Fakultät für Elektrotechnik und Informationstechnik
Technische Universität München

Efficient Inter-Task Communication in Tiled Many-Core System-on-Chip Architectures

Stefan Wallentowitz

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:

Prof. Dr.-Ing. Josef Kindersberger

Prüfende der Dissertation:

1. Prof. Dr. sc. techn. Andreas Herkersdorf
2. Prof. Dr. Hans Michael Gerndt

Die Dissertation wurde am 19.04.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 19.09.2018 angenommen.

Abstract

Multicore system-on-chip architectures are dominant in all domains, such as desktop and server computers, smartphones and embedded devices. In *tiled many-core system-on-chip* architectures a large number of processor cores are replicated in a regular structure. Tiles contain one or multiple processor cores and other resources such as memories. The inter-tile communication is based on the network-on-chip methodology. The interface between the tiles and the network-on-chip is the *network adapter*. The very basic functionality of the network adapter is the packetization of data and protocol bridging. Sometimes the network adapter implements flow control above the network level or synchronization primitives. Generally, the work related to on-chip network adapters focuses on improvements on the level of the network-on-chip. Anyhow, complex software stacks become increasingly important on the computing side of network adapters. Those stacks add for example high-level message passing which abstract from the underlying hardware, and device sharing by software tasks running on top of an operating system.

The network adapter shares similarities with network interface cards (NIC), especially in high performance computing (HPC). This thesis investigates the adoption of concepts from inter-node communication in HPC to on-chip communication and the co-optimization of the many-core architecture and programming model. The central contribution is the concept of a *Network Adapter for Message Passing (NAMP)*.

Motivated by the demands of aforementioned complex software stacks, the work focuses on features in the following areas: 1) Offloading higher-level protocol processing into the network adapter, 2) bypassing the operating system by virtualization, and 3) efficient event notification to the tasks. The NAMP addresses those areas with an offload of the message passing protocol handling into hardware, including the capability for collective communication between multiple tiles. It can be shared transparently by multiple task as a self-virtualized device. Furthermore, the NAMP concept includes a novel idea for efficient task notification, *hardware-based operating system queue manipulation (HW-OSQM)*, that allows for elimination of all overhead by interrupts and can be easily generalized for arbitrary devices. Finally, the thesis contributes a concept for the migration of the communication channels during task migration as an integral part of NAMP. The NAMP concept allows for flexible configuration and composability of the proposed features.

The NAMP features are evaluated with event-based simulations or analytically. For example, compared to an implementation that is assisted by an RDMA controller the NAMP offload of the message passing protocol to hardware reduces the overhead by 48% in case the message can be transferred immediately and 64% if messages need on average one re-transmission. On this baseline other features show for example a decrease in overhead of a multicast operation to 8 destinations of up to 84%. The virtualization of the NAMP interface allows for a saving of up to 62% of overhead in the sender software.

Beside the evaluation of the proposed improvements, the concept is validated with a prototype implementation. This implementation is used to validate the basic NAMP functionality with synthetic benchmarks in a cycle-accurate RTL simulation. Furthermore, the implementation is synthesized for an FPGA and an ASIC target to evaluate the impact on the hardware utilization. Five typical configurations using different sets of features are evaluated. The hardware overhead is moderate: A simple variant only adds 14,000 gates to the ASIC design, while a fully fledged NAMP takes approximately 46,000 gates. Finally, two example systems are implemented in FPGAs to demonstrate the use of NAMP in different scenarios: The first is a system with four cores in four tiles that targets at baremetal use cases, and the second is a system with 64 cores in 16 tiles that targets at flexible use cases with a full software stack.

To the best of my knowledge, work presented in this thesis is the first that comprehensively analyzes on-chip message passing with a full system stack, and contributes a configurable, scalable concept for a network adapter that enables efficient inter-tile communication in tiled many-core system-on-chip.

Zusammenfassung

Mehrkern System-on-Chip-Architekturen dominieren in allen Bereichen, wie Desktop- und Server-Computer, Smartphones und Embedded Systems. In gekachelten Vielkern-System-on-Chip-Architekturen wird eine große Anzahl von Prozessorkernen in einer regulären Struktur repliziert. Kacheln enthalten einen oder mehrere Prozessorkerne und andere Ressourcen wie Speicher. Die Kommunikation zwischen den Kacheln basiert auf Network-on-Chip. Die Schnittstelle zwischen den Kacheln und dem Network-on-Chip ist der Netzwerkadapter. Allgemein übernimmt der Netzwerkadapter die Paketierung von Daten und Protokollüberbrückung. Manchmal implementiert der Netzwerkadapter auch Flusskontrolle oberhalb der Netzwerkebene oder Synchronisierung zwischen Kacheln. Im Allgemeinen konzentrieren sich Beiträge im Bereich der on-chip Netzwerkadapter auf Verbesserungen auf der Ebene des Network-on-Chip. Gleichwohl werden komplexe Software-Stacks innerhalb der Kacheln immer wichtiger. Diese Stacks beinhalten beispielsweise Message Passing auf höheren Protokollschichten und die gemeinsame Verwendung der Hardware durch Software-Tasks, die auf einem Betriebssystem ausgeführt werden.

Der Netzwerkadapter weist Ähnlichkeiten mit Netzwerkschnittstellenkarten (Network Interface Cards, NIC) auf, insbesondere im Hochleistungsrechnen (High Performance Computing, HPC). Diese Arbeit untersucht die Übernahme von Konzepten von der Kommunikation zwischen Knoten in HPC zu On-Chip-Kommunikation und die Co-Optimierung der Vielkern-Architektur und des Programmiermodells. Der zentrale Beitrag ist das Konzept eines Netzwerkadapters für Message Passing (NAMP).

Motiviert durch die Anforderungen der oben genannten komplexen Software-Stacks konzentriert sich die Arbeit auf Features in folgenden Bereichen: 1) Entlastung der übergeordneten Protokollverarbeitung durch die Netzwerkadapter, 2) Umgehung des Betriebssystems durch Virtualisierung und 3) effiziente Ereignisbenachrichtigung an die Applikationen. Der NAMP adressiert diese Bereiche mit einer Entlastung der Massage Passing Protokollbehandlung in Hardware, einschließlich der Fähigkeit zur kollektiven Kommunikation zwischen mehreren Kacheln. Er kann transparent von mehreren Aufgaben als selbst-virtualisiertes Gerät geteilt werden. Darüber hinaus enthält das

NAMP-Konzept eine neuartige Idee für eine effiziente Applikationsbenachrichtigung, eine hardwarebasierte Betriebssystemwarteschlangenmanipulation (hardware-based operating system queue manipulation, OSQM), die den gesamten Overhead durch Interrupts eliminiert und leicht für beliebige Geräte verallgemeinert werden kann. Abschließend wird ein Konzept zur Migration der Kommunikationskanäle bei der Taskmigration als integraler Bestandteil von NAMP vorgestellt. Das NAMP-Konzept ermöglicht eine flexible Konfiguration und Zusammensetzbarkeit der vorgeschlagenen Features.

Die NAMP-Features werden mit ereignisbasierten Simulationen oder analytisch untersucht. Im Vergleich zu einer Implementierung, die von einem RDMA-Controller unterstützt wird, reduziert der NAMP-Offload des Message Passing Protokolls in Hardware den Overhead um 48% für den Fall, dass die Nachricht sofort übertragen werden kann, und 64%, wenn Nachrichten im Durchschnitt eine Wiederaufnahme benötige. Auf dieser Grundlage zeigen andere Feature beispielsweise eine Verringerung der Overheads einer Multicast-Operation auf 8 Ziele von bis zu 84%. Die Virtualisierung der NAMP-Schnittstelle ermöglicht eine Einsparung von bis zu 62% des Overheads in der Sendersoftware.

Neben der Untersuchung der vorgeschlagenen Verbesserungen wird das Konzept mit einer Prototypimplementierung validiert. Diese Implementierung wird verwendet, um die grundlegende NAMP-Funktionalität mit synthetischen Benchmarks in einer zyklusgenauen RTL-Simulation zu validieren. Darüber hinaus wird die Implementierung für ein FPGA- und ein ASIC-Target synthetisiert, um die Auswirkungen auf den Hardware-Verbrauch zu bewerten. Fünf typische Konfigurationen, die verschiedene Sets von Features verwenden, werden bewertet. Der Hardware-Overhead ist moderat: Eine einfache Variante fügt dem ASIC-Design nur 14.000 Gatter hinzu, während ein vollwertiger NAMP ungefähr 46.000 Gatter benötigt. Schließlich werden zwei Beispielsysteme in FPGAs implementiert, um den Einsatz von NAMP in verschiedenen Szenarien zu demonstrieren: Das erste ist ein System mit vier Kernen in vier Kacheln, das auf Barmetal-Anwendungsfälle zielt, und das zweite ist ein System mit 64 Kernen in 16 Kacheln und zielt auf flexiblen Anwendungsfällen mit einem vollen Software-Stack.

Meines Wissens nach sind die Beiträge, die in dieser Arbeit vorgestellt werden, die ersten, die On-Chip Message Passing mit einem vollständigen Software-Stack umfassend analysiert und ein konfigurierbares, skalierbares Konzept für einen On-Chip Netzwerkadapter beisteuert, der eine effiziente Kommunikation zwischen den Kacheln in Manycore System-on-Chip ermöglicht.

Contents

List of Figures

List of Tables

Listings

Acronyms

API	Application Programming Interface.
ASIC	Application-Specific Integrated Circuit.
CC	Collective Communication.
CMP	Chip Multicore Processor.
CPU	Central Processing Unit.
DLP	Data Level Parallelism.
DMA	Direct Memory Access.
DRAM	Dynamic Random Access Memory.
DSM	Distributed Shared Memory.
FPGA	Field-Programmable Gate Array.
FSM	Finite State Machine.
GPU	Graphical Processing Unit.
HPC	High Performance Computing.
HW	Hardware.
I/F	Interface.
I/O	Input/Output.
ILP	Instruction Level Parallelism.
IPC	Instructions per Cycle.
IPS	Instructions Per Second.
LUT	Look-Up Table (FPGA).
MCAPI	Multicore Communications API.

MPI	Message Passing Interface.
NA	Network Adapter.
NAMP	Network Adapter for Message Passing.
NIC	Network Interface Card.
NUMA	Non-Uniform Memory Access.
OSQM	Operating System Queue Manipulation.
PGAS	Partitioned Global Address Space.
PS	Protection Switching.
RDMA	Remote Direct Memory Access.
RISC	Reduced Instruction Set Computer.
RMA	Remote Memory Access.
RTL	Register Tranfer Logic.
SRAM	Static Random Access memory.
SV	Self-Virtualization.
SW	Software.

1 Introduction

The development of computer architecture took an astonishing pace in the recent decades. *Moore's "Law"* [150] – that actually was an early observation – describes how the integration density in semiconductor devices led to roughly a doubling of the number of transistors per chip every two years. Beside twice the number of transistors available for computations, shrinking transistor dimensions allowed for ever increasing clock frequencies. Hence, the performance increases of cores in the 1980s and 1990s were dominated by rapid development of two factors:

Micro-architecture improvements The goal of a good micro-architecture can be abstracted as converting the number of extra transistors into more software instructions completed per clock cycle. Starting from execution of one software instruction after the other, further improvements are achieved by exploiting parallelism and speculation. Most importantly, *instruction level parallelism* and *data level parallelism* of the software have enabled significant improvements.[1]

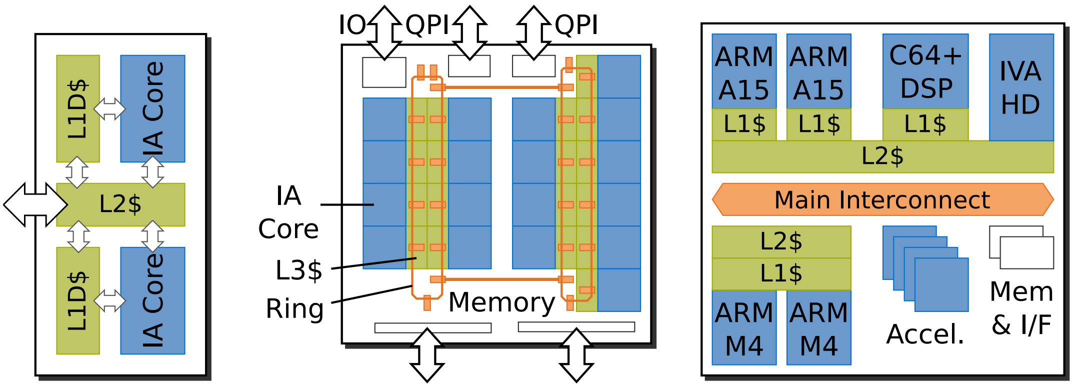
Aggressive frequency scaling With shrinking transistor dimensions the frequency of a circuit can be increased. Beside this, pipelining techniques in the micro-architecture further allow increasing processor frequencies. The number of clock cycles per second hence increased significantly.

Roughly speaking, those were the two major sources of steady performance increases, where processor performance can be sketched as executed software instructions per second (IPS):

$$\frac{instructions}{second} = \frac{instructions}{cycle} \times \frac{cycles}{second}$$

The first component, instructions per cycle (IPC), is influenced by improvements of the micro-architecture, for example by exploiting parallelism and speculation. The second part is the frequency of the processor core.

[1] This is of course an abstracted view on the topic of computer architectures. For a deeper insight into this fascinating topic I refer the reader to the standard text book of Hennessy and Patterson [100].



(a) Intel Core 2 Duo, 2006 [163]

(b) Xeon E7 8890 v3, 2015 [151]

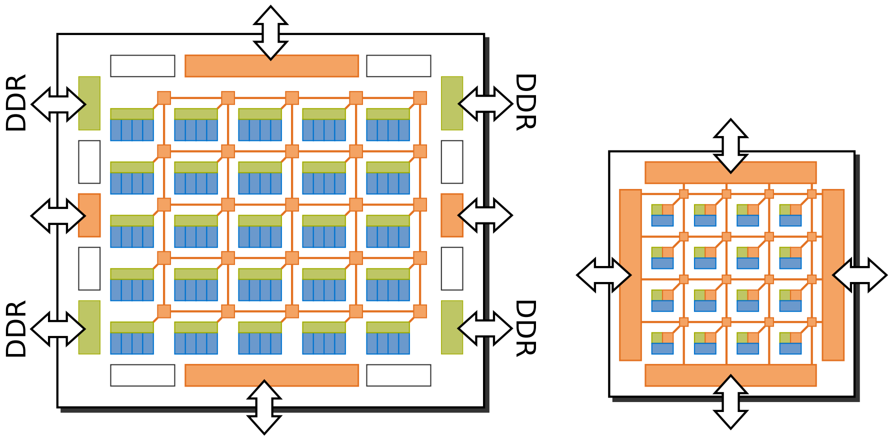
(c) TI OMAP 5430, 2013 [202]

Figure 1.1: Evolutionary milestones of multi-core architectures

Around the year 2000 further scaling both of those factors became increasingly harder. As the power dissipation cubically grows with the frequency, the "power wall" started limiting further frequency increases beyond 3 GHz. Despite the vast amount of extra transistors, micro-architecture improvements rapidly stopped delivering much extra instructions per cycle.

A solution to this problem is to spend the extra transistors on replicating the processor core. Such *multi-core system-on-chip* or *chip multi-core processors (CMP)* allow execution of multiple instruction streams in parallel. For N processor cores, the chip executes N-times the number of instructions per cycle. Scaling back the frequency to reduce the power consumption, multi-core system-on-chip are thus capable of delivering increasing performance with lower power consumption. Figure 1.1 shows a rough overview of common patterns that have become dominant in the evolution of commodity multi-core system-on-chip since then [212]: early dual core processors (Figure 1.1a), today's desktop and server processors (Figure 1.1b) and heterogeneous system-on-chip in embedded systems and smartphones (Figure 1.1c).

Processor cores can be replicated in a scalable way. The methodology of packet-based network-on-chip has been a key enabler of scalable *many-core system-on-chip* that integrate hundreds or even a thousand simple *reduced instruction set cores (RISC)* in massively parallel platforms (see Table 2.1 for an extensive overview). The cores, local memory and peripherals are grouped in *tiles* and connected via a network-on-chip in a regular structure. This organization is referred to as *tiled many-core system-on-chip*. There are two classes of tiled many-core system-on-chip: *Tightly-coupled* many-core



(a) EZ-Chip Mx-100, 2015 [63]

(b) Adapteva Epiphany-III 16-core, 2013 [2]

Figure 1.2: Commercial many-core system-on-chip

system-on-chip can be described as descendants of the Transputer [222]. The cores directly move small data items to their next neighbors with a small latency. This works best with algorithms often described as "embarrassingly parallel" from application domains such as media processing or machine learning. Graphical processing units (GPU) are widespread examples for such platforms too. Another class of many-core system-on-chip can be described as *loosely-coupled*: The communication is between any tiles in the system and not only neighbors and generally asynchronous. Two commercial many-core system-on-chip from this class are sketched in Figure 1.2. The Mellanox Tile-Gx platform (formerly Tilera and EZ-Chip) [63] is prominently deployed in cellular base station and network routers, while the Adapteva Epiphany platform [2] targets embedded systems.

So, why not simply replicate thousands of simple, low power processor cores? The answer is quite straight-forward: Because they have to be programmed efficiently. The sequential processor performance (instructions per cycle) is much better for highly optimized, larger processor cores. Hence, the software must be parallel to benefit from the parallel execution. Some algorithms are embarrassingly parallel, making this task easier, but generally the possible degree of parallelization is limited by the sequential parts of the software ("Amdahl's law"). Beside the software parallelization – which is beyond the scope of this thesis – the different parts ("tasks") of the software need to communicate. This synchronization and data exchange between tasks can quickly become the bottleneck.

	HPC (Infiniband)	On-Chip Communication
CPU Clock Rate	2 GHz to 3 GHz	100 MHz to 2000 MHz
Memory size at node	> 4 GB	8 kB-1 MB
Protocol Robustness Requirements	High	Low
Latency	10 µs (20k cycles)	20 ns to 400 ns (10-100 cycles)
Bandwidth	1,000-16,000 MB/s	400-32,000 MB/s

Table 1.1: Communication comparison

The problem of efficient *inter-tile communication* becomes more dominant with the increasing number of cores, which are grouped into tiles. In this thesis I focus on loosely-coupled many-core system-on-chip which are comparable to multi-computer networks. Anyhow, important communication conditions are compared in Table 1.1. It can be observed that on-chip communication is much more performant and easier to handle due to two orders of magnitude difference in the latency and the significant difference in the achievable bandwidth. Furthermore, on-chip communication needs less attention to physical and protocol robustness as the error probability is much lower on a chip (although increasingly important as discussed later).

It can therefore be concluded that on-chip communication generally has different conditions and capabilities. Architecture and message passing protocols from multi-computer systems can be transferred to tiled many-core system-on-chip, but need conceptual adaption to the changed boundary conditions.

There are generally two methods of communication between software tasks running on different processors: *Shared memory communication* implicitly keeps data coherent in the memory hierarchy. Beyond that, synchronization primitives are required, for example locks. Shared memory platforms face scalability issues with rising number of processors. *Message passing communication* is favorable instead for a large number of processor cores: The communication between the software tasks is explicit by exchanging data items in messages.

The structure of tiled many-core system-on-chip favors message passing due to the macro-architecture organization. They can therefore be seen as *supercomputers-on-a-chip*. While there are certain similarities, the boundary conditions between them are different with respect to latencies, bandwidth and the macro-architecture at each node[2].

[2] for a discussion of the differences see the respective section in [102]

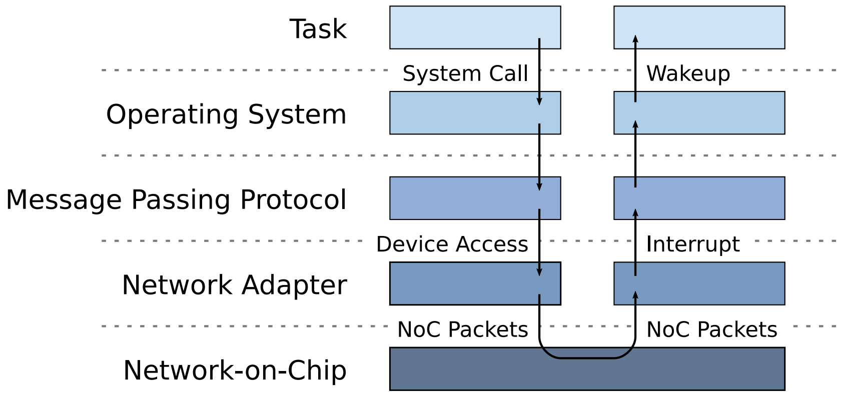
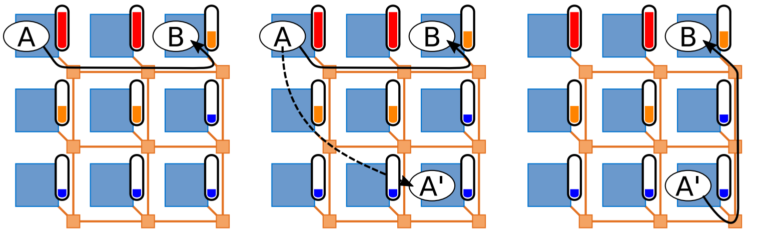


Figure 1.3: The message passing stack

1.1 Problem Statement

The bridge between computation resources and communication resources is the *network adapter (NA)*. A variety of network adapters are found in commercial products and in research nowadays, but their optimization mostly focuses on the network-on-chip interface. On the tile side they are bound to simple use cases, for example without efficient device sharing. Beside that message passing can be implemented with those network adapters, but only a few research proposals consider a network adapter that implements parts of the protocol handling in hardware to improve performance.

Figure 1.3 shows the software stack including an operating system. I anticipate a further integration of such complex stacks in many-core system-on-chip. As depicted in Figure 1.3 the message transfer may start from the task which runs in the userspace. The message gets handled by the operating system which interacts with the message passing protocol. Via memory mapped I/O the message is then send to the network adapter, which generates the network-on-chip packets. On the receiving side the network-on-chip packets are received by the network adapter which processes them and forwards the messages to the message passing protocol in the operating system, usually by an interrupt. The operating system then transfers the message to user space and wakes up the task. Although not all scenarios may involve the entire stack, it will become more important. When considering this whole stack it becomes apparent that not only the physical transfer of messages contributes to the end-to-end latency and throughput of messages, but that the entire stack and the interfaces between the layers need to be efficient.



(a) Overheated tile **(b)** Migrate task **(c)** Switch communication channel

Figure 1.4: Task migration as a method for thermal management

State-of-the-art network adapters with limited hardware capabilities impose a lot of overhead in software processing in this whole process. It is therefore desirable to reduce this overhead.

While the system performance is significantly improved by such techniques another problem of inter-task communication arises in future system-on-chip: System dependability becomes an issue due to the ever shrinking technology dimensions. Power dissipation densities and variability exposures lead to soft errors, device aging and thermal hotspots. Technologies have been introduced to manage power, dark silicon (power off parts of a chip) and temperature. Thermal hotspots can occur due to power imbalances. One common approach to mitigate thermal hotspots in tiled many-core system-on-chip is task management and task migration. As sketched in Figure 1.4, the basic idea is to migrate a task either pro-actively or as a result of measured errors at thermal hotspots. Once such a situation is detected (Figure 1.4a), the critical task gets migrated by the runtime system (Figure 1.4b). It is important that the migration occurs transparently and with minimal overhead.

1.2 Contribution

Motivated by the lack of efficient network adapters for the increasingly complex software stacks in tiled many-core system-on-chip architectures, this thesis presents a holistic approach to improve the application end-to-end communication in on-chip message passing. The contributions of this thesis are summarized in the following.

The key contribution of this work is the **concept of a network adapter for efficient inter-tile communication**: The *Network Adapter for Message Passing (NAMP)*. The concept targets the entire communication stack. Known approaches from multi-computer systems, such as programmable network interface cards (NICs), need adoption or be re-thought with the changed boundary conditions. Specifically the following proposals are developed, evaluated and validated:

Hardware offloading of the progress engine The *progress engine* is the central part of the protocol handling that performs the protocol actions. While the idea to offload it to a programmable network interface card is well-established in multi-computer systems, a full hardware implementation is viable under the boundary conditions of on-chip communication. Related work is limited in this field, because most of the research has focused on the network-on-chip. Known similar approaches [52, 110] are very limited in the supported communication types and are only suitable for static communication relations. The concept of a NAMP presented in this thesis instead is flexible with respect to the communication types (connection-less and connection-oriented). Beyond that it is suited to handle *mixed criticality* workloads of varying data sizes between dynamic or static communication partners.

Collective Communication An important feature of message passing in high performance computing is collective communication. The four functions multicast, scatter, gather and reduction enable efficient workload sharing between multiple tasks. While only multicasting is addressed by some network-on-chip implementations, efficient support for all four operations on the level of the message passing protocol by the presented *NAMP-CC* yields significant performance boosts.

Communication Virtualization The concept of self-virtualization recently gained a lot of traction in computer systems. In the scope of this work the goal is to share the network adapter between multiple user tasks transparently. For that the NAMP provides user tasks the exclusive view of a unique network adapter via virtual interfaces by *Self-Virtualization (NAMP-SV)*. As the state-of-the-art approaches require an operating system to multiplex access to the network adapter, the NAMP-SV is a novel concept in the context of on-chip inter-task communication and provides significant performance improvements over traditional approaches.

Efficient Event Signaling There are two traditional ways of notifying a software task of hardware events, such as the completion of a message send operation or the arrival of a new message: polling and interrupts. Both these approaches have significant drawbacks with respect to the common software performance. Approaches like interrupt coalescing [113] only soften this performance impact. This thesis presents *Hardware-based Operating System Queue Manipulation (HW-OSQM)*, a generic concept for transparent, zero overhead event signaling to kernel and user tasks. While such events can be manyfold, the focus in this thesis is on the tight integration with NAMP for task wake-up on message arrival or availability of resources.

Dependable Communication Migration Finally, the NAMP concept provides support for the migration of communication channels. This concept is applied to pro-active task migration to mitigate thermal hotspots. The derived **hardware-assisted on-chip protection switching (NAMP-PS)** supports task migration efficiently by transparent migration of communication relationships during and around the migration.

Finally, the NAMP concept is transformed into a configurable, scalable **reference implementation** of NAMP. The NAMP module can be configured at design time to include the aforementioned features independent from each other. This reference implementation (i) serves the *validation of the concept*, (ii) delivers a real world device for *synthetic performance benchmarks* on different data points, and (iii) is used for the *evaluation of the hardware overhead* introduced by the new functionalities and describes the trade-offs. Five *typical configurations* are presented that serve common usage scenarios of the NAMP. Finally, two FPGA-based example platforms are presented.

1.3 Organization

The contributions of the thesis are reflected in the thesis structure. It is organized as follows: Chapter 2 presents further technical context of this work and outlines the necessity for a network adapter for efficient message passing. It provides an overview of state-of-the-art in a broader context and closely related to this work. In Chapter 3 the NAMP is presented and evaluated. Chapter 4 describes the actual implementation of the NAMP. Finally, Chapter 5 summarizes the findings and concludes this thesis. It furthermore gives an outlook of potential for future work, extensions and optimization.

2 State-of-the-Art

This chapter gives technical background and motivation of the work presented in this thesis. I briefly present the reasons and challenges of the advent of many-core system-on-chip in Section 2.1, closing with a brief survey on research and industrial many-core system-on-chip in Section 2.1.3. After that the general space of inter-task communication is outlined in Section 2.2. Section 2.3 then segues from a more general presentation of the background into the most relevant related work. Section 2.4 provides a classification of network adapters from academic literature and commercial platforms. Based on this classification I position the work presented in this thesis and elaborate on the key differences and improvements compared to the state-of-the-art. The concepts for acceleration of message passing that are covered in the following Section 2.5 are only insufficiently covered by work on on-chip message passing, but widely deployed in other contexts. Hence, I describe common approaches that are for example known in computer networking, and relate them to the conceptual adaption in the proposed network adapter. Finally, Section 2.7 concludes this chapter with a summary.

2.1 Evolution and Trends of Many-Core System-on-Chip

This work was introduced with a brief overview of the challenges that the development of single core processor faced around 2000. The relevant background will be elaborated in the following.

Figure 2.1 shows a quantitive comparison of Intel's processor cores over time. Basically it can be observed that until the year 2005 the number of transistors ("Moore's Law") and the processor frequency increased exponentially. But the problem is that the frequency cubically increases the power dissipation. The critical power density is also depicted in Figure 2.1. Dennard et al. [58] observed that voltage and current can be reduced with the shrinking of the transistors. Therefore, it was possible to operate a circuit at a higher frequency with the same power. But despite this so called *Dennard Scaling*, the leakage current and the threshold voltage become more important, which can be seen as

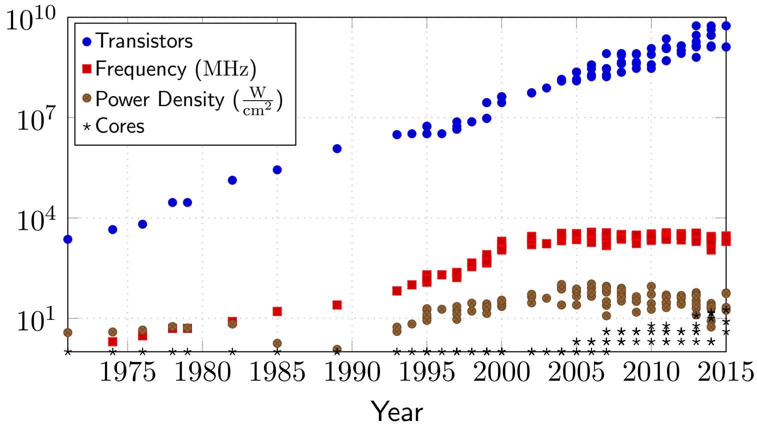


Figure 2.1: Moore's Law: Data points for Intel CPUs over time[1]

the operational baseline of the processor. With the ever increasing transistor densities, the power density hence increases and there is a practical limit of processor frequencies.

Around the year 2000 the power density reached the value equally to inside the nozzle of a rocket. Cooling a chip at this power density becomes critical and a further increase would make it even impossible. Hence, the frequency scaling inevitably had to stop and power management techniques became increasingly important. Thereby, one important driver of the performance increases ceased and also the improvement of the micro-architecture was decreasing. It became a lot harder to turn extra transistors into an extra performance improvement: the ratio of performance increase to required extra chip area dropped significantly. For example, Müller et al. [152] evaluated the area overhead of the Tomasulo scheduler [205] for out-of-order execution to nearly 100%, while the performance increase is around 15% on average.

Finally, both micro-architecture improvements for sequential code with data-level parallelism and aggressive frequency scaling ended effectively around the year 2000 and *thread-level parallelism (TLP)* became the dominant way to turn extra transistors into performance. TLP can be exploited with multi-thread processor cores, where the sequential instruction streams of two or more software threads are executed interleaved. But there are certain limits with the shared resources of the processor core. Actually, the TLP is nowadays exploited with at maximum two to eight hardware-supported threads. Hence, the most efficient way of exploiting TLP is replication of the processor cores.

[1] You can find the raw data at `http://github.com/wallento/mooreandmore`. Note that the drop in the power density in the early 1980s results from the transition from bipolar logic to CMOS logic.

The effects are best summarized by the chief architect for Sun Microsystems' Scalable Systems Group, Marc Tremblay: "We could build a slightly faster chip, but it would cost twice the die area while gaining only 20 percent speed increase"[79]. An overview of the efforts of the major seminconductor companies to switch multi-core processors for the desktop and server market is well summarized by Geer [79].

The basic idea of a multi-core processor is that multiple simpler cores can deliver the same computational performance as a single, complex core at a lower operating frequency. An algorithm may be "embarassingly parallel", meaning it is in itself parallel and has limited dependencies between the parallel parts. On the other hand a software program may not allow any parallelism. This trade-off was formalized by Amdahl [7] as *Amdahl's law*: Each program can be split in a parallel part (p) and a sequential part (s). The maximum speedup S of a program running on N processor cores is then limited as $\frac{1}{S} = s + \frac{p}{N}$, which means that the sequential part always limits the performance gain. The serial parts of a program are one part of the problem. But other effects such as thread imbalance, synchronization overhead or the thread management often limit the achievable multi-core speedup as for example analyzed by Fuerlinger and Gerndt [76]. Such limitations and practical problems are key in answering the question why multi-core processors do not simply contain hundreds of simple, power-efficient processor cores. One solution to the aforementioned problems is to run multiple programs in parallel, but the number of parallel programs is often limited. Summarizing, it is important that programmers and their tools are capable of efficiently exploiting the parallel parts of their programs. If they are able to do so, they can benefit significantly from multi-core processors.

Desktop computers and server computers are nowadays driven by the trade-off between single-core performance and the power advantages of multi-core. The number of cores steadily increases, but even single-threaded programs must execute equally good or better between the generations. Even in the smartphone market the number of processor cores steadily increases and as of early 2018 the system-on-chip of the flagship devices contain eight cores.

Anyhow, computer systems composed of many processing nodes have been deployed for several decades already: In High Performance Computing (HPC, also supercomputing) processing resources are connected at a computer level to large systems of up to ten million processor nodes [82]. Such HPC clusters of computers are perfectly suited for compute intense applications. While the interconnect of such clusters has reached multiple Gigabits from node to node with improved latency, they still favor applications that can be partitioned to independent computation nodes at a certain granularity.

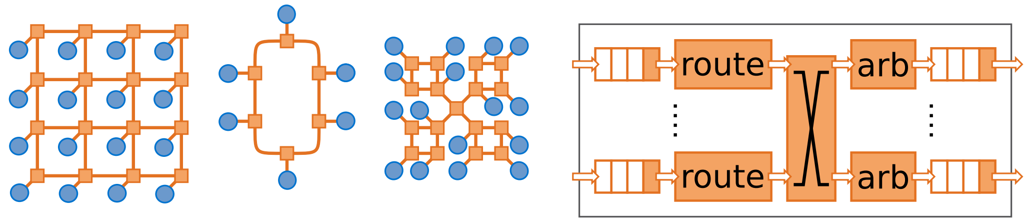
In the 1985 the Transputer has been introduced as a processing platform for parallel programs [222]. The Transputer contains processing nodes that are connected to four neighbors and allows to exchange small items between neighbors with low latency. So it is for example possible to process a data item, then push it out to the next neighbor and in the next cycle process another data item. This *tightly-coupled* platform configuration between multiple Transputer chips on a PCB favors communication-intense parallel processing.

Those room-scale or PCB-scale concepts have been transfered to system-on-chip even before general purpose processors were "forced" by the limitations of Moore's law. The general idea is to integrate many simple, power-efficient processing cores.

Currently, the most commercially successful approach of such massively parallel system-on-chip platforms are *Graphics Processing Units (GPU).* The graphics output of desktop computers has gained a significant importance not only for gaming and video output, but also for tasks as design, architecture, etc., and an improved user experience. GPUs execute are massively parallel operations, e.g., applying filter to multiple pixels or processing basic objects in parallel. GPUs are hence designed specifically to be suited for such tasks. The recent Nvidia Pascal GP100 GPU for example includes 3840 processing cores and 240 specialized texture units [158]. Due to their high processing power the use of GPUs are nowadays not limited to image and video processing. Instead they have become a more generic computing platform too [127, 156], often referred to as *General Purpose GPUs (GPGPU).* Nowadays, GPUs are even integrated into the main SoC, and they are used for computing intense, but embarrasingly parallel applications such as deep learning [228]. GPGPUs have a very regular structure and cores are grouped into clusters. Multiple of such clusters are connected with an efficient interconnect.

The concept of massively parallel processing has been generalized in multiple platforms over the last 15 years (see Section 2.1.3 for an extensive survey of prominent many-core platforms). The basic idea is massive replication of simple processing elements with an efficient interconnect. Such platforms are best suitable for embarrassingly parallel application, but can also execute multiple dynamic applications. Commercially, well established examples are graphics and video processing, communication devices such as cellular base stations and deep learning.

In the following I briefly introduce the specifics of such many-core system-on-chip and then present a brief survey of academic and commercial implementations of many-core system-of-chip.



(a) Network-on-chip topologies connecting the endpoints (blue) via point-to-point connections between routers (orange).

(b) Basic network-on-chip router anatomy.

Figure 2.2: Basic NoC components

2.1.1 Network-on-Chip

The fundamental technology that enables many-core system-on-chip are so called network-on-chip. The basic idea is to replace "ad-hoc global wiring structures" and "facilitate modular design" [53]. Network-on-chip are mostly packet-based networks of routers that are connected by point-to-point connections. In their original work, Dally and Towles [53], Hemani et al. [97] and Benini and Micheli [21] outline the most important aspects and research challenges of the network-on-chip paradigm:

- Network-on-chip improves the structure of communication by making it an orthogonal aspect of a design contrary to wiring between parts of the design.
- With network-on-chip architectures the timing improves because the communication paths are segmented. Contrary to different wire lengths crossing the entire chip, short wires enable higher clock speeds.
- System-on-chip platforms based on the network-on-chip paradigm are better composable and provide more modularity due to the clear and defined interfaces of a network-on-chip implementation.
- The network-on-chip paradigm enables a layered approach similar to OSI for large scale networking. While the traversal of a packet is managed by the actual network, complex protocols can be built on top of it.

A network-on-chip is built of the following basic components:

Network Adapters They are the bridge between the network-on-chip and the computational resources (processor cores, memory, I/O blocks, etc.).

Routers Similar to off-chip, large-scale networks those packets traverse the netwrok through routers. At arrival at a router input port, the output port is calculated and then forwarded in that direction (distributed routing), or the packet is forwarded based on routing information in the header (source routing).

Topology The topology describes how routers are distributed and connected (see Figure 2.2a). Popular topologies are for example mesh, ring, star and hybrid combinations of them.

Packets Communication endpoints exchange control information and data in the form of standardized packets.

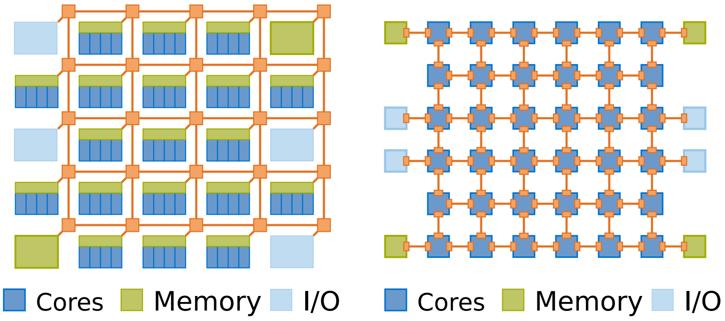
The essential research challenge that arose for network-on-chip architectures was how to map the chip design with custom point-to-point wires to this new paradigm. The approaches are essentially highly optimized network-on-chip routers as for example proposed by Bertozzi and Benini [26]. One important property needed for a variety of applications is the different handling of guaranteed services and best effort services as emphasized by Goossens et al. [86]. To maximize performance, the design of a network-on-chip for application-specific and domain-specific system-on-chip has been in research focus [27, 86]. With AMBA AXI an interface standard that favors implementation with network-on-chip architectures has become the standard for system-on-chip design [22].

While the presented approaches target at optimizing the network-on-chip to a certain application or domain, the 2D mesh and derivations of it are popular topologies for general purpose network-on-chip research [195] and favorable for scalable massively parallel many-core system-on-chip [91].

This work focuses on such general purpose, scalable many-core system-on-chip architectures. The network-on-chip paradigm is a huge research topic and there have been manyfold approaches to optimize the throughput on the level of the network-on-chip. The work presented in this thesis instead focuses on the layers above, specifically where the network-on-chip itself interfaces the computational resources: the network adapter. This network adapter will be in the focus of the thesis. More detailed discussions of network-on-chip research and directions in general are given for example by Bjerregaard and Mahadevan [29] and Owens et al. [161].

2.1.2 Many-Core Platform Organization

Resources are commonly organized as *tiles* in many-core system-on-chip. Each tile contains either one or more cores, other procesing elements, memory, I/O device etc. Tiles



(a) Loosely-coupled

(b) Tightly coupled

Figure 2.3: Tiled manycore system-on-chip integration variants

group resources in a hierarchical manner. All wiring is limited to the boundaries of the tile.

Nearly all modern multi-core system-on-chip architectures can be described as based on tiles. For example a modern Intel server multi-core processor contains multiple tiles of each a processor core and a local cache hierarchy, with all tiles connected via a ring interconnect. Anyhow, the tiled structure becomes more apparent and dominant with scaling the number of processing elements to many-core system-on-chip, occasionally also referenced as “sea of cores”. Most often tiled many-core system-on-chip are organized as meshes due to the 2D nature of a chip (see Figure 2.3) or as 3D meshes for 3D stacked SoCs.

Tiled many-core system-on-chip can be further differentiated by how tightly the cores are actually integrated with each other and the network-on-chip.

Tightly-coupled As sketched in Figure 2.3b, the processor cores are connected as “systolic arrays” of processor cores by point-to-point connections between cores. The basic property is that software directly interacts with a neighbor via the core interface. This concept can be found for off-chip multi-core in the popular Transputer [222] and the iWarp architecture [33]. Waingold et al. [216] have proposed a first on-chip many-core processor that tightly couples processor cores and a network that is dynamically switched by the processors with extra information emited by the compiler. Prominent examples of many-core system-on-chip with a broader use case are for example TRIPS [184], ADRES [143], REMARC [148], Micronmesh [110], Loki [19] and Invasic TCPA [95]. A tightly-coupled structure can also be found in GPUs.

Loosely-coupled This is the more generic intergration variant that covers a lot of possible platform layouts. One example is sketched in Figure 2.3a. The key difference is which communication channels are possible. While in tightly-coupled platforms communication is essentially limited to the next neighbors, loosely-coupled tiled many-core system-on-chip allow more flexibility in communcation up to arbitrary communication relations.

While this roughly describes loosely-coupled many-core system-on-chip, their actual layout, organization and parameters can vary largely. In the example in Figure 2.3a processor cores are grouped in tiles of four cores, each with a local memory and global memory and I/O resources.

Loosely coupled many-core system-on-chip can have a heterogeneous tile layout, but often expose a regular structure, for example the positioning of memory tiles or I/O devices follows a pattern. Platforms generally differentiate by:

Processing tiles Those tiles integrate a varying number of processor cores. Numbers from single cores per tile [31] up to 16 cores [55] are found in prominent platforms. They are interconnected by a bus and can share local resources, such as caches, accelerator IP blocks or similar. Beside that, they share their interface to the global interconnect, either as an explicit network adapter device or implicitly by the cache hierarchy.

Memory and I/O tiles There are usually other tiles that provide access to some shared global memory, such as a last level cache or DRAM directly. Similarly, I/O devices are attached to the system, but can also be part of specialized processing tiles. The memory and I/O tiles can sometimes appear not as tiles in the regular structure but as directly attached to links at the outline of a mesh or similar.

Memory hierarchy The memory hierarchy is an important difference, which manifests in the way how software running on the cores sees memory. The most prominent differentiation is between shared memory and distributed memory. In the former the software doesn't even know about the platform architecture and when running on different tiles, software may "only" observe the platform layout by varying memory access latencies (non-uniform memory access, NUMA). In the latter, the software is in opposite usually aware of the platform layout. I go into further details of this in the next sections (see Section 2.2.1).

Summarized, loosely-coupled many-core system-on-chip are a rather broad class of platforms and the actual implementations can vary a lot. The communication latencies are usually higher and vary much more for loosely-coupled platforms. Hybrids of loosely-coupled and tightly-coupled many-core system-on-chip are also possible [184, 95]. In the following I focus on loosely-coupled tiled many-core system-on-chip. While the programming and design of tightly-coupled many-core system-on-chip is challenging too, they do not expose the challenges addressed by this thesis.

2.1.3 Brief Survey of Many-Core System-on-Chip

Over the last 15 years tiled many-core system-on-chip have found increasing interest in academic research and industry and gain importance with the continuing dimension shrinking. In the following I give a brief survey of many-core system-on-chip, focusing on the most prominent projects and products.

There have been predecessors to many-core system-on-chip, that are integrated on board level. Those were necessary because the integration did not yet allow on-chip integration of tens or even hundreds of processor cores. As mentioned before, the Transputer was one of the first steps [222]. Around the same time, Annaratone et al. [9] proposed a similar multi-chip systolic array, named Warp. iWarp [33], was proposed as an extended implementation of the Warp multi-chip systolic arrays. Beside a higher computation and communication performance and lower cost, it introduces the concept of other, specialized topologies and a the communication on a coarse-gain: Not only direct neighbors can communicate, but communications can pass several nodes.

The RAW processor [216] and REMARC [148] extended this multi-hop communication for systolic processor arrays with different degrees of freedom of communication relations. The Piranha platform [16] was again a multi-chip design, but the first that looks quite similar to what we nowadays see as loosely-coupled many-core system-on-chip: Each chip node is a multi-core chip that share local memory and an interface to the board-level interconnect.

Starting from those, research on many-core system-on-chip flourished since the year 2000. The most prominent examples are discussed in the following and summarized in Table 2.1. The first research platforms which reached a wider perception was the TRIPS paltform introduced by Sankaralingam et al. [184]. It was platform with a loosely-coupled tiled system with tightly-coupled processor arrays as tiles. 16 small processing elements formed a core in TRIPS and in the reference platform four of those cores are connected together with distributed memory units coupled to memory controllers. In their followup work together with Intel and IBM [37, 88, 89, 185, 120] the TRIPS

Name	**Year**	**Chip**	**#cores/ #tiles**	**programming model**	**on-chip memory**	**interconnect**
Research Projects						
TRIPS [184, 37, 89]	2003	yes	32/32	data flow	1.2 MB	two separate mesh networks
Intel SCC [104]	2010	yes	48/24	Shared Memory and Message Passing	12.384 MB	mesh
P2012 [23]	2012	yes	flexible (4/4)	PGAS	flexible	flexible asynchronous NoC
InvasIC [99]	2012	no	flexible	various	flexible	mesh
OpTiMSoC [218]	2012	no	flexible	message passing	flexible	mesh
Nanomesh [207]	2013	yes	8/8	distributed memory	512 kB	asynchronous mesh with diagonal connections
SpiNNaker [77]	2013	yes	18/18	distributed memory	1760 kB	two networks: internal and external communication
Pulp [50]	2014	yes	flexible (8/1)	Distributed Memory	flexible (72kB)	bus hierarchy
Swarm [107]	2015	no	64/16	Shared Memory	21 MB	mesh
Kilocore [31]	2016	yes	1000/1000	Shared Memory	768 kB	mesh
OpenPiton [14]	2016	yes	25/25	Shared Memory	2000 kB	mesh
Commercial Products						
picoChip [64]	2003	yes	430/430	distributed/data flow	912 kB	TDM mesh
Tile Processor [3]	2007	yes	64/64	Shared Memory and Data Flow	5 MB	5 (functional partitioned) mesh networks
Epiphany-III 16-core [2]	2013	yes	16/16	?	512 kB	mesh
Kalray MPPA2-256 [55]	2013	yes	256/16	Data Flow, Shared and Distributed	16 MB	2 torus NoCs
Intel Xeon Phi [49]	2014	yes	72/72	Shared Memory	40.5 MB	ring
EZ-Chip Mx-100 [63]	2015	yes	100/25	Shared Memory	40 MB	mesh
Sunway SW-26010 [75]	2016	yes	260/260(4)	Hybrid	21.76 MB	meshes

Table 2.1: Comparison of Research and Commercial Many-Core System-on-Chip, focusing on loosely-coupled platforms and the most prominent and recent examples.

architecture was further evolved the concept of a composable architecture where simple processing elements can be composed to larger cores.

Around the same time of the orginal TRIPS work a commercial platform was release, the picoChip [64]. The chip contained four classes of tiles, each with one core: Control cores for the system, cores with extra memory, cores with a multiply-accumulate unit and standard cores, a layout that was optimized for the CDMA algorithm. PicoChip found its way into basestations and the company was finally acquired by Intel.

Generally, it can be said that cellular network basestations are a common use case for many-core system-on-chip, along with other networking applications or multimedia processing. Another successfull company that has penetrated such markets is Tilera [3]. It evolved from the RAW research group around 2007 and is the first commercial platform that deploys a loosely-coupled many-core system-on-chip based on a packet switched mesh network. The first Tilera processor integrated 64 tiles with each one processor core.

In 2010, Intel introduced the “Single-chip Cloud Computer (SCC)” [104]. It was a research chip, based on their previous work for general purpose GPUs (Larrabee, Seiler et al. [189]). While the processor cores in Larrabee were connected by a ring, the SCC was based on a 2D-mesh. Each of the 24 tiles integrated two processor cores. The cores share a message passing buffer for efficient on-chip communication. Beside that each core has a cache for shared memory accesses.

After that the research around tiled many-core system-on-chip reached its peak. The main research hypothesis that established around that time is that tiled many-core system on chip can be the scalable solution to build flexible computing platforms. The P2012 platform introduced by Benini et al. [23]. It integrates a flexible number of clusters that are connected through an asynchronous interconnect making it a *globally asynchronous, locally synchronous (GALS)* design. Inside a cluster a flexible number of processor cores share local resources inside one clock domain, while the clock domains between the clusters vary.

From the P2012 concept the PULP platform evolved [50]. As with the P2012 platform the PULP platform allows to set voltage and clock dynamically for clusters of multiple cores. It has recently (2016) become an open source platform The “Open Tiled Many-Core System-on-Chip (OpTiMSoC)” which I also base my work on was introduced as an open source many-core system-on-chip [218]. It is similar to Pulp with respect to the platform layout: Tiles integrate a variable number of processor cores and those tiles can be clocked independently (while that is not the usual use case). By employing tiles with different number of cores that all operate with “symmetric multiprocessing

(SMP)" it allows heteregenous platforms by thread-level parallelism. Contrary to PULP, OpTiMSoC was never demonstrated in a chip tapeout.

The research project "Invasive Computing" [99] generally builds on a similar tiled many-core layout as discussed so far, but provides more heterogeneity between the tiles. Some tiles integrate multiple homogeneous cores while other tiles integrate specialized cores. Finally, even tiles that contain tightly-coupled processor arrays are part of the concept. Beside that the approach differs in the programming models where the running applications organically share the resources of the system. There is also no chip tapeout of an invasive computing system so far.

The NanoMesh project [207] goes into a different direction than the other discussed projects by building an asynchronous logic which is supposed to lead to much better performance results. The design is centred around the network-on-chip routers and eight tiles are connected to a router, additionally to eight directions where packets can be routed.

The SpiNNaker project [77] builds an entire supercomputer based on 57.600 many-core system-on-chip. Each of those contains 18 cores that are connected by one network-on-chip to communicate with each other and with another network-on-chip to communicate with other nodes. Each of the chips has an SDRAM directly attached to it. The communication units are very simple packets of a few bytes and the interconnect between the SpiNNaker nodes is arranged so that the entire machine should mimic the operation of a neural network as part of the Human Brain Project.

Around the same time (2013) commercialization of more mainstrean architectures took place. For example, Adapteva commercialized the Epiphany-III which integrates 16 tiles with each one core in a 4x4 mesh, that can be connected with other chips to form a 1024-core system. The successor, Epiphany-IV integrated up to 4096 cores. While the Epiphany architecture was similar to the Tilera platform, the Kalray MPPA architecture [55] integrated tiles of each 16 cores. Each tile integrates shared memory for the 16 cores, while tiles communicate with each other using DMA transfers. A data flow programming language is used to program the system. Alternatively shared memory programming can be used to program the software in a tile and DMA transfers transport data between tiles.

In 2014 Intel commercialized its many-core research as part of the Many Integrated Core (MIC) architecture, commercially known as the Xeon Phi. It connects up to 72 cores with a ring interconnect. It is intended as a massively parallel co-processor. While it provides the traditional shared memory view to the user, it nevertheless needs special care to gain its actual performance by the programmer and programming tools respectively.

The currently fastest supercomputer in the world, Sunway TaihuLight, is based on 40,960 nodes. Each node is a many-core system-on-chip, the SW26010 [75]. It integrates 260 processor cores and the cores are organized in four groups. Each group contains a mesh of 64 simple processor cores and one auxiliary core. The groups are connected via a packet-switched network-on-chip, while the mesh inside the groups is considered tightly-coupled.

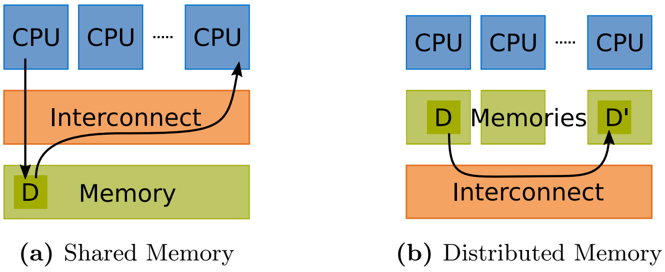
Most recently more research projects have evolved around scalability and programmability of tiled many-core system-on-chip. The Kilocore [31] takes a different approach with respect to memory than most other many-core system-on-chip: It does not integrate any caches to mitigate the coherency problems (see Section 2.2.1.1 below). It integrates 1000 cores at low energy arranged in a mesh.

Jeffrey et al. [107] introduce the SWARM architecture that has 16 tiles with each 4 cores. They focus on the parallel programming of algorithms that expose ordered irregular parallelism, which is a shift away from the embarassingly parallel algorithms usually considered. Similarly, Balkind et al. [14] published the OpenPiton platform that is based on thousands of single-core tiles, but with a cache hierarchy and cache coherency.

The implications of the memory hierarchy and the programming model are discussed in the following.

2.2 Inter-Task Communication

While Amdahl's Law formulates the theoretical limit of the computational speedup, the *efficient* programming of the parallel part of a program is critical. The pre-dominant software languages, development flows and even programmers' minds follow the sequential execution model. In accordance to the single processor execution instructions are issued one after the other and the micro-architecture has to be optimized to exploit implicit parallelism. The programming of the explicitly parallel part is challenging and requires different programming models, a topic widely discussed in high-performance computing. Diaz et al. [59] for example give a good overview about the prefered programming models and their different use cases. In the following I give a rough overview about the two dominant parallel programming models and their popular variants and relate them to on-chip communication.



(a) Shared Memory **(b)** Distributed Memory

Figure 2.4: Abstracted view on different memory hierarchies and common inter-processor communication techniques

2.2.1 Memory Hierarchy and Programming Model

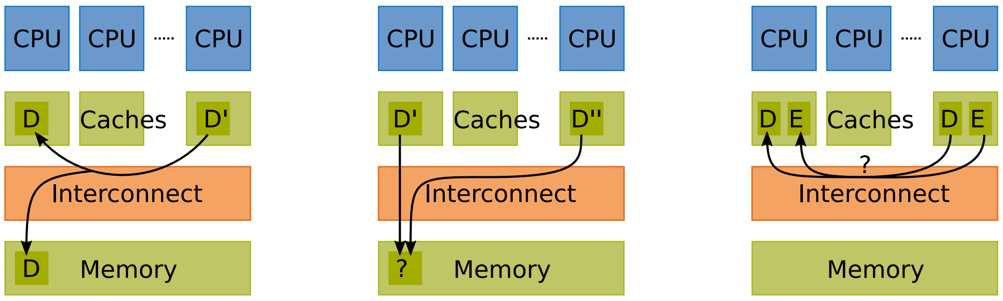
Before going into the details of parallel programming models it is important to understand different physical memory hierarchies and how they influence the programming model for software. Figure 2.4 illustrates the two dominant approaches to arrange memories in a computer system in an abstract fashion.

Shared memory systems (Figure 2.4a) have a global address space that is shared by all processor cores. This means that each data item has the same physical memory address on all processor cores. This has the obvious advantage of simplicity:

- The communication between tasks is implicit by writing to shared data items
- Moving a software task from one processor core to another does not require changes of the task and its data

Distributed memory systems (Figure 2.4b) instead don't share an address space. Each processor has its own dedicated address space. This is most often a dedicated local memory as illustrated in Figure 2.4b, but another well-established method is partitioned global address space (PGAS): In PGAS the cores share a global memory physically, but the address spaces are logically partitioned. Compared to shared memory, distributed memory is characterized by:

- The communication between tasks is explicit by transferring data items between memories.
- Moving a software task from one processor core to another requires moving the data along with the task



(a) Cache Coherency **(b)** Synchronization **(c)** Consistency

Figure 2.5: Challenges with Shared Memory

2.2.1.1 Shared Memory, Coherency, Synchronization and Consistency

As motivated before, shared memory programming is prefered by most programmers. The abstract illustration in Figure 2.4a highlights the ease of programming from a user perspective. Anyhow, the problems come with reality. For improving the performance the average memory access latency time is a key component and computer systems have integrated caches. A cache stores the most recent memory regions because temporal and spatial locality of the software makes it beneficial to have the data near to the processor core. A directly integrated cache (level 1) only needs one or a few clock cycles to access a data item from the set of recently accessed memory regions, while a DRAM memory access can easily take up to tens or hundreds of CPU cycles.

The problem with caches is depicted in Figure 2.5a: Each core can have an individual local copy of a data item. As long as they only read that seems okay, but it becomes problematic once software on one core writes to this data item. The data needs to stay *coherent* between caches and the main memory. So a protocol is needed to exchange information about the accesses to data items: a cache coherency protocol. Those protocols assign a sharing state to cache blocks and certain communication is required on state transitions.

One part of the problem of concurrent accesses is *synchronization* (Figure 2.5b): If software from two different cores read a data item and then write another value back, which of those should become visible? Mutexes, semaphores and other software mechanisms to ensure atomicity of code regions rely on the underlying hardware to ensure it. Finally, the problem of *consistency* (Figure 2.5c) arises when software running on different cores relies on assumptions about a certain order of read and write operations.

As mentioned, the memory hierarchy does not necessarily imply the only valid programming model. For example, it is easily possible to treat the shared global address space similar to distributed memory from software. By not sharing data in a physically shared memory it logically has the properties of a distributed memory system then.

Without going into the deep details of this topic, shared memory can become critical when it comes to scalability. There are often certain scalability issues projected for shared memory for tens or even hundreds of processor cores [116, 48, 104], mainly due to the hardware overhead in directories. Also the latencies and thus the average memory access time can suffer with a high sharing degree. Those concerns are supposedly mitigated by novel approaches as for example summarized by Martin et al. [137].

2.2.1.2 Distributed Memory

As illustrated in Figure 2.4b communication between processor cores in distributed memory system is *explicit*. This means that data items are moved from one memory to another by software operations. In multi-computer distributed shared memory systems, data is typically exchanged via a the network interface card over a network. This concept is transferred to many-core system-on-chip accordingly.

Message passing is the predominant programming model for distributed memory systems, but there are some alternatives and building blocks for it that are introduced in the following. Those approaches directly lead up to higher level protocols for efficient programming, such as message passing.

2.2.1.3 Distributed Shared Memory

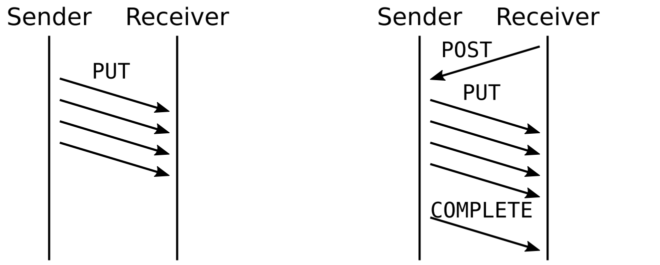
From the beginning of scaling clusters of computers there was the desire to provide the programmer the experience of shared memory programming on a distributed memory system as *distributed shared memory (DSM)*. DSM has emerged in the early 1980s for multi-computer network clusters and was a vivid research topic until the mid 1990s [157, 171]. DSM is strongly connected to the memory consistency model and ensures the shared memory view based on underlying replication and migration strategies [171]. Generally, three levels of distributed shared memory can be identified [157]: (i) the *compiler* emits primitives for coherency and consistency for shared data access. This often goes in hand with language extensions or restrictions. Midway [25] for example uses such language extensions. ii) *libraries and the operating system* use the virtual memory subsystem and consistency model to provide a shared memory abstraction. There have been pure software implementations on user-level [8], by language extensions [39, 13] or with a modified operating/runtime system [126, 24, 25]. iii) Hardware implementations focus on caching techniques and can for example be seen as shared memory systems where the memory is distributed among multiple computers (CC-NUMA) [57, 125]. Other approaches such as proposed by Hagersten et al. [93] only consist of caches and are actually shared memory system without a shared memory but dynamically managed locality of data. Finally, reflective memory approaches [73, 106] manage the locality of data in distributed memory on another granularity level.

Those concepts are not very relevant for many-core system-on-chip. This is mainly because the platforms deployed either focus on cache coherency, which is easier to handle on-chip for a relatively large number of cores (a few tens), and because the boundary conditions are slightly different and favor more light-weight approaches. Stemming from that, DSM also exists for many-core system-on-chip. But it defines a memory hierarchy and the fundamental difference between distributed shared memory in many-core system-on-chip and multi-computer systems is the ease of access to remote memory. In on-chip DSM all tile local memories are globally addressed. This *remote memory access (RMA)* is characterized by the fact that it is a *one-sided* operation that does not invoke the remote processor. The memory hierarchy and network-on-chip must provide the required consistency model as for example the Æthereal [86] network-on-chip provides.

The OSCAR multi-core system-on-chip [121] integrates hardware extensions along with compiler extensions to ensure consistency between tiles. Similarly, Chen et al. [47] propose a microcoded controller for the use of globally distributed shared memory and the synchronization primitives. The commercial Adapteva Epiphany platform [2, 214] and the InvasIC [225] platform provides a similar layout and hardware support. Overall those distributed shared memory approaches are similar to the implicit cache coherency, but managed explicitly and thus lowering the overhead of cache coherency operations. Finally, other approaches go beyond the exchange and synchronization of data. Monchiero et al. [149] for example propose a distributed dynamic memory management unit.

2.2.1.4 Remote DMA

Similar to the RMA approach described before, remote direct memory access (RDMA) is a method to move data to a remote memory. As RMA it also does not involve the remote processor. But additionally it offloads the local processor by directly copying data from the local memory to the remote memory. Direct memory access (DMA) is a well-established method for shared memory systems and suitable to transfer bulk data between memory regions or between a device and the memory. A DMA controller is even often deployed in shared memory systems to offload the task of data transfers. The



(a) One-sided communication without synchronization (b) One-sided communication with active synchronization

Figure 2.6: One-sided communication and synchronization

approach of RDMA is well adapted in multi-computer and supercomputing systems. For example, RDMA is a very popular feature in the Infiniband interconnect standard [12, 166, 129]. In tiled many-core system-on-chip remote DMA controllers copy data from a memory address in one tile to the a memory address in another tile, such as for example in InvasIC [225].

2.2.1.5 Synchronization

Both memory access methods, remote memory access and remote direct memory access, are often described as *one-sided communication*. As briefly introduced before, the problem arises that data transfer from one tile to another tile itself is not sufficient: The sender and the receiver need to synchronize about a data transfer. For example it must be sure that the receiver can receive the data, the sender does not overwrite data from a previous iteration, etc.

The problem and a common solution are sketched in Figure 2.6. As depicted in Figure 2.6b the sender and receiver exchange messages around the data transfer. Barriers are often used for this purpose, for example in MPI, that does provide a set of one-sided communication functions too. This synchronization introduces a significant overhead [109, 203] and efficient protocols are proposed, such as by Gerstenberger et al. [81].

2.2.1.6 Distributed Memory Summary

In this section I have given a comparative overview of shared memory and distributed memory. The memory architecture and the programming model usually correlate, but distributed shared memory in the form as NUMA plus inter-tile synchronization are popular. Based on the primitives of remote (direct) memory access a communication infrastructure can be build, while synchronization is an important aspect. Anyhow, building applications and providing a generic processing runtime system can quickly become challenging. In the following I hence present the message passing programming model that is the most popular distributed memory programming model. It is commonly built on top of the same primitives, but provides extra layers of functionality.

2.2.2 Message Passing

This chapter has so far been building up the motivation for message passing in many-core system-on-chip. It can be summarized that a distributed memory architecture is more promising to scale, and other distributed memory programming models are better suited in specialized cases.

Essentially, message passing programming is the exchange of messages between communication partners in an explicit fashion: One communication partner, the sender, calls a `send()` function with the target where to send a message and data to deliver to this receiver, which finally retrieves it by calling a `receive` function. The underlying software and hardware system must ensure that each of those messages at some point reaches its destination intact. It is up to the application to interpret the messages.

In this section I briefly introduce the Message Passing Interface (MPI) which dominates multi-computer and supercomputer programming. After that I go through the different aspects of message passing and highlight the differences between message passing on multi-computer systems and on-chip message passing. From that I derive the sensible design space for message passing on many-core system-on-chip.

2.2.2.1 The Message Passing Interface for Multi-Computers

Message passing was the dominant programming model in supercomputing from the beginning. Several approaches for a unified API have been proposed in the 1980s [167, 132, 98]. Around 1992 the *Message Passing Interface (MPI)* has emerged and been the de-facto standard ever since [217, 62]. The original standard contianed the following essential features [217]:

- Routines for point-to-point communincation in different communication modes and blocking modes

- The abstract concept of a communicator, which is roughly the grouping of processes on different cores, along with topologies that describe the communication relations between processes
- Collective functions for one-to-many and many-to-one communication
- Specialized data types to ease the exchange of data

In 1996 it was extended to the current MPI-2 standard [80]:

- Dynamic process management to allow the application to start new processes and manage them
- One-sided communication in the form of remote memory access (see Section 2.2.1.2
- Extensions to the collective functions

While MPI is clearly focused on massively parallel supercomputers, it serves as the reference for the following considerations. It becomes clear that one cannot simply compile MPI for a many-core system-on-chip, but adoptions are needed. One of the most prominent implementations of the MPI standard is OpenMPI [78] and I use this as a reference for the following discussion.

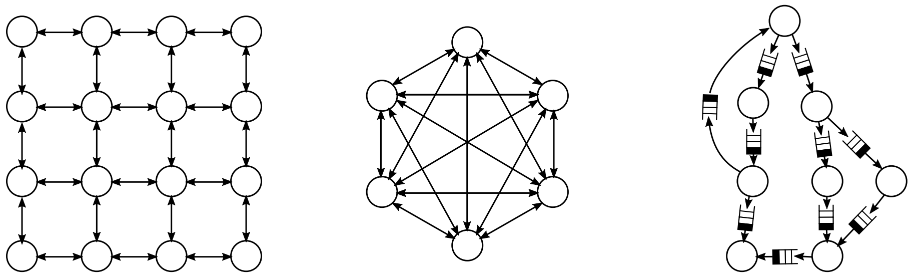
Beside the interface and its implementation the hardware ecosystem plays an important role. Infiniband [166] is among the most deployed communication infrastructures and its features and capabilities are of relevance for the contributions of this thesis.

2.2.2.2 Communication Partners

Before getting into the details of the message passing stack it is important to understand how applications communicate with each other. This does not mean the technical method, but the patterns in that parts of an application exchange data.

First of all, the application needs to be partitioned into discrete *tasks* (often also “process”). Stepping back from the application as code itself, this means that the problem needs to be decomposed so that i) the work is well distributed among the tasks, and ii) that the communcation between tasks is minimized. The latter is apparently the most efficient way of message passing – not exchanging more messages than necessary.

The problem of task partitioning is a challenging. Often the programmer can manually partition the problem or the algorithm can favorably be formulated in that way. Beyond that there is a lot of research in the field of automated or guided task partitioning, such



(a) Embarassingly parallel application

(b) All-to-all

(c) Data flow graph

Figure 2.7: Tasks graphs for partitioned parallel problems

as [45, 17, 90]. The question of how to partition a problem into tasks is beyond the scope of this thesis. More important in this context is the outcome: a *task graph* that is mapped to the platform (again the mapping is an interested research topic but beyond the scope of this thesis).

Figure 2.7 sketches three selected task graphs from the variety of different task graphs. One example for such a task graph in Figure 2.7a can be refered to as embarrassingly parallel. All communication is between neighbours and if the computation is well balanced and communication minimized such a task graph is well mapped to tiled many-core system-on-chip. Similarly, an application may be charachterized by all-to-all communication relationships (Figure 2.7b).

While the communication in supercomputers composed of hundreds of thousands of processors can often be characterized as a complex variant of communicating task graphs of that shape, another algorithm description is very popular for system-on-chip applications: Data flow graphs [54, 115]. Here, tasks communicate in a regular fashion by producing and consuming data (also “tokens”) at a certain rate. A similar description are Kahn Process Networks (KPN) [124, 42].

Figure 2.7c sketches a data flow graph. Buffers between the tasks are depicted to highlight the decoupling of the tasks. Those *channels* between tasks are buffered point-to-point relations. Throughout this thesis I use this model of describing applications without loss of generality, because it best matches the application programming in system-on-chip.

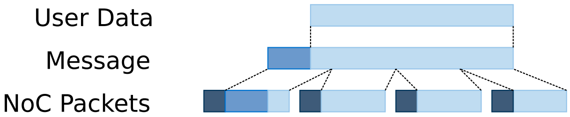


Figure 2.8: Packetization and protocol overhead

2.2.2.3 Packetization & Protocol Layers

Message passing was introduced as the transfer of data from a sender to a receiver. As mentioned the interpretation of those user messages is up to the application. But the transmission from the user task in one tile to the user task in another tile involves a detailed protocol.

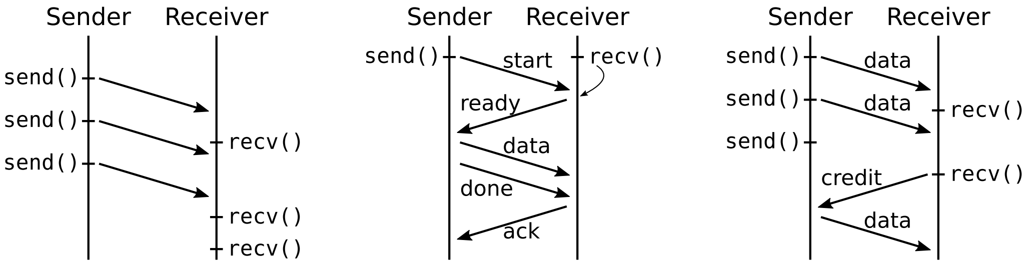
The protocol involves packetization and the "contract" of the implementation how messages are reliably transfered. The main part of this thesis is concerned about improving the protocol. It is important to understand that there are different levels of a protocol. Except for the rare, simplefied case of a point-to-point connection with sender and receiver directly connected, all computer communication is described by the ISO/OSI layer model [230]. On switching level, there is the routing information which applies to the network-on-chip communication. As depicted in Figure 2.8 the data that is transmitted physically between the sender and the receiver over the network-on-chip is divided into chunks of data, each with the essential routing information, that is at least the destination of the data packets. This packetization involves in smaller packets, where the packet size is optimized with respect to network contention.

With resource sharing on different levels (as specifically object of this thesis) each packet's header contains further hierarchical specific information, such as a data stream identifier and sequence number of a packet in the case the network does not guarantee strict ordering.

Beside this low-level packetization that has the purpose to transfer a chunk of data from the sender's position to the receiver's position, there is other protocol information of the other layers with adds to the actual user data during the transmission processes. This is the end-to-end application infromation, that are further described in this thesis.

2.2.2.4 Flow Control & Communication Modes

The protocol messages in message passing protocols are generally classified as either *eager protocols* or *rendezvous protocols* as sketched in Figure 2.9a and Figure 2.9b. In eager implementations the messages are sent from the receiver without knowledge about the receiver state. This means that the receiver must be able to receive the message



(a) Eager Protocol **(b)** Rendezvous Protocol **(c)** Credit-based Flow Protocol

Figure 2.9: Eager vs. Rendezvous Message Passing Protocols

and store it in a buffer, otherwise it can lead to backpressure on the network. This is mostly used for small messages. In rendezvous protocols the sender knows or discovers the receiver state and can hold a message back until the receiver is ready.

Figure 2.9b depicts a rendezvous protocol with all possible control messages. Essentially, rendezvous protocols care that the sender knows that the receiver is ready to receive a message. In the example further control messages, such as that the completion is signaled and the completion of the receive function is signaled. Other variants of the rendezvous protocol use credits (see Figure 2.9c): The receiver reserves a certain amount of buffers for the sender, the sender can post as many messages as it has credit, and finally receives new credit from the receiver. To reduce the number of control messages, the receiver often updates the credit only below a certain threshold.

In MPI flow control is only partly supported by the underlying hardware [128]. Instead implementation usually employ a user-level credit-based flow control. Due to the characteristics of the underlying transport protocols, MPI implementations rendezvous protocols are often implemented in a high performance fashion such as using RDMA [129, 199, 198]. This apparently needs to be reconsidered in the context of many-core system-on-chip as i) the access latencies between the network interface, the processor core and memories are very small, and ii) the main location of buffering is the local memory.

Generally, buffers play a critical role in the scalability of MPI implementations [35, 190]. There are two main issues with buffers: With up to millions of nodes in a supercomputer the provisioning of sufficient buffers for arbitrary communication can become critical. Also, the bandwidth of network communication has approached the memory bandwidth [35]. Thus it is desirable to get rid of any extra copying operation. A so called *zero-copy* protocol implementation transfers data from one user space to another user space.

If buffers are used, an implementation has further design decisions [178]: It can either transfer the data from the sender (push) or from the receiver (pull). Furthermore, the management of buffers and memory in general has a lot of degrees of freedom.

Finally, the MPI exposes different buffering schemes to the programmer by four variants of the `send()` function: i) ready: only sends when the matching receive was called, ii) standard: posted send, no flow control guaranteee, iii) buffered send: copy to local buffer and forget, iv) synchronous: returns after receive was also called.

As highlighted, the boundary conditions and the tight integration of the network and protocol layer with very high bandwidth and minimum latency ease the considerations for on-chip message passing significantly as discussed in the following.

2.2.2.5 Progress Engine

In message passing the *progress engine* is responsible for handling the sender requests and deliver incoming messages to the receiver. In MPI it is commonly implemented in software and a lot of effort is put into providing a thread-safe efficient progress engine.

Traditionally, the progress engine of MPI implementations often relied of calls to API functions. As part of function call the implementation then triggers the progress engine to check for example outstanding operations, timers, unexpected incoming messages etc. This is contrary to the MPI standard that actually guarantees asynchronous progress without calling the API functions. Performance and resource advances anyhow made asynchronous progress with an extra thread common.[194]

Finally, there is a trend to offload the MPI progress engine in multi-computer systems into the network interface card (NIC). Modern NICs deploy a programmable embedded processor for this kind of protocol processing [36]. For example the Quadrics network interface [165] provides a programmable subsystem to offload the progress engine. The idea is to offload a significant share also from the operating system to a "network operating system" on the NIC. Similarly, Cray Red Storm contains an embedded 500 MHz PowerPC at each node [34].

Underwood et al. [210] have identified that the offloading should not only focus on the sending and receiving operations, because this could negatively impact the throughput when many outstanding or unmatched messages are encountered. They therefore propose associative matching structures. Tanabe et al. [200] propose a mechanism that is integrated into the memory path and also improves the handling of unexpected messages.

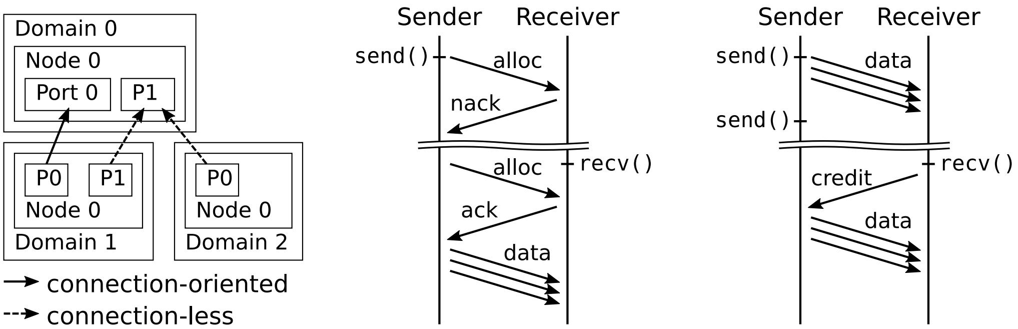
2.3 Message Passing APIs in Many-Core System-on-Chip

As mentioned before, message passing in multi-computer systems and on-chip message passing face different boundary conditions with respect to latency, bandwidth, memory space and integration between the protocol engine and the network. While MPI is the de facto standard in multi-computer programming of up to millions of cores, there is no established standard API for on-chip many-core programming.

Due to the wide acceptance of the MPI standard there have been proposals to down-strip MPI for embedded platforms. McMahon and Skjellum [142] for example compared two approaches to this idea with respect to the memory limitations of embedded systems: In a top down approach they took the MPICH implementation and stripped features like the MPI send modes and reducing code size. In a bottom-up approach they identify the most critical functions and add some further convenient features to create a configurable subset of MPI. Similarly, TMP-MPI [182] is a small subset of MPI including 11 functions. It is focused on softcores and custom accelerators among multiple FPGAs. They decided for a rendezvous protocol that sends request envelopes to the destination and the receiver then sends a message once it is ready to receive the data. It maintains one message queue at the receiver side that contains all pending requests. On a `MPI_RECV()` call the implementation than checks if a send is pending. The SoC-MPI [135] library also focuses on multi-FPGA systems of softcores and defines a subset of 18 functions from MPI. RampSoC-MPI [84] and ocMPI [72] is similar in this respect. Kohler et al. [122] propose another subset of MPI, but with a different target architecture. They target the Intel Single Chip Cloud Computer (SCC) and use the point-to-point message passing capabilities to build a reduced MPI around it.

While those approaches are doing well in reducing the size of the MPI implementation, they leave aside a thorough consideration of adding or modifying functions due to the different boundary considerations. In this respect. Another approach is to define an own message passing API. Poletti et al. [169] for example define a rather low level API that is limited by targeting a distributed shared memory platform.

Finally, there is an consortium approach to define a message passing API for embedded systems: The Multicore Association Communications API (MCAPI) [153]. Despite the specification has stalled and is not an open standard, it has found a reasonable acceptance. Matilainen et al. [138] for example implement the MCAPI for FPGA-based multi-core system-on-chip. A closer look between the downstripped MPI and MCAPI reveals that there is a vast overlap of provided functionality. MCAPI has a few concep-



(a) Adressing and communcation modes **(b)** Connection-less protocol **(c)** Connection-oriented protocol

Figure 2.10: Details of the message passing protocol used in this thesis

tual ideas that better fit the boundary conditions and communication patterns described in Section 2.2.2.2, specifically data flow graphs specifically targeted.

Without loss of generality I hence chose MCAPI as baseline. It is characterized by the following features:

Adressing Scheme MCAPI uses the notion of domains, nodes and endpoints. Domains are the tiles in this context, while nodes are tasks running on a tile. Each node can have multiple endpoints for communication, each uniquely addressed by the `<domain,node,port>` tuple. Examples for this hierarchy are sketched in Figure 2.10a. This is different to the MPI rank-based addressing.

Channel Communication This is connection-oriented communication, which means that two endpoints are connected to each other and only messages from the sender endpoint are accepted by the receiver enpoint (see Figure 2.10a). This eases flow control and such a communication mode does not exist in MPI.

Message Communication This is connection-less communication, which means that each endpoint can send messages to each other endpoint (see Figure 2.10a). This is like in MPI.

Blocking and Non-Blocking API Functions Both MPI and MCAPI define blocking and non-blocking variants of the send and receive functions. A non-blocking function call allows the software to perform other operations while the message passing operation completes (overlapping computation).

Dynamicity MCAPI willfully does not define dynamicity in a way of node discovery etc. This is a static view of a system, but as motivated before this work focuses on flexible, next-generation many-core system-on-chip that run userspace applications and not only baremetal systems.

The underlying protocol has to support two basic modes: Figure 2.10b shows the simple rendezvous protocol for connection-less communication. As the sender is not sure if the receiver is ready to store the message it queries for a slot in the buffer. Once this becomes available it receives the address. The base protocol for connection-oriented message passing is depicted in Figure 2.10c. In case credits are available, the sender can just send data to the next message buffer slot. Once it runs out of credits, the sender has to wait for the receiver to grant new credits.

The message passing API that is co-developed with the NAMP concept in this thesis is based on MCAPI and the aforementioned basic principles of a protocol implementation.

2.4 Network Adapter

It has been outlined how on-chip message passing differs from large-scale multi-computer message passing such as MPI. One of the key differences is the interface between computation and communication. In multi-computer systems this is the Network Interface Card (NIC). There is a trend to offload as much of the protocol processing to the NIC. The NIC thereby often integrates a processing element that can be even more powerful than the actual processor core in a tile in a many-core system-on-chip.

The link between the network-on-chip and the tiles is the *network adapter (NA)*, also refered to as network interface (NI). The network adapter is the critical element to properly complement communication and computation, and has a similar role as the NIC in multi-computer systems. Its role and style of integration is more flexible than for a NIC. It is essential to understand the difference of the concepts between data transfers in the computation domain (tile) and the communication domain (NoC). For a detailed introduction into the differences between network-on-chip and busses see the white-paper of [11].

Figure 2.11 visualizes the difference between the pin level protocols and flow control and highlights the role of the network adapter in this. At a first glance the difference in number of different signals and total amount of wires can generally be observed. But more importantly is the protocol nature: Bus transfers are generally organized in cycles. In the sketch in Figure 2.11 there is a write transfer, where the acknowledge in the next

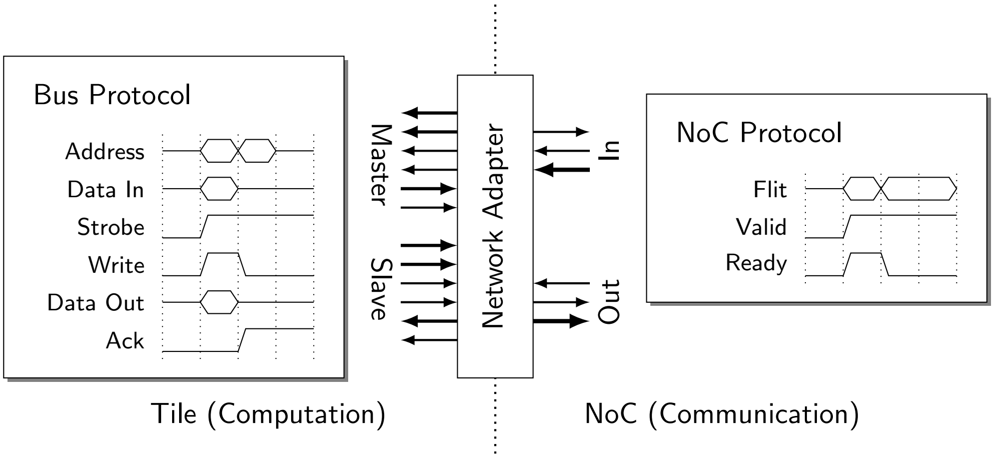


Figure 2.11: The network adapter as the interface between computation and communication and the different pin level protocols.

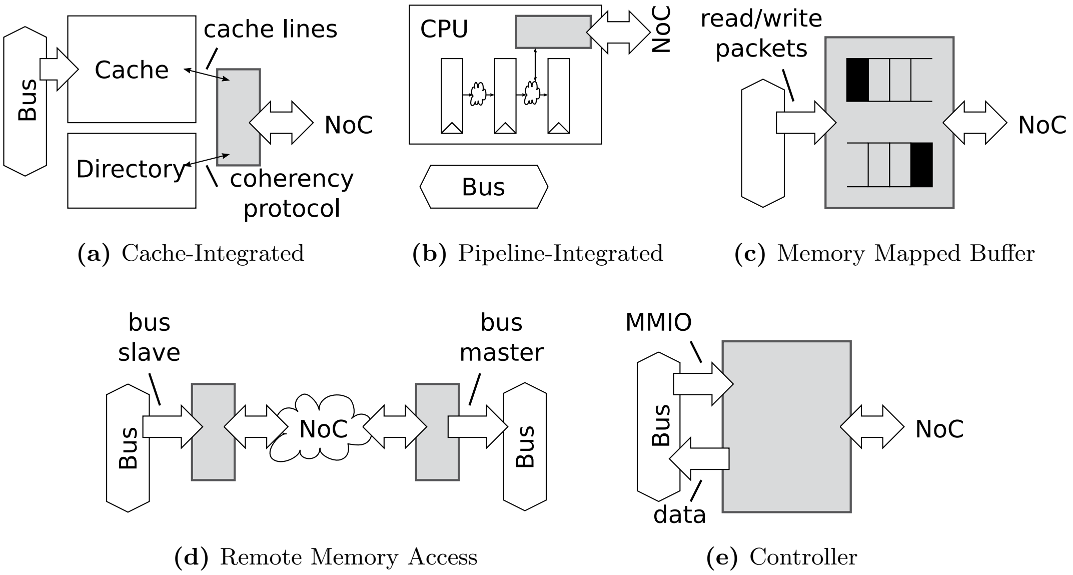
cycle overlays with a next read access. Each access is initiated by the master and gets a reply in the next cycle.

As introduced in the previous chapter, the flow control protocols are much easier in NoCs as channels are unidirectional there. Data transfers are performed with a handshake protocol, where the master side assigns a data item (flit) and a valid signal, while the slave assigns the ready signal whenever it accepts data. Traditional bus transfers are translated to a de-coupled sequence of request and response messages, similar to data transfers in the networking domain. While this is a bit simplified, network adapters commonly integrate with two such interfaces. In the following common network adapters are classified and compared to the proposed solution.

2.4.1 Classification

As mentioned, the actual integration point is manyfold, ranging from deep in the processor core, directly in the cache or as bus slave. Also, the style of access differs between transparent translation and memory-mapped device accesses. Generally, network adapters can be roughly categorized as depicted in Figure 2.12. For the sake of completeness, the classification includes also network adapters not directly related to the work presented in this thesis.

Cache-Integrated Network Adapter For example, shared memory platforms have a cache and directory infrastructure that directly interfaces the network-on-chip, as sketched in Figure 2.12a. Piranha [16] was one of the first research projects in the direction of



(a) Cache-Integrated **(b)** Pipeline-Integrated **(c)** Memory Mapped Buffer

(d) Remote Memory Access **(e)** Controller

Figure 2.12: Classification of Network Adapters (Note: Arrows denote a master to slave relationship)

loosely-coupled many-core system-on-chip. While it was still a multi-chip solution, it already addressed many conceptual aspects of simple processing nodes relevant in this field. It used the shared memory programming model and contained a specialized hardware network adapter. This network adapter is micro-programmable to execute a coherence protocol. While being, conceptually interesting it faces performance bootlenecks of shared memory coherency.

Commodity multi-core system-on-chip often implement cache coherency by interfacing the cache and directory directly with a (bufferless) ring. The first Intel Xeon Phi coprocessor [49] with 57 to 61 cores used a ring interconnect that directly interfaces the caches. Recent versions of it [192] moved to a mesh interconnect. Finally, state of the art research such as the Piton platform [141] do a similar integration of caches and the mesh interconnect. Also the Tilera platforms [221] integrate two of their five networks with the cache hierarchy for the cache coherency.

With respect to message passing, exchanging messages over the main shared memory suffers from performance and scalability issues due to the coherency protocol. Beside that the major task is to move large blocks of data in the main shared memory. Gu et al. [92] propose a “Direct Memory Manager” that offloads the message passing protocol to hardware and takes care of moving the large data blocks in the memory. Chatterjee et al. [46] propose a messaging controller that offloads the data movement in a similar way. Software implementations such as proposed by Poletti et al. [169] can benefit from such hardware improvements.

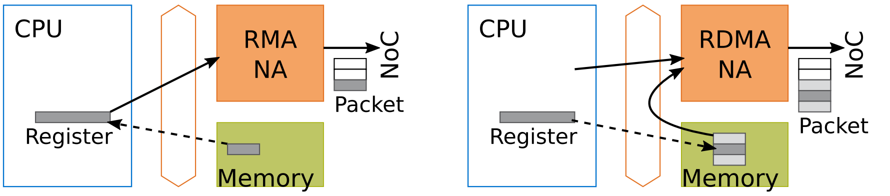
Such approaches may significantly improve the performance of message passing over shared memory. But they are still limited by the fact that data has to traverse the memory hierarchy instead of directly move data from one point to the other. While operations are offloaded from the software and the CPU caches, the latencies and also the power consumption are still significantly higher.

Finally, other approaches integrate explicit remote memory access functionality into the caches. For example, Kavadias et al. [114] propose to integrate remote memory access and remote DMA capabilities within the cache. Cache blocks can be configured to either serve as caches, access to remote memory or be part of remote DMA regions.

Pipeline-Integrated Network Adapter The second class of network adapters depicted in Figure 2.12b are those that are integrated into the pipeline of the processor core. This design is very common with tightly-couple many-core system-on-chip, because the communication relations are more static. Such an integration was also proposed early with multi-computer systems: Henry and Joerg [101] propose a register-mapped message interface that packetizes small messages and transfers them via a buffer.

As a more recent example, the Loki platform integrates configurable buffers into the processor pipeline so that software can directly read and write to the buffers of other cores [20, 18, 19]. All accesses are blocking so that a pipeline is stalled when a buffer is empty on receive or full on send. The Tilera platform is very similar [221]. But compared to most tightly-coupled many-core system-on-chip, the Tilera platform provides a software abstraction layer which allows either direct exchange of scalars with a few cycle overhead and latency. Beside that it allows for buffered transfer of larger memory blocks and messages. Similarly, ocMPI [72, 40, 41] proposed to interface a buffer for inter-tile communication into the pipeline. Their work added extensions to the ISA, which is mostly useful when blocking for messages, but in this case its mostly aliases for accesses to a register map.

Memory-Mapped Buffer Network Adapter The majority of approaches in loosely-coupled many-core system-on-chip chooses a different integration of buffers. Here some hardware message buffers are available via a memory map as depicted in Figure 2.12c. The network adapter is accessed by the processor via the standard data path and usually addressed as a standard device on the bus. This has two major advantages: The network adapter can be very easily shared among multiple processor cores, and the net-



(a) Remote Memory Access (RMA) **(b)** Remote Direct Memory Access (RDMA)

Figure 2.13: RMA vs. RDMA

work adapter is independent of the processor core and can be integrated with other implementations easily.

Similar to pipeline-integrated the buffer is used to exchange raw network packets between the processor and the network-on-chip. This means that the packets are formed and parsed in software. The TMD-MPI network adapter [182, 183] connects the processor cores and the routers with standard buffers. Multiple messages are either pushed one after the other into a buffer, which in the case of TMD-MPI is simple because the packet length is encoded in the header. Similarly, Marchesan Almeida et al. [136] prepend the destination address and packet size to each packet. Other approaches, such as MEDEA [44, 206], write a sequence number to each flit. A different approach is to integrate multiple buffers that only can hold one message with respective control and status register. Fernandez-Alonso et al. [71] for example claim to support two outstanding messages with dual buffering, without clearly describing the message format.

Multiple messages are also used by Heisswolf et al. [96]: Different buffers are used for different Quality-of-Service demands. Guaranteed service buffers are connected to reserved paths with guaranteed throughput, and best effort buffers can be freely used.

Remote Memory Access Network Adapter As part of Section 2.2.1.3 I have introduced the concept of remote memory access as a programming concept. As a hardware concept, the basic idea is that the entire memory of the system-on-chip is accessible from each processor as sketched in Figure 2.12d. This means that the memories are mapped into the memory map of the processor core. A slave element is attached to the bus or integrated into the data path of the processor then. It translates the memory addresses to network packets as depicted in Figure 2.13a. The corresponding counter part then translates this into a memory access and the response.

Bhojwani and Mahapatra [28] for example integrate this functionality in an extendible softcore and evaluate different packetization strategies in software, the core or a core peripheral. Bjerregaard et al. [30] present their network adapter that transfers requests of the "Open Core Protocol" (OCP) interface specification. The Aethereal network-on-chip integrates network interface "kernels" that similarly packetize interconnect accesses from different interface standards to network-on-chip packets [172, 60]. Modern AXI-based interconnects and interconnect generators contain similar modules, as they are internally based on flexible packet-based network-on-chips. Such works usually focus on heterogeneous network-on-chip. More in the direction of loosely-coupled many-core system-on-chip, the network interface proposed by Zaib et al. [225] provides access to other tiles memory via a "Remote Load/Store Unit".

Among the industry platforms, Intel's research platform SCC [140] provides a special kind of memory access, the "Message Passing Buffer Type (MPBT)". Such accesseses are either to a local message passing buffer SRAM region or a remote memory access. Each tile of two cores shares a message passing buffer. Beside that each core has a globally accessible test-and-set register. Similarly, the commercial Adapteva platform provides remote memory access via the memory map [2]. Beside that it provides a synchronization instruction that forces all cores on the chip to finish all outstanding transactions (barrier).

Network Adapter as Active Controller Finally, there are network adapters that act as active controllers. They are configured as a device by memory-mapped control registers (see Figure 2.12e. That means that an access to an address of those network adapters does not correspond to a memory address, but instead can store a value, trigger an operation or provide status. Two of the other network adapters we have discussed so far can be partly categorized here too: A memory-mapped buffer and the synchronization operations that accompany remote memory access network adapters can have a similar interface. But what differentiates this class from them is an increased complexity and offload capabilities that are in focus here.

A popular class of such a network adapter is based on "Remote Direct Memory Access (RDMA)" controllers to provide RDMA capabilities (see Section 2.2.1.4). As sketched in Figure 2.13, an RDMA controller directly transfers data between tile memories. This is generally more efficient than the remote memory access method, especially with an increasing data size. Many platforms contain DMA capabilities, often additionally to other features. Adapteva [2] for example provides RDMA additionaly to RMA. Similarly, the Kalray MPPA [56] includes a DMA controller in each cluster of 16 cores, that can transfer data inside the cluster SRAM or between the SRAM and the NoC. Han et al. [94] go well beyond that and propose a memory subsystem based on memory-to-memory transfers that goes beyond the basic DMA engines with scheduling.

Ly et al. [133] propose a message passing engine that uses DMA transfers. This functionality goes beyond the functionality of a pure DMA, because it knows the notion of messages. In a similar direction, the Resource Network Interface as proposed by Minhass et al. [146] that provides an interface to a buffer and the interface to connect to destinations and exchange messages. The Hydra network interface [213] provides DMA capabilities and program control capabilities. Finally, the Micronmesh [110, 113, 111, 112] provides capabilities beyond the raw data transfer, but integrates the message passing protocol.

2.4.2 Network Adapter for Message Passing

Previously, I classified network adapters, where the most relevant in the context of this thesis are memory-mapped buffers, remote memory access and active controllers. Those are essential building blocks and are often deployed in varying combinations. A message passing implementation is build on such features. The API and progress engine can generally be implemented using them and a varying degree of extra software.

Both RMA and RDMA transfers are limited by that they don't provide built-in synchronization. Hence, the RDMA is just one building block of the efficient message passing network adapter presented here. Other works build around DMA differently. FUNCAPI [138] for example is a full MCAPI message passing implementation that wraps around DMA transfers, but without giving much details of how synchronization is implemented. Fu et al. [74] and Ly et al. [133] similarly implement an MPI subset with DMA with the same lack of discussion of end-to-end flow control.

The RMA features of Intel's SCC can be used to implement a message passing protocol [179, 180]. The discussion by Rotta [179] demonstrates the complexity of transmitting simple messages. Finally, it doesn't address the concerns of this thesis because it focuses on the low level message interaction only. Ziavras et al. [229] take a different approach and propose a reduced MPI implementation that only features the one-sided MPI primitives. This shifts the flow control issues to the user, which is not very flexible and error prone. Marchesan Almeida et al. [136] build the message passing infrastructure around memory-mapped buffers. The software formats the NoC packets and a flexible message passing protocol with flow control would induce a lot of software overhead.

Summarized, the aforementioned proposals do all address the low level, bare-metal transfer of messages, but don't give an extensive insight about application level end-

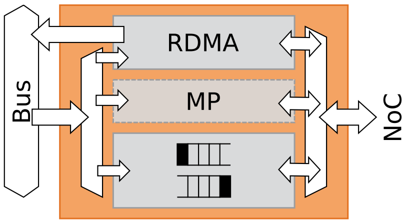


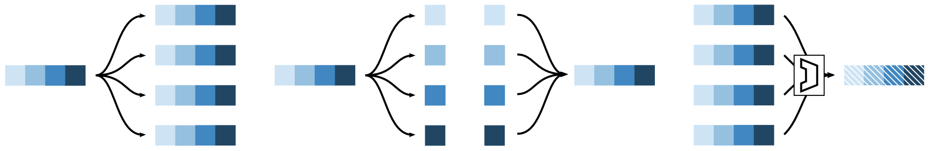
Figure 2.14: Baseline network adapter

to-end flow control. This problem is properly addressed by da Rosa et al. [52]. They propose a hardware extension to an RDMA that includes end-to-end flow control. It uses the MCAPI as programming API. Buffers are addressed by their `<node,port>` tuple and the software configures the controller to find the proper buffers in the tile local memory. The flow control is then maintained by the hardware. Micronmesh [110, 113, 111, 112] is a similar approach, but with their own message passing API which is similar in its core. The message descriptors are written to FIFOs and point to buffers, the hardware called Micronswitch Interface. Both proposals are the closest related work to the contribution of this thesis. But as the other works introduced above they focus on low level message passing for the bare-metal domain, but ignore the entire upper stack. First of all, the work presented in the remain of this thesis includes a better scalable and flexible hardware offload of the progress engine. Beside that it improves the overall message passing performance from one user task to another user task with concepts that are briefly introduced in the following.

Figure 2.14 summarizes the most relevant parts of this related work in a blueprint network adapter that combines the fundamental concepts. State-of-the-art message passing implementations focused on using the basic building blocks RDMA and buffers. Explicit protocol handling (MP) has only been deployed in the few aforementioned approaches and there is a huge potential to extend and optimize this. This thesis hence focuses on the optimization of the hardware-based protocol handling for flexible use cases. Beside that concepts are adopted and developed to efficiently support the whole software stack.

2.5 Concepts for Message Passing Acceleration

As mentioned before, the work presented in this thesis integrates efficient and flexible message passing support by the network adapter and concepts known from other do-



(a) Broadcast (b) Scatter (c) Gather (d) Reduction

Figure 2.15: Collective communication operations

mains. In the following I briefly introduce those concepts that are either adopted or conceptually developed in the main part of the thesis.

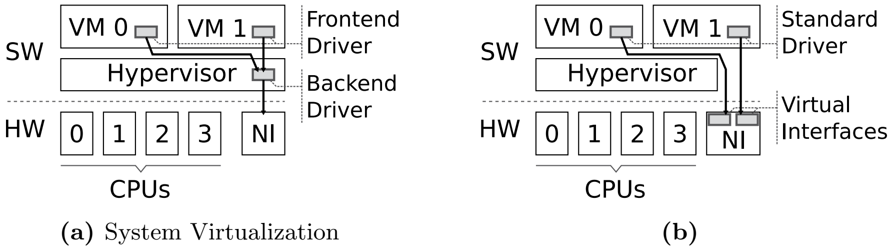
2.5.1 Collective Communication

Collective communication operations involve more than one sender and one receiver in a communication. They are one-to-many or many-to-one operations for distributing and recombining data. Figure 2.15 shows the standard four collective communication operations, such as also defined by the MPI standard.

Broadcast (Figure 2.15a) transfer the input data to multiple receivers and all of them receive the same data items. Scatter (Figure 2.15b) and gather (Figure 2.15c) are a pair of operations. Scatter distributes a data vector to multiple receivers so that each of them receives a chunk of the data. Gather is the opposite operation where multiple senders transfer a chunk of data to one receiver that combines it to one data vector. Finally, the reduction operation (Figure 2.15d) can be seen as the inverse operation of a broadcast: Multiple senders send equally sized data vectors that are merged into one vector of the same size at the receiver. To merge the multiple data vectors into one a reduction operation is applied to the individual elements such as a sum of all items.

Scatter and gather are well known operations in the context of DMA controllers and memory interfaces. There those operations help to efficiently utilize the interfaces. An example for such a DMA controller is ARM's PL080 [10]. Similarly, the Impulse memory controller [227] adds an additional virtual memory hierarchy and applies scattering and gathering operations to dense data transfers to the external memory.

As mentioned, collective communications are an essential part of MPI, because many algorithms contain those patterns. There are a lot of proposed algorithms and hardware extensions proposed for them [118, 164]. For example in the IBM BlueGene/L supercomputer, collective communication is supported with separate communication networks and a dedicated arithmetic and logic unit (ALU) for reduction operations [5]. Also com-



(a) System Virtualization **(b)**

Figure 2.16: System Virtualization and I/O Virtualization

mon Infiniband network interface cards include hardware support. For example ConnectX [196, 87, 215] enables tweaks of the protocol handling of collective communication by the programmable embedded processor.

In the system-on-chip context, collective communication is mostly tackled on the level of the network-on-chip. Ma et al. [134] for example propose multicast support for the network-on-chip, which is not only helpful to implement broadcasts but also for cache coherency operations. In the Æthereal NI [172] so called shells provide support for multicasts and simple narrowcasts.

The proposed network adapter goes beyond this basic support and can be configured to autonomously execute all four types of collective communication operations. Contrary to the work in high performance computing this is narrowed down to the restricted set of communication partners in typical scenarios.

2.5.2 Communication Virtualization

Virtualization generally refers to the abstraction of resources [181]. Its goal is to provide an entity the illusion of exclusive access to a resource. For example, virtual memory in computer architecture provides each user process the illusion of a large contiguous memory. Accesses are actually transparently translated to physical memory addresses. One of the key aspects is the seperation of entities so that they cannot interfere. System virtualization [85, 176] has been an increasingly important topic over the last decades. It allows multiple "virtual machines" to share one physical hardware platform. The major concern are the processor operations that alter the processor state and break the separation. Hence, system virtualization needs methods to enforce the separation, most notably binary translation [177], para-virtualization[15] and hardware virtualization [209].

Even with efficient virtualization support for user processes or entire operating systems, the I/O operations can become a bottleneck. Devices are shared between multiple entities and each of them has the illusion of exclusive access. In the context of multiple processes/tasks this means that the access takes place without invoking the operating system (OS bypass) . Therefore in early stages the approach to I/O virtualization was to emulate drivers [197] or use para-virtualization [170].

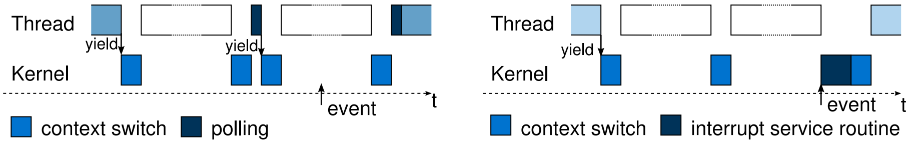
The state of the art for efficient I/O virtualization are self-virtualized devices [173]. The concept enables sharing and of a device by multiple entities by providing each a virtual interface and separate the accesses in the hardware device. In the context of high performance computing, the virtual interface architecture [65] was he first appearance of this concept despite it does not precisely describe the hardware architecture. With the appearance of programmable NICs, the idea arose to offload the functionality from the operating system kernel to the NIC [191]. RiceNIC [223] for example implemented this concept of a self-virtualized NIC based on an FPGA prototype. Liu et al. [130] discuss typical OS bypass with Infiniband and present a nested “VMM bypass” Infiniband NIC.

The concept of the RiceNIC [223] has focused on integration of control and data path operations in the firmware of a micro-processor running in the NIC. Rauchfuss et al. [175] proposed to replace the generic micro-processor that is in the critical path of packet processing. Instead, they propose to offload the processing to a set of configurable finite state machines to better suit the needs of embedded systems (ES-VNIC). The approach shares similarities to the work presented in this thesis in the way that it offloads virtualization and parts of the control and data path into hardware. The differences of on-chip message passing as presented in the main part of this thesis is in the complexity of the protocols they handle and thereby the complexity of the hardware FSMs. For example the task of header parsing is much less complex in the case of on-chip message passing, or the task of scheduling is much less critical as each operation of the NAMP is only a few clock cycles.

The work of the ES-VNIC was conceptually extended to integrate with a network adapter in a tile [174]. This integration focuses on the partitioning of the functionality and integration with the NIC. The work presented in this thesis is complementary to it and can be integrated in a way that the VNIC contains buffers that are endpoints in the message passing communication interface via the NAMP.

2.5.3 Event Signaling

While OS bypass removes the bottleneck from the user application to the shared device, event signaling concerns the other direction. Namely the question in this context is how to efficiently notify a task of an arrived message. Two methods are established: Polling is the repeated wakeup of the task to check if the event (message arrived) already occured.



(a) Polling **(b)** Interrupt-based

Figure 2.17: Timing diagrams for the dominant event signaling techniques

As sketched in Figure 2.17a this involves context switches to the waiting task (light blue) for a rather simple operation. This can be improved by having the kernel actually check for the occurrence, but it still has a significant impact if the number of waiting tasks grows. The more widespread approach are interrupts. As depicted in Figure 2.17b the task is not scheduled while waiting and the event interrupts the current execution. The operating system wakes the task up and it can later continue execution. A larger amount of interrupts can become problematic here.

Langendoen et al. [123] investigated the tradeoff between polling and interrupts for multi-computer systems. They propose an adaptive technique that switches between polling and interrupts depending on the event rate. In the state of the art interrupt-based event signaling dominates as it has a lower latency and the reduced overhead. To reduce the impact of the interrupt handling, *interrupt coalescing* has been introduced [226]: Multiple events are queued up and signaled with a single interrupt. For example network interface cards store several packets before raising an interrupt, such as by Goglin and Furmento [83]. For on-chip communication MSIQ [113] is an on-chip message passing adapter that coalesces interrupts. Scheler et al. [187] presented an approach to order interrupt requests by a programmable microcontroller targeted at real-time systems. While the overhead of the interrupt handling decreases, the contribution of this thesis is the complete elimination of event signaling overhead.

The technique applied to achieve this goal is novel and involves the direct interaction of the network adapter with the operating system task queues. The manipulation of those queues has been researched especially for hard real-time systems. Such hardware assists are often programmable. OSIP [43] for example is an application-specific instruction set processor that handles the OS task scheduling. HW-RTOS [154] implements a scheduler where threads are woken up when communication events occur. Nevertheless, HW-RTOS as other hardware operating systems is very limited in its flexibity. Similarly, the Transputer implemented a wake up mechanism controlled by the hardware [105, p. 32]. It automatically notifies waiting threads of external or internal events by putting them back to the scheduler queue. Anyways, that was a rather complex microcoded subsystem.

Summarized, the proposed *operating system queue manipulation* and its integration with the network adapter hence goes a new path with the ambitious goal to eliminate all overhead related to event notification. The key contribution is a configurable, yet lean queue manipulation that differs from the complex state-of-the-art solutions discussed before.

2.6 Dependability of Many-Core System-on-Chip

With the ever increasing feature size shrinking of CMOS devices dependability of such systems becomes an important topic. Examples for errors that can occur in the field are:

- transient faults such single event upsets caused by exposure to radiation [147],
- process variation due to challenges in the fabrication of small feature sizes [155],
- accelerated aging due to thermal hotspots, so called negative bias temperature instability [188]

Those error can manifest in a variety of effects, ranging from that they are not observed to a crash or security flaws [103]. One goal is to react to errors or ideally anticipate failures. Beside functional errors the management of tight power constraints can limit the utilization of the system-on-chip [70]. Running a large many-core system-on-chip at maximum frequency and utilization hits another power wall. Management of tasks and resources is needed.

Thermal gradients and similar effects can be reduced by task migration. The basic idea is migrate tasks from hotter to colder tiles. Thermal management with a distributed, agent-based runtime system is a scalable solution [108, 66, 68, 67]. TAPE [68] for example trades 'power units' between tiles. Power units are used to 'buy' tasks from other parts of the system. Such management approaches demand robust, seamless and transparent task migration. The downtime during a task migration is challenging in distributed memory systems. Two basic approaches are differentiated [38, 69]: Task replication is the permanent availability of each task's program code in multiple tiles. Migration is then the migration of the task state. Task recreation instead copies both the state and the code during migration. The focus of this thesis is on the communication migration, because the redirection of communication channels is an integral part of task migration. Some approaches pause all tasks in one application during migration and the full state transfer ensures consistency. Alternatively, a few approaches use reduced checkpoints [1, 168, 186], Finally, Tendulkar and Stuijk [201] for example specifically address the network-on-chip communication, but also pause the entire application during migration. Cannella et al. [38] finally propose a "forwarding" scheme. Here, the application is not paused entirely and the communication migration is more flexible. However, their work on communication migration is only discussed on a broad level and in a limited scope. The approach presented in this thesis is much more elaborated and covers a wider spectrum of communication methods and conditions.

2.7 Conclusions

In this chapter I have given a broad overview of the development of multi-core and many-core system-on-chip architectures and covered the challenges that come with them. On the one hand this gave a comprehensive understanding of relevant concepts, and on the other hand it pinned down the relevance of the contributions of this thesis. After this overview, the basic functionality of network adapters has been introduced and the state-of-the-art has been classified. A large number of existing network adapters allows the access to remote memory or the transfer of data from one tile's memory to another tile. While those basic building blocks may suffice to build arbitrary complex message passing protocols, there is only a very small number of network adapters that integrate protocol handling for message passing. But even those related concepts are operating on a low level and with a very limited scope with respect to supported communication modes, concurrent operations and flexibility. The proposed "Network Adapter for Message Passing (NAMP)" that is presented in this thesis covers message passing on a much broader baseline. Finally, the key contribution of this work is the integration of novel concepts for on-chip message passing. The presented concepts are well established in other domains or contexts, but this thesis discusses the proper adoption and integration of those concepts for efficient, resource-limited inter-tile message passing. The resulting network adapter improves the overhead at several critical interfaces and functionalities in more extensive software stacks beyond bare-metal programming. In the following those concepts are applied to the boundary conditions of on-chip communication and co-designed with the message passing protocol.

3 NAMP: Design of a Network Adapter for Efficient Inter-Task Communication

In the previous parts of this thesis I have motivated the need for a network adapter for efficient inter-task communication that enables high-throughput and low latency communication. The demands and potential areas of improvement have been discussed: i) hardware offloading of the progress engine, ii) operating system bypassing (self-virtualization), and iii) event notification. The state-of-the-art network adapters for inter-tile communication in many-core system-on-chip focus mostly on the network-on-chip. Their interface and functionality is designed for simple bare-metal software and do not support the higher level message passing protocol or interaction with an operating system. There is a lot of potential for improvement for the higher layers. Motivated by the deficiencies of the state-of-the-art this chapter presents the key contribution of this thesis: The concept of the *Network Adapter for Message Passing (NAMP)*. It is targeted at area-constrained, massively parallel many-core system-on-chip. The major features of the NAMP concept are:

- The scalable and flexible NAMP concept with basic progress engine offload (see Section 3.2). The key features are offload of parts of the data path and control path to hardware, keeping data structures in memory, allowing non-blocking operation and thread-safe buffer integration.

- The basic NAMP supports different communication modes: connection-oriented vs. connection-less messages for different degrees of connectivity and throughput between cores. The software can transfer arbitrary data from memory or from a local software buffer to another. Finally, it is possible to either setup a single transfer or configure the NAMP to autonomously transfer data from a software buffer (continuous communication mode). This allows the software to not interface the NAMP and instead send data into a simple software buffer.

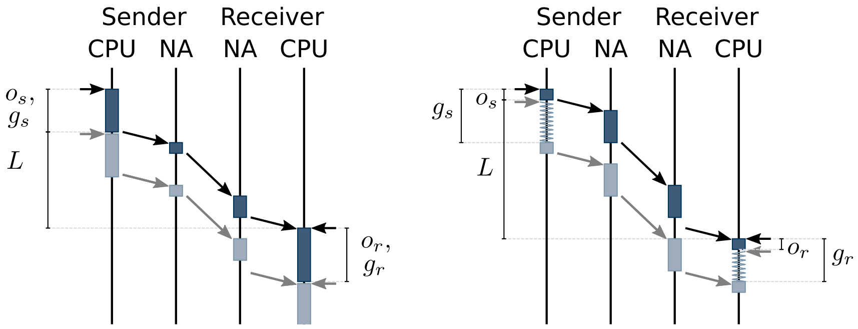
- The NAMP-CC feature which enables efficient collective communication (see Section 3.3). This is the full set of operations: multicast, scatter, gather and reduction operation.
- A self-virtualization feature that allows tasks to use a virtual network adapter (NAMP-SV) without interaction with the operating system (see Section 3.4). It transparently maps virtual endpoint identifiers to physical addresses.
- A method for efficient event notification that is directly integrated into NAMP (see Section 3.5). This hardware operating system queue manipulation (HW-OSQM) performs the wake-up of threads into the operating system ready queue autonomously.
- Support for protection switching of communication channels with the NAMP (see Section 3.6). This allows for dependable task migration with minimal impact on the communication relations of the migrated task.

In this chapter the basic message passing offload of NAMP and the features that improve critical parts of the software stack are discussed conceptually. This includes the derivation of the basic properties and functionalities. The key improvements of the concepts are validated with an evaluation framework that is presented in Section 3.1. Important trade-offs are discussed with respect to their impact on an implementation, but the conceptual discussion abstracts from an implementation sensibly. In Chapter 4 the actual prototype implementation is presented and used to estimate the impact on hardware area and timing. The NAMP features are composable, meaning they can be added individually in an implementation.

This chapter is organized as follows. In Section 3.1 I first present the evaluation framework and evaluation metrics that are utilized in this chapter. After that I introduce the basic NAMP concept in Section 3.2, followed by the features which are presented in Sections 3.3-3.6. Finally, this chapter closes with a summary of the findings.

3.1 Exploration Framework

In the following the proposed concept of an efficient network adapter needs to be validated. The goal is to derive estimates how much improvement can be expected. Beside that variations of the design space need to be explored and evaluated. For that I use a simple analysis framework that focuses on the end-to-end invocations of software and hardware elements in the message passing protocols.



(a) Software-dominated **(b)** Hardware-dominated

Figure 3.1: The LogP model of message delivery

3.1.1 Evaluation Metrics

To evaluate different design points it is necessary to define the optimization objectives:

- The **overhead of protocol processing in software** should be minimized. If protocol optimizations don't lead a further reduction, offloading parts of the protocol helps to further reduce this overhead. This leads to the availability of the processor or device for other computational tasks. For example another thread may be executed, or further processing overlap the hardware operation. Furthermore, hardware implementations are generally more power efficient.

- The **bandwidth** is the upper limit to the number of bytes that can be transferred from one software task to the other software task. It is important that this is the net throughput, which contains all packetization and other effects.

- The **latency** is the time individual messages need to traverse from the source task to the destination task. In this work I consider the end-to-end latency as the time from invocation of the `send()` call until the completion of the matching `receive()` call which is the effective end-to-end latency of a message. The predictability of this objective (mostly as an upper bound) is critical to real-time constraints.

The work presented in this thesis targets to optimize with respect to these three objectives. The objectives relate to each other. This relation is best captured with the so called LogP model [51] and its extensions LogGP [4] and PLogP [119]. The LogP model is named after its four parameters:

Latency L is the latency over the communication infrastructure

Overhead o is the overhead induced on the processor by sending and receiving

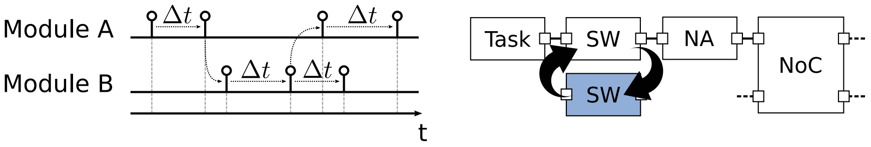
Gap g is the time required between two invocation of a send or receive operation. It is often described as the inverse of the bandwidth

Number of processors P is the number of tiles

The LogP model has originally been developed to provide a model of a platform and message passing implementation that can be used for estimation and optimization of algorithms. But beyond this it has become a well-established tool to describe an implementation and present its result.

Variable Definition In the context of this thesis I use the LogP model parameters as measurement results. Those can serve for calibration of a model, but more importantly they are here used for a quantitative analysis of the proposed improvements. As depicted in Figure 3.1 the sender side and receiver side are differentiated for the overhead and gap as o_s, g_s, o_r and g_r. Furthermore, the message size is a differentiating parameter of a message transaction. Similar to the parameterized LogP model presented by Kielmann et al. [119], I use the message size as a key parameter to analyze the elasticity of the proposed concept. Figure 3.1 sketches the relation of the analysis variables: L is the traversal time after the software handed the message to the network adapter until it arrives at the receiver software. o_s and o_r are the times that the software occupies the processor cores on send and receive operations. The gap is the time that elapses between the actual invocations. It differs between software-dominated implementations (Figure 3.1a) and hardware-dominated implementation (Figure 3.1b). In a hardware-dominated implementation the software is stalled to wait until the hardware becomes available again or can perform useful operations, while in a software-dominated implementation the overhead and gap are equivalent.

The Role of the Network-on-Chip As discussed before, the network-on-chip has been the focus of a lot of research for on-chip communication. Anyhow, I have introduced that it only is one puzzle piece in the end-to-end communication which involves a much larger stack. The network-on-chip influences the analysis in two ways: The topology and the placement of the sender and the receiver directly derives a lower boundary of the traversal latency as $l_{noc,min} = \#\text{hops} \times \frac{cycles}{hop}$. The second influencing factor is the contention which can become very critical. The contention can be understood as the traffic in the network and it manifests in the acceptance rate at which the network



(a) Events & timing annotation **(b)** Simulation model

Figure 3.2: Principle of discrete event simulation

adapter can inject packets into the network-on-chip. This rate strongly depends on the topology of the network and the other traffic. The contention influences the sender gap g_s and thus the bandwidth in a hardware-dominated infrastructure. A highly contended network-on-chip furthermore can significantly increase the latency, especially for packets traversing many hops. In a network-on-chip with dozens or hundreds of nodes the variance in the latency can therefore be multiple orders of magnitude. There have been many approaches to model network-on-chip throughput and latencies [32, 162, 117]. But all models rely on a lot of parameters and an in-depth exploration does not add much to the considerations here. In the following, I mostly abstract from this and only use some average cases of network-on-chip contentions where it makes a difference.

The described set of metrics are in the following used to quantitatively compare the proposed concepts and evaluate their parameters. They are furthermore similarly in the focus of the performance evaluation as part of the presentation of implementation results in Chapter 4.

3.1.2 Discrete Event Simulation

This chapter presents concepts and evaluates them. Only simple cases allow for an analytical evaluation. Due to the inherent parallelism and overlaps a simple discrete event simulation is used to evaluate a concept or explore its parameters with respect to the metrics.

The basic idea of discrete event simulation is sketched in Figure 3.2a: Parts of the functionality are abstracted and only those events that change the state of the simulation are considered. Such state changes are generally interactions between modules or events relevant to the exploration (such as completion events). The events are discrete in the time domain and the times between events can be abstracted which reduces the simulation time significantly. Internal processing between events is modeled with delays and communication between models with events on the interfaces.

In this thesis I use a model based on the SimPy [139] discrete event simulation framework. Here the functional units and modules are modeled as simulation objects as depicted in Figure 3.2b. The modules communicate via generic interface to exchange data with annotated delays. This model allows it to rapidly prototype the protocol flow and data exchanges. As shown in Figure 3.2b the simulation modules can easily be exchanged to explore implementation alternatives.

Parameterization The abstract discrete event simulation that used in this chapter is calibrated with parameters. Those parameters either could be easily derived or are calibrated with the baseline implementation described in Chapter 4. Without loss of generality the parameter values represent the class of loosely-coupled many-core system-on-chip described before.

3.2 NAMP: Network Adapter for Message Passing

As motivated before, the very basic approach to lower the overhead of a message passing protocol on the processor is to improve the protocol, most importantly reduce the number of messages. The protocol used in this work is based on the Multicore Association Communication API (MCAPI) [153]. MCAPI defines an interface and that is optimized for embedded systems and on-chip communication. But as an API it apparently doesn't define the protocol itself. In the following I derive a basic protocol, that can be integrated with MCAPI. A few extensions to the MCAPI interface are proposed to make efficiency improvements better accessible by the programmer. It is important to highlight that the basic protocol and the proposed NAMP are not specific to MCAPI. Instead, it serves as an industry-standard reference.

After deriving the base protocol, the performance of it is evaluated for state-of-the-art hardware support. Derived from this evaluation, the hardware offload of the protocol is discussed as a highly-optimized "Network Adapter for Message Passing (NAMP)". This is namely the implementation of the progress engine in hardware which are the basic algorithms to send and receive messages.

3.2.1 Message Passing API

The message passing protocol can be easily mapped to MCAPI and a few extensions are added. The basic MCAPI interface functions are prefixed with `mcapi_` and proposed extensions are in the following prefixed with `mcapix_`. As said, the API itself does not propagate an implementation in itself. In fact the reference implementation of MCAPI is implemented for shared memory platforms.

Due to the asynchronous nature of the message passing protocol, the task may have to actively wait for a certain amount of time on invocation until a message from a remote tile arrives. As other message passing protocols, MCAPI includes *non-blocking* function calls that are identified with the commonly used `_i` suffix. The return after the request started and the task can then overlap the waiting time with other computations and later check the result using the `mcapi_test()`, `mcapi_wait()`, `mcapi_wait_any()` or `mcapi_cancel()` functions. *Blocking* functions instead return only after the operation completed.

3.2.1.1 Basic API Functions

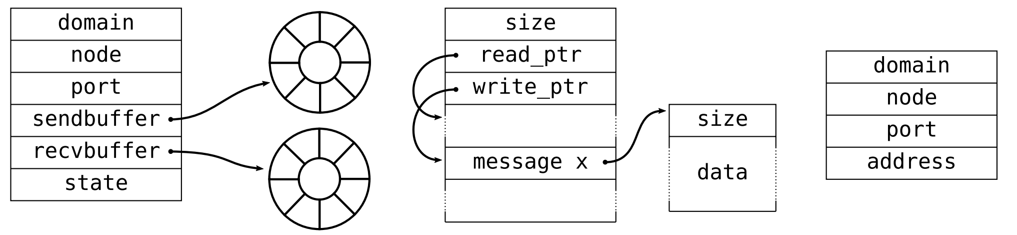
MCAPI defines a set of basic functions for initialization and introspection. The functions `mcapi_initialize()` and `mcapi_finalize()` are used to initialize the environment. The initialization can take implementation-specific node attributes that are highlighted throughout this thesis when needed. Two helper functions `mcapi_domain_id_get()` and `mcapi_node_id_get()` allow to read the `<domain,node>` tuple this instance was configured for. Finally, node attributes are (re-)initialized `mcapi_node_init_attributes()`. Individual attributes are modified and queried with `mcapi_node_set_attribute()` and `mcapi_node_get_attribute()` respectively.

These functions essentially create, manage and destroy some lists and other data structures. They don't imply any inter-tile communication per se and most importantly they are called rarely (once), so that there is not much gain by offloading them to hardware.

3.2.1.2 Endpoints API

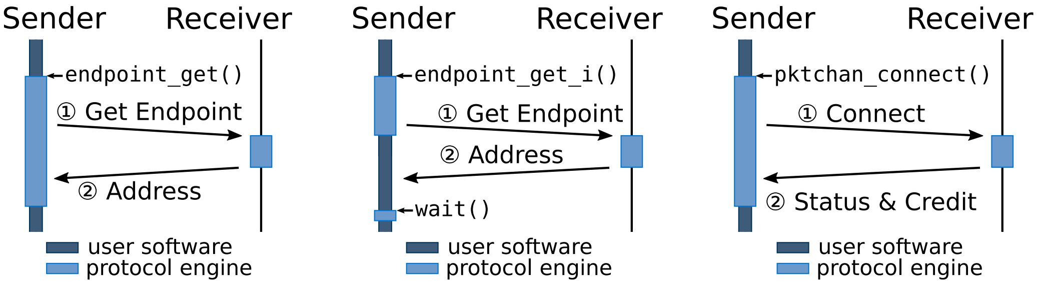
After a node instance has been initialized, the actual message passing functions can be called. A message passing interaction first needs to setup the endpoints that form the communication relation. The software differentiates between local and remote endpoints.

Manage a local endpoint The function `mcapi_endpoint_create()` allocates the local buffers and data structures for a given port number. This endpoint can then be addressed as `<domain,node,port>` globally. The endpoint data structures are outlined in Figure 3.3: The endpoint contains its own address, pointers to buffers and some state. The state is more important with the NAMP features discussed later in this



(a) Local endpoint data structure **(b)** Buffer data structure **(c)** Remote endpoint data structure

Figure 3.3: Endpoint data structure



(a) Blocking retrieval **(b)** Non-Blocking retrieval **(c)** Channel connect

Figure 3.4: Remote endpoint management and channel connect

chapter. First it is essentially used to capture the life-cycle of an endpoint (ready, deleted). An endpoint can point to one or two buffers, because the sending buffer is optional (see below). As sketched in Figure 3.3b, each buffer is configured to hold a maximum number of messages (size) and pointers to the next write and read position in the buffer. Each message has a size and data, more details on the implementation of the buffer are given in Section 4.2.2. The functions `mcapi_endpoint_set_attribute()` and `mcapi_endpoint_get_attribute()` are used to change this and other properties such as the size of buffers and maximum size of individual messages. The MCAPI extension `mcapix_endpoint_create()` is added that takes such attributes and it optimizes the creation process of an endpoint. Finally, `mcapi_endpoint_delete()` is called to deconstruct an endpoint. All of the local endpoint management functions do not involve any interaction with other tiles.

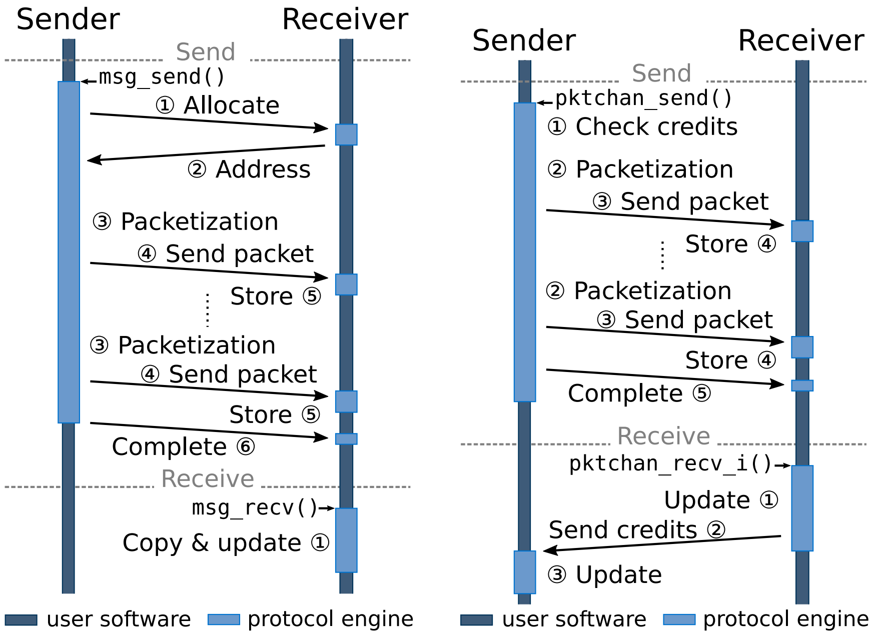
Retrieve a remote endpoint A handle of a remote endpoint is needed locally, most importantly at the receiver side of communication. Locally this handle is stored in a data structure as sketched in Figure 3.3c. It contains the endpoint address tuple and the memory address in the remote tile, which there points to a local endpoint data structure as described above (see Figure 3.3a). The setup of this data structure is part of the `mcapi_endpoint_get()` function. In the presented implementation it queries the remote tile for the remote address of the endpoint data structures as depicted in Figure 3.4a. The non-blocking variant is also sketched in Figure 3.4b, but is only given for other functions if needed in the following.

Channel Setup Channels are the communication medium for connection-oriented communication. As motivated before they are commonly used in the embedded system domain where a data flow programming model runs on top of a message passing infrastructure. In channels only one sender can transmit messages to the receiving endpoint. Figure 3.4c shows the common case where a connection is established by the sender by calling `mcapi_pktchan_connect_i()`. The receiver acknowledges the connection request if eligible and returns a credit to the sender. The credit determines how many messages the sender can send on this channel before waiting for new credit. Before communication starts, both communication partners must call `mcapi_pktchan_send_open_i()` and `mcapi_pktchan_recv_open_i()`. The open is decoupled from the connection because it is generally allowed that the connection is established from a third party and the details are out of scope of this thesis.

3.2.1.3 Connection-less Communication

Once the communication is set up, messages can be exchanged. As introduced, two types of communication can be used. In connection-less communication (or “messages” in the MCAPI nomenclature) the protocol requires a rendezvous-style allocation of a message buffer before sending it. As mentioned in the introduction the protocol alternative to eagerly send the message has the risk of backpressure on the network-on-chip that significantly impacts the performance, or alternatively has much higher memory demands that outweigh the benefits. The function pair `mcapi_msg_send()` and `mcapi_msg_recv()` or their non-blocking variants `mcapi_msg_send_i()` and `mcapi_msg_recv_i()` are used to exchanges message. `mcapi_msg_available()` can be used by the receiver to check for incoming messages.

Figure 3.5a shows the protocol for a connection-less message transmission. The protocol engine sends a message allocation request for the remote endpoint to the destination (1) and either a `NACK` message or the message of the allocated message are returned (2). The remote protocol handling checks if the receive buffer at the addressed endpoint



(a) Connection-less Message Transfer

(b) Connection-oriented message transfer

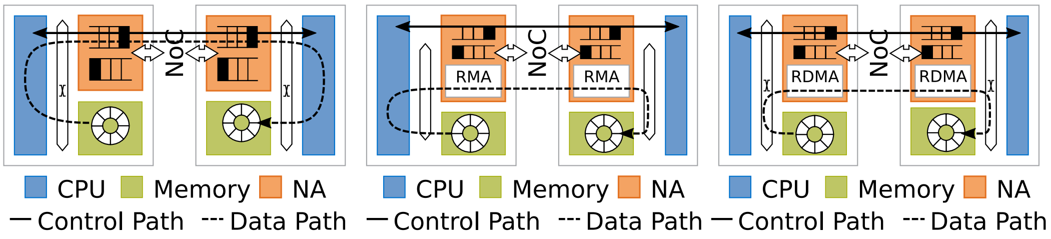
Figure 3.5: Basic message passing protocol

can receive a packet and then returns the message address. The size of an allocated message buffer is set to 0 until the message is complete. The sender then generates network-on-chip packets for the message (3) and sends them as protocol messages to the receiver (4). At the destination, the message chunk is stored in the message buffer (5). This is repeated until the message is complete. Each of the messages contains the offset of the data chunk in the message. After the last message a completion message is sent containing the overall sent data size (6).

Receiving a message is much simpler. If a message is available, it is copied to the buffer provided by the user. After that, the buffer data structure is updated and the read pointer is moved forward.

3.2.1.4 Connection-oriented Communication

Connection-oriented communication is only allowed with endpoints that were previously connected as a channel. Also the polarity (sender vs. receiver) needs to match the channel setup. The sender can then call `mcapi_pktchan_send()` or `mcapi_pktchan_send_i()` to transmit a message on this channel. As depicted in Figure 3.5b it first checks for sufficient credits. If there are no credits left it has to wait for the receiver to send new



(a) Buffer-based NA **(b)** Remote Memory Access **(c)** Remote Direct Memory Access

Figure 3.6: Data flow using different basic network adapters

credits. After that, the message transfer is identical to the connection-less case. The advantage is apparently that the allocation phase can be skipped as it is clear that the receiver can accept new messages. The latter has two significant properties: First, the sender has to constantly retry in the case of an allocation failure. Second, this protocol exchange is a critical protocol roundtrip. Commonly, some dead time arises until the answer arrives.

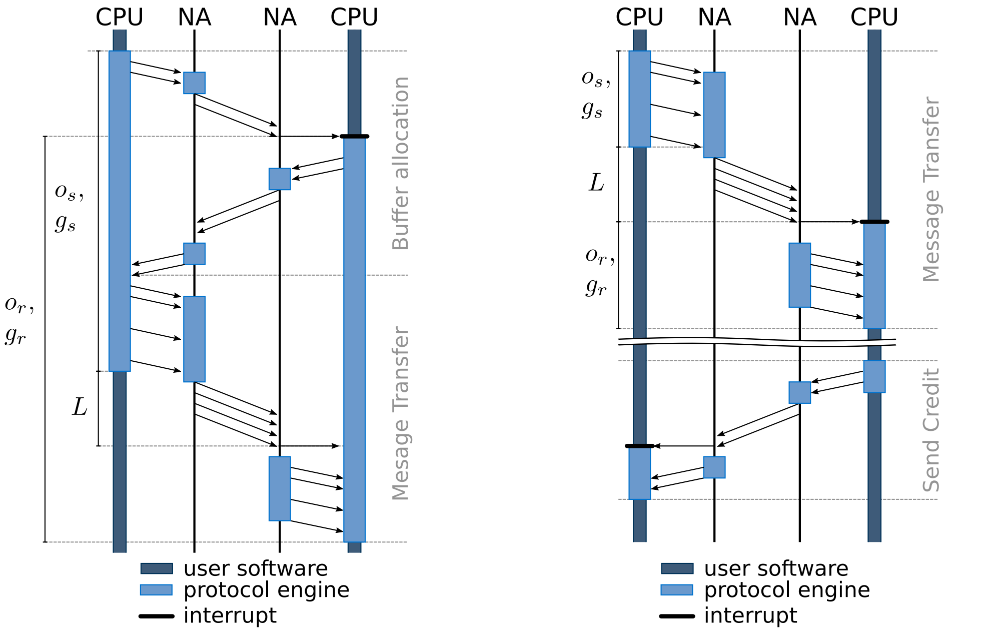
After the message is complete, the receiver can pick it up using `mcapi_pktchan_recv()` or `mcapi_pktchan_recv_i()`. The receiver does not supply a buffer to copy the data to, but instead the *zero-copy* concept is standardized: The system returns a pointer to a system buffer and the receiver mandated to release this buffer after the data is not needed anymore using `mcapi_pktchan_release()`.

3.2.2 Evaluation of the Base Protocol

Now that the base protocol is established, it is analyzed for implementations with the state-of-the network adapters introduced in Section 2.4. For that I derive sketches of the message sequence charts, analyze the involved delays and interactions and compare the implementations. This directly serves as a comparison baseline for the proposed NAMP in the following. Figure 3.6 sketches the data flow of the bulk message data from the buffer of the source endpoint to the buffer of the source endpoint.

3.2.2.1 Buffer-based Progress Engine

As depicted in Figure 3.6a the entire communication protocol is handled by software, including the message generation. Packetization and de-packetization are performed in software. The protocol is sketched in Figure 3.7 for connection-less communication



(a) Connection-less Message Transfer **(b)** Connection-oriented Message Transfer

Figure 3.7: Basic message passing protocol

(Figure 3.7a) and connection-oriented communication (Figure 3.7b). The main message transmission is identical, but the flow control apparently differs.

In the case of *connection-less communication* the software forms a buffer allocation message with the endpoint as parameter. The software writes flit per flit to the network adapter which stores them into a buffer. Once the packet is complete, the transfer starts. This waiting is necessary to keep up with the network speed and being able to transmit one flit of the packet after the other if the packet allows. Otherwise, serious contention can significantly impact the network-on-chip. After traversing the network on-chip the network adapter on the receiver side raises an interrupt to trigger the progress engine on the CPU. It reads the protocol message and performs the operation. It sends the reply to the sender (the address of the reserved message buffer position). The sender actively waits for this reply and then sends the message payload over the network. As the payload is usually larger than the maximum packet length in the network on chip, it needs to be split up into multiple packets. For each packet the CPU is involved to read it and copy the data to the buffer. With the first packet it involves the interrupt handler. But as the next packets are supposed to arrive in a time frame smaller than the interrupt handling, it waits after that.

For *connection-oriented communication* the actual message transfer is identical and the costly remote buffer allocation is not necessary. The sender can continuously send messages as long as it has sufficient credit on the channel. The receiver has to update the credit periodically. There is a trade-off involved for the threshold when to send credits: Sending them after each receive involves a lot of messages, while the backpressure can impact the sender when sending them only after used up. A sensible threshold is in after half of the credits have been used.

From the message sequence chart the evaluation model can be derived. As mentioned, the basic operations are calibrated with numbers gathered from the prototype implementation. The operations depicted in Figure 3.7 are composed of basic operations. The model results can be found in Figure 3.8. The metrics are plotted with the message size as the variable. Beside that it is parameterized for different network-on-chip maximum packet lengths l (in flits, here 4 bytes per flit). The latter influences the number of packets sent for the bulk message data. The number of packets sent varies between different lengths. As each transfer contains h header flits (here $h = 2$) the total number of transmitted flits increases with lower maximum packet sizes: $P = \lceil \frac{S}{l-h} \rceil$ is the number of packets, where S is the message size.

Figure 3.8 plots the overhead and latency of the software on the processor core for *connection-less* communication. Beside the aforementioned parameters the plot contains spans for each individual result that show the variation depending on the network-on-chip contention. As mentioned before, this evaluation is orthogonal to the network-on-chip latency, but the contention has a significant impact here. Backpressure from the network-on-chip manifests in the software execution time. The lower bar shows no significant traffic in the network-on-chip, while the upper bar shows a very contended network-on-chip. The contention here is modeled as the service time of the network-on-chip interface.

Interpretation As already visible in the message sequence chart in Figure 3.7 the message passing is largely dominated by the software overhead. The network traffic can influence the overhead by two orders of magnitude. Two effects influence the overhead: The number of packets that are generated by the software are dominating the overhead for large packets. But until a packet size of around 128 byte the remote buffer allocation dominates the overhead and latency as a basic offset, and only after that the overhead diverges for larger l. The receiver overhead has a similar pattern, but an offset due to the impact of the number of interrupt service routines can be observed. Finally, the end-to-end latency also follows a similar pattern. Due to averaging effects the influence of

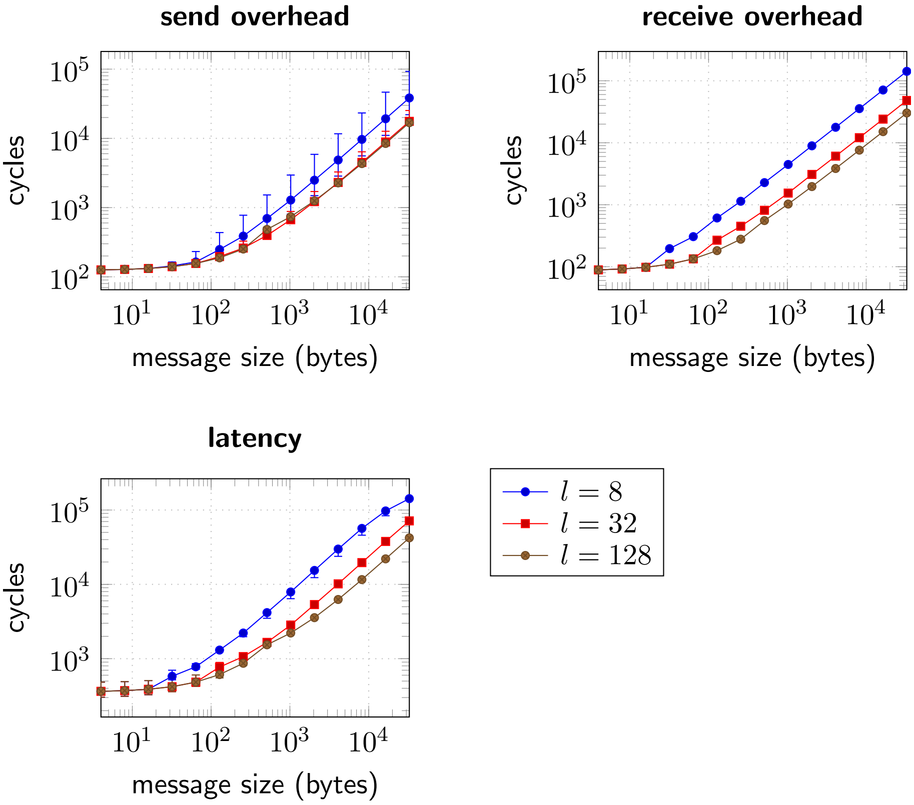


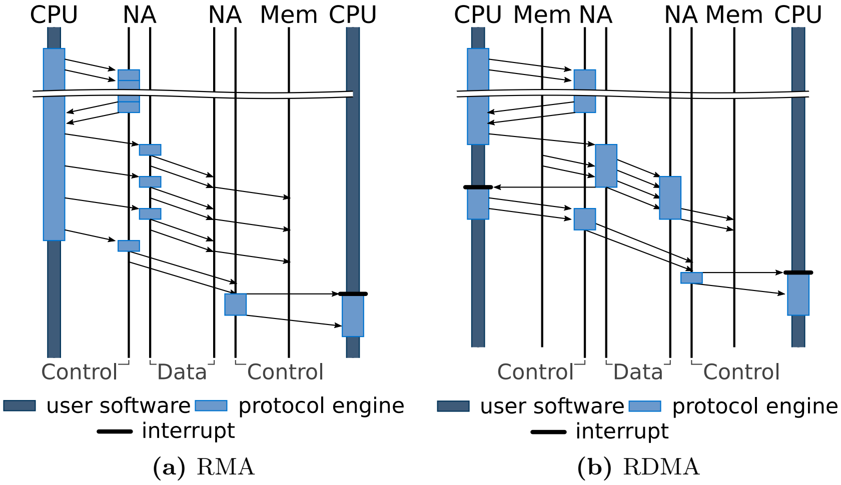
Figure 3.8: Evaluation of message passing with a simple buffer-based progress engine

the network contention is more limited than for the send overhead where the interaction of wait times and concurrent message transfers becomes visible.

3.2.2.2 RMA vs. RDMA

The massive overhead due to message handling of a simple buffer-based network adapter with software protocol engine can be mitigated with offloading. The most apparent and simple is offload of the data path. Figure 3.6 highlights the difference between buffer-based, RMA-based and RDMA-based network adapters. The basic idea of RMA is to get rid of the invocation of the remote CPU in the data path. The remote invocation is in general an expensive interrupt service routine. But RMA still has a similar impact on the sender software, because the data path still goes through the CPU (see Figure 3.9a). In RDMA the data path is handled entirely by hardware (see Figure 3.9b).

Figure 3.10 plots the evaluation metrics for RMA-based and RDMA-based implementations and compares them to the basic buffer-based implementation. In this plot and



(a) RMA

(b) RDMA

Figure 3.9: Basic message passing protocol using Remote Memory Access and Remote Direct Memory Access – shortened connection-oriented protocol

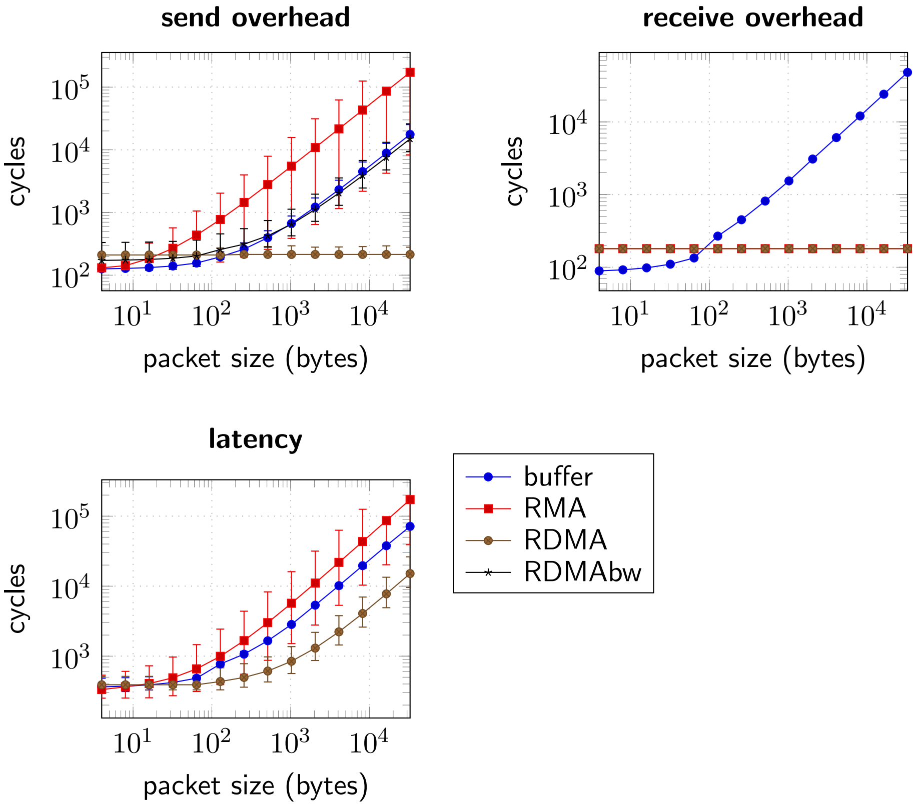


Figure 3.10: Evaluation of message passing based on RMA and RDMA

throughout the remaining thesis the network-on-chip packet length is fixed to 32 words which are assumed to be 4 Byte each without loss of generality. The error bars again denote the variation due to network contention. Figure 3.10 furthermore plots the overhead for the sender for the buffer-based implementation, RMA and RDMA. The evaluation here and in the remain of the chapter assume that the software can do something different useful while waiting for replies or a controller to complete. As also depicted in the message sequence charts an interrupt is used to notify about those completions. As a comparison the plot also contains an RDMA implementation that does busy waiting (RDMAbw). The receive overhead are plotted where RMA and RDMA are identical. The end-to-end-latency is finally depicted in Figure 3.10.

Interpretation As expected from the discussion, RMA and RDMA only induce a constant overhead for the receiver. The perform only the allocation operation and the finish operation, which are both independent of the packet size. Similarly, the sender overhead becomes constant for RDMA. The RMA overhead on the sender side is higher than the buffer-based implementation, because packetization and number of network packets is higher (one packet per word, multiple headers per packet). Also, the RMA implementation is therefore less robust against variations in the network contention. The RDMAbw implementation can be seen as a combination of RDMA and buffer-based implementation, each time the worse for the packet size. This again motivates to not wait actively for completion of operations. Finally, the trend of the latency shows a combinations of the described effects on the packet delivery time.

3.2.3 NAMP Basic Concept: Full Progress Engine Offload

The previous analysis has shown that the implementation of message passing protocols can benefit a lot from offloading the data path to a remote DMA engine and a bit less from a remote memory access engine. Anyhow, there is quite a high impact of the control path operations. There have been a few approaches to offload control path to hardware too [110, 52]. But those approaches are very limited in their scope and functionality. They typically only target one specific method for message passing (e.g., connection-less vs. connection-oriented, buffered, etc). While they are interesting candidates for one data point in the design space of message passing, the proposed *Network Adapter for Message Passing (NAMP)* tackles this design space in a much broader scope by providing a large variety of communication modes along with an efficient baseline concept.

Figure 3.11 depicts the basic concept of NAMP. It replaces the control path operations in software (see Figure 3.6) with a hardware-implemented protocol engine. Those are

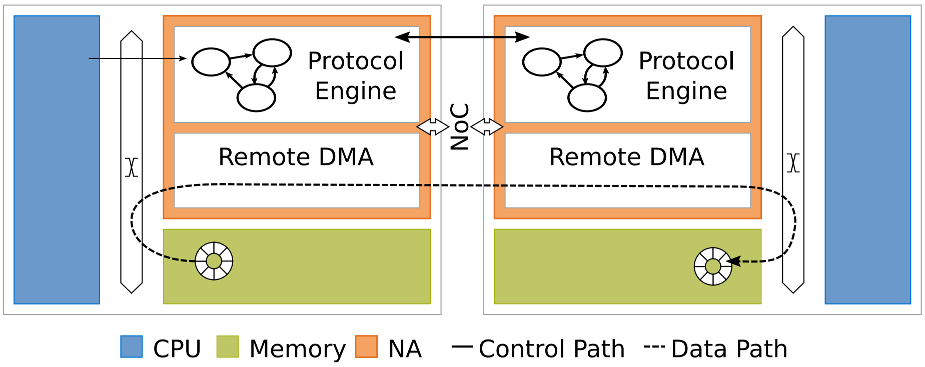


Figure 3.11: Basic overview of control and data flow with the Smart Network Adapter

the bottom line contributors to overhead and latency. As mentioned, the most critical is the turnaround allocation message for connection-less messages, because the sender is either idly waiting or needs to be interrupted (with the high cost of the interrupt handling). Even for connection-oriented communication the invocation of software at credit update messages is significant.

Beyond the imminent advantage to the processing of one message transfer, the benefit of offloading the control path processing to the hardware is even higher in the presence of multiple outstanding operations. The impact of waiting for a retry of buffer allocation and handling of credit messages – the progress engine – can sum up significantly with multiple outstanding operations.

The basic NAMP approach covers the following features:

Data Path and Control Path Offload The performance critical parts of the protocol are executed in hardware, which are the message transfer operations. Enumeration and connection management are still handled by software as described in Section 3.2.1.

Data Structures in Memory Data buffers and endpoint descriptors are stored in the tile local memory. Thereby it provides greater flexibility and less impact on the hardware area.

Configurable Number of Slots Each slot represents one ongoing transfer. The slot encodes the state of the transfer, for example a slot progress can be stalled due to a pending allocation response or while the data is transfered.

Non-Blocking Progress Engine The protocol message between two communication partners are non-blocking, which means that the receiver side always immediately replies to a protocol message from the sender. Thereby, the receiver itself is *stateless*.

Connection-oriented and Connection-less messages Both message types are supported by NAMP with a common software interface. As mentioned there is a major overlap in functionality for both message types.

Thread-Safe, Non-Blocking Circular Buffer Design The buffer data structure is designed in a way it can be efficiently interfaced by software and hardware. It can allow multiple concurrent readers and multiple concurrent writers. It is implemented as a non-blocking data structure, which means for example that there is no impact of stalled threads on the other readers and writers (no locks).

Receiver-Buffered and Dual-Buffered Messages are always delivered to a remote buffer. This decouples the hardware from the software. On the sender side data can be either send from a memory address or from a buffer. This eases the software handling of memory addresses in concurrent outstanding operations.

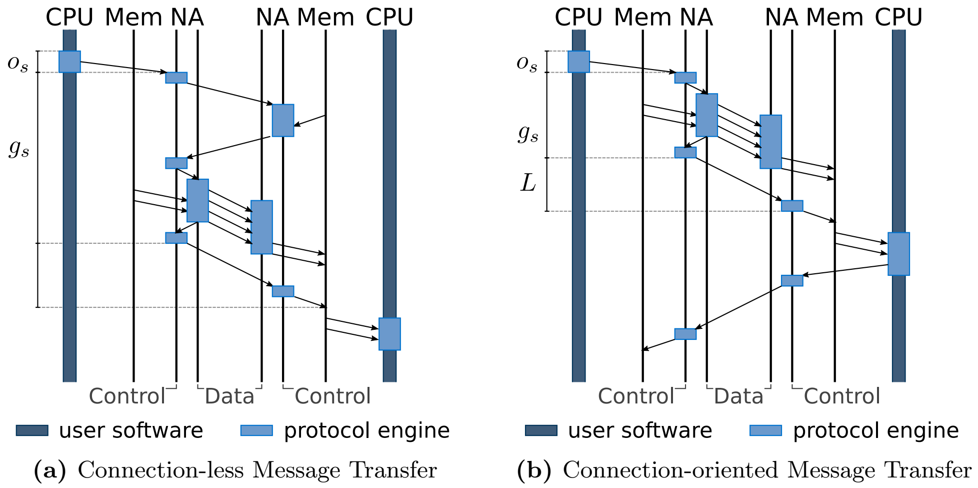
Message Completion and Continuous Communication Mode In the basic NAMP design the completion of a message operation can be configured to be signaled by an interrupt or via polling. The HW-OSQM concept presented in Section 3.5 improves the event signaling beyond that. Finally, a NAMP slot can be configured to continuously transfer messages from a sender buffer to a receiver buffer.

3.2.3.1 Evaluation of Full Progress Engine Offload

Figure 3.12 sketches the message passing protocol variants when the control flow is also offloaded to the network adapter as in NAMP. After starting a message passing transfer, the software immediately configures the network adapter to handle the transfer and can then return to other processing. The protocol engine takes care of flow control and then configures the actual bulk data transfer. The overhead is thus constant.

Figure 3.13 shows the sender overhead of the NAMP and compares it to RDMA, both for connection-less communication. As another variable it adds the number of retries for an allocation message. This is modeled as a poisson process which is a common statistical process in queuing theory. It has the parameter λ and different parameters are evaluated here. Figure 3.13 also compares the latency accordingly.

Interpretation Figure 3.13 plots the evaluation comparison between RDMA and NAMP, with a reduced x axis compared to the previous plots for more detail on the interesting



(a) Connection-less Message Transfer **(b)** Connection-oriented Message Transfer

Figure 3.12: MSCs of NAMP-based message passing protocols

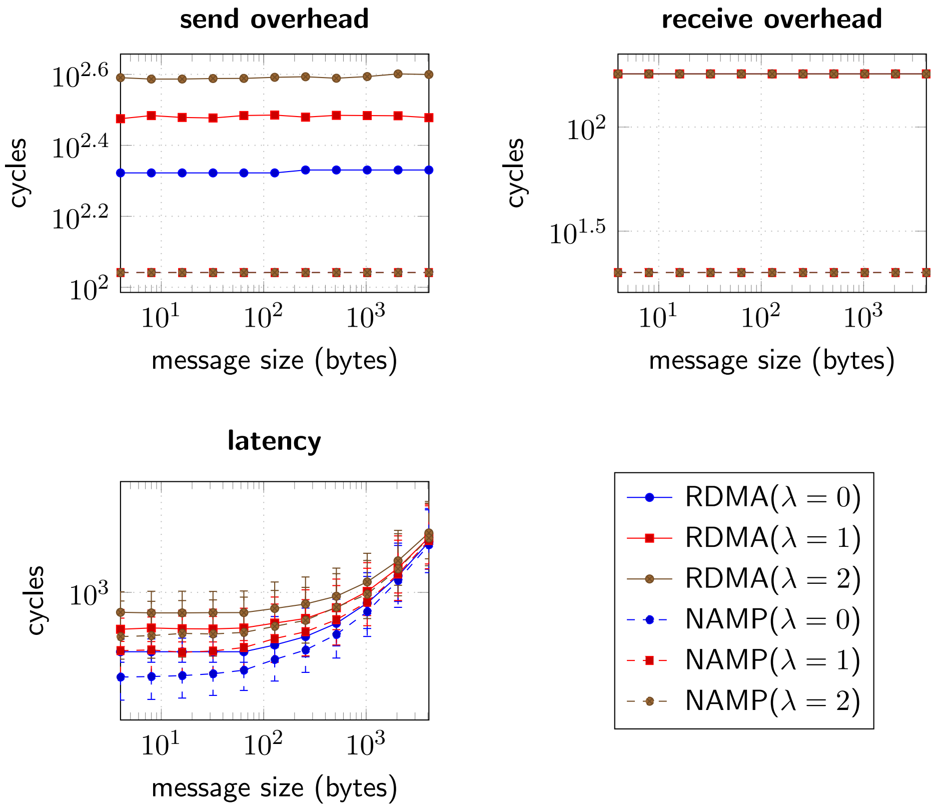


Figure 3.13: Evaluation results of NAMP compared to RDMA at different waiting times

part. Essentially two results are interesting: First, the configuration times required to setup the RDMA and NAMP respectively, and the protocol handling for allocation and finish in case of RDMA. It can be observed that the overhead for NAMP is smaller than for RDMA as it only needs to setup a single descriptor and then is notified once the transfer completes. Second, the difference grows with mean waiting time for an allocation (poisson process with mean λ), which the NAMP configuration is independent of. The latency is again dominated by the bulk data transfer delay and the time needed for allocation. As the NAMP hardware equally needs to wait and retry allocations, the general shape of the plot is similar but the latency is overall smaller.

3.2.4 NAMP Buffer Features

In the following I derive some important features around the role of buffers in the message passing system.

3.2.4.1 Non-Blocking Circular Buffer

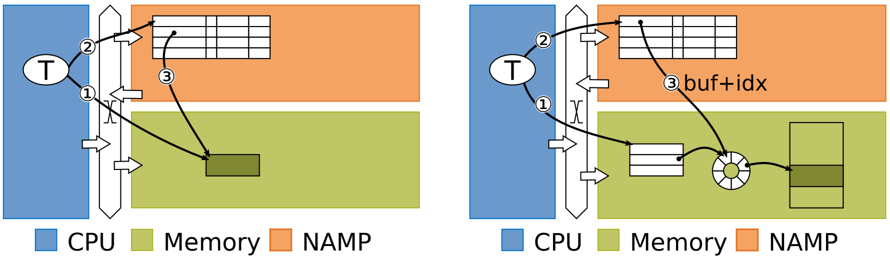
All buffers are stored in the local memory in the tile. They are interfaced by both the hardware and the software. Thus the buffer data structure is designed in a way that it can be efficiently accessed and modified by hardware and software. The basic buffer data structure as depicted in Figure 3.3b for example requires proper alignment for easier calculation of offsets. Beside that the index counters should wrap at boundaries that are power of two, which is the size of the buffer and the maximum message size.

Another important property is that the operations on the buffer only involve one variable that needs concurrency protection. This enables the circular buffer as a *non-blocking data structure*. Instead of using a locking mechanism between the software and hardware this leads to a progress guarantee for the operations.

The buffer implementation and operations are further detailed in Section 4.2.2 as part of the prototyping discussion.

3.2.4.2 Buffered vs. Unbuffered Data Sending

As mentioned before, there is always a buffer on the receiver side. Each message transfer involves the pushing of the message into the buffer. This is different on the sender side. As depicted in Figure 3.14 a sender can either configure the NAMP to send an arbitrary block of data from the memory (Figure 3.14a) or from a buffer (Figure 3.14b). In the following I briefly introduce the options and compare them.



(a) Unbuffered Send **(b)** Buffered Send

Figure 3.14: NAMP Send Features

Unbuffered Send The software writes the data to be transferred to an arbitrary address in memory. After that it configures a NAMP slot to send data from that address and the size of the data, along with the destination endpoint (not sketched in Figure 3.14a. The NAMP then handles the message protocol (connection-less or connection-oriented) and transfers this block of data to the destination. Once the NAMP is done, the sender can re-use the data block or free it. It requires notification of the NAMP as discussed above, and the slot remains reserved for a certain amount of time until the task received the completion notification.

Buffered Send The software allocates a buffer for the sender operation. After writing data to the buffer it configures the NAMP slot with the buffer address and the index it wrote the data to. The NAMP then first reads the data base address from the buffer and calculates the address of the data block. After completion of the data transfer, it frees the buffer element. It is not necessary to notify the software, because it implicitly communicates with the NAMP via the buffer.

Please note that the usage of the buffer differs in that case from the description above. The read index is not used at all, which is okay as the NAMP is the only reader and executes on a per-index basis. Using buffered send is advantageous when one task sends connection-less to multiple remote tasks. This has limitations if the delivery latencies vary a lot because a pending send can block new transfers while the buffer is nearly empty. In such cases, multiple, very small buffers are better suited as the memory overhead of the data structure is small.

Table 3.1 summarizes the differences between the buffered sender mode and the unbuffered sender mode. The apparent advantage of the unbuffered mode is that it does not require extra memory. The software can just use a variable and send it to the remote. In the buffered case the data must be pushed into the buffer. This usually requires

	Unbuffered	**Buffered**
Memory requirements	Low, data only	High, data structure and buffer elements
Data items	Arbitrary	Buffer element
Notification	Explicit	Implicit
Release slot	Explicit, by software	Implicit, on completion

Table 3.1: Comparison of Buffered and Unbuffered Send

copying data from a the original data field to the buffer. Alternatively, the software can perform the reservation function `mp_cbuffer_reserve()` from Listing 4.2 and assemble the data directly into the buffer. The second important difference is in the port-transfer handling, namely notification and the slot life cycle. In case of non-buffered send the slot gets into a "complete" state and the software gets somehow notified about the completion (notification is discussed subsequently). Hence, the slot cannot be used again immediately. With the buffered send the NAMP immediately clears the slot after freeing the buffer element. The software is not directly involved, but implicitly can re-use the NAMP slot in other contexts.

3.2.4.3 Continuous Communication Mode

The aforementioned buffered send mode is designed can be useful to increase the throughput under certain circumstances. As discussed, this is predominantly for connection-less messages. There is another mode which is primarily designed for connection-oriented messages but not limited to it. The basic idea of the continuous communication mode is similar to the buffered send mode. The software writes to a buffer and the NAMP transfers data from the buffers and interfaces the buffer data structure. The continuous communication mode goes one step further and doesn't release the slot as soon as a transfer is done. Instead it is configured to continuously send data from the buffer to a remote endpoint. For connection-less messages this is useful when only one remote is addressed (such as multiple-to-one) and for connection-oriented it is the channel.

As depicted in Figure 3.15 the software initially configures the NAMP to perform a continuous communication mode operation from a send buffer to the remote endpoint. After that the software only interacts with the buffer and pushes data into it whenever the buffer is ready. The NAMP also only interacts with this buffer and sends data as soon as it becomes available. This has a massive advantage as it removes all interaction between the NAMP and the software. The software virtually just writes into software buffers on one end and reads from software buffers on the other end.

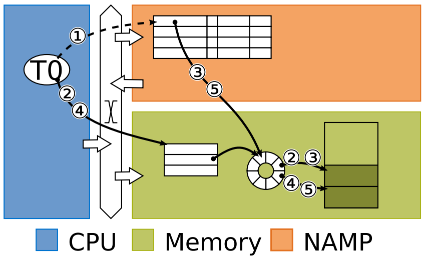


Figure 3.15: NAMP continuous communication mode

In continuous communication mode the NAMP operation is more complex, because it does not only involve a single indirect addressing, but also needs to implement the buffer pop operation (move the read index). Figure 3.15 sketches the data flow from the software writing to the buffer using the defined interface and the NAMP reading from the buffer accordingly. More details on the complexity of the continuous communication mode can be found in subsequent discussions.

3.2.5 NAMP Design

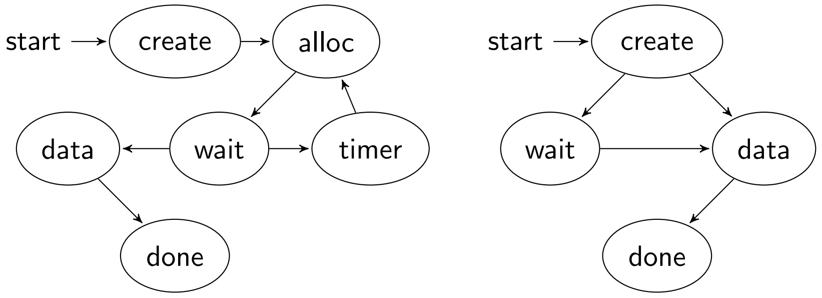
In this concluding section I discuss the design of the NAMP in a conceptual way. The actual implementation is presented in Chapter 4.

3.2.5.1 Functional Partitioning

The NAMP concept can be separated in two parts: The *initiator side* is responsible for the protocol progress and setup of the data transfer via RDMA. The *target side* responds to the protocol messages generated by the initiator side.

Initiator The initiator provides the concept of *slots*. Each slot corresponds to one outstanding message transfer. Hence, slots are a shared resource if the software wants to transfer multiple messages concurrently as there are slots. A sensible design point for the number of slots requires on the application scenario and is discussed as part of the implementation evaluation in Chapter 4. Each slot consists of two parts: The message configuration registers that are set by the software and describe the message transfer. Beside that each slot has a state that encodes progress of the message.

This state of a slot mostly to a message lifecycle as sketched in Figure 3.16. The lifecycles of connection-less and connection-oriented messages can be derived from the message sequence charts in Figure 3.12. The lifecycle diagrams do not describe the conditions and actions for transactions, which is derived in the following. Generally,



(a) Connection-less **(b)** Connection-oriented

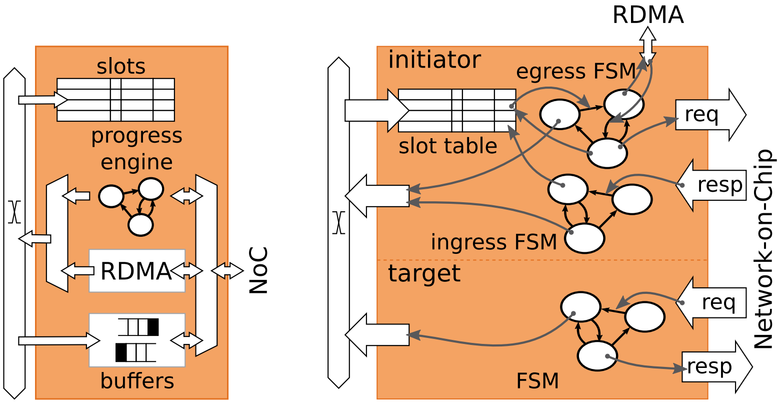
Figure 3.16: Lifecycles of messages during protocol handling at the sender

a connection-less message (Figure 3.16a) needs to generate an allocation message and wait for the response. If the response is a successful allocation, the allocated buffer slot address is stored. Otherwise a timer is set before retrying the allocation, because sending it too fast after the first most probably leads to the same result. In the case of connection-oriented messages (Figure 3.16b) the message can generally be sent when credit is available. Otherwise it requires waiting for a credit update. In both cases the RDMA is configured once the target is ready to receive the message. After the RDMA transfer is completed, the message is done.

Apparently not all transitions in the lifecycle are triggered by the progress engine itself, but also responses from the receiver can transfer the state: On arrival of an allocation response the connection-less message state transfers either to data processing or activates the timer. Similarly, the arrival of new credit triggers the transfer from waiting to data transmission for connection-oriented messages. Finally, there is one other class of messages: credit updates. They are generated on the receiving side and they are technically sent using the initiator path.

Target As introduced before, the target side does not need to keep track of the state of individual messages. All transfers are controlled by the sender side and the receiver only reacts to the protocol messages from the sender. It means that it can process incoming protocol messages immediately at any time. This is an important property in avoiding deadlocks and performance bottlenecks.

Figure 3.17a gives an overview of the functional blocks that form the NAMP. As mentioned before, the NAMP does not offload the setup phase operations as they occur rarely. Hence, the NAMP still contains buffers and the software can directly exchange network-on-chip packets via them. For the actual message passing transfer operations



(a) Design overview

(b) Functional partitioning

Figure 3.17: NAMP design overview and functional partitioning

the second slave interface on the bus is for the slot configuration, as described above. The progress engine handles the message transfers from the slots and tracks their individual progress in the slot state. The progress engine also sends and receives messages on the network-on-chip interface, the protocol messages. Finally, the NAMP contains an RDMA controller for the bulk data transfer.

The central element of the NAMP is the progress engine. Figure 3.17b sketches the functional partitioning between the sending and receiving operations around the progress engine. Apparently, the sender part is more complex than the receiver part. The functional partitioning also reflects in the assignment of network-on-chip channels. To mitigate message-dependent deadlocks two channels are used (for example see [193]). Those channels can be separate physical networks or virtual channels in the network-on-chip. The sender generates protocol messages on the request channel (`req`), which are at another tile's receiver part are the incoming messages. The receiver then sends its responses on the response channel (`resp`).

On the initiator side two FSMs are used: The *egress FSM* is responsible for the protocol handling and generates the protocol requests. It is triggered by changes in the slot table and performs protocol actions for one of the slots at a time. It generates messages and updates the slot table according to progress. Beside that it can access the bus to get further data from the data structures in the tile local memory. Finally, it interfaces the remote DMA controller to set up data transfers. The initiator *ingress FSM* is triggered by response messages and updates the slot table according to the responses. Beside that it accesses the memory.

The target part of NAMP is straight forward. The FSM is triggered by incoming protocol requests. It then accesses the data structures in the tile local memory and assembles protocol response messages as needed.

While this is a bird's view on the NAMP design, it gives an impression about where the complexity is in the design. In the course of this section I develop the NAMP engine based on the lifecycle and this functional partitioning.

3.2.5.2 Bus Interface & Operation Atomicity

The progress engine has two bus master interfaces which are most probably multiplexed inside the NAMP: One for the initiator accesses and one for the target accesses. The NAMP does not depend on a specific bus interface. An implementation needs to provide very essential read and write accesses and atomic read-modify-write operations.

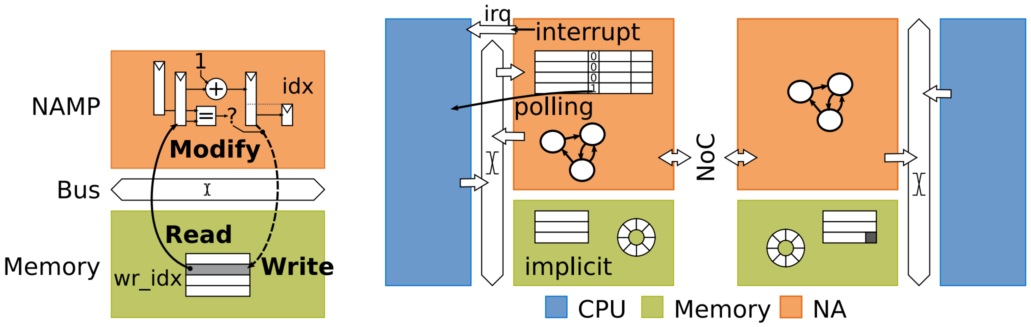
Initiator As depicted in Figure 3.17b both the egress FSM and the ingress FSM access the bus interface. The egress FSM for example needs to look up the endpoint data structures and credit information, or manipulate a sender buffer. The ingress FSM receives responses from the target, which may require an update of the endpoint data structure.

Target The target accesses the bus to perform the requested operations. Actually all target operations involve a bus access, to manipulate the buffer or update a credit.

Operation Atomicity The interaction with the buffers requires atomic operations to guarantee data integrity. As long as there is only one NAMP engine, only the receiver side may require atomic access. If there are no software threads that can write to the same buffer, atomicity is essentially not needed. In an implementation as presented in Chapter 4 support for atomic operations may therefore be configurable.

Figure 3.18a depicts an example for a read-modify-write cycle of the NAMP. When pushing data to the receiver buffer and the potential of software thread interference, the NAMP has to manipulate the index atomically. It reads the current index to check if the buffer is full (compare Listing 4.2) and then tries to atomically increment that index. For that it first *reads* the index again and checks if it still matches the old value. It then conditionally *modifies* the value and conditionally *writes* it back. The bus is held during the entire operation[1] and only released after the write or if the comparison fails. Figure 3.18a also depicts how the actual idx is extracted from the stored value as the

[1] Interconnects that aren't shared medium, such as AXI, have extensions that resemble this.



(a) Read-modify-write **(b)** Notification options

Figure 3.18: Read-modify-write example and notification options

lower bits of the value, where the number of bits is the `capacity` field value (modulo by power of two).

3.2.5.3 Notification

There are two events that the software may get notified about. First, when a *send operation is completed* the software for one can resume on a blocking call. Beside that the software knows that the NAMP slot can now be re-used and potentially that the sent data block can be reused or freed. Second, the software may wait for a *blocking receive operation*. The basic NAMP supports configurable notification for both cases that is briefly discussed in the following. As a major improvement to the basic state-of-the-art methods I present an improved method for efficient, zero-overhead event notification in Section 3.5.

Send Notification As depicted in Figure 3.18b there are three ways of signaling the completion of a completed message. When a sender buffer is used, the software may not be notified at all, but instead *implicitly* note that a buffer element was removed, i.e., the buffer can hold a new element. This can be queried in a form of so called *polling*: The software repeatedly reads the state until it changes, possibly with getting to sleep in between checks. In the absence of a sender buffer the software can instead poll the slot for completion of the transfer. Finally, the NAMP may be configured to raise a hardware *interrupt* when any of the slots is complete so that the software can handle the completions.

	Description
0	Flags (see Table 3.2b)
1	Remote endpoint
2	Depends on mode
3	Depends on mode

(a) Parameters

Name	Description
CoM	Credit update (1) or Message (0)
CON	Connection-oriented/-less
BUF	Buffered/Unbuffered
CCM	Continuous Communication Mode
`IEN`	Raise interrupt on completion
`VALID`	Transfer valid
`DONE`	Transfer completed (read only)

(b) Flags

Mode	`CoM/CON/BUF/CCM`	Parameter 2 & 3
Credit	1/0/0/0	2: credit amount, 3: –
Unbuffered	0/`CON`/0/0	2: data pointer, 3: size
Buffered	0/`CON`/1/0	2: local buffer, 3: offset
Continuous Communication Mode	0/`CON`/1/1	2: local buffer, 3: –

(c) Transfer Modes

Table 3.2: Slot configuration interface

Receive Notification On the receive side there again is the obvious choice to signal a new message by the change of the buffer fill level. As there is generally no state stored in the receiver side, the software needs to track all expected messages in an *expect queue*. An interrupt is now asserted on a new message arrival and the software checks the expect queue. To avoid race conditions between the interrupt signal, its reset by the software and new incoming messages, the NAMP counts up a register whenever it receives a new message and the software decreases it with every match. A positive counter value raises the interrupt level.

Finally, on both sides, the interrupts can be configured to be only risen when a flag is set in the endpoint (dashed example on the receive side in Figure 3.18b). Those notification options are useful for all use cases, but also serve as a motivation for the improvements of event signaling in general proposed in Section 3.5 due to the overhead they impose.

3.2.5.4 Slot Configuration Interface

The slot configuration interface is the memory-mapped interface for the software to setup message transfers and credit updates. Table 3.2 shows the interface as seen from the software that is used to configure the NAMP transfers. The configuration parameters (see Table 3.2a) map to configuration registers and there use varies with type of transfer. The transfer mode is defined by the flags outlined in Table 3.2b while only a limited number of modes are possible spanned by the introduced feature set. They are summarized in table 3.2c. Credit messages are a special case and are triggered in the receive logic of the software. All other transfer modes relate to the transferal of data to a remote endpoint as described above. Each of those modes can be used for connection-less or connection-oriented messages.

Finally, there is a flag that enables an interrupt once a transfer completes. Beside that the life cycle of a transfer can be controlled and monitored via the `VALID` and `DONE` flag. The former triggers a start of a transfer. The other parameters must be configured before. After NAMP signaled completion with the `DONE` flag, the software can finalize the transfer by resetting `VALID`. There is only one specialty with the policies to write the `CCM` flag: While in CCM, the software can set `CCM=0` to leave CCM. The NAMP then completes the current CCM transfer.

3.2.5.5 Message Transfer States and Progress Engine

From the described behavior I can now derive the transfer states and actions of the progress engine. The basic transfer states for connection-less and connection-oriented messages have been introduced in Figure 3.16 along the basic functional partitioning in Figure 3.17.

First, it is important to derive the states of an individual transfer. To maximize performance the NAMP obviously serves multiple transfers in parallel, because there are wait states in the progress of an individual transfer. Hence, the transfer state has to be encoded per NAMP slot. Finally, the different transfer modes have overlapping, but slightly different states and state transitions.

Figure 3.19 shows the flow diagram of a single transfer from the software creating the transfer until completion. There are certain persistent states that are drawn as dark boxes. Those are the points in the state diagram where the transfer is stalled for an event from remote or by the local software. The NAMP can serve another slot while the transfer is in this state. Hence this transfers's state is encoded in some extra flags per slot. Beyond that some extra registers are needed to store persistent look up information.

If the transfer is a credit update, then the NAMP just sends out the credit and the transfer is finished. For messages there are multiple paths the transfer can traverse the state diagram depending on the type of transfer. If it is a CCM transfer it checks for a message in the local endpoint buffer. If no message is ready for transfer, it waits until

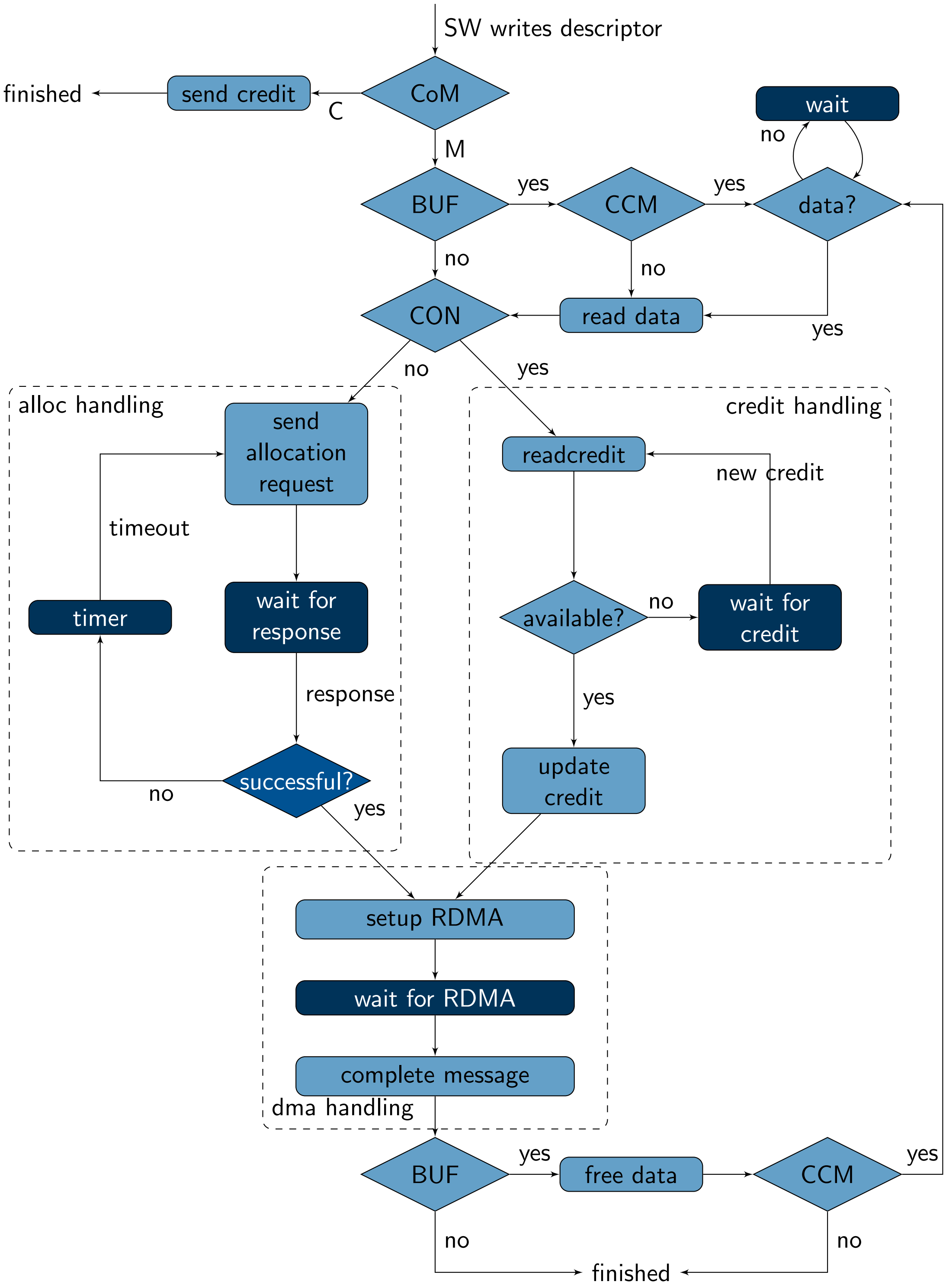


Figure 3.19: State diagram of the protocol handling with respect to one message lifecycle

it becomes available. Once data is ready it takes the same flow as the other transfers. The actual transfer is split into two phases:

The *first phase* is the flow control. In case of connection-less communication the NAMP tries to arbitrate a remote buffer space before continuing. It has to wait for the response, where the NAMP can then serve other slots. In case the buffer allocation was successful it continues, otherwise set a timer and enter a persistent state again until the timer expired. In case of a connection-oriented transfer the NAMP checks if sufficient credit is available and proceed to the second phase if that is the case. Before that it updates the credit atomically. In case there is no credit available the transfer is stalled until credit becomes available and another slot can be served.

The *second phase* is the actual data transfer. The NAMP configures the RDMA to perform this transfer. After the RDMA is configured the message progress is stalled until this RDMA transfer completes. Once it completes the message transfer is complete. In case of CCM it gets back to transfer the next transfer, otherwise the transfer is finished.

The actions and conditions of the progress engine are defined with the flow diagram (Figure 3.19) and the persistent states. As mentioned, the progress engine continuously serves NAMP slots that have met a condition to leave a persistent state and thereby advances the message transfer. The state machine of a NAMP progress engine can thus be derived as an arbitration policy that selects the next slot to be served and then performing the action for this slot according to Figure 3.19. The following arbitration policies are proposed:

Least Recently Served (LRS) The NAMP slot that has not been served for the longest duration is selected.

Quality-of-Service Priorization (QoS) The progress engine selects a NAMP slot based on a priority class.

Most Advanced Priorization (MAP) The progress engine selects the NAMP slot that is most advanced, because it advances toward completion and not generate new load on the Network-on-Chip, the NAMP subsystem and the system overall.

A NAMP implementation can provide a mix of those criteria and a certain degree of runtime configurability. In Chapter 4 I evaluate the design points for different arbitration policies.

3.2.5.6 Protocol Message Details

The NAMP concept may seem complex at this point. But although there is a large number of message transfer modes defined, they only break down to a different handling in the local tile and only a limited number of protocol messages are needed between the tiles. The bulk data transfer is configured as a RDMA transfer, so that only signaling messages for flow control remain. The three protocol messages are defined in the following.

Buffer Allocation This is the flow control message for connection-less communication.

Local action:	Lookup remote tile (destination) and remote endpoint address
Request message:	Remote endpoint address
Remote action:	Check if endpoint buffer has space and atomically lock if available
Response message:	NACK on failure, id and data address of buffer space otherwise

Credit update This is the flow control message for connection-oriented communication. It is sent from the remote tile to the sender tile.

Local action:	Lookup connected remote tile (destination) and remote endpoint address
Request message:	Remote endpoint address, credit to add
Remote action:	Atomically update credit
Response message:	–

Message Finalization The bulk data transfer is performed using RDMA, but the completion needs some extra signaling. This protocol message updates the reserved buffer space and marks the transferred data as ready.

Local action:	Lookup remote tile (destination) and remote endpoint address
Request message:	Remote endpoint address, buffer space id
Remote action:	Mark buffer space as completed
Response message:	–

3.2.6 Summary

In this section I have introduced the basic NAMP concept. The evaluation has shown great potential of this concept and I have therefore derived a set of properties and features that makes NAMP a valuable concept for a broad range of applications. Based on the derived functionalities, the prototyping in Chapter 4 yields actual data points with respect to area utilization and performance improvements.

The base message passing capabilities of the NAMP presented in this chapter contribute a significant improvement in the reduction of the message passing protocol overhead in software. The message passing protocol layer of the stack in Figure 1.3 has been partly offloaded into the hardware. In the following the NAMP concepts for improvement of the other criticial interfaces and actions are presented and discussed.

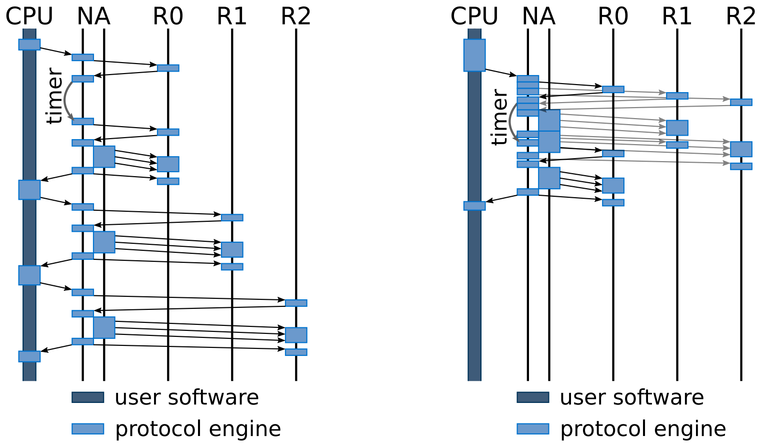
3.3 NAMP-CC: Collective Communication Acceleration

The basic NAMP concept focuses on simple one-to-one communication. It allows to use endpoints in one-to-many or many-to-one relations, but only on the granularity of communication relations. The concept of collective communication is well-adopted in high performance computing and other areas and instead works on the granularity of individual messages.

Basically, collective communication is about efficient work-sharing. The operations are used to compose algorithms with distributed workers. As introduced in Section 2.5.1 four operations are generally part of collective communication: i) broadcast to distribute a data set to multiple recipients, ii) scatter to split a data set and distribute the subsets to recipients, iii) gather to combine multiple data subsets from different senders to a large data set at a single recipient, iv) reduction to merge equally sized data sets from multiple senders at one recipient by applying an arithmetic operation to the data sets.

There are basically three requirements to support collective communication in the NAMP concept:

- A basic platform capability to enable collective communication is a scatter/gather-capable DMA controller. In our case the RDMA is required to support this feature whose operation is detailed below.
- The message passing data structures and API need extensions to support collective communication.
- The NAMP progress engine needs to support collective communication.



(a) Software handling

(b) NAMP-CC feature

Figure 3.20: Comparison of SW-based multicast handling and HW-assisted multicast handling

In the following I first evaluate the impact of those operations when using the basic NAMP approach and handle the operations in software. After that the required extensions to the buffer and message data structures are presented in the form of attributes for those data structures. Attaching those attributes to buffers and messages lowers the impact on extra hardware registers while adding look up operations in hardware, which is discussed after that.

3.3.1 Evaluation of Multi-Message Communication

The collective communication operations can be implemented in software using the NAMP. In the following the parameters and message sequence charts are derived and evaluated using the evaluation model. The overhead of the collective communications is not only message handling in software, but also the number of buffers and data copying operations are critical.

3.3.1.1 Broadcast and Multicast

In the context of this thesis broadcast operations are performed on a logical level between endpoints of an application. This is different from what a reader may know as a network broadcast. Instead it is the delivery of bulk data to multiple recipients that are spread over the system-on-chip. So, on the one hand the broadcast is physically a multicast operation and on the other hand the approach can then address logical multicast directly.

Basically, a multicast message can be characterized as the bulk data transfer as before and a list of receivers instead of a single destination. The naive approach to implement is to implement the multicast in software. Figure 3.20a depicts how the software configures a transfer for each of the destinations individually. Both important metrics can be addressed by offloading the multicast handling to the NAMP hardware as sketched in Figure 3.20b:

- The *overhead* o_s is reduced as the software is not involved with the continuous setup of the transfers. There is a slight overhead introduced with the setup of the multicast list on the configuration, but it is supposed to be over-compensated by the wake-up of the software to setup the next thread.

- Figure 3.20 drastically depicts the other huge impact on the *latency* L. By concurrently handling the progress of multiple message transfers, the extra latency for control flow delays does not block the other transfers. Instead the hardware can handle other transfers in parallel.

Figure 3.21 depicts the evaluation comparison of the SW-based multicast handling and the HW-assisted multicast that handles the operation. As described before, there are two fundamental effects that make the HW-assisted multicast efficient. One effect is the offloading of the iteration of the multicast destinations. As Figure 3.21 shows this overhead rises linearly when the multicast is implemented as a software function, while there is only a single setup phase for HW-assisted multicasts, independent from the number of destinations. The second effect was introduced before and stems from the fact that a software implementation configures each transfer one after the other. The *gap* is the time between two consecutive transfers. In this case (see the message sequence chart) this interleaves with the individual send operations. The gap is defined by the start and end of the transfer handling by the NAMP.

Figure 3.21 plots the gap for a variation of number of destinations. In comparison, the hardware offload is shown in comparison. It becomes obvious that the ability to post multiple concurrent operations in parallel can help reducing the gap. Anyhow, with increasing packet size beyond roughly 256 bytes this effect gets dominated by the actual data transfer again. The difference becomes more apparent when the messages can on average not be delivered immediately. A waiting process with poisson distribution and $\lambda = 1$ does not only increase the average gap as depicted in Figure 3.21. Instead it also introduces a high variation in the gap as depicted by the error bars. Again, Figure 3.21 shows the average gap is equally low as before for a hardware-based multicast. But

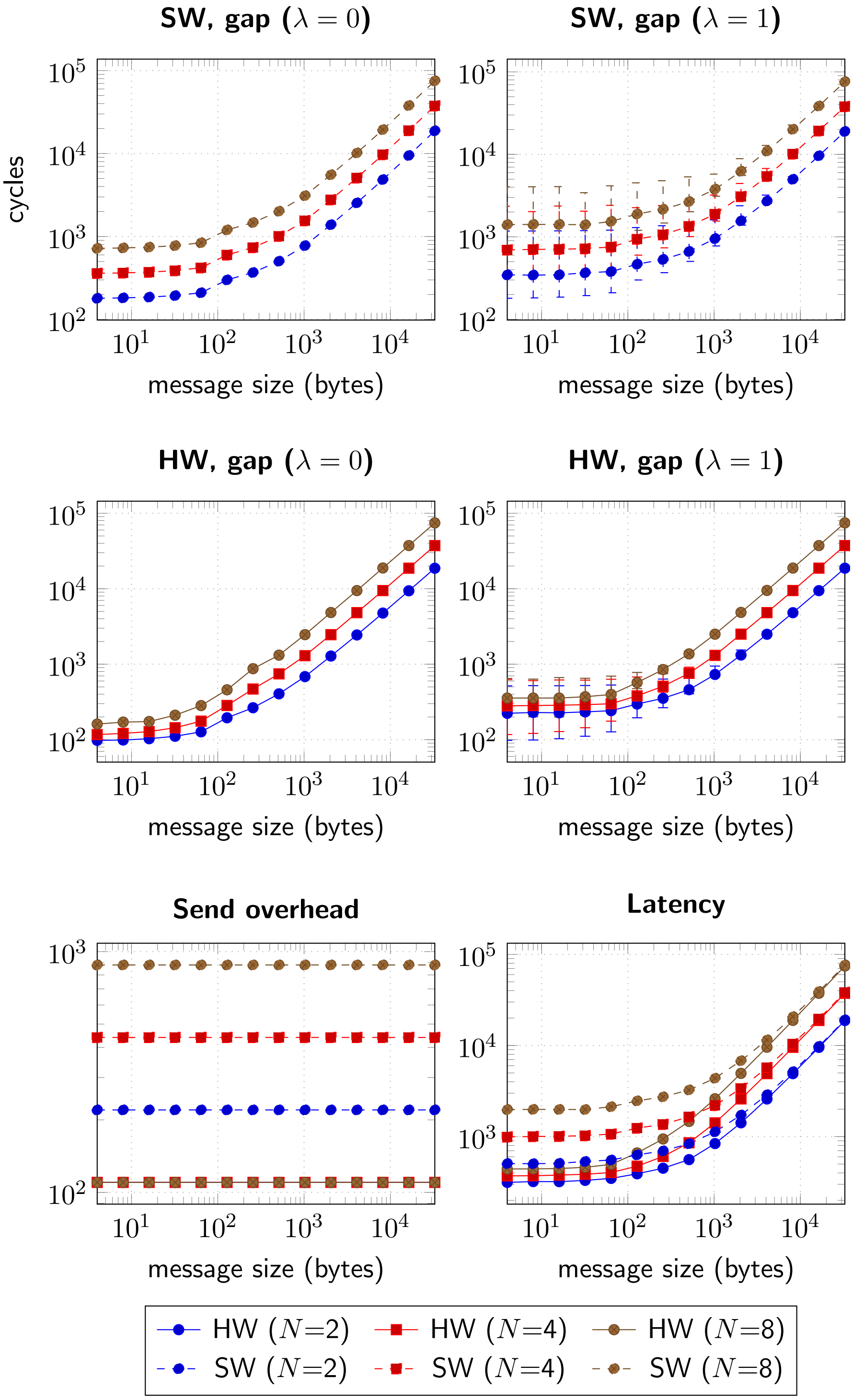


Figure 3.21: Evaluation comparison of a SW-based and HW-assisted multicast handling with different number of targets N.

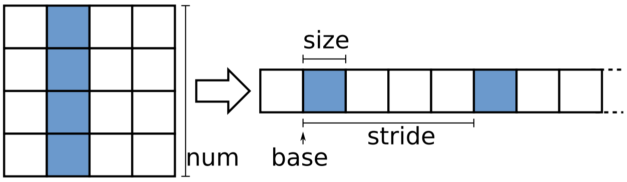


Figure 3.22: Mapping of a matrix to memory and scather/gather parameters

additionally, the variation is much smaller due to the fact that delays are hidden by other transfers. Finally, the effects vanish for large message sizes as they are dominated by the data transfer again. Hardware-assisted multicasts can therefore be a significant improvement.

3.3.1.2 Scatter

The scatter operation is to some extent comparable to a multicast, but instead of distributing the same data to a number of destinations it splits the data up and sends subsets to the individual destinations. Figure 3.22 visualizes the approach at the example of a two-dimensional matrix where vectors are distributed to different recipients. The given example also visualizes the four parameters:

- `base` is the initial offset in the data,
- `size` is the size of the single data elements to be scattered,
- `num` is the number of those elements to send, and
- `stride` is the offset from one element to the next

The receiver receives the data as a contiguous data space. So, in the given example the data set is $A[16]$ and the vector a_1 should be sent to recipient one. `num=4`, `stride=4`, `size=1` are constants to all transfers and the recipient 1 has `base=1`. Hence, the recipient receives the vector $a_1[4] = \{A[1], A[5], A[9], A[13]\}$. `num` and `stride` are generally constant for a scatter operation, while `base` always varies between the recipients and `size` can vary. Sending the horizontal vector works similar: `num=1`, `size=4` and `base=0,4,8,12` sends chunks of the data, while `stripe` is always irrelevant if `num=1`.

The comparison of the SW-based scatter handling and the HW-assisted scatter handling in the NAMP is comparable to the multicast handling (Figure 3.20) with the difference that the data transfer operations are smaller. Figure 3.23 shows the evaluation results for scatter handling. It can again be seen that the gap follows a similar

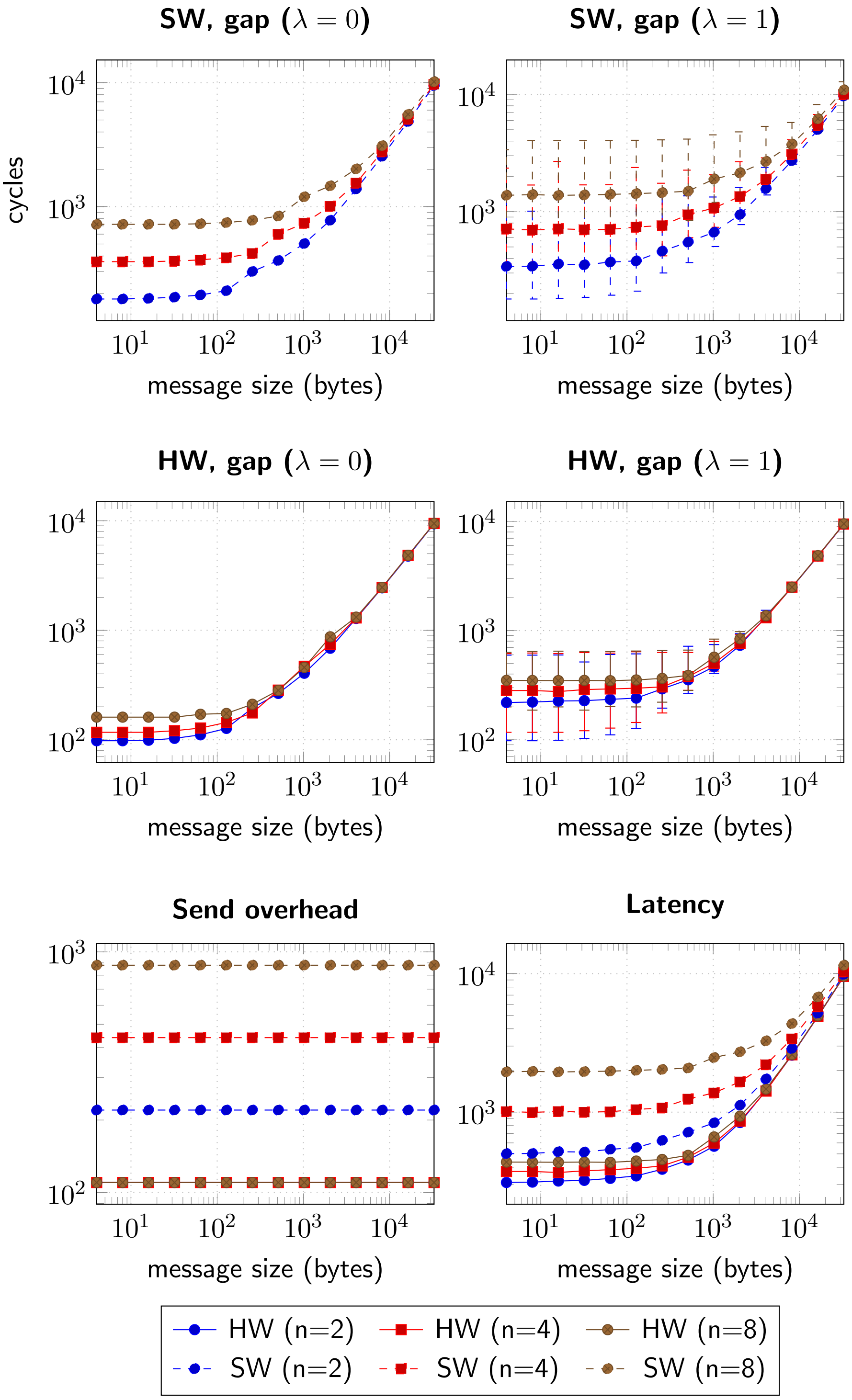
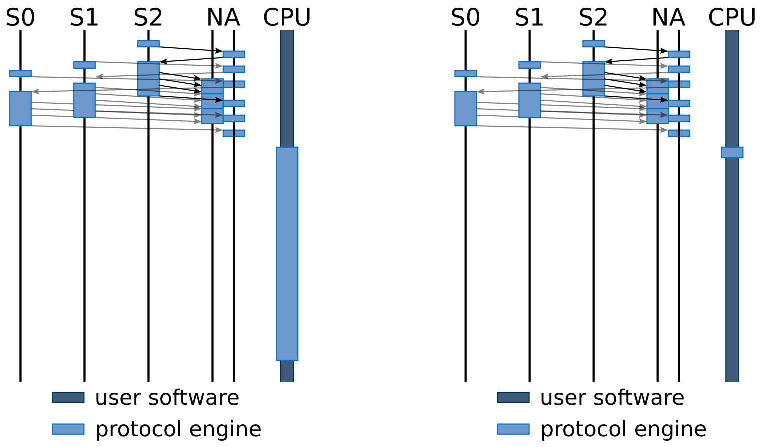


Figure 3.23: Evaluation comparison of a SW-based and HW-assisted scatter handling.



(a) Software handling

(b) NAMP-CC feature

Figure 3.24: Comparison of SW-based gather handling and HW-assisted gather handling

trend for increasing packet sizes. But especially for smaller packet sizes it is significantly smaller and more robust against waiting times during allocation ($\lambda = 1$). The send overhead remains static independent from the packet size. But for hardware-assisted scatter it is also independent from the number of destinations.

3.3.1.3 Gather

The gather operation is the inverse operation to the scatter operation: Multiple senders have their linear data set and at the destination this is gathered into one data set. The straightforward solution is to have multiple endpoints at the receiver and perform the operation as usual. Figure 3.24a sketches the data transfer by three senders. Implementing the gather operation in software requires to copy the data from the individual endpoint buffers and copy them into the expected data set. This operation follows the same semantics as the previously scatter operation, just in the opposite direction. This operation can introduce a large overhead when entirely done in software. Utilizing a local scatter/gather-enabled DMA controller can help here. Anyhow, there is a significant impact on latency and overhead.

The NAMP-CC feature for the gather operation allows the multiple senders to all transfer their data into one endpoint. The gather operation (according to a configured `base`, `size`, `num` and `stride`) is performed directly with this operation. When the last data item has arrived, the transfer is complete and the software receives it as if it was

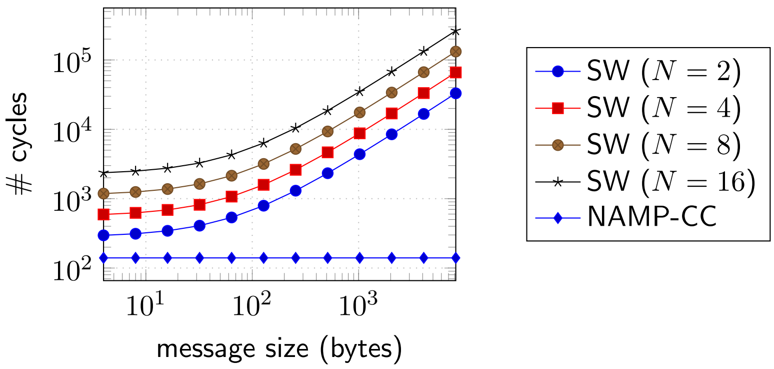


Figure 3.25: Comparison of receive overhead for gather operations

coming from one sender. Figure 3.24b visualizes this significant improvement in overhead and latency.

Figure 3.25 plots the receive overhead in clock cycles for gather operations depending on the number of sending endpoints (N). The impact of the software moving of the data from one endpoint to the final endpoint can be seen, with some static contribution by the interrupt handling and endpoint operations. The receiver gap is identical between the software-based gather handling and NAMP-CC. The sender side is not significantly influenced as the configuration of the DMA is just extended with one read from the local memory.

Finally, it has to be highlighted that the memory footprint of the operation is significantly higher with software-based handling because there needs to be N extra buffers. Each of the buffers is of $1/N$ size relative to the final destination buffer. The software-based hence has double the memory footprint in buffers, plus extra bytes for the data structures.

3.3.1.4 Reduction

Finally, the reduction can be seen as a mix of the gather and an inverse multicast operation: Multiple senders send data sets that are merged to one data set of equal size at the receiver. For merging the data, an operation is defined. Such an operation is often an addition, multiplication, averaging or other arithmetic operations. The message sequence charts are comparable to the ones for the gather operation (see Figure 3.24). The data transfer operations are longer apparently, and with software-based handling the reduction can again take significantly longer.

An offload to the NAMP operation is again desirable, but gets much more complex. The other collective communication operations are about calculating addresses to store

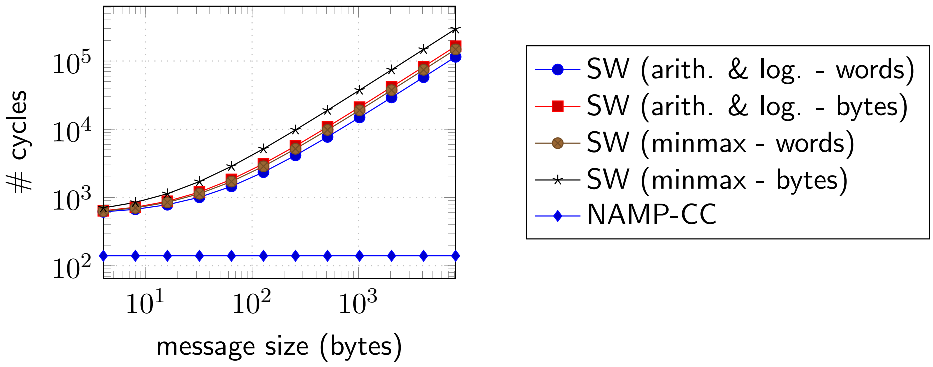


Figure 3.26: Comparison of receive overhead for reduction operations, $N = 4$

data items properly. Reduction instead needs to apply an arithmetic or logical operation to each data item in the data path of the NAMP, which is the DMA. The DMA controller hence needs to enhanced with the capability to: i) Read the current data item in the buffer, ii) apply an operation to the data item and the received value, and then iii) write it back to the buffer.

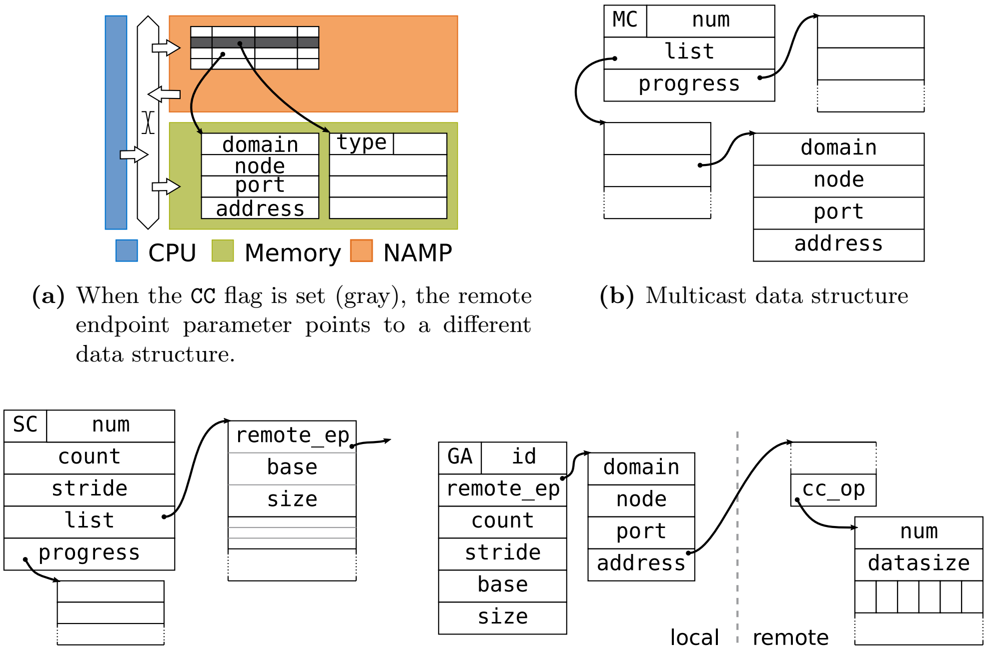
Figure 3.26 plots the evaluation results for the reduction operation for $N = 4$. Its trend regarding the number of senders (N) is generally similar to the gather operation. Another aspect of variation here is the actual reduction operation and the data type it is applied on. Arithmetic and logical operations are by default supported by most processor cores, as also here. But when no support for packed vector operations (Single Instruction Multiple Data, SIMD) is present, there are extra operations required for operations smaller than the word size (here: byte). Finally, minimum and maximum operations are supported as presented in the following. Here, the operation itself is not generally provided by the hardware and hence it adds to the receive overhead.

The receive overhead for the NAMP is always constant, because it only involves notifying the receiver. Also the gap is ideally constant as long as the operations can be executed in one clock cycle in hardware.

3.3.2 Transfer Setup and Data Structures

I have previously introduced the NAMP concept as a lean, yet efficient method for on-chip message passing. Introducing other concepts into the NAMP should not impact this basic goal. Hence the collective communication feature should integrate smoothly into the NAMP concept and not introduce any significant overhead to the basic functionality.

There are three parts that are proposed to support collective communication: The first part is to reduce the influence on the basic interface that was presented in Table 3.2. As



(a) When the `CC` flag is set (gray), the remote endpoint parameter points to a different data structure.

(b) Multicast data structure

(c) Scatter

(d) Gather

Figure 3.27: NAMP interface integration and data structures for collective communication.

sketched in Figure 3.27a the proposed change to the interface is minimal: An additional flag (`CC`) signals that the transfer is a collective communication transfer. The semantics of the “remote endpoint” parameter changes then. It does not point to the data structure of the remote endpoint as defined in Figure 3.3. The second proposed part are data structures specific to collective communication that it points to instead. The data structure varies between the operations and are discussed in the following. Based on those extensions, the modifications to NAMP hardware are then presented after that. Finally, the third part are properties that are attached to each buffer slot. Those are used at the remote side of an operation to track the progress.

Both the collective communication data structure and the buffer extensions are created and configure at connection setup. In the following I discuss the data structures for the different collective communication operations. This then leads to the integration with the basic NAMP hardware in Section 3.3.3.

3.3.2.1 Multicast

The multicast operation is the only collective communication operation that is available for connection-less and connection-oriented message transfers. Figure 3.27b depicts the data structure used for multicasts. As described, it is the one that the entry in Figure 3.27a. It essentially points to two lists, each with length `num` which is the number of multicast destinations. One of the lists contains the pointers to the individual remote endpoints to send the data two. This is the missing data for the interface, but beyond that the progress engine needs a second list: To allow for an arbitrary number of destinations the individual progress for each remote endpoint and to allow for the aforementioned parallelism, the NAMP needs to store the progress state for each of the destinations. I get into more details about that below.

3.3.2.2 Scatter

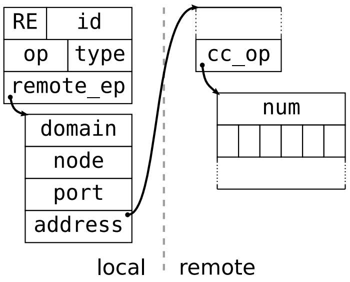
Both the scatter operation and the gather operation are only allowed on connection-oriented endpoints. This is because the endpoints are part of a complex setup and the operations must not mix with other arbitrary messages.

The data structure for the scatter operation (Figure 3.27c) is similar to multicasting. It adds the two constant parameters `count` and `stride` to the basic data structure and the list of recipients does not only contain pointers to the remote endpoints but also the `base` and `size` of the remote endpoints data set. The NAMP progress engine uses this data to configure the RDMA scatter operation. Finally, there is a progress list that is used by the NAMP to track the individual progress for each recipient.

3.3.2.3 Gather

For the gather operation the data structure is an indirection to the single remote endpoint, but with the gather parameters associated to it (see Figure 3.27d). This is used by the NAMP to setup the RDMA gather.

Beyond that at connection setup each buffer slot at the receiving endpoint needs an extension to track the progress of a gather operation, as sketched in Figure 3.27d. This is required to track the completeness of a gathered message. For that a bit field per buffer space is proposed, so that the messages can be tracked by the NAMP on i) the allocation operation, and ii) the finalization operation. This is discussed in more details below.



(a) Endpoint data structure

ADD	Arithmetic addition
BOR, BAND, BXOR	Bitwise or/and/xor
LOR, LAND, LXOR	Logical or/and/xor
MAX, MIN	Maximum/minimum element
U8, U16, U32	Unsigned integers
S8, S16, S32	Signed integers

(b) Proposed operations and types

Figure 3.28: Extensions to the endpoint data structures and proposed operations for the reduction operation

3.3.2.4 Reduction

Similar to the gather operation, the endpoint data structure for the reduction operation is a redirection to the remote endpoint data structure with data attached to it (see Figure 3.28a). This extra data is again an identifier and then the reduction op and the basic data type. Both the latter parameters are used to configure the DMA transfer. As mentioned, the DMA controller needs extra functionality in its data path to perform the reduction operations on-the-fly. Two parameters are therefore transferred with each DMA transfer: The operation (`op`) defines the operation to execute on each data item, and the `type` sets the basic unit inside the data stream.

Table 3.28b lists the proposed operations. They are adopted from the MPI standard with the exception of multiplication due to the hardware overhead and the supported types. The types can also be found in the table and are typical integer types. Supporting multiplication for integer numbers fast leads to problems with the precision, hence multiplication is explicitly excluded. The remaining operations and data types replicate a reasonable set of typical operations found in signal processing, video processing or data mining applications.

3.3.3 NAMP Hardware Integration

The previous evaluation has shown that collective communication support in the NAMP is advantageous. Some basic assumptions about the hardware implementation were made there, that are further elaborated in the following. The data structures and extensions discussed before have to be properly considered by the NAMP to achieve the projected benefits.

The hardware integration is twofold: As shortly introduced before, the scatter and gather operation have to be supported by the DMA engine. Beside that the progress engine needs to process the provided data structures to set up the DMA engine and handle multiple parallel transfer for one NAMP slot. In the following, the hardware integration is presented on a conceptual level, while again the prototype implementation is discussed in Chapter 4.

3.3.3.1 DMA Extensions

As mentioned before, a basic functional extension to the DMA controller is required to effectively support collective communications. The DMA transfer has to support the scatter and gather operations. Those allow to either scatter data from the memory region to a smaller contiguous memory region or to gather from a contiguous memory data region into a scattered region. This capability most basically requires an adoption of the address generation logic. Beside that the headers of the packets between the DMA controllers need to be adopted to carry the necessary information. Those extensions are briefly elaborated in the following in accordance to the interface extension described in Figure 3.27.

Scatter For the scatter operation the message to the remote does not require changes. The operation is focused on the local operation which is reflected in the data structure described in Figure 3.27c: The DMA is for each destination configured to read with `base`, `size`, `stride` and `count`. The local address generation is initialized to point to the proper base as normal transactions. The total transfer size is configured to `size*count` and the two extra parameters `size` and `stride` are provided to step through the memory.

Gather The situation is the opposite for the gather operation. The baseline for comparison is to generate one DMA transfer for each chunk. Apparently, this would create a large number of transfers for small chunks which leads to an increased number of flits and thus traffic. In a gather DMA version the local address generator is not modified, but both the packets sent to remote and the remote address generator need modification as sketched in Figure 3.27d. This is only useful as long as at least two chunks fit into one transfer. Note: I only consider *stateless* DMA transfers here. Keeping state in the receiver allows for a better packetization of the network packets, but adds a huge amount of complexity and hardware overhead.

Both approaches are compared in Figure 3.29. It plots the required number of flits for both approaches against the size of chunks in a gather operation for a transfer size

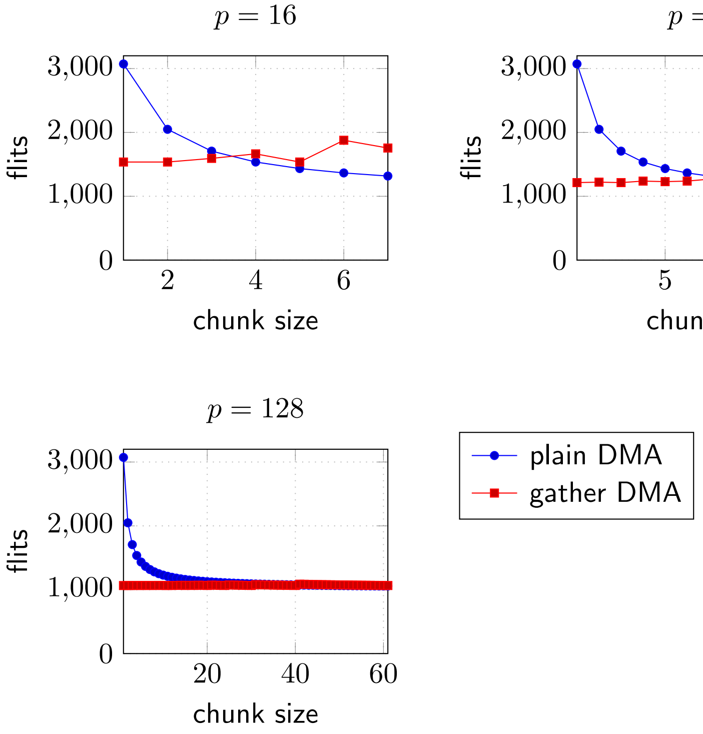


Figure 3.29: Total number of flits for a gather operation with plain DMA and scatter/gather-extended DMA

of $S = 4kB$. I only consider those transfers where the chunk size s is smaller than half the packet size ($s < \frac{p-h}{2}$). For plain DMA transfers the number of packets per chunk is $\lceil \frac{s}{p-2} \rceil$ and the number of packets is roughly $N_p = \left\lceil \frac{S}{s} \right\rceil$. The total number of flits can thus be derived as:

$$N = \left\lceil \frac{s}{p-h} \right\rceil \cdot h \cdot \left\lceil \frac{S}{s} \right\rceil + S$$

Here, a plain DMA header size of 2 is assumed. For a gather DMA operation multiple chunks are packed into one transfer packet. The packet header gets larger (here a header size of 5 is assumed). The number of chunks per packet is then $c = \lfloor \frac{p-5}{s} \rfloor$. Total number of chunks is $N_c = \frac{S}{s}$ and the number of packets $N_p = \frac{N_c}{c}$.

$$N = N_p(h + cs) = \frac{N_c}{c}(h + cs) = \frac{S}{s}\left\lfloor \frac{p-h}{s} \right\rfloor \left(h + \left\lfloor \frac{p-h}{s} \right\rfloor s\right)$$

Three plots for different, usual network-on-chip packet sizes (which limit the DMA transfer size) are plotted. The comparison only considers chunk sizes that fit at least two chunks into a transfer. For chunk sizes beyond that point, plain DMA transfers are always better. From the plots it can be derived that a special gather DMA reduces the number of packets significantly for small chunk sizes. Beyond a certain point plain DMA transfer are preferable. This motivates adding support for gather DMA transfers, potentially with dynamic selection to use either transfer type depending on the chunk size.

3.3.3.2 Integration with NAMP Progress Engine

During the introduction of the NAMP concept the state diagram for individual transfer was sketched in Figure 3.19. This is not the implementation-specific NAMP state machine, but the progress of an individual slot. As described before, a NAMP checks the slots based on certain criteria and tries to progress them while other slots are waiting on events.

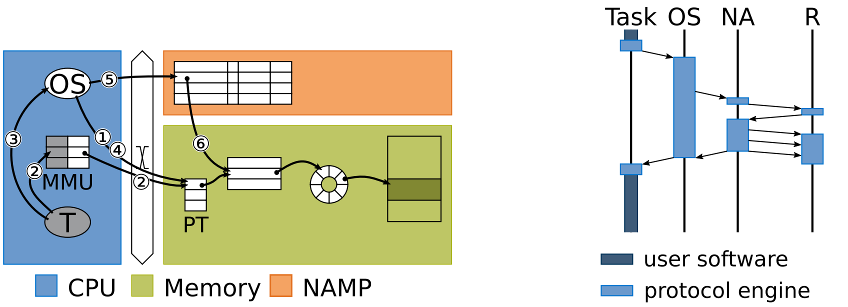
Generally, the NAMP-CC feature does not fundamentally change this, but add an extra dimension. When the `CC` flag is set, the parts *credit handling*, *alloc handling* and *DMA handling* are executed for different transfers inside a slot. This follows the interface extensions as described in Figure 3.27. The NAMP implementation thus needs an extra level of progress handling where it scans the lists for progress.

3.3.4 Summary

In this section I have presented the concept of NAMP-CC which is a modular feature of the NAMP. NAMP-CC is designed so that the hardware overhead is minimal and the basic operation is not influenced negatively when NAMP-CC is not used. The evaluations have shown a huge potential improvement for the four types of collective communication. In Chapter 4 the actual implementation is presented to verify the assumptions about hardware usage and evaluation calibration.

3.4 NAMP-SV: Self-Virtualized Network Adapter

So far the NAMP concept was discussed for software running directly on the processor, often referred to as *baremetal* software. There is a common misconception that baremetal software is simple code, but in fact baremetal software can be multi-threaded and arbitrarily complex. Contrary to the plain baremetal mode, an operating system in most cases adds the concept of resource sharing and strong separation. The idea is that tasks running on top of the operating system don't know about other tasks or interfere



(a) Overview **(b)** Message Sequence Chart

Figure 3.30: NAMP Virtualization by Software Emulation

with them. Instead each task has the view of running on the hardware exclusively. This is enabled with the concept of memory virtualization. Here, a processor provides a memory management unit (MMU) which transparently maps *virtual addresses* to physical addresses.

A problem arises when the tasks want to access a shared device. One simple approach is to assign the device exclusively to a task, but this is limited to such devices that themselves cannot access memory. If the latter is the case the strong separation property is violated. Beyond that, devices as network interfaces and the NAMP are desired to be shared between tasks. The most straightforward approach is device emulation in software, where the operating system performs the device access on behalf of the tasks. In the following this approach is analyzed with respect to the NAMP. Starting from this, the concept of self-virtualization of the NAMP is presented. A key element is the concept of virtual interfaces that is described for this NAMP-SV concept. This section closes with a description how this concept reflects in the message passing API.

3.4.1 Analysis of Software Emulation

In case the hardware does not provide any means to support virtualization by itself, the standard approach to virtualization is to emulate the device access in software. There is a degree of freedom in where to implement the majority of the driver and how much of it can be re-used, a topic I abstract from for the sake of simplicity. On an abstract level the software emulation is responsible for multiplexing the access to the device.

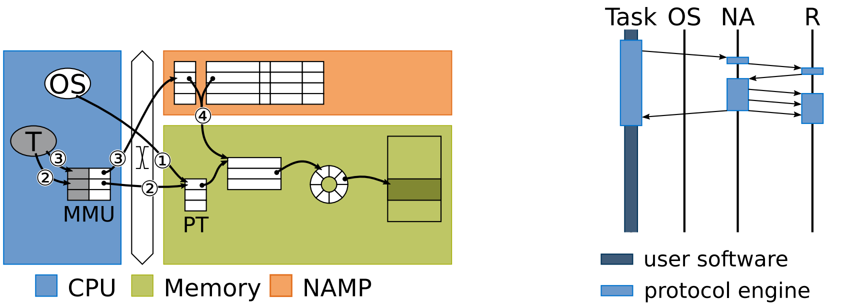
A generic overview of how software emulation may be implemented using the basic NAMP is sketched in Figure 3.30a. Virtual memory basically consists of a page table (PT) that is maintained by the operating system and maps virtual addresses to physical addresses. As mentioned before, the memory management unit (MMU) of the CPU is responsible for mapping memory accesses of the currently running task transparently and according to this page table.

Once a task T creates an endpoint, the operating system ensures that the virtual address the task operates on maps to the physical address of the endpoint in memory (①). This is a standard operation provided by an operating system. It does not make a difference if the operating system allocates the data structures and maps them to the task or if the task does the same on virtual addresses. When the task accesses the endpoints the MMU maps the addresses (②). Once the task wants to send a message, it triggers the operating system (③). The method for doing this are system calls. A system call transfers control from the task to the operating system kernel and contains parameters. The system call to send a message is equivalent to the API call. The operating system then performs the operations to configure the NAMP by first looking up the corresponding physical addresses to the virtual addresses that the task passed with the system call (④). It then configures the NAMP as described before (⑤) that then finally accesses the correct data via their physical address (⑥).

As mentioned before, this is one way of implementing this, but the overall flow is always similar. Figure 3.30b sketches the message sequence chart. The actual message transfer is identical to the base NAMP and the operating system (OS) configures the NAMP (NA) that handles message transfer to the receiver (R). The only difference is that the task has to transfer to the OS with a system call (the task and CPU are running on the same processor). There are two components that increase the overhead compared to the baseline NAMP. First, the translation adds extra operations inside the OS. Second, the system call from the task to the OS does not come for free: The system call is a context switch that comes with the cost of at least saving and restoring processor state. Especially for small messages this can become an important overhead. The performance of this approach is analyzed and compared to the NAMP-SV in the following.

3.4.2 NAMP Self-Virtualization

Device self-virtualization is the offload of the translation operations to the device, so that each task talks to the device as if it was the only user. One approach is to make the NAMP aware of the translation and perform the MMU operations in the NAMP too (so called IO-MMU). This has the advantage that in the end the task can run the unmodified driver can be run either in baremetal or as a task in an operating system. But it comes with the drawback of the complexity of memory management in the critical path of the NAMP state machine. Each memory access needs to be translated if the



(a) Overview **(b)** Message Sequence Chart

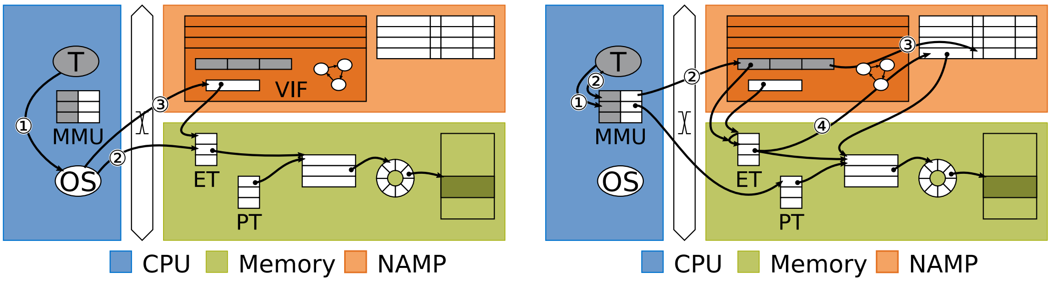
Figure 3.31: NAMP Self-Virtualization

slot is used by a task. The translations can be cached to a certain degree, but overall the concept of self-virtualized device as described in the following is much more elegant.

The concept of NAMP-SV starts from the other side of the operation: The API is the entry point for a message transfer. The design goal is to define a so called *virtual interface* in a way that:

- multiple tasks can use multiple virtual devices in a dynamic way,
- the operating system can efficiently map tasks to slots and configure their operation,
- the operating system is only involved during setup and destruction of those relationships, and
- the message passing API is efficiently mapped to the virtual device interface with minimal overhead.

Figure 3.31a depicts the NAMP-SV concept. Basically, the operating system allocates a slot for the task and maps it along with the data structures into the page table of the task (①). As before, the task can then access the endpoint and buffer data structures using the virtual memory subsystem (②). When it wants to send a message it can access the allocated slot via the virtual memory subsystem too (③). A simple virtual interface is used to configure the transfer and the NAMP then uses the configured data structure (④). The last two steps differ significantly from the default NAMP interface and the virtual interface is discussed subsequently. Figure 3.31b sketches the message sequence chart of NAMP-SV, where the task is able to bypass the OS during the send operation.



(a) Setup phase, once per task

(b) Send phase, bypass OS

	Description
`ET_BASE`	Base address of the endpoint table
`ET_BOUNDS`	Number of addressable endpoints (table size)

(c) Setup registers (accessible only by OS)

	Description
`FLAGS`	see Table 3.2b, no `BUF`)
`REMOTE`	Remote endpoint (vID)
`LOCAL`	Credit/local endpoint (vID)
`IDX`	Buffer index

(d) Transfer registers (accessible by task)

Figure 3.32: NAMP-SV Virtual Interface

3.4.3 NAMP-SV Virtual Interfaces

The design goals for the NAMP-SV Virtual Interface (VIF) are as follows:

- It must be composable, meaning it should be easy to configure it for each slot statically and dynamically,
- it interfaces a standard NAMP slot as described above,
- on the bus side it must not be configured with any virtual addresses to spare any MMU operation, and
- apparently it must be able to send messages without any interference with the operating system.

Figure 3.32 depicts the role of the virtual interface, its interfaces and components. There are essentially two sets of configuration registers that are configured in two different phases: the *task configuration registers* that is configured by the kernel once per setup and the *task interface registers* that are set by the task to control the individual message transfer operations.

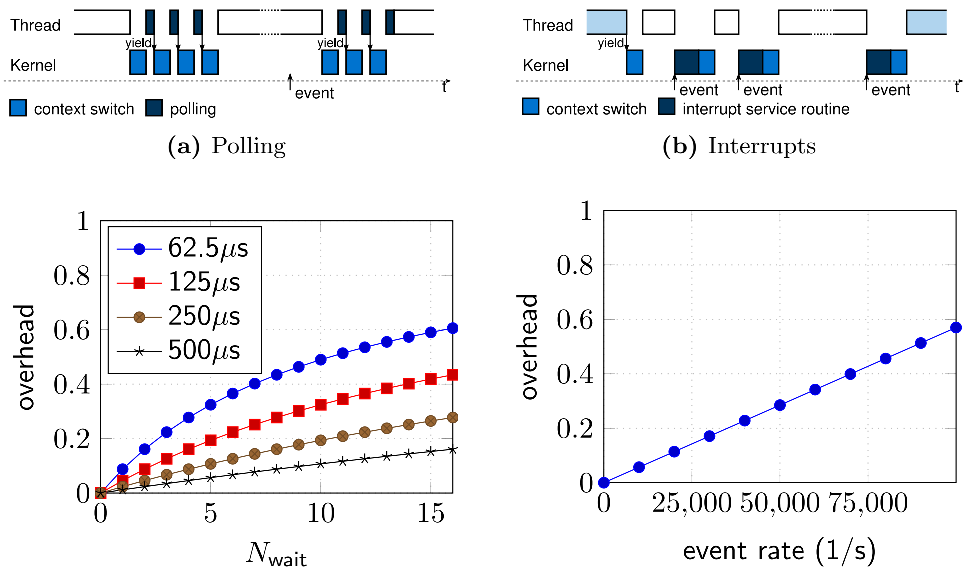
Setup Phase (Figure 3.32a) Once a task wants to use the NAMP-VIF it raises a system call with this request (①). The operating system maintains an extra data structure in physical memory, the *Endpoint Table (ET)*. As mentioned, whenever the task creates a local endpoint or retrieves a remote endpoint, it has to invoke the operating system. As a side effect of each of those operations the kernel maintains the ET by adding new endpoints to the ET (②). This assigns each endpoint a *virtual ID (vID)* that is used by the task in the following. During the setup phase it now allocates a virtual interface for the task. It then writes the base address of the ET into the task configuration register (③). Along with the base address it writes the bound for the index to guarantee the task does not address out of bounds. The two setup registers are summarized in Table 3.32c. Finally, the OS maps the virtual interface into the virtual memory space of the task.

Send Phase (Figure 3.32b) During the normal operation[2] the task does not communicate with the OS at all (OS bypass). Instead, it normally operates on the data set as described before (①). Once the message is ready to be transferred it accesses the virtual interface by its virtual address (②). The interface is described in Table 3.32d. It is roughly equal to the physical interface described in Table 3.2a. The essential difference is that the virtual interface addresses endpoints by their virtual identifier (vID) instead of their physical address. As a result of that the unbuffered transfer mode (`BUF=0`) is not supported for NAMP-SV. Once the `VALID` flag is set, the virtual interface is triggered and configures the physical interface. The `FLAGS` and potentially the credit are copied, while the vIDs are transparently translated to physical addresses (④). This involves a lookup into the ET. Finally the virtual interface triggers the NAMP to start the physical message transfer.

3.4.4 Summary

In this section I have presented the self-virtualization subsystem of the NAMP, which is essentially the virtual interface. This virtual interface can be mapped into the address space of a task by the operation system, and there can be multiple virtual interfaces per NAMP. The virtual interface transparently translates the virtual endpoint identifiers into the physical addresses. The task can neither interfere with the translation nor modify the physical addresses. This enables strong isolation of the communication between tasks. The NAMP-SV concept is entirely new for on-chip inter-task communication.

[2] Note: As throughout the entire thesis I here assume that there are more message transfers than initial setup operations.



(a) Polling

(b) Interrupts

(c) Polling Analysis

(d) Interrupt Analysis

3.5 HW-OSQM: Hardware-based Operating System Queue Manipulation

The notification of the software when a task is done was briefly introduced in Section 3.2.5.3. There I have described how a software can poll the NAMP (sending) or the endpoint (sending and receiving) to check for events. Alternatively, the standard approach of interrupting was introduced too. One of the major issues with interrupts is that they add overhead on the processing by the interrupt service routine.

In the following I briefly analyze the impact of both polling and interrupting on the task processing and present the so-called *Hardware-based Operating System Queue Manipulation* as a generic method. This method was originally presented in [219]. Finally, I briefly describe how the method is integrated with the NAMP concept.

3.5.1 Analysis

Both the classical methods add to the sender overhead o_s and receiver overhead o_r. For example polling requires to repeatedly check for the completion of a transfer or the interrupt service routing (ISR) for interrupts introduces a lot of extra cycles. Anyhow, the overhead depends on the application characteristics and parameters of the operating system. The basic building blocks of the overhead are the specific operations, which can be assumed to be constant for each implementation. A polling operation takes T_{poll}

cycles and the time for the interrupt service routine is T_{isr} respectively. Both add up with operating system specifics, such as the context switch duration T_{CS}. Furthermore the time slice length T_{slice} is a parameter of the kernel: The longer the time slices are the lower the impact of the context switches. But a longer time slice can increase the latency until an event becomes visible to a task. Beyond that an application characteristic is the average event rate r_{event}, which is the number of events generated events per second.

The impact of both methods on the processing overhead can be evaluated analytically, without the complex event-based simulation framework of the NAMP. I used the same setup to calibrate the basic building blocks as described before and derive a steady-state analysis for that. Figure 3.33a shows the impact of polling on the performance. Each waiting task has to be activated to performed its polling operation before it yields in case of no success. This takes away processing time from tasks that could actually run. The rate of events and the mean time between events (MTBE) determine the number of waiting tasks as $N_{\text{wait}} = r_{\text{event}} \cdot T_{\text{MTBE}}$. The overhead is then

$$o_{\text{polling}} = \frac{N_{\text{wait}} \cdot (T_{\text{poll}} + T_{\text{CS}})}{T_{\text{slice}} + N_{\text{wait}} \cdot (T_{\text{poll}} + T_{\text{CS}})}$$

Figure 3.33c plots the overhead depending on the numbers of average waiting threads N_{wait} and typical time slice lengths T_{slice}. Even long time slices lengths can have an overhead of 10% for 8 waiting threads, while two waiting threads already induce an overhead in that order at a shorter time slice length.

For interrupting only the event rate r_{event} is relevant as sketched in Figure 3.33b. Both the number of waiting tasks and the time slice length do not influence the overhead. The overhead is thus linear:

$$o_{\text{interrupting}} = r_{\text{event}} \cdot T_{\text{isr}}$$

Figure 3.33d plots the overhead. Event rates around $25kHz$ already lead to an overhead of 10%. So while novel message passing concepts such as NAMP allow to efficiently transfer small messages, the many events generated by those can have a significant impact on the overhead.

3.5.2 HW-OSQM Concept

The basic idea of HW-OSQM is to delete all overhead related to event signaling. Polling is generally not a good method to check the occurrence of an event, except in the case of very short delays between the start of the waiting and the event. When interrupts are used the operating system takes care of waking up a task waiting for the event. The basic scheduling unit of an operating system in the context of this thesis is generally a thread and not a task. Tasks can consist of multiple threads, for example a common pattern is to use a "communication thread" that handles the device interaction. So, in the following I refer to threads.

As mentioned before, the concept presented in the following is not focused on the NAMP or network adapters in general. Instead it applies to all device notification. The normal operation of such event signaling with interrupts serves as reference and can be described as:

1. The thread configures the device and sets an "interrupt enable" flag or similar. The device driver and/or operating system track the association between the thread and the device.

2. The thread is suspended, which means it cannot be scheduled any more until explicitly woken up.

3. Once the event occurs, the device raises an interrupt. The software interrupt handler is then triggered. It looks up the thread associated with the event.

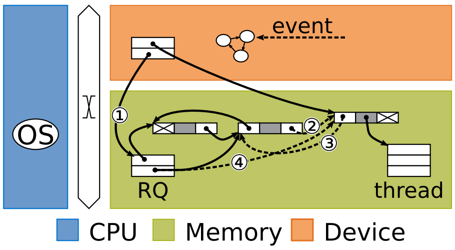
4. The thread is then resumed by the operating system which means it can be scheduled again.

5. The thread returns from the blocking call once it is scheduled again.

An operating system generally operates on a so called *ready queue (RQ)*. This is commonly a linked list and elements are put into the list at the end of a time slice or on thread resume. The scheduler then gets the next element from this queue and schedules the thread. In the following I briefly discuss the operation of appending a thread to the ready queue in two popular operating systems that are often found in systems relevant to this thesis.

Amazon FreeRTOS FreeRTOS is a real-time operating system that has recently been adopted by Amazon as IoT operating system for constrained devices [6]. It provides multiple priority levels and maintains one ready queue for each priority level. A thread is then scheduled by calling a list manipulation function[3]:

[3] see `https://github.com/aws/amazon-freertos/blob/6620031aed80bd411c87baeef501ac884cfd1b1a/lib/FreeRTOS/list.c#L74`



(a) Basic OSQM operation

	Description
QUEUE	Pointer to ready queue
THREAD	Pointer to thread queue element
TYPE	Single or Double Linked List
DIR	Insertion direction (front/back)
Q_LAYOUT	Layout of queue (offsets)
E_LAYOUT	Layout of queue elements

(b) Configuration flags and registers

Figure 3.33: Basic operation of the OSQM and Configuration

```
void vListInsertEnd( List_t * const, ListItem_t * const );
```

The function takes a pointer to the list and a pointer to a list item specific to the thread and appends the latter to the former.

seL4 Microkernel seL4 is a secure, formally verified microkernel operating system [204]. It also maintains multiple priority levels and a queue of type `tcb_queue`[4] for each priority. A thread is then scheduled with the `tcbSchedEnqueue`[5] function that adds the thread to the front of the queue (the scheduler picks from the end of the queue):

```
if (!queue.end) { /* Empty list */
  queue.end = tcb;
  addToBitmap(SMP_TERNARY(tcb->tcbAffinity, 0), dom, prio);
} else {
  queue.head->tcbSchedPrev = tcb;
}
tcb->tcbSchedPrev = NULL;
tcb->tcbSchedNext = queue.head;
queue.head = tcb;
```

Those two examples are chosen to show that a thread wake-up is a rather regular task that involves the adding of a thread-specific scheduling item to its ready queue. One important aspect is that those operations must be atomic operations to support multi-core operation.

The basic idea of HW-OSQM is now to asynchronously assist the operating system in hardware in adding threads to the ready queue. Figure 3.33a shows the basic operation

[4] see https://github.com/seL4/seL4/blob/ee28936d489fd8d37cf5f767fd380838aad8580a/include/object/tcb.h#L27

[5] see https://github.com/seL4/seL4/blob/8639dbcaea2afe2370e86db08ffc78afb94ebcab/src/object/tcb.c#L87

of HW-OSQM. The device is configured with the pointers to the ready queue and the thread element to add to the queue. This configuration is usually done per independently configured unit, such as NAMP transfer slots or DMA transfer descriptors etc. Once the event occurs, the HW-OSQM looks up the ready queue and from there the tail of the queue (①). It then links the current tail to the thread's queue element (②) and back from there (③). Finally, it updates the queue's tail (④).

3.5.2.1 Configuration & Flexibility

The two operating system examples showed that the way thread scheduling and resuming of suspended threads are handled is similar among a number of operating systems. Anyhow, there are a few variants of which kind of queue is used and how the data structures are organized. Figure 3.33b shows the fundamental configuration parameters of HW-OSQM. The first two items (`QUEUE` and `THREAD`) are used per OSQM event as sketched in Figure 3.33a. They are configured for each tracked event by the operating system and are in most cases tied to a device descriptor, slot etc.

The other four configuration items in Figure 3.33b are operating system-specific and configured once per OSQM instance: `TYPE` configures if the queues are single-linked lists or double-linked lists. `DIR` is the end where list elements are added, front or back. Finally, `Q_LAYOUT` and `E_LAYOUT` configure where the pointers are located in the data structures. `QUEUE` and `THREAD` point to data structures that are in general different among the operating systems and the layout configuration items set the offset of the required pointers in those data structures. `Q_LAYOUT` hence contains the offset of the head and tail pointers in the queue data structure and `E_LAYOUT` contains the offset in the thread list element data structure. Many operating systems place the latter pointers in the thread control block, such as seL4 in the example above.

Finally, the configuration items specific to operating system may be set statically if the platform is build for a specific operating system. But in the common case those are runtime-configurable options, that allow to use the OSQM functionality for a variety of operating systems and have more flexibility in the usage.

3.5.2.2 HW-OSQM Extra Features

The basic OSQM method described before allows for usage with a broad range of operating systems, far beyond the two operating systems used as examples before. Anyhow, the basic concept may fit a specific operating system or device conditions. Hence there are a few proposed extensions that I briefly discuss in the following.

Critical Events & Demand Interrupting The HW-OSQM concept is ideally to reduce any overhead related to event signaling. Anyhow, it has one drawback: The resuming of a thread occurs transparent to the operating system and there may be certain events that need to be handled *immediately*. This is not the average case and one could argue to simply use interrupting for such a device. But HW-OSQM can be extended to support critical events in multiple ways. For one, the device itself could just bypass HW-OSQM and raise an interrupt if certain conditions are met. Another option is that the HW-OSQM itself resumes the thread, but then also raises an interrupt so that the operating system can schedule the operating system immediately. The latter option is further elaborated in the following when HW-OSQM is applied to the NAMP concept.

Signaling with Ready Flag One class of schedulers is not very common, but can also easily be supported: bitmap schedulers. They don't use a linked list of runnable threads, but instead a single data field with one bit per runnable thread. That limits the number of threads that the operating system can execute to a statically configured number of threads. Anyhow, HW-OSQM can easily support them by using `QUEUE` as a pointer to the bitmap and `THREAD` as an offset into the bit field.

Separate Queue from Operating System In some cases HW-OSQM may still not be flexible enough, despite even Linux can be able to use HW-OSQM. Anyhow, if the target operating system has a ready that is too different or if there are other operations involved at thread wake-up, an alternative is desirable. For such cases I propose to use HW-OSQM in conjunction with a separate *wakeup-queue*. HW-OSQM is configured to add threads to this queue and the operating system then empties this queue for example when the scheduler is invoked the next time or the kernel is entered. HW-OSQM then has the advantage that it already presents the wake-up threads in the scheduler format and thereby abstracts from the nature of the event. This method is supported implicitly by HW-OSQM and is still a valuable alternative to standard event signaling methods like interrupting.

3.5.3 HW-OSQM for NAMP

In Section 3.2.5.3 I have briefly discussed the basic notification methods. Especially the receive side requires the software to keep track of expected messages that have an interrupt activated. HW-OSQM is orthogonal to the basic notification methods, but the NAMP prototype implementation for example makes them optional if HW-OSQM is present.

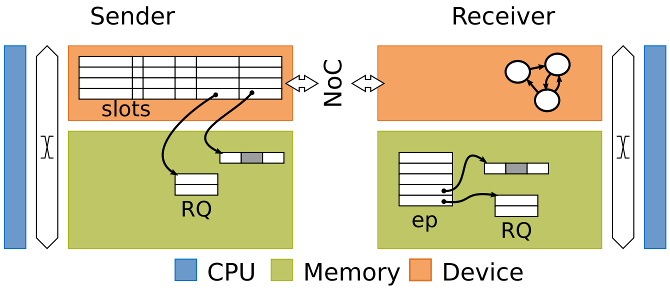


Figure 3.34: Integration of HW-OSQM with NAMP

Figure 3.34 shows the integration of HW-OSQM with the basic NAMP data structures. The OSQM itself is not depicted, it just adds another functional block on both the egress path and the ingress path. On the sender side (egress) the HW-OSQM is part of the slot configuration. The software can set two extra registers in the slot to get resumed after the transfer is completed. After that the software can safely suspend and continues transparently after the successful transfer. In the presence of a virtual interface this feature can only be configured by the kernel, so that the flow slightly differs: The the thread starts the operation and it may decide to suspend, but it is by far not required. A thread may do other computations, prepare the next message etc. If it blocks, the kernel configures the HW-OSQM for the slot.

On the receiver side there is still no state encoded in the NAMP (see Figure 3.34. Instead, the ingress path is a state machine that handles incoming network packets. During finalization of a message it then checks the endpoint data structure (ep) for two extra fields that contain pointers to the ready queue and thread queue element.

3.5.4 Summary

In this section I have discussed the problems with traditional event signaling and presented the HW-OSQM concept to mitigate the impact of event signaling on thread execution. The HW-OSQM does not tackle the event signaling by reducing the event rate or similar, but instead the result of the event: It wakes up a thread by putting it back into the ready queue of the operating system. The analysis has shown the improvement that can be gained by that and a flexible, configurable HW-OSQM feature in the NAMP has been developed.

3.6 NAMP-PS: Hardware-assisted Protection Switching

As discussed in Section 2.6 the main challenge in task migration is to minimize the downtime. The downtime influences the productive processing of the task on the one hand, but on the other hand also directly concerns the communication relations of the task. The migration process can induce significant backpressure on the tasks sending to the migrated task. This can influence the overall system performance. It is therefore desirable to keep the migration time of a task due to the data and code migration at a minimum, which is not in the focus of this thesis. But it is also required that the communication relations are efficiently transferred.

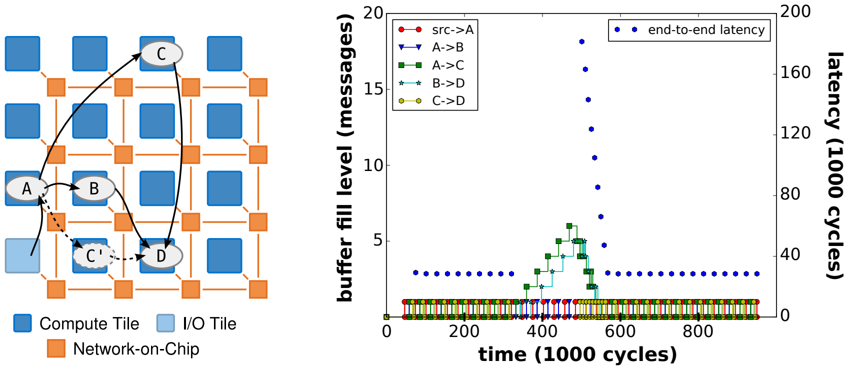
In the following I briefly discuss the issues with task migration and its influence on the communication layer. After that I present how the concept of protection switching can be applied to on-chip message passing. A software implementation of the proposed method is then presented, before I present the integration of protection switching with the NAMP.

3.6.1 Task Migration and Influence on Communication

Once a task migration is triggered the involved communication channels also need to be migrated by the communication subsystem. Three different strategies for the general layout of a task migration have been proposed where the choice depends on the task criticality [174]: *Cold standby* denotes that a new task is started and the communication is newly configured too. With *hot standby* in opposite a task copy already runs at the destination and is capable of taking over instantly. Finally, in *warm standby* a new task and the communication channels are prepared and the task state can then be transferred incrementally or in one transfer. After that the communication channels are switched. The process of switching the communication channels is challenging as mentioned before, especially with increasing number of communication channels.

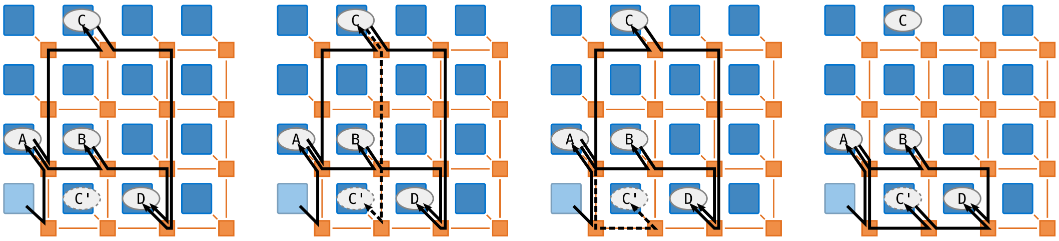
The conventional approach is to pause all communication during the migration. Due to the backpressure, packets are accumulating at the sender(s) during the migration. The transfer of the task state includes unprocessed messages in the endpoints. After the task state is transferred, the task is resumed at the destination. This approach has been implemented on the prototype platform using the simple buffer-based message passing.

The example scenario is sketched in Figure 3.35a. An external I/O interface forwards incoming data to task A. Task A generates data that is consumed by task B and C, which both then send data to task D. Due to some previous platform state, task C is placed in relatively large distance and the system now decides to migrate it to the new



(a) Evaluation scenario **(b)** Transient buffer fill levels and latency

Figure 3.35: Evaluation of transient behavior for pausing during task migration [220]



(a) 1. Pre-migration **(b)** 2. Alternative 1: Forwarding **(c)** 2. Alternative 2: Dualcast **(d)** 3. Switchover

Figure 3.36: Protection switching

position (C'). The transient behavior of the migration is plotted in Figure 3.35b. After the task migration starts first the buffer fill level between A and C starts increasing. As D is waiting on data from C it follows soon and pauses consuming data from B too. When the migration completes the buffers drain again as it is a well balanced application. Anyhow it can be seen that the end-to-end latency of the individual data items spikes after the migration, an effect that may cause trouble in soft real-time applications or similar.

3.6.2 Communication Migration with Protection Switching

In the following alternative approaches on the migration of the communication channels are discussed, that are dependable and have minimal impact on the application performance. A basic property is that the task migration and the communication migration are separated and handled mostly independently from each other. During migration

<table>
<tr><th></th><th>Pausing</th><th></th><th>Protection Switching</th></tr>
<tr><td rowspan="3">I</td><td></td><td>&lt;</td><td>replicate communication structure</td></tr>
<tr><td>suspend incoming channels</td><td>&lt;</td><td>dualcast/forward incoming channels</td></tr>
<tr><td></td><td>&lt;</td><td>wait until not distributed packets are processed</td></tr>
<tr><td rowspan="7">II</td><td colspan="3">stop task</td></tr>
<tr><td colspan="3">suspend outgoing channels</td></tr>
<tr><td>migrate communication structure & buffers</td><td>&gt;</td><td>sync receive buffers</td></tr>
<tr><td colspan="3">move control to destination</td></tr>
<tr><td colspan="3">remap task on local communication structure</td></tr>
<tr><td colspan="3">resume outgoing channels</td></tr>
<tr><td colspan="3">resume task</td></tr>
<tr><td>III</td><td>resume incoming channels</td><td>&lt;</td><td>switch-over incoming channels</td></tr>
</table>

Table 3.3: Migration sequence of different methods for communication migration. The individual steps are compared with respect to their analyzed complexity.

both layers prepare the switchover concurrently. Instead of pausing the communication during switchover, protection switching maintains a consistent and active communication subsystem. Two protocols have been proposed that are briefly discussed in the following. Figure 3.36 shows both alternatives for the previously introduced scenario.

Forwarding With forwarding, the idea is basically that the communication subsystem forwards incoming data transparently to the new destination. After the switchover the sender is then updated to send directly to the new destination. One major drawback is that it creates extra traffic at a critical part of the system that can easily form a bottleneck. Furthermore it is important to properly synchronize once the task takes over.

Dualcasting With the dualcast alternative the sender instead transfers the messages to both the old task location and the new task location concurrently. After switchover the sender has to switch to the new channel. Again, it is important to properly synchronize between both communication channels after switchover.

3.6.2.1 Comparison of Migration Phases

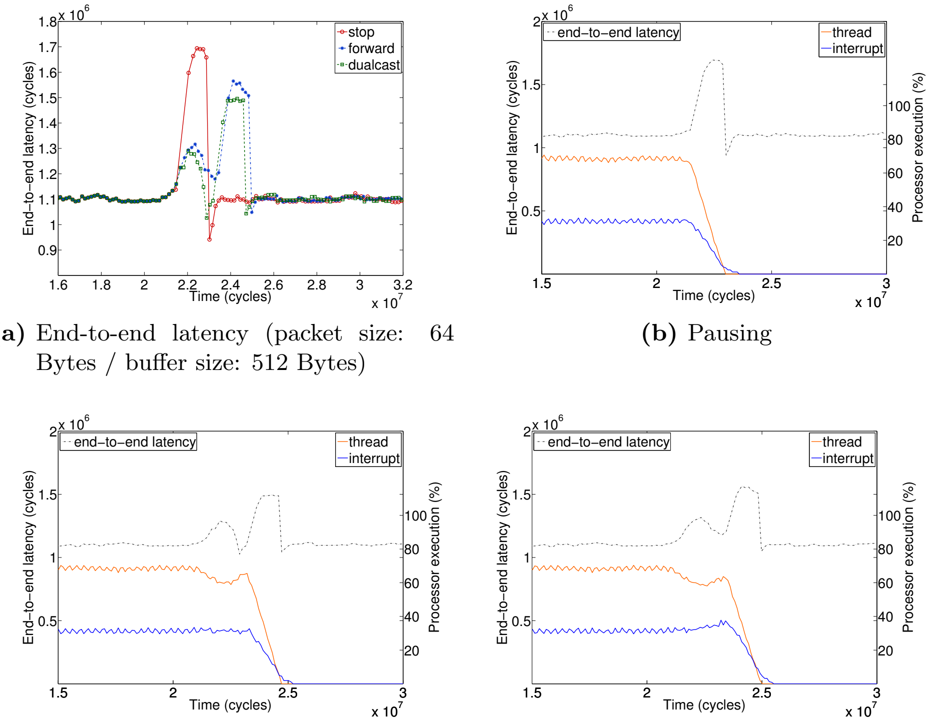
The different phases of the migration are listed in Table 3.3. The actions in the different phases are executed either in the current location or in the new location. Table 3.3 outlines a comparison between the traditional pausing and protection switching. The table is organized along the three phases of a migration: I) pre-migration, II) switchover, and III) post-migration. The phases are centered around the actual task migration in switchover phase. The communication migration does not play a role in this phase with protection switching: The buffer synchronization is significantly faster than the migration of the entire communication state. In exchange the communication migration performs more actions during the pre-migration phase and the post-migration phase. There are two aspects to protection switching now: First, the goal is to perform those extra operations with minimum impact. Second, the fundamental improvement is that the migration is moved out of the critical switchover phase and performed asynchronously. The phases are briefly discussed in the following.

Pausing In phase I the incoming channels are suspended by sending control messages to the sending tasks. Phase II is the full task switchover that included suspension of the outgoing channels. The migration of communication is then the transfer of all data structures and pending messages related to the communication. The control is transferred to the destination, where the task is mapped back and both outgoing communication and task are resumed. Phase III triggers the resuming of the incoming channels then.

Protection switching Phase I consists of more steps when using protection switching. The communication structure gets replicated at the new destination. The endpoint and buffer configuration is transferred, but the original endpoints and buffers stay active. Now the new packets are either forwarded to the destination or the predecessors are triggered to perform dualcasts. The new endpoints are configured in a special mode, where they drop messages on overflow and the flow control still occurs with the original tile. For consistency the previous packets must be processed or ultimately migrated. This synchronization is executed during switchover. Finally, in phase III the predecessors are instructed to send to the new task making the new communication channels active.

3.6.3 Analysis of Software-based Protection Switching

The concept of protection switching for on-chip message passing has been validated with a software-based implementation on our platform (see Chapter 4). A simple three task chain is evaluated where the middle task is migrated. The software impact and the latencies of packets are measured.



(a) End-to-end latency (packet size: 64 Bytes / buffer size: 512 Bytes)

(b) Pausing

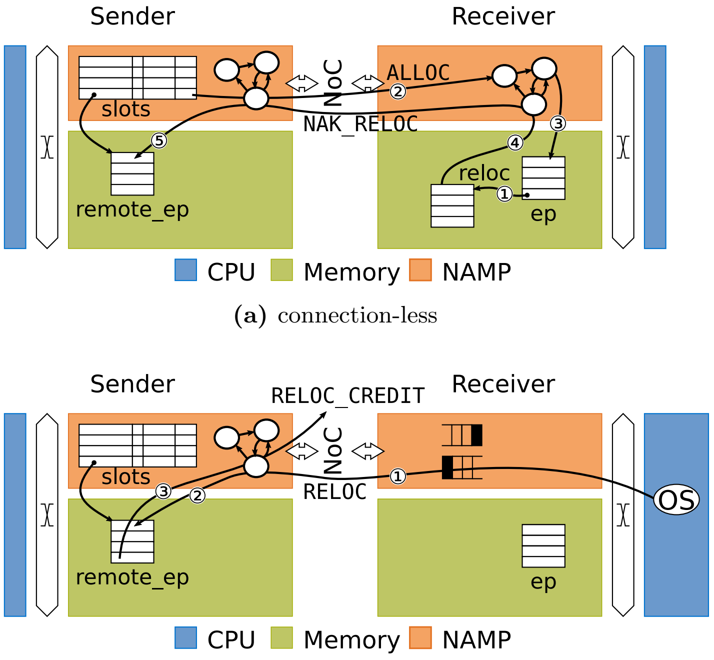
(c) Dualcasting

(d) Forwarding

Figure 3.37: End-to-end latency of communication migration, software implementation [220]

End-to-end latency Figure 3.37a plots the end-to-end latencies for all three migration alternatives in a transient plot. It is generally consistent with the analysis from before. While protection switching lead to a smaller peak latency, the number of affected packets increases. The reason that the peaks of protection switching are higher than expected lies in the fact that in this simulation the software handles all operations related to the communication.

Impact on Processor Execution Beside the latency, Figure 3.37 furthermore shows the relative time spent in the thread and in protocol handling at the original tile. It is clear that the overall share is similar with all three approaches, but the transient behavior of the relative times nicely replicates the aforementioned operations (see Table 3.3): For protection switching the impact of the pre-migration phase is visible. The migration operations temporarily suppresses the task execution. This also manifests in extra increases of the latency. Before switchover the task processing regains processor



(a) connection-less

(b) connection-oriented

Figure 3.38: NAMP integration for forwarding during migration

time. With forwarding the extra message handling at the original task location becomes visible, too.

3.6.4 NAMP Integration

The concept of protection switching for on-chip message passing shows good potential in analysis and simulation. Probably the most important factor is that it deskews the load on the current task position during switchover. Instead it performs migration operations before and after the switchover phase, separated from the actual task migration. One of the major issues with the software presented previously implementation is that the load on the current position is yet high due to the operations performed in software. Hence, those operations are a good candidate for hardware offloading and in the following I present the low complexity NAMP feature, but yet have the potential to significantly lower the impact of communication migration.

3.6.4.1 Forwarding/Relocating

The previous discussion suggested forwarding as protection switching mechanism. Forwarding pessimistically assumes that the message arrives at the current task position as a clear separable unit. With the NAMP concept this does not hold, because for the bulk data transfer it utilizes a potentially existing DMA controller and eager transfers are not supported. Only a limited number of protocol messages are required beyond this.

Beyond this aspect it is generally favorable to not transfer a message twice. So the path that NAMP-PS takes is using *endpoint relocation* instead of message forwarding. This essentially involves a relocation update on the sender side so that it sends to the new location. Figure 3.38 shows two NAMP-PS integrations for endpoint relocation, one for connection-less communication and one for connection-oriented communication.

For **connection-less communication** (Figure 3.38a) the notification about the relocation is propagated during the message allocation phase as the number and origin of messages is generally unknown in connection-less communication. For that the local endpoint data structure (see Figure 3.3a) is extended with a field for relocation information. There is no relocation as long as it is zero. After that it points to the new remote endpoint. Once the pre-migration phase starts (near to the actual switchover) the operating system sets this field to a new data structure for the newly created remote endpoint at the new destination (①). When a new allocation request now arrives (②) the NAMP target state machine checks for this field and finds that the endpoint is relocating (③). It then replies with a `NAK_RELOC` message (④). Beyond the status, this message contains the new (`reloc`) remote endpoint data. The initiator NAMP then updates its own data structure atomically (⑤). After that it will only send data to the new location and the current transfer is restarted.

For **connection-oriented communication** (Figure 3.38b) there are two main differences. First, there is no allocation message so that the protocol does not work in this case. But on the other hand it has the advantage that the communication partner is known. So, in this case there is a relocation message that signals an update to an endpoint location (remote tile and address). There is no extension required to the endpoint data structures for this method. As depicted in Figure 3.38b the relocation message is furthermore sent directly by the operating system (①). As the operating system is already running there is no significant overhead and saves extra hardware area. So the operating system sends the message with the basic buffer. Once the message arrives at the sender it again atomically updates the remote endpoint data structure (②). Finally, there may be messages still in transfer to the remote endpoint. To avoid inconsistencies

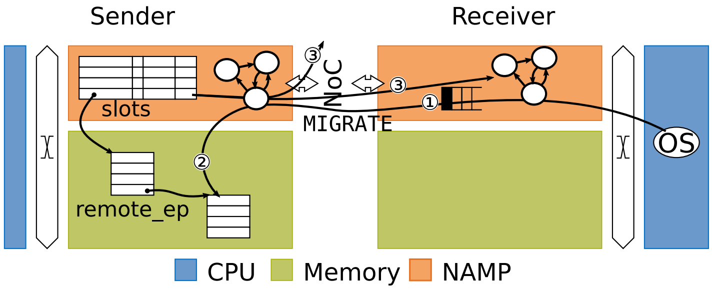


Figure 3.39: NAMP integration for dualcasting during migration

the current update is send to the remote endpoint with a special RELOC_CREDIT message (③).

3.6.4.2 Dualcasting

For the second protection switching method the required hardware integration is again limited, because the NAMP already includes dualcasting capabilities via the NAMP-CC multicast feature. As depicted in Figure 3.39 the required feature is again a message triggered by the operating system once the pre-migration phase starts (①). Multicasting requires a different remote endpoint data structure (see Figure 3.27b). If an endpoint is connected to a communication partner that may be relocated (worst case all) it therefore needs this data structure instead of the default remote endpoint. Anyhow, it only wants to dualcast when needed.

So the proposed feature involves a reserved memory area that can hold the multicast data structure for two targets where one target is initialized to the original task location. Beside that the normal remote endpoint data structure is extended by a pointer to this data structure. Once the MIGRATE message arrives it contains the updated endpoint location. The NAMP then updates the multicast data structure (②). In the following the NAMP software driver *and* the continuous communication mode (CCM) handling need to perform a dualcast operation instead (③).

3.6.5 Summary

In this section the impact of task migration on the communication infrastructure has been investigated. The impact of the migration of the endpoint metadata and actual buffer content on the overall task migration can be significant. Protection switching is a concept to separate the communication migration from the task migration. Two methods have been introduced: Forwarding/relocation redirects the arrival of new data to the updated task position while buffer data is still drained at the original position. With dualcasting the sender transfers data to both positions during the migration. Both concepts have been introduced as NAMP features.

3.7 Summary

In this central chapter of this thesis I have developed an efficient "Network Adapter for Message Passing (NAMP)" for on-chip communication in tiled many-core system-on-chip. The potential of a hardware offload was evaluated and compared to state-of-the-art approaches. The basic functionality of NAMP was developed throughout the chapter. Features that improve the usage of NAMP in many-to-many communication and a self-virtualization interface for elegant usage by isolated userspace tasks have been introduced. The wake-up of threads by HW-OSQM was presented as a significant improvement of event signaling between the NAMP and threads waiting for send or receive operations. Finally, protection switching has been introduced as a method for communication migration.

This chapter focused on the presentation and discussion of the basic concepts. An evaluation framework served for quantitative estimation of the improvement of a functionality in an abstracted way. Beside that the evaluation framework played an important role in validation of the protocol.

While the estimated improvements clearly indicate that NAMP dominates currently dominating approaches, there is a trade-off: On the one hand the platform designer can expect a certain performance improvement by the NAMP hardware assist. On the other hand the extra hardware contributes to the chip area and power profile. Hence, the a prototype of the concept will in the following be evaluated and validated. Based on those results, the design points of performance improvement and extra area can serve the decision process to integrate NAMP.

4 Prototype Implementation

In the previous chapter I have introduced the concept of the *Network Adapter for Message Passing (NAMP)*. The discussion abstracted from an implementation where sensible, but with many ideas that define the hardware implementation. The NAMP was validated against an analysis framework that models the functionality composed of basic building blocks. Those building blocks were calibrated based on the prototype framework used in this chapter. But on the one hand it is important to validate an actual implementation, and on the other hand there is always a trade-off. While the evaluation metrics (overhead, gap and latency) are all significantly improved for each of the proposed extensions, they don't come for free. In this chapter I present the prototype implementation of the NAMP. It is used to derive estimates for hardware area and timing, and for the concept validation. The features and configurable parameters of the NAMP are evaluated in a sensitivity analysis where they lead to differences. A set of typical configurations is used to define NAMP instances for common use cases.

The syntheses in this chapter have been performed for two targets:

FPGA For the measurement of resource utilization and timing, the modules are synthesized for a Xilinx Kintex 7 UltraScale (xcku035-1), which is an FPGA device of the newest generation with average performance and size. The example system designs are targeted at devices of typical FPGA boards and noted where introduced. The synthesis tool is Xilinx Vivado 2017.4.

ASIC The designs are synthesized for a state-of-the-art TSMC 40nm low power process. The design area is converted into gate equivalents (NAND2 gate size). The synthesis tool is Synopsys Design Compiler L-2016.03-SP3 and only a frontend synthesis is performed. This already gives a very good estimate of the timing and is generally precise with respect to the number of gates.

This chapter is organized as follows: In Section 4.1 I present the many-core system-on-chip prototype which was developed essentially for this thesis. The prototype is designed to be a generic platform that mimics an average layout and behavior. The

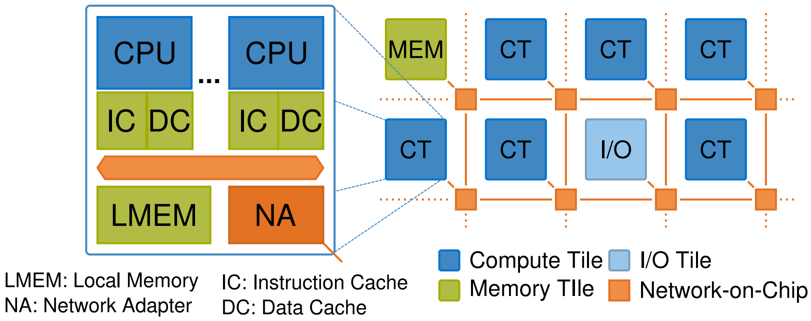


Figure 4.1: Basic OpTiMSoC Layout

NAMP implementation in this prototype is then presented and validated in Section 4.2. In Section 4.4 I then conclude the discussion with a presentation of typical configurations and their area and timing.

4.1 Many-Core System-on-Chip Prototype

Around my work presented in this thesis a many-core system-on-chip prototype was developed: the "Open Tiled Manycore System-on-Chip (OpTiMSoC)" [218, 160]. It is made publicly available as open source project and can be used for many related research topics. OpTiMSoC is not a single system-on-chip, but instead the set of basic building blocks, composite modules and tools to configure and build tiled manycore system-on-chip. The main components are:

OpenRISC mor1kx The main processor is a 32-bit CPU with a 6-stage pipeline. It has SMP extensions to support multiple instances in one tile.

Tiles A number of tile modules are available. The most important one is the compute tile: It contains a configurable number of processor cores, local SRAM memory, a boot ROM and the network adapter. Other important tile types are memory tiles that provide access to DDR memory and I/O tiles that allow access to external devices.

Network-on-Chip A basic network-on-chip router with configurable input and output buffers is the basis for each instances network-on-chip. The router's basic properties are that it is packet-based, wormhole and uses distributed dimension routing. The network-on-chip of an OpTiMSoC instance is automatically generated from those routers and it is configurable to build multiple physical networks or one network with virtual channels.

Support Libraries and Runtime System Libraries with the necessary drivers and a simple runtime system are part of OpTiMSoC to allow the user a simple quick start into programming of tiled many-core system-on-chip architectures.

Debug Infrastructure A modern trace-based debug infrastructure that allows the platform designer and application developer to observe the execution of software and state of the hardware. It has been spawned into a separate project "Open SoC Debug" [159] recently and is also used in other projects [131].

Figure 4.1 sketches the basic elements and hardware structure of an OpTiMSoC instance. The tiles are organized in a regular structure and the compute tiles contain a configurable number of processors. This allows tasks running in the tile spawning multiple threads to exploit fine-granular parallelism. The local memory allows for maximum locality and inter-tile communication is enabled by the network adapter. All components are configurable and the NAMP as default network adapter will become an integral component for efficient programming of OpTiMSoC platforms. In the remain of this chapter the NAMP implementation will be presented and discussed.

4.2 NAMP Implementation

In Chapter 3 I have presented the NAMP concept. In the following the prototype implementation of the NAMP is presented. I first present the NAMP module and its integration, followed by a presentation of the buffer implementation. This section then concludes with an overview of the individual components of the NAMP.

4.2.1 NAMP Module Overview

As mentioned before, the NAMP is a single self-contained module that contains the described features in a configurable way. Figure 4.2 shows the NAMP module and its components. There is not a one-to-one correspondence of features to components, because the majority of the features extend the main modules. There are two modules that are common in the state-of-the-art network adapters: the NoC buffers and RDMA modules are common modules. The NAMP interfaces the network-on-chip, the tile bus and and interrupt line that is fed to one of the cores.

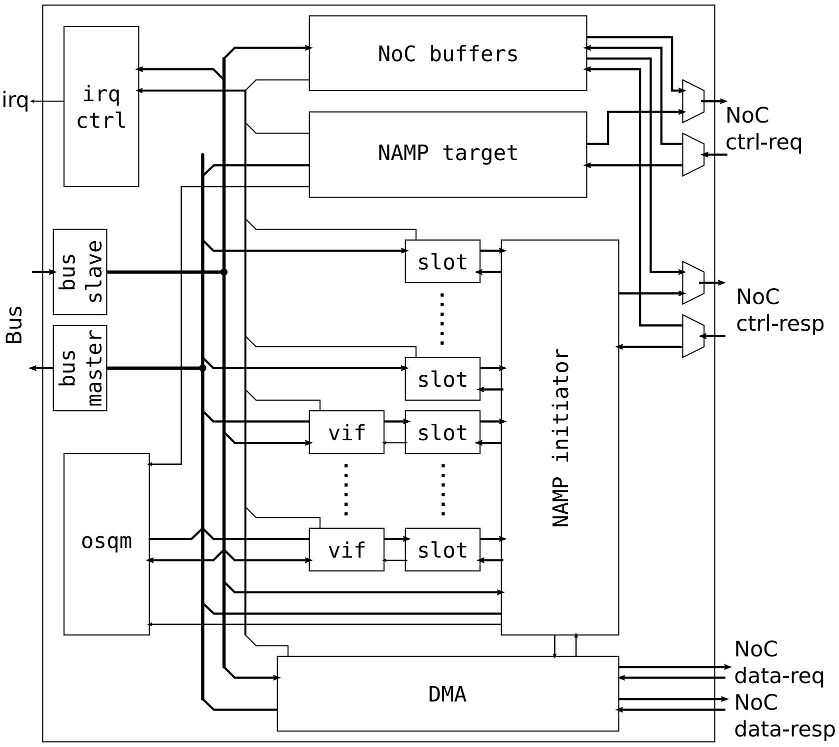


Figure 4.2: Overview of the NAMP Module

On the bus there are two interfaces: The slave interface is used to configure the NAMP components and gather status and configuration information. It is also used to read flits from the NoC buffers. The master interface has the same memory view as the processors. It is used by the modules to access the local memory. The DMA copies data into the memory and state machines from the other components use the master interface to read and manipulate data items as described throughout this thesis.

The NAMP module accesses four network-on-chip channels. A set of two channels is used for control messages as generated by the NAMP progress engone or injected via the NoC buffers. Another set of two channels is used to transfer data between tiles. Each of those two sets has a request and a response channel to mitigate message-dependent deadlocks.

The central modules of the NAMP are the `initiator` and the `target` component. The `initiator` component encapsulates the progress engine and generates message passing protocol requests. The `target` receives those requests, performs the corresponding action and sends the response.

Each `slot` component configures one transfer and tracks its state. The `initiator` selects a slot that can progress and performs the required (protocol) action and then updates the slot status.

4.2.1.1 Configurable Features

The NAMP module is freely configurable to allow the use in a variety of scenarios. The configuration settings (as Verilog parameters) allow to configure a NAMP instance by either enabling components or features or as parameter. In the following I briefly discuss the most important configuration settings.

`SLOTS_PHYS`: **Physical message passing slots** This parameter configures the number of slots mapped directly into the physical memory space. The NAMP instantiates that many `slot` components.

`SLOTS_VIRT`: **Virtual message passing slots** This parameters configures the number of slots that are accessed via a virtual interface by a task. Each of those virtual message passing slots instantiates a pair of the virtual interface component (`vif`) and a `slot` component.

`ENABLE_CC_MULTICAST`: **Enable multicast operations** This parameter enables the multicast capability of the NAMP. The collective communication features can be independently added, because a platform designer may choose to only add multicast.

`ENABLE_CC_SCATTERGATHER`: **Enable scatter and gather operations** This parameter adds both the scatter and the gather collective communication operations. As they are a pair of operations, I chose to keep them together as one feature.

`ENABLE_CC_REDUCTION`: **Enable reduction operations** This parameter enables the reduction collective communication operation.

`ENABLE_OSQM`: **Enable Operating System Queue Manipulation** The OSQM feature can be independently activated. It is important to note that it is not coupled to the virtualization extensions, because also bare-metal operating systems or kernel threads may use this feature.

ENABLE_PS: Enable Protection Switching This enables the protection switching operations described before.

There are some more parameters specific to components of the NAMP that are introduced in the following when needed.

4.2.1.2 NAMP Memory Map

The NAMP can be accessed from software via its slave interface. This interface can be mapped to any base address in the physical address space, as long as it is mapped aligned to the address space size. The address space size is 4 MB (`0x0-0x3fffff`). The memory map is shown in Table 4.1. The components are individually mapped into the address. It is important to note that the address space is only sparsely populated. For example only the first 12 addresses in the NAMP's "basic status & configuration" space are used, the remaining addresses all lead to a bus error when accessed.

The page size of the processor is 8 kB. This is mostly relevant for the "vif config" space, because it has to be mapped into a thread's virtual address space. Anyhow, all address sub-spaces are of this size to allow even more mapping opportunities. The addresses of the basic NAMP interface are used to query the configuration of the NAMP and gather information about the currently active slots. The NAMP IRQ controller provides the information which interrupts from the components are currently active and allows to mask interrupts individually to ignore them. This is especially useful when the OSQM feature is deployed.

There are two NoC buffers, one for requests and one for responses. Flits can be sent or received on those buffers as memory mapped I/O. Each of the buffers is mapped into one page. Again, this can be used for example allow mapping them to different microkernel services. The DMA *user* slots are equally mapped, and a maximum of 128 DMA slots can be used. The number of slots can be queried from the "basic status & configuration". The slots used by the NAMP cannot be accessed from outside the module.

Finally, the message passing transfers can be configured via the address space of the NAMP slots. Each of the slots has the address space of two pages. This is due to the fact that the lower addresses always access the slot configuration, while the upper addresses are used by the virtual interface. A thread running in virtual memory thus has the upper page mapped into its address space. The operating system kernel could still inspect the physical addresses as configured in the slot or configure the OSQM registers in the slot.

Address Range	Description	Module/ Function
`0x000000 - 0x001fff`	Basic status & configuration	NAMP
`0x002000 - 0x003fff`	Interrupt interface (read pending, configure masking, ..)	NAMP IRQ Controller
`0x004000 - 0x005fff`	OSQM basic configuration (optional)	OSQM
`0x010000 - 0x011fff`	Network-on-chip basic buffer 0	Buffers
`0x012000 - 0x013fff`	Network-on-chip basic buffer 1	
`0x100000 - 0x101fff`	DMA slot 0 (optional)	DMA
`0x102000 - 0x103fff`	DMA slot 1 (optional)	
`...`		
`0x1d0000 - 0x1fffff`	DMA slot 128 (optional)	
`0x200000 - 0x200fff`	NAMP slot 0, base interface	NAMP
`0x201000 - 0x201fff`	NAMP slot 0, vif config (optional)	
`0x202000 - 0x203fff`	NAMP slot 0, virtual interface (optional)	
`0x204000 - 0x204fff`	NAMP slot 1, base interface (optional)	
`0x205000 - 0x205fff`	NAMP slot 1, vif config (optional)	
`0x206000 - 0x207fff`	NAMP slot 1, virtual interface (optional)	
`...`		
`0x3a0000 - 0x3affff`	NAMP slot 128, base interface (optional)	
`0x3b1000 - 0x3b1fff`	NAMP slot 128, vif config (optional)	
`0x3c0000 - 0x3fffff`	NAMP slot 128, virtual interface (optional)	

Table 4.1: NAMP memory map

In case of physical slots only the lower page is available, the upper page will raise a bus error when accessed. There is space reserved for up to 128 NAMP slots.

4.2.2 Circular Buffer Design

Before I get into the details of the components of the NAMP module, I will elaborate on the circular buffers that are interfaced by the NAMP and the software. It is important that the circular buffers i) are non-blocking data structures, ii) can handle both concurrent read and write operations, and iii) can be accessed efficiently from hardware and software. As introduced in Section 3.2.4.1, the circular buffers reside in the tile local memory. To support efficient use of the circular buffer, it should support both *multiple producers* and *multiple consumers*. By supporting multiple producers the buffer implementation allows first multiple software threads writing to one sender buffer, and second local software threads writing to the same receiver buffer as the NAMP. By supporting multiple consumers multiple software threads can read from the receiver buffer safely.

Having parallel write and read operations requires synchronization. Atomic operations must guarantee that concurrent accesses are sequentialized to avoid inconsistent states. A common approach to guarantee atomicity is using locks. Each producer or consumer tries to set a lock and waits if not successfull. Only the one holding the lock may update the data structure. After completion it then releases the lock and another entity can resume. While this is a straight forward approach, it has problems even if producer and consumer accesses are spli into two locks. Most importantly in the context of this thesis, a thread could be paused while writing to the data structure. This would lock out the NAMP to write to the buffer until the thread is resumed, which induces critical backpressure to the network-on-chip. Of course a kernel (privileged) thread or baremetal software could block interrupts to avoid this issue. But an important aspect in this thesis is the availability of the message passing infrastructure to user level (non-privileged) threads.

The solution – and generally favorable – are non-blocking data structures, which are carefully implemented using atomic primitives such as *compare-and-swap (CAS)* [208, 145]. The CAS operation allows a thread to read a value, compare it to a value and only swap it with a new value in case of a match. This operation is implemented by the CPUs and the NAMP by a *read-modify-write (RMW)* cycle on the bus. Here those three sub-operations are executed without interference from other parties. Non-blocking data structures are now built by updating elements of the data structures by executing such CAS operations, and thereby allow for efficient and fast concurrent access to the shared data structure.

```c
struct mp_cbuffer {
  uint32_t capacity; // buffer capacity (2^capacity)
  uint32_t max_msg_size; // max element size (2^max_msg_size)
  volatile uint32_t wr_idx; // next write index
  volatile uint32_t rd_idx; // next read index
  uint8_t *data; // data field (size=capacity*max_msg_size)
  uint32_t *size; // field of message sizes
};
```

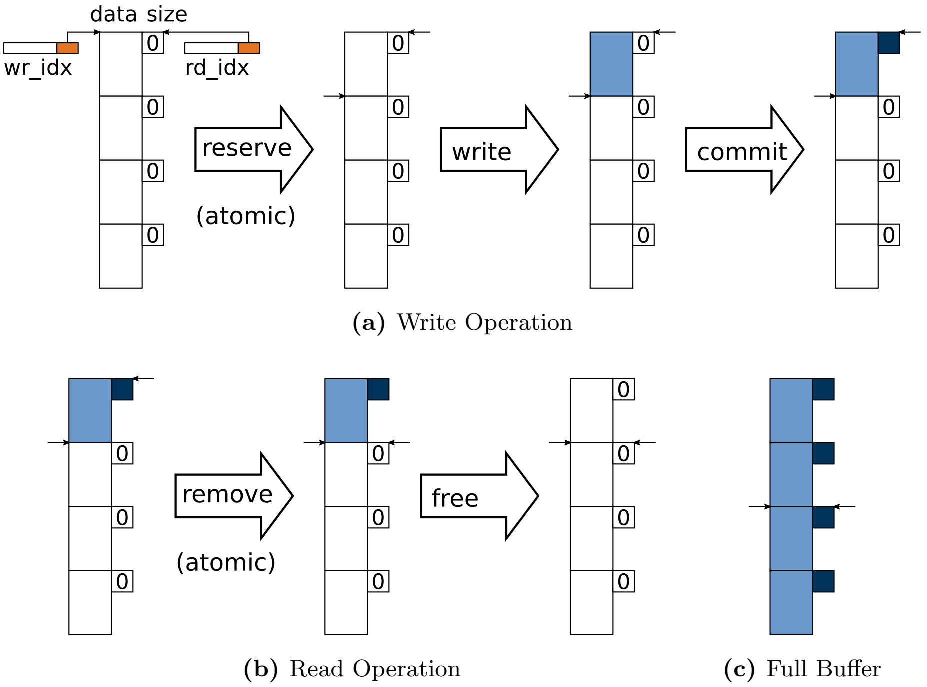
Listing 4.1: Buffer data structure

The buffer is organized as introduced before in Figure 3.3b: The messages are stored into a contiguous data block. A write index (`wr_idx`) points to the position where the next data will be stored and the a read index (`rd_idx`) points to the position where the next data will be read from. Listing 4.1 shows a reference implementation of such a buffer. I chose a fixed size, pre-allocated circular buffer as it avoids dynamic memory allocation that increases complexity of hardware implementations. The operations on the buffer consist each of two steps briefly described in the following.

A write or *push operation* consists of two steps as sketched in Figure 4.3a. First, the next buffer space is reserved by moving the write index forward. This has to be an atomic operation so that only one producer is successful. After successful reservation the data is written, which is now safe to do. Combining both steps is not possible. Finally, a commit operation is a simple write operation that signals that this element is complete. It writes the size of the written data to a metadata field. A receiver only reads a buffer elements after that field is written.

At a first glance the described procedure seems to be sufficient to ensure thread-safe concurrent access. But a problem arises due to the nature of non-blocking data structures with CAS, often refered to as the ABA problem [211, 144]. If a producer reads the index, a situation can occur where other producers increment and wrap the index in a way that it has the same value as it read, for example because the thread was paused. The CAS operation would then be successful, but the buffer can now be full as depicted in Figure 4.4a.

Generally, the ABA problem is most often mitigated by using load-linked/store-conditional operations or multi-word compare-and-swap, but they induce hardware requirements beyond a simple CAS. Instead I chose an *optimistic* approach: The indices are counted up to their maximum 32 bit value (or processor register size) and wrap around auomatically. For addressing, only the relevant bits are used from the index as

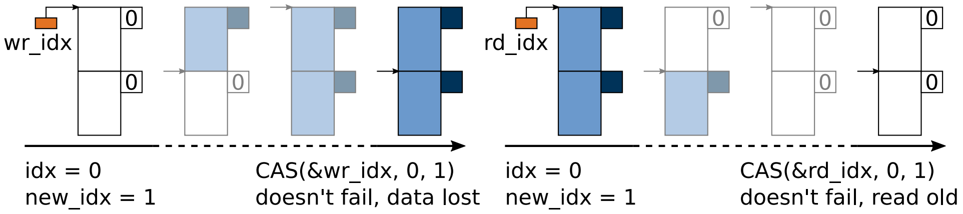


(a) Write Operation

(b) Read Operation

(c) Full Buffer

Figure 4.3: Multi-Producer Multi-Consumer Buffer Operations



(a) Write

(b) Read

Figure 4.4: ABA problem for circular buffers

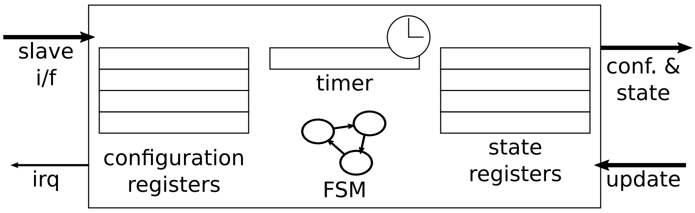


Figure 4.5: NAMP `slot` component

depicted in Figure 4.3a. The remaining bits are interpreted as a tag. Assuming sensible values for the maximum number of messages in a buffer of 2^1 to 2^6 there need to be at least 2^{26} operations until the tag has the same value again. This is more then sufficient to mitigate the ABA problem, but other software-based measured could be added to ensure it to the maximum extend (which is beyond the scope of this thesis).

Listing 4.2 shows the software implementation of the push and pop operations. Using powers of two for the number of elements and maximum message size (reserved chunk of data) eases a hardware implementation significantly. Reading data from the buffer, the *pop operation* similarly consists of two steps. First the index pointing to the next element to read is moved forward. After data is read the element is then freed (see Figure 4.3b). Listing 4.2 also shows the reference implementation of the pop operation. Again, the implementation is hardware friendly which is a major goal of this thesis. Finally, a circular buffer is empty (or not yet ready to read) when the currently pointed to element has size 0 (see initial state in Figure 4.3a. It is full when the element that the write index points to has (still) a size and hence not writeable (see Figure 4.3c.

Summarizing, the buffer implementation described above is a flexible, scalable concept. It is designed to be integrated with software and hardware efficiently.

4.2.3 NAMP Components

As depicted in Figure 4.2 the NAMP module is composed of multiple components. The NoC buffers and RDMA components form a typical state-of-the-art network adapter. The NAMP adds the message passing progress engine, handling of concurrent transfers and extra functionality added by the features discussed in this thesis. In the following I will briefly describe the important components.

4.2.3.1 NAMP Slots

Each message passing transfer is configured with one NAMP slot. As depicted in Figure 4.5 each slot has a configuration slave interface that is addressed according to the

```
// Push operation
int mp_cbuffer_reserve(struct mp_cbuffer *buf, uint32_t *idx) {
  uint32_t cur_idx, new_idx, rd_idx;
  do {
    cur_idx = buf->wr_idx; // read current counter
    *idx = cur_idx % (1<<buf->capacity); // extract index
    rd_idx = buf->rd_idx;
    if ((*idx == (rd_idx % (1<<buf->capacity)))
        (rd_idx != cur_idx)) return -1; // full (reserved)
    if (buf->size[*idx] != 0) return -1; // full
    new_idx = cur_idx + 1; // calculate new counter
  } while(CAS(&buf->wr_idx, cur_idx, new_idx) != cur_idx);
  return 0;
}

void mp_cbuffer_commit(struct mp_cbuffer *buf, uint32_t idx,
                       uint32_t size) {
  buf->size[idx] = size;
}

int mp_cbuffer_push(struct mp_cbuffer *buf, uint8_t *data,
                    uint32_t size, uint32_t *idx) {
  if (mp_cbuffer_reserve(buf, idx) != 0) return -1;
  memcpy(buf->data + *idx*(1 << buf->max_msg_size), data, size);
  mp_cbuffer_commit(buf, *idx, size);
  return 0;
}

// Pop operation
int mp_cbuffer_read(struct mp_cbuffer *buf, uint32_t *idx) {
  uint32_t cur_idx, new_idx;
  do {
    cur_idx = buf->rd_idx;
    new_idx = cur_idx + 1;
    *idx = cur_idx % 2^buf->capacity; // extract index
    if (buf->size[*idx] == 0) return -1; // empty
  } while(CAS(&buf->rd_idx, cur_idx, new_idx) != cur_idx);
  return 0;
}

void mp_cbuffer_free(struct mp_cbuffer *buf, uint32_t idx) {
  buf->size[idx] = 0;
}

int mp_cbuffer_pop(struct mp_cbuffer *buf, uint8_t *data,
           uint32_t *size) {
  uint32_t idx;
  if (mp_cbuffer_read(buf, &idx) != 0) return -1;
  *size = buf->size[idx];
  memcpy(data, buf->data + idx*(1 << buf->max_msg_size), size);
  mp_cbuffer_free(buf, idx);
  return 0;
}
```

Listing 4.2: Circular buffer push and pop operation

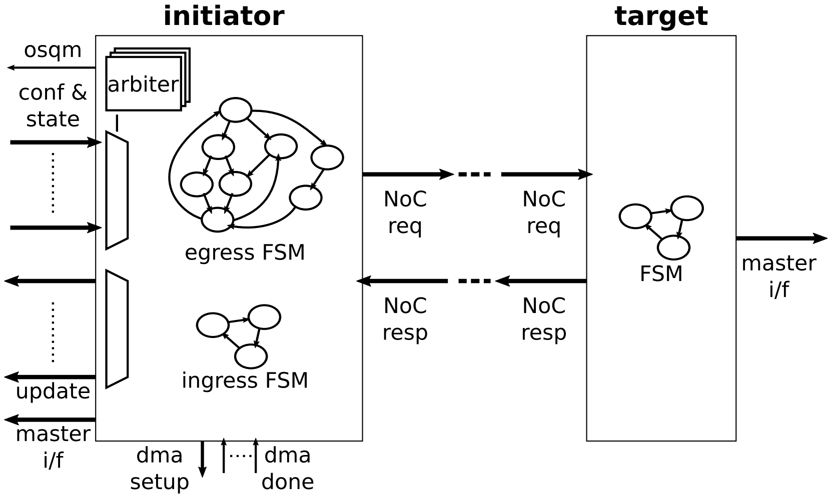


Figure 4.6: NAMP progress engine

memory map. Via this interface the software writes the configuration registers in the slot. Beside that there are state registers that capture the progress of the transfer and temporary data such as generated addresses.

The configuration and the state registers are exposed to the progress engine that resides in the NAMP initiator module. Once the `VALID` flag is written the progress engine is triggered and performs actions according to the current state. After each time the slot was served an update is triggered, that contains an update of the state registers. Some of the state transitions trigger a timer that is also part of the slot. A tiny finite state machine controls the implicit state transitions and timer. It also sets the `READY` flag once the transfer is done. This also triggers the interrupt (irq) eventually.

4.2.3.2 NAMP Progress Engine

The NAMP progress engine is the main functionality of the NAMP. It implements the control path of the message passing protocol. The main part of the NAMP progress engine is the *initiator* which serves the slot requests at the sender side. The *target* at the receiver side serves incoming requests, but has no state. As depicted in Figure 4.6 the network-on-chip channels complement via the network between the sender (initiator) and the receiver (target).

As sketched in Figure 4.6 the initiator has two state machines: The *egress FSM* serves the slots and generates the protocol messages, accesses the bus to read and write endpoint data, and handles the slot state updates. It is the most complex part of the

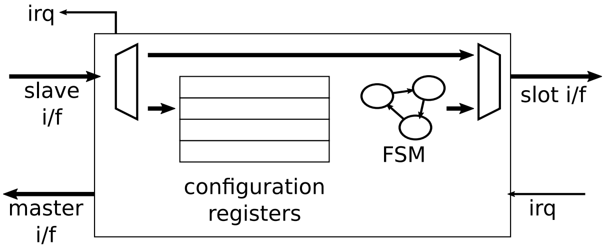


Figure 4.7: NAMP `vif` component

entire NAMP. The number of FSM states varies depending on the configured features. Generally, the collective communication and protection switching mainly contribute to the state machines in the progress engine. The *ingress FSM* serves incoming network-on-chip packets and updates the slots accordingly.

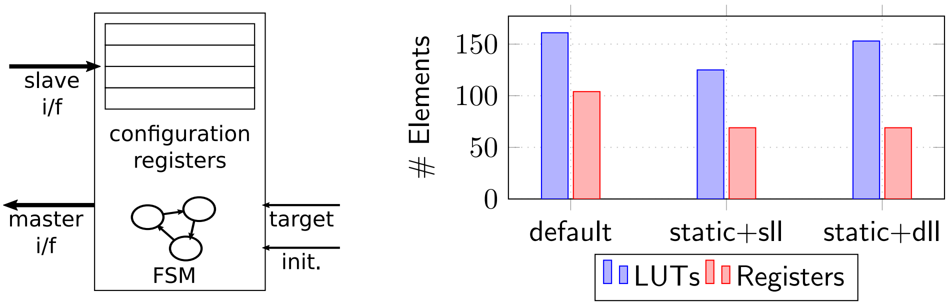
Whenever idle the egress FSM uses a set of arbiters to select the next slot to process. There is one arbiter for each progress state of a slot. The arbiters serve the slots fairly and the egress FSM picks the from the arbiter that selected the most advanced slot. The selected slot is then processed with its configuration and state and the slot state gets updated by the egress FSM. It also configures the RDMA controller and processes the completion of RDMA transfers to progress the associated slot.

The complexity of the FSM in the target module also depends on the configured features. This is mostly which packet types it accepts and processes. Once a packet arrives it accesses the addressed data structures in the tile memory via the bus. Based on the performed action it then generates a reply for some of the requests, such as allocation requests.

4.2.3.3 NAMP Virtual Interfaces

The NAMP virtual interface for all configured virtual slots. Figure 4.7 shows that the virtual interface essentially is a transparent translation from the bus slave interface to the slot interface. For the lower page addresses (see above) the virtual interface is bypassed. This is useful for the kernel to access the slot and configure the OSQM once the thread suspends.

As sketched in Figure 4.7 the upper page addresses are mapped to the virtual configuration registers. Once the `VALID` flag is written, a state machine is triggered in the virtual interface. It accesses the thread's endpoint table as configured by the kernel. The virtual identifiers are then translated to physical addresses and written to the slot



(a) Details

(b) FPGA Synthesis with different configuration parameters

Figure 4.8: NAMP OSQM module

interface. In the last step the virtual interface transfers the flags and thereby completes the slot configuration.

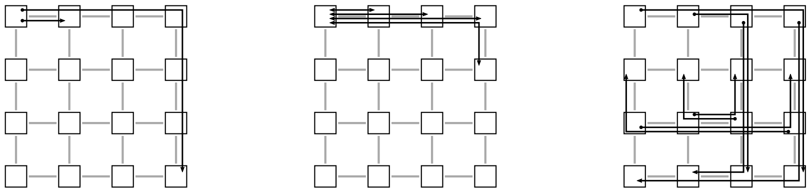
Once the slot raises an interrupt, the virtual interface updates the `READY` flag in its configuration register. Finally, it raises an interrupt itself if configured to do so.

4.2.3.4 HW-OSQM Component

The operating system queue manipulation is a separate module. It is only instantiated if the feature is enabled. It can be triggered via two interfaces as depicted in Figure 4.8a: The initiator can trigger an OSQM operation when a transfer is completed and the target can trigger an OSQM operation when it delivered a message to an endpoint with a waiting thread.

The OSQM module is configured by the operating system if dynamic configuration is supported. A state machine is triggered by the requests from initiator and target. It then performs the OSQM operations on addresses calculated from the trigger data (thread and queue) and the offsets.

As described before, the offsets can be either configured statically ("in hardware") or dynamically during start-up of the operating system. Figure 4.8b compares the FPGA resource utilization of the OSQM module for three configuration: The default configuration is the dynamic setup which supports both double linked lists and single linked lists, along with arbitrary offsets in the data structures. When configured statically the resource utilization is lower: For a single linked list the combinational LUTs drop by 22% and the registers drop by 32%. For a double linked list the LUTs are only reduced by 5% and the saved configuration registers equally reduce the register count by 32%. Anyhow,



(a) Simple 1-to-1 traffic **(b)** 1-to-N and N-to-1 traffic **(c)** Advanced traffic scenario

Figure 4.9: Traffic scenarios

these savings are more relevant when the OSQM concept is used in other contexts. In the following I will use it in the default configuration, because of the flexibility.

4.3 Synthetic Benchmarks

The basic NAMP capability is in the following validated with synthetic benchmarks of the prototype implementation. The validation is based on the simulation of the RTL model of an entire platform. The platform is a platform of 16 tiles organized in 4 rows and 4 columns. Each tile has one processor. The benchmarks are synthetic, meaning that they are not realistic applications but are designed to trigger the important corner cases. They allow to observe the performance metrics of the NAMP isolated from other effects. The RTL model is 100% cycle accurate. As all communication is based on data in the tile local memory, the timing is identical to the behavior of a timing accurate model, FPGA or ASIC (assuming the timing of the SRAM is one clock cycle). The full visibility of the system state allows to measure exact timings non-intrusively.

4.3.1 Basic connection-less message passing

In a first validation, the basic connection-less message passing is evaluated. In this example one sender basically sends data to one receiver continuously.

Figure 4.10 shows the results for this synthetic benchmark for different scenarios. In the most simple scenario there is one sender that continuously sends messages to one of its next neighbors (see Figure 4.9a). In the second scenario the data transfer traverses the maximum distance in the network-on-chip of six hops (also in Figure 4.9a). A first observation is that the send overhead is constant and independent of the traffic scenario. This is because it only is the device configuration and matches the evaluation as depicted in Figure 3.13. The static offset in this simulation compared to the previous evaluation is due to processor caching effects that were not part of the discrete event simulation model.

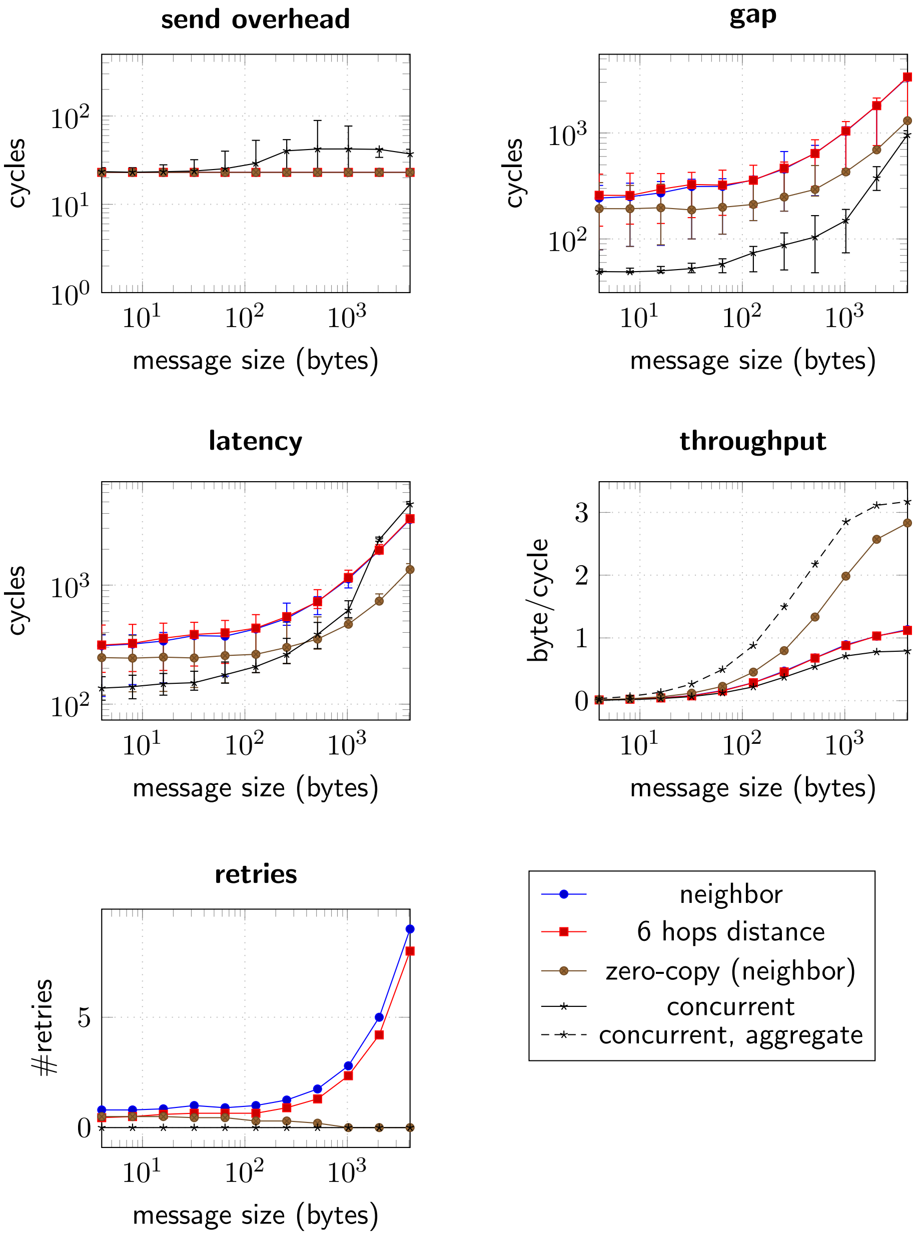


Figure 4.10: RTL simulation benchmark results for connection-less message passing with NAMP support

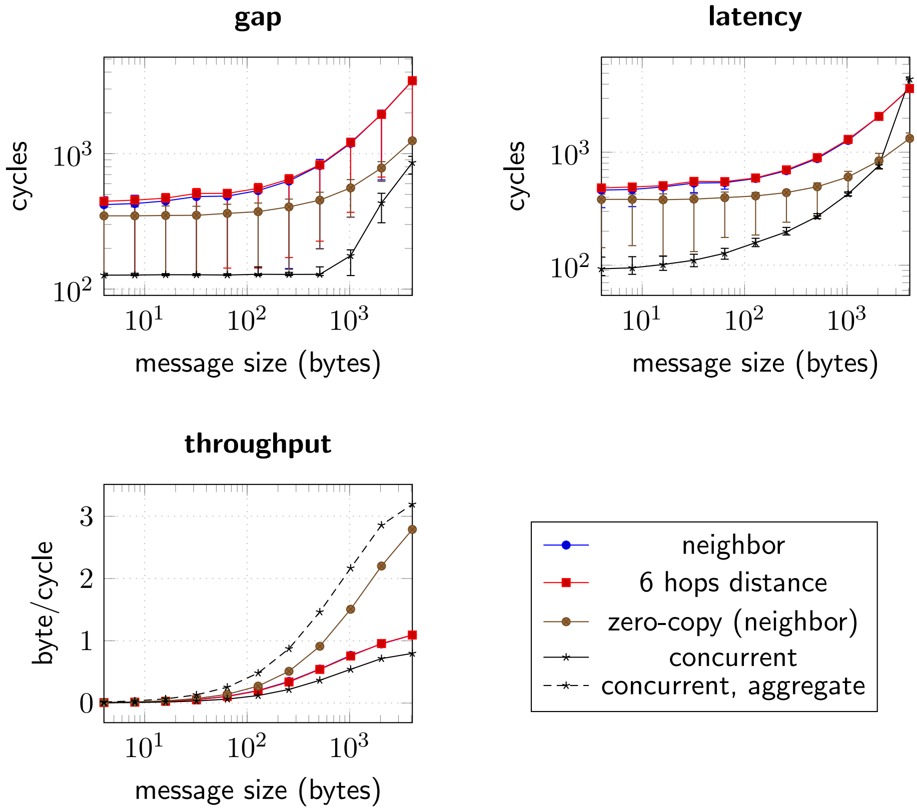


Figure 4.11: RTL simulation benchmark results for connection-oriented message passing with NAMP support

It can be observed that the latency matches the trend from the previous evaluation accurately. Also, there is no significant difference between the communication with the neighboring tile and the tile at maximum distance, due to the fact that the traversal time of the NoC adds three times to the latency: one round trip for buffer allocation and then once for the message transfer. Otherwise, the latency is dominated by the actual buffer allocation operations for smaller message sizes and by the message size for larger messages.

Additionally, the gap is depicted in Figure 4.10. As introduced before this is the duration of protocol handling in hardware that one message transfer takes. Also here, the network distance only has minor impact. The figure furthermore shows the average effective throughput from the application. There is one major reason why the gap and thereby the throughput do not benefit much from increasing message sizes: The receiver is slower with processing the data, because it copies the data from the buffer when it

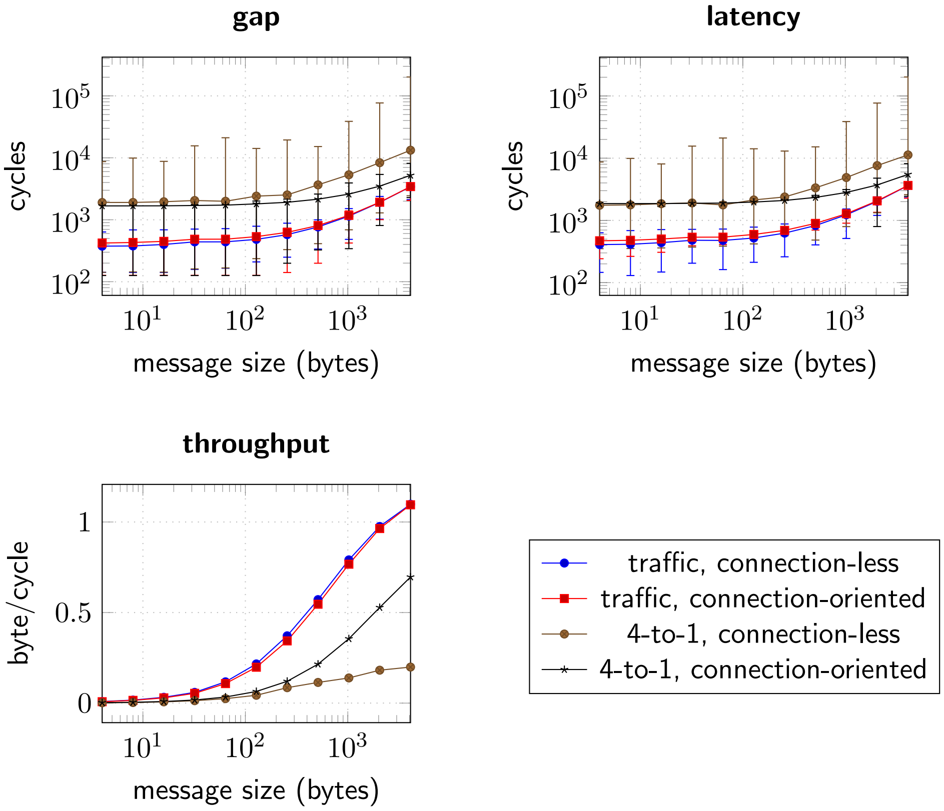


Figure 4.12: Simulation benchmark results for advanced traffic scenarios

receives it. This amplifies with increasing message sizes. The plot also shows the average number of retries by each transfer which significantly increases.

A third experiment in Figure 4.10 uses the functionality to receiving a message from the buffer by a pointer into the buffer instead of copying (zero-copy). For this it can be observed that the number of retries is similar to before for smaller message sizes, but then lowers down to zero for large message sizes. Thereby latency, gap and throughput improve significantly and can be seen as practical boundaries.

Anyhow, there is a last component that remains for connection-less messages: The protocol handling in hardware actually is not one long operation as the gap may suggest. As the message sequence chart in Figure 3.12 suggests, there are actually many delays by the waiting for the allocation response and the DMA transfer completion. Especially the former can effectively be used by other transfers. Hence, the fourth experiment in Figure 4.10 shows the performance metrics for a scenario where the sender sends to four receivers as sketched in Figure 4.9b. The message transfers are handled in parallel, so that the different progresses in the protocol handling overlap. For each individual message the throughput is smaller than before, because of the serialization of the protocol handlings in the NAMP. Anyhow, the aggregate of all four transfers improves significantly over the previous scenarios, up to around 75% of the theoretical limit of 4 bytes/cycle.

4.3.2 Basic connection-oriented message passing

In a second set of experiments the same experiments have been executed for connection-oriented message passing. The overhead is still constant and therefore omitted here and in the following.

Figure 4.11 shows the results for the simulations. It can be observed that connection-oriented behaves similar to connection-oriented communication. But again, in the case of this synthetic benchmark the receiver dominates the message transfer. In the case of the connection-oriented communication this is even worse because the credit update after the reception of the message is in the critical path, and hence defines a general bottom to the gap.

4.3.3 Advanced traffic scenarios

As mentioned before, the absence of any computation and concurrent traffic is an assumption for a synthetic benchmark. Hence, Figure 4.12 shows experiments for traffic scenarios that add contention at the network level and the buffer level. The advanced traffic scenario as depicted in Figure 4.9a has multiple parallel communication streams crossing the chip and leading to contentions on the network. Some streams may conflict with each other and the sensitivity range in Figure 4.12 gives a good indication of the order of magnitude that gap and latency can differ.

The variation in gap and latency vary even more when the contention is at the destination buffer. In the last two experiments four senders send messages to one destination as depicted in Figure 4.9b. Gap and latency are largely dominated with the ability of the receiver to cope with the messages. Anyhow, it can be seen that for connection-less communication the effects of concurrent arbitration and the round trips for that limit the individual throughput significantly.

	SLOTS_PHYS	SLOTS_VIRT	ENABLE_MESSAGE	ENABLE_CHANNEL	ENABLE_BUF	ENABLE_CCM	ENABLE_CC_MC	ENABLE_CC_SG	ENABLE_CC_RED	ENABLE_OSQM	ENABLE_PS
baremsg	✳	0	✓								
barechan	✳	0		✓	✳	✳					
bare	✳	0	✓	✓	✓	✓	✓	✳	✳		✳
svbase	1	✳	✓	✓	✓	✓	✓				
svfull	1	4	✓	✓	✓	✓	✓	✳	✳	✳	✳

Table 4.2: Typical NAMP configurations. Configuration items marked with ✳ are parameters in the evaluation to cover all sensible design point.

4.4 Typical Configurations and Hardware Utilization

Now that the implementation of the NAMP and its components is described, the resource utilization of the NAMP is of interest. As mentioned before, this resource utilization is part of a trade-off: While the NAMP can deliver significant improvements for the software, it adds extra hardware area.

Instead of doing a full sweep of all configuration parameters or measuring the components and features individually, I have derived a set of *typical configurations*. Those configurations replicate sensible usage scenarios. Table 4.2 contains this list of five typical configuration. The table shows the enabled features and parameter values for each configuration. The configurations are themselves not fixed, but some settings are explored for each of the configurations (marked as ✳ in the table cells). The synthesis results presented in the following hence cover a broad range of sensible configurations and it is easy to extrapolate estimates of other, less typical configurations from the presented results.

In the following I discuss each of the configuration and explain the effect of the configuration parameters from Table 4.2 accordingly. Both the NoC Buffers and the DMA controller are mostly excluded from the synthesis results. They are assumed to be the baseline that resembles the state of the art and hence present anyways. Those two modules are large, because they include a variety of buffers, that are still mapped into registers. But even SRAM wouldn't reduce the size significantly. The NoC buffers sum up to approximately 37 kGates in the ASIC technology and the standard DMA controller (2 slots) has approximately 27 kGates. Those are also a good reference when considering the extra hardware added by NAMP.

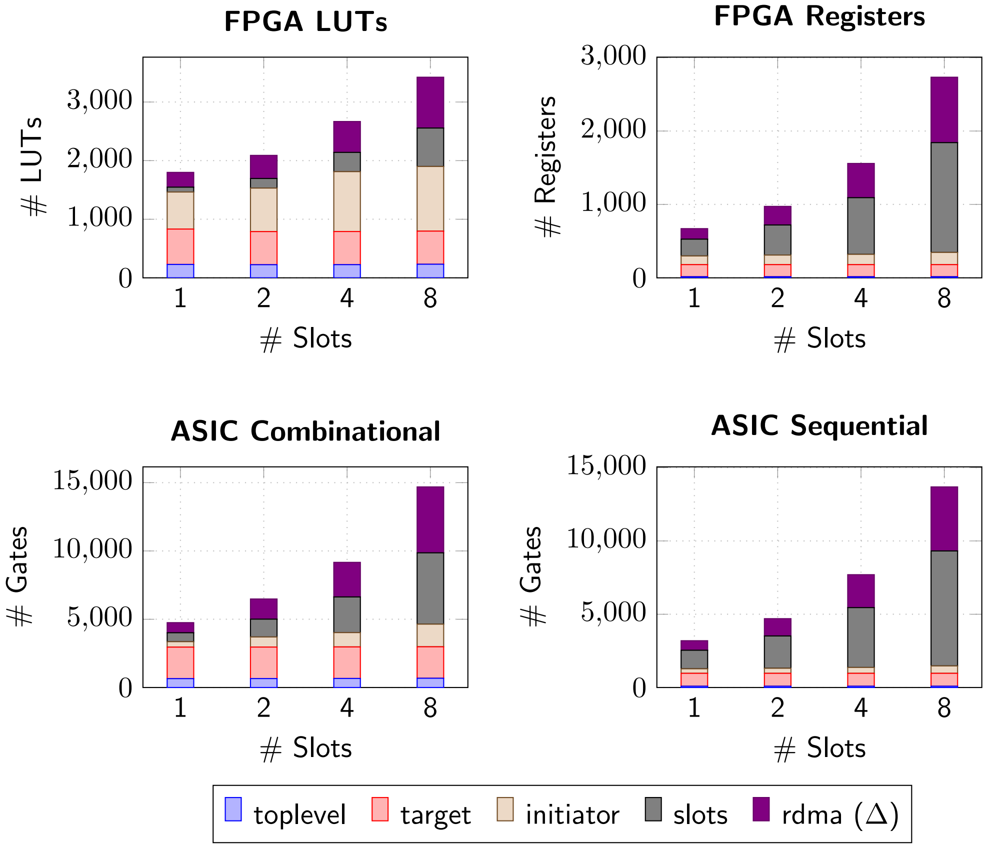


Figure 4.13: Implementation details of NAMP `baremsg` configurations. The variants are different feature sets.

4.4.1 `baremsg` Configuration

The NAMP `baremsg` implementation is a simple bare-metal configuration. It is hence limited to physically addressed slots and the impact of the number of slots is evaluated. It only allows for connection-less communication, meaning the protocol handling and state for allocation handling are included. It also does not integrate support for message sending from buffers, and none of the advanced features. Its usage scenario is in simple tiled many-core system-on-chip platforms where only connection-less messages are needed.

Figure 4.13 shows the resource utilization of the `baremsg` NAMP configuration with the number of slots being the evaluated parameter. Unsurprisingly, the resource utilization increases with the number of slots. The main contributor to the increase are apparently the slots themselves (sum of all slots) and the extra NAMP resources in the DMA which are slots too. Finally, there is a slight increase with the slots in the initiator component due to the arbitration and multiplexing. The toplevel is the sum of all glue logic. It also scales with the number slots but only in the range of just a few gates. Finally, the target module is of constant utilization because it does not depend on the slots at all.

FPGA and ASIC synthesis generally show a similar trend and are more or less equal for the registers and sequential gates respectively, which is not very surprising. The relative increase in combinational resources is smaller with the FPGA synthesis because the packing of logic into the six input lookup tables gets much better with increasing slots.

4.4.2 `barechan` Configuration

The NAMP `barechan` configuration is similar as the `baremsg` targeted at bare-metal use. Contrary to the latter it only supports connection-oriented communication. The implementation thus includes the credit handling and leaves out the message allocation. Furthermore, as depicted in Table 4.2 I evaluate the impact of adding buffer-based sending. As an additional future the integration of the continuous communication mode (`ENABLE_CCM`) is evaluated. This NAMP configuration is a good candidate for example for platforms that only support a data flow programming model (as Kahn Process Networks or similar).

Figure 4.14 plots the resulting resource utilization for the `barechan` configuration. The synthesis results are shown for instances with one slot and with four slots. For each, three variants are shown: The `default` variant is the fully described NAMP. The `noccm` variant does not have support for the continuous communication mode. Finally, the `nobuf` variant does not have support to send from a buffer at all.

It can be seen that the FPGA and ASIC synthesis are more or less equal. The `barechan` NAMP without support for buffers (`nobuf`) is roughly equally large as the `baremsg` variant, but more distributed over the components. The difference by adding buffer support (`noccm`) is around 15% in gates. Adding support for the continuous communication mode (`default`) finally adds another 4% to the gates.

4.4.3 `bare` Configuration

The NAMP `bare` configuration is intended to be the full blown bare-metal variant. It includes both connection-oriented and connection-less communication along with multicast support. Further collective communication support and support for protection switching are configurable.

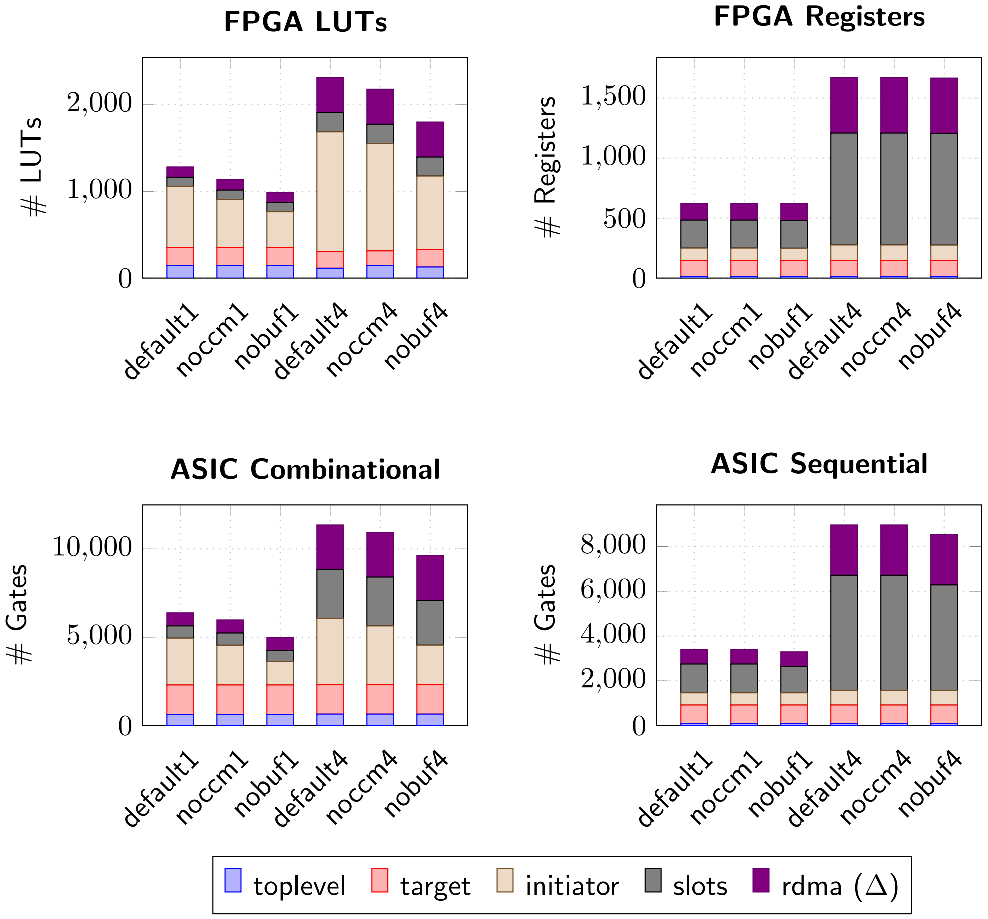


Figure 4.14: Implementation details of NAMP `barechan` configurations. The variants are different feature sets.

Figure 4.15 shows the synthesis results for different variants and a fixed slot number of four.

4.4.4 `svbase` Configuration

The `svbase` configuration is the first one that supports self-virtualization. As listed in Table 4.2 it has one slot that is only addressable in the physical address space. All other slots are configurable as virtual interfaces. This NAMP configuration supports multicast, but not the other collective communication operations.

Figure 4.16 shows the synthesis results for `svbase` for different number of slots. The virtual interfaces now increment the number of gates. Compared to the slot itself a virtual interface is smaller. It adds only 8% gates to the slot.

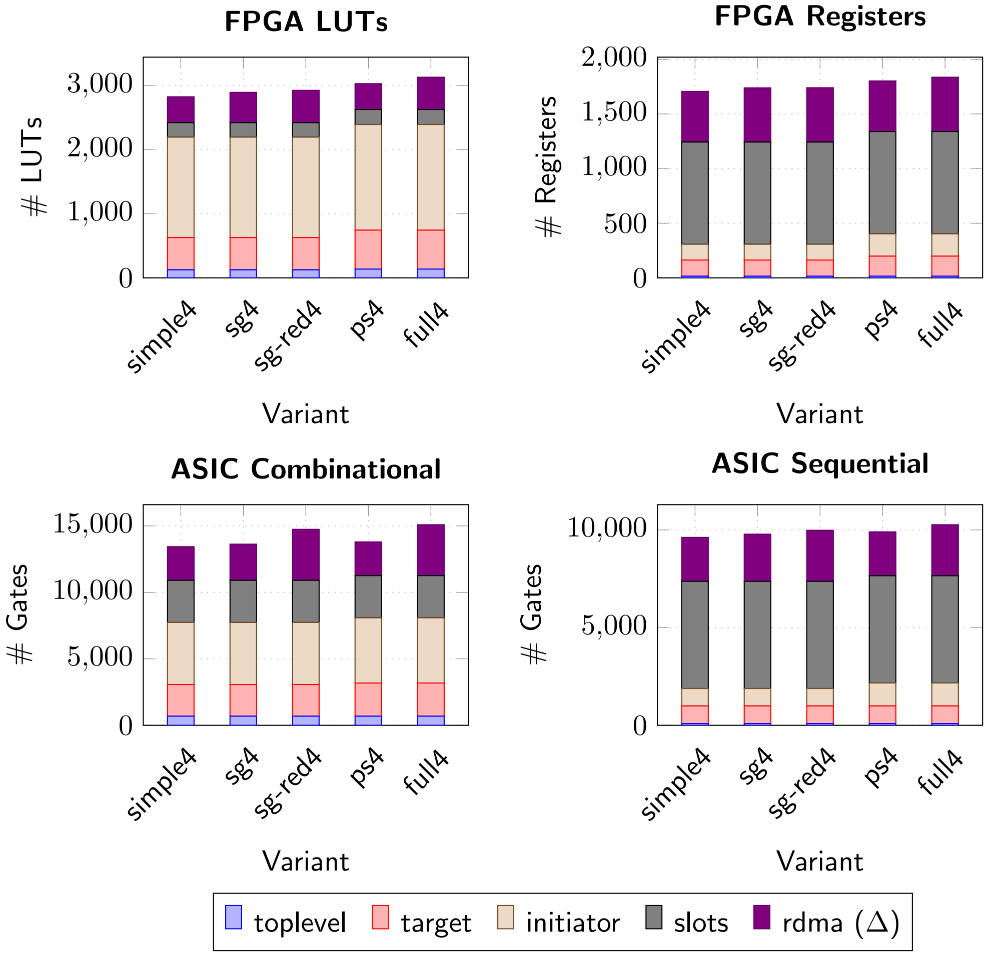


Figure 4.15: Implementation details of NAMP `bare` configurations. The variants are different feature sets.

4.4.5 `svfull` Configuration

Finally, the `svfull` configuration includes the missing features into the `svbase` configuration. It delivers all performace gains described throughout this thesis.

Figure 4.17 shows the synthesis results for the `svfull` module. The full NAMP implementation uses around 40 kGates in a 40nm ASIC.

4.5 Example Systems

Before concluding the discussion of the prototype implementation, I present two example systems that target two different usage scenarios and FPGA boards.

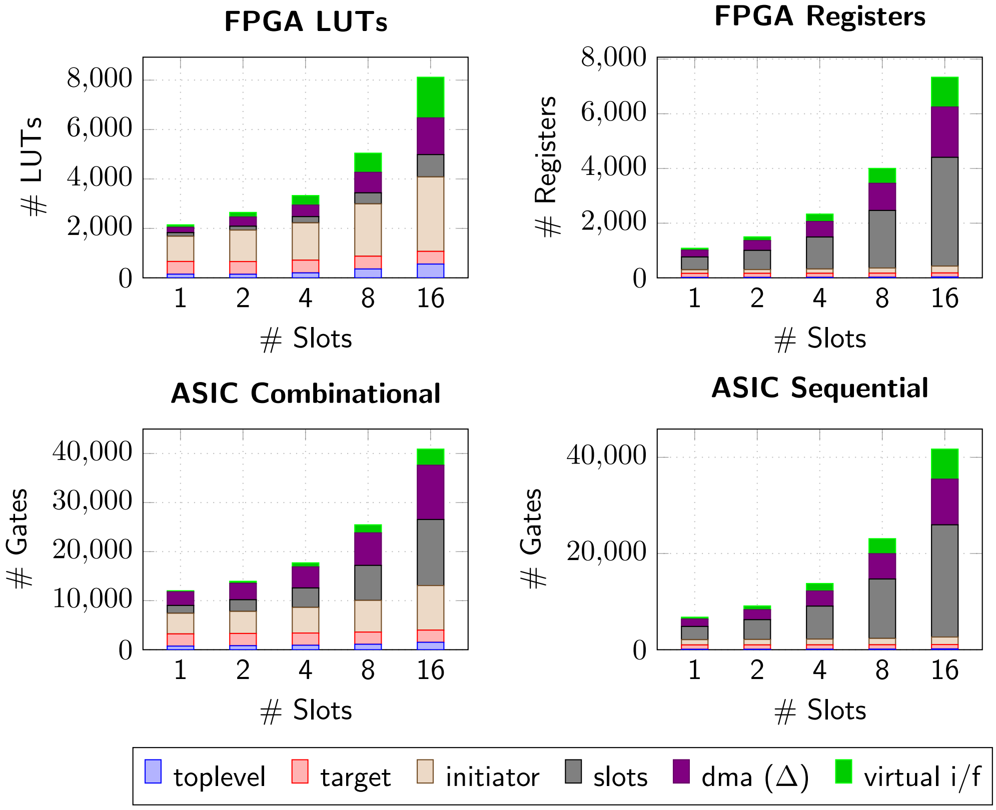


Figure 4.16: Implementation details of NAMP `svbase` configuration. Different number of slots are evaluated.

4.5.1 Small baremetal platform

A small system of four tiles with each one processor core is generated to target the Digilent Nexys4 DDR board [61]. It is a simple board with a small Xilinx Artix 7 FPGA. The NAMP is configured as follows:

- Base feature set only. No NAMP-CC, NAMP-SV, NAMP-PS and HW-OSQM.
- Connection-oriented communication only.
- Support for the continuous communication mode.
- Two NAMP slots.

This configuration can be useful for mapping KPNs to baremetal software. The software then just directly interfaces the buffer in the tile local memory.

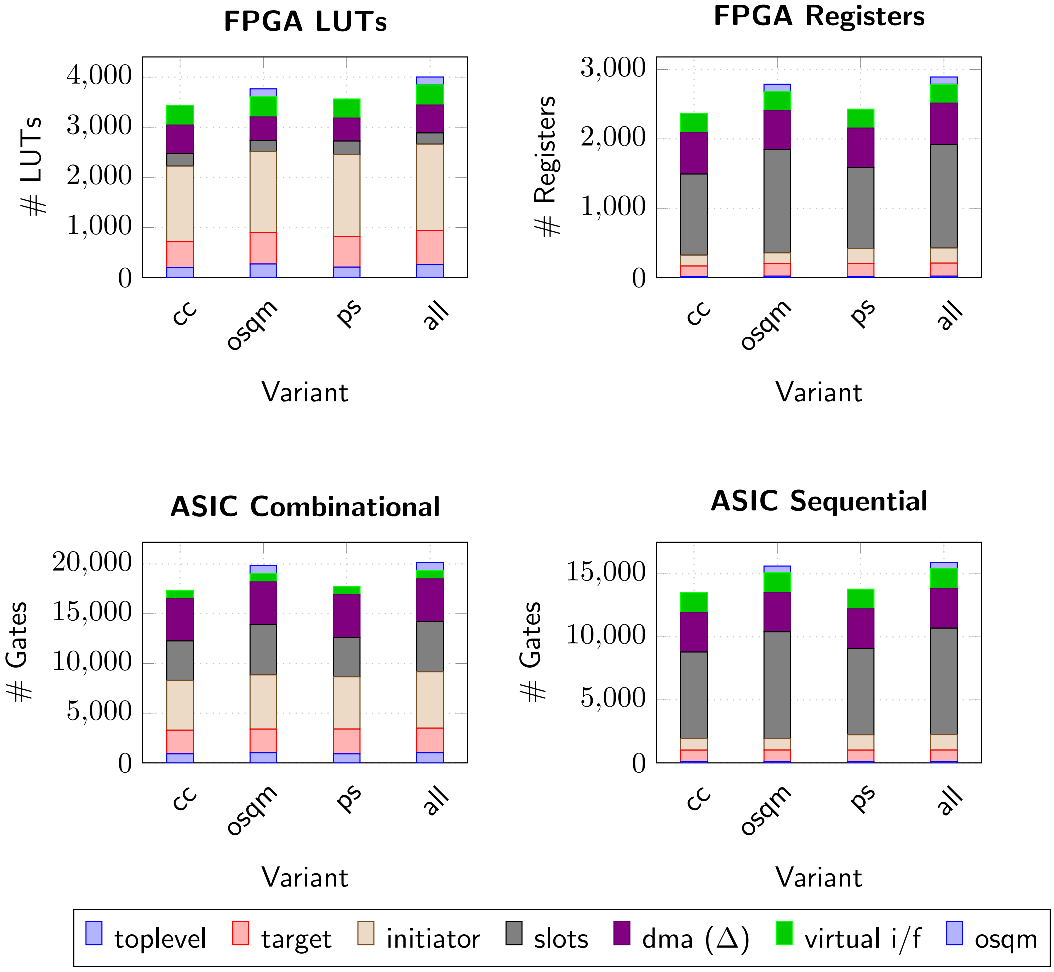


Figure 4.17: Implementation details of NAMP `svfull` configuration. Different configurations are evaluated.

The layout of the implemented design can be found in Figure 4.18. The four tiles are colored differently and the network-on-chip is also highlighted. The base system utilizes many resources because it uses DDR memory to emulate the tile local memory due to the very limited number of block SRAM in the device. For one tile the individual interesting components are highlighted. It can be seen that the core utilizes a relatively small part of the tile. The network adapter instead is relatively large, but the NAMP itself is only a medium sized part of it. “Other tile” logic is mostly the debug infrastructure.

For a better analysis of the implemented design, Table 4.3 summarizes the resource utilization of the most relevant modules in a hierarchical manner. It can be seen that the network adapter is significantly larger than the processor core. The role of the NAMP

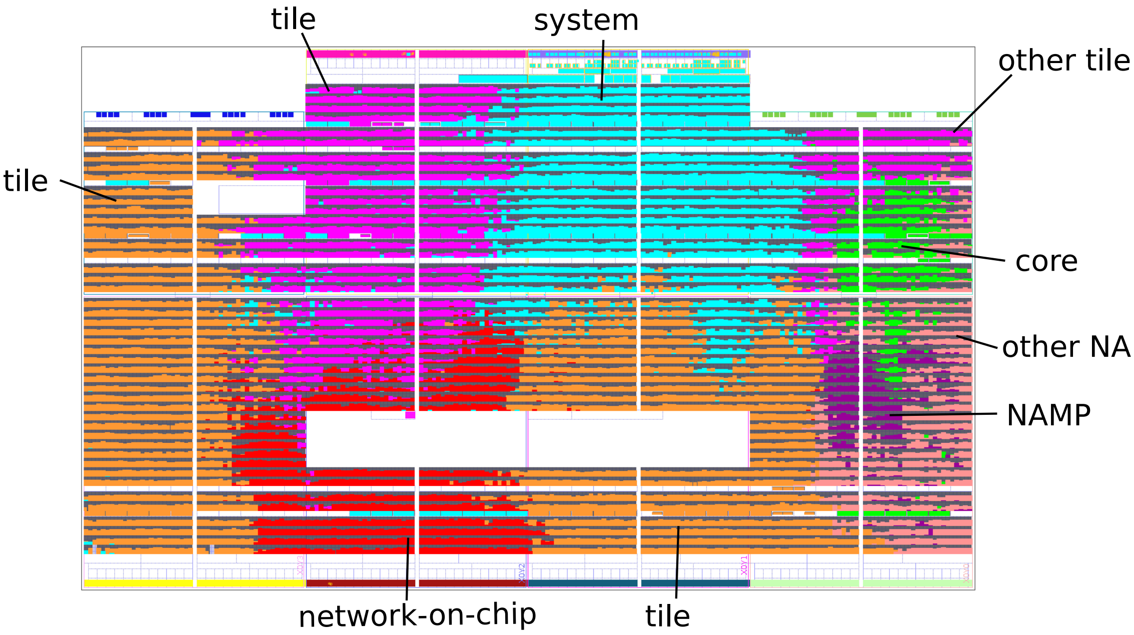


Figure 4.18: FPGA Layout of small baremetal platform

Module	**LUT**	**Registers**
System	85.5% of device	48% of device
NoC	8.4% of system	14.4% of system
Tile	each 18.7% of system	each 16.9% of system
Core	29.5% of tile	18.1% of tile
NA	44.5% of tile	59.8% of tile
NAMP	31.3% of NA	12.3% of NA
Buffers	20.2% of NA	36.5% of NA
DMA	42.2% of NA	48.6% of NA

Table 4.3: Resource utilization of small baremetal platform

Module	LUT	Registers
System	92% of device	41.5% of device
NoC	5.5% of system	10.4% of system
Tile	each 5.7% of system	each 5.4% of system
Core	each 11.4% of tile	each 8.8% of tile
NA	22.7% of tile	31.2% of tile
NAMP	44.1% of NA	19.7% of NA
Buffers	14.1% of NA	29.9% of NA
DMA	36.9% of NA	48.3% of NA

Table 4.4: Resource utilization of large platform

inside the network adapter is that it contributes much more to the look up tables than to the registers. The majority of resources are utilized by the network-on-chip buffers and the DMA controller. This is mostly due to the large number of buffered network packets. This is a countermeasure against congestion in the network: Packets are generally only transmitted one they are complete. This is an important property to decouple the computation domain from the communication domain.

It has to be noted that the deployed processor core in OpTiMSoC is a very simple, 32-bit processor core without much optimization. A more sophisticated core that yields better single thread performance could change the relations drastically.

4.5.2 Large platform with full NAMP

A second, much larger platform has been implemented on one of the largest FPGA evaluation kits of Xilinx, the VCU108 board [224]. The system consists of 64 cores organized in 16 tiles of each four cores. The NAMP is configured as follows:

- All features are supported (svfull).
- There are four NAMP slots, each with virtual interface support.

This kind of platform is intended to demonstrate the usage of NAMP in a realistic scenario with a complex software stack.

Figure 4.19 visualizes the device resource utilization on the FPGA. The tiles are colored in three different colors and the network-on-chip is also highlighted. For better readability one tile is magnified. It can be seen that the relative size of the NAMP in the tile is moderate, but its relative share in the network adapter is much higher.

Table 4.4 shows the relative resource utilization in the large system. The network adapter occupies around 20% of the LUTs and 30% of the registers in the tile. The

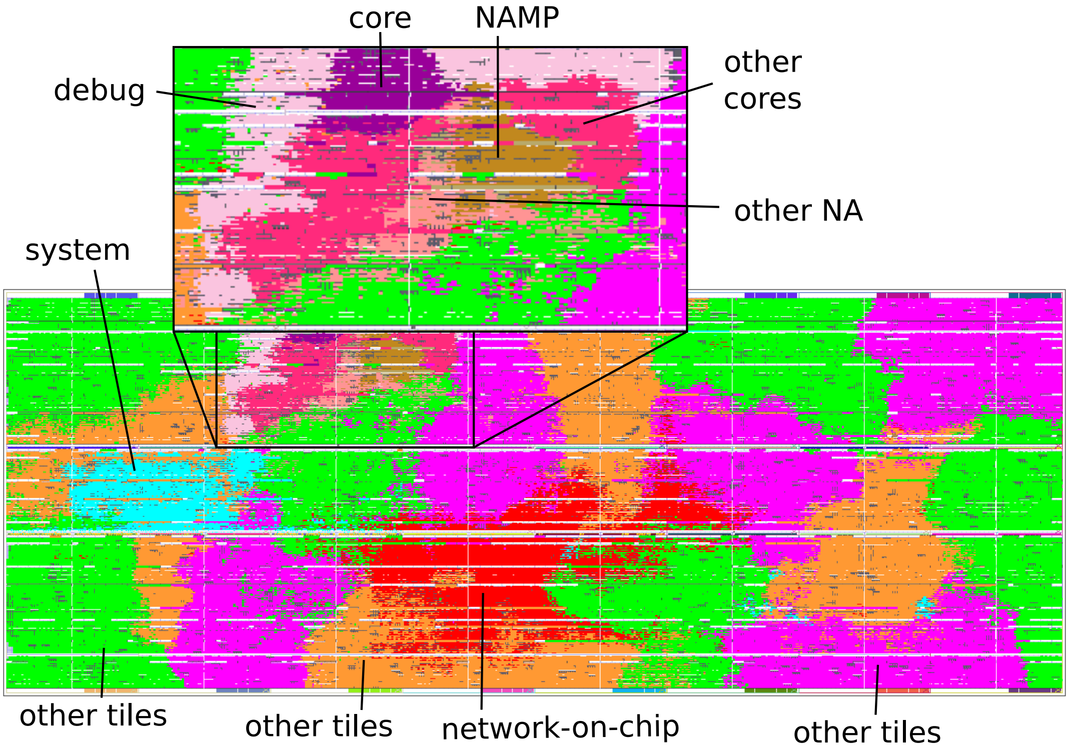


Figure 4.19: FPGA Layout of large platform (64 cores in 16 tiles)

NAMP is the largest module in the network adapter with respect to LUTs for aforementioned reasons. The DMA and the buffers of it are the largest occupant of registers in the design.

Those two systems gave an impression about the overall system layout and the impact that NAMP has on it. Despite the fact taht the prototype implementation can be further improved, it should be highlighted again that the performance gains are significant if the NAMP is used.

4.6 Summary

In this chapter I have given insights into the implementation of NAMP. Based on the described concept, the NAMP has been implemented and validated inside a prototype tiled many-core system-on-chip platform. The results have shown that the extra gates required by the NAMP-specific features are in the same order as the other two modules DMA and NoC buffers. This makes NAMP a promising network adapter that delivers significant performance gains at moderate area overhead.

5 Conclusion & Outlook

In this thesis I presented contributions to the inter-task communication in many-core system-on-chip. The existing state-of-the-art network adapters for tiled many-core system-on-chip platforms focus on the network-on-chip interface and bare-metal programs. The work presented here specifically contributed a network adapter whose functional enhancements have been derived from a more holistic view of the entire software stack. In the following I will summarize and conclude this thesis, and close with an outlook.

5.1 Conclusion

The work in this thesis was motivated by the expectation that the software running on processors in tiled many-core system-on-chip platforms will in the future be more complex than the bare-metal programming that is dominant nowadays. Having a look at the software stack of such platforms a lot of overhead is in the interfaces and certain functionalities. Table 5.1 summarizes the most critical aspects, describes the state-of-the-art and relates that to the future demand. The table furthermore references the contributions of this thesis that covers all of the aspects in a number of features in the proposed network adapter. This *Network Adapter for Message Passing (NAMP)* has been conceptually developed and implemented as presented in this thesis.

Message Passing Support As summarized in Table 5.1 the existing network adapters in academic proposals and commercial products focus mostly on the network-on-chip interface of the network adapter. Some approaches exist that provide message passing support, but they are limited in their capabilities to operate in different communication modes and are narrowed to a specific use case. The fundamental feature of NAMP is the offload of the progress engine for message passing communication in a variety of modes, such as connection-less vs. connection-oriented, different buffering schemes and the continuous communication mode.

	State-of-the-Art	**Future Demand**	**Contribution**
Message passing support	Network-on-chip focus, few (limited) bare-metal message passing	Hardware acceleration, flexible communication modes	Basic NAMP functionality (Section 3.2)
Workload sharing	Limited support for hardware-assisted multicast, software else	Hardware support for collective communications	NAMP-CC (Section 3.3)
Task isolation & resource sharing	Bare-metal software or operating system multiplexes devices	Isolation by operating system and device virtualization	NAMP-SV (Section 3.4)
Event signaling	Polling & interrupts, interrupt coalescing	non-intrusive thread wake-up	HW-OSQM (Section 3.5)
Communication migration	Pausing of communication during task migration	Minimal impact of migration on communication	NAMP-PS (Section 3.6)

Table 5.1: Expected trends and critical functions: coverage by thesis contributions

The baseline NAMP has been analytically compared to RMA-based and RDMA-based message passing implementations which dominate the state-of-the-art. The analysis has shown that the software overhead of the message passing is significantly reduced and becomes static independent from the message size. In my event-based simulation model at least 81% of clock cycles can be saved compared the RDMA-based implementation. Synthetic benchmarks have been used to validate the base NAMP implementation in a full-system cycle-accurate RTL simulation.

The baseline message passing forms the central parts of the NAMP implementation. Variants with the NAMP as a baremetal device have been synthesized to approximately 24 kGates in a 40nm ASIC technology. So called slots store the transfer configuration and its state and many slots can be instantiated in a NAMP. Each slots adds around 4 kGates. Finally, a configuration that only supports connection-oriented communication saves 16% and one that only support connection-less communication saves 34% compared to the baseline.

Collective Communication Support The support for workload sharing in the state-of-the-art network adapters is limited to a few that support multicasts. As highlighted in Figure 5.1 the NAMP progress engine offload can support the four basic collective communication operations: multicast, scatter, gather and reduction. This support is for one the extended progress engine engine of multiple concurrent operation. Beside that, the RDMA controller has been extended to support scatter and gather, and reduction operations have been added in the data path of the RDMA.

The impact of hardware-based collective communication has been analytically compared to full software-based implementations. The overhead at the sender is limited to a one-time configuration that is independent from the number of nodes in a collective communication. By that, a multicast operation to eight destinations for example faces 64% less overhead. Beside that, the concurrent behavior leads to a throughput increment by 72% for 4 byte messages and 4% for 2048 byte messages.

Self-Virtualized Network Adapter As compared in Table 5.1 the support for resource sharing of a network adapter for isolated tasks is limited to multiplexing by the operating system for existing network adapters. This means that tasks must use an operating system driver to map virtual addresses to physical addresses and ensure the safe sharing of the device. The NAMP self-virtualization (NAMP-SV) feature gives tasks the impression of exclusive access to a network adapter without interaction with the operating system. The virtual interface translates virtual endpoint identifiers to physical addresses on the fly and can thus be mapped into the virtual address space of a task. With multiple virtual interfaces several tasks hence in parallel have the view of their "own" network adapter.

The NAMP-SV feature improves the setup of the transfer at the sender, because it eliminates the path via the operating system. The hardware overhead of NAMP-SV is limited to the virtual interfaces. The virtual interfaces only consist of the configuration registers and a state machine, all other data structures are stored in memory, an essential feature of NAMP in general. Each virtual interface for adds example x kGates in an ASIC design.

Hardware Operating System Queue Manipulation One of the main contributors to the overhead especially for small messages is the event signaling. As summarized in Table 5.1 the two dominant methods for event signaling are polling and interrupts. With increased number of waiting threads or increased event rate the impact of event signaling on the performance can become significant. In the context of the NAMP the effect of an event is that a thread is resumed that was either blocking on a send or waiting for a message. Hence, the essential idea of HW-OSQM is to perform that task instead of raising an interrupt and interfering with the software execution for that task. The HW-OSQM is thus configured to re-insert a thread element back into an operating system's ready queue.

Analysis has shown that this technique can easily save 15% for event rates around 20,000 per second, which corresponds to a throughput of 5 MB/s for an average packet size of 256 kB. The saved overhead increases linearly with the event rate.

The OSQM is a separate module in the NAMP and triggered by both the initiator and the target part. In its default configuration the OSQM synthesizes to 1.4 kGates. By static configuration to one operating sytem (or a few) 32% can be saved, but with losing the flexibility of the configuring the HW-OSQM at runtime to the data structure layout of the operating system.

Communication Migration Task migration is an increasingly important feature in tiled many-core system-on-chip architectures. This can be a functional task to reduce the dimensions of an application. But it is even more critical in dependability management of the platform. A task migration may for example be triggered to mitigate thermal hotspots. As noted in Table 5.1 the communication channels have to be migrated together with the task. This communication migration in the state-of-the-art generally means to pause the communication, which can lead to performance degradation during the migration. The analysis has shown that the proposed protection switching techniques (NAMP-PS) can lead to a reduction of 12% in extra latency. The NAMP-PS feature synthesizes to an extra x kGates.

Two example systems have demonstrated the usage of NAMP in the Open Tiled Many-core System-on-Chip for different scenarios. A small and simple version targeted on baremetal applications with connection-oriented communication has been shown for a small system of four cores in four tiles. A large system with 64 cores in 16 tiles has been used to demonstrate the use case of a larger version of the NAMP with a rich feature set.

This overview summarized the contributions of this thesis and the related improvements over the state-of-the-art. To the best of my knowledge, work presented in this thesis was the first to comprehensively analyze on-chip message passing with a full system stack and derive the concept for a message passing network adapter. The contributed NAMP concept is a configurable, scalable concept for a network adapter that enables efficient inter-tile communication in tiled many-core system-on-chip.

5.2 Outlook

While the work in this thesis contributed a concept for a network adapter that reduces overhead significantly, there are certainly areas for further investigations to improve the presented NAMP.

The presented work did not cover functions related to management of endpoints and connection channels, because those events are rare and thus the message transfers are a better candidate for improvements. Only the migration of the communication channels has been integrated into the NAMP concept. Anyhow, more dynamicity in the starting and termination of tasks can be anticipated. The setup and management of the communication relations can have a significant impact on the performance, so that the NAMP could also incorporate support for those operations. This allows for fast setup of tasks and thereby more dynamic behavior.

The number of reduction operations is currently limited to a subset of MPI operations. A thorough investigation of other operations that are more specific to the use case could lead to other reduction operations worth adding. A micro-programmable reduction unit could allow for custom reduction operations.

Another potential improvement to the NAMP concept is the offloading of the task management and scheduling entirely to the NAMP. Together with the aforementioned support for communication management functions, this can allow the hardware platform to provide an interface for spawning many short-term stateless tasks. Such "microservices" could be invoked by sending messages to a reserved communication channel. The message on this channel would then trigger the start and lifecycle of the task.

Regarding the implementation of the NAMP there are three major fields of investigation. First of all it is a prototype implementation and an analysis of the resource utilization has shown that the requried area can be further reduced by optimizing the mapping of temporary registers in the finite state machines so that the sharing of resources like adders can be optimized. Beside that, there currently is a one-to-one relation between NAMP slots and RDMA slots. With further elaboration of the load and average utilization of RDMA slots, a smaller number of RDMA slots could be shared. This would reduce the required area accordingly.

Another potential improvement to the implementation is the elimination of registers by SRAM. While each slot or virtual interface for itself does not include enough registers to benefit from SRAM blocks instead, the sum of all registers certainly does. The problem that arises is that the SRAM interface differs from the current random access to registers in the NAMP. Anyhow, the tradeoff between the access complexity and area improvement could be investigated.

Bibliography

[1] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP Journal on Embedded Systems*, 2008(1), November 2007. ISSN 1687-3963. doi: 10.1155/2008/518904. URL `http://jes.eurasipjournals.com/content/2008/1/518904/abstract`.

[2] Adapteva. Epiphany Architecture Reference, 2013. URL `http://adapteva.com/docs/epiphany_arch_ref.pdf`.

[3] Anant Agarwal. The tile processor: A 64-core multicore for embedded processing. In *Proceedings of HPEC Workshop*, 2007.

[4] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. Loggp: incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM, 1995.

[5] George Almási, Charles Archer, José G Castanos, John A Gunnels, C Christopher Erway, Philip Heidelberger, Xavier Martorell, José E Moreira, Kurt Pinnow, Joe Ratterman, et al. Design and implementation of message-passing services for the blue gene/l supercomputer. *IBM Journal of Research and Development*, 49(2.3): 393–406, 2005.

[6] Amazon FreeRTOS authors. Amazon FreeRTOS Website. `https://aws.amazon.com/de/freertos/`, 2017.

[7] 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*, pages 483–485. ACM, 1967.

[8] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[9] Marco Annaratone, Emmanuel Arnould, Thomas Gross, HT Kung, Monica Lam, Onat Menzilcioglu, and Jon A Webb. The warp computer: Architecture, implementation, and performance. *IEEE transactions on Computers*, 100(12):1523–1538, 1987.

[10] ARM PrimeCell DMA Controller (PL080). http://www.arm.com, 2011.

[11] Arteris. A comparison of network-on-chip and busses. *white paper*, 2005.

[12] InfiniBand Trade Association et al. The infiniband architecture specification. *http://www. infinibandta. org/specs/*, 2000.

[13] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE transactions on software engineering*, 18(3):190–205, 1992.

[14] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, et al. Openpiton: An open source manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 217–232. ACM, 2016.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[16] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 282–293. ACM, 2000.

[17] Sanjoy K Baruah. Task partitioning upon heterogeneous multiprocessor platforms. In *IEEE real-time and embedded technology and applications symposium*, pages 536–543, 2004.

[18] Daniel Bates, Alex Bradbury, Andreas Koltes, and Robert Mullins. Spatial computation on a homogeneous, many-core architecture. *Proceedings of the Second International Workshop on Parallelism in Mobile Platforms*, 2014.

[19] Daniel Bates, Alex Bradbury, Andreas Koltes, and Robert Mullins. Exploiting tightly-coupled cores. *Journal of Signal Processing Systems*, 80(1):103–120, 2015.

[20] Daniel Bates, Alex Chadwick, and Robert Mullins. Configurable memory systems for embedded many-core processors. *arXiv preprint arXiv:1601.00894*, 2016.

[21] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, Jan 2002. ISSN 0018-9162. doi: 10.1109/2.976921.

[22] Luca Benini and Davide Bertozzi. Network-on-chip architectures and design methods. *IEE Proceedings-Computers and Digital Techniques*, 152(2):261–272, 2005.

[23] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 983–987, 2012.

[24] John K Bennett, John B Carter, and Willy Zwaenepoel. *Munin: Distributed shared memory based on type-specific memory coherence*, volume 25. ACM, 1990.

[25] Brian N Bershad, Matthew J Zekauskas, and Wayne A Sawdon. *The Midway distributed shared memory system.* IEEE, 1993.

[26] Davide Bertozzi and Luca Benini. Xpipes: A network-on-chip architecture for gigascale systems-on-chip. *IEEE circuits and systems magazine*, 4(2):18–31, 2004.

[27] Davide Bertozzi, Antoine Jalabert, Srinivasan Murali, Rutuparna Tamhankar, Stergios Stergiou, Luca Benini, and Giovanni De Micheli. Noc synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE transactions on parallel and distributed systems*, 16(2):113–129, 2005.

[28] Praveen Bhojwani and Rabi Mahapatra. Interfacing cores with on-chip packet-switched networks. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 382–387, 2003.

[29] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1, 2006.

[30] Tobias Bjerregaard, Shankar Mahadevan, Rasmus Grøndahl Olsen, and Jens Sparsø. An OCP compliant network adapter for GALS-based SoC design using the MANGO network-on-chip. In *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, pages 171–174, 2005.

[31] Brent Bohnenstiehl, Aaron Stillmaker, Jon Pimentel, Timothy Andreas, Bin Liu, Anh Tran, Emmanuel Adeagbo, and Bevan Baas. A 5.8 pj/op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *to appear in the proceedings of the VLSI Symposium 2016*, 2016.

[32] Luciano Bononi and Nicola Concer. Simulation and analysis of network on chip architectures: ring, spidergon and 2d mesh. In *Proceedings of the conference on Design, automation and test in Europe: Designers' forum*, pages 154–159. European Design and Automation Association, 2006.

[33] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, and Thomas Gross. Warp: an integrated solution of high-speed parallel computing. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 330–339. IEEE Computer Society Press, 1988.

[34] Ron Brightwell. A comparison of three mpi implementations for red storm. *Lecture notes in computer science*, 3666:425, 2005.

[35] Ron Brightwell and Arthur Maccabe. Scalability limitations of via-based technologies in supporting mpi. In *Proceedings of the Fourth MPI Developer's and User's Conference*, 2000.

[36] Ron Brightwell and Keith D Underwood. An analysis of nic resource usage for offloading mpi. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 183. IEEE, 2004.

[37] Doug Burger, Stephen W Keckler, Kathryn S McKinley, Mike Dahlin, Lizy K John, Calvin Lin, Charles R Moore, James Burrill, Robert G McDonald, and William Yoder. Scaling to the end of silicon with edge architectures. *Computer*, 37(7): 44–55, 2004.

[38] E. Cannella, E. Beigne, O. Derin, P. Meloni, G. Tuveri, and T. Stefanov. Adaptivity support for MPSoCs based on process migration in polyhedral process networks. *VLSI Des.*, 2012:2:2–2:2, January 2012. ISSN 1065-514X. doi: 10.1155/2012/987209. URL `http://dx.doi.org/10.1155/2012/987209`.

[39] Nicholas J Carriero, David Gelernter, Timothy G Mattson, and Andrew H Sherman. The linda alternative to message-passing systems. *Parallel computing*, 20(4): 633–655, 1994.

[40] David Castells-Rufas and Jordi Carrabina. 128-core Many-Soft-Core Processor with MPI support. *JCRA Conference*, 2015.

[41] David Castells-Rufas, Sergi Risueño, Eduard Fernandez, Jordi Carrabina, and Jaume Joven. Instruction Set Extensions to Reduce Latency in Soft-Core Clusters. In *Proceedings of DCIS 2010*, 2010.

[42] Jeronimo Castrillon, Andreas Tretter, Rainer Leupers, and Gerd Ascheid. Communication-aware mapping of kpn applications onto heterogeneous mpsocs. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1262–1267. IEEE, 2012.

[43] Jeronimo Castrillon et al. Task management in MPSoCs: an ASIP approach. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 587–594, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-800-1.

[44] Mario R Casu, Massimo Ruo Roch, Sergio V Tota, and Maurizio Zamboni. A noc-based hybrid message-passing/shared-memory approach to cmp design. *Microprocessors and Microsystems*, 35(2):261–273, 2011.

[45] Jianjiang Ceng, Jerónimo Castrillón, Weihua Sheng, H Scharwachter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki, and Hiroaki Kunieda. Maps: an integrated framework for mpsoc application parallelization. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 754–759. IEEE, 2008.

[46] Niladrish Chatterjee, Seth H Pugsley, Josef Spjut, and Rajeev Balasubramonian. Optimizing a multi-core processor for message-passing workloads. In *Proceedings of the Workshop on Unique Chips and Systems (UCAS-5)*, 2009.

[47] Xiaowen Chen, Zhonghai Lu, Axel Jantsch, and Shuming Chen. Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 39–44. European Design and Automation Association, 2010.

[48] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Bocchino, Sarita Adve, and Vikram Adve. Denovo: Rethinking hardware for disciplined parallelism. In *Proceedings of the Second USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010.

[49] George Chrysos. Intel® xeon phi^{TM} coprocessor-the architecture. *Intel Whitepaper*, 2014.

[50] Francesco Conti, Davide Rossi, Antonio Pullini, Igor Loi, and Luca Benini. Energy-efficient vision on the PULP platform for ultra-low power parallel computing. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pages 1–6, 2014.

[51] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. Logp: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*, volume 28, pages 1–12. ACM, 1993.

[52] Thiago Raupp da Rosa, Thomas Mesquida, Romain Lemaire, and Fabien Clermidy. Mcapi-compliant hardware buffer manager mechanism to support communication in multi-core architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1006–1011. IEEE, 2016.

[53] William J Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689. IEEE, 2001.

[54] Alan L Davis and Robert M Keller. Data flow program graphs. *IEEE Computer*, 1982.

[55] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, et al. A clustered many-core processor architecture for embedded and accelerated applications. In *HPEC*, pages 1–6, 2013.

[56] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[57] Gary S Delp, David J Farber, Ronald G Minnich, Jonathan M Smith, and M-C Tam. Memory as a network abstraction. *IEEE Network*, 5(4):34–41, 1991.

[58] Robert H Dennard, VL Rideout, E Bassous, and AR LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[59] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems*, 23(8):1369–1386, 2012.

[60] John Dielissen, Andrei Radulescu, Kees Goossens, and Edwin Rijpkema. Concepts and implementation of the philips network-on-chip. In *IP-Based SoC Design*, pages 1–6, 2003.

[61] Digilent Inc. Nexys4 DDR FPGA board. `https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start`, 2018.

[62] Jack J Dongarra, Rolf Hempel, Anthony JG Hey, and David W Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical report, Oak Ridge National Lab., TN (United States), 1993.

[63] Bob Doud. Accelerating the Data Plane With the TILE-Mx Many-core Processor, 2015. URL `http://www.tilera.com/files/drim__EZchip_LinleyDataCenterConference_Feb2015_7671.pdf`.

[64] Andrew Duller, Gajinder Panesar, and Daniel Towner. Parallel processing-the picochip way. *Communicating Processing Architectures*, 2003:125–138, 2003.

[65] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The virtual interface architecture. *IEEE micro*, 18(2):66–76, 1998.

[66] T. Ebi, M. Faruque, and J. Henkel. TAPE: thermal-aware agent-based power economy for multi/many-core architectures. In *IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009*, pages 302–309, November 2009.

[67] T. Ebi, H. Rauchfuss, A. Herkersdorf, and J. Henkel. Agent-based thermal management using real-time I/O communication relocation for 3D many-cores. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*, number 6951 in Lecture Notes in Computer Science, pages 112–121. Springer Berlin Heidelberg, January 2011. ISBN 978-3-642-24153-6, 978-3-642-24154-3.

[68] Thomas Ebi, David Kramer, Wolfgang Karl, and Jörg Henkel. Economic learning for thermal-aware power budgeting in many-core architectures. In *Proceedings*

of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '11, page 189–196, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0715-4. doi: 10.1145/2039370.2039401. URL `http://doi.acm.org/10.1145/2039370.2039401`.

[69] Ashraf Elantably and Frédéric Rousseau. Task migration in multi-tiled mpsoc: Challenges, state-of-the-art and preliminary solutions. *Technical report*, 2012.

[70] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376, 2011.

[71] Eduard Fernandez-Alonso, David Castells-Rufas, Sergi Risueño, Jordi Carrabina, and Jaume Joven. A NoC-based multi-{soft} core with 16 cores. In *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*, pages 259–262, 2010.

[72] Eduard Fernandez-Alonso, Jordi Carrabina, David Castells-Rufas, and Jaume Joven. Embedding MPI in Many-Soft-Core Processors. *HIP3ES Workshop*, 2013.

[73] Steven Frank, Henry Burkhardt, and James Rothnie. The ksr 1: bridging the gap between shared memory and mpps. In *Compcon Spring'93, Digest of Papers.*, pages 285–294. IEEE, 1993.

[74] Fangfa Fu, Siyue Sun, Xin'an Hu, Junjie Song, Jinxiang Wang, and Minyan Yu. Mmpi: A flexible and efficient multiprocessor message passing interface for noc-based mpsoc. In *SOC Conference (SOCC), 2010 IEEE International*, pages 359–362, 2010.

[75] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59(7):072001, 2016.

[76] Karl Fuerlinger and Michael Gerndt. Analyzing overheads and scalability characteristics of openmp applications. In *International Conference on High Performance Computing for Computational Science*, pages 39–51. Springer, 2006.

[77] Steve B Furber, David R Lester, Luis A Plana, Jim D Garside, Eustace Painkras, Sally Temple, and Andrew D Brown. Overview of the spinnaker system architecture. *Computers, IEEE Transactions on*, 62(12):2454–2467, 2013.

[78] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004.

[79] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[80] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. Mpi-2: Extending the message-passing interface. In *Euro-Par'96 Parallel Processing*, pages 128–135. Springer, 1996.

[81] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. Enabling highly-scalable remote memory access programming with mpi-3 one sided. *Scientific Programming*, 22(2):75–91, 2014.

[82] Prometeus GmbH. TOP500 Supercomputer Site, 2017. URL `http://www.top500.org`.

[83] B. Goglin and N. Furmento. Finding a tradeoff between host interrupt load and MPI latency over Ethernet. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –9, 31 2009-sept. 4 2009. doi: 10.1109/CLUSTR.2009.5289165.

[84] Diana Göhringer, Michael Hübner, Laure Hugot-Derville, and Jürgen Becker. Message passing interface support for the runtime adaptive multi-processor system-on-chip rampsoc. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 357–364, 2010.

[85] Robert P Goldberg. Architectural principles for virtual computer systems. Technical report, PhD Thesis, Harvard University, 1973.

[86] Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421, 2005.

[87] Richard L Graham, Steve Poole, Pavel Shamis, Gil Bloch, Noam Bloch, Hillel Chapman, Michael Kagan, Ariel Shahar, Ishai Rabinovitz, and Gilad Shainer. Connectx-2 infiniband management queues: First investigation of the new support for network offloaded collective operations. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 53–62, 2010.

[88] Paul Gratz, Changkyu Kim, Robert McDonald, Stephen W Keckler, and Doug Burger. Implementation and evaluation of on-chip network architectures. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 477–484. IEEE, 2006.

[89] Paul Gratz, Changkyu Kim, Karthikeyan Sankaralingam, Heather Hanson, Premkishore Shivakumar, Stephen W Keckler, and Doug Burger. On-chip interconnection networks of the trips chip. *IEEE Micro*, 27(5):41–50, 2007.

[90] Dominik Grewe and Michael O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Compiler Construction*, pages 286–305. Springer, 2011.

[91] Boris Grot, Joel Hestness, Stephen W Keckler, and Onur Mutlu. Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 401–412. ACM, 2011.

[92] Junli Gu, Rakesh Kumar, Steven S Lumetta, and Yihe Sun. Accelerating data movement on future chip multi-processors. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, page 3, 2010.

[93] Erik Hagersten, Anders Landin, and Seif Haridi. Ddm-a cache-only memory architecture. *Computer*, 25(9):44–54, 1992.

[94] Sang-Il Han, Amer Baghdadi, Marius Bonaciu, Soo-Ik Chae, and Ahmed A Jerraya. An efficient scalable and flexible data transfer architecture for multiprocessor soc with massive distributed memory. In *Proceedings of the 41st annual Design Automation Conference*, pages 250–255, 2004.

[95] Frank Hannig, Vahid Lari, Srinivas Boppu, Alexandru Tanase, and Oliver Reiche. Invasive tightly-coupled processor arrays: A domain-specific architecture/compiler

co-design approach. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):133, 2014.

[96] Jan Heisswolf, Aurang Zaib, Andreas Weichslgartner, Martin Karle, Maximilian Singh, Thomas Wild, Jürgen Teich, Andreas Herkersdorf, and Jürgen Becker. The invasive network on chip-a multi-objective many-core communication infrastructure. In *Architecture of Computing Systems (ARCS), 2014 Workshop Proceedings*, pages 1–8. VDE, 2014.

[97] Ahmed Hemani, Axel Jantsch, Shashi Kumar, Adam Postula, Johnny Oberg, Mikael Millberg, and Dan Lindqvist. Network on chip: An architecture for billion transistor era. In *Proceeding of the IEEE NorChip Conference*, volume 31, page 11, 2000.

[98] Rolf Hempel, Hans-Christian Hoppe, and Alexander Supalov. A proposal for a parmacs library interface. Technical report, Technical report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, 1992.

[99] Jörg Henkel, Andreas Herkersdorf, Lars Bauer, Thomas Wild, Michael Hübner, Ravi Kumar Pujari, Artjom Grudnitsky, Jan Heisswolf, Aurang Zaib, Benjamin Vogel, et al. Invasive manycore architectures. In *ASP-DAC*, pages 193–200, 2012.

[100] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[101] Dana S Henry and Christopher F Joerg. A tightly-coupled processor-network interface. In *ACM SIGPLAN Notices*, volume 27, pages 111–122. ACM, 1992.

[102] Andreas Herkersdorf, Andreas Lankes, Michael Meitinger, Rainer Ohlendorf, Stefan Wallentowitz, Thomas Wild, and Johannes Zeppenfeld. Hardware support for efficient resource utilization in manycore processor systems. In *Multiprocessor System-on-Chip*, pages 57–87. Springer, 2011.

[103] Andreas Herkersdorf, Hananeh Aliee, Michael Engel, Michael Glaß, Christina Gimmler-Dumont, Jörg Henkel, Veit B Kleeberger, Michael A Kochte, Johannes M Kühn, Daniel Mueller-Gritschneder, et al. Resilience articulation point (rap): Cross-layer dependability modeling for nanometer system-on-chip resilience. *Microelectronics Reliability*, 54(6):1066–1074, 2014.

[104] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella,

P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, 2010. doi: 10.1109/ISSCC.2010.5434077. 00460.

[105] INMOS Limited. *Transputer Reference Manual.* Prentice Hall, 1992.

[106] David V James. The scalable coherent interface: scaling to high-performance systems. In *Compcon Spring'94, Digest of Papers.*, pages 64–71. IEEE, 1994.

[107] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th annual IEEE/ACM international symposium on Microarchitecture (MICRO-48)*, 2015.

[108] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1): 7–38, January 1998. ISSN 1387-2532. doi: 10.1023/A:1010090405266. URL `http://dx.doi.org/10.1023/A:1010090405266`.

[109] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K Panda, William Gropp, and Rajeev Thakur. High performance mpi-2 one-sided communication over infiniband. In *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 531–538. IEEE, 2004.

[110] Heikki Kariniemi and Jari Nurmi. Micronmesh for fault-tolerant gals multiprocessors on fpga. In *System-on-Chip, 2008. SOC 2008. International Symposium on*, pages 1–8, 2008.

[111] Heikki Kariniemi and Jari Nurmi. Fault-tolerant communication over micronmesh noc with micron message-passing protocol. In *System-on-Chip, 2009. SOC 2009. International Symposium on*, pages 005–012, 2009.

[112] Heikki Kariniemi and Jari Nurmi. Noc interface for fault-tolerant message-passing communication on multiprocessor soc platform. In *NORCHIP, 2009*, pages 1–6, 2009.

[113] Heikki Kariniemi and Jari Nurmi. High-performance noc interface with interrupt batching for micronmesh mpsoc prototype platform on fpga. In *NORCHIP, 2010*, pages 1–6, 2010.

[114] Stamatis G Kavadias, Manolis GH Katevenis, Michail Zampetakis, and Dimitrios S Nikolopoulos. On-chip communication and synchronization mechanisms with cache-integrated network interfaces. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 217–226. ACM, 2010.

[115] Krishna M. Kavi, Bill P. Buckles, and U. Narayan Bhat. A formal definition of data flow graph models. *IEEE Transactions on computers*, 35:940–948, 1986.

[116] John H Kelm, Daniel R Johnson, William Tuohy, Steven S Lumetta, and Sanjay J Patel. Cohesion: An adaptive hybrid memory model for accelerators. *IEEE micro*, 31(1):42–55, 2011.

[117] Abbas Eslami Kiasari, Zhonghai Lu, and Axel Jantsch. An analytical latency model for networks-on-chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(1):113–123, 2013.

[118] Thilo Kielmann, Henri E Bal, and Sergei Gorlatch. Bandwidth-efficient collective communication for clustered wide area systems. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 492–499. IEEE, 2000.

[119] Thilo Kielmann, Henri E Bal, and Kees Verstoep. Fast measurement of logp parameters for message passing platforms. In *International Parallel and Distributed Processing Symposium*, pages 1176–1183. Springer, 2000.

[120] Changkyu Kim, Simha Sethumadhavan, Madhu S Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W Keckler. Composable lightweight processors. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 381–394. IEEE, 2007.

[121] Keiji Kimura, Yasutaka Wada, Hirofumi Nakano, Takeshi Kodaka, Jun Shirako, Kazuhisa Ishizaka, and Hironori Kasahara. Multigrain parallel processing on compiler cooperative chip multiprocessor. In *Interaction between Compilers and Computer Architectures, 2005. INTERACT-9. 9th Annual Workshop on*, pages 11–20. IEEE, 2005.

[122] Adán Kohler, Juan Manuel Castillo-Sanchez, Joachim Gross, and Martin Radetzki. Minimal mpi as programming interface for multicore system-on-chips. In *Specification and Design Languages (FDL), 2012 Forum on*, pages 127–134, 2012.

[123] Koen Langendoen et al. Integrating Polling, Interrupts, and Thread Management. In *In Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 13–22. IEEE Computer Society, 1996.

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

[125] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, W-D Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[126] Kai Li. Shared virtual memory on loosely coupled multiprocessors. Technical report, Yale Univ., New Haven, CT (USA), 1986.

[127] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2), 2008.

[128] Jiuxing Liu and Dhabaleswar K Panda. Implementing efficient and scalable flow control schemes in mpi over infiniband. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 183. IEEE, 2004.

[129] Jiuxing Liu, Weihang Jiang, Pete Wyckoff, Dhabaleswar K Panda, David Ashton, Darius Buntinas, William Gropp, and Brian Toonen. Design and implementation of mpich2 over infiniband with rdma support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 16. IEEE, 2004.

[130] Jiuxing Liu, Wei Huang, Bülent Abali, and Dhabaleswar K Panda. High performance vmm-bypass i/o in virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 29–42, 2006.

[131] lowRISC team. lowRISC debug release. `http://www.lowrisc.org/docs/debug-v0.3/`, 2018.

[132] Ewing Lusk, James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ross Overbeek, James Patterson, and Rick Stevens. *Portable programs for parallel processors*. Holt, Rinehart & Winston, 1988.

[133] D.L. Ly, M. Saldana, and P. Chow. The challenges of using an embedded MPI for hardware-based processing nodes. In *International Conference on Field-Programmable Technology, 2009. FPT 2009*, pages 120–127, 2009. doi: 10.1109/FPT.2009.5377688. 00014.

[134] Sheng Ma, Natalie Enright Jerger, and Zhiying Wang. Supporting efficient collective communication in nocs. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, 2012.

[135] Philipp Mahr, Christian Lorchner, Harold Ishebabi, and Christophe Bobda. Socmpi: A flexible message passing library for multiprocessor systems-on-chips. In *Reconfigurable Computing and FPGAs, 2008. ReConFig'08. International Conference on*, pages 187–192, 2008.

[136] Gabriel Marchesan Almeida, Gilles Sassatelli, Pascal Benoit, Nicolas Saint-Jean, Sameer Varyani, Lionel Torres, and Michel Robert. An adaptive message passing mpsoc framework. *International Journal of Reconfigurable Computing*, 2009, 2009.

[137] Milo MK Martin, Mark D Hill, and Daniel J Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, 2012.

[138] Lauri Matilainen, Erno Salminen, Timo D Hämäläinen, and Marko Hännikäinen. Multicore communications api (mcapi) implementation on an fpga multiprocessor. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 286–293, 2011.

[139] Norm Matloff. Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August*, 2:2009, 2008.

[140] T.G. Mattson, R.F. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: the Programmer's View. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, 2010. doi: 10.1109/SC.2010.53. 00192.

[141] Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Jonathan Balkind, Alexey Lavrov, Mohammad Shahrad, Samuel Payne, and David Wentzlaff. Piton: A manycore processor for multitenant clouds. *Ieee micro*, 37(2):70–80, 2017.

[142] Thomas P McMahon and Anthony Skjellum. empi/empich: embedding mpi. In *MPI Developer's Conference, 1996. Proceedings., Second*, pages 180–184, 1996.

[143] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-

grained reconfigurable matrix. In *International Conference on Field Programmable Logic and Applications*, pages 61–70. Springer, 2003.

[144] Maged M Michael. Aba prevention using single-word instructions. *IBM Research Division, RC23089 (W0401-136), Tech. Rep*, 2004.

[145] Maged M Michael and Michael L Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *journal of parallel and distributed computing*, 51(1):1–26, 1998.

[146] Wajid Hassan Minhass, Johnny Öberg, and Ingo Sander. Design and implementation of a plesiochronous multi-core 4x4 network-on-chip fpga platform with mpi hal support. In *Proceedings of the 6th FPGAworld Conference*, pages 52–57, 2009.

[147] Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43–52, 2005.

[148] Takashi Miyamori and Kunle Olukotun. Remarc: Reconfigurable multimedia array coprocessor. *IEICE Transactions on information and systems*, 82(2):389–397, 1999.

[149] Matteo Monchiero, Gianluca Palermo, Cristina Silvano, and Oreste Villa. Exploration of distributed shared memory architectures for noc-based multiprocessors. *Journal of Systems Architecture*, 53(10):719–732, 2007.

[150] Gordon E Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp. 114 ff. *IEEE Solid-State Circuits Newsletter*, 3(20):33–35, 2006.

[151] Timothy Prickett Morgan. Intel Puts More Compute Behind Xeon E7 Big Memory, 2015. URL `http://www.nextplatform.com/2015/05/05/intel-puts-more-compute-behind-xeon-e7-big-memory/`.

[152] Silvia M Müller, Holger W Leister, Peter Dell, Nikolaus Gerteis, and Daniel Kroening. The Impact of Hardware Scheduling Mechanismus on the Performance and Cost of Processor Designs. In *ARCS*, pages 65–73, 1999.

[153] Multicore Association. Multicore Communications API Working Group. `http://www.multicore-association.org/workgroup/mcapi.php`, 2017. online, accessed June 2, 2017.

[154] André C. Nácul, Francesco Regazzoni, and Marcello Lajolo. Hardware scheduling support in SMP architectures. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pages 642–647, San Jose, CA, USA, 2007. EDA Consortium. ISBN 978-3-9810801-2-4. URL `http://dl.acm.org/citation.cfm?id=1266366.1266502`.

[155] Vijaykrishnan Narayanan and Yuan Xie. Reliability concerns in embedded system designs. *Computer*, 39(1):118–120, 2006.

[156] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2), 2010.

[157] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24:52–60, 1991.

[158] NVIDIA Corporation. NVIDIA Tesla P100. Whitepaper, 2016.

[159] Open SoC Debug team. Open SoC Debug website. `https://opensocdebug.org/`, 2018.

[160] OpTiMSoC team. Open Tiled Manycore System-on-Chip (OpTiMSoC) website. `https://optimsoc.org/`, 2018.

[161] John D Owens, William J Dally, Ron Ho, DN Jayasimha, Stephen W Keckler, and Li-Shiuan Peh. Research challenges for on-chip interconnection networks. *IEEE micro*, 27(5):96–108, 2007.

[162] Partha Pratim Pande, Cristian Grecu, Michael Jones, Andre Ivanov, and Resve Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE transactions on Computers*, 54(8):1025–1040, 2005.

[163] Steve Pawlowski and Ofri Wechsler. Intel Core Microarchitecture, 2006. URL `http://www.intel.com/pressroom/kits/core2duo/pdf/ICM_tech_overview.pdf`.

[164] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware- and software-based collective communication on the quadrics network. In *Network Computing and Applications, 2001. NCA 2001. IEEE International Symposium on*, pages 24–35. IEEE, 2001.

[165] Fabrizio Petrini, Wu-chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The quadrics network: High-performance clustering technology. *Ieee Micro*, 22(1):46–57, 2002.

[166] Gregory F Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[167] Paul Pierce. The nx/2 operating system. In *Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues-Volume 1*, pages 384–390. ACM, 1988.

[168] M. Pittau, A. Alimonda, Salvatore Carta, and A. Acquaviva. Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation. In *IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia, 2007. ESTIMedia 2007*, pages 59–64, 2007. doi: 10.1109/ESTMED.2007.4375803.

[169] Francesco Poletti, Antonio Poggiali, and Marchal Paul. Flexible hardware/software support for message passing on a distributed shared memory architecture. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 736–741, 2005.

[170] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheimer, Jun Nakajima, and Asit Mallick. Xen 3.0 and the art of virtualization. In *Linux symposium*, volume 2, pages 65–78. Ottawa, Ontario, Canada, 2005.

[171] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):63–71, 1996.

[172] Andrei Radulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees Goossens. An efficient on-chip ni offering guaranteed services, shared-memory abstraction, and flexible network configuration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1):4–17, 2005.

[173] Himanshu Raj and Karsten Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *Proceedings of the 16th international symposium on High performance distributed computing*, pages 179–188. ACM, 2007.

[174] H. Rauchfuss, T. Wild, and A. Herkersdorf. Enhanced reliability in tiled manycore architectures through transparent task relocation. In *ARCS Workshops (ARCS), 2012*, 2012.

[175] Holm Rauchfuss, Thomas Wild, and Andreas Herkersdorf. A network interface card architecture for i/o virtualization in embedded systems. In *Proceedings of the 2nd conference on I/O virtualization*, pages 2–2. USENIX Association, 2010.

[176] Robert Rose. Survey of system virtualization techniques. Technical report, Oregon State University, 2004.

[177] Mendel Rosenblum. Vmwares virtual platform. In *Proceedings of hot chips*, volume 1999, pages 185–196, 1999.

[178] Randolf Rotta. On efficient message passing on the intel scc. In *3rd Many-core Applications Research Community (MARC) Symposium*, volume 7598. KIT Scientific Publishing, 2011.

[179] Randolf Rotta. On Efficient Message Passing on the Intel SCC. In *Proceedings of the MARC Symposium 2011*, 2011. 00014.

[180] Randolf Rotta, Thomas Prescher, Jana Traue, and Jörg Nolte. In-memory communication mechanisms for many-cores–experiences with the intel scc. In *TACC-Intel Highly Parallel Computing Symposium (TI-HPCS)*, 2012.

[181] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222–226. IEEE, 2010.

[182] Manuel Saldana and Paul Chow. TMD-MPI: An MPI implementation for multiple processors across multiple FPGAs. In *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, pages 1–6, 2006.

[183] Manuel Saldana, Emanuel Ramalho, and Paul Chow. A message-passing hardware/software co-simulation environment to aid in reconfigurable computing design using tmd-mpi. In *Reconfigurable Computing and FPGAs, 2008. ReConFig'08. International Conference on*, pages 265–270, 2008.

[184] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. Exploiting

ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 422–433, 2003.

[185] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, MS Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, et al. Distributed microarchitectural protocols in the trips prototype processor. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 480–491. IEEE, 2006.

[186] P. K. Saraswat, P. Pop, and J. Madsen. Task migration for fault-tolerance in mixed-criticality embedded systems. *SIGBED Rev.*, 6(3), October 2009. ISSN 1551-3688. doi: 10.1145/1851340.1851348. URL http://doi.acm.org/10.1145/1851340.1851348.

[187] Fabian Scheler et al. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '09, pages 167–174, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-626-7.

[188] Dieter K Schroder and Jeff A Babcock. Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing. *Journal of applied Physics*, 94(1):1–18, 2003.

[189] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 18. ACM, 2008.

[190] Galen M Shipman, Ron Brightwell, Brian Barrett, Jeffrey M Squyres, and Gil Bloch. Investigations on infiniband: Efficient network buffer utilization at scale. In *PVM/MPI*, pages 178–186. Springer, 2007.

[191] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. Emp: zero-copy os-bypass nic-driven gigabit ethernet message passing. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 49–49, 2001.

[192] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu.

Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2): 34–46, 2016.

[193] Yong Ho Song and Timothy Mark Pinkston. A progressive approach to handling message-dependent deadlock in parallel computer systems. *Parallel and Distributed Systems, IEEE Transactions on*, 14(3):259–275, 2003.

[194] Jeff Squyres. `https://blogs.cisco.com/performance/mpi-progress`, 2012.

[195] Mikkel Bystrup Stensgaard and Jens Sparsø. Renoc: A network-on-chip architecture with reconfigurable topology. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 55–64. IEEE, 2008.

[196] Hari Subramoni, Krishna Kandalla, Sayantan Sur, and Dhabaleswar K Panda. Design and evaluation of generalized collective communication primitives with overlap using connectx-2 offload engine. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 40–49, 2010.

[197] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2001.

[198] Sayantan Sur, Lei Chai, Hyun-Wook Jin, and Dhabaleswar K Panda. Shared receive queue based scalable mpi design for infiniband clusters. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp, 2006.

[199] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K Panda. Rdma read based rendezvous protocol for mpi over infiniband: design alternatives and benefits. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 32–39. ACM, 2006.

[200] Noboru Tanabe, Akira Kitamura, Tomotaka Miyashiro, Yasuo Miyabe, Takeshi Araki, Zhengzhe Luo, Hironori Nakajo, and Hideharu Amano. Hardware support for MPI in DIMMnet-2 network interface. In *Innovative Architecture for Future Generation High Performance Processors and Systems, 2006. IWIA'06. International Workshop on*, pages 73–82, 2006.

[201] P. Tendulkar and S. Stuijk. A case study into predictable and composable MPSoC reconfiguration. In *Parallel and Distributed Processing Symposium Workshops*

PhD Forum (IPDPSW), 2013 IEEE 27th International, pages 293–300, 2013. doi: 10.1109/IPDPSW.2013.12.

[202] Texas Instruments. OMAP Mobile Processors: OMAP5432. `http://www.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=12842&contentId=103103`, 2011. Online; accessed 15 November 2016.

[203] Rajeev Thakur, William Gropp, and Brian Toonen. Optimizing the synchronization operations in message passing interface one-sided communication. *The International Journal of High Performance Computing Applications*, 19(2):119–128, 2005.

[204] The seL4 authors. seL4 Website. `https://sel4.systems`, 2017.

[205] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[206] Sergio V Tota, Mario R Casu, Massimo Ruo Roch, Luca Rostagno, and Maurizio Zamboni. Medea: a hybrid shared-memory/message-passing multiprocessor noc-based architecture. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 45–50, 2010.

[207] Jonathan Tse and Andrew Lines. Nanomesh: an asynchronous kilo-core system-on-chip. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 40–49, 2013.

[208] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 212–222. ACM, 1992.

[209] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38:48–56, 2005.

[210] Keith D Underwood, K Scott Hemmert, Arun Rodrigues, Richard Murphy, and Ron Brightwell. A hardware acceleration unit for MPI queue processing. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 96b–96b, 2005.

[211] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

[212] CH Van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1260–1265, 2009.

[213] Marcel D van de Burgwal, Gerard JM Smit, Gerard K Rauwerda, and Paul M Heysters. Hydra: An energy-efficient and reconfigurable network interface. In *ERSA*, pages 171–177, 2006.

[214] Anish Varghese, Bob Edwards, Gaurav Mitra, and Alistair P Rendell. Programming the adapteva epiphany 64-core network-on-chip coprocessor. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 984–992. IEEE, 2014.

[215] Manjunath Gorentla Venkata, Richard L Graham, Joshua S Ladd, Pavel Shamis, Ishai Rabinovitz, Vasily Filipov, and Gilad Shainer. Connectx-2 core-direct enabled asynchronous broadcast collective communications. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 781–787, 2011.

[216] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, et al. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[217] David Walker. Standards for message passing in a distributed memory environment. Technical report, Technical Report TM-12147, Oak Ridge National Laboratory, 1992.

[218] Stefan Wallentowitz, Andreas Lankes, Aurang Zaib, Thomas Wild, and Andreas Herkersdorf. A framework for open tiled manycore system-on-chip. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 535–538. IEEE, 2012.

[219] Stefan Wallentowitz, Thomas Wild, and Andreas Herkersdorf. Hw-osqm: Reducing the impact of event signaling by hardware-based operating system queue manipulation. In *ARCS*, pages 280–291. Springer, 2013.

[220] Stefan Wallentowitz, Michael Tempelmeier, Thomas Wild, and Andreas Herkersdorf. Network-on-chip protection switching techniques for dependable task migration on an open source MPSoC platform. In *Proceedings of the eda Workshop*. VDE-Verlag, 2014. ISBN 978-3-8007-3620-1.

[221] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE micro*, 27(5):15–31, 2007.

[222] Colin Whitby-Strevens. The Transputer. In *ACM SIGARCH Computer Architecture News*, volume 13, pages 292–300, 1985.

[223] Paul Willmann, Jeffrey Shafer, Domenic Carr, Scott Rixner, Alan L Cox, and Willy Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 306–317, 2007.

[224] Xilinx Inc. Xilinx Virtex UltraScale FPGA VCU108 Evaluation Kit. `https://www.xilinx.com/products/boards-and-kits/ek-u1-vcu108-g.html`, 2018.

[225] Aurang Zaib, Jan Heißwolf, Andreas Weichslgartner, Thomas Wild, Jürgen Teich, Jürgen Becker, and Andreas Herkersdorf. Network interface with task spawning support for noc-based dsm architectures. In *International Conference on Architecture of Computing Systems*, pages 186–198. Springer, 2015.

[226] Marko Zec, Miljenko Mikuc, and Mario Zagar. Estimating the impact of interrupt coalescing delays on steady state tcp throughput. In *International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2002.

[227] Lixin Zhang, Zhen Fang, Mike Parker, Binu K Mathew, Lambert Schaelicke, John B Carter, Wilson C Hsieh, and Sally A McKee. The impulse memory controller. *IEEE Transactions on Computers*, 50(11):1117–1132, 2001.

[228] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *Advances in neural information processing systems*, pages 487–495, 2014.

[229] Sotirios G Ziavras, Alexandros V Gerbessiotis, and Rohan Bafna. Coprocessor design to support MPI primitives in configurable multiprocessors. *Integration, the VLSI Journal*, 40(3):235–252, 2007.

[230] Hubert Zimmermann. OSI reference model–The ISO model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4): 425–432, 1980.