs) [25] facilitate FIFO streams to build connections between the PEs running the compute units.

The traditional FIFO hardware unit resembles a dual-port memory bank with one port for writing and one for reading data. Since these ports lack an address field, a FIFO unit is mapped to a magic address in the master's memory space. This mapping must be done at design time, fixing the sender and receiver masters to specific PEs. In addition, the size of the channel has to be defined at design time by selecting a unit internal data memory. Since the memory must be sufficient for the worst-case scenario (e.g. the biggest application) and subsequent application changes should be possible, the size is often over-provisioned. As a result, a huge chip area is not or badly utilized as the internal memory is private to the hardware unit. Furthermore, in traditional FIFO chan-

nel units, the data has to be passed twice through the unit ports. Unless the application runs a very specialized algorithm that manages to write results directly from the register file into the FIFO this means that each token has to be loaded from memory and then written to the memory-mapped FIFO. Yet another issue is the dual-port nature of this system. In the hardware platform that was introduced in Section 1.3, single port memories are assumed to save chip area because dual port memories need almost twice the area for the same capacity than the single ported counterparts [60, 48, 7]. It would be possible to design a traditional FIFO unit with only single port memory banks, but it would reduce the bandwidth to 0.5 tpc because producer and consumer could only access the unit in turns.

Another possibility is to implement a FIFO unit in software only [77], without the need for additional hardware. In software, the buffer is usually presented as an allocated memory block in a memory shared by producer and consumer. It is interpreted as a cyclic buffer, and both producer and consumer manage a set of pointers that chase each other around the circle to define in which position data can be written or read. In that sense, the FIFO channel is just a pointer juggling controller. The benefit is that the masters do not have to pass their data through a FIFO interface but can produce or consume the data directly in the FIFO buffer. Another benefit is to have random access to arbitrary sized part of the FIFO buffer. This windowed access was even implemented in a hardware solution in [17].

A way to circumvent both the bandwidth limitation of single port access and the increased size of dual-port is to use double buffering [17]. The FIFO buffer is split onto two separate memory banks. While the producer has access to the first memory bank, the consumer accesses the second bank. When both are finished writing in their respective bank, the access is switched. While this is an excellent way of overcoming most of the problems mentioned earlier, it introduces a new assumption. Both masters must always be half of the buffer length apart. Even in low load situations, the consumer must wait for the producer to finish a half buffer worth of data before gaining access, which increases the data delay.

While all the mentioned solutions have their drawback, they all contribute a piece to the solution that will be presented in the following. The four ideas that are brought together are:

1. Use a hardware unit to parallelize FIFO access and application computation.

2. Use shared memory to prevent high hardware costs due to private and over-provisioned memory.
3. Use pointer juggling to use in-memory data manipulation and prevent extra passing of data through interfaces.
4. Use double buffering on single port memories to prevent high hardware costs due to dual port memories.

Implementing these four ideas promise to create high performance and low-cost FIFO channel implementation. Only the delay introduced by the double buffering is not covered by these ideas and has to be dealt with separately. The basic idea of putting the data into a central shared memory greatly benefits from the memory management system proposed by [83] (and introduced to the assumed system in Section 1.3.3). It allows the efficient access of multiple users to a multi banked local memory system which is a perfect match for the FIFO implementation to be proposed in the following.

4.3 System Environment

The newly designed FIFO unit is embedded in a system environment with a certain structure and attributes. Although the system is designed to feature the full FIFO channel with NoC transfer, only the on-PE part will be examined here. Moving data from the CPU to the NoC interface unit already requires the full FIFO channel functionality. As described in Section 1.3, a PE features a memory system with multiple banks and multiple masters. Most of the masters are memory ports of processors, but two belong to the networking unit. These are used for streaming data from the local memory to the NoC and vice versa. On the memory side, all but one bank are standard data storing units. The last one is a config space that allows special units to appear as memory-mapped functions to the masters. On PE level, a FIFO channel breaks down to a data transfer from one master to another, may it be two processors ports for a local FIFO channel or one of the masters be a network unit port for a channel reaching to another PE. In any case, the masters can use the memory-mapped config space of the FIFO unit to control the channel.

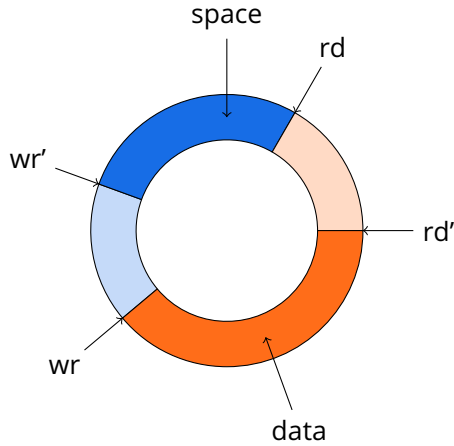


Figure 4.2: Concept of cyclic windowed FIFO with read pointers rd , rd' and write pointers wr , wr' . The pointers rd and wr partition the buffer into valid data and free space. The space between rd and rd' as well as wr and wr' describe actively used windows. (From [38])

4.4 Sliced Windowed FIFO buffer

The concept of a windowed FIFO has its origins in the software development community [77]. In contrast to a standard FIFO, no data is exchanged between the FIFO unit and its using masters. Instead, the FIFO unit merely acts as a controller. One FIFO channel connects exactly two masters with a data pipeline, namely the producing and the consuming master. Upon a request, it will return an access window, defined by a start pointer and a length. The window itself does not originate in the FIFO itself but within a shared memory area determined at runtime by the FIFO controller. The controller ensures exclusive access to the window for one master to produce or consume data. After the master has finished its transactions, it releases the window to the FIFO controller, handing over the regained memory space to the other master.

The organization of the FIFO buffer is depicted in Fig. 4.2 and explained in [38] in detail. There are four pointers rd , wr , rd' , wr' that chase each other in the cyclic buffer, dividing the buffer into regions of data, space, and current windows for reading and writing data. This organization allows the implementation of a FIFO

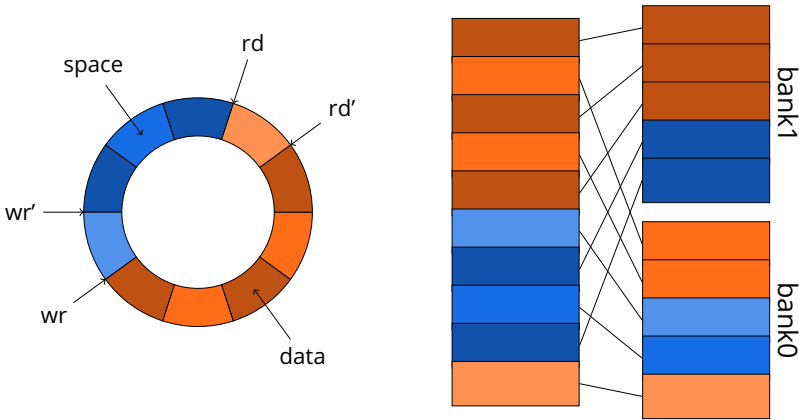


Figure 4.3: Mapping of a sliced cyclic FIFO to a memory system with two banks. (From [38])

without any locks. Virtually expanding the buffer by a full length also prevents the ambiguity of $rd = wr$. Without the extension, the situation must be mapped to the meaning “empty buffer”, thus making a full buffer impossible. With the extension the “full buffer” state is represented with $rd = wr + len$. Pointers are passed to the masters as the modulo of the length ($wr \bmod len$) to prevent access outside the real buffer size. A master can get access to the FIFO buffer by requesting a window for reading or writing. Opening and closing these windows by using one of the four atomic operations: `peek`, `pop`, `poke`, `push` will move the four pointers accordingly to a set of rules to protect data integrity.

Double buffering can be implemented within windowed FIFO buffers to avoid the 0.5 tpc bandwidth boundary caused by single port memories. It is extended by an interleaving technique to improve on the data delay caused by double buffering. The cyclic buffer is sliced, and its parts are distributed in the memory to randomize access patterns of the masters and giving the unit its name “Sliced Windowed FIFO buffer”. It uses multiple physical memory locations, called bases, to describe the FIFOs data store. The number is configurable at runtime but is bounded by a design-time parameter of the unit. Preferably the memory locations are in different physical memory blocks to achieve the desired performance improvement. The slicing of a FIFO buffer of two bases into two memory banks is shown in Fig. 4.3. Traversal of the access windows

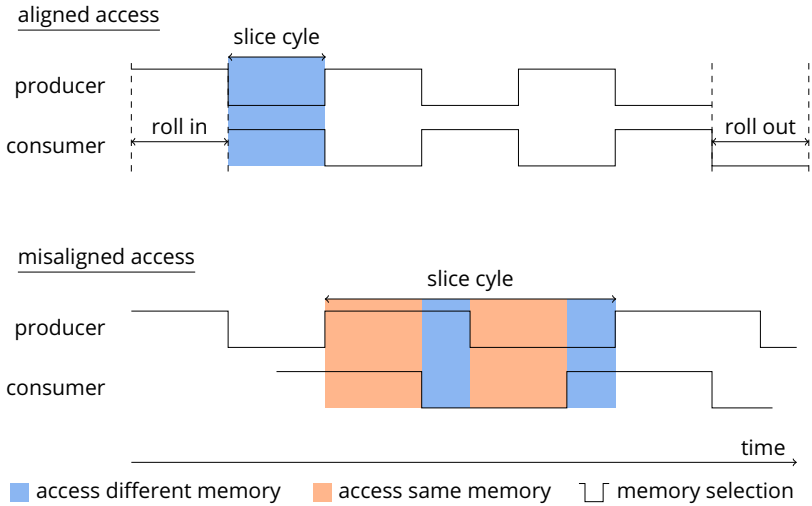


Figure 4.4: Memory access behavior with and without access pattern alignment. (From [38])

around the buffer will cause the pointers to jump between the physical memories every time reaching the end of a slice. The FIFO controller ensures that a window is contained only within a single slice to avoid fragmentation of the accessed memory.

4.5 Single FIFO Evaluation

For detailed single FIFO performance we refer to our work in [38]. It serves as an example to understand the effect of the presented FIFO buffer for throughput and delay. The performance was measured concerning the problems described in Section 4.2. For the single FIFO buffer, a 2-master, 2-memory-bank system was proposed. The two masters—in this setup called “producer” and “consumer”—use simple data source and sink models with a constant data rate. The sole parameter “data rate” describes the number of data tokens handled by the master per cycle. The results of interest are the runtime of a test, the throughput and the delay, which is the time a single data token will remain in

Table 4.1: Standard parameters for FIFO unit simulation setup.

parameter	value
buffer length	4096 tok
slice length	128 tok
producer rate	1.0 tpc
consumer rate	1.0 tpc
transfer length	8192 tok
arbitration policy	consumer

memory after production before being consumed. For the standard parameters, which are listed in Tab. 4.1, the setup achieves a runtime of 9021 cyl (throughput of 0.908 tpc) with a speedup of 1.91 compared to the non-sliced FIFO buffer example with a runtime of 17 249 cyl (throughput of 0.475 tpc). The theoretical speedup that can be achieved when switching from a single port to a dual port memory is 2.0. A double buffering setup can be created by setting the slice length to buffer length e.g. 4096. A data delay of 4118 cyl was measured in this setup. That is 14 times longer than the 283 cyl measured in the sliced setup with a slice length of 128.

This maximum speedup, however, is bounded by the system architecture and the masters' parameters. The theoretical throughput boundaries derived from the memory interface speed and the data rate are marked in Fig. 4.5. Each memory bank can process one request in each cycle. Each token needs two accesses to be processed, one for entering the memory and one for being read back, resulting in memory throughput boundaries of 0.5 tpc and 1.0 tpc for one and two used banks, respectively. Similarly, the data rate bounds the system performance because the throughput cannot be higher than the minimum of the produce and consume rate. For data rates below 0.5 tpc, this boundary limits the throughput, rendering the exploitation of the presented features impossible. In other words, if the data rate cannot even utilize a single memory bank, it does not help to add another. The speedup curve in Fig. 4.5 summarizes the matter. Until a data rate of 0.5 tpc, there is no benefit from using slicing. The speedup takes off from there in a quasi-linear fashion until it reaches its maximum of 0.9 tpc at a data rate of 1.0 tpc.

The experiments show that these results also depend on the alignment of memory access patterns explained in Fig. 4.4. When aligned, the producer and the consumer always access different banks resulting in a perfect memory in-

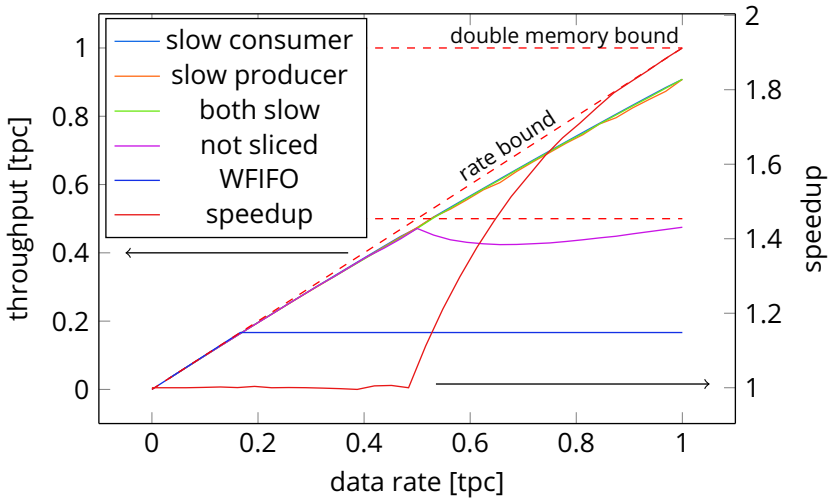


Figure 4.5: Sliced FIFO system throughput for different data rates and speed up over the not sliced variant. As a comparison estimated throughput of the WFIFO [39] is included. (From [38])

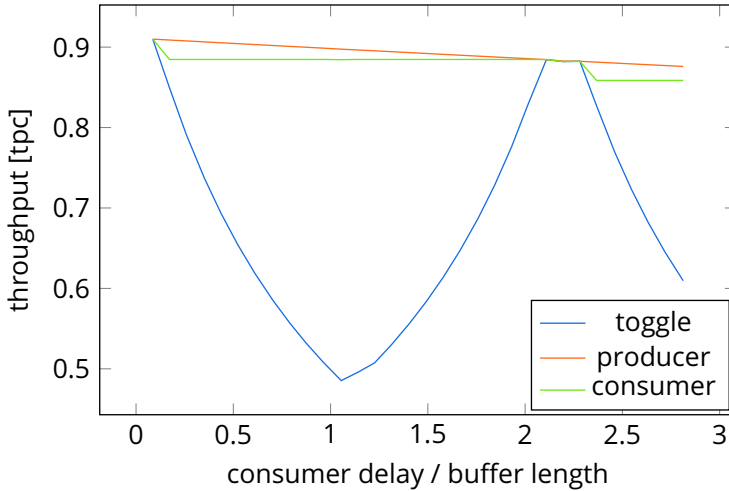


Figure 4.6: Performance of slicing FIFO for different arbitration policies depending on the delayed start of one actor. (From [38])

terface utilization, which leads to high throughput. Without proper alignment, both masters access the same bank in some time intervals, and memory congestions lead to lower throughput. This problem can be overcome by prioritizing one of the masters, forcing an access pattern alignment. In Fig. 4.6 the effect of not forcing alignment is shown. Depending on the consumer's start delay, the throughput can vary significantly.

4.6 Multiple FIFO Evaluation

To further explore the capabilities of the FIFO controller unit, the system is extended to host multiple FIFO buffers at once. The system parameters used in the multi buffer tests are summarized and used as defined in Tab. 4.2, if not otherwise stated. The results will be compared to a trivial reference system that does not use slicing while still using the same amount of FIFO buffers and the same traffic model parameters. The buffer instances are spread evenly across a number of memory banks equal to the sliced FIFO buffers bank number. To fully exploit the multi FIFO buffer system, a different traffic model and prioritization policy must be used. In the new traffic model, the throughput measure

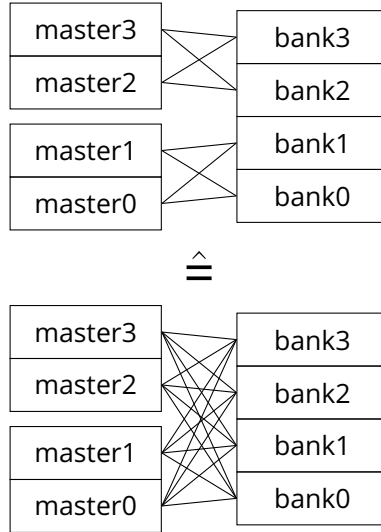


Figure 4.7: Equal performant setups. Top figure shows two FIFOs using two bases each on two physically separated memories. Bottom setup show two FIFOs using four bases on four shared memory banks. With constant data producer/consumer in all cases all memories are fully utilized.

is not expressive anymore and is replaced by the message delay.

4.6.1 Traffic Model

In Section 4.5, it was stated that to increase performance using a sliced FIFO system, a constantly producing master has to have a data rate greater than 0.5 tpc. Otherwise the FIFO channel can just be allocated in a single memory bank and the two masters (i.e. producer and consumer) would share the memory bank bandwidth without even exhausting it, because the bandwidth requirement of the two masters $b_{\text{master}} < 0.5 \text{ tpc}$ is less than the bandwidth provided by the memory bank $b_{\text{bank}} = 1.0 \text{ tpc}$:

$$b_{\text{bank}} > b_{\text{producer}} + b_{\text{consumer}}$$

With the sliced FIFO system, the two masters of each channel can draw the required bandwidth from multiple memory banks. In general the bandwidth is shared equally by all masters so that the bandwidth each master can expect is:

$$b_{\text{master,max}} = \frac{n_{\text{banks}} b_{\text{bank}}}{n_{\text{master}}}$$

To maintain a master bandwidth of close to 1.0 tpc

$$n_{\text{banks}} \geq n_{\text{masters}}$$

must hold. Any system that adheres to this rule is decomposable into the basic structure shown in Fig. 4.7 of two masters working on two memory banks.

To model a producer with a fixed constant data rate is sufficient to understand the basic coherences, but does not allow the study of more complex systems. A typical producer is event-driven and will produce messages depending on the arrival of events. For purpose of performance analysis we assume a statistical model with message arrival time described by a Poisson process depending on a parameter λ , the message arrival rate. Similarly, the message consumption is described by another Poisson process with the parameter μ being the message consumption rate. Both processes are implemented in the producer as a stream of randomly emitted messages. The arrival rate and length of the messages, normalized to tokens, are λ and $1/\mu$ token, respectively. The consumers read data from the channel with a constant 1.0 tpc if possible, implementing an average message consumption of μ with the given message length distribution. From arrival and consumption rate a theoretical channel utilization can be derived as

$$\rho = \frac{\lambda}{\mu}$$

describing the fraction of time the channel is actively using the memory. With this model, the average memory bandwidth each master requires is

$$b_{\text{master,req}} = \rho \times 1.0 \text{ tpc}$$

to allow multiple masters per memory bank while still exploiting the increased momentaneous bandwidth in the active intervals.

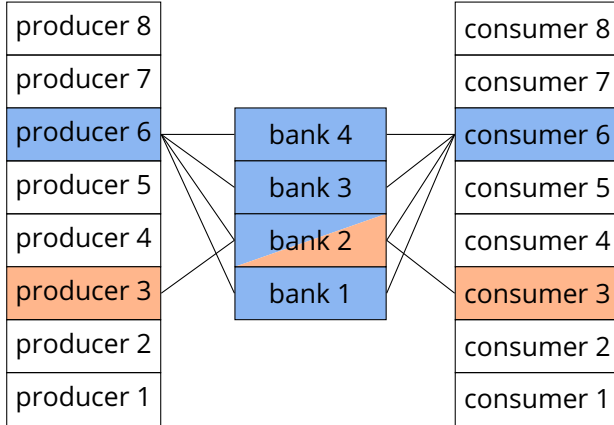


Figure 4.8: Simulation setup to evaluate the performance of multi FIFO environments. Each sliced FIFO channel (in blue) use every memory bank whilst the non-sliced variant (in orange) only uses one

4.6.2 Evaluation Setup

A simulation environment was used that features eight producers and eight consumers connected to a memory system of four banks, to evaluate the multi FIFO channel setup. The maximum average bandwidth each master than may use is

$$b_{\text{master,max}} = \frac{n_{\text{bank}} b_{\text{bank}}}{n_{\text{master}}} = \frac{4 \times 1.0 \text{ tpc}}{16} = 0.25 \text{ tpc.}$$

and subsequently the channel utilization must hold $\rho < 0.25$.

The presented sliced FIFO architecture is compared to a not-sliced variant that constructs a channel with producer and consumer working on a single memory bank. The non-sliced setup also features eight channels with pairs of two sharing one memory bank, to match the sliced FIFO setup. With four independent systems of one bank and four masters the maximum average bandwidth for each master also resolves to $b_{\text{master,max}} = 0.25 \text{ tpc}$, like in the sliced setup.

The standard parameter to set up the simulations are summarized in Tab. 4.2. The consumption rate of $\mu = 2.0 \times 10^{-4} \text{ cyl}^{-1}$ results in an average message length of $s_{\text{message}} = 5000 \text{ tok} > 2048 \text{ tok} = s_{\text{slice}}$, which is bigger than the slice

Table 4.2: Parameters for multi FIFO unit simulation setup

parameter	symbol	value
slice length		2048 tok
message arrival rate	λ	up to $0.5 \times 10^{-4} \text{ cyl}^{-1}$
message consumption rate	μ	$2.0 \times 10^{-4} \text{ cyl}^{-1}$
number FIFOs	n_{FIFO}	8
number masters	n_{master}	16
number bases	n_{base}	4
number memory banks	n_{banks}	4
memory bank bandwidth	b_{bank}	1.0 tpc

length, assuring a message transfer actually incorporates memory bank hopping. The channel utilization constraint is assured by choosing the message arrival rate of $\lambda < 0.5$.

4.6.3 Effective Channel Bandwidth

Multiple factors prevent a channel from actually reaching the theoretical bandwidth of 1.0 tpc. Firstly, the opening and closing of channel access windows cost a few cycles, effectively lowering the overall throughput. With a sufficiently large slice length, this effect is negligible.

Secondly, the bandwidth of a message is defined as the ratio between message length s_{message} and its transfer delay d_{message} :

$$b_{\text{message}} = \frac{s_{\text{message}}}{d_{\text{message}}}$$

The transfer delay is the time from the first write access of the producer until the consumer has read the last token. Although producer and consumer are acting most of the time simultaneously, the consumer's start is delayed creating two phases (e.g. roll-in and roll-out) at the beginning and the end, where only one master is active. The consumer's delayed start is necessary to achieve the desired aligned memory access, as shown previously in Fig. 4.4. So, even if both producer and consumer experience a full 1.0 tpc for reading and writing, the message bandwidth is decreased by this effect. It can be calculated as

$$d_{\text{message}} = \frac{s_{\text{message}} + s_{\text{roll}}}{b_{\text{master}}}$$

with s_{roll} being the length of roll-in and roll-out phase in token and the b_{master} the actual experienced bandwidth of both masters. The length of the rolling phases is at most the slice length, but may be less if a message is shorter than a slice $s_{\text{roll}} = \min(s_{\text{message}}, s_{\text{slice}})$. Putting all together the message bandwidth depends on the actual—or effective—master bandwidth as:

$$b_{\text{message}} = \frac{s_{\text{message}} b_{\text{master}}}{s_{\text{message}} + \min(s_{\text{message}}, s_{\text{slice}})}$$

For long messages, the rolling phases do not have a significant impact on the bandwidth. The smaller the messages, the bigger the impact.

The third factor affecting the effective message bandwidth is the effective master bandwidth, which is the memory bandwidth each master receives. It is decreased from its optimum of 1.0 tpc by interference with masters of other channels. Even with low channel utilization, there is a probability that messages transfers of different channels may overlap and cause memory access collision. Assuming a balanced distribution of all active masters over all available memory banks each master receives a bandwidth of

$$b_{\text{master,active}}(n) = \min\left(\frac{n_{\text{bank}} b_{\text{bank}}}{n}, b_{\text{bank}}\right)$$

depending on the currently active masters n capped by the bank bandwidth $b_{\text{bank}} = 1.0$ tpc. The number of active masters can be regarded as a binomial random variable with a parameters $p = \rho_{\text{effective}}$ and $n = 2n_{\text{FIFO}}$. Because $\rho = \lambda/\mu$ still assumes a master bandwidth of $b_{\text{master}} = 1.0$ tpc it has to be replaced by:

$$\rho_{\text{effective}} = \frac{\lambda}{\mu_{\text{effective}}} = \frac{\lambda}{\mu b_{\text{master}}} = \frac{\rho}{b_{\text{master}}}$$

Without the assertion that $b_{\text{master}} = 1.0$ tpc the consumption of a message of length $1/\mu$ is prolonged by a factor of $1/b_{\text{master}}$, meaning the effective consumption rate is $\mu_{\text{effective}} = \mu b_{\text{master}}$. The effective channel utilization then becomes a function of the number of active channels $\rho_{\text{effective}}(n)$.

A Markov chain is constructed with one state for each active channel number, to obtain the probability that a certain number of channels is active. The state transition probabilities are the probability that a certain number of channels are active under the assumption of the number of currently active channels. They can be described with a binomial distribution

$$t_{f,t} = \binom{n_{\text{FIFO}}}{t} (1 - \rho_{\text{effective}}(f))^{n_{\text{FIFO}}-t} (\rho_{\text{effective}}(f))^t$$

for a transition from state with f to t active channels. With more current active channels, the effective bandwidth decreases, increasing the probability of having a high number of active channels. The results of the Markov chain solution are the probabilities $P(n)$ that the system has n active channels. With these probabilities and the effective bandwidths, the expected effective bandwidth can be derived as:

$$E(b_{\text{master}}) = \sum_n P(n) b_{\text{master,active}}(n)$$

In Fig. 4.9, the master bandwidth estimation is plotted for sliced and non-sliced variants. As a comparison, the measured bandwidth for both messages and masters is included. The constant offset between memory access and transfer bandwidth due to the roll-in and roll-out phases discussed earlier. The sliced provides significantly higher bandwidth and can sustain a constant benefit above the non-sliced variant up to a system load of $\rho = 0.7$. Above this value, the bandwidth drops to join the non-sliced bandwidth, reaching the physical memory boundary at a fully loaded system.

4.6.4 Memory Access Model

Compared to the single FIFO setup, the priority ordering for a multi-channel setup is much more complex. Because the number of configurations grows exponentially with the number of masters, there are only two in the simple setup (favoring “consumer” or “producer”) but a large number in a complex system. In Section 4.5 a significant performance impact could be observed depending on the chosen configuration. Because the masters’ access patterns are not deterministically aligned, it is even more critical to have a mechanism that distributes multiple masters over the available memory banks. A hard priority system worked well with two masters, but with multiple masters, severe problems are to be expected. With 16 masters ordered in a fixed priority, some of the lower priority masters likely starve when masters with higher priority experience an increased utilization. In Fig. 4.10 a situation is displayed that would result in the starvation of one master under a fixed priority system. As soon as the number of masters exceeds the number of banks, it is mandatory to replace the priority system with a more sophisticated solution.

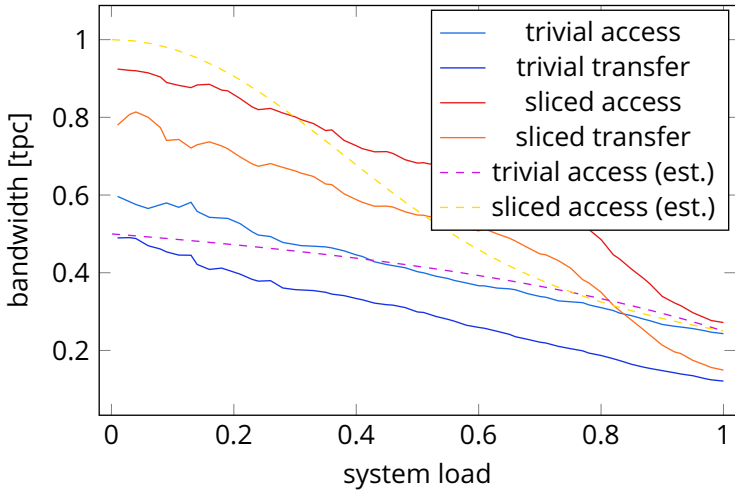


Figure 4.9: Per channel available bandwidth depending on the system load. Values are shown for both trivial and the proposed (sliced) setup. “Access” measures the memory bandwidth a master receives from the memory system. “Transfer” measures bandwidth a message (block of data) takes to travers the FIFO. From first write to last read.

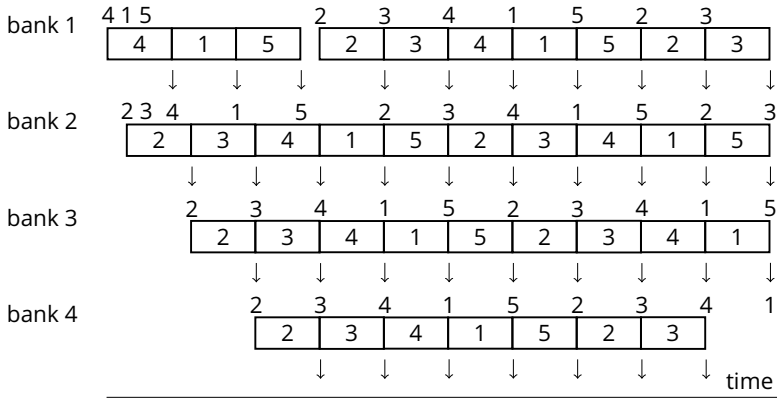


Figure 4.10: Memory access queue on a 4-bank, 5-master configuration. For each bank the access intervals are shown, labeled with the accessing master's id. Above, the access request to a bank are indicated, also with ids. Whenever a master finishes an access interval, it immediately requests the next bank, as indicated by the down arrows. A waiting queue forms in bank 2 where every master has to wait between access request and granting.

The solution proposed here are memory access queues. A requirement for this approach is that the masters are trying to access the same memory bank for a sufficiently long time without skipping a cycle. In case of a collision, the arbitration system grants access to the master that started its access earliest. A master having access will keep it till it intermits its requesting streak. The arbitration is done per memory bank so that the jumping between banks assures that no master is starved, and every master has the same chance of getting access, thus receiving the same memory bandwidth. Masters trying to access the same bank simultaneously are moved apart from each other till they are accessing different banks, a state that they most likely will keep automatically.

To implement this behavior, the memory arbiter features an access queue for each bank that can hold all master ids. Whenever a master starts requesting access to a bank, it will be appended to the access queue. Stopping to request access will cause a master to drop out of the queue. The access to the memory will be given to the master in front of the queue.

In Fig. 4.10 this behaviour is pictured. On memory bank 2 masters 4, 1 and 5 are queued for access while 2 and 3 are queuing for bank 1. In a priority based system with the masters having a priority equal to their number, master 1 would never gain memory access. The four high masters would spread through the system each occupying one bank at any time. With the queue based system each master has to wait occasionally before gaining access to a bank. In the example in Fig. 4.10 the queue forms in bank 2 where every master has to wait for another master before gaining access. This way the bandwidth is equally shared and memory access starvation is assured to not happen. The transition from priority based to a queue based access control is a necessary step for the extension from single to multi FIFO systems.

4.7 Hardware Implementation

A hardware implementation of the FIFO controller was included in the Kachel 1 chip [36]. It is produced in a 22 nm FDSOI technology to run at 500 MHz. The hardware implementation matches the simulations done for the single-channel setup. It does not support the multi-channel setup because only two bases are supported. The inclusion of those two additional bases will not increase the area consumption significantly. From place and route of the chip the area consumption of the unit was extracted to be:

Table 4.3: On-chip area consumption of 4 kB memory macros

unit	area [μm^2]	density [$\text{kB } \mu\text{m}^{-2}$]
single port 22nm	13 705	2.91×10^{-4}
dual port 22nm (est.)	25 863	1.54×10^{-4}
single port 28nm	8873	4.50×10^{-4}
dual port 28nm	16 745	2.38×10^{-4}

$$A_{\text{FIFO}} = 15\,826 \mu\text{m}^2$$

One main advantage of the proposed FIFO controller is to allow the use of single-port memory macros in the memory system in contrast to the dual-port macros that need to be used usually to prevent congestions in FIFO channel applications. The area reduction by using single port macros is significant. In [38] a comparison of dual-port and single-port static random access memory (SRAM) memory macros is done. This table is replicated in Tab. 4.3 and extended by the storage density. The storage density is clearly doubled when switching from dual-port to single-ported SRAM macros as also observed by [60, 48, 7].

The total chip area of memory banks —together with the FIFO implementation in the single port case— depending on total memory capacity is shown in Fig. 4.11. The discrete values are based on the described 4 kB memory macros. Interpolating between the discrete points reveals the break-even point of equal chip area for traditional and proposed hardware:

$$x = \frac{A_{\text{FIFO}}}{A_{\text{dp},22} - A_{\text{sp},22}} 4 \text{ kB} = 5.2 \text{ kB}$$

At this point the savings from replacing dual-port with single-port memories is equal to the area of the added FIFO controller implementation. Already the minimal setup for employing a sliced FIFO channel of two single ported memory banks (i.e. 8 kB) and the FIFO controller is smaller than a traditional setup with two dual port banks and (close to) no additional logic.

4.8 Key Learnings

The optimization of FIFO channels for embedded computation system has multiple facets. In the current state of the art, there are excellent solutions for

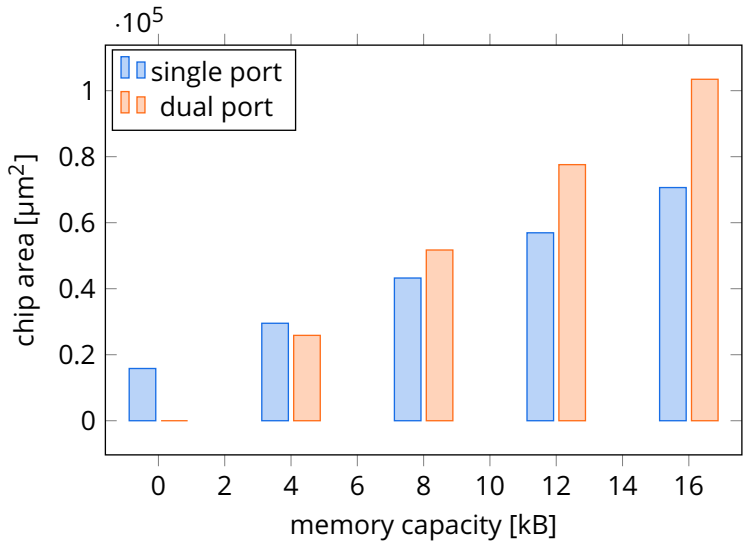


Figure 4.11: Chip area consumption of FIFO system composed of controller unit and memroy banks based on 4 kB memory marcos. When using dual port memories the proposed FIFO controller is left out. The break-even point can be interpolated to 5.2 kB.

any attribute that might be optimized. Every solution has severe drawbacks in other attributes. The challenging problem is to find a solution that finds a sweet spot that exploits the advantages of each solution without incorporating the full amount of drawbacks. When optimizing only for bandwidth/delay, a dedicated hardware (HW) unit can provide a perfect throughput of 1.0 tpc and a minimal delay. The drawbacks are the exclusiveness of the used memory to the HW unit and the need for area expensive dual-port memory macros. Since the on-chip memory is one of the most precious resources in an embedded device, this is unacceptable. The apparent area optimal solution is a software FIFO channel implementation with a single port memory. It suffers from delay and bandwidth degradation, where even a single channel cannot exceed 0.5 tpc. When masters need to prosume¹ token at a higher rate, the multiple parallel accesses have to be conducted. The memory system must be able to expose multiple of the memory banks simultaneously to multiple masters to increase the usable memory bandwidth. The data buffer has to be sliced and distributed to different memory banks to exploit the higher bandwidth. When multiple channels are active at once, the memory access patterns must be aligned to exploit the parallel memory access.

In the here proposed hardware FIFO unit data and access patterns are distributed over shared memory banks to find a sweet spot between optimal bandwidth and optimal chip area consumption. In the completed simulations it was shown, that the proposed design yields 80% of the theoretical possible bandwidth of 1.0 tpc. From the implementation of the unit into an manufactured MPSoC it was derived that the unit fills an area equal to 1.15 memory macros of 4 kB with a single port or 0.53 of the dual ported variant. That allows building a 8 kB memory system with single port memories and a FIFO unit that is smaller than an equivalent sized system with dual-port memories without such a unit.

¹portmanteau of produce and consume

5 Message Passing Hardware

Apart from the first in first out (FIFO) channel communication discussed in the last chapter, a second type of communication, namely message passing (MP) plays an essential role in the efficient execution of parallel applications. Similar to the data streaming communication, MP suffers a severe performance loss when implemented on top of standard on-chip communication like remote direct memory access (RDMA). The transfer of small messages with random source and destination within a cluster of many processing elements (PEs) has some bottlenecks that can be addressed with dedicated hardware implementation. The two key performance indicators that are of interest for an MP implementation within an embedded environment like the one discussed throughout this work are message delay and resource efficiency while bandwidth is a secondary goal.

The abstract use case for MP in this type of environment is a client-server relation between several nodes. A potentially great number of clients are sending requests to a server and are expecting a fast response. These requests may for example be a command to a file system service in a distributed operating system (OS), but also the search for a blockid in an dynamic static data flow (SDF) scheduling system like seen in Section 3. In neither case it is needed to transfer a lot of data but rather get a small message to another node as fast and as efficient as possible. This chapter will introduce an extension to an existing RDMA unit that handles an MP protocol featuring connectionless transfers and memory-efficient buffer management.

5.1 Addressed Performance Hotspot

The usual way to implement MP is by utilizing an existing RDMA implementation since it is widely available on many platforms [43, 44, 2, 73]. Using an RDMA unit as the hardware foundation to implement the transfer of data across the chip leaves the processor to run the MP protocol in software, which not only introduces an additional delay into the message delivery pipeline but also interrupts the processor from running the application code. Also, RDMA lacks some features that would allow connectionless transfers, which results in some significant network traffic overhead and memory consumption to save the connection state. A dedicated connection setup for every peer to peer connection has, in the best case, a fixed resource consumption on both sender and receiver. When a service has to receive messages from many clients, this may accumulate to significant memory consumption. Additionally, each connection must be checked for liveness once in a while, meaning some network traffic and processing overhead for each link, which would also accumulate on service units to a significant amount of the processing time. And lastly, for connection-based communication, a connection establishment functionality must be provided. Without any message passing mechanics, this is non-trivial and requires a separate service, connected to every process, and manages the establishment of a new connection from a centralized position. That is contradictory to the idea of a decentralized, distributed execution environment.

Since the use cases state that MP is mainly for signaling and not for bulk data transfer, the bandwidth that can be utilized for a message is a secondary goal. More critical are the delays caused by the message stages before the message enters and leaves the network on chip (NoC).

The mentioned drawbacks like connection establishment and resource consumption is not significant in the biggest RDMA application field, namely high performance computation in datacenters. However, for the use in embedded systems as signaling service for dynamic scheduling these attributes are crucial for the overall performance.

5.2 State of the Art

There are not many systems that try to implement a message-passing protocol stack directly in hardware. The reason is that in most cases, a software-based implementation is sufficient. Message delay and data bandwidth are

good enough for the application, or the system is based on a shared memory system. In that case, a hardware MP implementation may, in the best case, optimize a part of the system that is not considered a hotspot, thus have very little impact. In a worse case, it may be simply not applicable and a waste of resources (e.g. chip area). A software implementation and the importance of MP for multi processor system on chips (MPSoCs) is acknowledged in [57] but without the need for hardware acceleration.

Since including new hardware to a processing system is a tremendous effort, reusing already available functionality is often a much cheaper, and often an almost as efficient solution. In [73] a MP stack is built upon an existing RDMA unit. And still, there are attempts to do a direct hardware implementation [87], although not very frequent. As mentioned before, the necessity to implement such a highly optimized piece of hardware is not given in most projects. In Section 3.5, however, it was mentioned that the delay of network messages can influence the system performance significantly with sufficiently large systems.

5.3 Message Passing Regarded as Queueing

Generally, MP can be partitioned into four modes of communication: (1) 1-to-1 (2) N-to-1, (3) 1-to-N and (4) N-to-N. Each communication mode requires different hardware functionality to be implemented efficiently. This work will focus to the N-to-1 mode (Fig. 5.1) since it is the one used to build client-server system. To implement N-to-1 MP a message queue is placed at a node. A messages sent to the queue will be enqueued and made available to the destination node. This way, messages from multiple nodes are sequelized and can easily be processed by the destiantion node. Additionally, this work will restrict the messages sent to a queue to be of constant size. As long as MP is only used to tranfser request/-command messages a fixed message size does not pose serious restrictions.

On the MPSoC system described in Section 1.3 the management system from Section 3.4 can be implemented using such queues. Each management PE instanciates a queue to implement the interface to a service. The worker PEs can now query those services by enqueueing requests. It is possible for one worker to query multiple services at once and, more importantly one service can answer to request from multiple clients on the same queue. The implemented services may be a task placing service, a quality of service (QoS) network connection allocator, or a worker PE command queue, to name a few examples. But also examples from other applications are possible like a file system service or a pe-

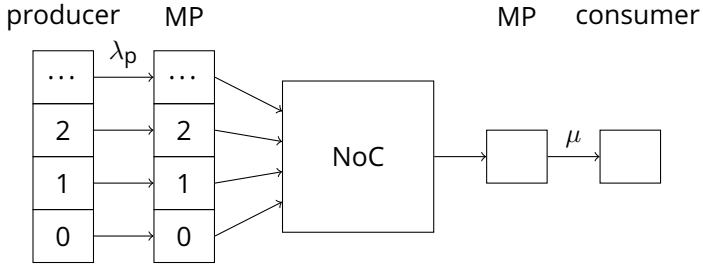


Figure 5.1: Overview of Simulation setup. Multiple producer each using an MP unit to transfer messages over the NoC to a single consumer. A producer p generates messages at a rate of λ_p . The consumer takes messages from the local HW unit with a rate of μ .

ripheral interface in a distributed OS. All services have in common that they are usually designed to have a lot of possible client nodes. Most PEs will only offer a single service, thus only have a single receiver. But a PE may potentially want to request a lot of services, so it will have to implement many message sending endpoints. Having many senders for a single receiver implies that the senders' message rate must be relatively low to prevent overloading the service provider. Assuming that a typical service can process a message in a constant time D gives a message consumption rate of:

$$\mu = \frac{1}{D}$$

A widespread way of describing message production in a sender is as a Poisson Process. It defines exponentially distributed inter-message timings with a mean message rate of λ . The ratio of arrival and consumption rate

$$\rho = \frac{\lambda}{\mu}$$

describes the queues utilization, which obviously may not exceed 1.0 because service times would approach infinity. For a queue with multiple independent Poisson Process driven senders the total arrival rate μ_t can be obtained as the sum of the individual rates or, with a constant arrival rate across the system, as product of the number of senders n_{sender} and the basis rate μ_b [47]:

$$\mu = \sum_i \mu_i = n_{\text{senders}} \mu_b$$

Therefore, the queues regarded in the following are expected to be close to an M/D/1¹ queue in their behavior. Only a single queue will be examined at once, assuming that the PE connecting NoC does not get congested.

The whole message queue from one subprogram to another spans over several units within the platform described in Section 1.3. Included are the source and the destination network interface units and the NoC. The life cycle of each message can be divided into five stages that describe the complete transfer. Depending on the used implementation and length of the message the stages may overlap. First, the message is “pushed” from the source processor into the network interface. Secondly, it must be prepared and “send” into the NoC. The “transfer” through the NoC is the third stage, after which the message will be “received” at the receivers networking unit. Finally, the message data is “pulled” from the MP stack to the receiving processor.

5.4 A Remote Direct Memory Access Based Implementation

In Section 1.3.4 the functionality of RDMA is explained. While the “put” and “get” primitives are sufficient for one-to-one communication, they introduce problems when working on a multi-point communication. In a one-to-one communication, specific local addresses are reserved for access by the communication peer. The peer knows about the location of these reserved buffers and can safely write or read data from or to it. The problem that causes the protocol overhead is that a peer does not know who, when, or how often the reserved area was written to or read from by a remote peer. Still, a safe and functional MP protocol is possible. The basis is a connection state that each peer holds for each connection. It features an off-band buffer that is used to signal the protocol state to the peer, containing a least three essential attributes. In conjunction with the off-band buffer, the connection state contains the address of the remote off-band buffer. This remote off-band address must be set in the connection establishment phase, possibly by a communication manager instance. Thirdly, the connection stage contains the local address of a message buffer.

¹ A/S/c is the Kendall notation to classify queues. The parameters describe “A”: arrival time distribution; “S”: service time distribution ; “c”: number of service channels. Often used codes for “A” and “S” are “M”: markovian (poisson process), “D”: degenerate (fixed time) [46]

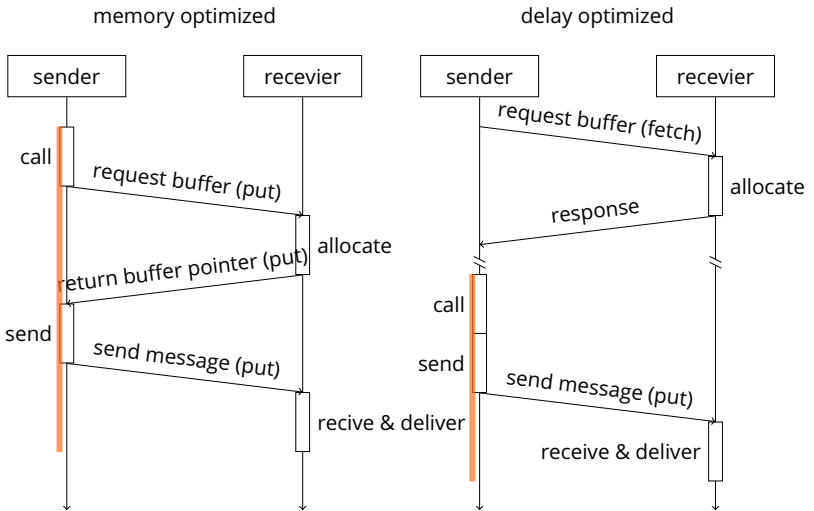


Figure 5.2: Flow of memory and delay optimized RDMA based message passing implementation. The effective delay from the sending to the receiving subprogram is marked which prefers the "delay optimized" variant.

There are two protocol versions, one that favors message delay, the other favoring memory consumption. For both, the transfer of a message is displayed in Fig. 5.2. This message flow assumes that there has been a connection establishment phase that exchanged off-band buffer locations between the two peers. Each peer now holds a connection state structure with a cleared “off-band buffer” field and a “remote off-band address” set to the location of the peer’s off-band buffer. When using the memory-optimized protocol, the sender must, before sending the message, query the receiver for an empty message buffer. It is done by writing a command to the receiver’s off-band buffer and wait for a reply in the own off-band buffer. After that, the sender can transfer the message to the assigned message buffer. On the other side, the receiver must find the buffer allocation request in the off-band buffer corresponding to the sender. It must then allocate a message buffer and send its address back to the senders off-band buffer and then wait for the message to arrive. For the delay-optimized version, the buffer allocation is moved out of the critical path. Immediately after connection establishment, the receiver will notify the sender about an allocated buffer location. Because the allocation has already been done, the sender can start immediately when the application wants to send a message. It sends a message and immediately requests a new message buffer.

Under the preallocated buffer strategy, the receiver must hold at least one allocated buffer for every sender, although it might be idle most of the time. The memory-optimized version allocates memory on demand to prevent this memory wasting, with the drawback that this request and allocation process prolongs the message delay.

5.5 Hardware Implementation Concept

Goal of the development of an MP hardware unit should be to improve the performance issues a reference MP-over-RDMA implementation has. The issues at hand are:

1. The actual MP protocol is implemented in software, which is slow and steals processing time from the application.
2. Connection establishment and a connection state are necessary in order for MP to work properly. Connections consume resources, may it be memory for the state, or time for the establishment and keep-alive handling.

3. In an RDMA based implementation a receive buffer must be kept for each connection, which results in memory consumption proportional to the number of clients.
4. To save resources buffer allocation can be done on demand, which has the downside of delaying the message significantly.

The hereby proposed implementation is based upon the existing networking unit briefly described in Section 1.3.4. Before development on the MP stack began, the unit already implemented a simple RDMA protocol. It also featured a FIFO channel bridge that can be used to connect FIFO channels described in Chapter 4 across the network to build an inter PE channel. Internally the network unit uses a multipurpose configuration file holding the configuration of a set of communication entities. Each entity can be individually chosen to describe an RDMA operation or a FIFO channel. For the MP protocol implementation, a third mode will be added to allow a configuration slot to hold a MP queue description. This recycling of the config registers attempts to reduce the area increase caused by the MP protocol implementation.

The MP configuration slot leans on the RDMA and FIFO channel versions. It only describes the metadata and holds pointers to the local shared memory for data storage. The application is responsible for local memory allocation since it is assumed to have a dynamic memory allocation. When initializing the MP queue, the application will grant the hardware unit a piece of dynamically allocated memory that it can manage until the MP queue is being destroyed.

The queue divides the assigned memory space into a list of message positions. Within the configuration file, parameters about the message and list size are stored. Further, a bitfield holds the status of each message position. It stores if a position is occupied and, in the sender's case, whether the message was sent. The message will remain in the buffer until a reception receipt is received from the receiver.

The sequence to transfer a message from one application processor to another is shown in Fig. 5.3. Both sides must set up a queue configuration and prepare a message buffer. On the sender side, the destination PE must be specified as well. The message transfer is handled by three control flows running in different finite state machines (FSMs), thus being independent of each other. It allows a decoupling of the application processors from the actual transfer.

The sending processor controls the first flow, and the HW unit is merely reacting. To initiate a message transfer, the processor requests a message position from the HW unit. After filling in the message data, the processor releases the

message to allow the second FSM to take over control and relieve the processor of any further actions.

A second flow handles the transfer between two HW units. The HW unit's state machine will continuously observe the queue configurations to find a message that is not yet sent. It then sends the message to the configured peer and change the message state to "sent". On the receiving side, the message will arrive at the HW unit. It selects a not occupied message position and stores the message there. Once the whole message is received, the message is made accessible by the application processor, and an acknowledgment signal is issued to the sender PE. This signal tells the sending HW unit that the message is stored to the remote memory and can now safely removed from the local message buffer. Back on the receiving end, a third flow controlled by the receiving processor can now commence. The processor polls the HW unit for new messages. If a new message is present, the HW unit will return one to the processor for reading. After being finished with the message, the processor releases the message, allowing the HW unit to free the message position.

There are several situations where the flow may differ from the optimal, already described, course. Both requesting a message for reading and writing may fail when no position in the local buffer is free. The application must then wait some time and retry its request. A message arriving at the HW unit from the NoC may also be unable to find a free position and then might be dropped. In that case, the receiver will send a negative acknowledgment to the sender, which will then retry the transmission later.

5.6 Evaluation of Performance

In Section 5.5, four performance issues were indicated that the new HW is supposed to improve. The value that describes the system's performance is the delay a message must expect to traverse the system. This delay is dependent on a range of parameters. One parameter the performance depends on is the system utilization, described as the ratio of message arrival and message processing rate $\rho = \lambda/\mu$. Other parameters that may affect the delay are the message buffer size or the number of senders.

To evaluate these values, an register transfer level (RTL) based simulation was set up like displayed in Fig. 5.1. The system consists of a set of nodes connected with a NoC allowing them to communicate. Each node contains a network interface unit featuring the RDMA and the MP implementation and a processor that

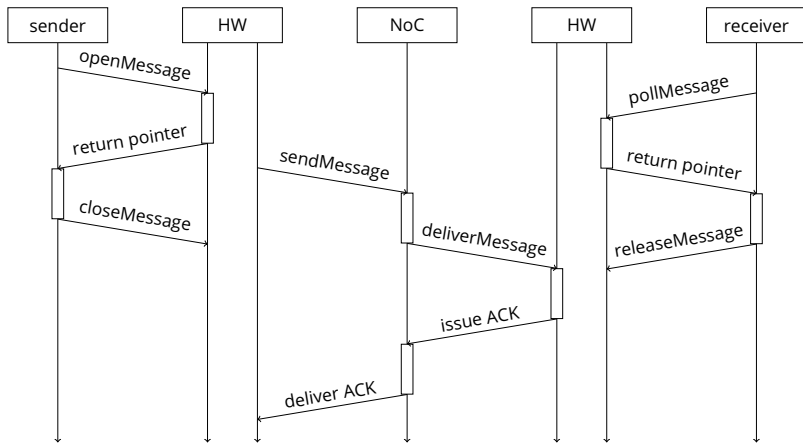


Figure 5.3: Flow of message passing protocol implementation. The transfer is done by three decoupled processes. Both sender and receiver query the HW unit for access to a (new) message. In a decoupled process the two HW units transfer the data independent of producer/receiver. All three flows can run simultaneously. The NoC transfer can start “sendMessage” at any time after “closeMessage” and “pollMessage” can already be issued after “deliverMessage” has finished.

Table 5.1: Standard parameters for measurements

parameter	symbol	value
consumer rate	μ	1 / 50 cyl
system load	ρ	0.0 to 1.0
message length		8 tok
number of producers	n_p	5
buffer size		4
NoC delay		10 cyl
producers message rate	λ_n	

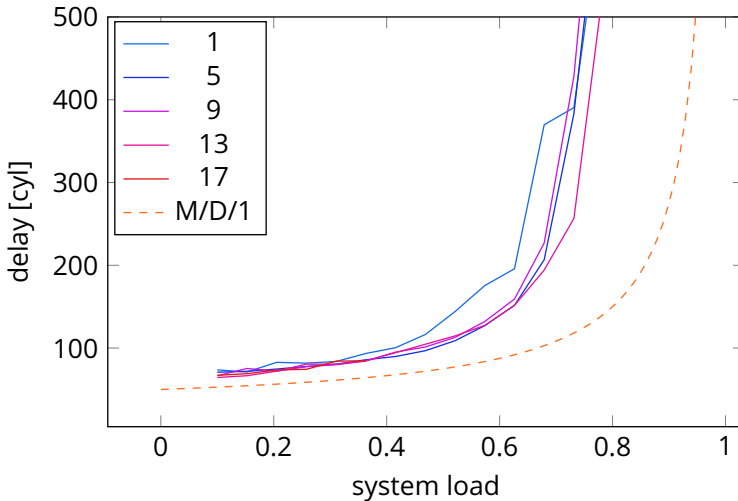


Figure 5.4: Performance of Hardware MP unit with different number of producers.

allows modeling either a producer or a consumer of messages. When configured as a producer, a processor will randomly generate messages and pushes them to the network unit. The message generation follows a Poisson Distribution parametrized with a producer message rate λ_p . A consumer node works almost the same but polls messages from the HW unit and consumes them in constant time $d_c = 1/\mu$. The simulations always feature exactly one consumer and a set of n_p producers. The system load is calculated combining the producer rates to a total message arrival rate and the consumption rate as:

$$\rho = \frac{\lambda_p n_p}{\mu}$$

Other parameters of the simulation are given by Tab. 5.1. In Fig. 5.4, setups with a different number of active producers are compared. The individual message rates λ_p are scaled to keep the system load constant. In theory, a queuing system queried by multiple producers should behave the same as with a single producer with a message rate equal to the sum of the many producers assumed that the produced messages are Poisson distributed. However, with the different stages in the queuing system, it is not trivial to see if this relation holds. The

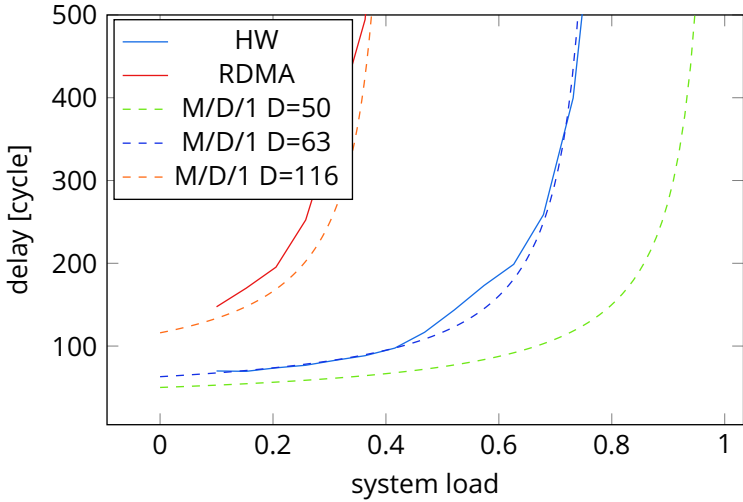


Figure 5.5: Comparison of message delay of HW implementation and RDMA based version.

Fig. 5.4 shows that, for the HW implementation, this relation does indeed hold, since the message delay is independent of the number of active producer with constant system load ρ . Although the implementation does not quite reach the theoretical performance of an ideal M/D/1 queue, it keeps good performance even in high load situations. Compared to that, the RDMA based version drives the queue into saturation at a much lower system load (Fig. 5.5).

The queue must be regarded as the concatenation of the three parts: the sender, the NoC, and the receiver. Replacing the message processing time d_p with the total message delay

$$d_{\text{total}} = d_{\text{send}} + d_{\text{NoC}} + d_{\text{receive}} + d_p$$

as the sum of all message stage delays, the theoretical model matches the simulated delay times. The same method can be applied for the RDMA implementation. The only difference is that the message buffer's acquiring is added as additional stage:

$$d_{\text{acquire}} = d_{\text{allocate}} + 2d_{\text{NoC}}$$

Table 5.2: Time consumption for MP operations.

parameter	description	HW	RDMA
NoC	NoC traversal	10	10
send	local memory to NoC	10	10
receive	NoC to local memory	10	10
p	message processing (CPU)	50	50
aquire	aquiring a remote buffer location	—	100
allocate	allocating buffer (CPU)	—	34
retransmit	retransmission in case of buffer overflow	100	—
pause	pause before new retransmission attempt	100	—

The additional delay further reduces the performance, which manifests in the system being saturated already at a system load of 0.4. It is possible to very accurately model this delayed queuing system by replacing μ in the standard M/D/1 delay model [13]

$$d = \frac{1}{\mu} + \frac{\rho}{2\mu(1-\rho)}$$

with the real message delay $\mu_{\text{total}} = 1/d_{\text{total}}$ but keep the normalization to system load $\rho = \lambda/\mu$ with a μ from the processing time only. The stage delays are system implementation-specific values and are obtained from RTL simulations of the presented hardware. Values are summarized in Tab. 5.2. With these timings available the total message delay for the HW implementation and for the RDMA implementation can be calculated as:

$$d_{\text{total,ASIC}} = 60 \text{ cycl}$$

$$d_{\text{total,RDMA}} = 110 \text{ cycl}$$

M/D/1 delay functions with these modified values for μ are plotted into Fig. 5.5 along with simulation results. For the sake of comparability, both application specific integrated circuit (ASIC) and RDMA version feature a single producer. The theoretical model matches the simulated results and shows the importance of processor to processor delay

One attempt to compensate the dependence on the transmission time is to exploit the use of a pipelining system. The availability of message buffers on both the sender and the receiver side is supposed to allow multiple messages to

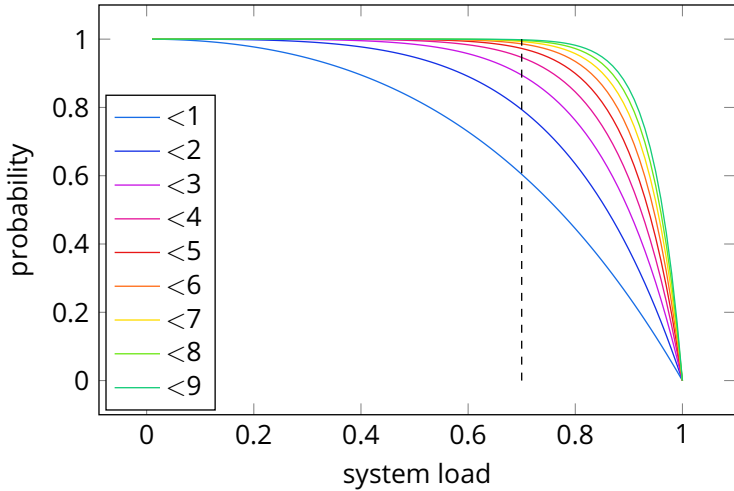


Figure 5.6: Probability that a certain number of messages are in the queuing system

be handled by the system simultaneously. Additional delays may be eliminated by storing a message in case the next stage is not available. For example, a message may be parked in the receiver buffer when the central processing unit (CPU) is busy avoiding a signaling path and retransmission all the ways back to and from the sender CPU.

Since the maximum number of message positions is fixed in the ASIC design itself, it is crucial to understand the benefit of a bigger buffer to decide at design time the parameters limiting this. In other words, the question that should be answered is how the buffer size affects the functionality and performance of a queue. Since the ASIC unit implements message rejection and retransfer mechanism, a single message position is enough to assure a functioning messaging system. A retransfer does, however, increase the message delay by:

$$d_{\text{retransmit}} = 2d_{\text{NoC}} + d_{\text{pause}}$$

It describes the transmission of a not-acknowledge message, the original message and a retransmission pause to prevent overloading the NoC. At last the retransmission is not protected against a repeated rejection which alters the total message delay to incorporate the number of rejections:

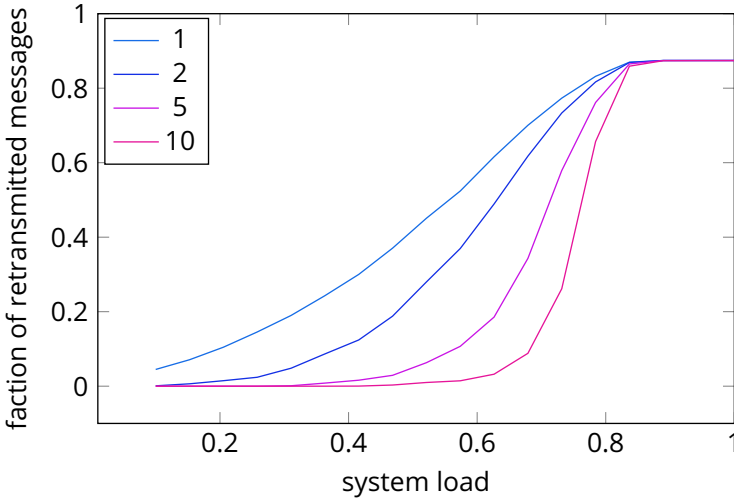


Figure 5.7: Ratio of messages that have to be retransmitted because of a buffer overflow depending on buffer size.

$$d_{\text{total}} = d_{\text{send}} + d_{\text{NoC}} + n_{\text{retransmit}} d_{\text{retransmit}} + d_{\text{receive}} + d_p$$

Luckily, these retransmissions are a relatively seldom event and depend on the size of the receiving buffer. To estimate the probability of a rejection, the probability that a number of messages are in an M/D/1 queue can be used. Normalized to a processing time of $D = 1$ it is given as [59]:

$$\pi_n = (1 - \lambda) \left(e^{n\lambda} + \sum_{k=1}^{n-1} e^{k\lambda} (-1)^{n-k} + \left[\frac{(k\lambda)^{n-k}}{(n-k)!} + \frac{(k\lambda)^{n-k-1}}{(n-k-1)!} \right] \right)$$

With $\mu = 1/D = 1.0$ the queue utilization becomes $\rho = \lambda/\mu = \lambda$. The probability of less than a certain number of messages being in the queue can be calculated as

$$P(X \leq n) = \sum_{k=0}^n \pi_k$$

and plotted as shown in Fig. 5.6. In the range of small queue utilization, a

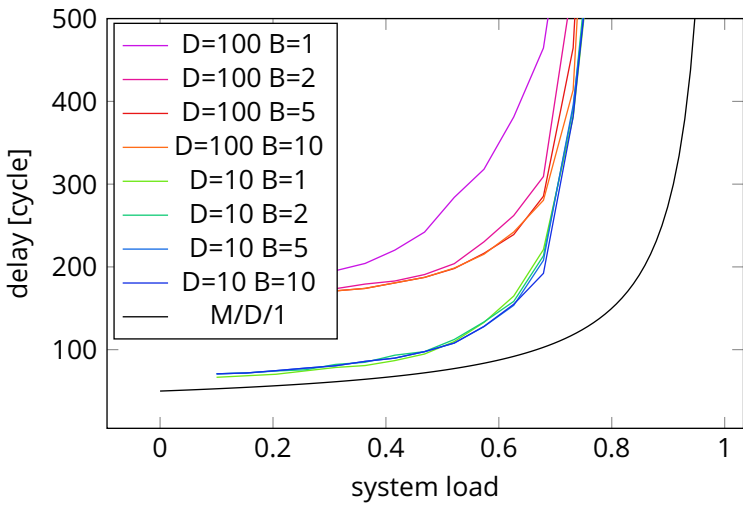


Figure 5.8: Message delay dependent on system load, network delay (D) and receiving buffer size (B). Only with a very small buffer coupled with a very high NoC delay has an effect on message delay.

single message position is sufficient to catch most messages directly, causing minimal rejections. But even at saturation utilization $\rho = 0.7$, a set of four positions assures a message acceptance of more than 90%. It is important to note that the remaining 10% are not lost but are only delayed by $d_{\text{retransmit}}$ cycles. Another experiment was set up to run the queuing system with different buffer sizes. Although a certain number of retransmissions can be seen in Fig. 5.7, no significant impact on average message delay can be observed. In Fig. 5.8, not even with a single message sized buffer, a dramatic performance reduction could be observed compared to the bigger buffers. Only an increase of the NoC delay from 10 cyl to 100 cyl could show the effects of choosing the buffer size too small. The increased probability of retransmissions and the high penalty of retransmission of over 200 cyl a performance degradation for a very small message buffer can be observed.

5.7 Key Learnings

In this chapter, a hardware solution is proposed that significantly improves on performance of signaling flow of distributed applications. With the immediate memory being so close to the processor, the querying of microservices or synchronizing with coworkers in the distributed application becomes the new bottleneck. It is essential to keep the overhead for the application processor for communication as low as possible. The two key factors are the software effort to access the communication stack and the delay the application potentially has to wait for an answer to its request. An ASIC MP unit can help with both issues. Implementing most of the MP stack in hardware moves effort away from the application and at the same time accelerates the processing. Firstly, that reduces the overhead for the application, allowing other processing while waiting for a response. And secondly, the improvements on the MP protocol itself decrease the message delay significantly, effectively lowering the waiting time the application experiences waiting for the response to a request. The smaller transmission delay also allows increasing the relative system load without saturating the queue. The proposed hardware implementation can reduce the memory consumption on the receiver (or service) side of a MP communication by leaving out connection state information. The connectionless protocol also simplifies communication management as connections do not have to be setup up by a central entity, but a client can simply send a message to a known receiver without the receiver being aware of the sender beforehand. Neither receivers nor

5 Message Passing Hardware

senders have to worry about connection aliveness and implement additional connection keep-alive protocols.

6 Summary

In this work, the attempt was made to grasp the scope that has to be considered when designing efficient resource management for embedded systems. Optimizing one part of the system to address one hotspot may result in increased efficiency. However, a big improvement in efficiency can only be achieved when regarding multiple hotspots at once. It is then possible to apply one optimization that only shifts the bottleneck and creates another hotspot. Applying a second optimization on the new hotspot could potentially result in greater efficiency improvement than a direct solution for the original hotspot might have brought.


In this work, for example, in Chapter 2 a solution was found to the problem of processing networks that don't allow certain types of parallelism. The introduction of task emitters for few static data flow (SDF) actors allows for more potential parallelism to be exploited. The increased amount of management overhead shifts the hotspot from the denied potential parallelism to the management processing. Without any further investigations, the done measures would be unsuitable for a real application because it would limit the number of usable worker processing elements (PEs) to very few. However, Chapter 3 picks up this increased and variable overhead ratio and presents a solution which ensures efficient utilization of all PEs. This way, the relevance of the results from Chapter 2 are reinforced for a broader range of applications. The key to this success is the layer crossing handling of performance hotspots. In this case, from the application layer specific parallelism to the runtime environment (RTE) layer-specific overhead handling.

The distribution of the overhead processing described in Chapter 3 relies on an efficient and fast message passing (MP) functionality. Without that, the

quick signaling between client and server and synchronization between servers would make the efficiency of such a distributed management limited. Because of that, Chapter 5 examined the possibility of implementing a MP solution of multi processor system on chips (MPSoCs) with a focus on resource efficiency, fast message delivery, and low overhead, meaning that the initiation of a message sending is cheap for the local central processing unit (CPU). In this chapter, a MP hardware implementation is presented to be faster and more resource-efficient than a remote direct memory access (RDMA) based software solution. It is an excellent example of a hotspot optimization through improved implementation in lower levels. The symptom (the observable hotspot) is the slow execution of MP operations provided by the driver layer. Instead of limiting the number of MP usages or implementation tweaks in the driver layer itself—which would probably be not very beneficial—the solution is implemented in the layer below, the hardware layer. Here the needed increases in efficiency can be achieved without constraining the usage of MP but rather embrace the increased needs.

Finally, in Chapter 4 another hardware improvement is introduced, targeting constant high throughput data streams. Data streams are necessary to implement the performant execution of process networks. The benefits of a custom first in first out (FIFO) hardware as presented in that chapter are reduced overhead for the CPU and efficient usage of single ported memory banks for the PE local memory. Implementing the whole FIFO buffer handling in hardware moves the protocol processing away from the CPU freeing resources for the payload effort. The proposed FIFO controller can distribute memory accesses from multiple FIFO users evenly across several memory banks. This results in very high utilization of the banks' interfaces allowing single ported memory with only marginal bandwidth losses. Since the single ported memory banks consume roughly half the chip area, the amount of available on-chip memory can be increased tremendously.

Considering the targeted application—mobile signal processing—the efficient usage of the available hardware is paramount. With the abstraction of the application to a stream of SDF graphs, the need for efficiency becomes a scheduling problem. Applying static scheduling, while being almost optimal in terms of makespan, has an inefficient resource consumption. The necessary step towards dynamic scheduling is followed by a set of problems that can only be dealt with by adopting the system at every possible layer. However, when addressing these problems one by one, this work shows that dynamic scheduling can be done without increasing makespan too much. On the contrary, the

increased resource efficiency of dynamic scheduling results in higher performance in terms of total work done per time. It is, in the end, the better choice for base-station signal processing, reducing the need for overprovisioning of hardware resources. 

Abbreviations

3GPP	3rd generation partnership project
API	application programming interface
ASIC	application specific integrated circuit
ASIP	application specific integrated processor
CPU	central processing unit
DMA	direct memory access
DSP	digital signal processor
FIFO	first in first out
FPGA	field programmable gate array
FSM	finite state machine
HW	hardware
IO	input/output
ISA	instruction set architecture
KPN	kahn process network
LTE	Long Term Evolution

Abbreviations

MoC	model of computation
MP	message passing
MPSoC	multi processor system on chip
Nif	network interface
NoC	network on chip
OS	operating system
PC	personal computer
PCB	printed circuit board
PE	processing element
QoS	quality of service
RDMA	remote direct memory access
RISC	reduced instruction set computer
RTE	runtime environment
RTL	register transfer level
SDF	static data flow
SoC	system on chip
SRAM	static random access memory
TTI	transmission time interval

Symbols

Unit	Name	Type	Description
scl	simcycle	time	simulation time slice
cyl	cycle	time	register transfer level (RTL) cycle
B	byte	data	
bit	bit	data	
tok	token	data	native word length
tpc	token per cycle	data rate	

Variables

Name	Symbol	Unit	Description
effort	e	—	computational work for a CPU
speedup	S	—	
size	s	B or bit	
delay	d	cyl or scl	
ratio	r	—	
	r_{over}	—	overhead/payload effort
	r_{manager}	—	manager/worker processing power
load	ρ	—	
service rate	λ	cyl ⁻¹ , scl ⁻¹	e.g. requests handled by service
arrival rate	μ	cyl ⁻¹ , scl ⁻¹	e.g. requests issued to service
distance	h	—	number of hops in NoC
bandwidth	b	tpc	data rate e.g. of a memory bank
area	A	μm ²	chip area

Abbreviations

Publications

1. S. Haas et al. "An MPSoC for Energy-Efficient Database Query Processing". In: *Design Automation Conference (DAC)*. Austin/Texas, USA, June 2016. DOI: 10.1145/2897937.2897986
2. M. Völp et al. "The Orchestration Stack: The Impossible Task of Designing Software for Unknown Future Post-CMOS Hardware". In: *International Workshop on Post-Moores Era Supercomputing (PMES)*. Salt Lake City, Utah, USA, Nov. 2016
3. Sebastian Haas et al. "A Heterogeneous SDR MPSoC in 28 nm CMOS for low-latency wireless applications". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM. 2017, p. 47
4. J. Castrillon et al. "A Hardware/Software Stack for Heterogeneous Systems". In: *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)* (Nov. 2017). DOI: 10.1109/TMSCS.2017.2771750
5. Mattis Hasler et al. "Slicing FIFOs for on-chip memory bandwidth exhaustion". In: *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE. 2018, pp. 513–516
6. R. Wittig et al. "Queue Based Memory Management Unit for Heterogeneous MPSoCs". In: *Design Automation and Test in Europe (DATE)*. Florence, Italy, Mar. 2019. DOI: 10.1007/978-3-030-27562-4_16
7. R. Wittig et al. "Statistical Access Interval Prediction for Tightly Coupled Memory Systems". In: *IEEE Symposium on Low-Power and High-Speed Chips*

- (COOLCHIPS). Yokohama, Japan, Apr. 2019. DOI: 10.1007/978-3-030-27562-4_16
8. R. Wittig et al. "Probabilistic Models for Off-Line Arbiters in Embedded Systems". In: *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. Cuzco, Peru, Oct. 2019. DOI: 10.1007/978-3-030-27562-4_16
 9. M. Hasler et al. "A Hybrid Execution Approach to Improve the Performance of Dataflow Applications". In: *International System-on-Chip Design Conference (ISOCC)*. Jeju, South Korea, Oct. 2019
 10. G. Fettweis et al. "5G-and-Beyond Scalable Machines". In: *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. Cuzco, Peru, Oct. 2019
 11. Mattis Hasler et al. "Slicing FIFOs for on-chip memory bandwidth exhaustion". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 67.2 (2019), pp. 441–450
 12. M. Hasler et al. "Balancing Dynamic Scheduling Overhead to Maximize SDF Performance". In: *2020 2nd 6G Summit (6G SUMMIT)*. Levi, Finland, Finland, Mar. 2020, pp. 1–5. DOI: 10.1109/6GSUMMIT49458.2020.9083734

Bibliography

- [1] S. S. A. Abbas, P. A. J. Sheeba, and S. J. Thiruvengadam. "Design of downlink PDSCH architecture for LTE using FPGA". In: *2011 International Conference on Recent Trends in Information Technology (ICRTIT)*. June 2011, pp. 947–952. DOI: 10.1109/ICRTIT.2011.5972424.
- [2] Jude Ambrose et al. "Composable Local Memory Organisation for Streaming Applications on Embedded MPSoCs". In: *Proceedings of the 8th ACM International Conference on Computing Frontiers*. CF '11. Ischia, Italy: Association for Computing Machinery, 2011. ISBN: 9781450306980. DOI: 10.1145/2016604.2016631. URL: <https://doi.org/10.1145/2016604.2016631>.
- [3] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.
- [4] Paul-Antoine Arras et al. "DKPN: A Composite Dataflow/Kahn Process Networks Execution Model". In: *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*. IEEE. 2016, pp. 27–34.
- [5] Md Rabiul Awal and MM Hafizur Rahman. "Network-on-chip implementation of midimew-connected mesh network". In: *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE. 2013, pp. 265–271.

- [6] Johannes Ax et al. "CoreVA-MPSoC: A Many-Core Architecture with Tightly Coupled Shared and Local Data Memories". In: *IEEE Transactions on Parallel and Distributed Systems* 29.5 (2018), pp. 1030–1043. DOI: 10.1109/TPDS.2017.2785799.
- [7] Fujun Bai et al. "A two-port SRAM using a single-port cell array with a self-timed write-after-read control scheme to save 47% area & 63% standby power". In: *ASIC (ASICON), 2017 IEEE 12th International Conference on*. IEEE, 2017, pp. 426–428.
- [8] Saehee Bang et al. "Implementation of LTE system on an SDR platform using CUDA and UHD". In: *Analog Integrated Circuits and Signal Processing* 78.3 (Mar. 2014), pp. 599–610. ISSN: 1573-1979. DOI: 10.1007/s10470-013-0229-1. URL: <https://doi.org/10.1007/s10470-013-0229-1>.
- [9] Luca Benini and G De Micheli. "Networks on chips: A new SoC paradigm". In: *Computer-IEEE Computer Society-* 35 (2002), pp. 70–78.
- [10] Shuvra S Bhattacharyya, Ed F Deprettere, and Bart D Theelen. "Dynamic dataflow graphs". In: *Handbook of Signal Processing Systems*. Springer, 2013, pp. 905–944.
- [11] Bruno Bodin et al. "Fast and efficient dataflow graph generation". In: *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems, SCOPEs 2014* (June 2014). DOI: 10.1145/2609248.2609258.
- [12] Jani Boutellier, Shuvra S Bhattacharyya, and Olli Silvén. "A low-overhead scheduling methodology for fine-grained acceleration of signal processing systems". In: *Journal of Signal Processing Systems* 60.3 (2010), pp. 333–343.
- [13] Robert Cahn. *Wide area network design: concepts and tools for optimization*. Morgan Kaufmann, 1998.
- [14] Nicholas P Carter et al. "Runnemed: An architecture for ubiquitous high-performance computing". In: *High Performance Computer Architecture, 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 198–209.
- [15] Aaron G Cass et al. "Logically central, physically distributed control in a process runtime environment". In: *University of Massachusetts, Computer Science Department, Amherst, MA, Technical Report UM-CS-1999-065* (1999).
- [16] J. Castrillon et al. "A Hardware/Software Stack for Heterogeneous Systems". In: *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)* (Nov. 2017). DOI: 10.1109/TMSCS.2017.2771750.

- [17] Y. N. Chang. "An efficient VLSI architecture for normal I/O order pipeline FFT design". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 55.12 (2008), pp. 1234–1238.
- [18] Weijia Che and Karam S. Chatha. "Scheduling of Synchronous Data Flow Models on Scratchpad Memory Based Embedded Processors". In: *Proceedings of the International Conference on Computer-Aided Design*. ICCAD '10. San Jose, California: IEEE Press, 2010, pp. 205–212. ISBN: 978-1-4244-8192-7. URL: <http://dl.acm.org/citation.cfm?id=2133429>.2133471.
- [19] Jack B Dennis. "First version of a data flow procedure language". In: *Programming Symposium*. Springer. 1974, pp. 362–376.
- [20] K. Desnos et al. "Distributed Memory Allocation Technique for Synchronous Dataflow Graphs". In: *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. Oct. 2016, pp. 45–50. DOI: 10.1109/SiPS.2016.16.
- [21] J Eker and JW Janneck. "CAL language report language version 1.0 document edition 1". In: *Electronics Research Laboratory, University of California at Berkeley, Tech. Rep. UCB/ERL M03/48* (2003).
- [22] Joachim Falk et al. "A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications". In: *Proceedings of the 8th ACM international conference on Embedded software*. ACM. 2008, pp. 189–198.
- [23] G. Fettweis et al. "5G-and-Beyond Scalable Machines". In: *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. Cuzco, Peru, Oct. 2019.
- [24] G. P. Fettweis and E. Matus. "Scalable 5G MPSoC architecture". In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*. Oct. 2017, pp. 613–618. DOI: 10.1109/ACSSC.2017.8335414.
- [25] KAHN Gilles. "The semantics of a simple language for parallel programming". In: *Information processing 74* (1974), pp. 471–475.
- [26] Michael I Gordon and Saman Amarasinghe. "Compiler techniques for scalable performance of stream programs on multicore architectures". PhD thesis. Massachusetts Institute of Technology, Department of Electrical Engineering, 2010.
- [27] N. Grigoryan, E. Matúš, and G. Fettweis. "DF4CRAN: Dataflow Framework for Cloud-RAN Signal Processing". In: *IEEE 5G World Forum (WF-5G)*. Dresden, Germany, Sept. 2019.

- [28] N. Grigoryan, E. Matúš, and G. Fettweis. "Scalable 5G Signal Processing on Multiprocessor System: A Clustering Approach". In: *IEEE 5G World Forum (WF-5G)*. Bangalore ,India, Sept. 2020, pp. 389–394.
- [29] Nairuhi Grigoryan, Emil Matus, and Gerhard P Fettweis. "Scalable 5G Signal Processing on Multiprocessor System: A Clustering Approach". In: *2020 IEEE 3rd 5G World Forum (5GWF)*. IEEE. 2020, pp. 389–394.
- [30] William D Gropp et al. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999.
- [31] Soonhoi Ha and Hyunok Oh. "Decidable dataflow models for signal processing: Synchronous dataflow and its extensions". In: *Handbook of Signal Processing Systems*. Springer, 2013, pp. 1083–1109.
- [32] S. Haas et al. "An MPSoC for Energy-Efficient Database Query Processing". In: *Design Automation Conference (DAC)*. Austin/Texas, USA, June 2016. DOI: 10.1145/2897937.2897986.
- [33] Sebastian Haas et al. "A Heterogeneous SDR MPSoC in 28 nm CMOS for low-latency wireless applications". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM. 2017, p. 47.
- [34] M. Hasler et al. "A Hybrid Execution Approach to Improve the Performance of Dataflow Applications". In: *International System-on-Chip Design Conference (ISOC)*. Jeju, South Korea, Oct. 2019.
- [35] M. Hasler et al. "Balancing Dynamic Scheduling Overhead to Maximize SDF Performance". In: *2020 2nd 6G Summit (6G SUMMIT)*. Levi, Finland, Finland, Mar. 2020, pp. 1–5. DOI: 10.1109/6GSUMMIT49458.2020.9083734.
- [36] Mattis Hasler et al. "A Random Linear Network Coding Platform MPSoC Designed in 22nm FDSOI". In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2022, pp. 217–222.
- [37] Mattis Hasler et al. "Slicing FIFOs for on-chip memory bandwidth exhaustion". In: *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE. 2018, pp. 513–516.
- [38] Mattis Hasler et al. "Slicing FIFOs for on-chip memory bandwidth exhaustion". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 67.2 (2019), pp. 441–450.

- [39] Kai Huang, David Grunert, and Lothar Thiele. "Windowed FIFOs for FPGA-based multiprocessor systems". In: *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on.* IEEE. 2007, pp. 36–41.
- [40] Charles R Johns and Daniel A Brokenshire. "Introduction to the cell broadband engine architecture". In: *IBM Journal of Research and Development* 51.5 (2007), pp. 503–519.
- [41] James A Kahle et al. "Introduction to the cell multiprocessor". In: *IBM Journal of Research and Development* 49.4.5 (2005), pp. 589–604.
- [42] Gilles Kahn and David MacQueen. "Coroutines and networks of parallel processes". In: (1976).
- [43] George Kalokerinos et al. "FPGA implementation of a configurable cache / scratchpad memory with virtualized user-level RDMA capability". In: *2009 International Symposium on Systems, Architectures, Modeling, and Simulation.* IEEE. 2009, pp. 149–156.
- [44] George Kalokerinos et al. "Prototyping a Configurable Cache/Scratchpad Memory with Virtualized User-Level RDMA Capability". In: *Transactions on High-Performance Embedded Architectures and Compilers V.* Ed. by Cristina Silvano, Koen Bertels, and Michael Schulte. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, pp. 100–120. ISBN: 978-3-662-58834-5. DOI: 10.1007/978-3-662-58834-5_6. URL: https://doi.org/10.1007/978-3-662-58834-5_6.
- [45] H. Kee et al. "FPGA-based design and implementation of the 3GPP-LTE physical layer using parameterized synchronous dataflow techniques". In: *2010 IEEE International Conference on Acoustics, Speech and Signal Processing.* Mar. 2010, pp. 1510–1513. DOI: 10.1109/ICASSP.2010.5495504.
- [46] David G Kendall. "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain". In: *The Annals of Mathematical Statistics* (1953), pp. 338–354.
- [47] J. F. C. Kingman. *Poisson Processes.* Oxford Studies in Probability. Clarendon Press.
- [48] Jaydeep P Kulkarni et al. "5.6 Mb/mm 1R1W 8T SRAM Arrays Operating Down to 560 mV Utilizing Small-Signal Sensing With Charge Shared Bitline and Asymmetric Sense Amplifier in 14 nm FinFET CMOS Technology". In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 229–239.

- [49] E. A. Lee and S. Ha. "Scheduling strategies for multiprocessor real-time DSP". In: *1989 IEEE Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond'*. Nov. 1989, 1279–1283 vol.2. DOI: 10.1109/GLGCOM.1989.64160.
- [50] Edward A Lee and David G Messerschmitt. "Synchronous data flow". In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245.
- [51] Edward A Lee and Thomas M Parks. "Dataflow process networks". In: *Proceedings of the IEEE* 83.5 (1995), pp. 773–801.
- [52] Edward Ashford Lee and David G Messerschmitt. "Static scheduling of synchronous data flow programs for digital signal processing". In: *IEEE Transactions on computers* 100.1 (1987), pp. 24–35.
- [53] Tang Lei and Shashi Kumar. "A two-step genetic algorithm for mapping task graphs to a network on chip architecture". In: *Digital System Design, 2003. Proceedings. Euromicro Symposium on*. IEEE. 2003, pp. 180–187.
- [54] T. Limberg et al. "A fully programmable 40 GOPS SDR single chip baseband for LTE/WiMAX terminals". In: *ESSCIRC 2008 - 34th European Solid-State Circuits Conference*. Sept. 2008, pp. 466–469. DOI: 10.1109/ESSCIRC.2008.4681893.
- [55] Torsten Limberg et al. "A fully programmable 40 GOPS SDR single chip baseband for LTE/WiMAX terminals". In: *ESSCIRC 2008-34th European Solid-State Circuits Conference*. IEEE. 2008, pp. 466–469.
- [56] Yue Liu, MengMeng Cao, and Kong Jie. "A New Static Data Flow Clustering Algorithm for Task Scheduling of Irregular Mesh in NoCs Based on Complex Networks". In: *International Journal of Future Generation Communication and Networking* 9.9 (2016), pp. 181–190.
- [57] Philipp Mahr et al. "Soc-mpi: A flexible message passing library for multi-processor systems-on-chips". In: *2008 International Conference on Reconfigurable Computing and FPGAs*. IEEE. 2008, pp. 187–192.
- [58] Changwoo Min and Young Ik Eom. "DANBI: Dynamic scheduling of irregular stream programs for many-core systems". In: *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press. 2013, pp. 189–200.
- [59] Kenji Nakagawa. "On the series expansion for the stationary probabilities of an M/D/1 queue". In: *Journal of the operations research society of Japan* 48.2 (2005), pp. 111–122.

- [60] Koji Nii et al. "A 45-nm single-port and dual-port SRAM family with robust read/write stabilizing circuitry under DVFS environment". In: *VLSI Circuits, 2008 IEEE Symposium on*. IEEE. 2008, pp. 212–213.
- [61] Benedikt Noethen et al. "10.7 A 105GOPS 36mm² heterogeneous SDR MPSoC with energy-aware dynamic scheduling and iterative detection-decoding for 4G in 65nm CMOS". In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*. IEEE. 2014, pp. 188–189.
- [62] Vincent Nollet, Diederik Verkest, and Henk Corporaal. "A safari through the mpsoC run-time management jungle". In: *Journal of Signal Processing Systems* 60.2 (2010), pp. 251–268.
- [63] B. Nöthen. *Untersuchungen von Kommunikationsmechanismen in heterogenen Mehrprozessorsystemen*. Jörg Vogt Verlag, 2015.
- [64] *OpenMPI website (<https://www.open-mpi.org/>)*. Accessed: 2020-11-11.
- [65] Pier S Paolucci et al. "SHAPES:: a tiled scalable software hardware architecture platform for embedded systems". In: *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. ACM. 2006, pp. 167–172.
- [66] M. Pelcat, J. Nezan, and S. Aridhi. "Adaptive multicore scheduling for the LTE uplink". In: *2010 NASA/ESA Conference on Adaptive Hardware and Systems*. June 2010, pp. 36–43. DOI: 10.1109/AHS.2010.5546233.
- [67] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. "A hierarchical multiprocessor scheduling system for DSP applications". In: *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*. Vol. 1. Oct. 1995, 122–126 vol.1. DOI: 10.1109/ACSSC.1995.540525.
- [68] Saurabh-Kumar Raina. "FLIP: a floating-point library for integer processors". PhD thesis. École Normale Supérieure de Lyon, 2006.
- [69] Martino Ruggiero et al. "A fast and accurate technique for mapping parallel applications on stream-oriented MPSoC platforms with communication awareness". In: *International Journal of Parallel Programming* 36.1 (2008), pp. 3–36.
- [70] Martino Ruggiero et al. "Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip". In: *Design, Automation and Test in Europe, 2006. DATE'06. Proceedings*. Vol. 1. IEEE. 2006, 6–pp.

- [71] Scott Schneider et al. "Elastic scaling of data parallel operators in stream processing". In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–12.
- [72] Jason Scott et al. "Hardware/software runtime environment for dynamically reconfigurable systems". In: *ISIS 6* (2000).
- [73] Alessandro Secco et al. "Message passing on InfiniBand RDMA for parallel run-time supports". In: *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE. 2014, pp. 130–137.
- [74] Weihua Sheng et al. "FIFO exploration in mapping streaming applications onto the TI OMAP3530 platform: Case study and optimizations". In: *Embedded Multicore Socs (MCSoc), 2012 IEEE 6th International Symposium on*. IEEE. 2012, pp. 51–58.
- [75] Hayden Kwok-Hay So and Robert Brodersen. "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH". In: *ACM Transactions on Embedded Computing Systems (TECS) 7.2* (2008), p. 14.
- [76] William Thies, Michal Karczmarek, and Saman Amarasinghe. "StreamIt: A language for streaming applications". In: *International Conference on Compiler Construction*. Springer. 2002, pp. 179–196.
- [77] Philippas Tsigas and Yi Zhang. "A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems". In: *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*. ACM. 2001, pp. 134–143.
- [78] M. Völpl et al. "The Orchestration Stack: The Impossible Task of Designing Software for Unknown Future Post-CMOS Hardware". In: *International Workshop on Post-Moores Era Supercomputing (PMES)*. Salt Lake City, Utah, USA, Nov. 2016.
- [79] John Von Neumann. "First Draft of a Report on the EDVAC". In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75.
- [80] Drew Wingard. "MicroNetwork-based integration for SOCs". In: *Design Automation Conference, 2001. Proceedings*. IEEE. 2001, pp. 673–677.
- [81] Alexander Wise. *Little-JIL 1.0 language report*. Tech. rep. Technical Report 98-24, University of Massachusetts at Amherst, 1998.

- [82] R. Wittig et al. "Probabilistic Models for Off-Line Arbiters in Embedded Systems". In: *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. Cuzco, Peru, Oct. 2019. DOI: 10.1007/978-3-030-27562-4_16.
- [83] R. Wittig et al. "Queue Based Memory Management Unit for Heterogeneous MPSoCs". In: *Design Automation and Test in Europe (DATE)*. Florence, Italy, Mar. 2019. DOI: 10.1007/978-3-030-27562-4_16.
- [84] R. Wittig et al. "Statistical Access Interval Prediction for Tightly Coupled Memory Systems". In: *IEEE Symposium on Low-Power and High-Speed Chips (COOLCHIPS)*. Yokohama, Japan, Apr. 2019. DOI: 10.1007/978-3-030-27562-4_16.
- [85] M. Wu et al. "Large-Scale MIMO Detection for 3GPP LTE: Algorithms and FPGA Implementations". In: *IEEE Journal of Selected Topics in Signal Processing* 8.5 (Oct. 2014), pp. 916–929. ISSN: 1941-0484. DOI: 10.1109/JSTSP.2014.2313021.
- [86] Wai Gen Yee and Ophir Frieder. "Scalable synchronization of intermittently connected database clients". In: *Proceedings of the 6th international conference on Mobile data management*. 2005, pp. 299–303.
- [87] C. Zimmer and F. Mueller. "Nocmsg: Scalable NoC-based message passing". In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2014, pp. 186–195.